Streaming First Story Detection with application to Twitter

Saša Petrović
School of Informatics
University of Edinburgh
sasa.petrovic@ed.ac.uk

Miles Osborne
School of Informatics
University of Edinburgh
miles@inf.ed.ac.uk

Victor Lavrenko
School of Informatics
University of Edinburgh
vlavrenk@inf.ed.ac.uk

Abstract

With the recent rise in popularity and size of social media, there is a growing need for systems that can extract useful information from this amount of data. We address the problem of detecting new events from a stream of Twitter posts. To make event detection feasible on web-scale corpora, we present an algorithm based on locality-sensitive hashing which is able overcome the limitations of traditional approaches, while maintaining competitive results. In particular, a comparison with a state-of-the-art system on the first story detection task shows that we achieve over an order of magnitude speedup in processing time, while retaining comparable performance. Event detection experiments on a collection of 160 million Twitter posts show that celebrity deaths are the fastest spreading news on Twitter.

1 Introduction

In the recent years, the microblogging service Twitter has become a very popular tool for expressing opinions, broadcasting news, and simply communicating with friends. People often comment on events in real time, with several hundred micro-blogs (*tweets*) posted each second for significant events. Twitter is not only interesting because of this real-time response, but also because it is sometimes ahead of newswire. For example, during the protests following Iranian presidential elections in 2009, Iranian people first posted news on Twitter, where they were later picked up by major broadcasting corporations. Another example was the swine flu outbreak when the US Centre for disease control (CDC) used Twitter to post latest updates on the pandemic. In addition to this, subjective opinion expressed in posts is also an important feature that sets Twitter apart from traditional newswire.

New event detection, also known as first story detection (FSD)[1] is defined within the topic detection and tracking as one of the subtasks (Allan, 2002). Given a sequence of stories, the goal of FSD is to identify the first story to discuss a particular *event*. In this context, an event is taken to be something that happens at some specific time and place, e.g., an earthquake striking the town of L'Aquila in Italy on April 6th 2009. Detecting new events from tweets carries additional problems and benefits compared to traditional new event detection from newswire. Problems include a much higher volume of data to deal with and also a higher level of noise. A major benefit of doing new event detection from tweets is the added social component – we can understand the impact an event had and how people reacted to it.

The speed and volume at which data is coming from Twitter warrants the use of *streaming* algorithms to make first story detection feasible. In the streaming model of computation (Muthukrishnan, 2005), items (tweets in our case) arrive continuously in a chronological order, and we have to process each new one in bounded space and time. Recent examples of problems set in the streaming model include stream-based machine translation (Levenberg and Osborne, 2009), approximating kernel matrices of data streams (Shi et al., 2009), and topic modelling on streaming document collections (Yao et al., 2009). The traditional approach to FSD, where each new story is compared to all, or a constantly growing subset, of previously seen stories, does not scale to the Twitter streaming setting. We present a FSD system that works in the streaming model and takes constant time to process each new document, while also using constant space. Constant processing time is achieved by employing *locality sensitive hashing (LSH)* (Indyk and Motwani, 1998), a randomized technique that dramatically reduces the time needed

[1] We will be using the terms first story detection and new event detection interchangeably.

to find a nearest neighbor in vector space, and the space saving is achieved by keeping the amount of stories in memory constant.

We find that simply applying pure LSH in a FSD task yields poor performance and a high variance in results, and so introduce a modification which virtually eliminates variance and significantly improves performance. We show that our FSD system gives comparable results as a state-of-the-art system on the standard TDT5 dataset, while achieving an order of magnitude speedup. Using our system for event detection on 160 million Twitter posts shows that i) the number of users that write about an event is more indicative than the volume of tweets written about it, ii) spam tweets can be detected with reasonable precision, and iii) news about deaths of famous people spreads the fastest on Twitter.

2 First Story Detection

2.1 Traditional Approach

The traditional approach to first story detection is to represent documents as vectors in term space, where coordinates represent the (possibly IDF-weighted) frequency of a particular term in a document. Each new document is then compared to the previous ones, and if its similarity to the closest document (or centroid) is below a certain threshold, the new document is declared to be a first story. For example, this approach is used in the UMass (Allan et al., 2000) and the CMU system (Yang et al., 1998). Algorithm 1 shows the exact pseudocode used by the UMass system. Note that $dis_{min}(d)$ is the novelty score assigned to document d. Often, in order to decrease the running time, documents are represented using only n features with the highest weights.

Algorithm 1: Traditional FSD system based on nearest-neighbor search.

```
foreach document d in corpus do
    foreach term t in d do
        foreach document d' that contains t do
            update distance(d, d')
        end
    end
    dis_min(d) = min_d' {distance(d, d')}
    add d to inverted index
end
```

2.2 Locality Sensitive Hashing

The problem of finding the nearest neighbor to a given query has been intensively studied, but as the dimensionality of the data increases none of the current solutions provide much improvement over a simple linear search (Datar et al., 2004). More recently, research has focused on solving a relaxed version of the nearest neighbor problem, the *approximate nearest neighbor*, where the goal is to report any point that lies within $(1+\epsilon)r$ distance of the query point, where r is the distance to the nearest neighbor. One of the first approaches to solving the approximate-NN problem in sublinear time was described in Indyk and Motwani (1998), where the authors introduced a new method called *locality sensitive hashing (LSH)*. This method relied on hashing each query point into buckets in such a way that the probability of collision was much higher for points that are near by. When a new point arrived, it would be hashed into a bucket and the points that were in the same bucket were inspected and the nearest one returned.

Because we are dealing with textual documents, a particularly interesting measure of distance is the cosine between two documents. Allan et al. (2000) report that this distance outperforms the KL divergence, weighted sum, and language models as distance functions on the first story detection task. This is why in our work we use the hashing scheme proposed by Charikar (2002) in which the probability of two points colliding is proportional to the cosine of the angle between them. This scheme was used, e.g., for creating similarity lists of nouns collected from a web corpus in Ravichandran et al. (2005). It works by intersecting the space with random hyperplanes, and the buckets are defined by the subspaces formed this way. More precisely, the probability of two points x and y colliding under such a hashing scheme is

$$P_{coll} = 1 - \frac{\theta(x,y)}{\pi}, \tag{1}$$

where $\theta(x,y)$ is the angle between x and y. By using more than one hyperplane, we can decrease the probability of collision with a non-similar point. The number of hyperplanes k can be considered as a number of bits per key in this hashing scheme. In particular, if $x \cdot u_i < 0, i \in [1 \dots k]$ for document x and hyperplane vector u_i, we set the i-th bit to 0, and 1 otherwise. The higher k is, the fewer collisions we will have in our buckets but we will spend more time computing the hash values.[2] However, increasing k also decreases the probability of collision with the nearest neighbor, so we need multiple hash tables (each with k independently chosen random hyperplanes) to increase the chance that the nearest neighbor will collide with our point in at least one of

[2] Probability of collision under k random hyperplanes will be P_{coll}^k.

them. Given the desired number of bits k, and the desired probability of missing a nearest neighbor δ, one can compute the number of hash tables L as

$$L = \log_{1-P_{coll}^{k}} \delta. \quad (2)$$

2.3 Variance Reduction Strategy

Unfortunately, simply applying LSH for nearest neighbor search in a FSD task yields poor results with a lot of variance (the exact numbers are given in Section 6). This is because LSH only returns the true near neighbor if it is reasonably close to the query point. If, however, the query point lies far away from all other points (i.e., its nearest neighbor is far away), LSH fails to find the true near neighbor. To overcome this problem, we introduce a strategy by which, if the LSH scheme declares a document new (i.e., sufficiently different from all others), we start a search through the inverted index, but only compare the query with a 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. The pseudocode shown in algorithm 2 summarizes the approach based on LSH, with the lines 11 and 12 being the variance reduction strategy.

Algorithm 2: Our LSH-based approach.

```
   input: threshold t
1  foreach document d in corpus do
2      add d to LSH
3      S ← set of points that collide with d in LSH
4      dis_min(d) ← 1
5      foreach document d' in S do
6          c = distance(d, d')
7          if c < dis_min(d) then
8              dis_min(d) ← c
9          end
10     end
11     if dis_min(d) >= t then
12         compare d to a fixed number of most
           recent documents as in Algorithm 1 and
           update dis_min if necessary
13     end
14     assign score dis_min(d) to d
15     add d to inverted index
16 end
```

3 Streaming First Story Detection

Although using LSH in the way we just described greatly reduces the running time, it is still too expensive when we want to deal with *text streams*. Text streams naturally arise on the Web, where millions of new documents are published each hour. Social media sites like Facebook, MySpace, Twitter, and various blogging sites are a particularly interesting source of textual data because each new document is timestamped and usually carries additional metadata like topic tags or links to author's friends. Because this stream of documents is unbounded and coming down at a very fast rate, there is usually a limit on the amount of space/time we can spend per document. In the context of first story detection, this means we are not allowed to store all of the previous data in main memory nor compare the new document to all the documents returned by LSH.

Following the previous reasoning, we present the following desiderata for a streaming first story detection system: we first assume that each day we are presented with a large volume of documents in chronological order. A streaming FSD system should, for each document, say whether it discusses a previously unseen event and give confidence in its decision. The decision should be made in bounded time (preferably constant time per document), and using bounded space (also constant per document). Only one pass over the data is allowed and the decision has to be made immediately after a new document arrives. A system that has all of these properties can be employed for finding first stories in real time from a stream of stories coming down from the Web.

3.1 A constant space and time approach

In this section, we describe our streaming FSD system in more depth. As was already mentioned in Section 2.2, we use locality sensitive hashing to limit our search to a small number of documents. However, because there is only a finite number of buckets, in a true streaming setting the number of documents in any bucket will grow without a bound. This means that i) we would use an unbounded amount of space, and ii) the number of comparisons we need to make would also grow without a bound. To alleviate the first problem, we limit the number of documents inside a single bucket to a constant. If the bucket is full, the oldest document in the bucket is removed. Note that the document is removed only from that single bucket in one of the L hash tables – it may still be present in other hash tables. Note that this way of limiting the number of documents kept is in a way topic-specific. Luo et al. (2007) use a global constraint on the documents they keep and show that around 30 days of data needs to be kept in order to achieve reasonable performance. While using this approach also ensures that the number of comparisons made is constant, this constant can be

rather large. Theoretically, a new document can collide with all of the documents that are left, and this can be quite a large number (we have to keep a sufficient portion of the data in memory to make sure we have a representative sample of the stream to compare with). That is why, in addition to limiting the number of documents in a bucket, we also limit ourselves to making a constant number of comparisons. We do this by comparing each new document with at most 3L documents it collided with. Unlike Datar et al. (2004), where any 3L documents were used, we compare to the 3L documents that collide most frequently with the new document. That is, if S is the set of all documents that collided with a new document in all L hash tables, 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.

4 Detecting Events in Twitter Posts

While doing first story detection on a newspaper stream makes sense because all of the incoming documents are actual stories, this is not the case with Twitter posts (tweets). The majority of tweets are not real stories, but rather updates on one's personal life, conversations, or spam. Thus, simply running a first story detection system on this data would yield an incredible amount of new stories each day, most of which would be of no interest to anyone but a few people. However, when something significant happens (e.g., a celebrity dies), a lot of users write about this either to share their opinion or just to inform others of the event. Our goal here is to automatically detect these significant events, preferably with a minimal number of non-important events.

Threading. We first run our streaming FSD system and assign a novelty score to each tweet. In addition, since the score is based on a cosine distance to the nearest tweet, for each tweet we also output which other tweet it is most similar to. This way, we can analyze *threads* of tweets, i.e., a subset of tweets which all discuss the same topic (Nallapati et al., 2004). To explain how we form threads of tweets, we first introduce the *links* relation. We say that tweet a *links* to tweet b if b is the nearest neighbor of a and $1 - \cos(a, b) < t$, where t is a user-specified threshold. Then, 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. By changing t we can control the granularity of threads. 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 $t \in [0.5, 0.6]$ results are very much the same, whereas setting t outside this interval starts to impact the results in the way we just explained.

Once we have threads of tweets, we are interested in which threads grow fastest, as this will be an indication that news of a new event is spreading. Therefore, for each time interval we only output the fastest growing threads. This growth rate also gives us a way to measure a thread's impact.

5 Related Work

In the recent years, analysis of social media has attracted a lot of attention from the research community. However, most of the work that uses social media focuses on blogs (Glance et al., 2004; Bansal and Koudas, 2007; Gruhl et al., 2005). On the other hand, research that uses Twitter has so far only focused on describing the properties of Twitter itself (Java et al., 2007; Krishnamurthy et al., 2008).

The problem of online new event detection in a large-scale streaming setting was previously addressed in Luo et al. (2007). Their system used the traditional approach to FSD and then employed various heuristics to make computation feasible. These included keeping only the first stories in memory, limiting the number of terms per document, limiting the number of total terms kept, and employing parallel processing. Our randomized framework gives us a principled way to work out the errors introduced and is more general than the previously mentioned approach because we could still use all the heuristics used by Luo et al. (2007) in our system. Finally, while Luo et al. (2007) achieved considerable speedup over an existing system on a TDT corpus, they never showed the utility of their system on a truly large-scale task.

The only work we are aware of that analyzes social media in a streaming setting is Saha and Getoor (2009). There, the focus was on solving the *maximum coverage* problem for a stream of blog posts. The maximum coverage problem in their setting, dubbed *blog watch*, was selecting k blogs that maximize the cover of interests specified by a user. This work differs from Saha and Getoor (2009) in many ways. Most notably, we deal with the problem of detecting new events, and determining who was the first to report them. Also, there is a difference in the type and volume of data – while Saha and Getoor (2009) use 20 days of blog data totalling two million posts, we use Twitter data from a timespan of six

months, totalling over 160 million posts.

6 Experiments

6.1 TDT5 Experimental Setup

Baseline. Before applying our FSD system on Twitter data, we first compared it to a state-of-the-art FSD system on the standard TDT5 dataset. This way, we can test if our system is on par with the best existing systems, and also accurately measure the speedup that we get over a traditional approach. In particular, we compare our system with the UMass FSD system (Allan et al., 2000). The UMass system has participated in the TDT2 and TDT3 competitions and is known to perform at least as well as other existing systems who also took part in the competition (Fiscus, 2001). Note that the UMass system uses an inverted index (as shown in Algorithm 1) which optimizes the system for speed and makes sure a minimal number of comparisons is made. We compare the systems on the English part of the TDT5 dataset, consisting of 221,306 documents from a time period spanning April 2003 to September 2003. To make sure that any difference in results is due to approximations we make, we use the same settings as the UMass system: 1-NN clustering, cosine as a similarity measure, and TFIDF weighted document representation, where the IDF weights are incrementally updated. These particular settings were found by Allan et al. (2000) to perform the best for the FSD task. We limit both systems to keeping only top 300 features in each document. Using more than 300 features barely improves performance while taking significantly more time for the UMass system.[3]

LSH parameters. In addition, our system has two LSH parameters that need to be set. The number of hyperplanes k gives a tradeoff between time spent computing the hash functions and the time spent computing the distances. A lower k means more documents per bucket and thus more distance computations, whereas a higher k means less documents per bucket, but more hash tables and thus more time spent computing hash functions. 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 because it achieved a reasonable balance between time spent computing the distances and the time spent computing the hash functions.

[3]In other words, using more features only increases the advantage of our system over the UMass system.

Evaluation metric. The official TDT evaluation requires each system to assign a confidence score for its decision, and this assignment can be made either immediately after the story arrives, or after a fixed number of new stories have been observed. Because we assume that we are working in a true streaming setting, systems are required to assign a confidence score as soon as the new story arrives. The actual performance of a system is measured in terms of *detection error tradeoff (DET)* curves and the *minimal normalized cost*. Evaluation is carried out by first sorting all stories according to their scores and then performing a threshold sweep. For each value of the threshold, stories with a score above the threshold are considered new, and all others are considered old. Therefore, for each threshold value, one can compute the probability of a *false alarm*, i.e., probability of declaring a story new when it is actually not, and the *miss probability*, i.e., probability of declaring a new story old (missing a new story). Having computed all the miss and false alarm probabilities, we can plot them on a graph showing the tradeoff between these two quantities – such graphs are called detection error tradeoff curves. The normalized cost C_{det} is computed as

$$C_{det} = C_{miss} * P_{miss} * P_{target} + C_{FA} * P_{FA} * P_{non-target},$$

where C_{miss} and C_{FA} are costs of miss and false alarm, P_{miss} and P_{FA} are probabilities of a miss and false alarm, and P_{target} and $P_{non-target}$ are the prior target and non-target probabilities. Minimal normalized cost C_{min} is the minimal value of C_{det} over all threshold values (a lower value of C_{min} indicates better performance).

6.2 TDT5 Results

All the results on the TDT5 dataset are shown in Table 1. In this section, we go into detail in explaining them. As was mentioned in Section 2.2, simply using LSH to find a nearest neighbor resulted in poor performance and a high variance of results. In particular, the mean normalized cost of ten runs of our system without the variance reduction strategy was 0.88, with a standard deviation of 0.046. When using the strategy explained in Section 2.2, the mean result dropped to 0.70, with a standard deviation of 0.004. Therefore, the results were significantly improved, while also reducing standard deviation by an order of magnitude. This shows that there is a clear advantage in using our variance reduction strategy,

Table 1: Summary of TDT5 results. Numbers next to LSH'_{ts} indicate the maximal number of documents in a bucket, measured in terms of percentage of the expected number of collisions.

	Baseline	Unbounded		Bounded			
		Pure	Variance Red.	Time	Space and Time		
System	UMass	LSH	LSH′	LSH'_t	LSH'_{ts} 0.5	LSH'_{ts} 0.3	LSH'_{ts} 0.1
C_{min}	0.69	0.88	0.70	0.71	0.76	0.75	0.73

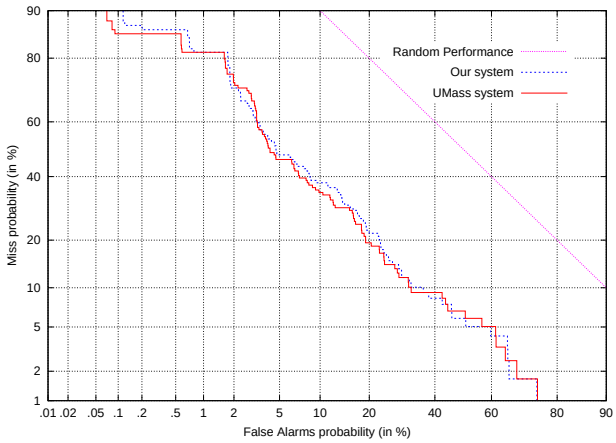

Figure 1: Comparison of our system with the UMass FSD system.

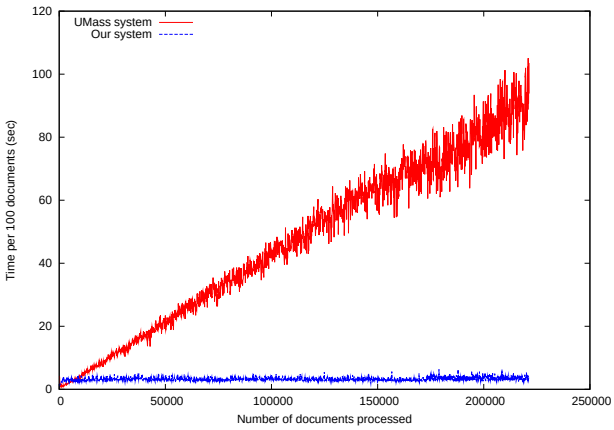

Figure 2: Comparison of processing time per 100 documents for our and the UMass system.

and all the following results we report were obtained from a system that makes use of it.

Figure 1 shows DET curves for the UMass and for our system. For this evaluation, our system was not limited in space, i.e., buckets sizes were unlimited, but the processing time per item was made constant. It is clear that UMass outperforms our system, but the difference is negligible. In particular, the minimal normalized cost C_{min} was 0.69 for the UMass system, and 0.71 for our system. On the other hand, the UMass system took 28 hours to complete the run, compared to two hours for our system. Figure 2 shows the time required to process 100 documents as a function of number of documents seen so far. We can see that our system maintains constant time, whereas the UMass system processing time grows without a bound (roughly linear with the number of previously seen documents).

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. Not surprisingly, limiting the bucket size reduced performance compared to the space-unlimited version, but even when the size is reduced to 10% of the expected number of collisions, performance remains reasonably close to the UMass system. Figure 3 shows the memory usage of our system on a month of Twitter data (more detail about the data can be found in Section 6.3). We can see that most of the memory is allocated right away, after which the memory consumption levels out. If we ran the system indefinitely, we would see the memory usage grow slower and slower until it reached a certain level at which it would remain constant.

6.3 Twitter Experimental Setup

Corpus. We used our streaming FSD system to detect new events from a collection of Twitter data gathered over a period of six months (April 1st 2009 to October 14th 2009). Data was collected through Twitter's streaming API.[4] Our corpus consists of 163.5 million timestamped tweets, totalling over 2 billion tokens. All the tweets in our corpus contain

[4] `http://stream.twitter.com/`

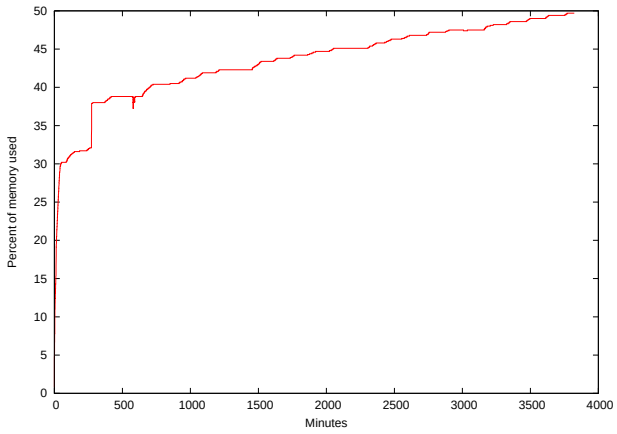

Figure 3: Memory usage on a month of Twitter data. X-axis shows how long the system has been running for.

only ASCII characters and we additionally stripped the tweets of words beginning with the @ or # symbol. This is because on Twitter words beginning with @ indicate a reply to someone, and words beginning with # are topic tags. Although these features would probably be helpful for our task, we decided not to use them as they are specific to Twitter and our approach should be independent of the stream type.

Gold standard. In order to measure how well our system performs on the Twitter data, we employed two human experts to manually label all the tweets returned by our system as either *Event, Neutral*, or *Spam*. Note that each tweet that is returned by our system is actually the first tweet in a thread, and thus serves as the representative of what the thread is about. Spam tweets include various advertisements, automatic weather updates, automatic radio station updates, etc. For a tweet to be labeled as an event, it had to be clear from the tweet alone what exactly happened without having any prior knowledge about the event, and the event referenced in the tweet had to be sufficiently important. Important events include celebrity deaths, natural disasters, major sports, political, entertainment, and business events, shootings, plane crashes and other disasters. Neutral tweets include everything not labeled as spam or event. Because the process of manual labeling is tedious and time-consuming, we only labeled the 1000 fastest growing threads from June 2009. 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. Agreement between our two annotators, measured using Cohen's kappa coefficient, was substantial (kappa = 0.65). We use 820 tweets on which both annotators agreed as the gold standard.

Evaluation. Evaluation is performed by computing *average precision* (AP) on the gold standard sorted according to different criteria, where event tweets are taken to be relevant, and neutral and spam tweets are treated as non-relevant documents. Average precision is a common evaluation metric in tasks like ad-hoc retrieval where only the set of returned documents and their relevance judgements are available, as is the case here (Croft et al., 2009). Note that we are not evaluating our FSD system here. There are two main reasons for this: i) we already have a very good idea about the first story detection performance from the experiments on TDT5 data, and ii) evaluating a FSD system on this scale would be prohibitively expensive as it would involve human experts going through 30 million tweets looking for first stories. Rather, we are evaluating different methods of ranking threads which are output from a FSD system for the purpose of detecting important events in a very noisy and unstructured stream such as Twitter.

6.4 Twitter Results

Results for the average precisions are given in Table 2. Note that we were not able to compare our system with the UMass FSD system on the Twitter data, as the UMass system would not finish in any reasonable amount of time. Different rows of Table 2 correspond to the following ways of ranking the threads:

- Baseline – random ordering of threads
- Size of thread – threads are ranked according to number of tweets
- Number of users – threads are ranked according to number of unique users posting in a thread
- Entropy + users – if the entropy of a thread is < 3.5, move to the back of the list, otherwise sort according to number of unique users

Results show that ranking according to size of thread performs better than the baseline, and ranking according to the number of users is slightly better. However, a sign test showed that neither of the two ranking strategies is significantly better than the baseline. We perform the sign test by splitting the labeled data into 50 stratified samples and ranking each sample with different strategies. We then measure the number of times each strategy performed better (in terms of AP) and compute the significance levels based on these numbers. Adding the information about the entropy of the thread showed to be

Table 2: Average precision for *Events* vs. *Rest* and for *Events* and *Neutral* vs. *Spam*.

Ranking method	events vs. rest	spam vs. rest
Baseline	16.5	84.6
Size of thread	24.1	83.5
Number of users	24.5	83.9
Entropy + users	34.0	96.3

Table 3: Average precision as a function of the entropy threshold on the *Events* vs. *Rest* task.

Entropy	2	2.5	3	3.5	4	4.5
AP	24.8	27.6	30.0	34.0	33.2	29.4

very beneficial. Entropy of a thread is computed as

$$H_{thread} = -\sum_i \frac{n_i}{N} \log \frac{n_i}{N},$$

where n_i is the number of times word i appears in a thread, and $N = \sum_i n_i$ is the total number of words in a thread. We move the threads with low 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 ($p \leq 0.01$) 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. This approach works well because most spam threads have very low entropy, i.e., contain very little information.

We conducted another experiment where events and neutral tweets are considered relevant, and spam tweets non-relevant documents. Results for this experiment are given in the third column of Table 2. Results for this experiment are much better, mostly due to the large proportion of neutral tweets in the data. The baseline in this case is very strong and neither sorting according to the size of the thread nor according to the number of users outperforms the baseline. However, adding the information about entropy significantly ($p \leq 0.01$) improves the performance over all other ranking methods.

Finally, in Table 4 we show the top ten fastest growing threads in our data (ranked by the number of users posting in the thread). Each thread is represented by the first tweet. We can see from the table that events which spread the fastest on Twitter are mostly deaths of famous people. One spam thread that appears in the list has an entropy of 2.5 and doesn't appear in the top ten list when using the entropy + users ranking.

Table 4: Top ten fastest growing threads in our data.

# users	First tweet
7814	TMZ reporting michael jackson has had a heart attack. We r checking it out. And pulliing video to use if confirmed
7579	RIP Patrick Swayze...
3277	Walter Cronkite is dead.
2526	we lost Ted Kennedy :(
1879	RT BULLETIN – STEVE MCNAIR HAS DIED.
1511	David Carradine (Bill in "Kill Bill") found hung in Bangkok hotel.
1458	Just heard Sir Bobby Robson has died. RIP.
1426	I just upgraded to 2.0 - The professional Twitter client. Please RT!
1220	LA Times reporting Manny Ramirez tested positive for performance enhancing drugs. To be suspended 50 games.
1057	A representative says guitar legend Les Paul has died at 94

7 Conclusion

We presented an approach to first story detection in a streaming setting. Our approach is based on locality sensitive hashing adapted to the first story detection task by introducing a backoff towards exact search. This adaptation greatly improved performance of the system and virtually eliminated variance in the results. We showed that, using our approach, it is possible to achieve constant space and processing time while maintaining very good results. A comparison with the UMass FSD system showed that we gain more than an order of magnitude speedup with only a minor loss in performance. We used our FSD system on a truly large-scale task of detecting new events from over 160 million Twitter posts. To the best of our knowledge, this is the first work that does event detection on this scale. We showed that our system is able to detect major events with reasonable precision, and that the amount of spam in the output can be reduced by taking entropy into account.

Acknowledgments

The authors would like to thank Donnla Osborne for her work on annotating tweets.

References

James Allan, Victor Lavrenko, Daniella Malin, and Russell Swan. 2000. Detections, bounds, and timelines: Umass and tdt-3. In *Proceedings of Topic Detection and Tracking Workshop*, pages 167–174.

James Allan. 2002. *Topic detection and tracking: event-based information organization*. Kluwer Academic Publishers.

Nilesh Bansal and Nick Koudas. 2007. Blogscope: a system for online analysis of high volume text streams. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1410–1413. VLDB Endowment.

Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, New York, NY, USA. ACM.

W.B. Croft, D. Metzler, and T. Strohman. 2009. *Search Engines: Information Retrieval in Practice*. Addison-Wesley Publishing.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, New York, NY, USA. ACM.

J. Fiscus. 2001. Overview of results (nist). In *Proceedings of the TDT 2001 Workshop*.

N. Glance, M. Hurst, and T. Tomokiyo. 2004. BlogPulse: Automated Trend Discovery for Weblogs. *WWW 2004 Workshop on the Weblogging Ecosystem: Aggregation, Analysis and Dynamics*, 2004.

Daniel Gruhl, R. Guha, Ravi Kumar, Jasmine Novak, and Andrew Tomkins. 2005. The predictive power of online chatter. In *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 78–87, New York, NY, USA. ACM.

Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, New York, NY, USA. ACM.

Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. 2007. Why we twitter: understanding microblogging usage and communities. In *WebKDD/SNA-KDD '07: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pages 56–65, New York, NY, USA. ACM.

Balachander Krishnamurthy, Phillipa Gill, and Martin Arlitt. 2008. A few chirps about twitter. In *WOSP '08: Proceedings of the first workshop on Online social networks*, pages 19–24, New York, NY, USA. ACM.

Abby Levenberg and Miles Osborne. 2009. Stream-based randomised language models for smt. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 756–764.

Gang Luo, Chunqiang Tang, and Philip S. Yu. 2007. Resource-adaptive real-time new event detection. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 497–508, New York, NY, USA. ACM.

S. Muthukrishnan. 2005. *Data streams: Algorithms and applications*. Now Publishers Inc.

Ramesh Nallapati, Ao Feng, Fuchun Peng, and James Allan. 2004. Event threading within news topics. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 446–453, New York, NY, USA. ACM.

Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. 2005. Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In *ACL '05: Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 622–629, Morristown, NJ, USA. Association for Computational Linguistics.

Barna Saha and Lise Getoor. 2009. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *2009 SIAM International Conference on Data Mining (SDM09)*, April.

Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, Alex Strehl, and Vishy Vishwanathan. 2009. Hash kernels. In *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 496–503.

Yiming Yang, Tom Pierce, and Jaime Carbonell. 1998. A study of retrospective and on-line event detection. In *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 28–36, New York, NY, USA. ACM.

Limin Yao, David Mimno, and Andrew McCallum. 2009. Efficient methods for topic model inference on streaming document collections. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 937–946, New York, NY, USA. ACM.