Online Generation of Locality Sensitive Hash Signatures

Benjamin Van Durme
HLTCOE
Johns Hopkins University
Baltimore, MD 21211 USA

Ashwin Lall
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332 USA

Abstract

Motivated by the recent interest in streaming algorithms for processing large text collections, we revisit the work of Ravichandran et al. (2005) on using the Locality Sensitive Hash (LSH) method of Charikar (2002) to enable fast, approximate comparisons of vector cosine similarity. For the common case of feature updates being additive over a data stream, we show that LSH signatures can be maintained *online*, without additional approximation error, and with lower memory requirements than when using the standard *offline* technique.

1 Introduction

There has been a surge of interest in adapting results from the streaming algorithms community to problems in processing large text collections. The term *streaming* refers to a model where data is made available sequentially, and it is assumed that resource limitations preclude storing the entirety of the data for offline (batch) processing. Statistics of interest are approximated via online, randomized algorithms. Examples of text applications include: collecting approximate counts (Talbot, 2009; Van Durme and Lall, 2009a), finding top-n elements (Goyal et al., 2009), estimating term co-occurrence (Li et al., 2008), adaptive language modeling (Levenberg and Osborne, 2009), and building top-k ranklists based on pointwise mutual information (Van Durme and Lall, 2009b).

Here we revisit the work of Ravichandran et al. (2005) on building word similarity measures from large text collections by using the Locality Sensitive Hash (LSH) method of Charikar (2002). For the common case of feature updates being additive over a data stream (such as when tracking lexical co-occurrence), we show that LSH signatures can be maintained online, without additional approximation error, and with lower memory requirements than when using the standard offline technique.

We envision this method being used in conjunction with dynamic clustering algorithms, for a variety of applications. For example, Petrovic et al. (2010) made use of LSH signatures generated over individual *tweets*, for the purpose of *first story detection*. Streaming LSH should allow for the clustering of Twitter *authors*, based on the tweets they generate, with signatures continually updated over the Twitter stream.

2 Locality Sensitive Hashing

We are concerned with computing the *cosine similarity* of feature vectors, defined for a pair of vectors $\vec{u}$ and $\vec{v}$ as the dot product normalized by their lengths:

$$\text{cosine}-\text{similarity}(\vec{u}, \vec{v}) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}.$$

This similarity is the cosine of the angle between these high-dimensional vectors and attains a value of one (i.e., $\cos(0)$) when the vectors are parallel and zero (i.e., $\cos(\pi/2)$) when orthogonal.

Building on the seminal work of Indyk and Motwani (1998) on *locality sensitive hashing* (LSH), Charikar (2002) presented an LSH that maps high-dimensional vectors to a much smaller dimensional space while still preserving (cosine) similarity between vectors in the original space. The LSH algorithm computes a succinct signature of the feature set of the words in a corpus by computing d independent dot products of each feature vector $\vec{v}$ with a random unit vector $\vec{r}$, i.e., $\sum_i v_i r_i$, and retaining the sign of the d resulting products. Each entry of $\vec{r}$ is drawn from the distribution $N(0, 1)$, the normal distribution with zero mean and unit variance. Charikar's algorithm makes use of the fact (proved by Goemans and Williamson

Proceedings of the ACL 2010 Conference Short Papers, pages 231–235,
Uppsala, Sweden, 11-16 July 2010. ©2010 Association for Computational Linguistics

(1995) for an unrelated application) that the angle between any two vectors summarized in this fashion is proportional to the expected Hamming distance of their signature vectors. Hence, we can retain length d bit-signatures in the place of high dimensional feature vectors, while preserving the ability to (quickly) approximate cosine similarity in the original space.

Ravichandran et al. (2005) made use of this algorithm to reduce the computation in searching for similar nouns by first computing signatures for each noun and then computing similarity over the signatures rather than the original feature space.

3 Streaming Algorithm

In this work, we focus on features that can be maintained additively, such as raw frequencies.[1] Our streaming algorithm for this problem makes use of the simple fact that the dot product of the feature vector with random vectors is a linear operation. This permits us to replace the $v_i \cdot r_i$ operation by v_i individual additions of r_i, once for each time the feature is encountered in the stream (where v_i is the frequency of a feature and r_i is the randomly chosen Gaussian-distributed value associated with this feature). The result of the final computation is identical to the dot products computed by the algorithm of Charikar (2002), but the processing can now be done online. A similar technique, for stable random projections, was independently discussed by Li et al. (2008).

Since each feature may appear multiple times in the stream, we need a consistent way to retrieve the random values drawn from $N(0, 1)$ associated with it. To avoid the expense of computing and storing these values explicitly, as is the norm, we propose the use of a precomputed pool of random values drawn from this distribution that we can then hash into. Hashing into a fixed pool ensures that the same feature will consistently be associated with the same value drawn from $N(0, 1)$. This introduces some weak dependence in the random vectors, but we will give some analysis showing that this should have very limited impact on the cosine similarity computation, which we further support with experimental evidence (see Table 3).

Our algorithm traverses a stream of words and maintains some state for each possible word that it encounters (cf. Algorithm 1). In particular, the state maintained for each word is a vector of floating point numbers of length d. Each element of the vector holds the (partial) dot product of the feature vector of the word with a random unit vector. Updating the state for a feature seen in the stream for a given word simply involves incrementing each position in the word's vector by the random value associated with the feature, accessed by hash functions h_1 through h_d. At any point in the stream, the vector for each word can be processed (in time $O(d)$) to create a signature computed by checking the sign of each component of its vector.

Algorithm 1 STREAMING LSH ALGORITHM

Parameters:
m : size of pool
d : number of bits (size of resultant signature)
s : a random seed
$h_1, ..., h_d$: hash functions mapping $\langle s, f_i \rangle$ to $\{0, \ldots, m-1\}$

INITIALIZATION:
1: Initialize floating point array $P[0, \ldots, m-1]$
2: Initialize H, a hashtable mapping words to floating point arrays of size d
3: **for** $i := 0 \ldots m-1$ **do**
4: $\quad P[i]$:= random sample from $N(0, 1)$, using s as seed

ONLINE:
1: **for** each word w in the stream **do**
2: $\quad$ **for** each feature f_i associated with w **do**
3: $\quad\quad$ **for** $j := 1 \ldots d$ **do**
4: $\quad\quad\quad H[w][j] := H[w][j] + P[h_j(s, f_i)]$

SIGNATURECOMPUTATION:
1: **for** each $w \in H$ **do**
2: $\quad$ **for** $i := 1 \ldots d$ **do**
3: $\quad\quad$ **if** $H[w][i] > 0$ **then**
4: $\quad\quad\quad S[w][i] := 1$
5: $\quad\quad$ **else**
6: $\quad\quad\quad S[w][i] := 0$

3.1 Analysis

The update cost of the streaming algorithm, per word in the stream, is $O(df)$, where d is the target signature size and f is the number of features associated with each word in the stream.[2] This results in an overall cost of $O(ndf)$ for the streaming algorithm, where n is the length of the stream. The memory footprint of our algorithm is $O(n_0d+m)$, where n_0 is the number of distinct words in the stream and m is the size of the pool of normally distributed values. In comparison, the original LSH algorithm computes signatures at a cost of $O(nf + n_0dF)$ updates and $O(n_0F + dF + n_0d)$ memory, where F is the (large) number of unique

[1] Note that Ravichandran et al. (2005) used pointwise mutual information features, which are not additive since they require a global statistic to compute.

[2] For the bigram features used in § 4, $f = 2$.

features. Our algorithm is superior in terms of memory (because of the pooling trick), and has the benefit of supporting similarity queries online.

3.2 Pooling Normally-distributed Values

We now discuss why it is possible to use a fixed pool of random values instead of generating unique ones for each feature. Let g be the c.d.f. of the distribution $N(0,1)$. It is easy to see that picking $x \in (0,1)$ uniformly results in $g^{-1}(x)$ being chosen with distribution $N(0,1)$. Now, if we select for our pool the values

$$g^{-1}(1/m), g^{-1}(2/m), \ldots, g^{-1}(1-1/m),$$

for some sufficiently large m, then this is identical to sampling from $N(0,1)$ with the caveat that the accuracy of the sample is limited. More precisely, the deviation from sampling from this pool is off from the actual value by at most

$$\max_{i=1,\ldots,m-2}\{g^{-1}((i+1)/m) - g^{-1}(i/m)\}.$$

By choosing m to be sufficiently large, we can bound the error of the approximate sample from a true sample (i.e., the loss in precision expressed above) to be a small fraction (e.g., 1%) of the actual value. This would result in the same relative error in the computation of the dot product (i.e., 1%), which would almost never affect the sign of the final value. Hence, pooling as above should give results almost identical to the case where all the random values were chosen independently. Finally, we make the observation that, for large m, randomly choosing m values from $N(0,1)$ results in a set of values that are distributed very similarly to the pool described above. An interesting avenue for future work is making this analysis more mathematically precise.

3.3 Extensions

Decay The algorithm can be extended to support *temporal decay* in the stream, where recent observations are given higher relative weight, by multiplying the current sums by a decay value (e.g., 0.9) on a regular interval (e.g., once an hour, once a day, once a week, etc.).

Distributed The algorithm can be easily distributed across multiple machines in order to process different parts of a stream, or multiple different streams, in parallel, such as in the context of the MapReduce framework (Dean and Ghemawat, 2004). The underlying operation is a linear operator that is easily composed (i.e., via addition), and the randomness between machines can be tied based on a shared seed s. At any point in processing the stream(s), current results can be aggregated by summing the d-dimensional vectors for each word, from each machine.

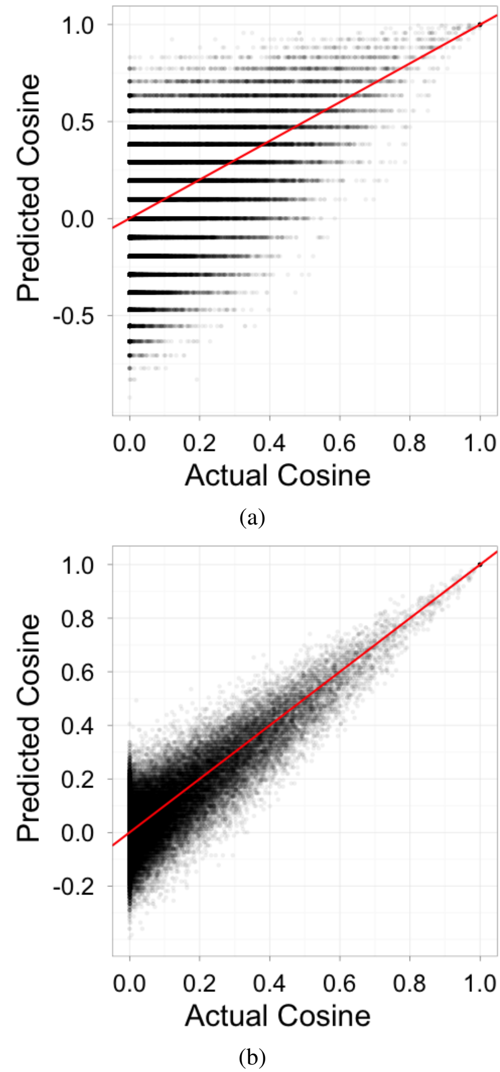

Figure 1: Predicted versus actual cosine values for 50,000 pairs, using LSH signatures generated online, with $d = 32$ in Fig. 1(a) and $d = 256$ in Fig. 1(b).

4 Experiments

Similar to the experiments of Ravichandran et al. (2005), we evaluated the fidelity of signature generation in the context of calculating distributional similarity between words across a large text collection: in our case, articles taken from the NYTimes portion of the Gigaword corpus (Graff, 2003). The collection was processed as a stream, sentence by sentence, using bigram fea-

d	16	32	64	128	256
SLSH	0.2885	0.2112	0.1486	0.1081	0.0769
LSH	0.2892	0.2095	0.1506	0.1083	0.0755

Table 1: Mean absolute error when using signatures generated online (StreamingLSH), compared to offline (LSH).

tures. This gave a stream of 773,185,086 tokens, with 1,138,467 unique types. Given the number of types, this led to a (sparse) feature space with dimension on the order of 2.5 million.

After compiling signatures, fifty-thousand $\langle x, y \rangle$ pairs of types were randomly sampled by selecting x and y each independently, with replacement, from those types with at least 10 tokens in the stream (where 310,327 types satisfied this constraint). The true cosine values between each such x and y was computed based on offline calculation, and compared to the cosine similarity predicted by the Hamming distance between the signatures for x and y. Unless otherwise specified, the random pool size was fixed at $m = 10,000$.

Figure 1 visually reaffirms the trade-off in LSH between the number of bits and the accuracy of cosine prediction across the range of cosine values. As the underlying vectors are strictly positive, the true cosine is restricted to $[0, 1]$. Figure 2 shows the absolute error between truth and prediction for a similar sample, measured using signatures of a variety of bit lengths. Here we see horizontal bands arising from truly orthogonal vectors leading to step-wise absolute error values tracked to Hamming distance.

Table 1 compares the online and batch LSH algorithms, giving the mean absolute error between predicted and actual cosine values, computed for the fifty-thousand element sample, using signatures of various lengths. These results confirm that we achieve the same level of accuracy with online updates as compared to the standard method.

Figure 3 shows how a pool size as low as $m = 100$ gives reasonable variation in random values, and that $m = 10,000$ is sufficient. When using a standard 32 bit floating point representation, this is just 40 KBytes of memory, as compared to, e.g., the 2.5 GBytes required to store 256 random vectors each containing 2.5 million elements.

Table 2 is based on taking an example for each of three part-of-speech categories, and reporting the resultant top-5 words as according to approximated cosine similarity. Depending on the intended application, these results indicate a range

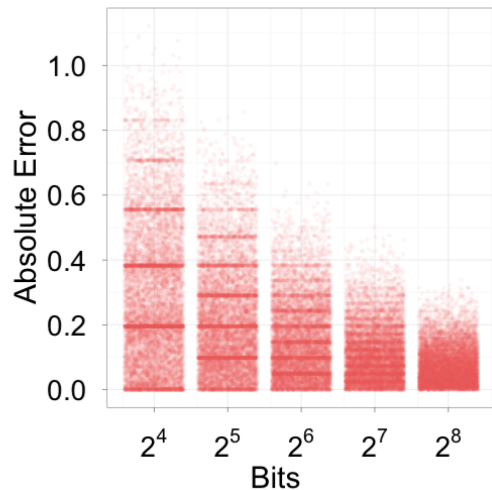

Figure 2: Absolute error between predicted and true cosine for a sample of pairs, when using signatures of length $\log_2(d) \in \{4, 5, 6, 7, 8\}$, drawn with added *jitter* to avoid overplotting.

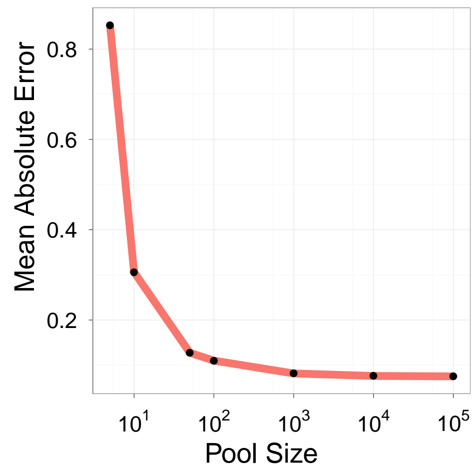

Figure 3: Error versus pool size, when using $d = 256$.

of potentially sufficient signature lengths.

5 Conclusions

We have shown that when updates to a feature vector are additive, it is possible to convert the offline LSH signature generation method into a streaming algorithm. In addition to allowing for online querying of signatures, our approach leads to space efficiencies, as it does not require the explicit representation of either the feature vectors, nor the random matrix. Possibilities for future work include the pairing of this method with algorithms for dynamic clustering, as well as exploring algorithms for different distances (e.g., L_2) and estimators (e.g., asymmetric estimators (Dong et al., 2009)).

London

Milan$_{.97}$, Madrid$_{.96}$, Stockholm$_{.96}$, Manila$_{.95}$, Moscow$_{.95}$
ASHER$_0$, Champaign$_0$, MANS$_0$, NOBLE$_0$, come$_0$
Prague$_1$, Vienna$_1$, suburban$_1$, synchronism$_1$, Copenhagen$_2$
Frankfurt$_4$, Prague$_4$, Taszar$_5$, Brussels$_6$, Copenhagen$_6$
Prague$_{12}$, Stockholm$_{12}$, Frankfurt$_{14}$, Madrid$_{14}$, Manila$_{14}$
Stockholm$_{20}$, Milan$_{22}$, Madrid$_{24}$, Taipei$_{24}$, Frankfurt$_{25}$

in

during$_{.99}$, on$_{.98}$, beneath$_{.98}$, from$_{.98}$, onto$_{.97}$
Across$_0$, Addressing$_0$, Addy$_0$, Against$_0$, Allmon$_0$
aboard$_0$, mishandled$_0$, overlooking$_0$, Addressing$_1$, Rejecting$_1$
Rejecting$_2$, beneath$_2$, during$_2$, from$_3$, hamstringing$_3$
during$_4$, beneath$_5$, of$_6$, on$_7$, overlooking$_7$
during$_{10}$, on$_{13}$, beneath$_{15}$, of$_{17}$, overlooking$_{17}$

sold

deployed$_{.84}$, presented$_{.83}$, sacrificed$_{.82}$, held$_{.82}$, installed$_{.82}$
Bustin$_0$, Diors$_0$, Draining$_0$, Kosses$_0$, UNA$_0$
delivered$_2$, held$_2$, marks$_2$, seared$_2$, Ranked$_3$
delivered$_5$, rendered$_5$, presented$_6$, displayed$_7$, exhibited$_7$
held$_{18}$, rendered$_{18}$, presented$_{19}$, deployed$_{20}$, displayed$_{20}$
presented$_{41}$, rendered$_{42}$, held$_{47}$, leased$_{47}$, reopened$_{47}$

Table 2: Top-5 items based on true cosine (bold), then using minimal Hamming distance, given in top-down order when using signatures of length $\log_2(d) \in \{4, 5, 6, 7, 8\}$. Ties broken lexicographically. Values given as subscripts.

Acknowledgments

Thanks to Deepak Ravichandran, Miles Osborne, Sasa Petrovic, Ken Church, Glen Coppersmith, and the anonymous reviewers for their feedback. This work began while the first author was at the University of Rochester, funded by NSF grant IIS-1016735. The second author was supported in part by NSF grant CNS-0905169, funded under the American Recovery and Reinvestment Act of 2009.

References

Moses Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of STOC*.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI*.

Wei Dong, Moses Charikar, and Kai Li. 2009. Asymmetric distance estimation with sketches for similarity search in high-dimensional spaces. In *Proceedings of SIGIR*.

Michel X. Goemans and David P. Williamson. 1995. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *JACM*, 42:1115–1145.

Amit Goyal, Hal Daumé III, and Suresh Venkatasubramanian. 2009. Streaming for large scale NLP: Language Modeling. In *Proceedings of NAACL*.

David Graff. 2003. English Gigaword. Linguistic Data Consortium, Philadelphia.

Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of STOC*.

Abby Levenberg and Miles Osborne. 2009. Stream-based Randomised Language Models for SMT. In *Proceedings of EMNLP*.

Ping Li, Kenneth W. Church, and Trevor J. Hastie. 2008. One Sketch For All: Theory and Application of Conditional Random Sampling. In *Advances in Neural Information Processing Systems 21*.

Sasa Petrovic, Miles Osborne, and Victor Lavrenko. 2010. Streaming First Story Detection with application to Twitter. In *Proceedings of NAACL*.

Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. 2005. Randomized Algorithms and NLP: Using Locality Sensitive Hash Functions for High Speed Noun Clustering. In *Proceedings of ACL*.

David Talbot. 2009. Succinct approximate counting of skewed data. In *Proceedings of IJCAI*.

Benjamin Van Durme and Ashwin Lall. 2009a. Probabilistic Counting with Randomized Storage. In *Proceedings of IJCAI*.

Benjamin Van Durme and Ashwin Lall. 2009b. Streaming Pointwise Mutual Information. In *Advances in Neural Information Processing Systems 22*.