Bayesian Locality Sensitive Hashing for Fast Similarity Search

Venu Satuluri and Srinivasan Parthasarathy
Dept. of Computer Science and Engineering
The Ohio State University
{satuluri,srini}@cse.ohio-state.edu

ABSTRACT

Given a collection of objects and an associated similarity measure, the all-pairs similarity search problem asks us to find all pairs of objects with similarity greater than a certain user-specified threshold. Locality-sensitive hashing (LSH) based methods have become a very popular approach for this problem. However, most such methods only use LSH for the first phase of similarity search - i.e. efficient indexing for candidate generation. In this paper, we present **BayesLSH**, a principled Bayesian algorithm for the subsequent phase of similarity search - performing candidate pruning and similarity estimation using LSH. A simpler variant, BayesLSH-Lite, which calculates similarities exactly, is also presented. Our algorithms are able to quickly prune away a large majority of the false positive candidate pairs, leading to significant speedups over baseline approaches. For BayesLSH, we also provide probabilistic guarantees on the quality of the output, both in terms of accuracy and recall. Finally, the quality of BayesLSH's output can be easily tuned and does not require any manual setting of the number of hashes to use for similarity estimation, unlike standard approaches. For two state-of-the-art candidate generation algorithms, AllPairs and LSH, BayesLSH enables significant speedups, typically in the range 2x-20x for a wide variety of datasets.

1. INTRODUCTION

Similarity search is a problem of fundamental importance for a broad array of fields, including databases, data mining and machine learning. The general problem is as follows: given a collection of objects D with some similarity measure s defined between them and a query object q, retrieve all objects from D that are similar to q according to the similarity measure s. The user may be either interested in the top-k most similar objects to q, or the user may want all objects x such that $s(x, q) > t$, where t is the similarity threshold. A more specific version of similarity search is the *All Pairs similarity search* problem, where there is no explicit query object, but instead the user is interested in all pairs of objects with similarity greater than some threshold. The number of applications even for the more specific all pairs similarity search problem is impressive: clustering [21], semi-supervised learning [29], information retrieval (including text, audio and video), query refinement [3], near-duplicate detection [26], collaborative filtering, link prediction for graphs [17], and 3-D scene reconstruction [1] among others. In many of these applications, approximate solutions with small errors in similarity assessments are acceptable if they can buy significant reductions in running time e.g. in web-scale clustering [5, 21], information retrieval [9], near-duplicate detection for web crawling [19, 11] and graph clustering [24].

Roughly speaking, similarity search algorithms can be divided into two main phases - *candidate generation* and *candidate verification*. During the candidate generation phase, pairs of objects that are good candidates for having similarity above the user-specified threshold are generated using one or another indexing mechanism, while during candidate verification, the similarity of each candidate pair is verified against the threshold, in many cases by exact computation of the similarity. The traditional indexing structures used for candidate generation were space-partitioning approaches such as kd-trees and R-trees, but these approaches work well only in low dimensions (less than 20 or so [7]). An important breakthrough was the invention of locality-sensitive hashing [12, 10], where the idea is to find a family of hash functions such that for a random hash function from this family, two objects with high similarity are very likely to be hashed to the same bucket. One can then generate candidate pairs by hashing each object several times using randomly chosen hash functions, and generating all pairs of objects which have been hashed to the same bucket by at least one hash function. Although LSH is a randomized, approximate solution to candidate generation, similarity search based on LSH has nonetheless become immensely popular because it provides a practical solution for high dimensional applications along with theoretical guarantees for the quality of the approximation [2].

In this article, we show how LSH can be exploited for the phase of similarity search subsequent to candidate generation i.e. candidate verification and similarity computation. We adopt a principled Bayesian approach that allows us to reason about the probability that a particular pair of objects will meet the user-specified threshold by inspecting only a few hashes of each object, which in turn allows us to quickly prune away unpromising pairs. Our Bayesian

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 38th International Conference on Very Large Data Bases, August 27th - 31st 2012, Istanbul, Turkey.
Proceedings of the VLDB Endowment, Vol. 5, No. 5
Copyright 2012 VLDB Endowment 2150-8097/12/01... $ 10.00.

approach also allows us to estimate similarities to a user-specified level of accuracy without requiring any tuning of the number of hashes, overcoming a significant drawback of standard similarity estimation using LSH. We develop two algorithms, called **BayesLSH** and **BayesLSH-Lite**, where the former performs both candidate pruning and similarity estimation, while the latter only performs candidate pruning and computes the similarities of unpruned candidates exactly. Essentially, our algorithms provide a way to trade-off accuracy for speed in a controlled manner. Both BayesLSH and BayesLSH-Lite can be combined with any existing candidate generation algorithm, such as AllPairs [3] or LSH. Concretely, BayesLSH provides the following probabilistic guarantees:

Given a collection of objects D, an associated similarity function $s(.,.)$, and a similarity threshold t; recall parameter ϵ and accuracy parameters δ, γ; return pairs of objects (x, y) along with similarity estimates $\hat{s}_{x,y}$ such that:

1. $Pr[s(x, y) \geq t] > \epsilon$ *i.e. each pair with a greater than ϵ probability of being a true positive is included in the output set.*
2. $Pr[|\hat{s}_{x,y} - s(x, y)| \geq \delta] < \gamma$ *i.e. each associated similarity estimate is accurate up to δ-error with probability $> 1 - \gamma$.*

With BayesLSH-Lite, the similarity calculations are exact, so there is no need for guarantee 2, but guarantee 1 from above stays. We note that the parametrization of BayesLSH is intuitive - the desired recall can be controlled using ϵ, while δ, γ together specify the desired level of accuracy of similarity estimation.

The advantages of BayesLSH are as follows:

1. The general form of the algorithm can be easily adapted to work for any similarity measure with an associated LSH family (see Section 2 for a formal definition of LSH). We demonstrate BayesLSH for Cosine and Jaccard similarity measures, and believe that it can be adapted to other measures with LSH families, such as kernel similarities.
2. There are no restricting assumptions about the specific form of the candidate generation algorithm; BayesLSH *complements* progress in candidate generation algorithms.
3. For applications which already use LSH for candidate generation, it is a natural fit since it exploits the hashes of the objects for candidate pruning, further amortizing the costs of hashing.
4. It works for both binary and general real-valued vectors. This is a significant advantage because recent progress in similarity search has been limited to binary vectors [26, 28].
5. Parameter tuning is easy and intuitive. In particular, there is no need for manually tuning the number of hashes, as one needs to with standard similarity estimation using LSH.

We perform an extensive evaluation of our algorithms and comparison with state-of-the-art methods, on a diverse array of 6 real datasets. We combine BayesLSH and BayesLSH-Lite with two different candidate generation algorithms AllPairs [3] and LSH, and find significant speedups, typically in the range 2x-20x over baseline approaches (see Table 2). BayesLSH is able to achieve the speedups primarily by being extremely effective at pruning away false positive candidate pairs. To take a typical example, BayesLSH is able to prune away 80% of the input candidate pairs after examining only 8 bytes worth of hashes per candidate pair, and 99.98% of the candidate pairs after examining only 32 bytes per pair. Notably, BayesLSH is able to do such effective pruning without adversely affecting the recall, which is still quite high, generally at 97% or above. Furthermore, the accuracy of BayesLSH's similarity estimates is much more consistent as compared to the standard similarity approximation using LSH, which tends to produce very error-ridden estimates for low similarities. Finally, we find that parameter tuning for BayesLSH is intuitive and works as expected, with higher accuracies and recalls being achieved without leading to undue slow-downs.

2. BACKGROUND

Following Charikar [6], we define a locality-sensitive hashing scheme as a distribution on a family of hash functions $\mathcal{F}$ operating on a collection of objects, such that for any two objects $\mathbf{x}, \mathbf{y}$,

$$Pr_{h \in \mathcal{F}}[h(\mathbf{x}) = h(\mathbf{y})] = sim(\mathbf{x}, \mathbf{y}) \quad (1)$$

It is important to note that the probability in Eqn 1 is for a random selection of the hash function from the family $\mathcal{F}$. Specifically, it is not for a random pair $\mathbf{x}, \mathbf{y}$ - i.e. the equation is valid for *any* pair of objects $\mathbf{x}$ and $\mathbf{y}$. The output of the hash functions may be either bits (0 or 1), or integers. Note that this definition of LSH, taken from [6], is geared towards similarity measures and is more useful in our context, as compared to the slightly different definition of LSH used by many other sources [7, 2], including the original LSH paper [12], which is geared towards *distance* measures.

Locality-sensitive hashing schemes have been proposed for a variety of similarity functions thus far, including Jaccard similarity [4, 16], Cosine similarity [6] and kernelized similarity functions (representing e.g. a learned similarity metric) [13].

Candidate generation via LSH:
One of the main reasons for the popularity of LSH is that it can be used to construct an index that enables efficient candidate generation for the similarity search problem. Such LSH-based indices have been found to significantly outperform more traditional indexing methods based on space partitioning approaches, especially with increasing dimensions [12, 7]. The general method works as follows [12, 7, 5, 21, 11]. For each object in the dataset, we will form l signatures, where each signature is a concatenation of k hashes. All pairs of objects that share at least one of the l signatures will be generated as a candidate pair. Retrieving each pair of objects that share a signature can be done efficiently using hashtables. For a given k and similarity threshold t, the number of length-k signatures required for an expected false negative rate ϵ can be shown to be $l = \lceil \frac{\log \epsilon}{\log(1-t^k)} \rceil$ [27].

Candidate verification and similarity estimation:
The similarity between the generated candidates can be computed in one of two ways: (a) by exact calculation of the similarity between each pair, or (b) using an estimate of the similarity, as the fraction of hashes that the two objects agree upon. The pairs of objects with estimated similarity greater than the threshold are finally output. In terms of

running time, approach (b) is often faster, especially when the number of candidates is large and/or exact similarity calculations are expensive, such as with more complex similarity measures or with larger vector lengths. The main overhead with approach (b) is in hashing each point sufficient number of times in the first place, but this cost is amortized over many similarity computations (especially in the case of all-pairs similarity search), and furthermore we need the hashes for candidate generation in any case. However, what is less clear is how good this simple estimation procedure is in terms of accuracy, and whether it can be made any faster. We will address these questions next.

3. CLASSICAL SIMILARITY ESTIMATION FOR LSH

Similarity estimation for a candidate pair using LSH can be considered as a statistical parameter inference problem. The parameter we wish to infer is the similarity, and the data we observe is the outcome of the comparison of each successive hash between the candidate pair. The probability model relating the parameter to the data is given by the main LSH equation, Equation 1. There are two main schools of statistical inference - classical (frequentist) and Bayesian.

Under classical (frequentist) statistical inference, the parameters of a probability model are treated as fixed, and it is considered meaningless to make probabilistic statements about the parameters - hence the output of classical inference is simply a point estimate, one for each parameter. The best known example of frequentist inference is maximum likelihood estimation, where the value of the parameter that maximizes the probability of the observed data is output as the point estimate. In the case of similarity estimation via LSH, let us say we have compared n hashes and have observed m agreements in hash values. The maximum likelihood estimator for the similarity $\hat{s}$ is:[1]

$$\hat{s} = \frac{m}{n}$$

While previous researchers have not explicitly labeled their approaches as using the maximum likelihood estimators, they have implicitly used the above estimator, tuning the number of hashes n [21, 6]. However, this approach has some important drawbacks, which we turn to next.

3.1 Difficulty of tuning the number of hashes

While the above estimator is unbiased, the variance is $\frac{s*(1-s)}{n}$, meaning that the variance of the estimator depends on the similarity s being estimated. This indicates that in order to get the same level of accuracy for different similarities, we will need to use *different* number of hashes.

We can be more precise and, for a given similarity, calculate exactly the the probability of a smaller-than-δ error in $\hat{s}_n$, the similarity estimated using n hashes.

$$\begin{aligned} Pr[|\hat{s}_n - s| < \delta] &= Pr[(s-\delta)*n \leq m \leq (s+\delta)*n] \\ &= \sum_{m=(s-\delta)*n}^{(s+\delta)*n} \binom{n}{m} s^m (1-s)^{n-m} \end{aligned}$$

Using the above expression, we can calculate the minimum number of hashes needed to ensure that the similarity estimate is sufficiently concentrated, i.e within δ of the true value with probability $1-\gamma$. A plot of the number of hashes required for $\delta = \gamma = 0.05$ for various similarity values is given in Figure 1. As can be seen, there is a great difference in the number of hashes required when the true similarities are different; similarities closer to 0.5 require far more hashes to estimate accurately than similarities close to 0 or 1. A similarity of 0.5 needs 350 hashes for sufficient accuracy, but a similarity of 0.95 needs only 16 hashes! Stricter accuracy requirements lead to even greater differences in the required number of hashes.

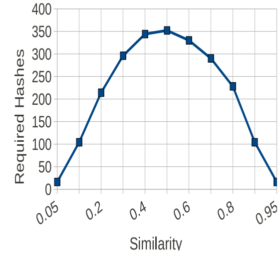


Figure 1: Hashes vs. similarity

Since we don't know the true similarity of each pair a priori, we cannot choose the right number of hashes beforehand. If we err on the side of accuracy and choose a large n, then performance suffers since we will be comparing many more hashes than are necessary for some candidate pairs. If, on the other hand, we err on the side of performance and choose a smaller n, then accuracy suffers. With standard similarity estimation, therefore, it is *impossible* to tune the number of hashes for the entire dataset so as to achieve both optimal performance and accuracy.

3.2 Absence of early pruning

In the context of similarity search with a user-specified threshold, the standard similarity estimation procedure also misses opportunities for *early candidate pruning*. The intuition here is best illustrated using an example: Let us say the similarity threshold is 0.8 i.e. the user is only interested in pairs with similarity greater than 0.8. Let us say the similarity estimation is going to use $n = 1000$ hashes. But if we are examining a candidate pair for which, out of the first 100 hashes, only 10 hashes matched, then intuitively it seems very likely that this pair does not meet the threshold of 0.8. In general, it seems intuitively possible to be able to prune away many false positive candidates by looking only at the first few hashes, without needing to compare all the hashes. As we will see, most candidate generation algorithms produce significant number of false positives, and the standard similarity estimation procedure using LSH does not exploit the potential for early pruning of candidate pairs.

4. CANDIDATE PRUNING AND SIMILARITY ESTIMATION USING BAYESLSH

The key characteristic of Bayesian statistics is that it allows one to make probabilistic statements about any aspect of the world, including things that would be considered "fixed" under frequentist statistics and hence meaningless to make probabilistic statements about. In particular, Bayesian statistics allows us to make probabilistic statements about the parameters of probability models - in other words, parameters are also treated as random variables. Bayesian inference generally consists of starting with a *prior* distribution over the parameters, and then computing a *posterior* distribution over the parameters, conditional on the data that we have actually observed, using Bayes' rule. A commonly cited drawback of Bayesian inference is the need

[1]Proofs are elementary and are omitted.

for the prior probability distribution over the parameters, but a reasonable amount of data generally "swamps out" the influence of the prior (see Appendix in [22]). Furthermore, a good prior can lead to *improved* estimates over maximum likelihood estimation - this is a common strategy for avoiding overfitting the data in machine learning and statistics. The big advantage of Bayesian inference in the context of similarity estimation is that instead of just outputting a point estimate of the similarity, it gives us the complete posterior distribution of the similarity. In the rest of this section, we will avoid discussing specific choices for the prior distribution and similarity measure in order to keep the discussion general.

Fix attention on a particular pair (x, y), and let us say that m out of the first n hashes match for this pair. We will denote this event as $M(m, n)$. The conditional probability of the event $M(m, n)$ given the similarity S (here S is a random variable), is given by the binomial distribution with n trials, where the success of each trial is S itself, from the Equation 1. Note that we have already observed the event $M(m, n)$ happening i.e. m and n are not random variables, they are the data.

$$Pr[M(m,n) \,|\, S] \quad = \quad \binom{n}{m} S^m (1-S)^{n-m} \tag{2}$$

What we are interested in knowing is the probability distribution of the similarity S, *given* that we already know that m out of n hashes have matched. Using Bayes' rule, the posterior distribution for S can be written as follows: [2]

$$\begin{aligned} p(S \,|\, M(m,n)) &= \frac{p(M(m,n) \,|\, S)p(S)}{p(M(m,n))} \\ &= \frac{p(M(m,n) \,|\, S)p(S)}{\int_0^1 p(M(m,n), s)ds} \\ &= \frac{p(M(m,n) \,|\, S)p(S)}{\int_0^1 p(M(m,n) \,|\, s)p(s)ds} \end{aligned}$$

By plugging in the expressions for $p(M(m,n) \,|\, S)$ from Equation 2 and a suitable prior distribution $p(S)$, we can get, for every value of n and m, the posterior distribution of S conditional on the event $M(m, n)$. We calculate the following quantities in terms of the posterior distribution:

1. If after comparing n hashes, m matches agree, what is the probability that the similarity is greater than the threshold t?

$$Pr[S \geq t \,|\, M(m,n)] \quad = \quad \int_t^1 p(s \,|\, M(m,n))ds \tag{3}$$

2. If after comparing n hashes, m matches agree, what is the maximum-a-posteriori estimate for the similarity i.e. the similarity value with the highest posterior probability? This will function as our estimate $\hat{S}$

$$\hat{S} = \arg\max_s p(s|M(m,n)) \tag{4}$$

3. Assume after comparing n hashes, m matches agree, and we have estimated the similarity to be $\hat{S}$ (e.g. as indicated above). What is the concentration probability of $\hat{S}$ i.e. probability that this estimate is within δ of the true similarity?

$$\begin{aligned} Pr[|S - \hat{S}| < \delta \,|\, M(m,n)] &= Pr[\hat{S} - \delta < S < \hat{S} + \delta \,|\, M(m,n)] \\ &= \int_{\hat{S}-\delta}^{\hat{S}+\delta} p(s \,|\, M(m,n))ds \end{aligned} \tag{5}$$

Assuming we can perform the above three kinds of inference, we design our algorithm, BayesLSH, so that it satisfies the probabilistic guarantees outlined in Section 1. The algorithm is outlined in Algorithm 1. For each candidate pair (x, y) we incrementally compare their respective hashes (line 8, here the parameter k indicates the number of hashes we will compare at a time), until either one of two events happens. The first possibility is that the candidate pair gets pruned away because the probability of it being a true positive pair has become very small (lines 10, 11 and 12), where we use Equation 3 to calculate this probability. The alternative possibility is that the candidate pair does not get pruned away, and we continue comparing hashes until our similarity estimate (line 14) becomes sufficiently concentrated that it passes our accuracy requirements (lines 15 and 16). Here we use Equation 5 to determine the probability that our estimate is sufficiently accurate. Each such pair is added to the output set of candidate pairs, along with our similarity estimate (lines 19 and 20).

Our second algorithm, BayesLSH-Lite (see Algorithm 2) is a simpler version of BayesLSH, which calculates similarities exactly. Since the similarity calculations are exact, there is no need for parameters δ, γ; however, this comes at the cost of some intuitiveness, as there is a new parameter h specifying the maximum number of hashes that will be examined for each pair of objects. BayesLSH-Lite can be faster than BayesLSH for those datasets where exact similarity calculations are cheap, e.g. because the object representations are simpler, such as binary, or if the average size of the objects is small.

BayesLSH clearly overcomes the two drawbacks of standard similarity estimation explained in Sections 3.1 and 3.2. Any candidate pairs that can be pruned away by examining only the first few hashes will be pruned away by BayesLSH. As we will show later, this method is very effective for pruning away the vast majority of false positives. Secondly, the number of hashes for which each candidate pair is compared is determined automatically by the algorithm, depending on the user-specified accuracy requirements, completely eliminating the need to manually set the number of hashes. Thirdly, each point in the dataset is only hashed as many times as is necessary. This will be particularly useful for applications where hashing a point itself can be costly e.g. for kernel LSH [13]. Also, outlying points which don't have any points with whom their similarity exceeds the threshold need only be hashed a few times before BayesLSH prunes all candidate pairs involving such points away.

In order to obtain a concrete instantiation of BayesLSH, we will need to specify three aspects: (i) the LSH family of hash functions, (ii) the choice of prior and (iii) how to tractably perform inference. Next, we will look at specific instantiations of BayesLSH for different similarity measures.

4.1 BayesLSH for Jaccard similarity

We will first discuss how BayesLSH can be used for approximate similarity search for Jaccard similarity.

[2] In terms of notation, we will use lower-case $p(.)$ for probability density functions of continuous random-variables. $Pr[.]$ is used for probabilities of discrete events or discrete random variables.

Algorithm 1 BayesLSH

```
1: Input: Set of candidate pairs C; Similarity threshold t; recall
   parameter ϵ; accuracy parameters δ, γ
2: Output: Set O of pairs (x, y) along with similarity estimates
   Ŝ_{x,y}
3: O ← ∅
4: for all (x, y) ∈ C do
5:     n, m ← 0 {Initialization}
6:     isPruned ← False
7:     while True do
8:         m = m + Σ_{i=n}^{n+k} I[h_i(x) == h_i(y)] {Compare
           hashes n to n + k}
9:         n = n + k
10:        if Pr[S ≥ t | M(m, n)] < ϵ then
11:            isPruned ← True
12:            break {Prune candidate pair}
13:        end if
14:        Ŝ ← arg max_s p(s|M(m, n))
15:        if Pr[|S − Ŝ | M(m, n) < δ] < γ then
16:            break {Similarity estimate is sufficiently con-
               centrated}
17:        end if
18:    end while
19:    if isPruned == False then
20:        O ← O ∪ {((x, y), Ŝ)}
21:    end if
22: end for
23: return O
```

Algorithm 2 BayesLSH-Lite

```
1: Input: Set of candidate pairs C; Similarity threshold t; recall
   parameter ϵ; Number of hashes to use h
2: Output: Set O of pairs (x, y) along with exact similarities
   Ŝ_{x,y}
3: O ← ∅
4: for all (x, y) ∈ C do
5:     n, m ← 0 {Initialization}
6:     isPruned ← False
7:     while n < h do
8:         m = m + Σ_{i=n}^{n+k} I[h_i(x) == h_i(y)] {Compare
           hashes n to n + k}
9:         n = n + k
10:        if Pr[S ≥ t | M(m, n)] < ϵ then
11:            isPruned ← True
12:            break {Prune candidate pair}
13:        end if
14:    end while
15:    if isPruned == False then
16:        s_{x,y} = similarity(x, y) {Exact similarity}
17:        if s_{x,y} > t then
18:            O ← O ∪ {((x, y), s_{x,y})}
19:        end if
20:    end if
21: end for
22: return O
```

LSH family: The LSH family for Jaccard similarity is the family of minwise independent permutations [4] on the universe from which our collection of sets is drawn. Each hash function returns the minimum element of the input set when the elements of the set are permuted as specified by the hash function (which itself is chosen at random from the family of minwise independent permutations). The output of this family of hash functions, therefore, is an integer representing the minimum element of the permuted set.

Choice of prior: It is common practice in Bayesian inference to choose priors from a family of distributions that is *conjugate* to the likelihood distribution, so that the inference is tractable and also that the posterior belongs to the same distribution family as the prior (indeed, that is the definition of a conjugate prior). The likelihood in this case is given by a binomial distribution, as indicated in Equation 2. The conjugate for the binomial is the *Beta* distribution, which has two parameters $\alpha > 0, \beta > 0$ and is defined on the domain $(0, 1)$. The pdf for $Beta(\alpha, \beta)$ is defined as follows.

$$p(s) = \frac{s^{\alpha-1} * (1-s)^{\beta-1}}{B(\alpha, \beta)}$$

Here $B(\alpha, \beta)$ is the beta function, and it can also be thought of as a normalization constant to ensure the entire distribution integrates to 1.

Even assuming we want to model the prior using a Beta distribution, how do we choose the parameters α, β? A simple choice is to set $\alpha = 1, \beta = 1$, which results in a uniform distribution on $(0, 1)$. However, we can actually learn α, β so as to best fit a random sample of similarities from candidate pairs output by the candidate generation algorithm. Let us assume we have r samples chosen uniformly at random from the total population of candidate pairs generated by the particular candidate generation algorithm being used, and their similarities are $s_1, s_2, \ldots, s_r$. Then we can estimate α, β so as to best model the distribution of similarities among candidate pairs. For Beta distribution, a simple and effective method of learning the parameters is via method-of-moments estimation. In this method, we calculate the sample moments (sample mean and sample variance), assume that they are the true moments of the distribution and solve for the parameter values that will result in the obtained moments. In our case, we have the following estimates for α, β:

$$\hat{\alpha} = \bar{s}\left(\frac{\bar{s}(1-\bar{s})}{\bar{s}_v} - 1\right) \; ; \; \hat{\beta} = (1-\bar{s})\left(\frac{\bar{s}(1-\bar{s})}{\bar{s}_v} - 1\right)$$

where $\bar{s}$ and $\bar{s}_v$ are the sample mean and variance, given as follows:

$$\bar{s} = \frac{\sum_{i=1}^{r} s_i}{r} \; ; \; \bar{s}_v = \frac{\sum_{i=1}^{r} (s_i - \bar{s})^2}{r}$$

Assuming a prior $Beta(\alpha, \beta)$ distribution on the similarity, and we observe the event $M(m, n)$ i.e. m out of the first n hashes match, then the posterior distribution of the similarity looks as follows:

$$\begin{aligned} p(s|M(m,n)) &= \frac{\binom{n}{m} s^m (1-s)^{n-m} s^{\alpha-1}(1-s)^{\beta-1}}{\int_0^1 \binom{n}{m} s^m (1-s)^{n-m} s^{\alpha-1}(1-s)^{\beta-1}} \\ &= \frac{s^{m+\alpha-1}(1-s)^{n-m+\beta-1}}{B(m+\alpha, n-m+\beta)} \end{aligned}$$

Hence, the posterior distribution of the similarity also follows a Beta distribution with parameters $m+\alpha$ and $n-m+\beta$.

Inference: We next show concrete ways to perform inference, i.e. computing Equations 3, 4 and 5.

The probability that similarity is greater than the threshold after observing that m out of the first n hashes match is:

$$\begin{aligned} Pr[S \geq t|M(m,n)] &= \int_t^1 p(s|M(m,n)) \\ &= 1 - I_t(m+\alpha, n-m+\beta) \end{aligned}$$

Above $I_t(.,.)$ refers to the *regularized incomplete beta* function, which gives the cdf for the beta distribution. This function is available in standard scientific computing libraries, where it is typically approximated using continued fractions [8].

Our similarity estimate, after observing m matches in n hashes, will be the mode of the posterior distribution $p(s|M(m,n))$. The mode of $Beta(\alpha,\beta)$ is given by $\frac{\alpha-1}{\alpha+\beta-2}$. Therefore, our similarity estimate after observing that m out of the first n hashes agree is $\hat{S} = \frac{m+\alpha-1}{n+\alpha+\beta-1}$.

The concentration probability of the similarity estimate $\hat{S}$ can be derived as follows (the expression for $\hat{S}$ indicated above can be substituted in the below equations):

$$\begin{aligned} Pr[|\hat{S}-S| < \delta|M(m,n)] &= \int_{\hat{S}-\delta}^{\hat{S}+\delta} p(s|M(m,n))ds \\ &= I_{\hat{S}+\delta}(m+\alpha, n-m+\beta) \\ &- I_{\hat{S}-\delta}(m+\alpha, n-m+\beta) \end{aligned}$$

Thus by substituting the above computations in the corresponding places in Algorithm 1, we obtain a version of BayesLSH specifically adapted to Jaccard similarity.

4.2 BayesLSH for Cosine similarity

We will next discuss instantiating BayesLSH for Cosine similarity.

LSH family: For Cosine similarity, each hash function h_i is associated with a random vector r_i, each of whose components is a sample from the standard Gaussian ($\mu = 0, \sigma = 1$). For a vector x, $h_i(x) = 1$ if $dot(r_i, x) \geq 0$ and $h_i(x) = 0$ otherwise [6]. Note that each hash function outputs a bit, and hence these hashes can be stored with less space.

However, there is one challenge here that needs to be overcome that was absent for BayesLSH with Jaccard similarity: this LSH family is for a slightly different similarity measure than cosine - it is instead for $1 - \frac{\theta(x,y)}{\pi}$, where $\theta(x,y) = \arccos(\frac{dot(x,y)}{||x||.||y||})$. For notational ease, we will refer to this similarity function as $r(x,y)$ i.e. $r(x,y) = 1 - \frac{\theta(x,y)}{\pi}$. Explicitly,

$$Pr[h_i(x) == h_i(y)] = r(x,y)$$

$$Pr[M(m,n)|r] = \binom{n}{m} r^m (1-r)^{n-m}$$

Since the similarity function we are interested in is $cos(x,y)$ and not $r(x,y)$ - in particular, we wish for probabilistic guarantees on the quality of the output in terms of $cos(x,y)$ and not $r(x,y)$ - we will need to somehow express the posterior probability in terms of $s = cos(x,y)$. One can choose to re-express the likelihood in terms of $s = cos(x,y)$ instead of in terms of r but this introduces $cos()$ terms into the likelihood, and makes it very hard to find a suitable prior that keeps the inference tractable. Instead we compute the posterior distribution of r which we transform appropriately into a posterior distribution of s.

Choice of prior: We will need to choose a prior distribution for r. Previously, we used a Beta prior for Jaccard BayesLSH; unfortunately r has range $[0.5, 1]$, while the standard Beta distribution has support on the domain $(0,1)$. We can still map the standard Beta distribution onto the domain $(0.5, 1)$, but this distribution will no longer be conjugate to the binomial likelihood.[3] Our solution is to use a simple uniform distribution on $[0.5, 1]$ as the prior for t. Even when the true similarity distribution is very far from being uniform (as is the case in real datasets, including the ones used in our experiments), this prior still works well because the posterior is strongly influenced by the actual outcomes observed (see Appendix in [22]).

The prior pdf therefore is:

$$p(r) = \frac{1}{1-0.5} = 2$$

The posterior pdf, after observing that m out of the first n hashes agree, is:

$$\begin{aligned} p(r|M(m,n)) &= \frac{2\binom{n}{m} r^m (1-r)^{n-m}}{\int_{0.5}^1 2\binom{n}{m} r^m (1-r)^{n-m} dr} \\ &= \frac{r^m (1-r)^{n-m}}{\int_{0.5}^1 r^m (1-r)^{n-m} dr} \\ &= \frac{r^m (1-r)^{n-m}}{B_1(m+1, n-m+1) - B_{0.5}(m+1, n-m+1)} \end{aligned}$$

Here $B_x(a,b)$ is the incomplete Beta function, defined as $B_x(a,b) = \int_0^x y^{a-1}(1-y)^{b-1}dy$.

Inference: In order to calculate Equations 3, 5 and 4, we will first need a way to convert from r to s and vice-versa. Let $r2c : [0.5, 1] \rightarrow [0, 1]$ be the 1-to-1 function that maps from $r(x,y)$ to $cos(x,y)$; $r2c()$ is given by $r2c(r) = \cos(\pi * (1-r))$. Similarly, let $c2r$ be the 1-to-1 function that does the same map in reverse; $c2r()$ is given by $c2r(c) = 1 - \frac{\arccos(c)}{\pi}$.

Let R be the random variable such that $R = c2r(S)$ and let $t_r = c2r(t)$. After observing that the m out of the first n hashes agree, the probability that cosine similarity is greater than the threshold t is:

$$\begin{aligned} & Pr[S \geq t|M(m,n)] = Pr[c2r(S) \geq c2r(t)|M(m,n)] \\ & = Pr[R \geq t_r|M(m,n)] \\ & = \int_{t_r}^1 p(r|M(m,n))dr \\ & = \frac{\int_{t_r}^1 r^m (1-r)^{n-m} dr}{B_1(m+1, n-m+1) - B_{0.5}(m+1, n-m+1)} \\ & = \frac{B_1(m+1, n-m+1) - B_{t_r}(m+1, n-m+1)}{B_1(m+1, n-m+1) - B_{0.5}(m+1, n-m+1)} \end{aligned}$$

[3] The pdf of a Beta distribution supported only on $(0.5, 1)$ with parameters α, β is $p(x) \propto (x-0.5)^{\alpha-1}(1-x)^{\beta-1}$. With a binomial likelihood, the posterior pdf takes the form $p(x|M(m,n)) \propto x^m (x-0.5)^{\alpha-1}(1-x)^{n-m+\beta-1}$. Unfortunately there is no simple and fast way to integrate this pdf.

The first step in the above derivation follows because $c2r()$ is a 1-to-1 mapping. Thus, we have a concrete expression for calculating Eqn 3.

Next, we need an expression for the similarity estimate $\hat{S}$, given that m out of n hashes have matched so far. Let $\hat{R} = \arg\max_r p(r|M(m,n))$. We can obtain a closed form expression for $\hat{R}$ by solving for $\frac{\partial p(r|M(m,n))}{\partial r} = 0$; when we do this, we get $r = \frac{m}{n}$. Hence, $\hat{R} = \frac{m}{n}$. Now $\hat{S} = r2c(\hat{R})$, therefore $\hat{S} = r2c(\frac{m}{n})$. This is our expression for calculating Eqn 4.

Next, let us consider the concentration probability of $\hat{S}$.

$$
\begin{aligned}
&Pr[|\hat{S} - S| < \delta | M(m,n)] = Pr[\hat{S} - \delta < S < \hat{S} + \delta | M(m,n)] \\
&= Pr[c2r(\hat{S} - \delta) < c2r(S) < c2r(\hat{S} + \delta) | M(m,n)] \\
&= Pr[c2r(\hat{S} - \delta) < R < c2r(\hat{S} + \delta) | M(m,n)] \\
&= \frac{\int_{c2r(\hat{S}-\delta)}^{c2r(\hat{S}+\delta)} r^m (1-r)^{n-m} dr}{B_1(m+1, n-m+1) - B_{0.5}(m+1, n-m+1)} \\
&= \frac{B_{c2r(\hat{S}+\delta)}(m+1, n-m+1) - B_{c2r(\hat{S}-\delta)}(m+1, n-m+1)}{B_1(m+1, n-m+1) - B_{0.5}(m+1, n-m+1)}
\end{aligned}
$$

Thus, we have concrete expressions for Equations 3, 4 and 5, giving us an instantiation of BayesLSH adapted to Cosine similarity.

4.3 Optimizations

The basic BayesLSH can be optimized without affecting the correctness of the algorithm in a few ways. The main idea behind the optimizations here is to minimize the number of times inference has to be performed, in particular the Equations 3 and 5.

Pre-computation of minimum matches: We pre-compute the minimum number of matches a candidate pair needs to have in order for $Pr[S \geq t | M(m,n)] > \epsilon$ to be true, thus completely eliminating the need for any online inference in line 10 of Algorithm 1. For every value of n that we will consider (upto some maximum), we pre-compute the function $minMatches(n)$ defined as follows:

$$minMatches(n) = \arg\min_m Pr[S \geq t | M(m,n)] \geq \epsilon$$

This can be done via binary search, since $Pr[S \geq t | M(m,n)]$ increases monotonically with m for a fixed n. Now, for each candidate pair, we simply check if the actual number of matches for that pair at every n is at least $minMatches(n)$. Note that we will not encounter every possible value of n upto the maximum - instead, since we compare k hashes at a time, we need to compute $minMatches()$ only once for all multiples of k upto the maximum.

Cache results of inference: We maintain a cache indexed by (m,n) that indicates whether or not the similarity estimate that is obtained after m hashes out of n agree is sufficiently concentrated or not (Equation 5). Note that for each possible n, we only need to cache the results for $m \geq minMatches(n)$, since lower values of m are guaranteed to result in pruning. Thus, in the vast majority of cases, we can simply fetch the result of the inference from the cache instead of having to perform it afresh.

Cheaper storage of hash functions: For cosine similarity, storing the random Gaussian vectors corresponding to each hash function can take up a fair amount of space. To reduce this storage requirement, we developed a scheme for storing each float using only 2 bytes, by exploiting the fact that random Gaussian samples from the standard 0-mean, 1-standard deviation Gaussian lie well within a small interval around 0. Let us assume that all of our samples will lie within the interval (-8,8) (it is astronomically unlikely that a sample from the standard Gaussian lies outside this interval). For any float $x \in (-8, 8)$, it can be represented as a 2-byte integer $x' = \lfloor (x+8) * \frac{2^{16}}{16} \rfloor$. The maximum error of this scheme is 0.0001 for any real number in $(-8, 8)$.

5. EXPERIMENTS

We experimentally evaluated the performance of BayesLSH and BayesLSH-Lite on 6 real datasets with widely varying characteristics (see Table 1).

- **RCV1** is a text corpus of Reuters articles and is a popular benchmarking corpus for text-categorization research [15]. We use the standard pre-processed version of the dataset with word stemming and tf-idf weighting.
- **Wiki** datasets. We pre-processed the article dump of the English Wikipedia[4] - Sep 2010 version - to produce both a text corpus of Wiki articles as well as the directed graph of hyperlinks between Wiki articles. Our pre-processing includes the removal of stop-words, removal of insignificant articles, and tf-idf weighting (for both the the text and the graph). Words occurring at least 20 times in the entire corpus are used as features, resulting in a dimensionality of 344,352. The **WikiWords100K** dataset consists of text vectors with at least 500 non-zero features, of which there are 100,528. The **WikiWords500K** dataset consists of vectors with at least 200 non-zero features, of which there are 494,244. The **WikiLinks** dataset consists of the entire article-article graph among ˜1.8M articles, with Tf-Idf weighting.
- **Orkut** consists of a subset of the (undirected) friendship network among nearly 3M Orkut users, made available by [20]. Each user is represented as a weighted vector of their friends, with Tf-Idf weighting.
- **Twitter** consists of the directed graph of follower / followee relationships among the subset of Twitter users with at least 1,000 followers, first collected by Kwak et. al. [14]. Each user is represented as a weighted vector of the users they follow, with Tf-Idf weighting.

We note that all our datasets represent realistic applications for all pairs similarity search. Similarity search on text corpuses can be useful for clustering, semi-supervised learning, near-duplicate detection etc., while similarity search on the graph datasets can be useful for link prediction, friendship recommendation and clustering. Also, in our experiments we primarily focus on similarity search for general real-valued vectors using Cosine similarity, as opposed to similarity search for binary vectors (i.e. sets). Our reasons are as follows:

1. Representations of objects as general real-valued vectors are generally more powerful and lead to better similarity assessments, Tf-Idf style representations being the classic example here (see [23] for another example from graph mining).

[4]http://download.wikimedia.org

Dataset	Vectors	Dimensions	Len	Nnz
RCV1	804,414	47,236	76	61e6
WikiWords100K	100,528	344,352	786	79e6
WikiWords500K	494,244	344,352	398	196e6
WikiLinks	1,815,914	1,815,914	24	44e6
Orkut	3,072,626	3,072,626	76	233e6
Twitter	146,170	146,170	1369	200e6

Table 1: Dataset details. Len stands for average length of the vectors and Nnz stands for total number of non-zeros in the dataset.

2. Similarity search is generally harder on real-valued vectors. With binary vectors (sets), most similarity measures are directly proportional to the overlap between the two sets, and it is easier to obtain bounds on the overlap between two sets by inspecting only a few elements of each set, since each element in the set can only contribute the same, fixed number (1) to the overlap. On the other hand, with general real-valued vectors, different elements/features have different weights (also, the same feature may have different weights across different vectors), meaning that it is harder to bound the similarity by inspecting only a few elements of the vector.

5.1 Experimental setup

We compare the following methods for all-pairs similarity search.

1. AllPairs [3] (AP) is one of the state-of-the-art approaches for all-pairs similarity search, especially for cosine similarity on real-valued vectors. AllPairs is an exact algorithm.

2,3. AP+BayesLSH, AP+BayesLSH-Lite: These are variants of BayesLSH and BayesLSH-Lite where the input is the candidate set generated by AllPairs.

4,5. LSH, LSH Approx: These are two variants of the standard LSH approach for all pairs similarity search. For both LSH and LSH Approx, candidate pairs are generated as described in Section 2 ; for LSH, similarities are calculated exactly, whereas for LSH Approx, similarities are instead *estimated* using the standard maximum likelihood estimator, as described in Section 3. For LSH Approx, we tuned the number of hashes and set it to 2048 for cosine similarity and 360 for Jaccard similarity. Note that the hashes for Cosine similarity are only bits, while the hashes for Jaccard are integers.

6,7. LSH+BayesLSH, LSH+BayesLSH-Lite: These are variants of BayesLSH that take as input the candidate set generated by LSH as described in Section 2.

8. PPJoin+ [26] is a state-of-the-art exact algorithm for all-pairs similarity search, however it only works for binary vectors and we only include it in the experiments with Jaccard and binary cosine similarity.

For all BayesLSH variants, we report the full execution time i.e. including the time for candidate generation. For BayesLSH variants, $\epsilon = \gamma = 0.03$ and $\delta = 0.05$ (γ, δ don't apply to BayesLSH-Lite). For the number of hashes to be compared at a time, k, it makes sense to set this to be a multiple of the word size, since for cosine similarity, each hash is simply a bit. We set $k = 32$, although higher multiples of the word size work well too. In the case of BayesLSH-Lite, the number of hashes to be used for pruning was set to $h = 128$ for Cosine and $h = 64$ for Jaccard. For LSH and LSH Approx, the expected false negative rate is set to 0.03 . The randomized algorithms (LSH variants, BayesLSH variants) were each run 3 times and the average results are reported.

All of the methods work for both Cosine and Jaccard similarities, for both real-valued as well as binary vectors, except for PPJoin+, which only works for binary vectors. The code for PPJoin+ was downloaded from the authors' website, all the other methods were implemented by us.[5] All algorithms are single-threaded and are implemented in C/C++. The experiments were run by submitting jobs to a cluster, where each node on the cluster runs on a dual-socket, dual-core 2.3 GHz Opteron with 8GB RAM. Each algorithm was allowed 50 hrs (180K secs) before it was declared timed out and killed.

We executed the different algorithms on both the weighted and binary versions of the datasets, using Cosine similarity for the weighted case and both Jaccard and Cosine for the binary case. For Cosine similarity, we varied the similarity threshold from 0.5 to 0.9, but for Jaccard we found that very few pairs satisfied higher similarity thresholds (e.g. for Orkut, a 3M record dataset, only 1648 pairs were returned at threshold 0.9), and hence varied the threshold from 0.3 to 0.7. For Jaccard and Binary Cosine, we only report results on WikiWords500K, Orkut and Twitter, which are our three largest datasets in terms of total number of non-zeros.

5.2 Results comparing BayesLSH variants with baselines

Figure 3 shows a comparison of timing results for all algorithms across a variety of datasets and thresholds. Table 2 compares the fastest BayesLSH variant with all the baselines. The quality of the output of BayesLSH can be seen in Table 3 where we show the recall rates for AP+BayesLSH and AP+BayesLSH-Lite, and in Table 4 where we compare the accuracies of LSH and LSH+BayesLSH. The recall and accuracies of the other BayesLSH variants follow similar trends and are omitted. The main trends from the results are distilled and discussed below:

1. BayesLSH and BayesLSH-Lite improve the running time of both AllPairs and LSH in almost all the cases, with speedups usually in the range **2x-20x**. It can be seen from Table 2 that a BayesLSH variant is the fastest algorithm (in terms of total time across all thresholds) for the majority of datasets and similarities, with the exception of Orkut for Jaccard and binary cosine. Furthermore, the quality of BayesLSH output is high; the recall rates are usually above 97% (see Table 3), and similarity estimates are accurate, with usually no more than 5% output pairs with error above 0.05 (see Table 4).

2. BayesLSH is fast primarily by being able to prune away the vast majority of false positives after comparing only a few hashes. This is illustrated in Figure 4. For WikiWords100K at a threshold of 0.7, (see Figure 4(a)) AllPairs supplies BayesLSH with nearly 5e09 candidates, while the result set only has 2.2e05. BayesLSH is able to prune away **4.0e+09** (80%) of the input candidate pairs after examining only 32 hashes - in this case, each hash is a bit, so BayesLSH

[5] Our AllPairs implementation is slightly faster than the original implementation of the authors due to a simple implementational fix. This has since been incorporated into the authors' implementation.

compared only 4 bytes worth of hashes between each pair. By the time BayesLSH has compared 128 hashes (16 bytes) there are only 1.0e06 candidates remaining. Similarly LSH supplies BayesLSH with 6.0e08 candidates - better than AllPairs, but nonetheless orders of magnitude larger than the final result set - and after comparing 128 hashes (16 bytes), BayesLSH is able to prune that down to only 7.4e05, only about 3.5x larger than the result set. On the WikiLinks dataset (see Figure 4(b)), we see a similar trend with the roles of AllPairs and LSH reversed - this time it is AllPairs instead which supplies BayesLSH with fewer candidates. After examining only 128 hashes, BayesLSH is able to reduce the number of candidates from 1.3e09 down to 1.2e07 for AllPairs, and from 1.8e11 down to 5.1e07 for LSH. Figure 4(c) shows a similar trend, this time on the binary version of WikiWords100K.

3. We note that BayesLSH and BayesLSH-Lite often (but not always) have comparable speeds, since most of the speed benefit is coming from the ability of BayesLSH to prune, which is an aspect that is common to both algorithms. The difference between the two is mainly in terms of the hashing overhead. BayesLSH needs to obtain many more hashes of each object in order for similarity estimation; this cost is amortized at lower thresholds, where the number of similarity calculations needed to perform is much greater. BayesLSH-Lite is faster at higher thresholds or when exact similarity calculations are cheaper, such as datasets with low average vector length.

4. AllPairs and LSH have complementary strengths and weaknesses. On the datasets RCV1, WikiWords100K, WikiWords500K and Twitter (see Figures 3(a)-3(c),3(f)), LSH is clearly the faster algorithm than AllPairs (in the case of WikiWords500K, AllPairs did not finish execution even for the highest threshold of 0.9). On the other hand, AllPairs is the much faster algorithm on WikiLinks and Orkut (see Figures 3(d)-3(e)), with LSH timing out in most cases. Looking at the characteristics of the datasets, one can discern a pattern: AllPairs is faster on datasets with smaller average length and greater variance in the vector lengths, as is the case with the graph datasets WikiLinks and Orkut. The variance in the vector lengths allows AllPairs to upper-bound the similarity better and thus prune away more false positives, and in addition the exact similarity computations that AllPairs does are faster when the average vector length is smaller. However, BayesLSH and BayesLSH-Lite enable speedups on *both* AllPairs and LSH, not only when each algorithm is slow, but even when each algorithm is *already fast*.

5. The accuracy of BayesLSH's similarity estimates is much more consistent as compared to the standard LSH approximation, as can be seen from Table 4. LSH generally produces too many errors when the threshold is low and too few errors when the threshold is high. This is mainly because LSH uses the same number of hashes (set to 2048) for estimating all similarities, low and high. This problem would persist even if the number of hashes was set to some other value, as explained in Section 3.1. BayesLSH, on the other hand, maintains similar accuracies at both low and high thresholds, *without requiring any tuning at all on the number of hashes to be compared*, and only based on the user's specification of the desired accuracy using δ, γ parameters.

6. LSH Approx is often much faster than LSH with exact similarity calculations, especially for datasets with higher average vector lengths, where the speedup is often 3x or more - on Twitter, the speedup is as much as 10x (see Figure 3(f)).

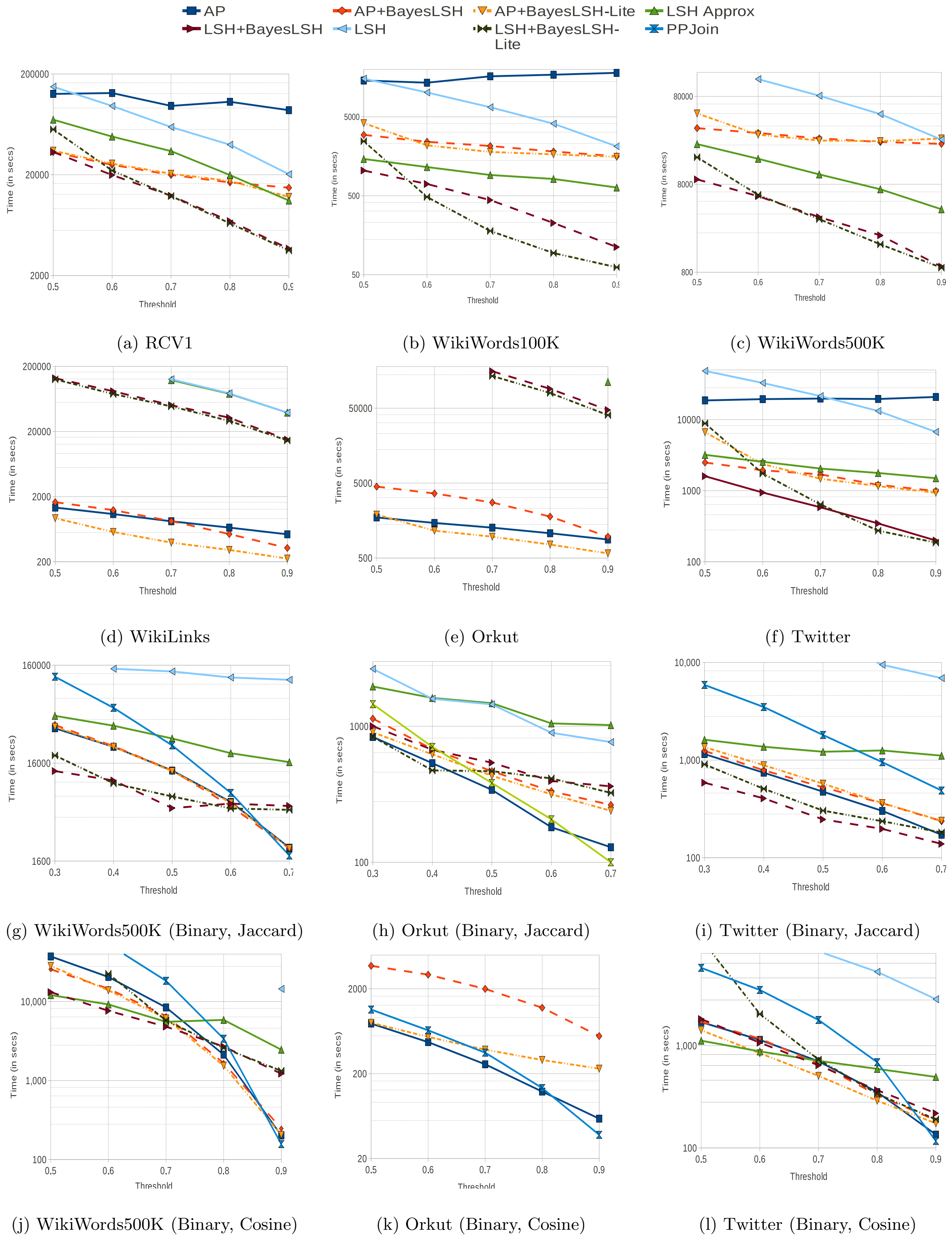
7. BayesLSH does not enable speedups that are as significant for AllPairs in the case of binary vectors. We found that this was because AllPairs was already doing a very good job at generating a small candidate set, thus not leaving much room for improvement. In contrast, LSH was still generating a large candidate set, leaving room for LSH+BayesLSH to enable speedups. Interestingly, even though LSH generates about 10 times more candidates than AllPairs, the LSH variants of BayesLSH are about 50-100% *faster* than AllPairs and its BayesLSH versions, on WikiWords500K and Twitter (see Figures 3(g),3(i)). This is because LSH is a faster indexing and candidate generation strategy, especially when the average vector length is large.

8. PPJoin+ is often the fastest algorithm at the highest thresholds (see Figures 3(g)-3(l)), but its performance degrades very rapidly with lower thresholds. A possible explanation is that the pruning heuristics used in PPJoin+ are effective only at higher thresholds.

5.3 Effect of varying parameters of BayesLSH

We next examine the effect of varying the parameters of BayesLSH - namely the accuracy parameters γ, δ and the recall parameter ϵ. We vary each parameter from 0.01 to 0.09 in increments of 0.02, while fixing the other two parameters to 0.05, and fix the dataset to WikiWords100K and threshold to 0.7 (cosine similarity). The effect of varying each of these parameters on the execution time is plotted in Figure 2. Varying the recall parameter ϵ and the accuracy parameter γ have barely any effect on the running time - however setting δ to lower values does increase the running time significantly. Why does lowering δ penalize the running time much more than lowering γ? This is because lowering δ increases the number of hashes that have to be compared for *all* result pairs, while lowering γ increases the number of hashes that have to be compared only for those result pairs that have uncertain similarity estimates. It is interesting to note that even though $\delta = 0.01$ requires 2691 secs, it achieves a very low mean error of 0.001, while being much faster than LSH exact, which requires 6586 secs. Approximate LSH requires 883 secs but is much more error-prone, with a mean error of 0.014. With $\gamma = 0.01$, BayesLSH achieves a mean error of 0.013, while still being around 2x faster than approximate LSH.

In Table 5, we show the result of varying these parameters on the output quality. When varying a parameter, we show the change in output quality only for the relevant quality metric - e.g. for changing γ we only show how the fraction of errors > 0.05 changes, since we find that recall is largely unaffected by changes in γ and δ (which is as it should be). Looking at the column corresponding to varying γ, we find that the fraction of errors > 0.05 increases as we expect it to when we increase γ, without ever exceeding γ itself. When varying δ, we can see that the mean error reduces as expected for lower values of δ. Finally, when varying the recall parameter ϵ, we find that the recall reduces with higher values of ϵ as expected, with the false negative rate always less than ϵ itself.



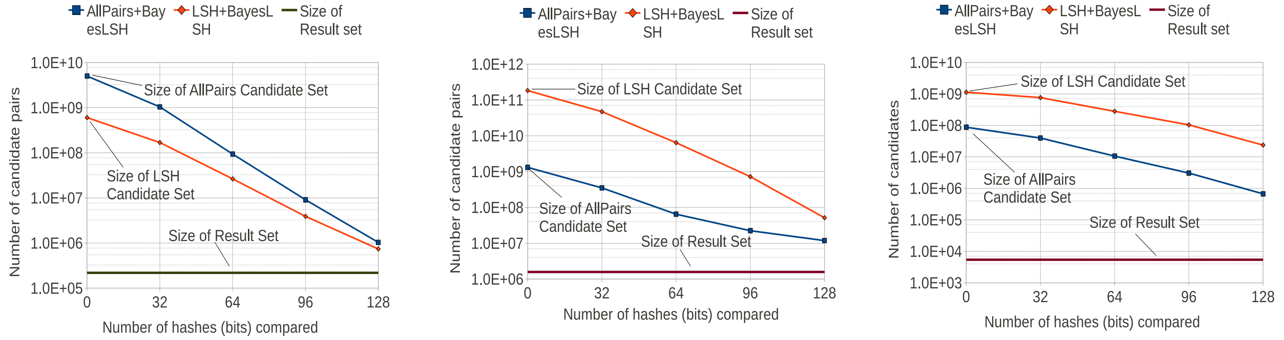
(a) RCV1 (b) WikiWords100K (c) WikiWords500K

(d) WikiLinks (e) Orkut (f) Twitter

(g) WikiWords500K (Binary, Jaccard) (h) Orkut (Binary, Jaccard) (i) Twitter (Binary, Jaccard)

(j) WikiWords500K (Binary, Cosine) (k) Orkut (Binary, Cosine) (l) Twitter (Binary, Cosine)

Figure 3: Timing comparisons between different algorithms. Missing lines/points are due to the respective algorithm not finishing within the allotted time (50 hours).



(a) WikiWords100K, t=0.7, Cosine (b) WikiLinks, t=0.7, Cosine (c) WikiWords100K, t=0.7, Binary Cosine

Figure 4: BayesLSH can prune the vast majority of false positive candidate pairs by examining only a small number of hashes, resulting in major gains in the running time.

Dataset	Fastest BayesLSH variant	Speedup w.r.t baselines			
		AP	LSH	LSH Approx	PPJoin
Tf-Idf, Cosine					
RCV1	LSH + BayesLSH	7.1x	4.8x	2.4x	-
WikiWords-100K	LSH + BayesLSH	31.4x	15.1x	2.0x	-
WikiWords-500K	LSH + BayesLSH	$\geq$ 42.1x	$\geq$ 13.3x	2.8x	-
WikiLinks	AP + BayesLSH-Lite	1.8x	$\geq$ 248.2x	$\geq$ 246.3x	-
Orkut	AP + BayesLSH-Lite	1.2x	$\geq$ 114.9x	$\geq$ 155.6x	-
Twitter	LSH + BayesLSH	26.7x	33.4x	3.0x	-
Binary, Jaccard					
WikiWords-500K	LSH + BayesLSH	2.0x	$\geq$ 16.8x	3.7x	5.2x
Orkut	AP + BayesLSH-Lite	0.8x	2.9x	2.8x	1.1x
Twitter	LSH + BayesLSH	1.8x	48.4x	4.2x	8.0x
Binary, Cosine					
WikiWords-500K	LSH + BayesLSH	2.3x	$\geq$ 10.2x	1.2x	5.6x
Orkut	AP + BayesLSH-Lite	0.8x	$\geq$ 201x	$\geq$ 201x	1.0x
Twitter	AP + BayesLSH-Lite	1.2x	27.4x	1.2x	3.7x

Table 2: Fastest BayesLSH variant for each dataset (based on total time across all thresholds), and speedups over each baseline. BayesLSH variants are fastest in all cases except for binary versions of Orkut, where it is only slightly sub-optimal. The range of thresholds for Cosine was 0.5 to 0.9, and for Jaccard was 0.3 to 0.7. PPJoin is only applicable to binary datasets. In some cases, only lower-bound on speedup is available as the baselines timed out (indicated with $\geq$).

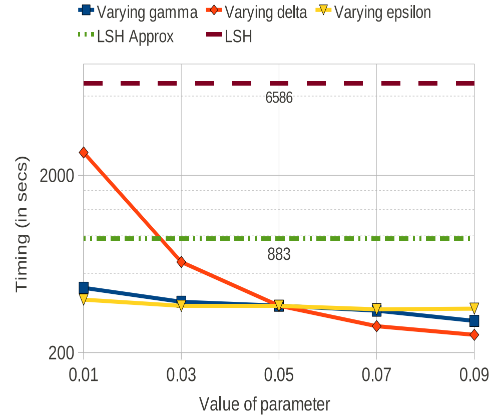


Figure 2: Effect of varying γ, δ, ϵ separately on the running time of LSH+BayesLSH. The dataset is WikiWords100K, with the threshold fixed at t=0.7 for cosine similarity. For comparison, the times for LSH Approx and LSH are also shown.

Dataset	t=0.5	t=0.6	t=0.7	t=0.8	t=0.9
AllPairs+BayesLSH					
RCV1	97.97	98.18	98.47	99.08	99.36
WikiWords100K	98.52	98.84	99.2	98.58	96.69
WikiWords500K	97.54	97.82	98.21	98.16	96.66
WikiLinks	97.45	98.04	98.46	98.68	99.18
Orkut	97.1	97.8	98.86	99.84	99.99
Twitter	97.7	96	96.88	97.33	98.77
AllPairs+BayesLSH-Lite					
RCV1	98.73	98.82	98.89	99.26	99.55
WikiWords100K	98.88	99.31	99.62	99.69	99.5
WikiWords500K	98.79	98.72	98.98	98.74	98.83
WikiLinks	98.53	98.91	99.16	99.18	99.45
Orkut	98.4	98.64	99.3	99.87	99.99
Twitter	99.44	98.82	97.17	97.18	99.06

Table 3: Recalls (out of 100) of AllPairs+BayesLSH and AllPairs+BayesLSH-Lite across different datasets and different similarity thresholds.

	t=0.5	t=0.6	t=0.7	t=0.8	t=0.9
LSH Approx					
RCV1	7.8	4.3	2.25	0.8	0.04
WikiWords100K	4.7	3.6	1	0.3	0.02
WikiWords500K	8.3	5.7	2.9	0.9	0.1
WikiLinks	-	-	1.6	0.4	0.06
Orkut	-	-	-	-	0.0072
Twitter	4	5.1	2.6	0.4	0.02
LSH + BayesLSH					
RCV1	3.2	2.9	3.2	2	1.4
WikiWords100K	2.7	2.3	3.5	4.9	2.2
WikiWords500K	3.4	3.4	3.2	2.9	2.1
WikiLinks	2.96	2.82	2.3	2	1.6
Orkut	-	-	1.5	0.6	0.09
Twitter	2.3	4	3.1	4.8	4.3

Table 4: Percentage of similarity estimates with errors greater than 0.05; comparison between LSH Approx and LSH + BayesLSH

Parameter value	Fraction errors $>$ 0.05 for varying γ	Mean error for varying δ	Recall for varying ϵ
0.01	0.7%	0.001	98.76%
0.03	2%	0.01	97.79%
0.05	3%	0.017	97.33%
0.07	4.2%	0.022	96.06%
0.09	5.4%	0.027	95.35%

Table 5: The effect of varying the parameters γ, δ, ϵ one at a time, while fixing the other two parameters at 0.05. The dataset is WikiWords100K, with the threshold fixed at t=0.7; the candidate generation algorithm was LSH.

6. CONCLUSIONS AND FUTURE WORK

In this article, we have presented BayesLSH (and a simple variant BayesLSH-Lite), a general candidate verification and similarity estimation algorithm for approximate similarity search, which combines Bayesian inference with LSH in a principled manner and has a number of advantages compared to standard similarity estimation using LSH. BayesLSH enables significant speedups for two state-of-the-art candidate generation algorithms, AllPairs and LSH, across a wide variety of datasets, and furthermore the quality of BayesLSH is easy to tune. As can be seen from Table 2, a BayesLSH variant is typically the fastest algorithm on a variety of datasets and similarity measures.

BayesLSH takes a largely orthogonal direction to a lot of recent research in LSH, which concentrates on more effective indexing strategies, ultimately with the goal of candidate generation, such as Multi-probe LSH [18] and LSB-trees [25]. Furthermore, a lot of research on LSH is concentrated on nearest-neighbor retrieval for distance measures, rather than all pairs similarity search with a similarity threshold t.

There are two promising avenues for future research with BayesLSH. First is to extend BayesLSH for similarity search with learned (kernelized) metrics, since such similarity measures are often superior for complex domains [13]. Secondly, we believe that a BayesLSH-Lite analogue can be developed for candidate pruning in the case of nearest neighbor retrieval for Euclidean distances (although the final distance may have to be calculated exactly).

Acknowledgments: We thank Luis Rademacher and anonymous reviewers for helpful comments, and Roberto Bayardo for clarifications on the AllPairs implementation. This work is supported in part by the following NSF grants: IIS-1141828 and IIS-0917070.

7. REFERENCES

[1] S. Agarwal, N. Snavely, I. Simon, S. Seitz, and R. Szeliski. Building rome in a day. In *ICCV*, pages 72–79, 2009.
[2] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51:117–122, 2008.
[3] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.
[4] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC '98*, pages 327–336, New York, NY, USA, 1998. ACM.
[5] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *WWW*, 1997.
[6] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC '02*, 2002.
[7] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SOCG*, pages 253–262. ACM, 2004.
[8] A. R. Didonato and A. H. Morris, Jr. Algorithm 708: Significant digit computation of the incomplete beta function ratios. *ACM Trans. Math. Softw.*, 18, 1992.
[9] T. Elsayed, J. Lin, and D. Metzler. When close enough is good enough: Approximate positional indexes for efficient ranked retrieval. In *CIKM*, 2011.
[10] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.
[11] M. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, 2006.
[12] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, 1998.
[13] P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. In *IEEE CVPR*, 2008.
[14] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.
[15] D. Lewis, Y. Yang, T. Rose, and F. Li. Rcv1: A new benchmark collection for text categorization research. *JMLR*, 5:361–397, 2004.
[16] P. Li and C. König. b-bit minwise hashing. In *WWW*, 2010.
[17] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci. Technol.*, 58:1019–1031, May 2007.
[18] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.
[19] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.
[20] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *IMC*, 2007.
[21] D. Ravichandran, P. Pantel, and E. Hovy. Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In *ACL*, 2005.
[22] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. arXiv:1110.1328v2 [cs.DB].
[23] V. Satuluri and S. Parthasarathy. Symmetrizations for clustering directed graphs. In *EDBT*, 2011.
[24] V. Satuluri, S. Parthasarathy, and Y. Ruan. Local graph sparsification for scalable clustering. In *SIGMOD*, 2011.
[25] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, 2009.
[26] C. Xiao, W. Wang, X. Lin, and J. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, 2008.
[27] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near duplicate detection. *ACM Transactions on Database systems*, 2011.
[28] J. Zhai, Y. Lou, and J. Gehrke. Atlas: a probabilistic algorithm for high dimensional similarity search. In *SIGMOD*, 2011.
[29] X. Zhu and A. Goldberg. Introduction to semi-supervised learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 3(1):1–130, 2009.