A Bayesian Perspective on Locality Sensitive Hashing with Extensions for Kernel Methods

Aniket Chakrabarti, The Ohio State University
Venu Satuluri, Twitter Inc.
Atreya Srivathsan, Amazon Inc.
Srinivasan Parthasarathy, The Ohio State University

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. In order to reduce the number of candidates to search, Locality-sensitive hashing (LSH) based indexing methods are very effective. 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.

We also extend the BayesLSH algorithm for kernel methods – where the similarity between two data objects is defined by a kernel function. Since the embedding of data points in the transformed kernel space is unknown, algorithms such as AllPairs which rely on building inverted index structure for fast similarity search do not work with kernel functions. Exhaustive search across all possible pairs is also not an option since the data set can be huge and computing the kernel values for each pair can be prohibitive. We propose **K-BayesLSH** an all pairs similarity search problem for kernel functions. K-BayesLSH leverages a recently proposed idea – *kernelized locality sensitive hashing*(KLSH)- for hash bit computation and candidate generation, and uses the aforementioned BayesLSH idea for candidate pruning and similarity estimation. We ran a broad spectrum of experiments on a variety of datasets drawn from different domains and with distinct kernels and find a speedup of 2x-7x over vanilla KLSH.

CCS Concepts: • **Information systems** → **Probabilistic retrieval models;**

Additional Key Words and Phrases: Locality Sensitive Hashing, All Pairs Similarity Search, Bayesian Inference, Kernel Similarity Measure

ACM Reference Format:
Aniket Chakrabarti, Venu Satuluri, Atreya Srivathsan, and Srinivasan Parthasarathy, 2014. A Bayesian Perspective on Locality Sensitive Hashing with Extensions for Kernel Methods. *ACM Trans. Embedd. Comput. Syst.* V, N, Article A (January 2015), 31 pages.
DOI: http://dx.doi.org/10.1145/2778990

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

---

The authors consider Venu Satuluri and Aniket Chakrabarti as joint first authors of this work.

DOI: http://dx.doi.org/10.1145/2778990

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 [Ravichandran et al. 2005], semi-supervised learning [Zhu and Goldberg 2009], information retrieval (including text, audio and video), query refinement [Bayardo et al. 2007], near-duplicate detection [Xiao et al. 2011], collaborative filtering, link prediction for graphs [Liben-Nowell and Kleinberg 2007], and 3-D scene reconstruction [Agarwal et al. 2009] 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 [Broder et al. 1997; Ravichandran et al. 2005], information retrieval [Elsayed et al. 2011], near-duplicate detection for web crawling [Manku et al. 2007; Henzinger 2006] and graph clustering [Satuluri et al. 2011].

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 [Datar et al. 2004]). An important breakthrough was the invention of locality-sensitive hashing [Indyk and Motwani 1998; Gionis et al. 1999], 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 [Andoni and Indyk 2008].

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 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 [Satuluri and Parthasarathy 2012], called **BayesLSH** and **BayesLSHLite**, where the former performs both candidate pruning and similarity estimation, while the latter only performs candidate pruning and computes the similarities of remaining candidates exactly. In this work we additionally extend the BayesLSH [Satuluri and Parthasarathy 2012] algorithm to support arbitrary kernel similarity measures. We use the *kernelized locality hashing* [Kulis and Grauman 2012] technique to generate the LSH sketches in the kernel induced feature

space. 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 [Bayardo et al. 2007] 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; accuracy parameters ϵ $(False\ Omission\ Rate), \delta$ $(interval)$ and γ $(coverage)$; 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 less than ϵ probability of being a true positive is pruned away from 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 parameterization of BayesLSH is intuitive - the desired recall can be controlled using ϵ (though ϵ is not recall, but it can control the recall as it is the false omission rate, and lesser the value of ϵ, higher the recall will be), 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.
(2) The BayesLSH technique can be adapted to kernel spaces (called K-BayesLSH) as well. This is done by leveraging the hyperplane rounding algorithm attributed to Charikar[Charikar 2002] and the sketch generation algorithm from arbitrary kernels developed by Kulis[Kulis and Grauman 2012].
(3) There are no restricting assumptions about the specific form of the candidate generation algorithm; BayesLSH *complements* progress in candidate generation algorithms.
(4) 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.
(5) 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 [Xiao et al. 2011; Zhai et al. 2011].
(6) 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 Bayes-LSH and BayesLSH-Lite with two different candidate generation algorithms AllPairs [Bayardo et al. 2007] and LSH, and find significant speedups, typically in the range 2x-20x over baseline approaches (see Table II). 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 approxima-

tion 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.

In many application domains, such as image search, similarity measures such as Jaccard and Cosine does not work well. In such domains, ranging from bioinformatics to image processing, specialized kernel functions (Gaussian RBF Kernel for image data) seem to provide quantifiable improvement in search quality over traditional measures. We use K-BayesLSH for the cosine similarity in a kernel induced feature space. In kernel space, the explicit representation of the data objects are not known, as a result the candidate generation algorithms such as AllPairs[Bayardo et al. 2007] relying on the vector forms of the objects does not work. Furthermore, since kernel computations can be very expensive, pairwise kernel value calculation of all points is infeasible in a large enough data set. Only a sub-sample of the kernel matrix is available at runtime, as a result exact similarity for any random pair is not computable at runtime, rendering the BayesLSH-Lite technique ineffective as it relies on exact similarity computation.

We evaluated our K-BayesLSH algorithm for kernels on 5 different data sets drawn from 3 different domains, with different kernel methods suitable for each application domain. We find that K-BayesLSH provides a speedup of 2x-7x over vanilla KLSH while providing similar quality results to an exhaustive (brute-force) search technique of the candidate space.

2. BACKGROUND

Following Charikar [Charikar 2002], 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 $\mathbf{x}, \mathbf{y}$,

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

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 [Charikar 2002], 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 [Datar et al. 2004; Andoni and Indyk 2008], including the original LSH paper [Indyk and Motwani 1998], which is geared towards *distance* measures.

Locality-sensitive hashing schemes have been proposed for a variety of similarity functions thus far, including Jaccard similarity [Broder et al. 1998; Li and König 2010], Cosine similarity [Charikar 2002] and kernelized similarity functions (representing e.g. a learned similarity metric) [Jain et al. 2008; Kulis and Grauman 2012].

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 [Indyk and Motwani 1998; Datar et al. 2004]. The general method works as follows [Indyk and Motwani 1998; Datar et al. 2004; Broder et al. 1997; Ravichandran et al. 2005; Henzinger 2006].

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 sig-

nature 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 $l = \lceil \frac{\log \epsilon}{\log(1-t^k)} \rceil$ [Xiao et al. 2011].

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 [Rice 2007]:

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

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 [Ravichandran et al. 2005; Charikar 2002]. 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 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.

Since we don't know the true similarity of each pair apriori, 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.

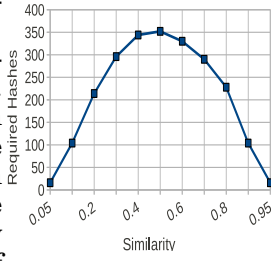


Fig. 1. Hashes vs. similarity

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 for the prior probability distribution over the parameters, but a reasonable amount of data generally "swamps out" the influence of the prior (see section 8.3). Furthermore, a good prior can lead to *improved* estimates over maximum likelihood estimation - this is a common strategy for avoiding over fitting 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.

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

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: [1]

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

By plugging in the expressions for $p(M(m, n) \mid S)$ 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?

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

(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

[1] 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.

of $\hat{S}$ i.e. probability that this estimate is within δ of the true similarity?

$$Pr[|S - \hat{S}| < \delta \mid M(m,n)] = Pr[\hat{S} - \delta < S < \hat{S} + \delta \mid M(m,n)]$$
$$= \int_{\hat{S}-\delta}^{\hat{S}+\delta} p(s \mid M(m,n))ds \quad (5)$$

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 [Jain et al. 2008]. 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.

LSH family: The LSH family for Jaccard similarity is the family of minwise independent permutations [Broder et al. 1998] 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

ALGORITHM 1: BayesLSH

Input: Set of candidate pairs C; Similarity threshold t; recall parameter ϵ; accuracy parameters δ, γ
Output: Set O of pairs (x, y) along with similarity estimates $\hat{S}_{x,y}$
$O \leftarrow \emptyset$
for all $(x, y) \in C$ **do**
    $n, m \leftarrow 0$ {Initialization}
    $isPruned \leftarrow$ False
    while True **do**
        $m = m + \sum_{i=n}^{n+k} I[h_i(x) == h_i(y)]$ {Compare hashes n to $n + k$}
        $n = n + k$
        if $Pr[S \geq t \,|\, M(m, n)] < \epsilon$ **then**
            $isPruned \leftarrow$ True
            break {Prune candidate pair}
        end if
        $\hat{S} \leftarrow \arg\max_s p(s|M(m, n))$
        if $Pr[|S - \hat{S} \,|\, M(m, n) < \delta] < \gamma$ **then**
            break {Similarity estimate is sufficiently concentrated}
        end if
    end while
    if $isPruned$ == False **then**
        $O \leftarrow O \cup \{((x, y), \hat{S})\}$
    end if
end for
return O

ALGORITHM 2: BayesLSH-Lite

Input: Set of candidate pairs C; Similarity threshold t; recall parameter ϵ; Number of hashes to use h
Output: Set O of pairs (x, y) along with exact similarities $\hat{S}_{x,y}$
$O \leftarrow \emptyset$
for all $(x, y) \in C$ **do**
    $n, m \leftarrow 0$ {Initialization}
    $isPruned \leftarrow$ False
    while $n < h$ **do**
        $m = m + \sum_{i=n}^{n+k} I[h_i(x) == h_i(y)]$ {Compare hashes n to $n + k$}
        $n = n + k$
        if $Pr[S \geq t \,|\, M(m, n)] < \epsilon$ **then**
            $isPruned \leftarrow$ True
            break {Prune candidate pair}
        end if
    end while
    if $isPruned$ == False **then**
        $s_{x,y} = similarity(x, y)$ {Exact similarity}
        if $s_{x,y} > t$ **then**
            $O \leftarrow O \cup \{((x, y), s_{x,y})\}$
        end if
    end if
end for
return O

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 [Didonato and Morris 1992].

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 [Charikar 2002]. 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.[2] Our solution is to use a simple uniform distribution on $[0.5, 1]$ as the prior for t. The rational behind doing this is since it is hard to choose a correct conjugate prior, we choose a uninformative prior instead, and let posterior be learned solely from data. This may introduce some approximation in the estimates and exact correctness of the posterior is hard to argue, but with sufficiently high number of hashes, even when the true prior 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 distribution gets peaky(concentrated) around the true similarity value with sufficiently high n, which is the case for us(see section 8.3). Similar Bayesian confidence bounds under the Beta-binomial model and uniform-binomial model is already studied for the bandits problem [Kaufmann et al. 2012].

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

[2]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.

greater than the threshold t is:

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

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.

5. BAYESLSH FOR ARBITRARY KERNEL FUNCTIONS

Now we discuss on how the BayesLSH algorithm can be extended for arbitrary normalized kernel functions. Since a kernel function gives us the dot product in some feature space, a normalized kernel always gives the cosine similarity of the points in the same kernel induced feature space. Let the embedding of the data points in that space be ϕ. The dot product of points X and Y is given by:

$$\begin{aligned}
cos(\theta) &= \frac{< \phi(X), \phi(Y) >}{\sqrt{< \phi(X), \phi(X) >}\sqrt{< \phi(Y), \phi(Y) >}} \\
&= \frac{K(X,Y)}{\sqrt{K(X,X)K(Y,Y)}}
\end{aligned}$$

Therefore, we can use the same principles that guided the development of BayesLSH for cosine similarity, in case of kernels.

5.1. LSH for kernels

We directly use the method proposed by Kulis[Kulis and Grauman 2012] to compute the hash bits in a kernel induced feature space. According to Charikar's hyperplane rounding algorithm, for each hash function, a random multi-variate Gaussian distributed vector with zero mean and identity covariance is required. As a result in kernelized spaces, such a vector has to be approximated from a sub-sample of the kernel matrix entries using the whitening transform. To compute the covariance matrix as required by the whitening transform, the sub-sample of the kernel matrix is projected to its eigen vectors. According to kernelPCA[Schölkopf et al. 1998], eigen decomposition of both the covariance matrix and kernel matrix yields the same non-zero eigen values and the eigen vectors of the covariance matrix can be computed from the eigen vectors of the kernel matrix.

If the data set contains N points, and as input we have the kernel values of each point to p other points in the data set, then the algorithm to compute the hash bits is as follows:

(*1*) *From the $N \times p$ input kernel matrix, form the $p \times p$ kernel matrix and call it K.*
(*2*) *Center the matrix **K**.*
(*3*) *Compute a hash function h by forming a binary vector e by selecting q indices at random from $1, ..., p$, then form $w = K^{-1/2}e$ and assign bits according to the hash function*

$$h(\phi(x)) = sign(\sum_i w(i)k(x_i, x))$$

5.2. Inference

The inference procedure can be done in exactly the same manner as discussed in the previous section - BayesLSH for cosine similarity. This is because, once the hash functions are computed, the inference of Bayes-LSH only depends on the hash keys, i.e., the two inferences of equations 3 and 5 does not depend on the kernel functions anymore. We can thus directly adapt and plug in the aforementioned inference procedure.

6. DISCUSSION

1. We note that the kernelized version of the Bayes-LSH algorithm is more constrained than the ordinary one. While it can use the LSH candidate generation technique, other algorithms such as Allpairs[Bayardo et al. 2007], PPJoin[Xiao et al. 2011] will not work with K-BayesLSH. The BayesLSH-Lite variant is also not feasible for kernel spaces. The reason is, these algorithms build smart index structures which rely on the explicit vector representation of the data points. Since this is not available in kernel spaces, algorithms such as AllPairs are not an option. In fact, for a sufficiently large data set, all we have are the kernel values of each point to p other points in the data base. Computing a full kernel matrix over all points is not feasible both in terms of time and space. Hence the exact kernel similarity cannot be computed for all pairs of points making the BayesLSH-Lite option impractical for kernel methods.
2. A subtle point to note here is that n, in the likelihood function of equation 2,should ideally be modeled as a random variable. The reason is that the hash comparison process is adaptive and n (no. of hashes) is determined by when the pair gets pruned or concentrated. Therefore, if one is to model the likelihood function precisely, then n has to be modeled as a random variable as the stochastic process in question is sequential in nature. We note that treating n as fixed in this work is intentional as this has substantial performance benefits over precisely modeling a sequential stochastic process. The resulting loss in quality is negligible and is discussed elsewhere [Chakrabarti and Parthasarathy 2015].

7. 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 $Pr[S \geq t|M(m,n)] > 1-\epsilon$ 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. Additionally, we only check the inference probabilities after comparing a batch of hashes, rather than after comparing individual hash bits. This effectively minimizes the overhead of online inference computations. In fact in the majority of our runs, we have seen that less 2% of the total execution time is used for inference computations.

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 $(-8,8)$.

Precomputing the weight matrix for kernels: This optimization is specific to the kernelized variant of Bayes-LSH. We used a similar trick used in [Kulis and Grauman 2012] to optimize the runtime. The tradeoff between runtime and memory requirement for this optimization is analyzed below. For cosine similarity, each hash function requires a random vector. When only a subset of kernel matrix is known about the data, each random vector has to be approximated by selecting q random indices from $1,..,p$ as specified in the KLSH algorithm in section 4.3. So storage requirement for the hash functions is very low. For b hash functions, we needed to store $b \times q$ two byte integers. But the problem with this approach is, every time a hash bit for a point is computed, the weight vector W(whose dimensionality is p) is computed first from the stored q indices for that hash, and then from the weight vector the hash bit is computed. In other words every hash bit computation involves a matrix vector product and a vector vector scalar product. The matrix vector product involves $p \times p$ matrix multiplied by $p \times 1$ vector. The scalar product is among two p dimensional vectors. An

alternative implementation choice is, instead of storing vectors of q indices for each hash, precompute the weight vectors from the q indices using the matrix vector product and store one vector per hash function. Of course the storage requirement will increase. For b hash functions, the storage requirement will be $b \times q$ eight byte doubles. But the hash bit computation will be much faster. For each point, for each hash bit, only a scalar product of two p dimensional vectors is required. By precomputing the weight matrix we are saving the matrix vector product cost from the hash computation cost. This improves the run time by orders of magnitude.

Making the algorithm I/O aware: For very large datasets, the number of similar pairs could potentially become $N \times N$ if the similarity threshold is set very low. Building this adjacency list in-memory is not feasible in such cases. This is why we made our similar pairs generation process I/O aware. Our algorithm requires only one row of the adjacency list to be in-memory. Of course if more memory bandwidth is available, multiple rows can be processed together and written to/read from a file. We applied the same principles while reading the available subset($N \times p$) of the kernel matrix. This whole matrix need not be loaded in memory. As long as a single row can be stored in-memory, our algorithm works. Obviously making it I/O aware resulted in a slow down due to the disk IO.

8. EXPERIMENTS

8.1. Non-kernel similarity measures

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

— **RCV1** is a text corpus of Reuters articles and is a popular benchmarking corpus for text-categorization research [Lewis et al. 2004]. 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[3] - 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 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 **WikiWords-500K** 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 [Mislove et al. 2007]. 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. [Kwak et al. 2010]. 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 corpora can be useful for clustering, semi-super-vised 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

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

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

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

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 [Satuluri and Parthasarathy 2011] for another example from graph mining).
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.

8.1.1. Experimental setup. We compare the following methods for all-pairs similarity search.
1. AllPairs [Bayardo et al. 2007] (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+ [Xiao et al. 2011] 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.[4] 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.

8.1.2. 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 II compares the fastest BayesLSH variant with all the baselines. The quality of the output of BayesLSH can be seen in Table III where we show the recall rates for AP+BayesLSH and AP+BayesLSH-Lite, and in Table IV 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 II 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 III), and similarity estimates are accurate, with usually no more than 5% output pairs with error above 0.05 (see Table IV).

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 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.

[4]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.

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 IV. 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.

8.1.3. Effect of varying parameters of BayesLSH. We next examine the effect of varying the parameters of BayesLSH - namely the accuracy parameters γ, δ and ϵ. We vary each parameter from 0.01 to 0.09 in increments of 0.02, while fixing the other two

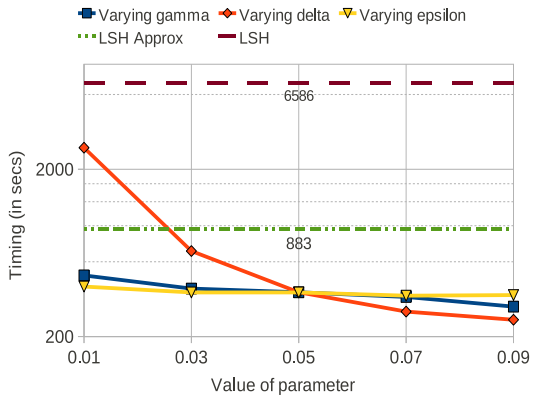


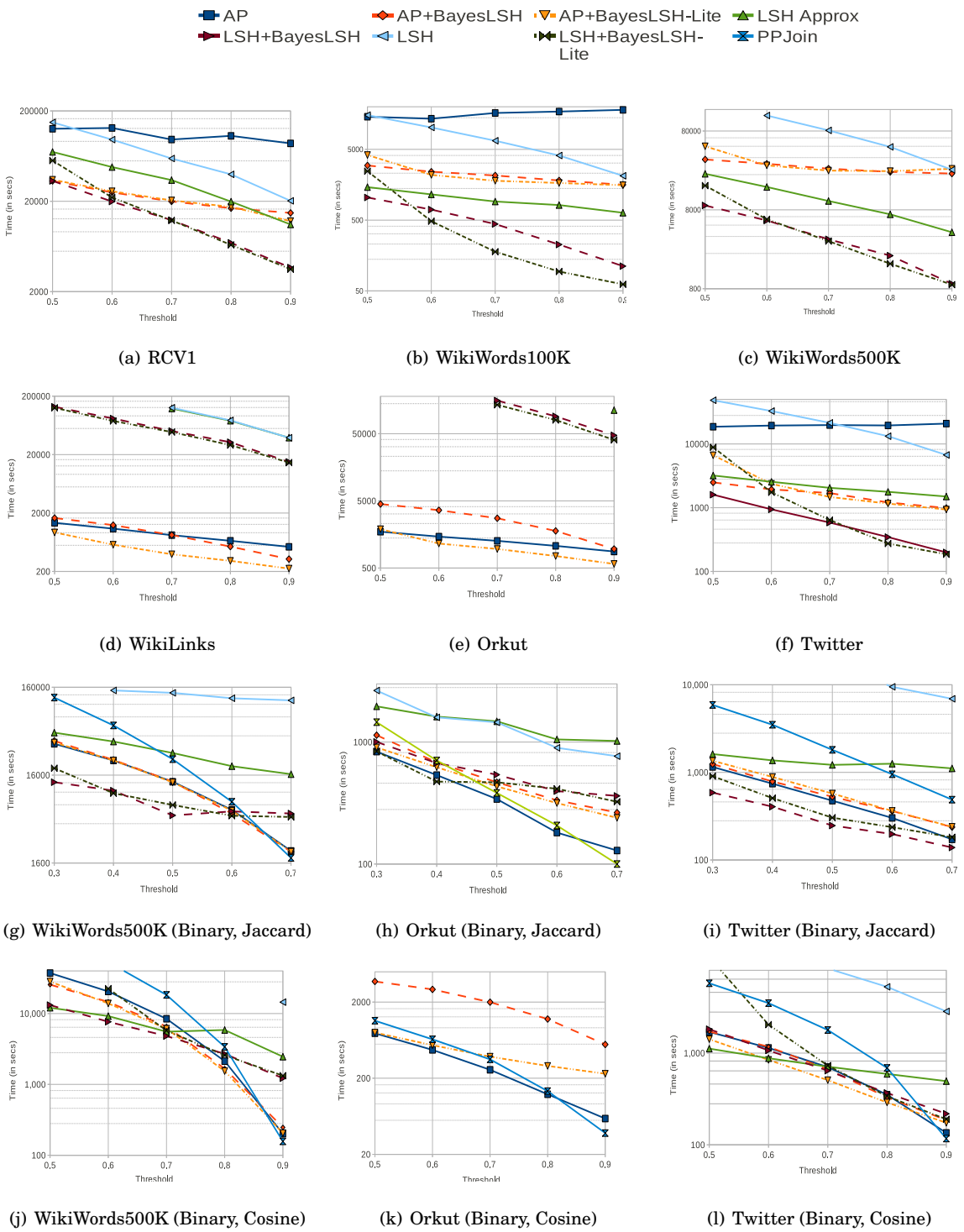
Fig. 2. Effect of varying γ, δ, ϵ separately on the running time of LSH+BayesLSH. The dataset is WikiWords100K, with the threshold fixed at t=0.7 for cosine similarity. For comparison, the times for LSH Approx and LSH are also shown.

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 ϵ and 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 V, 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 parameter ϵ, we find that the recall reduces with higher values of ϵ as expected.

8.2. Kernel similarity measures

8.2.1. Experimental setup. We experimented with a variety of data sets from different domains while using specialized kernels for each domain. The **K-BayesLSH** method uses the *kernelized locality sensitive hashing* algorithm to compute the hashes for cosine similarity, LSH index structure for candidate generation and BayesLSH for candidate pruning and similarity estimation. In **KLSH** instead of BayesLSH, a standard *maximum likelihood estimator* is used for similarity estimation and of course there is no candidate pruning. We implemented our *kernelized Bayesian locality sensitive hashing* technique as a single threaded C++ application. We ran all of our experiments



(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)

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

Table II. 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 $\geq$).

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

on the Ohio super computing center's Oakley cluster consisting of HP SL390 G7 nodes with Intel Xeon. Each node consists of 12 cores and up to 48GB of memory. We used one node for each of our experiments.

We set the following parameters for all experiments. For the KLSH algorithm we set the sampling parameters $p = 1000$ and $q = 200$. We set the BayesLSH accuracy parameters as $\epsilon = 0.03, \gamma = 0.05$ and $\delta = 0.07$ for the **K-BayesLSH** experiments. For **KLSH approx** we set the number of hashes to be 2048 and expected recall as 0.97. For performance evaluation, we ran each method for the cosine similarity while varying the similarity threshold from $0.9 - 0.5$. We set the total time limit to be 50 hours.

8.2.2. Datasets and kernels. We used 5 different datasets:

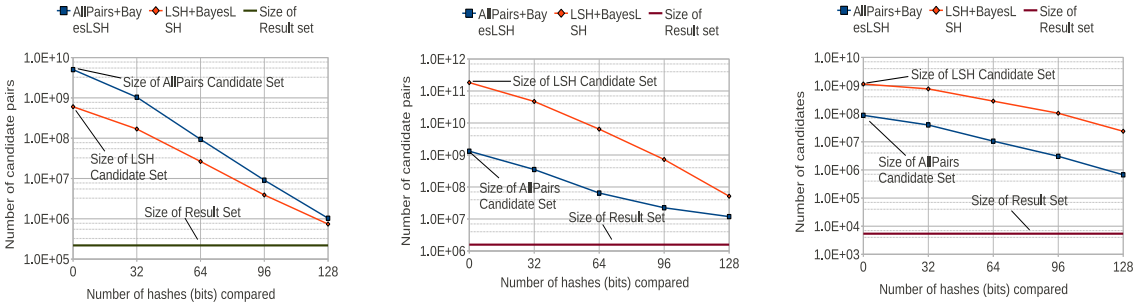
(1) **Caltech 101:** We use 3K images from the Caltech 101 data set which is a popular object recognition benchmark[Fei-Fei et al. 2004]. The data set contains 101

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

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 IV. Percentage of similarity estimates with errors greater than 0.05; comparison between LSH Approx and LSH + BayesLSH.

	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



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

Fig. 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.

Table V. 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.

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%

image categories. We use the correspondence based local feature kernel(CORR) as proposed in [Zhang et al. 2006].

(2) **PASCAL VOC 2007:** We use the PASCAL VOC 2007 training-validation data set[Everingham et al.]. The data set contains 5K images and 20 image categories. This is also a object recognition benchmark. We employ the χ^2 kernel built on top of a bag-of-words based feature vectors as described in [Chatfield et al. 2011].

(3) **SCOP protein sequences:** We use the SCOP classification of protein sequences from the Protein Data Bank(PDB) version 1.75[Murzin et al. 1995; Chandonia et al. 2004]. We downloaded 10K sequences such that no two sequence have over 95% identity. According to SCOP the protein sequences are classified as - each class of protein has a number of folds, each fold is subdivided into super families and each super family further consists of a number of families. There are a total of 1195 folds, 1962 super families and 3902 families. We used the local alignment kernel as described in [Vert et al. 2004; Saigo et al. 2004].

(4) **Local image patches:** We use the local patch data set from the photo tourism project[Winder et al. 2009; Snavely et al. 2008; Goesele et al. 2007]. The idea is, each image in a data set is broken down into a number of local patches and those patches are stored. We used feature vectors built in [Simonyan et al. 2012] as descriptors for the local patches. We use the Notre Dame Cathedral data set which consists of around $460K$ local image patches. We employed the Gaussian RBF kernel over the feature vectors of the local patches.

(5) **2M SIFT vectors:** To test K-BayesLSH at scale, we downloaded 2 million SIFT vectors from the INRIA Holidays project[Jégou et al. 2008]. We run both K-BayesLSH and KLSH employing the Gaussian RBF kernel over the SIFT vectors.

8.2.3. Tasks. **Performance evaluation:** For each data set, to evaluate the performance, we measure the raw runtime of K-BayesLSH and compare it with the raw runtime of KLSH. We vary the similarity threshold from 0.9 to 0.5 and run both K-BayesLSH and KLSH for each similarity threshold in the range.

Quality evaluation: To evaluate and compare the quality of K-BayesLSH with that of KLSH we use different quality measures which is suitable for different data sets, described below. For the first two tasks knn object classification and near duplicate image patch retrieval, we designed a similar experimental setting as described in [Kulis and Grauman 2012; He et al. 2010]. For the large scale data, we created the ground truth for the query objects by explicitly computing the kernel values for them and reported the accuracy in terms of that ground truth.

It should be noted that, these indirect quality measures such as classification accuracy or image retrieval quality are very common in evaluating kernel hashing algorithms [Kulis and Grauman 2012; He et al. 2010], rather than the more direct measure in terms recall/precision of candidate pairs above certain threshold. The reason is, the

similarity estimates from the hash embeddings get distorted a lot from the original kernel function value, but the relative rank order among data points is somewhat better preserved. The said distortion depends on the actual kernel function and what kinds assumptions the kernel hashing algorithm makes(for eg. positive definiteness of the subsample of kernel matrix, or independence of kernel hash functions). Our algorithm in no way improves the quality of the hash signatures, through the early pruning and concentration probability inferences it provides a very efficient way to prune candidates and estimate similarity once the hash signatures are generated. However for completeness we provide the recall table similar to the one in the non-kernelized experiments.

- Since the first 3 data sets - Caltech 101, PASCAL VOC 2007 and SCOP protein sequences are widely used as object classification benchmarks, we use classification accuracy as the quality metric for them. Ground truth(which class they belong) is available for all these data sets. For each data set we use 1000 training samples and the rest for testing. We compare the results of $knn(k = 1)$ classifier using K-BayesLSH, KLSH and using exhaustive search to find the exact nearest neighbor. As mentioned above we also provide the recall table VI which gives what percent of the true positives(actual candidates above threshold t) are returned by K-BayesLSH and compare it with the KLSH algorithm [Kulis and Grauman 2012].
- We evaluate the quality with the Notre Dame patch data set by recall rate. We ran a near duplicate image patch retrieval experiment - given a query image patch, the problem is to retrieve all duplicate patches from the patch database. We retrieve the top k nearest neighbors and check how many of the duplicate patches were recalled. The value of k was varied from 50 to 500 and the recall rates were studied for K-BayesLSH, KLSH and exhaustive search for top k nearest neighbors. The ground truth of which patches are duplicate were available. We used 1000 randomly selected patches as query objects.
- We evaluated the quality with the data set containing 2M SIFT vectors in an unsupervised setting. Given a query object, we retrieved top k nearest neighbors using K-BayesLSH and KLSH and compared it with the top k nearest neighbors generated by exhaustive search over the kernel matrix. We report the accuracy, i.e. what percentage of the exact neighbors are correctly identified by the approximate algorithms(K-BayesLSH, KLSH). We varied k from 50 to 500 and used a 1000 randomly chosen objects as query objects.

8.2.4. Results

- The speedup in terms of raw runtimes of K-BayesLSH over KLSH varied from $2x$ to $7x$. For the Caltech 101 and the PASCAL VOC 2007 data sets, we get upto $2x$ speedup,for the SCOP protein sequences, we get upto $3x$ speed up, for the Notre Dame image patch database, we get up to $6x$ speedup and for the large scale experiment with 2 million data objects we get upto $7x$ speedup. The results are reported in Figure 8. It is worthwhile to note that the performance improvement of K-BayesLSH could not achieve the figures of the non-kernelized version(upto 20x) because of the reasons discussed in section 5.
- Regarding quality for Caltech 101, PASCAL VOC and SCOP proteins, KLSH and K-BayesLSH provide comparable classification accuracy to the *exact nearest neighbor* classifier using exhaustive search. For Caltech 101 data set, K-BayesLSH reports a classification accuracy of 41% as compared to 44% for exact NN classifier, for PASCAL VOC, K-Bayes-LSH reports a classification accuracy of 47% as compared to 49% for exact NN classifier and for the SCOP protein sequences, K-BayesLSH reports a

Table VI. Recalls (out of 100) of K-BayesLSH and KLSH Approx across different datasets and different similarity thresholds.

Recall	t=0.5	t=0.6	t=0.7	t=0.8	t=0.9
K-BayesLSH					
Caltech 101	69.77	94.84	99.76	100	100
Pascal VOC	98.75	98.04	98.32	99.46	100
SCOP proteins	16.37	15.78	27.74	39.6	44.4
KLSH approx					
Caltech101	64.7	96.11	99.76	100	100
Pascal VOC	98.8	99.33	99.52	99.97	98.58
SCOP proteins	14.01	15.02	26.5	33.77	32.01

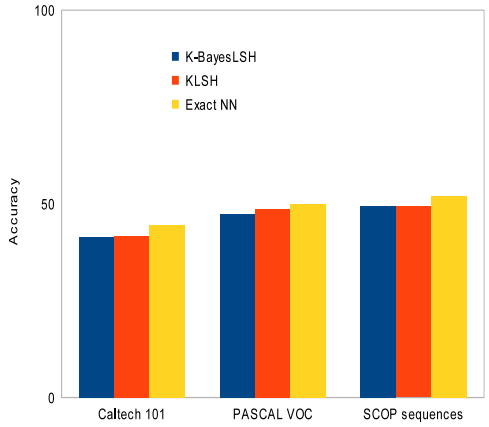


Fig. 5. Classification accuracy of the different algorithms

classification accuracy of 49% as compared to 51% for exact NN classifier. The results are reported in Figure 5.

- In case of the image patch data set, we got comparable recall figures for the exhaustive search method and K-BayesLSH method. The recall of K-BayesLSH varied from 58% to 82% and that of exhaustive search varied from 60% to 84%. The results are present in Figure6.
- For the large scale experiment with 2M SIFT vectors, K-BayesLSH has a top-k neighbor finding accuracy of upto 51% compared to the 52% accuracy of KLSH. The qualitative results are provided in Figure 7. Essentially the difference in quality is negligible while the speedup is significant as noted above (7-fold improvement).

8.3. The Influence of Prior vs. Data

In this section, we show how the observed outcomes (i.e. hashes) are much more influential in determining the posterior distribution than the prior itself. Even if we start with very different prior distributions, the posterior distributions typically become very similar after observing a surprisingly small number of outcomes.

Consider the similarity measure we worked with in the case of cosine similarity, $r(x, y) = 1 - \frac{\theta(x,y)}{\pi}$, which ranges from [0.5,1] - note that $r(x, y) = 0.5$ corresponds to an actual cosine similarity of 0 between x, y. Consider three very different prior

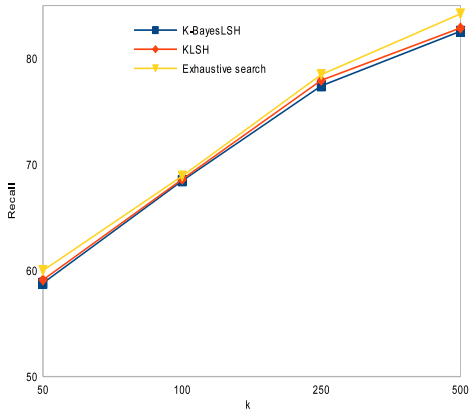


Fig. 6. Recall rates of the different algorithms on the Notre Dame patches

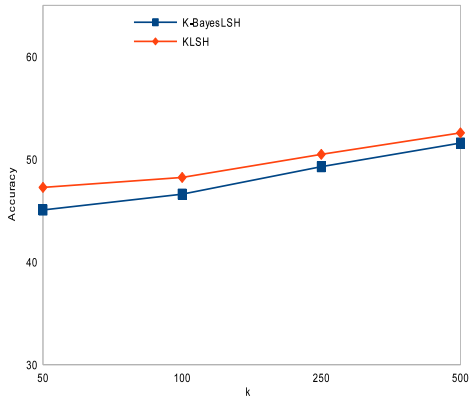


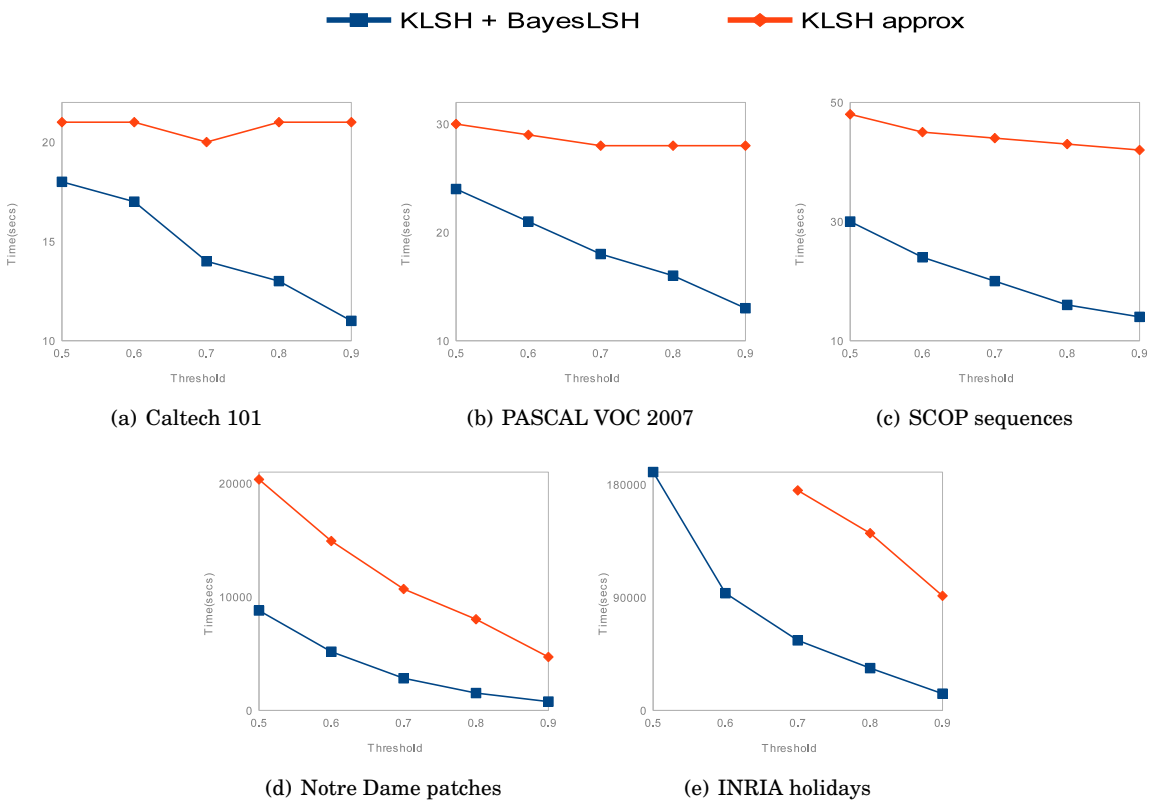
Fig. 7. Accuracy(percentage) of identifying top k neighbors on INRIA holidays data set

distributions for this similarity measure, as follows (the normalization constants have been omitted):

— Negatively sloped power law prior: $p(s) \propto x^{-3}$
— Uniform prior: $p(s) \propto 1$
— Positively sloped power law prior: $p(s) \propto x^3$

In Figure 9, we show the posteriors for each of these three priors after observing a hypothetical series of outcomes for a pair of points x, y with cosine similarity 0.70, corresponding to $r(x, y) = 0.75$. Although to start off with, the three priors are very different (see Figure 9(a)), the posteriors are already quite close after observing only 32 hashes and 24 agreements (Figure 9(b)), and the posteriors get closer quickly with increasing number of hashes(Figures 9(c) and 9(d)).

In general, the likelihood term - which, after observing n hashes with m agreements, is $s^m(1-s)^{(n-m)}$ - is much more sharply concentrated than a justifiable prior, very quickly as we increase n. In other words, a prior would itself have to be very sharply concentrated for it to match the influence of the likelihood - using sharply concentrated priors however brings the danger of not letting the data speak for themselves.



(a) Caltech 101 (b) PASCAL VOC 2007 (c) SCOP sequences

(d) Notre Dame patches (e) INRIA holidays

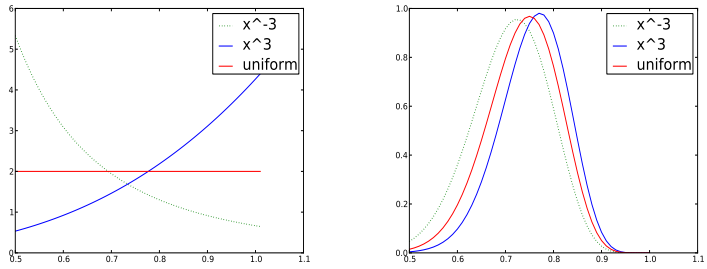
Fig. 8. Timing comparisons of the algorithms on all datasets. Missing points are due to the respective algorithm not finishing within the allotted time (50 hours).

9. CONCLUSIONS AND FUTURE WORKS

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. We have also the extended the BayesLSH algorithm in cases where the similarity in question is given by a specialized kernel function rather than the standard similarity measures. Such algorithms have found wide-spread use within the machine learning community. We call the kernelized variant the K-BayesLSH algorithm.

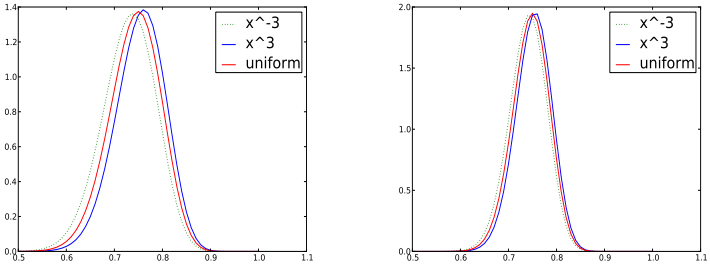
BayesLSH and its variants (BayesLSH-Lite, K-BayesLSH) typically enables significant speedups for two state-of-the-art candidate generation algorithms, AllPairs(only for non-kernels) and LSH, across a wide variety of datasets, and furthermore the quality of BayesLSH is easy to tune. As can be seen from Table II, a BayesLSH variant is typically the fastest algorithm on a variety of datasets and similarity measures. Notably this improvement is also sustained in the kernelized context.

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 [Lv et al. 2007] and LSB-trees [Tao et al. 2009]. Furthermore, a lot of research on LSH is concentrated on nearest-neighbor



(a) Prior distributions

(b) Posterior after 32 hashes, with 24 agreements



(c) Posterior after 64 hashes, with 48 agreements

(d) Posterior after 128 hashes, with 96 agreements

Fig. 9. Even very different prior distributions converge to very similar posteriors after examining a small number of outcomes (hashes)

retrieval for distance measures, rather than the all pairs similarity search with a similarity threshold t (the focus of this study).

We also believe our principled Bayesian framework can be easily extended to other kernel hashing algorithms and hashing algorithms for the Euclidean distance. The requirement is that the hashing algorithm has to be locality sensitive. Each data point has to be projected into a hash code such that the similarity(1-normalized distance) in the original space is proportional to the number of matching hashbits in the embedded space. The intuition is, BayesLSH is a principled framework for estimating a similarity which is given by the fraction of matching hashbits, where the collision probability for each hash bit is same. In fact such algorithms do exist for Euclidean distance [Andoni and Indyk 2008]. There are other kernel hashing algorithms as well which produces locality sensitive hash codes in Hamming space [Joly and Buisson 2011]. So the idea is, we expect to provide quality guarantees of those algorithms while being much faster due to aggressive pruning and fast similarity estimation. Pertaining to the kernelized version of the algorithms, there is an alternate approach one could take. Instead of applying KLSH directly, the approximate explicit embedding of the data objects in the kernel induced feature space can be computed using KPCA or Nystrom method and then LSH can be explicitly applied to it. We believe that such approaches can benefit from the performance aspect by using our Bayes variants.

Acknowledgments: We thank Luis Rademacher and the anonymous reviewers for helpful comments, Roberto Bayardo for clarifications on the AllPairs implementation,

and Brian Kulis for clarifications on the Kernel LSH implementation. This work is supported in part by the following NSF grants: SHF-1217353 and DMS-1418265.

REFERENCES

S. Agarwal, N. Snavely, I. Simon, S.M. Seitz, and R. Szeliski. 2009. Building rome in a day. In *ICCV*. 72–79.

Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* 51 (2008), 117–122.

R.J. Bayardo, Y. Ma, and R. Srikant. 2007. Scaling up all pairs similarity search. In *WWW*.

Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. 1998. Min-wise independent permutations (extended abstract). In *STOC '98*. ACM, New York, NY, USA, 327–336. DOI:http://dx.doi.org/10.1145/276698.276781

Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. 1997. Syntactic clustering of the Web. In *WWW*. 10. http://portal.acm.org/citation.cfm?id=283554.283370

Aniket Chakrabarti and Srinivasan Parthasarathy. 2015. Sequential Hypothesis Tests for Adaptive Locality Sensitive Hashing. In *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 162–172.

John-Marc Chandonia, Gary Hon, Nigel S Walker, Loredana Lo Conte, Patrice Koehl, Michael Levitt, and Steven E Brenner. 2004. The ASTRAL compendium in 2004. *Nucleic acids research* 32, suppl 1 (2004), D189–D192.

Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *STOC '02*. 9. DOI:http://dx.doi.org/10.1145/509907.509965

K. Chatfield, V. Lempitsky, A. Vedaldi, and A. Zisserman. 2011. The devil is in the details: an evaluation of recent feature encoding methods. In *British Machine Vision Conference*.

M. Datar, N. Immorlica, P. Indyk, and V.S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SOCG*. ACM, 253–262.

Armido R. Didonato and Alfred H. Morris, Jr. 1992. Algorithm 708: Significant digit computation of the incomplete beta function ratios. *ACM Trans. Math. Softw.* 18 (1992). Issue 3.

Tamer Elsayed, Jimmy Lin, and Don Metzler. 2011. When Close Enough Is Good Enough: Approximate Positional Indexes for Efficient Ranked Retrieval. In *CIKM*.

M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html. (????).

Li Fei-Fei, Rob Fergus, and Pietro Perona. 2004. Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories. In *Computer Vision and Pattern Recognition Workshop, 2004. CVPRW'04. Conference on*. IEEE, 178–178.

A. Gionis, P. Indyk, and R. Motwani. 1999. Similarity search in high dimensions via hashing. In *VLDB*.

Michael Goesele, Noah Snavely, Brian Curless, Hugues Hoppe, and Steven M Seitz. 2007. Multi-view stereo for community photo collections. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*. IEEE, 1–8.

Junfeng He, Wei Liu, and Shih-Fu Chang. 2010. Scalable similarity search with optimized kernel hashing. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1129–1138.

Monika Henzinger. 2006. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*. 8. DOI:http://dx.doi.org/10.1145/1148170.1148222

Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*. 10. DOI:http://dx.doi.org/10.1145/276698.276876

P. Jain, B. Kulis, and K. Grauman. 2008. Fast image search for learned metrics. In *IEEE CVPR*.

Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2008. Hamming embedding and weak geometric consistency for large scale image search. In *European Conference on Computer Vision (LNCS)*, Andrew Zisserman David Forsyth, Philip Torr (Ed.), Vol. I. Springer, 304–317. http://lear.inrialpes.fr/pubs/2008/JDS08

Alexis Joly and Olivier Buisson. 2011. Random maximum margin hashing. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE, 873–880.

Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. 2012. On Bayesian upper confidence bounds for bandit problems. In *International Conference on Artificial Intelligence and Statistics*. 592–600.

Brian Kulis and Kristen Grauman. 2012. Kernelized locality-sensitive hashing. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 34, 6 (2012), 1092–1104.

H. Kwak, C. Lee, H. Park, and S. Moon. 2010. What is Twitter, a social network or a news media?. In *WWW*.

D.D. Lewis, Y. Yang, T.G. Rose, and F. Li. 2004. Rcv1: A new benchmark collection for text categorization research. *JMLR* 5 (2004), 361–397.

Ping Li and Christian König. 2010. b-Bit minwise hashing. In *WWW*. 10. DOI:http://dx.doi.org/10.1145/1772690.1772759

David Liben-Nowell and Jon Kleinberg. 2007. The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci. Technol.* 58 (May 2007), 1019–1031. Issue 7. DOI:http://dx.doi.org/10.1002/asi.v58:7

Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*. 950–961. http://dl.acm.org/citation.cfm?id=1325851.1325958

Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting near-duplicates for web crawling. In *WWW*.

Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *IMC*.

Alexey G Murzin, Steven E Brenner, Tim Hubbard, and Cyrus Chothia. 1995. SCOP: a structural classification of proteins database for the investigation of sequences and structures. *Journal of molecular biology* 247, 4 (1995), 536–540.

D. Ravichandran, P. Pantel, and E. Hovy. 2005. Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In *ACL*.

John A Rice. 2007. *Mathematical statistics and data analysis*. Cengage Learning.

Hiroto Saigo, Jean-Philippe Vert, Nobuhisa Ueda, and Tatsuya Akutsu. 2004. Protein homology detection using string alignment kernels. *Bioinformatics* 20, 11 (2004), 1682–1689.

V. Satuluri and S. Parthasarathy. 2011. Symmetrizations for clustering directed graphs. In *EDBT*.

Venu Satuluri and Srinivasan Parthasarathy. 2012. Bayesian locality sensitive hashing for fast similarity search. *Proceedings of the VLDB Endowment* 5, 5 (2012), 430–441.

Venu Satuluri, Srinivasan Parthasarathy, and Yiye Ruan. 2011. Local graph sparsification for scalable clustering. In *SIGMOD*.

Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. 1998. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation* 10, 5 (1998), 1299–1319.

Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2012. Descriptor learning using convex optimisation. In *Computer Vision–ECCV 2012*. Springer, 243–256.

Noah Snavely, Steven M Seitz, and Richard Szeliski. 2008. Modeling the world from internet photo collections. *International Journal of Computer Vision* 80, 2 (2008), 189–210.

Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*.

Jean-Philippe Vert, Hiroto Saigo, and Tatsuya Akutsu. 2004. Local alignment kernels for biological sequences. *Kernel methods in computational biology* (2004), 131–154.

Simon Winder, Gang Hua, and Matthew Brown. 2009. Picking the best daisy. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 178–185.

C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. 2011. Efficient Similarity Joins for Near Duplicate Detection. *ACM Transactions on Database systems* (2011).

Jiaqi Zhai, Yin Lou, and Johannes Gehrke. 2011. ATLAS: a probabilistic algorithm for high dimensional similarity search. In *SIGMOD*.

Hao Zhang, Alexander C Berg, Michael Maire, and Jitendra Malik. 2006. SVM-KNN: Discriminative nearest neighbor classification for visual category recognition. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, Vol. 2. IEEE, 2126–2136.

X. Zhu and A.B. Goldberg. 2009. Introduction to semi-supervised learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 3, 1 (2009), 1–130.