DSH: Data Sensitive Hashing for High-Dimensional k-NN Search

Jinyang Gao[‡] H. V. Jagadish[§] Wei Lu[‡] Beng Chin Ooi[‡]
[‡]School of Computing
National University of Singapore
{jinyang.gao, luwei1, ooibc}@comp.nus.edu.sg
[§]Department of EECS
University of Michigan
jag@umich.edu

ABSTRACT

The need to locate the k-nearest data points with respect to a given query point in a multi- and high-dimensional space is common in many applications. Therefore, it is essential to provide efficient support for such a search. Locality Sensitive Hashing (LSH) has been widely accepted as an effective hash method for high-dimensional similarity search. However, data sets are typically not distributed uniformly over the space, and as a result, the buckets of LSH are unbalanced, causing the performance of LSH to degrade.

In this paper, we propose a new and efficient method called *Data Sensitive Hashing* (DSH) to address this drawback. DSH improves the hashing functions and hashing family, and is orthogonal to most of the recent state-of-the-art approaches which mainly focus on indexing and querying strategies. DSH leverages data distributions and is capable of directly preserving the nearest neighbor relations. We show the theoretical guarantee of DSH, and demonstrate its efficiency experimentally.

Keywords

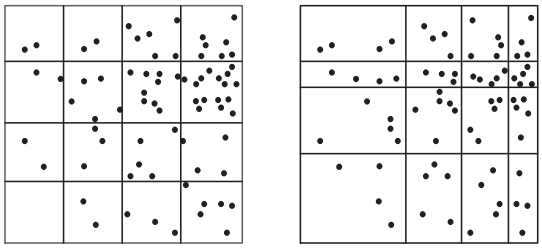
High Dimensions, Similarity Search, Hashing, LSH

1. INTRODUCTION

In a wide range of applications, data can be represented as points in a multi-dimensional space. For example, feature vectors are often used to represent multi-media data such as images and music. Similarly, a company may use a number of attributes for profiling each customer, or for each product. Each attribute, or each element of a feature vector, can be considered as a dimension in a multi-dimensional space in which each object is a point. Similarity search consequently is transformed to finding points in this multi-dimensional space that are close to a given query point. In many applications, we may be interested in data points that are within some distance of a query point. However, we may not have a meaningful way of setting a distance threshold, and instead we seek the k-nearest points (e.g. the results displayed in the first page of search engine). Therefore, it is essential to support efficient k-nearest neighbor (k-NN) search.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2588565.



(a) LSH (b) DSH

Figure 1: Motivation

Access methods in the multi-dimensional space, including the k-NN problem, have been studied extensively. However, most of them suffer from the so-called curse of dimensionality and demonstrate poor performance when the number of dimensions is high [21]. One technique that shows promise in high-dimensional spaces is locality sensitive hashing (LSH) [10]. LSH relaxes the k-NN problem to a *c-approximate k-NN problem* that aims to find k points within distance $c \times R$ where R is the maximum distance between the query point and its k-nearest neighbors. In essence, LSH solves k-NN problem by first obtaining a set of *similar points* to q and then extracting the k-nearest points from these similar points. For this purpose, LSH designs a novel scheme to hash the points so that the possibility of collision is much higher for similar points than for dissimilar points. Similar objects are then identified by examining a certain number of points that collide with the query point. However, defining similar points is challenging, and as such LSH simply adopts a constant value r to define two points as similar if their distance is no greater than r. In essence, LSH searches for near neighbors within a radius r. For k-NN search, LSH performs well in a uniform data distribution setting in which a good-quality r can be derived. When the data are skewed, the distance between k-NN pairs can vary greatly, using a consistent r to define all similar points is inadequate. It is for this reason that LSH performance suffers. Figure 1a shows an example of the hashing results of LSH. On the indexing level, we see that the hashing is unbalanced, where some buckets are empty while a few buckets contain too many points. From the k-NN perspective, as the distance between k-nearest neighbor pairs can vary greatly, LSH solutions either have to maintain a huge set of indexes for different values r, or use one (or a few fixed) r which may

lead to arbitrary bad results. More discussion will be given in Section 2.

Recently, there has been increasing attention focused on designing new indexes [20] or query strategies [13, 9] to alleviate the limitations of LSH. However, to the best of our knowledge, all of them are still based on locality-sensitive hashing families, i.e. random projections, which cannot adapt based on data distributions. Real data distributions are often non-uniform and frequently very skewed. Consequently, these methods still suffer from poor hashing effectiveness. In this paper, instead of trying to enhance LSH with the use of more efficient structures or processing strategies, we seek to address a more fundamental question: are there methods that can provide consistently effective hashing results regardless of the data distribution. To this end, we propose the concept of data-sensitive hashing (DSH) as an efficient mechanism for addressing the k-NN problem.

DSH directly deals with the k-NN problem instead of finding neighbors within a distance r. Hence, DSH avoids the issue of selecting the distance r that is required in LSH. DSH designs a novel scheme to facilitate the retrieval of a set of k-nearest neighbors with respect to a query point. Figure 1b shows the intuition of DSH. Since each bucket has a similar number of objects, its radius is adapted to the distance of k-NN. The basic idea of DSH is to hash the points such that the possibility of collision is much higher for k-NN pairs(unidirectional or bidirectional) than for non-k-NN pairs. It follows the same procedure as LSH for similarity search, the key difference being within the corresponding hashing families. Compared with random projections which have a large probability to hash the objects within a distance r together, our hashing family needs to have a large probability to hash the k-NN pairs together.

To find such hashing families, we learn from the data. Since we expect each k-NN pair to collide in most of the hashing functions, the requirement for hashing family can be represented as a strong classifier. Obviously, a single hashing function cannot protect all the k-NN pairs while separating all the non-k-NN pairs. Using spectral techniques, we can do an optimization on preserving the k-NN pairs while cutting the others. It is well known that adaptive boosting [8] is an efficient algorithm to generate a strong classifier using a set of weak classifiers. Therefore, we treat the hashing family as the strong classifier, and each hashing function in the family as a weak classifier. We adopt the adaptive boosting technique to ensure that the k-NN relations are theoretically protected by most of the hashing functions.

We summarize the contributions of the paper as follows.

- We propose a novel access method called Data Sensitive Hashing (DSH) to answer the k-nearest neighbor queries. Compared with LSH, DSH is able to capture the data distribution more effectively. Equipped with the distribution knowledge, DSH is able to deal with the k-NN problem in a more direct and efficient manner. DSH focuses on the hashing family, and thus most LSH extensions designed to enhance LSH are orthogonal to our proposal, and can be applied to DSH easily. (Sec. 3).

- We design an efficient algorithm to generate the data-sensitive hashing family – the key challenge for DSH. The algorithm combines adaptive boosting and spectral techniques, resulting in a good theoretical guarantee. The indexing time is also comparable with LSH. (Sec. 4).

- We experimentally verify the effectiveness and efficiency of our proposed DSH using three real datasets. Compared with LSH, our hashing results are much more balanced. Consequently, DSH is three times more efficient than LSH in query response time and index size in order to achieve the same quality of search results. (Sec. 5).

2. PRELIMINARIES

In this section, we provide the problem definition followed by a brief introduction and analysis of LSH. In the end, we review related work.

2.1 Problem Definition

In this paper, we consider data objects represented as points in a d-dimensional vector space R^d. Let $\|p, q\|$ be the distance measure between two points p and q in R^d. The k-nearest neighbor(k-NN) problem is defined as follows:

DEFINITION 1. **(k-nearest neighbor problem)** *Given a set O of n data points, a point $q \in R^d$ and an integer k, we aim to find the k data points that are nearest to q in O. We denote this answer set by $NN(q, k)$. Formally, $|NN(q, k)| = k$, $\forall o \in NN(q, k)$, $o' \in O \backslash NN(q, k)$, $\|q, o\| \leq \|q, o'\|$.*

Typically, finding exact k-NN potentially results in a sequential scan of the entire dataset O, and its cost grows linearly with the cardinality of O. For this reason, approximate k-nearest neighbor problem is accepted as a compromise since the cost of the solution grows sub-linearly with the cardinality of O while the quality is within an acceptable level. We define the approximate k-nearest neighbor problem as follows:

DEFINITION 2. **(δ-recall k-NN problem)** *Given a set O of n data points, a point $q \in R^d$ and an integer k, the δ-recall k-NN problem aims to find a set of k points $\delta NN(q, k)$, such that $|\delta NN(q, k)| = k$, $|\delta NN(q, k) \cap NN(q, k)| \geq \delta \times k$.*

The definition provides a lower bound of the recall that at least $k \times \delta$ points of exact k-NN are returned. δ-recall k-NN problem is indeed compatible with c-approximate k-NN problem discussed in Section 1. In c-approximate k-NN problem, it provides the upper bound of the distance for the points to be returned. To achieve a similar upper bound in the δ-recall k-NN problem, we can relax k to a larger k' ($k' = k/\delta$), and select k points from $\delta NN(q, k')$ with the smallest distances. As a result, we can guarantee that the farthest distance of these k points to q is within R' where R' is the maximum distance between q and the k'th-NN.

2.2 Locality Sensitive Hashing

Locality Sensitive Hashing is an efficient approximate algorithm for high dimensional similarity search. It is efficient and provides a rigorous quality guarantee for finding similar points within a distance r, i.e. the r-NN problem.

DEFINITION 3. **(r-NN problem)** *Given a set O of n data points, a point $q \in R^d$ and a distance r, we aim to find the data points that are within a distance r to q in O. We denote the sphere centered at point q by $B(q, r)$. Formally, $B(q, r) = \{p | p \in R^d, \|p, q\| \leq r\}$. The r-NN problem aims to find $\{o | o \in O$ and $o \in B(q, r)\}$.*

LSH leverages a family of functions where each function hashes the points in such a way that the possibility of collision is higher for similar points than for dissimilar points. Formally, an LSH family can be defined as follows [1]:

DEFINITION 4. **(LSH Family, $\mathcal{H}$)** *A family $\mathcal{H} = \{h : R^d \rightarrow U\}$ of functions is called (r, cr, p_1, p_2)-sensitive if for any $p, q \in R^d$*

- *if $p \in B(q, r)$, then $Pr_{\mathcal{H}}(h(p) = h(q)) \geq p_1$;*
- *if $p \notin B(q, cr)$, then $Pr_{\mathcal{H}}(h(p) = h(q)) \leq p_2$;*

The family of hash functions are generated through the use of random projections – the intuition is that points that are nearby in the space will also be nearby in all projections. While two distance points can also be close in all projections, the possibility is extremely small if enough number of projections is taken. Usually, the difference between p_1 and p_2 is not enough to be used directly. To enlarge the difference, a concatenation of LSH functions is applied to generate a hash key for each point in R^d.

DEFINITION 5. **(Concatenation LSH Functions, $\mathcal{G}$)** *A set of concatenation LSH functions $\mathcal{G} = \{g : R^d \rightarrow U^m\}$. Each $g_i \in \mathcal{G}$ consists of a sequence of m hash functions randomly extracted from $\mathcal{H}$. Formally,*

$$g_i(p) = (h_{i_1}(p), ..., h_{i_m}(p)),$$

where m is the number of hash functions in each concatenation and $h_{i_1}, ..., h_{i_m}$ are randomly selected from the LSH Family $\mathcal{H}$.

LSH applies these concatenation LSH functions to construct the hash tables. As a result,

- $\forall p \in B(q, r)$, $Pr(g(p) = g(q)) \geq p_1^m$
- $\forall p \in R^d \backslash B(q, cr)$, $Pr(g(p) = g(q)) \leq p_2^m$

Further, the expected number of points in O that collide with q but are outside the ball $B(q, cr)$ is less than $p_2^m * |O|$. However, the recall for one hash table, p_1^m, is not large. To raise the overall recall, LSH typically applies l concatenation LSH functions and constructs l hash tables. Each concatenation LSH function randomly chooses m functions in $\mathcal{H}$. When the number of functions in $\mathcal{H}$ is large enough, the hash results of different concatenation functions can be regarded as independent. Thus, for any $o \in B(q, r)$, the possibility that o collides with q in at least one hash table is at least $1 - (1 - p_1^m)^l$, which is very close to 1. To summarize, LSH answers c-approximate r-NN problem as follows:

1. **Pre-processing:** LSH maintains l hash tables, and each hash table is attached with a concatenation hash function. Each hash table applies its concatenation hash function to hash the points in O, where each concatenation hash function consists of m hash functions randomly selected from the hashing family $\mathcal{H}$;
2. **Query processing:** Given a query object q, LSH finds c-approximate r-NN by examining points that collide with q in each of the l hash tables. In particular, the expected number of objects outside the $B(q, cr)$ is limited by $l * p_2^m * |O|$. This property guarantees its query efficiency.

While LSH has been shown to be very effective in finding nearby points in high-dimensional space, we should note that this is accomplished with respect to a specified radius r, and an approximation allowance factor c. For a k-NN problem, the corresponding radius for different query points may vary by orders of magnitude. In such a case, LSH has to either (i) be run repeatedly with different values of r, cr,c^2r ... which leads to substantial increase in the query time and index storage cost, or (ii) use an ad-hoc r which leads to low quality guarantee.

Further, LSH implicitly splits the whole space into lattices so that points in the same lattice cells are hashed into the same bucket. As a result, for real data distributions that are non-uniform, the hashing results are often unbalanced. We find that an unbalanced hashing leads to performance degradation of LSH. The reasons are two-fold. First, for the case that the number of points in one bucket is too small, LSH cannot find c-approximate r-NN and a further examination on a larger r is required; second, for the case that too many points collide in one bucket, LSH needs to examine each of these and so its performance suffers.

2.3 Related Work

There is extensive work on high-dimensional indexes for k-NN queries, and there are surveys [7, 11] that provide good literature on the indexes. In this paper, since we are concerned with a specific well known indexing mechanism, namely LSH, we shall focus on work related to it (the comparison among LSH-based indexes and other indexes, such as iDistance [23], can be found in [20]).

Approximate k-NN provides approximate answers with acceptable error while constraining the growth of the cost sub-linearly with respect to the cardinality of the dataset. Locality sensitive hashing (LSH) methods [10, 1] are the best known approaches for approximate nearest neighbor search. The LSB-tree [20], entropy-LSH [15], multi-probe LSH [13] and Bayesian LSH [17] improve the performance of LSH by using some novel indexes or query strategies. Data Sensitive Hashing is orthogonal to these state-of-the-art methods, and can be regarded as a replacement of the LSH family. In the design of hashing families, query-sensitive embeddings(QSE)[4] and distance-based hashing(DBH)[5] have been proposed as the hash family for embedding and non-metric space, and [19, 12] were proposed for distance measure in the manifold. These works illustrate the potential for hashing family improvement. However, unlike existing work, DSH is the first hashing family leveraging the data distribution to directly improve the k-NN search.

There are also some spectral hashing [22] and data representation [16, 6, 14, 4, 3] methods that focus on optimizing the hashing function based on the data distribution. They significantly outperform LSH-based methods on precision. However, for the similarity search problem, we place much more emphasis on recall. This is because the recall affects the quality of the final result, while precision only affects performance by determining how many points have to be checked. DSH is designed to optimize both the recall and query efficiency.

3. DATA SENSITIVE HASHING

In this section, we introduce the basic concepts and principles of Data Sensitive Hashing (DSH), and propose two vari-

ants of DSH hashing families, DSH-basic and DSH-relaxed. The associated algorithms are presented in Section 4.

Intuitively, the problem we wish to solve is to directly search for the k-NN, whereas LSH is designed to find the r-NN. Therefore, we seek to define a new hashing family DSH. We begin with a DSH hashing family definition that migrates all the properties from LSH hashing family, called the DSH-utopia family. Here, the basic binary hashing $R^d \to \{0,1\}$ is employed for DSH.

DEFINITION 6. **(DSH-utopia Family)** *A family $\mathcal{H} = \{h : R^d \to \{0,1\}\}$ is called (k, ck, p_1, p_2)-sensitive if for any $q \in R^d$ and $o \in O$*

- *if $o \in NN(q,k)$, then $Pr_{\mathcal{H}}(h(o) = h(q)) \geq p_1$;*
- *if $o \notin NN(q,ck)$, then $Pr_{\mathcal{H}}(h(o) = h(q)) \leq p_2$;*

Unfortunately, while the above definition allows us to express what we desire, it has no known efficient implementation. Therefore, we fall back on the intrinsic properties that affect the effectiveness of a hashing-based approach. We define the following properties that should be satisfied in our DSH concatenation functions.

DEFINITION 7. **(Effectiveness for k-NN)** *For two probability values P_a and P_b, s.t. $P_a \gg P_b$,*

- **(Recall)** *for any query point $q \in R^d$, and $o \in NN(q,k)$, $\forall g_i \in \mathcal{G}$, $Pr(g_i(q) = g_i(o)) \geq P_a$;*
- **(Efficiency)** *for any query point $q \in R^d$, $\forall g_i \in \mathcal{G}$, $|\{o | o \in O \backslash NN(q,ck), g_i(q) = g_i(o)\}| \leq P_b \times |O|$.*

This definition provides two conditions to qualify a DSH concatenation function. The first condition provides a lower bound of recall and the second condition gives an upper bound (i.e., $ck + P_b \times |O|$) of the number of points in each bucket. By applying multiple hash tables to raise the recall, high quality results can be achieved, and the algorithm is still efficient. In addition, the results of different hash tables should be independent. If each concatenation function randomly chooses some functions from a large enough hashing family $\mathcal{H}$, this condition is always satisfied. The following theorem shows the relation between the quality and efficiency under such concatenation functions.

THEOREM 1. *If the concatenation functions satisfy the properties in Definition 7, to achieve an expected recall δ, the number of points to be checked is at most :*

$$\left\lceil \frac{\log(1-\delta)}{\log(1-P_a)} \right\rceil \times (ck + P_b \times |O|)$$

PROOF. Considering that l hash tables are used, we have:

$$(1-\delta) = (1-P_a)^l$$

To achieve an expected recall δ, the number of hash tables required is:

$$l = \left\lceil \frac{\log(1-\delta)}{\log(1-P_a)} \right\rceil$$

On the other hand, the number of points in each bucket is at most $ck + P_b \times |O|$. Thus, the number of points to be checked is at most:

$$\left\lceil \tfrac{\log(1-\delta)}{\log(1-P_a)} \right\rceil \times (ck + P_b \times |O|) \qquad \square$$

The quality of results is decided by δ, while c is only an efficiency factor. c and P_b decide the total number of points to be checked, and there is a trade-off between them. Obviously, the pairs with a longer distance are easier to be hashed to different values, and choosing a higher c often results in a smaller P_b.

To devise an effective hashing-based approach, according to the above guiding principles in Definition 7, there is no need that each **individual hash function** $h \in \mathcal{H}$ has a large probability to hash a pair correctly. Instead, we only need to design a **hash family** $\mathcal{H}$ such that most of the hash functions hash data correctly. As a concatenation LSH function is obtained by randomly picking m hash functions from $\mathcal{H}$, such a family can produce a good set of effective $g_i \in \mathcal{G}$ that satisfies the properties of recall and efficiency. Intuitively, based on following definition, we may design a set of hash functions each of which complements the others.

DEFINITION 8. **(DSH-basic Family)** *A family $\mathcal{H} = \{h : R^d \to \{0,1\}\}$ is called (k, ck, p_1, p_2)-sensitive if for any query point $q \in R^d$ and $o \in O$*

- *if $o \in NN(q,k)$, then $\frac{|\{h|h(o)=h(q), h \in \mathcal{H}\}|}{|\mathcal{H}|} \geq p_1$;*
- *if $o \notin NN(q,ck)$, then $\frac{|\{h|h(o)=h(q), h \in \mathcal{H}\}|}{|\mathcal{H}|} \leq p_2$;*

THEOREM 2. *The concatenation functions generated using DSH-basic family are effective, i.e. they have the properties given in Definition 7.*

PROOF. Using m randomly chosen hash functions in the DSH-basic family, setting $P_a = {p_1}^m$ and $P_b = {p_2}^m$, we have:
Recall: $\forall q \in R^d, o \in NN(q,k), g_i \in \mathcal{G}$,

$$\begin{aligned} & Pr(g_i(q) = g_i(o)) \\ &= \prod_{j=1}^{m} Pr(h_{i_j}(q) = h_{i_j}(o)) \\ &= \{\frac{|\{h|h(o) = h(q), h \in \mathcal{H}\}|}{|\mathcal{H}|}\}^m \\ &\geq {p_1}^m \end{aligned}$$

Efficiency: Likewise, $\forall q \in R^d, o \in O \backslash NN(q,ck), g_i \in \mathcal{G}$,
$Pr(g_i(q) = g_i(o)) \leq {p_2}^m$. Thus,
$\sum_{o \in O \backslash NN(q,ck)} Pr(g_i(q) = g_i(o)) \leq |O| \times {p_2}^m \quad \square$

We further note that the purpose of the second condition in the **Effectiveness** is to prevent too many false hits. In the DSH-basic family, this is accomplished by keeping the probability of inclusion low for each non-ck-NN. However, we do not really care what the probability of inclusion is for any **individual non-answer point**: all we wish to do is to limit the **total number of false positives**. Accordingly, we can relax the second condition:

DEFINITION 9. **(DSH-relaxed Family)** *A family $\mathcal{H} = \{h : R^d \to \{0,1\}\}$ is called (k, ck, p_1, p_2)-sensitive if for any query point $q \in R^d$*

- *for all $o \in NN(q,k)$, $\frac{|\{h|h(o)=h(q), h \in \mathcal{H}\}|}{|\mathcal{H}|} \geq p_1$;*
- *$\sum_{o \notin NN(q,ck)} (\frac{|\{h|h(o)=h(q), h \in \mathcal{H}\}|}{|\mathcal{H}|})^m \leq {p_2}^m * |O|$;*

Similar to Theorem 2, the concatenation functions generated using the DSH-relaxed family still have the **Effectiveness** properties given in Definition 7.

4. DSH FAMILY GENERATION

After having specified the properties of data sensitive hash functions, we now show how we can find such functions. For LSH, there is a straightforward geometric interpretation, so random projection turns out to work well. For DSH, there is no such easy geometric interpretation. Instead, the hash function must adapt based on the data distribution. We borrow and adapt machine learning techniques for this purpose. In particular, we first learn good atomic hash functions, and then use boosting to get even better results with an ensemble of hash functions, as we describe in this section.

4.1 Overview

Given an query-object pair $\langle q_i, o_j \rangle$, let

$$\varphi_h(\langle q_i, o_j \rangle) = (h(q_i) - h(o_j))^2. \quad (1)$$

In particular, if q_i collides with o_j according to hash function h, then $\varphi_h(\langle q_i, o_j \rangle)$ equals to 0; otherwise, it becomes 1. For DSH-basic family, based on Definition 8, the requirements for each hash function can be rewritten as:
For every pair q_i and o_j,

$$\begin{cases} \sum_{h \in \mathcal{H}} \varphi_h(\langle q_i, o_j \rangle) \leq (1 - p_1)|\mathcal{H}|, \text{if } o_j \text{ is a } k\text{-NN of } q_i; \\ \sum_{h \in \mathcal{H}} \varphi_h(\langle q_i, o_j \rangle) \geq (1 - p_2)|\mathcal{H}|, \text{if } o_j \text{ is a non-}ck\text{-NN of } q_i; \end{cases} \quad (2)$$

Through this expression, we observe that the hashing family is actually required to be a strong classifier for the pairs, i.e. if we combine all the hash function in $\mathcal{H}$ together and use Equation 2 to classify the pairs, every k-NN pair and non-ck-NN pair can be well classified. [We can also obtain similar requirements for DSH-relaxed family. The difference here is that DSH-relaxed only requires all the k-NN pairs to be well classified. For those non-ck-NN pairs, DSH-relaxed limits the total number of false-positives for each query q.]

Adaptive boosting [8] is an efficient meta-algorithm to generate a strong classifier. Given an algorithm that can generate a weak classifier, adaptive boosting can generate a set of weak classifiers by tweaking the weight of training instances, such that their combination constitutes a strong classifier. In our problem, we do not really combine them to generate a strong classifier. Instead, each weak classifier is used as a hash function h in the family $\mathcal{H}$, while the family $\mathcal{H}$ has the above desirable properties. Therefore, we have to adapt standard adaptive boosting.

We divide the problem into two parts: (1) computing the optimal weak classifier, and (2) applying appropriate adaptive boosting algorithm to tune the input weights of the weak classifier. In both parts, we use a weight matrix W to represent the k-NN and non-ck-NN relations. For the reason that non-ck-NN relations are much more than k-NN relations, we only sample parts of them to reduce the computation cost, and make the number of samples comparable with the number of k-NN relations. Thus, W is a sparse matrix, as it only contains some sampled query-object pairs ($O(k)$ non-zero elements per query, and $O(qk)$ non-zero elements in total). Using $i \in [1, m]$ to denote those queries, and $j \in [1, n]$ to denote those data points, we have:

$$W_{ij} = \begin{cases} 1, \text{if } o_j \text{ is a } k\text{-NN of } q_i \ ; \\ -1, \text{if } o_j \text{ is a sampled non-}ck\text{-NN of } q_i \ ; \\ 0, \text{else.} \end{cases} \quad (3)$$

Each hash function h hashes the points. On the other hand, we regard it as a weak classifier φ_h for the pairs. Therefore, we expect it to optimize the resultant pairs based on the weight matrix. The weight of the matrix will be tuned by the adaptive boosting procedure. And the optimization for each function is defined as follows.

Figure 2 outlines how we generate DSH family. Given some sampled queries based on the query distribution, we get a series of query-object pairs. The k-NN pairs are expected to collide more in DSH family and vice versa. We represent this optimization target in a weight matrix form. And based on the weight matrix, we compute the optimal hash function and put it into hashing family. We then hash the points based on the hash function and check if it performs well on each pair. After that, we run the adaptive boosting procedure: for the pairs that are hashed correctly (k-NN pairs collide and vice versa), we reduce their weight in the matrix; for the pairs that are hashed mistakenly, we increase their weight in the matrix. Then we get a new weight matrix and compute a new optimal hash function. As shown in [8], after repeating this procedure multiple times, the weight of every pair becomes smaller than the original one, and a smaller weight means the pair has a larger probability to be hashed correctly.

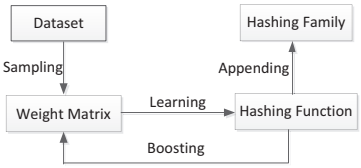


Figure 2: Overview of DSH Family Generation

The formal optimization problem set up requires specification of the data points, the query points, and the parameter k for k-NN. However, the principle of DSH is to leverage the distribution knowledge, but not tune the hash function based on specific points. We only seek to obtain the properties of DSH family for sampled queries based on the query distributions. Every sampled query should achieve high recall and be efficient. When the query distribution is hard to estimate, we can still use the uniform query distribution to cover all the potential queries. In the experimental section we show that the results obtained are not very sensitive to the query distribution or to the value of k. However, they do depend strongly on the data distribution.

DEFINITION 10. **(Hash Function Optimization)** Given the dataset O, the sampled queries Q and the weight matrix W, we aim to find the best hash function such that

$$\arg\min_h \sum_{ij} \varphi_h(\langle q_i, o_j \rangle) W_{ij} \quad (4)$$

s.t. $q_i \in Q, o_j \in O, \forall p \in R^d, h(p) \in \{0, 1\}$.

4.2 Single Hash Function Optimization

We would like to solve the optimization problem defined in Definition 10. However, finding the exact optimal hash function is an NP-hard problem.

THEOREM 3. *Hash Function Optimization (HFO) is an NP-hard problem.*

Algorithm 1: Hash Function Optimization

Input: Weight matrix $W_{q\times n}$, data point matrix $X_{d\times n}$, query point matrix $Q_{d\times q}$.
Output: Hash Function h.

1 Compute the eigenvector $\mathbf{a}$ with the minimal eigenvalue in Theorem 4 ;
2 $h \leftarrow \mathbf{a}^T X$;
3 Generate h' based on Equation 10;
4 **return** h'

PROOF. We can reduce a well-known NP-hard problem, Minimum Graph Bisection (MGB) [2] to the hash function optimization (HFO) problem. To make this reduction, we construct a mapping in the following way: let the $n \times n$ weight matrix in MGB be W and the largest weight in W be w_{max}. For each vertex v_i in MGB, we create o_i and q_i in HFO. We begin with the initial weight matrix W in MGB. Then, we ensure that o_i and q_i must be assigned to the same value by setting $W'_{ii} = n^4 \times w_{max}$. Finally, we set $W'_{ij} = W_{ij} - n^2 \times w_{max}$. The intuition here is that if a large penalty is added for every pair that is not separated, then HFO will absolutely choose a balanced partition(the minimal penalty) while doing the optimization in MGB.

□

To tackle this problem efficiently, we consider the hash function in a linear form, i.e. separating the space by a hyperplane. This has been widely adopted, and shown to be effective [14]. Moreover, it has been shown in [18] that linear classifiers fit well with boosting algorithms.

That is, $h(o_j)$,$h(q_i)$ can be represented as $\mathbf{a}^T X_j$,$\mathbf{a}^T Q_i$, where $\mathbf{a}$ is the projection vector and X_j,Q_i is the d dimension vector presentation of o_j, q_i(suppose it has been regularized to $\overline{X_j} = \mathbf{0}$).

However, the range of $\mathbf{a}^T X_j$ is R instead of $\{0, 1\}$. The mean of $h(o_j)$ is always 0, as $\overline{h(o_j)} = \overline{\mathbf{a}^T X_j} = \mathbf{a}^T \overline{X_j} = 0$. To protect the result from being affected by the scaling of $\mathbf{a}$, we give a constraint $\mathbf{a}^T X X^T \mathbf{a} = 1$ to fix the variance of $h(o_j)$. We transform the problem of hash function generation as follows:

$$\arg\min_{\mathbf{a}} \sum_{ij} (\mathbf{a}^T Q_i - \mathbf{a}^T X_j)^2 W_{ij} \tag{5}$$

subject to $\mathbf{a}^T X X^T \mathbf{a} = 1$.

THEOREM 4. *The vector in Equation 5 is equal to the minimal general eigenvector of*

$$((XD - QW)X^T + (QD' - XW^T)Q^T)\mathbf{a} = \lambda X X^T \mathbf{a} \tag{6}$$

where $D_{n\times n}$ and $D'_{q\times q}$ are two diagonal matrices defined as follows:

$$D_{jj} = \sum_i W_{ij} \qquad D'_{ii} = \sum_j W_{ij} \tag{7}$$

PROOF.

$$\sum_{ij} (\mathbf{a}^T Q_i - \mathbf{a}^T X_j)^2 W_{ij}$$
$$= \sum_j \mathbf{a}^T X_j D_{jj} X_j^T \mathbf{a} - 2 \sum_{ij} \mathbf{a}^T Q_i W_{ij} X_j^T \mathbf{a} + \sum_i \mathbf{a}^T Q_i D'_{ii} Q_i^T \mathbf{a}$$
$$= \mathbf{a}^T ((XD - QW)X^T + (QD' - XW^T)Q^T)\mathbf{a}$$

Therefore, the vector in Equation 5 is equal to

$$\arg\min_{\mathbf{a}} \mathbf{a}^T ((XD - QW)X^T + (QD' - XW^T)Q^T)\mathbf{a} \tag{8}$$

subject to $\mathbf{a}^T X X^T \mathbf{a} = 1$. All the two matrices are symmetric. Therefore, $\mathbf{a}$ is equal to the eigenvector with the minimal eigenvalue in the following generalized eigenvector problem:

$$((XD - QW)X^T + (QD' - XW^T)Q^T)\mathbf{a} = \lambda X X^T \mathbf{a} \tag{9}$$

□

The computational complexity of solving the eigenvector problem is $O(nd^2 + qkd + d^3)$, where n is the number of objects plus sampled queries, d is the number of dimensions and q is the number of sampled queries. The main cost here is to compute $((XD-QW)X^T+(QD'-XW^T)Q^T)$. Here D only has n non-zero values, D' has q non-zero values, and W has $O(qk)$ non-zero values. And for the general eigenvector problem, the computation cost is $O(d^3)$. Noted that the size of dataset is nd, our algorithm runs in time that is at most **linear** with the size of dataset.

After computing the hashing function $h : R^d \to R$, we convert it to the desired binary hash function $h' : R^d \to \{0, 1\}$ as:

$$h'(o) = \begin{cases} 0, \text{if } h(o) \le 0 \\ 1, \text{else} \end{cases} \tag{10}$$

4.3 Adaptive Boosting for DSH-basic

Adaptive boosting [8] is a meta-algorithm for building a strong classifier using a set of weak classifiers, and can be used in conjunction with many other learning algorithms. Theoretical results that adaptive boosting always obtains a strong classifier can be found in [8]. Here we only show the intuitions. The weight for each instance depends on its results (well classified or not) on the weak classifiers, and thus represents if it is well classified using their combination. If its weight is smaller than the original, then it is well classified using the linear combination. On the other hand, in each step, the weak classifier performs better than random so that the weight of well classified instances is larger than the weight of misclassified instances. Therefore, the total weight of all instances is always reduced by the weak classifier. As the weight is tuned exponentially, the total weight also reduces exponentially. After logarithmic number of steps, the total weight is reduced to be less than the original weight for a single instance, and by then, every instance has a weight less than its initial weight and is therefore well classified.

For our specific problem, not only a strong classifier is needed, but also the difference between p_1 and p_2 is expected to be large. Obviously we cannot expect the boosting algorithm to achieve $p_1 = 1$ and $p_2 = 0$. However, the boosting algorithm optimizes to enlarge the difference [18]. Therefore, we choose some appropriate p_1 and p_2 and iterate based on the following weight updating function in which α is a weight tuning parameter of boosting:
For k-NN pairs, we have:

$$W_{ij}^{(t+1)} = W_{ij}^{(t)} * \alpha^{(p_1 - 1 + \varphi_{h_t}(\langle q_i, o_j \rangle))} \tag{11}$$

For non-ck-NN pairs, we have:

$$W_{ij}^{(t+1)} = W_{ij}^{(t)} * \alpha^{-(p_2 - 1 + \varphi_{h_t}(\langle q_i, o_j \rangle))} \quad (12)$$

Algorithm 2 describes our boosting procedure in which *itrCounter* is the number of hash functions that we aim to obtain. We next prove that if every instance has a weight smaller than its initial, then $\mathcal{H}$ conforms to DSH-basic family defined in Definition 8 for the sampled data when the updating function is based on Equation 11 and 12.

THEOREM 5. *$\forall \langle q_i, o_j \rangle$, if $|w_{ij}^{(t)}| \leq |w_{ij}^{(0)}|$, $\mathcal{H}$ conforms to Definition 8 for all sampled pairs.*

PROOF. Let T_{ij} be $|\{h|h(q_i) = h(o_j), h \in \mathcal{H}\}|$, and S_{ij} be $|\{h|h(q_i) \neq h(o_j), h \in \mathcal{H}\}|$. Obviously, $S_{ij} = |\mathcal{H}| - T_{ij}$. Hence, for every k-NN pair $\langle q_i, o_j \rangle$,

$$\begin{aligned} & w_{ij}^{(0)} \geq w_{ij}^{(t)} \\ \Longrightarrow & w_{ij}^{(0)} \geq w_{ij}^{(0)} \times \alpha^{(p_1 - 1)|\mathcal{H}| + S_{ij}} \\ \Longrightarrow & \alpha^{(p_1 - 1)|\mathcal{H}| + S_{ij}} \leq 1 \\ \Longrightarrow & (p_1 - 1)|\mathcal{H}| + S_{ij} \leq 0 \\ \Longrightarrow & p_1|\mathcal{H}| - T_{ij} \leq 0 \\ \Longrightarrow & \frac{|\{h|h(o_i) = h(o_j), h \in \mathcal{H}\}|}{|\mathcal{H}|} \geq p_1 \end{aligned}$$

For every non-ck-NN pair $\langle q_i, o_j \rangle$, it is analogous to prove that $\frac{|\{h|h(q_i) = h(o_j), h \in \mathcal{H}\}|}{|\mathcal{H}|} \leq p_2$. □

4.4 Adaptive Boosting for DSH-relaxed

Now we give the adaptive boosting solution for DSH-relaxed. The procedure is similar to that for DSH-basic except one major difference, i.e., we only limit the total number of points in each bucket. Therefore, for every query, the non-ck-NN relations now construct one big instance. For ease of presentation, we assume that all the pairs are sampled and the size of the current H is large enough. However, as only a small number of pairs are sampled and the algorithm begins with empty H, some trivial regularization should be applied in the implementation. We use the expected number of points that collide with q_i to measure the efficiency of query q_i, which is defined as:

$$Collision_i = \sum_j (1 - \frac{\sum_{h \in \mathcal{H}} (\varphi_h(\langle q_i, o_j \rangle)}{|\mathcal{H}|})^m \quad (13)$$

Note that if the collision rate bound is p_2, then the upper bound of the false-positives is $n * {p_2}^m$. Thus, $(\frac{Collision_i}{|O|})^{\frac{1}{m}}$ is the equivalent collision rate to get the same upper bound of the false positives. After being regularized, the weight for each query is:

$$weight_i^{(t)} = \alpha^{t*(p_2 - (\frac{Collision_i}{|O|})^{\frac{1}{m}})} \quad (14)$$

For k-NN pairs, we tune their weight as usual. On the other hand, we tune the weight for each query q_i based on $Collision_i$. However, our hash function optimization algorithm is based on the weight for every query-object pair. For this reason, we have to assign the weight of each query q_i to its related pairs. Here, we still expect the hash function to minimize the total weight of training instances. Thus,

Algorithm 2: Adaptive Boosting Procedure

```
initialize t ← 0;   H ← ∅ ;
initialize W^(0) = W given in Equation 3;
while t < itrCounter || ∃ w_ij^(t), w_ij^(0), |w_ij^(t)| > |w_ij^(0)| do
    compute h^(t) based on Algorithm 1 using W^(t) as input;
    H ← H ∪ {h^(t)};
    updating W^(t+1);
return H;
```

we study the "gradient" vector of $Collision_i$, i.e. how much it will be changed when a new hash function hashes each point $o_1 ... o_n$ together with q_i or not. We then adjust the weight for each related pair based on its "partial derivative". Formally, the "partial derivative" is defined as:

$$\Delta Collision_{ij} = \frac{m}{|\mathcal{H}|} \times (1 - \frac{\sum_{h \in \mathcal{H}} (\varphi_h(\langle q_i, o_j \rangle)}{|\mathcal{H}|})^{m-1} \quad (15)$$

Intuitively, if an object has a larger probability to collide with the query, it will have a larger weight. For those objects that are almost impossible to collide, there is little benefit to separate them with the query in the new hash function, and the weight for them is close to zero.

Now we define the weight updating function for those non-ck-NN pairs:

$$W_{ij}^{(t)} = weight_i^{(t)} \Delta Collision_{ij} \quad (16)$$

The weight updating function for k-NN pairs remains the same.

4.5 Difference with other learning based methods

So far, all the discussion is based on the similarity search prospective. There are many other learning based hashing or embedding methods such as [22, 4, 3, 16, 6, 14]. Now we will discuss our difference with them. Compared with these methods, DSH is distinctive in the optimization target. Traditional learning based methods mostly focus on the high precision end, and evaluate their performance using PR-value, F_1-measure or MAP. However, this may not be appropriate for our similarity search problem. As we have a validation step, the precision itself is not a crucial measure. We aim to achieve a high recall with acceptable candidate size. For example, to solve a 20-NN search problem in 1M points, we expect the algorithm to return 2000 points that contain all the positives, but not to return 20 points that contain 12 of them. We can validate those 2000 points one by one and obtain the exact answer. However, for the 20 points set, we cannot expand it to obtain a higher recall. Although the algorithm that outputs 2000 points has a 1% precision, the check rate is only 0.2%. For this reason, we focus on the relatively lower precision area, and optimize our algorithm to yield a higher recall. Distinguished with traditional methods, DSH does not pay much to filter every false positive, but to ensure every true answer is in the result, while most other methods pay equal emphasis to these two targets.

Table 1: Overall comparative study: DSH-relaxed is the most efficient in terms of query time and space usage

(a) Forest					(b) Flickr				(c) DBPedia			
method	recall	query time	error ratio	# hash tables	recall	query time	error ratio	# hash tables	recall	query time	error ratio	# hash tables
LSH		18.6	1.012	100		40.8	1.004	90		/	/	/
DSH-basic	0.94	12.8	1.012	80	0.96	28.6	1.004	72	0.95	25.6	1.002	64
DSH-relaxed		6.2	1.013	30		16.0	1.004	36		15.6	1.002	32
LSH		12.1	1.024	50		26.0	1.008	40		38.8	1.006	75
DSH-basic	0.9	6.8	1.027	32	0.92	17.2	1.009	36	0.91	14.4	1.006	28
DSH-relaxed		3.8	1.025	14		9.8	1.010	16		9.4	1.006	16
LSH		8.2	1.036	25		19.2	1.013	25		24.8	1.018	35
DSH-basic	0.86	4.4	1.036	16	0.88	12.0	1.014	20	0.87	11.6	1.017	20
DSH-relaxed		2.4	1.038	8		8.0	1.014	10		7.6	1.021	10

Following this guideline, in the design of DSH-relaxed, we provide zero tolerance to the false negatives, and a much higher tolerance to the false positives. For the false negatives, we use boosting techniques to ensure all the positives are well classified. And for the false positives, we only give a very weak limitation on the total numbers, and do our best for each individual. By using this heterogeneous solution, our optimization target is realized.

4.6 Incremental Maintenance

DSH is sensitive to data distribution, unlike LSH. Indeed, that is the whole point. This raises the question of what to do if the data distribution changes over time in a dynamic setting. Fortunately, incremental update of DSH hashing family is straightforward: simply replace any hash function that has a negative effect on the weight of boosting by a newly computed one since the boosting procedure aims at reducing the total weight of the weight matrix. The cost of such updates can be limited by changing at most one hash function each time. In practice, it turns out to be enough to make one such incremental update after 1-5% of the data has changed.

5. EXPERIMENTS

5.1 Experimental Setup

We evaluate the performance using three real datasets: scientific data, images and text. Properties of these three datasets are summarized in Table 2 and described in detail below:

- **Forest**[1] dataset is provided by the US Forest Service. We observe that some attributes are strongly correlated since they describe related information. For example, there are some attributes describing Hillshade Index at different times of a day. For this reason, Forest dataset is ideal to study the performance of indexing methods with respect to **data skew**;
- **Flickr**[2] is an image hosting website in which members can upload their images. Over this dataset, we do similarity search based on the feature space of images instead of pixel space. Following common practice, we use the PCA technique to pre-process the images and keep 80 feature dimensions. In contrast to the Forest dataset, the PCA feature space is **orthogonal**;
- **DBPedia**[3] is a project aiming to extract structured content from the Wikipedia database. We utilize LDA to pre-process the dataset, and keep 150 topic dimensions. The data points after employing LDA topic model transformation lie in the probability distribution space, and hence we use KL-divergence as the distance measure since it is widely used in this space. We use DBPedia dataset to study the effectiveness of DSH when a **non-Euclidean distance** measure is used.

Table 2: Dataset properties

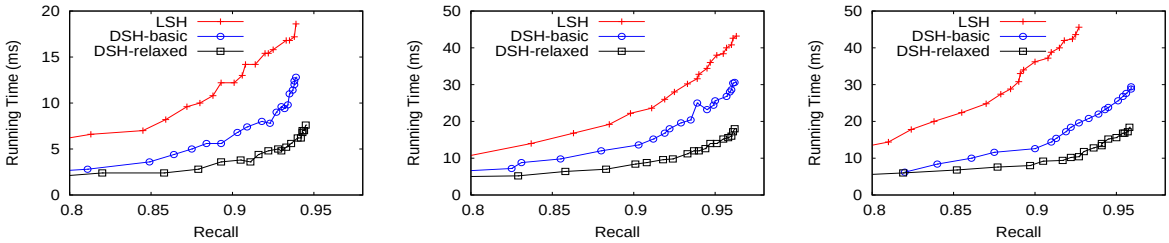
Dataset	# Objects	# Dimension	Property
Forest	580K	54	skewness
Flickr	1M	80	orthogonal
DBPedia	1M	150	non-Euclidean

We evaluate the following methods in the experiments:

- **LSH.** E2LSH[1] is a recent state-of-the-art implementation for LSH.
- **DSH-basic.** Our method proposed in Section 4.3.
- **DSH-relaxed.** Our method proposed in Section 4.4.
- **Query-Sensitive Embedding**[4] is one efficient embedding hashing method that also applies boosting to guarantee the performance of similarity search.
- **Spectral Hashing**[22] is a state-of-the-art learning based hashing methods.

We randomly remove 1000 points from each dataset and use them as query points in our performance study. The ground-truth for each query point is obtained by a linear scan of the entire dataset. The sampled queries used in DSH are generated via random selection that is independent of the query set. Unless otherwise specified, the sample rate is 0.5%, k is set to 20 and c is set to 5 throughout the experiments. For all algorithms, each hash table contains 2000 buckets(i.e. m is set to 11). To avoid duplicating the

[1] http://archive.ics.uci.edu/ml/datasets/Covertype
[2] http://www.flickr.com/
[3] http://wiki.dbpedia.org



(a) Forest (b) Flickr (c) DBPedia

Figure 3: Average running time executed using different access methods to achieve a certain query quality: DSH-relaxed achieves the highest recall in the same running time, followed by DSH-basic and LSH.

points, entries in the table are in form of point IDs instead of the points. For this reason, the storage cost can be reduced by about 100 times. Besides, all these methods keep the entire indexes as well as the original dataset in main memory.

We evaluate the performance in terms of the following three aspects:

- **Query Quality** is measured by *recall* and *error ratio*. Let $\widehat{NN(q,k)}[i]$ (resp. $NN(q,k)[i]$) be the point in $\widehat{NN(q,k)}$ (resp. $NN(q,k)$) with the ith smallest distance to q. The *error ratio* [13] is to measure how close are the distances between points in $\widehat{NN(q,k)}$ and points in $NN(q,k)$:

$$error\ ratio = \frac{1}{|Q| \times k} \sum_{q \in Q} \sum_{i=1}^{k} \frac{|\widehat{NN(q,k)}[i], q|}{|NN(q,k)[i], q|} \quad (17)$$

- **Query Efficiency** is measured by the *running time* to answer k-NN queries. The *precision* in those learning based methods can also be viewed as a measurement of efficiency, since $\frac{recall \times k}{precision}$ is the number of points to be checked.

- **Space Requirement** is measured by the *number of hash tables being used*. The space cost for one hash table is about the same for each method, and is a good surrogate for index size.

5.2 Comparison with LSH

In this study, we compare our proposed methods, namely DSH-basic and DSH-relaxed, with LSH. We choose the best width parameter W for LSH, and the parameters for DSH tuned as discussed in Section 5.5.

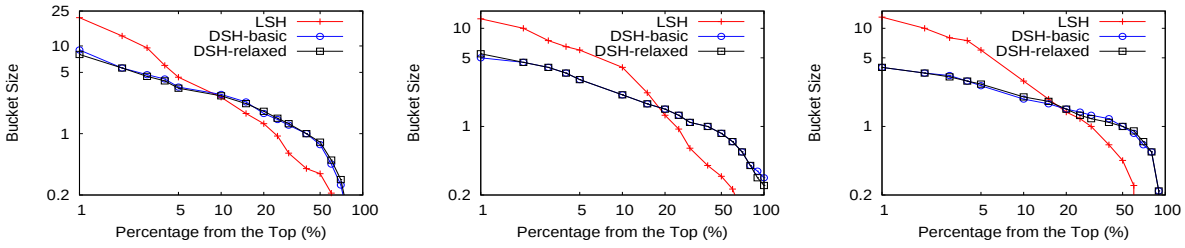
Table 1 summarizes the average results over all the datasets. In each table, we report the query time, the error ratio and the number of hash tables required per query to achieve three different query quality levels (in terms of recall). First, we observe that DSH-relaxed is significantly more space and time efficient than DSH-basic and LSH. It is worth mentioning that for the DBPedia dataset, LSH cannot find approximate k-nearest neighbors to achieve recall 0.94 within acceptable time. Further, we note that all three methods provide almost the same error ratio across all the datasets and each error ratio is close to 1, in which the exact k-NN are identified.

Typically, a larger δ will result in a higher query cost. We therefore study the relationship between recall δ and query performance in detail for all three access methods. Figure 3 summarizes the results. First, we observe that the query time increases super-linearly to achieve a higher recall. Second, both DSH-relaxed and DSH-basic perform better than LSH by a wide margin and there is an obvious trend of increasing time to achieve a higher recall. Finally, we observe that DSH-relaxed and DSH-basic outperform LSH by a wider margin over the DBPedia dataset than the other two datasets. The reason will be provided when we discuss the relationship between the recall and the space usage.

Regarding the comparative methods, the query performance basically depends on the number of candidates to be probed in the same buckets that collide with the query points. Hence, we report the distribution of bucket size of hash tables and discuss how this affects the query performance. Figure 4 shows the distribution of bucket size of hash tables. The points are distributed more unevenly in hash tables of LSH than those of DSH-basic and DSH-relaxed. In particular, for hash tables in LSH, the top 1% buckets take 21%, 13%, 13% points of the Forest, Flickr and DBPedia datasets, respectively. Obviously, the query performance suffers when the query points are hashed to these buckets. Typically, the query distribution follows the data distribution, and in this case, the query points are often hashed to the buckets with a large number of points. Consequently, the number of candidates in LSH is larger on average than that in DSH methods, and incurs a higher query cost.

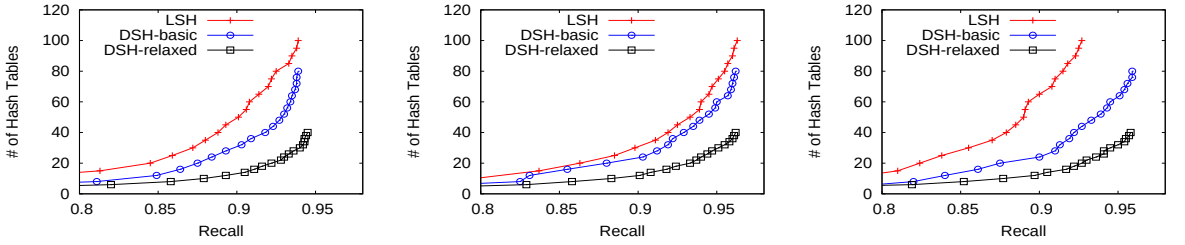
We proceed to study the relationship between the recall and the space usage, and report the results in Figure 5. To achieve a higher recall, a larger number of hash tables is typically required. In particular, DSH-relaxed outperforms DSH-basic and LSH by a wide margin. The trends are fairly similar to Figure 3. However, there exists a slight difference between them. We note that using similar number of hash tables, LSH incurs about 25% longer query time than DSH methods. The reasons are two-fold. First, due to the unbalanced hashing in LSH, the number of points to be checked in each table is about 3 times larger than that in DSH. Second, there exist many more duplicates from different tables in LSH than those in DSH where each point will be checked at most once. As a result, DSH methods still benefit from its balanced hash tables.

In addition, DSH-relaxed and DSH-basic outperform LSH by a wider margin on the DBPedia dataset than the other two datasets. Due to the difference between Euclidean distance and KL-divergence, the recall of LSH is limited at 94%. Although DSH methods are not specially designed for



(a) Forest (b) Flickr (c) DBPedia

Figure 4: Distribution of Bucket Size



(a) Forest (b) Flickr (c) DBPedia

Figure 5: Number of hash tables that are required by different access methods to achieve a certain query quality: DSH-relaxed achieves the highest recall by using the same number of hash tables, followed by DSH-basic and LSH.

Table 3: Query time and recall for various distributions

Query		Dense	Sparse	Original
LSH	query time	581	42	130
	recall	0.99	0.803	0.916
DSH-basic	query time	161	58	98
	recall	0.978	0.893	0.932
DSH-relaxed	query time	150	59	90
	recall	0.984	0.922	0.962

KL-divergence measurement, they achieve a recall of 96%, which can be improved further by using a larger number of hash tables.

We also investigate the performance of each method under different query distributions. Besides the *original* queries that are randomly extracted from each dataset, we generate two additional query sets for each dataset and each set consists of 100 queries. Queries in the first set (labeled as *dense*) are randomly selected and the distances between each query and its k-NNs are small. Queries in the second set (labeled as *sparse*) are randomly selected as well and the distances between each query and its k-NNs are large. The results (dense, sparse and the original) of using $l = 40$ on Flickr dataset are shown in Table 3. Due to the space constraint, in what follows, we only report the results on the Flickr dataset whenever the results on the other two datasets exhibit similar trend. From the result we can see that the performance for DSH is generally consistent in all cases. However, the recall of LSH drops greatly in the sparse area, and the query time increases significantly in the dense area.

5.3 Scalability

Table 4 summarizes the indexing time for the comparative methods. To test the scalability of our indexing method, we also conduct an experiment where all the data points are duplicated for 10 times. Notice that both DSH and LSH treat those duplicated points as new points, thus indicating the expected performance with a larger data set. First we can see that both LSH and DSH take 9-10 times longer in the 10 times larger dataset, suggesting that the indexing time grows linearly with size of data set for all methods.

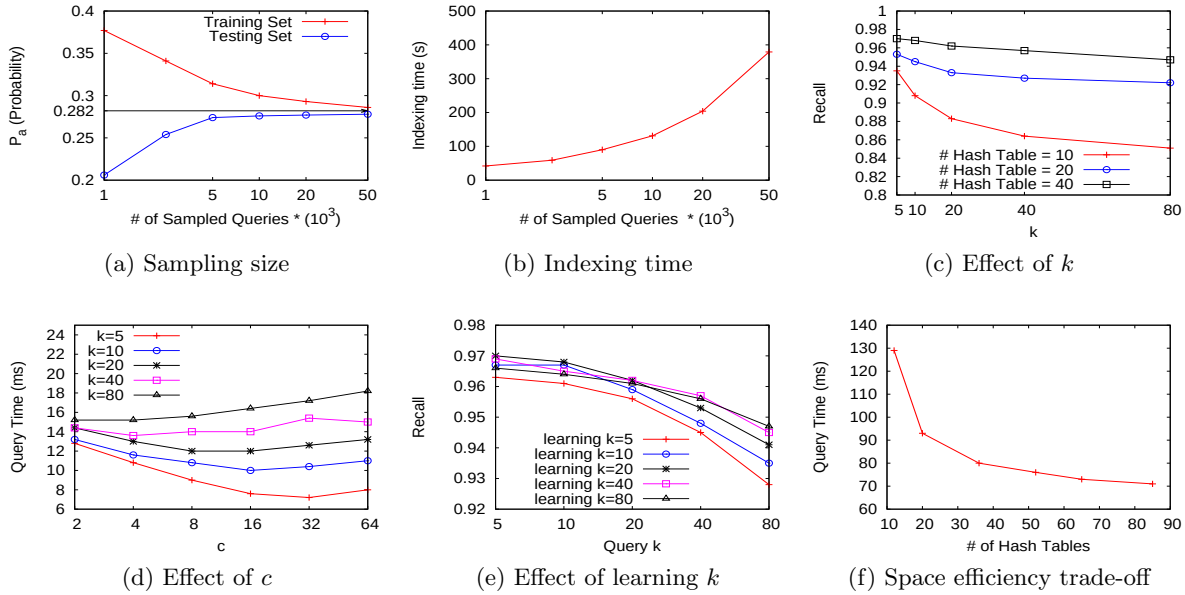
Table 4: Indexing time (s)

Dataset	Forest		Flickr		DBPedia	
Size	1×	10×	1×	10×	1×	10×
LSH	57	502	92	875	105	1001
DSH-basic	37	331	148	1350	84	798
DSH-relaxed	32	286	109	980	77	719

We also note that neither DSH methods nor LSH takes the minimum indexing time across all three datasets, and the indexing time over different datasets for all methods varies slightly. Specifically, DSH-relaxed and DSH-basic are more efficient than LSH on Forest and DBPedia datasets while we get a reverse result on Flickr dataset. Note that in all comparative methods, the indexing time consists two components: (1) the time to obtain the hash functions; (2) the cost of hashing points to the hash tables. LSH uses the random projection to generate hash functions and hence it performs much better than DSH-relaxed and DSH-basic in obtaining the hash functions. However, DSH-relaxed and DSH-basic require fewer hash tables, and incur less cost to hash points. By considering both factors, the overall indexing time for DSH is comparable with LSH.

5.4 Comparison with Learning Based Methods

We now compare DSH with other learning based methods. As discussed in Section 4.5, the optimization target of DSH



(a) Sampling size (b) Indexing time (c) Effect of k

(d) Effect of c (e) Effect of learning k (f) Space efficiency trade-off

Figure 6: Effect of parameters in DSH

Table 5: Precision-recall comparison with learning methods

Precision	0.1	0.03	0.01	0.003	0.001
C/N(%)	0.01	0.03	0.1	0.3	1
LSH	0.551	0.704	0.8	0.852	0.916
DSH-basic	0.684	**0.816**	0.869	0.904	0.932
DSH-relaxed	0.652	0.812	**0.894**	**0.936**	**0.962**
[22]	**0.738**	0.808	0.864	0.888	0.919
[4]	0.677	0.794	0.864	0.897	0.925

is different from that of other learning based methods. Table 5 gives the comparison with query-sensitive embedding and spectral hashing. In this table, we use the precision-recall trade-off to show the basic difference illustrated in Section 4.5. We give each method equal space limitation $l = 40$, and evaluate their precision and recall when m is varied.

DSH-relaxed performs worse than the others in the high precision end, as it allows too many false positives into the results. But as a payback, it wins the highest recall in the relative lower precision area. On the other hand, spectral hashing performs the best in the high precision area. However, in the low precision area, its performance is the worst (similar with the performance of LSH) since it does not make any guarantee for each individual and some positive objects are totally lost. DSH-basic and Query Sensitive Hashing lie in the middle, as they both enforce a strict constraint on both false positives and false negatives. And DSH-basic outperforms query-sensitive hashing about 1% for the reason that it preserves the collision probability for each k-NN point, while query-sensitive hashing only guarantees that the probability is larger than those non-k-NNs. When the precision is 1%, the check rate(candidate size/datasize) is only 0.1%. Thus, we focus more on the performance in the relatively lower precision area and DSH-relaxed outperforms other algorithms.

For embedding or data representation applications, there is a need to compute the similarity in the transformation space. Thus, the precision itself is very important. However, to answer the similarity search, the validation is conducted in the original space. For this reason, all those false-positives can be pruned. From this experiment we see that the most suitable algorithms for these two scenarios are different.

5.5 Parameter Studies

In this study, we investigate the parameters that potentially affect the performance of DSH, and suggest guidelines for parameter setting.

We first study the effect of sample rate by varying from 0.1%(1K queries) to 5%(50K). We evaluate the recall property P_a DSH can achieve when the efficiency property is guaranteed by is $P_b = 0.005$. Figure 6a shows the results. By enlarging the sample size, a more accurate model can be trained to capture the distribution of the whole dataset. The value in the training set shows that the upper bond of P_a is less than 0.282. On the other hand, the real performance becomes very close to that value when sample rate is larger than 0.5%(5K). Therefore, we set the sample rate to 0.5%. Also, for each query, we only acquire 10 k-NN relations and 10 non-ck-NN relations. The recall needed here is only 50%. Thus, we can process the sampling process efficiently by some approximate search algorithm. Figure 6b shows the indexing time when different sample size are used.

We next study the effect of k by varying k from 5 to 80. Figure 6c shows the results. In general, the recall drops slightly when k increases. It turns out to be increasingly insensitive to k by using a larger number of hash tables.

We also study the relationship between c and query efficiency, and report the query time in Figure 6d using different k. In general, the optimal c that achieves the minimal query time differs when k varies. However, we can see that for every k, the value ck that achieves the minimal query time always falls in the range from 100 to 200. As shown in

Table 6: Augmentation with multi-probe proposed for LSH

Method	# table		Query time	
	original	multi-probe	original	multi-probe
LSH	90	9	40.8	44.2
DSH-basic	72	7	28.6	29.4
DSH-relaxed	36	4	16	17.2

Theorem 1, the efficiency is decided by both $P_b \times N$ and ck. Thus, the best ck value depends on the bucket size. In our experiments, the bucket size is about 500 points, and the ck that achieves the best query performance lies in the range from 100 to 200.

Besides, there is a need to generate a DSH family for various k. Thus, we study the performance using different k in both the learning and query phase. Using $l = 40$, the result in Figure 6e shows a larger k in the learning phase performs well over various query k in the query phase. For DSH, using $k = 80$ only introduces a cost of checking 80 points for each bucket at most. Meanwhile, when a larger k is applied in the learning phase, the query quality performance for those smaller k will also be guaranteed. Thus, we can use a larger k to learn DSH in the various query k scenario.

Finally, we study the space-efficiency trade-off of DSH. Typically, to achieve the same level of recall, using larger m and l will reduce the query time, while leading to additional storage cost. Thus, we study the relationship between query time and storage cost when the recall requirement is fixed to 0.96, and the results are shown in Figure 6f. We can observe that by increasing the number of hash tables, the query time drops and the performance gain becomes smaller.

5.6 Extensibility

DSH is orthogonal to most of the research works that use LSH family as a foundation and improve on LSH. Therefore, the results from most of these works can be applied to DSH as an alternative to LSH. As an example, we adapt Multi-probe LSH [13] into DSH. The results to obtain 0.96 recall are given in Table 6. We can see that the combination further reduces the number of tables needed while requiring almost the same query time.

6. CONCLUSION

In this paper, we propose an efficient indexing method called *Data Sensitive Hashing* (DSH) for high-dimensional approximate k-NN search problem. The hashing family is directly designed for k-NN search. By leveraging knowledge from the sampled data, DSH balances its buckets even in non-uniform data distributions, and aims to preserve most k-NN relations in its hashing. We conducted extensive experiments using three real datasets, and the results confirm the efficiency and robustness of the DSH. The DSH-relaxed variant is the most efficient, which requires only 1/3 of the hash tables and query time of the LSH. Furthermore, existing LSH extensions can easily be applied to the DSH.

Acknowledgments

This work was in part supported by the National Research Foundation, Prime Minister′s Office, Singapore under its Competitive Research Programme (CRP Award No. NRF-CRP8-2011-08) and Singapore Ministry of Education Grant No. R-252-000-454-112. Jagadish was partially supported by NSF grant IIS 1250880.

7. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, pages 459–468, 2006.

[2] K. Andreev and H. Racke. Balanced graph partitioning. *Theory of Computing Systems*, pages 929–939, 2006.

[3] V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. Boostmap: A method for efficient approximate similarity rankings. In *CVPR*, pages II–268, 2004.

[4] V. Athitsos, M. Hadjieleftheriou, G. Kollios, and S. Sclaroff. Query-sensitive embeddings. In *SIGMOD*, pages 706–717, 2005.

[5] V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios. Nearest neighbor retrieval using distance-based hashing. In *ICDE*, pages 327–336, 2008.

[6] M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, pages 1373–1396, 2003.

[7] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, pages 273–321, 2001.

[8] Y. Freund and R. E. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory*, pages 23–37, 1995.

[9] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, pages 541–552, 2012.

[10] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[11] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). *TODS*, pages 517–580, 2003.

[12] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, pages 2130–2137, 2009.

[13] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[14] X. Niyogi. Locality preserving projections. In *NIPS*, pages 153–160, 2004.

[15] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, pages 1186–1195, 2006.

[16] R. Salakhutdinov and G. E. Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. In *AI and Statistics*, pages 412–419, 2007.

[17] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *VLDB*, pages 430–441, 2012.

[18] R. E. Schapire. The boosting approach to machine learning: An overview. *LECTURE NOTES IN STATISTICS*, pages 149–172, 2003.

[19] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *ICCV*, pages 750–757, 2003.

[20] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576, 2009.

[21] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[22] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, pages 1753–1760, 2008.

[23] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB*, pages 421–430, 2001.