Neighbor-Sensitive Hashing

Yongjoo Park
University of Michigan
2260 Hayward Street
Ann Arbor, MI, USA
pyongjoo@umich.edu

Michael Cafarella
University of Michigan
2260 Hayward Street
Ann Arbor, MI, USA
michjc@umich.edu

Barzan Mozafari
University of Michigan
2260 Hayward Street
Ann Arbor, MI, USA
mozafari@umich.edu

ABSTRACT

Approximate *k*NN (*k*-nearest neighbor) techniques using binary hash functions are among the most commonly used approaches for overcoming the prohibitive cost of performing exact *k*NN queries. However, the success of these techniques largely depends on their hash functions' ability to distinguish *k*NN items; that is, the *k*NN items retrieved based on data items' *hashcodes*, should include as many *true k*NN items as possible. A widely-adopted principle for this process is to ensure that similar items are assigned to the same hashcode so that the items with the hashcodes similar to a query's hashcode are likely to be true neighbors.

In this work, we abandon this heavily-utilized principle and pursue the opposite direction for generating more effective hash functions for *k*NN tasks. That is, we aim to *increase* the distance between similar items in the hashcode space, instead of reducing it. Our contribution begins by providing theoretical analysis on why this revolutionary and seemingly counter-intuitive approach leads to a more accurate identification of *k*NN items. Our analysis is followed by a proposal for a hashing algorithm that embeds this novel principle. Our empirical studies confirm that a hashing algorithm based on this counter-intuitive idea significantly improves the efficiency and accuracy of state-of-the-art techniques.

1. INTRODUCTION

Finding the k most similar data items to a user's query item (known as k-nearest neighbors or kNN) is a common building block of many important applications. In machine learning, fast kNN techniques boost the classification speed of non-parametric classifiers [10, 30, 49, 60]. kNN is also a crucial step in collaborative filtering, a widely used algorithm in online advertisement and movie/music recommendation systems [31,48]. Moreover, in databases and data mining, finding the k-most similar items to users' queries is the crux of image and text searching [34,50,52].

Unfortunately, despite thirty years of research in this area [7, 12, 19, 23, 28, 35, 58], *exact* kNN queries are still prohibitively expensive, especially over high-dimensional objects, such as images, audio, videos, documents or massive scientific arrays [55]. For example, one of the most recent approaches for exact kNN that exploits sophisticated pruning is only $9\times$ faster than a baseline table scan [28]. This prohibitive cost has given rise to *approximate* kNN.

One of the most common approaches to finding *approximate* kNN is using a set of binary hash functions that map each data item into a binary vector, called a hashcode. The database then finds the kNN items using the Hamming distance[1] among these (binary) hashcodes. Due to special properties of these binary vectors [42], searching for kNN in the hash space (a.k.a. Hamming space) can be performed much more efficiently than the original high-dimensional space. These approaches are approximate because a query item's kNN in the Hamming space may be different than its kNN in the original space. Thus, the accuracy of hashing-based approximations is judged by their effectiveness in preserving the kNN relationships (among the original items) in the Hamming space. In other words, given a query item q and its kNN set $\{v_1, \cdots, v_k\}$, a hash function $\mathbf{h}$ should be chosen such that most of the hashcodes in $\{\mathbf{h}(v_1), \cdots, \mathbf{h}(v_k)\}$ fall in the kNN set of $\mathbf{h}(q)$ in the Hamming space.

Existing Approaches— Starting with the pioneering work of Gionis, Indyk, and Motwani on *locality sensitive hashing (LSH)* over 15 years ago [17], numerous techniques have been proposed to improve the accuracy of the hashing-based kNN procedures [5, 11, 13, 18, 20, 21, 26, 27, 32, 33, 36, 38, 45, 56, 57]. Implicitly or explicitly, almost all hashing algorithms pursue the following goal: to preserve the relative distance of the original items in the Hamming space. That is, if we denote the distance between items v_1 and v_2 by $\|v_1 - v_2\|$ and the Hamming distance between their respective hashcodes by $\|\mathbf{h}(v_1) - \mathbf{h}(v_2)\|_H$, the hash function $\mathbf{h}$ is chosen such that the value of $\|\mathbf{h}(v_1) - \mathbf{h}(v_2)\|_H$ is (ideally) a linear function of $\|v_1 - v_2\|$, as shown in Figure 1(c). Thus, while previous techniques use different ideas, they tend to minimize the Hamming distance of nearby items while maximizing it for far apart items.

Figure 1(a) uses a toy example with nine data items (including a query item q) to illustrate how existing methods choose their hashcodes. In this example, four hyperplanes h_1, h_2, h_3, and h_4 are used to generate 4-bit hashcodes. Here, each hyperplane h_i acts as a binary separator to determine the i-th leftmost bit of the output hashcode; the i'th bit of $\mathbf{h}(x)$ is 0 if the item x falls on the left side of h_i and this bit is 1 otherwise. For instance, the hashcode for v_3 is 1000 because v_3 falls in the region to the right of h_1 and to the left of h_2, h_3 and h_4 hyperplanes.

To preserve the original distances, existing hashing techniques tend to place fewer (more) separators between nearby (far apart) items to ensure that their hashcodes differ in fewer (more) positions and have a smaller (larger) Hamming distance. In other words, the expected number of separators between a pair of items tends

[1]Hamming distance between two binary vectors of equal length is the number of positions at which the corresponding bits are different.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 3
Copyright 2015 VLDB Endowment 2150-8097/15/11.

to be roughly proportional to their relative distance in the original space.[2] In this example, v_5's distance from q is twice v_4's distance from q. This ratio remains roughly the same after hashing: $\|\mathbf{h}(v_5)-\mathbf{h}(q)\|_H = \|1100-0000\|_H = 2$ while $\|\mathbf{h}(v_4)-\mathbf{h}(q)\|_H = \|1000-0000\|_H = 1$. Likewise, since v_8 is the farthest item from q, four separators are placed between them, causing their hashcodes to differ in four positions (0000 versus 1111) and thus yielding a greater Hamming distance, namely $\|\mathbf{h}(v_8)-\mathbf{h}(q)\|_H = 4$.

This goal is intuitive and can capture the intra-item similarities well. However, this approach requires a large number of hash bits (i.e., separators) to accurately capture all pair-wise similarities. Since using longer hashcodes increases the response time of search operations, we need a better strategy than simply increasing the number of separators. In this paper, we make the following observation. Since the ultimate goal of the hashing phase is to simply find the kNN items, preserving all pair-wise similarities is unnecessary and wasteful. Rather, we propose to spend our hash bits on directly maximizing the accuracy of the kNN task itself, as described next.

Our Approach— In this work, we pursue the opposite goal of previous approaches. Instead of preserving the proximity of similar items in the Hamming space, we maximize their Hamming distance. In other words, instead of placing fewer separators between nearby items and more between far apart items, we do the opposite (compare Figures 1(a) and (b)).

We argue that this seemingly counter-intuitive idea is far more effective at solving the kNN problem, which is the ultimate goal of hashing. The key intuition is that we should not use our limited hash bits on capturing the distances among far apart items. Instead, we use our hash bits to better distinguish nearby items, which are likely to be in each other's kNN sets. Given a fixed number of hash bits (i.e., separators), we can achieve this distinguishing power by placing more separators among similar items. In the previous example, to find the 3-NN (i.e., $k=3$) items for item q, we must be able to accurately compare v_3 and v_4's distances to q using their respective hashcodes. In other words, we need to have $\|\mathbf{h}(v_3)-\mathbf{h}(q)\|_H < \|\mathbf{h}(v_4)-\mathbf{h}(q)\|_H$ in order to infer that $\|v_3-q\| < \|v_4-q\|$.

However, due to the proximity of v_3 and v_4, existing methods are likely to assign them the same hashcode, as shown in Figure 1(a). In contrast, our strategy has a higher chance of correctly differentiating v_3 and v_4, due to its higher number of separators among nearby items. This is shown in Figure 1(b), v_3 and v_4's hashcodes differ by one bit. Obviously, our idea comes at the cost of confusing far apart items. As shown in Figure 1(b), we will not be able to differentiate q's distance to any of v_5, v_6, v_7, or v_8. However, this is acceptable if the user is interested in $k \leq 4$.

This intuition can be applied to more general cases, where the query item q or the value of k may not be necessarily known in advance. If we choose a neighborhood size just large enough to include most of the kNN items that typical users are interested in, e.g., for $k=1$ to 1000, we expect an increased accuracy in correctly ordering such items, and hence returning the correct kNN items to the user. Since the value of k is typically much smaller than the total number of the items in a database, we expect significant gains over existing techniques that seek to preserve all pair-wise distances using a fixed number of hash bits.

The stark difference between our strategy and previous techniques is summarized in Figure 1(c). The goal of existing methods is to assign hashcodes such that the Hamming distance between each pair of items is as close to a linear function of their original distance as possible. Our method changes the shape of this function, shown as a solid line; we impose a larger slope when the original distance between a pair of items is small, and allow the curve to level off beyond a certain point. This translates to a higher probability of separating the kNN items from others in our technique (we formally prove this in Section 3.1).

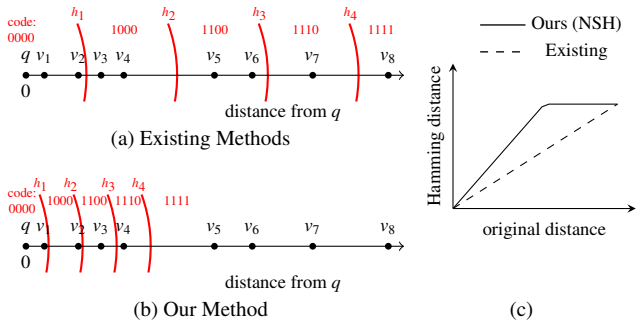


Figure 1: In (a) and (b), the vertical arcs indicate the boundaries where the Hamming distance from q increases by 1. The third figure (c) shows the relationship between data items' original distance and their *expected* Hamming distance.

The main challenge then is how to devise a hashing mechanism that can achieve this goal. We solve this problem by proposing a special transformation that stretches out the distance between similar items (compared to distant items). Our method, called *Neighbor-Sensitive Hashing (NSH)*, uses these transformed representations of items to achieve the goal described above.

Contributions— In this paper, we make several contributions.

1. We formally prove that increasing the distance between similar items in the Hamming space increases the probability of successful identification of kNN items (Section 3.1).
2. We introduce *Neighbor-Sensitive Hashing (NSH)*, a new hashing algorithm motivated by our seemingly counter-intuitive idea (Sections 3.2, 3.3, and 3.4).
3. We confirm our formal results through extensive experiments, showing the superiority of *Neighbor-Sensitive Hashing* over *Locality-Sensitive Hashing* [5] and other state-of-the-art hashing algorithms for approximate kNN [16, 21, 26, 36, 38, 56]. (Section 4).

In summary, our algorithm for NSH achieves 250× speedup over the baseline, obtaining an average recall of 57.5% for 10-NN retrieval tasks. Compared to the state-of-the-art hashing algorithms, our algorithm reduces the search time by up to 34% for a fixed recall (29% on average), and improves the recall by up to 31% for a fixed time budget.[3]

We overview the end-to-end workflow of hashing-based techniques in Section 2. We present our NSH strategy in Section 3. Section 4 reports our empirical analysis and comparisons against existing hashing algorithms. Section 5 overviews the related work, and Section 6 concludes our paper with future work.

2. HASHING-BASED KNN SEARCH

In this section, we provide the necessary background on hashing-based approximate kNN. Section 2.1 explains a typical workflow in hashing-based approximate kNN. Section 2.2 reviews a well-known principle in designing hash functions to compare with ours.

[2] We provide a more precise dichotomy of previous work in Section 5.

[3] In approximate kNN, a simple post ranking step is used to mitigate the impact of low precision while preserving the recall [24, 25]. See Section 4.1.

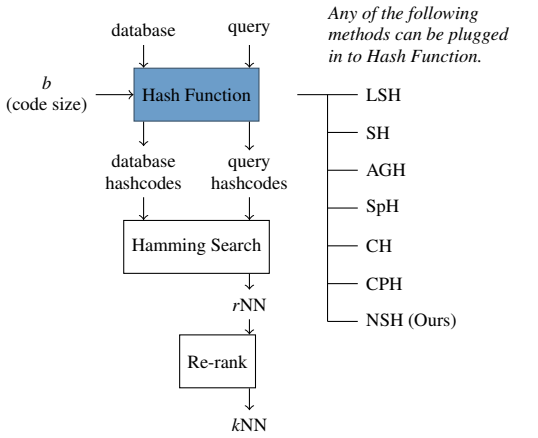


Figure 2: The workflow in hashing-based search consists of two main components: Hash Function and Hamming Search. Re-rank is an extra step to boost the search accuracy. This paper improves the most critical component of this workflow, i.e., Hash Function.

2.1 Workflow

Figure 2 summarizes the prototypical workflow of a hashing-based kNN system. Among the three componenets, Hash Function and Hamming Search are more important. Hash Function is the component that converts the data items residing in the database at index time into binary vectors, known as *hashcodes*. The same function is used to also convert the query item provided at run time into a hashcode. The choice of Hash Function is critical for the accuracy of the kNN task. Ideally, the hashcodes should be generated such that the kNN set retrieved based on these hashcodes is always identical to the kNN set based on the original distances. However, no tractable method is known for achieving this goal; even a simpler problem is proven to be NP-hard [56]. As a result, hashing-based techniques are only *approximate*; they aim to return as many true kNN items as possible. In this paper, we focus on improving the accuracy of this component (or its efficiency, given a required level of accuracy).

The length of the hashcodes (b) is an important design parameter that must be determined at index time. In general, there is a trade-off between the hashcode length and the search speed. The longer hashcodes tend to capture the original distances more accurately, while they slow down the other runtime component, namely the Hamming Search.

Once the hashcodes that capture the original distance information (to some extent) are generated, Hamming Search is responsible for time-efficient retrieval of the kNN items in the Hamming space. The simplest approach would be a linear scan during which the distance between the query's hashcode and the hashcode of every item in the database is computed. Although this simple approach improves over a linear scan over the original data vectors, there have been more efficient algorithms, such as *Multi-Index Hashing* (MIH), developed for exact Hamming Search [42] in sub-linear time complexity. Note that this search speedup is only possible because the hashcodes are (small) binary vectors; the data structure cannot be used to speed up the search for general multi-dimensional data representations.

The last component of the system, Re-rank, is a post-lookup re-ranking step designed for mitigating the negative effects of the hashing step on accuracy. Instead of requesting exactly k data items to Hamming Search, we can request r ($\geq k$) data items. Next, we recompute the similarity of each retrieved item to the query item (in their original representations), then sort and choose the top-k among these r items. Naturally, the larger the value of r is, the more accurate the final kNN items are. However, using a larger r has two drawbacks. First, it needs more time to obtain answers from Hamming Search. Second, the re-ranking process takes more time.

2.2 Hash Function Design

Since the Hash Function choice in Figure 2 is independent of the Hamming Search component, the primary objective in designing a good hash function has been finding a hash function that produces high average recalls for a given hashcode length. The Hash Function component is in fact composed of *b bit functions*, each responsible for generating an individual bit of the overall hashcode. Next, we define the role of these bit functions more formally.

Definition 1. (Bit Function) A function h that takes a data item v and produces $h(v) \in \{-1,1\}$ is called a bit function. Here, v can be a novel query item or any of the existing items in the database. The value of the bit function, $h(v)$, is called a *hash bit*.

Note that in reality binary bits are stored in their $\{0,1\}$ representations. However, using signed bits $\{-1,1\}$ greatly simplifies our mathematical expressions. Thus, we will use signed binary bits throughout the paper.

Definition 2. (Hash Function) A hash function $\mathbf{h}$ is a series of b bit functions $(h_1,\ldots,h_b)$. The hash bits produced by h_1 through h_b are concatenated together to form a *hashcode* of length b.

We consider a hashcode of v as a b-dimensional vector whose elements are either of $\{-1,1\}$. A natural distance measure between two hashcodes is to count the number of positions that have different hash bits, known as the Hamming distance. As mentioned in Section 1, we denote the Hamming distance between data items v_i and v_j by $\|\mathbf{h}(v_i)-\mathbf{h}(v_j)\|_H$.

Finally, we formally state the *locality-sensitive* property [11,13], which is a widely accepted principle for designing hash functions. Let q be a query, and v_i, v_j be two arbitrary data items. We say that a bit function h satisfies the locality-sensitive property, if

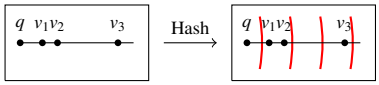
$$\|q-v_i\| < \|q-v_j\| \Rightarrow \Pr(h(q)\neq h(v_i)) < \Pr(h(q)\neq h(v_j)). \quad (1)$$

where $\Pr(\cdot)$ denotes the probability.

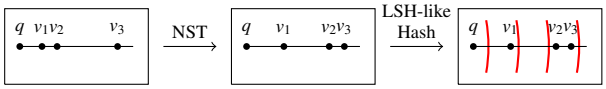
Datar *et al.* [13] showed that assigning hash bits based on their relative locations with respect to a set of randomly-drawn hyperplanes satisfies the locality-sensitive property. That is, for a b-bit hashcode, b independent hyperplanes are drawn from a normal distribution to compose a hash function. Using an unlimited number of hash bits using this approach could perfectly capture the original distances. However, many recent algorithms have shown that this simple approach does not achieve high kNN accuracy when the hashcodes need to be short [21,26,27,36,56].

3. NEIGHBOR-SENSITIVE HASHING

This section describes our main contribution, *Neighbor-Sensitive Hashing* (NSH). First, Section 3.1 formally verifies our intuition introduced in Section 1: using more separators for nearby data items allows for more accurate distinction of kNN items. As depicted in Figure 3, NSH is the combination of a hashing algorithm and our proposed Neighbor-Sensitive Transformation (NST). Section 3.2 lays out a set of abstract mathematical properties for NST, and Section 3.3 presents a concrete example of NST that satisfies those properties. Lastly, Section 3.4 describes our final algorithm (NSH) that uses the proposed NST as a critical component.



(a) Regular Hashing



(b) Hashing with NST

Figure 3: The motivation behind using *Neighbor-Sensitive Transformation* (NST) before hashing: applying NST to data items makes the same hashing algorithm place more separators between nearby items (v_1 and v_2), and place fewer separators between distant items (v_2 and v_3).

3.1 Formal Verification of Our Claim

In this section, we formally verify our original claim: using more separators between data items leads to a more successful ordering of data items based on their hashcodes. First, let us formalize the intuitive notions of "having more separators" and "correct ordering based on hashcodes":

- Let q be a query point, v_1, v_2 be two data items where $\|q-v_1\| < \|q-v_2\|$, and $\mathbf{h}$ be a hash function that assigns hashcodes to these items. Then, having more separators between v_1 and v_2 means a larger gap in terms of their Hamming distance, namely $\|\mathbf{h}(q)-\mathbf{h}(v_2)\|_H - \|\mathbf{h}(q)-\mathbf{h}(v_1)\|_H$ will be larger.
- For v_1 and v_2 to be correctly ordered in terms of their distance to q, v_1's hashcode must be closer to q's hashcode compared to v_2's hashcode. In other words, $\|\mathbf{h}(q)-\mathbf{h}(v_2)\|_H - \|\mathbf{h}(q)-\mathbf{h}(v_1)\|_H$ must be a positive value.

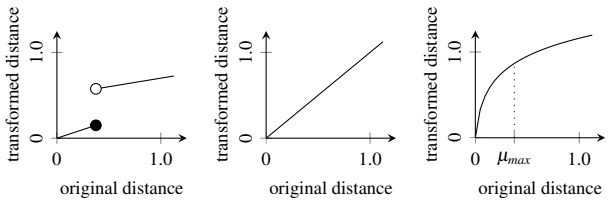
In the rest of this paper, without loss of generality, we assume that the coordinates of all data items are normalized appropriately, so that the largest distance between pairs of items is one. Next, we define the class of hash functions we will use.

Definition 3. (LSH-like Hashing) We call a bit function h *LSH-like*, if the probability of two items v_i and v_j being assigned to different hash bits is linearly proportional to their distance, namely $\Pr(h(v_i) \neq h(v_j)) = c \cdot \|v_i - v_j\|$ for some constant c. We call a hash function $\mathbf{h}$ LSH-like if all its bit functions are LSH-like.

Note that not all existing hashing functions are LSH-like. However, there are several popular hashing algorithms that belong to this class, such as LSH for Euclidean distance [13]. With these notions, we can now formally state our claim.

Theorem 1. Let q be a query, and v_1 and v_2 two data items. Also, let $\mathbf{h}$ be an LSH-like hash function consisting of b *independent* bit functions $h_1,\ldots,h_b$. Then, the following relationship holds for all v_1 and v_2 satisfying $0.146 < \|q-v_1\| < \|q-v_2\|$: A larger value of $E\|\mathbf{h}(q)-\mathbf{h}(v_2)\|_H - E\|\mathbf{h}(q)-\mathbf{h}(v_1)\|_H$ implies a larger value of $\Pr(\|\mathbf{h}(q)-\mathbf{h}(v_1)\|_H < \|\mathbf{h}(q)-\mathbf{h}(v_2)\|_H)$, i.e., the probability of successful ordering of v_1 and v_2 based on their hashcodes.

The proof is quite involved, as it uses an integration for computing the probability of successful ordering. Due to space constraints, we refer the reader to our technical report for the complete proof [44]. This theorem implies that having more separators between two data items helps with their successful ordering using their hashcodes. Since the total number of separators is a fixed budget b, we need to borrow some of the separators that would otherwise be used for distinguishing distant (or non-kNN) items.



(a) Only Monotonic (b) No Larger Gap (c) Satisfies All

Figure 4: Visual demonstration of NST properties.

The following sections describe how this theorem can be used for designing such a hash function.

3.2 Neighbor-Sensitive Transformation

As shown in Figure 3, the main idea of our approach is that combining our *Neighbor-Sensitive Transformation* (NST) with an LSH-like hash function produces a new hash function that is highly effective in distinguishing nearby items. In this section, we first define NST, and then formally state our claim as a theorem.

Definition 4. (Neighbor-Sensitive Transformation (NST)) Let q be a query. A coordinate-transforming function f is a q-neighbor-sensitive transformation for a given distance range (η_{min}, η_{max}), or simply a q-(η_{min}, η_{max})-sensitive transformation, if it satisfies the following three properties:

1. Continuity: f must be continuous.[4]
2. Monotonicity: For all constants t_i and t_j where $t_i \leq t_j$, f must satisfy $E(\|f(q)-f(v_i)\|) < E(\|f(q)-f(v_j)\|)$, where the expectations are computed over data items v_i and v_j chosen uniformly at random among items whose distances to q are t_i and t_j, respectively.
3. Larger Gap: For all constants t_i and t_j where $\eta_{min} \leq t_i \leq t_j \leq \eta_{max}$, f must satisfy $E(\|f(q)-f(v_j)\| - \|f(q)-f(v_i)\|) > t_j - t_i$, where the expectation is computed over data items v_i and v_j chosen uniformly at random among items whose distances to q are t_i and t_j, respectively.

The coordinates are re-normalized after the transformation, so that the maximum distance between data items is 1.

To visually explain the properties described above, three example functions are provided in Figure 4. Among these three functions, Figure 4(c) is the only function that satisfies all three properties — the function in Figure 4(a) is neither continuous nor satisfies the Larger Gap property, and the function in Figure 4(b) is continuous and monotonic but does not satisfy the Larger Gap property.

The third property of NST (Larger Gap) plays a crucial role in our hashing algorithm. Recall that our approach involves an LSH-like hashing whereby two data items are distinguished in the Hamming space with a probability proportional to their distance. This implies that if we alter the data items to stretch out their pairwise distances, their pairwise Hamming distances are also likely to increase. Thus, such data items become more distinguishable in Hamming space after the transformation.

Thus far, we have defined NST using its three abstract properties. Before presenting a concrete example of a NST, we need to formally state our claim.

[4]This condition is to prevent a pair of similar items in the original space from being mapped to radically different points in the transformed space.

Theorem 2. Let $\mathbf{h}$ be an LSH-like hash function and f be a q-(η_{min}, η_{max})-sensitive transformation. Then, for all constants t_i and t_j, where $\eta_{min} \leq t_i \leq t_j \leq \eta_{max}$, we have the following:

$$\begin{aligned} &E(\|\mathbf{h}(f(q)) - \mathbf{h}(f(v_j))\|_H - \|\mathbf{h}(f(q)) - \mathbf{h}(f(v_i))\|_H) \\ &> E(\|\mathbf{h}(q) - \mathbf{h}(v_j)\|_H - \|\mathbf{h}(q) - \mathbf{h}(v_i)\|_H) \end{aligned} \quad (2)$$

where the expectations are computed over data items v_i and v_j chosen uniformly at random among data items whose distances to q are t_i and t_j, respectively.

The proof of this theorem can be found in our technical report [44]. Now that we have established that NST can help with constructing more effective hashcodes, our next task is to find a concrete transformation function that satisfies NST's three properties.

3.3 Our Proposed NST

In this section, we first propose a coordinate transformation function for a known query q, and describe its connection to NST (Definition 4). Then, we extend our transformation to handle unknown queries as well. The proposed NST is also a crucial component of our hashing algorithm, which will be presented in the following section.

Definition 5. (Pivoted Transformation) Given a data item p, a pivoted transformation f_p transforms an arbitrary data item v as follows:

$$f_p(v) = \exp\left(-\frac{\|p-v\|^2}{\eta^2}\right) \quad (3)$$

where η is a positive constant. We call p the pivot.

For a pivoted transformation $f_p(v)$ to be a q-neighbor-sensitive transformation, we need the distance between p and q to be small enough. The proximity of p and q is determined by the ratio of their distance to the value of η. For example, our lemma below shows that $\|p-q\| < \eta/2$ is a reasonable choice.

To gain a better understanding of the connection between a pivoted transformation and NST, suppose that the pivot p is at the same point as the query q, and that v_1 and v_2 are two data items satisfying $\|q-v_2\| \geq \|q-v_1\|$. We consider two cases: the first is that v_1 and v_2 are close to q so that their distances are less than η, and the second case is that v_1 and v_2 are distant from q so that their distances are much larger than η. In the first case, the distance between v_1 and v_2 after the transformation is $\exp(-\|q-v_1\|^2/\eta^2) - \exp(-\|q-v_2\|^2/\eta^2)$, which tends to be relatively large because the exponential function $\exp(-x^2)$ decrease fast around 1. In the second case, when the data items are far from the query, the value of the exponential function becomes almost zeros, and so does the distance between those data items after the transformation. In other words, the transformation has an effect of stretching out the space near the query while shrinking the space distant from the query.

Next, we establish a connection between a pivoted transformation and NST. First, it is straightforward that a pivoted transformation satisfies *continuity*, since it only uses continuous functions. The second property of NST, monotonicity, is shown by the following lemma.

Lemma 1. A pivoted transformation f_p satisfies the second property of NST, i.e., monotonicity.

This lemma can be proven using the law of cosines as we present in our technical report [44]. The next lemma is regarding the third property of NST, namely a Larger Gap.

Lemma 2. A pivoted transformation f_p with $\|p-q\| < \eta/2$ and $\eta < 0.2$ satisfies the third property of NST, i.e., Larger Gap, for $(\eta_{min}, \eta_{max}) = (0.13\eta, 1.6\eta)$. That is, f_p is a q-$(0.13\eta, 1.6\eta)$-sensitive transformation.[5]

A q-$(0.13\eta, 1.6\eta)$-sensitive transformation implies that our intended effect may not be achieved for those data items whose distances to q are smaller than 0.13η. Fortunately, a simple case study shows that the number of such data items is negligibly small: consider 100 million data points that are uniformly distributed in a 10-dimensional space; then, the number of data items that fall within the distance $0.13\,\eta$ (or 0.026) from q will be $100 \cdot 10^6 \cdot 0.026^{10} = 1.4 \times 10^{-8}$. Considering that users are typically interested in a relatively small number of results from their search engines, say the top 1–1000 items, we see that this condition can cover most of the practical cases.

Handing Novel Queries— Thus far, we have described our NST for a known query q. However, we also need to handle queries that are not known *a priori*. Note that, from Lemma 2, we know that NST properties hold for all queries that are within a $\eta/2$ distance from a pivot. Handling queries that are extremely far apart from *all* data items in the database will therefore be difficult. However, assuming that novel queries will be relatively close to at least one of the data items, we can handle such queries by selecting multiple pivots that can collectively cover the existing data items. Based on this observation, we propose the following transformation to handle novel queries.

Definition 6. (Multi-Pivoted Transformation) Let f_p be a pivoted coordinate transformation in Definition 5 using a pivot p. Our extended version to handle novel queries is as follows. Choose m pivots $\{p_1, \ldots, p_m\}$, and compute the below to obtain a multi-dimensional representation of a data item v:

$$f(v) = (f_{p_1}(v), \ldots, f_{p_m}(v)) \quad (4)$$

To understand how a multi-pivoted transformation works for novel queries, assume for the moment that there is at least one pivot p_i that is close enough to a novel query q. Then, this pivot works in the same way as in a single pivoted transformation: it stretches out the distances between this novel query and other data items around it. As a result, when combined with an LSH-like hash function, more separators are used to distinguish q and its nearby items. On the other hand, from the perspective of other (far-apart) pivots, the distances between the q and its nearby items tend to be very small after the transformation, due to the exponential function used in the pivoted transformation. Consequently, those far-apart pivots are effectively ignored by a multi-pivoted transformation when computing the distance of q and its neighbors. This effect of the multi-pivoted transformation is examined empirically in Section 4.2. However, one question remains: how can we ensure that there will be at least one nearby pivot for every novel query?

Parameter η— To ensure that there is at least one pivot close enough to each novel query, we use the assumption that each novel query is close to at least one data item in the database. Then, it will suffice to select pivots in such a way that every data item in the database is close to at least one pivot. Specifically, assume that m pivots are chosen by one of the algorithms presented in the next section (Section 3.4), and let γ denote the average distance between a pivot and its closest neighbor pivot. Then, to ensure that any data item is within a $\eta/2$ distance from its closest pivot, we should set

[5] When working with *non-normalized distances*, η should be smaller than $0.2 \cdot t_{max}$, where t_{max} is the maximum distance between data items.

η to a value larger than γ. This is because the maximum distance between data items and their respective closest pivots will be larger than $\gamma/2$. Our numerical study in Section 4.2 shows that, with our choice of $\eta = 1.9\gamma$, most of the novel queries fall within a $\eta/2$ distance from their closest pivot. We also show, in Section 4.6, that the search accuracy does not change much when η is larger than γ.

For a multi-pivoted transformation, we also need to determine (1) the number of pivots (m) and (2) a strategy for selecting m pivots. We discuss these two issues in Section 3.4 after presenting the technical details of our algorithm.

3.4 Our NSH Algorithm

This section describes our algorithm, *Neighbor-Sensitive Hashing* (NSH). Besides NST, another ingredient for our hashing algorithm is enforcing the *even distribution* of data items in Hamming space, which is a widely-adopted heuristic. Let h_i^* represent a column vector $(h_i(v_1),\ldots,h_i(v_N))^T$ where $v_1,\ldots,v_N$ are the data items in a database. In other words, h_i^* is a column vector of length N that consists of all i-th bits collected from the generated hashcodes. Then, the technical description of the even distribution of the data points in the Hamming space is as follows:

$$(h_i^*)^T\mathbb{1} = 0 \quad \forall i = 1,\ldots,b \tag{5}$$

$$(h_i^*)^T h_j^* = 0 \quad \forall i,j = 1,\ldots,b \text{ and } i \neq j \tag{6}$$

The first expression induces that the hash bits are *turned on* with 50% chance. The second expression induces that two hash bits in different positions are uncorrelated so that they have different hash bits in different positions. The second condition also means that the conditional probability that a data item receives 1 for i-th hash bit is independent of the probability that the data item receives 1 for j-th hash bit if $i \neq j$.

The primary objective NSH is to generate a hash function using NST, while best ensuring the above requirement for the data items that reside in the database. First, if we consider a data item v as a d-dimensional column vector, the hash bits are determined by NSH in the following way: $h_i = \text{sign}(w_i^T f(v) + c_i)$, where f is a multi-pivoted transformation with m pivots, w_i is a m-dimensional vector, and c_i is a scalar value. Our main goal in this section is to find the appropriate values for w_i and c_i that can satisfy Equations 5 and 6. To find these values, NSH performs the following procedure:

1. Choose m pivots.
2. Convert all data items using a multi-pivoted transformation in Definition 6, and obtain N number of m-dimensional transformed items.
3. Generate a vector w_1 and a bias term c_1 using an $(m+1)$-dimensional Gaussian distribution.
4. Adjust the vector w_1 and the bias term c_1 so that the resulting hash bits satisfy Equation 5.
5. For each $i = 2$ to b (hashcode length),
   (a) Generate a vector w_i and a bias term c_i from an $(m+1)$-dimensional Gaussian distribution.
   (b) Adjust w_i and the bias term c_i so that the resulting hash bits h_i^* satisfy Equations 5 and 6 with respect to the already generated hash bits h_j^* for $j = 1,\ldots,i-1$.
6. Collect w_i and c_i for $i = 1,\ldots,b$, which compose our hash function of length b.

A natural question is how to adjust the random vectors w_i and compute the bias terms so that the hash bits follow Equations 5 and 6. For this purpose, we maintain another series of $(m+1)$-dimensional vectors z_j where $j = 1,\ldots,b$. When we generate w_i, the set of vectors z_j for $j = 1,\ldots i$ work as a basis to which w_i must be orthogonal.[6] From now on, we think w_i for $i = 1,\ldots,b$ is $(m+1)$-dimensional vectors including the bias term in its last element. Let F denote a N-by-$(m+1)$ *design matrix*, for which the rows are the transformed data items $(f(v_1),\ldots,f(v_N))$ and the number of the columns is the number of the pivots plus one (the last column is one-padded to be multiplied with the bias component of w_i). Then the collection of i-th hash bits can be expressed compactly as follows: $h_i^* = \text{sign}(Fw_i)$.

Algorithm 1: Neighbor Sensitive Hashing

input : $V = \{v_1,\ldots,v_N\}$, N data items
b, code length
η, a parameter for coodinate transformation
output: W, a $(m+1)$-by-b coefficient matrix

```
P ← m pivots
F ← transform(V, P, η)                          // Definition 6
W ← [ ]
Z ← F^T 1/norm(F^T 1)
for k = 1 to b do
    w ← random (m+1)-by-1 vector
    w ← w − ZZ^T w
    z ← F^T sign(Fw)
    z ← z − ZZ^T z
    Z ← [Z, z/norm(z)]                          // append as a new column
end
return W
```

When we compute the coefficient w_1 for the first bit function, h_1^* must be orthogonal to $\mathbb{1}$ according to Equation 5. As a result, when generating w_1 for the first hash bits, we aim to satisfy the following expression: $\text{sign}(Fw_1)^T\mathbb{1} = 0$. We relax this expression for efficient computation as follows: $w_1^T F^T\mathbb{1} = 0$. From this expression, we can easily see that z_1 can be set to $F^T\mathbb{1}/\text{norm}(F^T\mathbb{1})$, then w_1 is obtained by first generating a random vector l and subtracting the inner product of l and z_1 from l

When we compute the coefficient vector w_2 for the second bit function, it should satisfy the following two conditions according to Equations 5 and 6:

$$\text{sign}(Fw_2)^T\mathbb{1} = 0, \qquad \text{sign}(Fw_2)^T h_1^* = 0.$$

For computational efficiency, these conditions are relaxed as:

$$w_2^T F^T\mathbb{1} = 0, \qquad w_2^T F^T h_1^* = 0.$$

We can simply ignore the first requirement among the two because z_1 already holds the necessary information. For the second requirement, we set z_2 to $F^T h_1^* - (F^T h_1^*)^T z_1$ and normalize it, which is the component of $F^T h_1^*$ that is orthogonal to z_1. With those two vectors of z_1 and z_2, the process of finding w_2 is as straightforward as before: generate a random vector, project the vector onto the subspace spanned by z_1 and z_2, and subtract the projected component from the random vector. Computing other coefficients w_i for $i = 3,\ldots,b$ can be performed in the same way. Our algorithm is presented in more detail in Algorithm 1.

The resulting time complexity of the process is $O(Nmd + b(mb + Nm))$, which is linear with respect to the database size. We have empirical runtime analysis in Section 4.5.

Number of Pivots (m) and Pivot Selection Strategy— Using a large number of pivots helps keep the ratio of η to the maximum distance small, which is one of the conditions for Lemma 2. However, in practice, we observed that increasing the number of pivots

[6] More concretely, w_1 must be orthogonal to z_1, and w_2 must be orthogonal both to z_1 and z_2, and so on.

beyond b (where b is the length of hashcodes) only marginally improved the search accuracy. This is shown in Figure 9(b). On the other hand, the technical conditions in Equations 5 and 6 and the time complexity analysis above imply important criteria for determining the number of pivots (m):

1. m must be equal to or larger than the hashcode length (b).
2. The smaller the m, the faster the hashing computation.

For these reasons, we recommend $m = c \cdot b$, where c is a small positive integer, e.g., $1, \ldots, 10$. To obtain a value of m that is well tailored to a given dataset, one can additionally employ a standard *cross validation* procedure that is widely used in machine learning literature. For this, we should first partition our original dataset into two, which are called training set and holdout set, respectively. Next, we generate a b-bit hash function with m pivots based on the training set, and test the performance of the generated hash function by using the holdout set as our queries. This procedure is repeated with different values of m, and the value yielding the highest search accuracy on the holdout set is chosen for the actual hashcode generation process.

Once the number of pivots is determined, we need to generate these pivots. We consider three different strategies for this:

1. Uniform strategy: Given the min and max coordinate of existing data items along each dimension, determine the respective coordinates of the pivots by picking m values from that interval uniformly at random.
2. Random strategy: Pick m data items from the database at random, and use them as pivots.
3. k-means strategy: Run the k-means++ algorithm on the existing items, and use the resulting centroids as pivots.

In Section 4.6, we study how these pivot selection strategies produce different search accuracies for different query workloads.

Impact of the Data Distribution— Unlike traditional hashing algorithms such as LSH [5,11,13,17], different data distributions lead to different hash functions in our approach. This effect is due to the pivot selection process; once the m pivots are chosen, the remaining steps of our algorithm are agnostic to the data distribution.

The random and k-means strategies tend to choose more pivots in the dense areas. Thus, the locations of the selected pivots are balanced for a balanced dataset, and are skewed for a skewed dataset. In contrast, the uniform strategy is not affected by the skewness of the data distribution, Rather, it is only affected by the range of the data items (i.e., the boundary items).

Our search results are more accurate when there are data items around queries. This is because our algorithm is more effective when there is a pivot close to each query, and we select pivots from areas where data items exist. Thus, when the random or k-means strategy is used, our algorithm is more effective when queries are mostly from the dense areas. When the uniform strategy is used, our algorithm is effective when queries are generated *uniformly* at random within the data items' boundary.

On the other hand, since our algorithm is based on a q-(η_{min}, η_{max})-sensitive transformation, we may not be successful at retrieving all k items when there are fewer than k items within a η_{max} distance of the query. We empirically study this observation in Section 4.6.

Finally, note that even for uniformly distributed items, our proposed approach outperforms traditional counterparts for kNN tasks. This is because the core benefit of our approach lies in its greater distinguishability for nearby data items, which is still valid for uniform distributions. Our empirical studies in Section 4.3 confirm that our technique outperforms not only LSH but also other state-of-the-art learning-based approaches even for uniform datasets.

Dataset	# Items	Dim	Note
`MNIST` [54]	69,000	784	Bitmap datasets
`80M Tiny` [54]	79,301,017	384	GIST image descriptors
`SIFT` [25]	50,000,000	128	SIFT image descriptors
`LargeUniform`	1,000,000	10	Standard uniform dist.[8]
`SmallUniform`	10,000	10	Standard uniform dist.
`Gaussian`	10,000	10	Standard normal dist.[9]
`LogNormal`	10,000	10	A log-normal dist.[10]
`Island`	10,000	10	`SmallUniform` + two clusters. See Section 4.6.

Table 1: Dataset Summary. Three real and five synthetic datasets in order. For each dataset, 1,000 data items were held out as queries.

4. EXPERIMENTS

The empirical studies in this section have two following goals: first, we aim to verify our claim (more hash bits for neighbor items) with numerical analysis, and second, we aim to show the superiority of our algorithm compared to various existing techniques. The results of this section include the following:

1. *Neighbor-Sensitive Transformation* enlarges the distances between close by data items, and the same goal is achieved for the hashcodes generated by *Neighbor-Sensitive Hashing* in terms of their Hamming distance.
2. Our hashing algorithm was robust for all settings we tested and showed superior performance in kNN tasks. Specifically, our method achieved the following improvements:
   (a) Up to 15.6% recall improvement[7] for the same hashcode length,
   (b) Up to 22.5% time reduction for the same target recall.

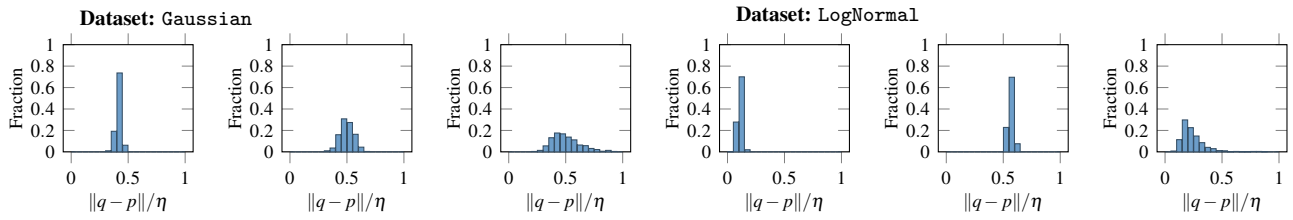
We start to describe our experimental results after stating our evaluation settings.

4.1 Setup

Datasets and Existing Methods— For numerical studies and comparative evaluations, we use three real image datasets and five synthetic datasets. A *database* means a collection of data items from which the k most similar items must be identified, and a *query set* means a set of query items we use to test the search performance. As in the general search setting, the query set does not belong to the *database* and is not known in advance; thus, offline computation of kNN is impossible. Table 1 summarizes our datasets.

For a comprehensive evaluation, we compared against three well-known approaches and five recent proposals: *Locality Sensitive Hashing* (LSH) [13], *Spectral Hashing* (SH) [56], *Anchor Graph Hashing* (AGH) [38], *Spherical Hashing* (SpH) [21], *Compressed Hashing* (CH) [36], *Complementary Projection Hashing* (CPH) [26], *Data Sensitive Hashing* (DSH) [16], and *Kernelized Supervised Hashing* (KSH) [37]. Section 5 describes the motivation behind each approach. Except for CH and LSH, we used the source code provided by the authors. We did our best to follow the parameter settings described in their papers. We exclude a few other works that assume different settings for kNN item definitions [22]. For our algorithm, we set the number of pivots (m) to $4b$ and used k-means++ [6] to generate the pivots unless otherwise

[7]Recall improvement is computed as (NSH's recall - competing method's recall).
[8]For each dimension, the standard uniform distribution draws a value between 0 and 1, uniformly at random.
[9]For each dimension, a value is drawn from a Gaussian distribution with a mean value of 0 and a standard deviation of 1.
[10]Log-normal is a popular heavy-tailed distribution. In our experiments, we used $(\mu, \sigma) = (1, 0)$.



(a) Dense Area (b) Sparse Area (c) Same Dist as Data (d) Dense Area (e) Sparse Area (f) Same Dist as Data

Figure 5: The histogram of distances between queries q and their respective closest pivots p divided by η (the parameter from Definition 5). These histograms are computed for a balanced and a skewed dataset under three query distributions: queries drawn from the dense area of the dataset, from its sparse area, and from the same distribution as the dataset itself. These figures show that, in most cases, the distances between novel queries and their respective closest pivot satisfy the condition of Lemma 2 for NSH to work effectively, namely $\|q-p\| < \eta/2$.

mentioned. The value of η was set to 1.9 times of the average distance from a pivot to its closest pivot. (Section 4.6 provides an empirical analysis of different pivot selection strategies and different values of m and η parameters.) All our time measurements were performed on a machine with AMD Opteron processor (2.8GHz) and 512GB of memory. All hashing algorithms used 10 processors in parallel.

Quality Metric— Recall that Hamming Search is the component responsible for searching in the Hamming space. Once the Hamming Search component returns r candidate items, finding the k most similar items to a query is a straightforward process. Note that if there exists a data item that belongs to the *true* kNN among those r items returned, the data item is always included in the answer set returned by Re-rank. Therefore, a natural way to evaluate the quality of the entire system is to measure the fraction of the data items that belong to the true kNN among the r data items returned by Hamming Search. In other words, we compute the following quantity:

$$recall(k)@r = \frac{(\#\text{ of } \textit{true}\ k\text{NN in the retrieved})}{k} \times 100. \quad (7)$$

This is one of the most common metrics for evaluating approximate kNN systems [24, 25, 46]. When choosing the r data items with the smallest Hamming distance from a query, it is possible for multiple items to have the same distance at the decision boundary. In such case, a subset of them are chosen randomly. This implies that the recall score of a trivial approach i.e., mapping all the data items to the same hashcodes, cannot be high. Typically, r is chosen as $r = 10k$ or $r = 100k$ to keep the Re-rank speed fast.[11]

Evaluation Methodology— Note that the choice of the Hash Function is independent of the Hamming Search component, and using a different algorithm for Hamming Search can only affect the runtime. Thus, we consider two evaluation settings in this paper.

1. Hashcode Length and Recall: This setting is to purely evaluate the quality of hashing algorithms without any effects from other components. This evaluation setting is to answer the following question: "what is the best hashing algorithm if the search speed is identical given the same hashcode size?" This evaluation is repeated for different choices of hashcode sizes, because the accuracy of different hashing algorithms can differ greatly based on their hashcode size. For example, it is not uncommon if some methods become less effective as the hashcode size increases.
2. Search Speed and Recall: Another evaluation methodology is to study the trade-off between search speed and resulting search accuracy. The search time consists of the time required to convert a query into a hashcode and the time to find the rNN data items in the Hamming space. Usually, the time required for hashcode generation is marginal compared to the Hamming Search process.

Another important criteria is the memory requirement for the generated hashcodes. This quantity can be easily inferred from the hashcode size because the size of the bit vectors in memory is identical regardless of the hashing algorithm used. When there are N data items in a database and we generate hashcodes of length b, the amount of memory required to store all hashcodes is $Nb/8$ bytes (since hashcodes are stored as bit vectors). Usually, this quantity is several orders of magnitude smaller than the size of the original database (e.g., 128-dimensional floating point type vectors will take $128 \cdot 4 \cdot N$ bytes), making it possible to keep all the hashcodes in memory.

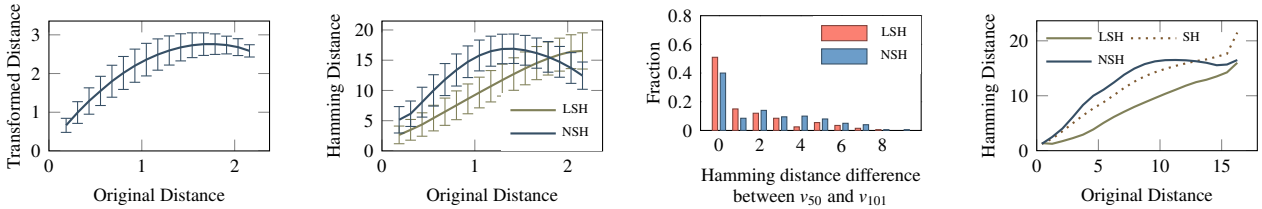
4.2 Validating Our Main Claims

In this section, we numerically verify two important claims we have made: (i) the distance between novel queries and their closest pivot is small, and (ii) NST and NSH achieve their intended goal of placing more separators between closeby items.

First, to study how data items are mapped to pivots, we used `Gaussian` and `LogNormal` as representatives of balanced and skewed datasets, respectively. For each dataset, we considered three different cases: (1) when queries are chosen from the dense area of the dataset, (2) when they are drawn from the sparse area (i.e., outlier items), and (3) when queries come from the same distribution as the original dataset. In each case, we selected 128 pivots using the k-means strategy and measured the distances between queries and their respective closest pivot. Figure 5 shows the histogram of these distances. The results show that, for queries from the dense area and from the same distribution as the datasets, the measured distances mostly belong to the range with which NSH can work effectively, i.e., smaller than $\eta/2$. For queries from the sparse area, there were some cases where the measured distances were outside the desired range. This is expected since our pivot selection strategy assumes that queries are from an area where existing data items reside. Interestingly, as shown in Section 4.6, our final hashing algorithm still provides reasonable performance even for those queries that are drawn from sparse areas.

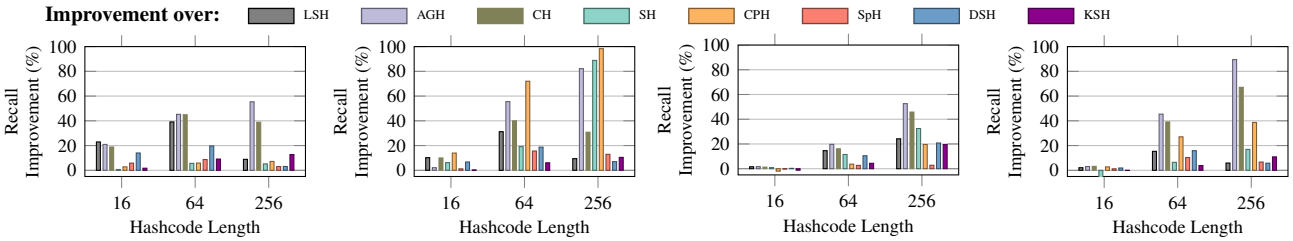
We also studied whether our proposed mechanisms (NST and NSH) achieve their intended properties, namely increasing the distance gap exclusively for close-by items. Here, we first transformed the data items of the `SmallUniform` dataset using 128 pivots to see how our multi-pivoted transformation (from Definition 6) alters the distances between items. Figure 6(a) shows the result. When the original distances between two items were smaller than the average

[11] Larger values of r (e.g., close to the total number of items in the database) will improve recall; however, this will also make the search process as slow as the exact kNN.



(a) NST's effect for the `SmallUniform` dataset (b) Hamming distance for the `SmallUniform` dataset (c) Hamming gap comparison for the `SmallUniform` dataset (d) Hamming distance for the `MNIST` dataset

Figure 6: The effects of NST and NSH. Figure (a) shows that NST enlarges the distances among nearby data items. Figure (b) shows that NSH makes nearby data items have larger Hamming distances compared to LSH. Figure (c) shows that there are more separators (hence, a larger Hamming distance gap) between pairs of data items when they are close to queries. Figure (d) shows using a real dataset (`MNIST`) that NSH produces larger Hamming distances between nearby data items compared to SH (a learning-based algorithm) and LSH.



(a) The `MNIST` dataset (b) The `LargeUniform` dataset (c) The `80M Tiny` dataset (d) The `SIFT` dataset

Figure 7: Hashcode length and recall improvements. The recall improvement is computed as *(NSH's recall - competing method's recall)*.

distance between a query and its k-th closest item (e.g., 0.53 for 10-NN, 0.71 for 100-NN, and 0.97 for 1000-NN), their distances were amplified by NST in the transformed space. When we generated 32-bit hashcodes using NSH and LSH, NSH also produced larger Hamming distances (compared to its traditional counterpart, LSH), as reported in Figure 6(b). Figure 6(c) depicts the same effect by NSH, but using a histogram of the number of separators between the 50th and 101st closest data items to queries. Figure 6(d) shows NSH's effectiveness using a real dataset (`MNIST`). Here, NSH again achieved larger Hamming distance gaps than both SH (a learning-based hashing algorithm) and LSH. Note that while the difference between NSH and SH may seem small, it translates to a significant difference in search performance (see Sections 4.3 and 4.4).

4.3 Hashcode Length and Search Accuracy

Recall that the hashcode length (b) is an important design parameter that determines the accuracy of approximate kNN and runtime of Hamming Search. In general, those two factors (search accuracy and runtime) are in a trade-off relationship, i.e., larger hashcodes result in a more accurate but also slower search, and vice versa.

This subsection compares the search accuracies of various hashing algorithms with fixed hashcode lengths. For this experiment, we used the four datasets (`MNIST`, `LargeUniform`, `80M Tiny`, and `SIFT`) and generated different lengths of hashcodes ranging from 16 to 256. Next, we examined how accurately different algorithms capture 10-NN data items for novel queries. For this, we report $recall(10)@100$ for the two relatively small datasets (`MNIST` and `LargeUniform`) and $recall(10)@1000$ for the other two large datasets. We present the experimental results for different choices of k in Section 4.7.

Figure 7 shows the results. we report the recall improvements over other competing hashing algorithms. In most cases, the second best methods were SpH and KSH. However, the other recently developed algorithms (such as SH and CPH) worked relatively well too. AGH and CH showed surprisingly bad performance. In all cases, our proposed algorithm showed significant search accuracy gains, showing up to 15.6% improvement of recall over SpH and up to 39.1% over LSH.

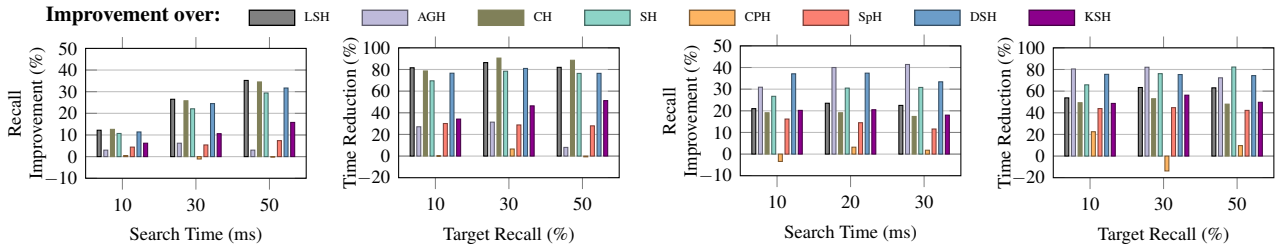
4.4 Search Time and Search Accuracy

The second setting for performance evaluation is seeing the recall scores by different hashing algorithms when the search time is bounded. For the Hamming Search module, we used *Multi-Index Hashing* (MIH) [42], a recently developed data structure for exact kNN search in Hamming space. MIH has a parameter that determines the number of internal tables, and the search speed varies depending on the parameter setting. We followed a few different options based on the suggestions by its author, and reported the best results for each hashing algorithm.[12]

There are two ways we can improve the search accuracy at the cost of search speed. The first is to increase the hashcode length, and the second is to increase the number of data items returned by Hamming Search (r) and let Re-rank find the k most similar answers. When we tested the first approach, however, we observed that MIH's performance degrades rapidly whenever the hashcode length is over 128, and MIH did not show considerable speed boost compared to a linear scan over hashcodes. For this reason, we used the second approach — increasing the value of r — to adjust the search accuracy and search speed. Then, we collected all 64-bit hashcodes generated by different hashing algorithms, configured MIH to return different number (r) of data items as answers, and measured the recall scores of those answers as well as the time MIH took to return them. Note that even if we use the same data structure (MIH) for Hamming Search, systems with different hashing algorithms produce very different results since the hashing mechanism is key in producing high search accuracies.

Figure 8 reports recall improvements for a target time bound using two large datasets of `80M Tiny` and `SIFT`. In most cases, NSH showed significant improvements over existing methods. Also, it

[12] We set the number of tables to either 2 or 3.



(a) The 80M Tiny dataset

(b) The SIFT dataset

Figure 8: Search time and recall improvements. The recall improvement is computed as *(NSH's recall - competing method's recall)*. Time reduction is *(competing method's search time - NSH's search time) / (competing method's search time)* $\times 100$.

Method	Hash Gen (sec)		Compression (min)	
	32bit	64bit	32bit	64bit
LSH	0.38	0.29	22	23
SH	28	36	54	154
AGH	786	873	105	95
SpH	397	875	18	23
CH	483	599	265	266
CPH	34,371	63,398	85	105
DSH	3.14	1.48	24	23
KSH	2,028	3,502	24	29
NSH (Ours)	231	284	37	46

Table 2: Time requirement for hash function generation and database compression, i.e., converting 79 million data items in a database to hashcodes.

is impressive that the system with our algorithm achieved as high as 50% average recall for 80M Tiny within only 49 ms search time requirement. Note that a simple linear scan over the original data items took more than 17 seconds.

4.5 Indexing Speed

As we generate hashcodes of different lengths in the above experiments, we also measured the times they took to generate hash functions and to convert the whole database (80M Tiny) to hashcodes. The result is summarized in Table 2. CPH uses expensive statistical optimization to generate hash functions, so it took a much longer time than other methods. All other methods, including NSH, reported reasonable indexing time.

4.6 The Effect of Parameters on NSH

This section studies the effect of various parameters on the search accuracy of NSH. The parameters we consider are pivoting (the number of pivots and the selection strategy), the neighborhood parameter (η), the type of query workloads, and the data distribution.

Pivot Selection— As discussed after presenting Definition 6, the goal of choosing pivots is to ensure that the average distance of every data item in the database to its closest pivot is minimized. We studied the three different strategies described in Section 3.4: uniform strategy, random strategy, and k-means. For each strategy, we generated 32-bit hashcodes and used the Gaussian dataset with two different sets of queries: one set from the dense area (center of the normal distribution) and the other from the sparse area (tail of the distribution). Figure 9(a) shows the results. Both uniform and k-means strategies produced almost the same search accuracy, regardless of the query workload. However, the random strategy failed for queries from the spare area. This is because most of the randomly-chosen pivots are naturally from dense areas; thus, the pivots cannot cover the queries from sparse areas. Also, in our experiments, k-means strategy exhibited slightly better performance than the uniform one.

The number of pivots (m) is also important. To empirically study the effect of m on search accuracy, we generated 32-bit hashcodes for three datasets (Gaussian, SmallUniform, and LogNormal) and varied m between 3 and 512. Figure 9(b) shows the results. The results suggest that our algorithm is not very sensitive to the value of m, as long as $m \geq b$.

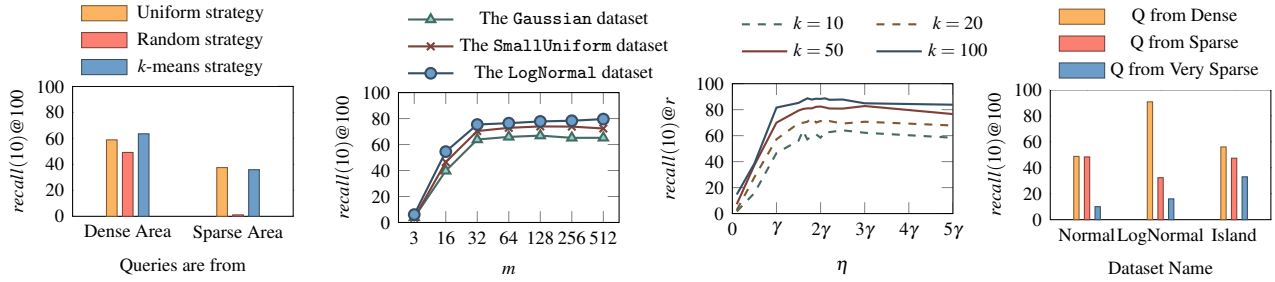
Neighborhood Parameter (η)— The value of η is closely related to the size of the neighborhood for which we want to amplify the Hamming gap. To see η's effect on our algorithm's performance, we generated 32-bit hashcodes for the MNIST dataset and measured the recall while varying η between 0 and 5γ, where γ was the average distance between pairs of closest pivots. The results, plotted in Figure 9(c), indicate that our algorithm yields high accuracy when $\eta > \gamma$, and its accuracy curve improves unto $\eta \approx 2\gamma$. Note that this empirical result is consistent with our discussion in Section 3.3.

Data Distribution— To study the effect of data distribution, we generated two datasets from standard distributions: Gaussian and LogNormal. Note that LogNormal has a skewed distribution with a heavy tail. In addition, to see how our kNN search accuracy is affected when there are fewer than k data items around a query, we created another dataset, called Island. In Island, we added two small clusters to the SmallUniform dataset, where each clusters consisted of 3 data items, and they were placed far away (a distance of 1 and 5, respectively) from other data items.

For each dataset, we generated three different queries. For the Gaussian dataset, the first query was from its mode.[13] The second and the third queries were twice and three times the standard deviation away from the mode, respectively. The three queries were similarly generated for LogNormal. For Island, the first query was from the area where the SmallUniform dataset resides, and the second and the third queries were from the two small clusters we added. Due to the placements of the two clusters, the third query was much far away from the other data items compared to the second query. For every dataset, we refer to these three queries as 'Q from Dense', 'Q from Sparse', and 'Q from Very Sparse', respectively.

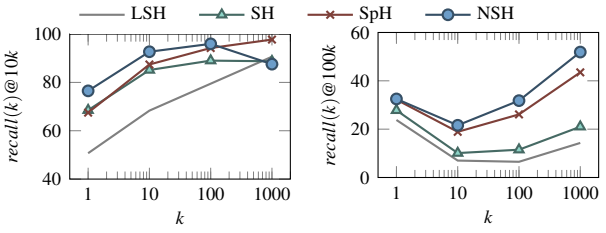
We repeated each experiment 30 times, each time with a different random seed, and reported the average recall scores in Figure 9(d). In general, the performance drops were more significant for queries drawn from 'Very Sparse' areas. One reason is the lack of nearby pivots around such queries. The second reason (especially for the Island dataset) is that the distance to kNN items were outside the neighborhood size for which NSH can work effectively.

[13] A mode is the point at which a probability distribution function takes its maximum.



(a) Pivoting Strategy (b) Numbers of Pivots (c) Neighborhood Size (d) Data distribution

Figure 9: We study our method's search performance by varying four important parameters: (a) the pivot selection strategy, (b) the number of pivots, (c) neighborhood parameter η, and (d) the data distribution.



(a) The MNIST dataset (b) The 80M Tiny dataset

Figure 10: kNN Accuracies with different values of k.

Method	Year	Motivation / Intuition
LSH [13]	2004	Random hyperplanes tend to preserve locality
SH [56]	2009	Minimize Hamming distance in proportion to the similarity between items
AGH [38]	2011	Speed up SH by approximation
SpH [21]	2012	Spheres for capturing similarities
CH [36]	2013	Adopt sparse coding theory
CPH [26]	2013	Hash functions should go through sparse areas
DSH [16]	2014	Keep kNN items together using Adaptive Boosting

Table 3: Several Notable Hashing Algorithms.

4.7 Neighbor Sensitivity

Since our motivation was moving separators (or equivalently, bit functions) to have higher power in distinguishing neighbor items, it is likely that our algorithm loses its power to capture the similarities to the distant items. This potential concern leads to a natural question: up to what value of k does our algorithm have advantage over other algorithms?

To answer this question, we varied the value of k while fixing $r = 10 \cdot k$ (Recall r is the number of the items returned by Hamming Search, and Re-rank module returns final k answers) and observed how the search accuracy changed. More concretely, we generated 128-bit and 64-bit hashcodes respectively for MNIST and 80M Tiny, and varied k from 1 to 1,000. See Figure 10 for the results.

Interestingly, for MNIST, we observed that our algorithm outperformed other methods only up to until $k = 100$ (0.14% of the database). The reason that our method showed superior performance only up to $k = 100$ was that due to the small size of the dataset (only 69K items in the database). When we ran the same experiment with a big dataset (80M Tiny), we did not observe the performance decrease until $k = 1000$, and our algorithm consistently outperformed other methods regardless of the choice of k. Considering that the dataset sizes in the real-world are large — the reason of approximate kNN— our algorithm can achieve superior performance in most practical cases.

5. RELATED WORK

The growing market for 'Big Data' and interactive response times has created substantial interest in Approximate Query Processing both from academia [3,14,43,59] as well as the commercial world [1, 2, 4]. While these techniques focus on general aggregate queries, this work focuses on approximating kNN queries as an important sub-class of them (see [41] and the references within).

Gionis *et al.* [17] were the first to apply Locality Sensitive Hashing to approximate search problems. In their work, unary representation of integers were used as hash functions to generate binary hash codes. Later, Charikar [11] proposed to use random hyperplanes as hash functions. These random hyperplanes were drawn from multi-dimensional Gaussian distributions, and generated hashcodes that could retain the locality sensitive property for cosine similarity. Datar *et al.* [13] proposed a variant of random hyperplane method for the Euclidean distance. Athitsos *et al.* [8] employed L1-embedding for a similar purpose. Distance-based Hashing [9] generalizes this technique to non-Euclidean distances.

Recent work in this area, however, has started to exploit statistical optimization techniques to learn more intelligent hash functions. These techniques, known as *learning-based* or *data-dependent* hashing, take the distribution of data into account in order to obtain more effective hash functions. A notable approach in this category is Spectral Hashing [56], which motivated others including Binary Reconstructive Embedding [32], Anchor Graph Hashing [38], Random Maximum Margin Hashing [27], Spherical Hashing [21], Compressed Hashing [36], Complementary Projection Hashing [26], and Data Sensitive Hashing [16]. All these methods use different motivations to learn more accurate hashcodes for the kNN task. See Table 3 for a summary.

There are several techniques developed for efficient indexing and querying of the hashcodes generated by LSH [15,39,47,51,53,58]. As explained in Section 2.1 these methods belong to the Hamming Search stage; thus, they are orthogonal to our contribution. In our implementation, we employed *Multi-Index Hashing* (MIH) [42], as the state-of-the-art in this area.

Finally, it is important to note that using alternative representations of the original data points for hashing is not a new topic. Different approaches have used different representations to achieve their own motivations. For instance, AGH [38] used one to speed-up SH [56], and CH [36] used one to obtain sparse representations. When we used their representations in place of ours, the resulting

algorithm produced a much worse performance. Finally, in machine learning, non-linear transformations have also been used for learning distance metrics for kNN classifiers [29, 40].

6. CONCLUSION AND FUTURE WORK

We have proposed *Neighbor-Sensitive Hashing*, a mechanism for improving approximate kNN search based on an unconventional observation that magnifying the Hamming distances among neighbors helps in their accurate retrieval. We have formally proven the effectiveness of this novel strategy. We have also shown empirically that NSH yields better recall than its state-of-the-art counterparts given the same number of hash bits. NSH is a "drop-in replacement" for existing hashing methods. As a result, any application that chooses NSH can either enjoy an improved search quality, or trade NSH's recall advantage for a significant speed-up (i.e., reduced time and memory footprint by using shorter hashcodes).

We have several goals for future work. First, we aim to build a highly-parallel in-memory database system that supports extremely fast kNN operations. Second, we plan to develop a hashing technique that can handle more generic distance metrics including cosine-similarity and earth mover's distance.

7. ACKNOWLEDGEMENTS

We would like to thank Jia Deng and Suchee Shah for their helpful feedback and suggestions. This work was funded by NSF grants IIS-1064606 and ACI-1342076.

8. REFERENCES

[1] Presto: Distributed SQL query engine for big data. `https://prestodb.io/docs/current/release/release-0.61.html`.
[2] SnappyData. `http://www.snappydata.io/`.
[3] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *SIGMOD*, 2014.
[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.
[5] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, 2006.
[6] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *SODA*, 2007.
[7] I. Assent, R. Krieger, F. Afschari, and T. Seidl. The ts-tree: efficient time series search and retrieval. In *EDBT*, 2008.
[8] V. Athitsos, M. Hadjieleftheriou, G. Kollios, and S. Sclaroff. Query-sensitive embeddings. *TODS*, 2007.
[9] V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios. Nearest neighbor retrieval using distance-based hashing. In *ICDE*, 2008.
[10] O. Boiman, E. Shechtman, and M. Irani. In defense of nearest-neighbor based image classification. In *CVPR*, 2008.
[11] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.
[12] B. Cui, B. C. Coi, J. Su, and K.-L. Tan. Indexing high-dimensional data for efficient in-memory similarity search. *TKDE*, 2005.
[13] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, 2004.
[14] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-charging estimate convergence in dbo. *PVLDB*, 2(1), 2009.
[15] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, 2012.
[16] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. Dsh: data sensitive hashing for high-dimensional k-nnsearch. In *SIGMOD*, 2014.
[17] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, 1999.
[18] Y. Gong and S. Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *CVPR*, 2011.
[19] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.
[20] J. He, R. Radhakrishnan, S.-F. Chang, and C. Bauer. Compact hashing with joint optimization of search accuracy and time. In *CVPR*, 2011.
[21] J.-P. Heo, Y. Lee, J. He, S.-F. Chang, and S.-E. Yoon. Spherical hashing. In *CVPR*, 2012.
[22] G. Irie, Z. Li, X.-M. Wu, and S.-F. Chang. Locally linear hashing for extracting non-linear manifolds. In *CVPR*, 2014.
[23] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *TODS*, 2005.
[24] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAM*, 2011.
[25] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *ICASSP*, 2011.
[26] Z. Jin, Y. Hu, Y. Lin, D. Zhang, S. Lin, D. Cai, and X. Li. Complementary projection hashing. In *ICCV*, 2013.
[27] A. Joly and O. Buisson. Random maximum margin hashing. In *CVPR*, 2011.
[28] S. Kashyap and P. Karras. Scalable knn search on vertically stored time series. In *SIGKDD*, 2011.
[29] D. Kedem, S. Tyree, F. Sha, G. R. Lanckriet, and K. Q. Weinberger. Non-linear metric learning. In *NIPS*, 2012.
[30] D. Keysers, C. Gollan, and H. Ney. Local context in non-linear deformation models for handwritten character recognition. In *ICPR*, 2004.
[31] Y. Koren and R. Bell. Advances in collaborative filtering. In *Recommender Systems Handbook*. 2011.
[32] B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. In *NIPS*, 2009.
[33] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing. *TPAM*, 2012.
[34] M. S. Lew, N. Sebe, C. Djeraba, and R. Jain. Content-based multimedia information retrieval: State of the art and challenges. *TOMCCAP*, 2006.
[35] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The tv-tree: An index structure for high-dimensional data. *The VLDB Journal*, 1994.
[36] Y. Lin, R. Jin, D. Cai, S. Yan, and X. Li. Compressed hashing. In *CVPR*, 2013.
[37] W. Liu, J. Wang, R. Ji, Y.-G. Jiang, and S.-F. Chang. Supervised hashing with kernels. In *CVPR*, 2012.
[38] W. Liu, J. Wang, S. Kumar, and S.-F. Chang. Hashing with graphs. In *ICML*, 2011.
[39] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.
[40] R. Min, D. Stanley, Z. Yuan, A. Bonner, Z. Zhang, et al. A deep non-linear feature mapping for large-margin knn classification. In *ICDM*, 2009.
[41] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Engineering Bulletin*, 2015.
[42] M. Norouzi, A. Punjani, and D. J. Fleet. Fast search in hamming space with multi-index hashing. In *CVPR*, 2012.
[43] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4, 2011.
[44] Y. Park. Supplementary material for neighbor-sensitive hashing. `http://www-personal.umich.edu/~pyongjoo/vldb2016sup.pdf`.
[45] R. Salakhutdinov and G. Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 2009.
[46] H. Sandhawalia and H. Jégou. Searching with expectations. In *ICASSP*, 2010.
[47] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 2012.
[48] J. B. Schafer, D. Frankowski, J. Herlocker, and S. Sen. Collaborative filtering recommender systems. In *The adaptive web*. 2007.
[49] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *ICCV*, 2003.
[50] A. W. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain. Content-based image retrieval at the end of the early years. *TPAM*, 2000.
[51] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. Srs: Solving c-approximate nearest neighbor queries in high dimensional euclidean space with. *PVLDB*, 2014.
[52] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *PVLDB*, 2013.
[53] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, 2009.
[54] A. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *TPAM*, 2008.
[55] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.
[56] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, 2009.
[57] H. Xu, J. Wang, Z. Li, G. Zeng, S. Li, and N. Yu. Complementary hashing for approximate nearest neighbor search. In *ICCV*, 2011.
[58] C. Yu, B. C. Ooi, K.-L. Tan, and H. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB*, 2001.
[59] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *SIGMOD*, 2014.
[60] H. Zhang, A. C. Berg, M. Maire, and J. Malik. Svm-knn: Discriminative nearest neighbor classification for visual category recognition. In *CVPR*, 2006.