Efficient Nearest Neighbor Search by Removing Anti-hub

Kimihiro Tanaka
The University of Tokyo

Yusuke Matsui
The University of Tokyo
matsui@hal.t.u-tokyo.ac.jp

Shin'ichi Satoh
National Institute of Informatics
satoh@nii.ac.jp

ABSTRACT

The central research question of the nearest neighbor search is how to reduce the memory cost while maintaining its accuracy. Instead of compressing each vector as is done in the existing methods, we propose a way to subsample unnecessary vectors to save memory. We empirically found that such unnecessary vectors have low hubness scores and thus can be easily identified beforehand. Such points are called *anti-hubs* in the data mining community. By removing anti-hubs, we achieved a memory-efficient search while preserving accuracy. In million-scale experiments, we showed that any vector compression method improves search accuracy by partial replacement with anti-hub removal under the same memory usage. A billion-scale benchmark showed that our data reduction combined with the best search method achieves higher accuracy under the assumption of fixed memory consumption. For example, our method had a much higher recall@100 (0.53) compared with the existing method (0.23) for the same memory consumption (6GB).

CCS CONCEPTS

• **Information systems** → **Retrieval efficiency**.

KEYWORDS

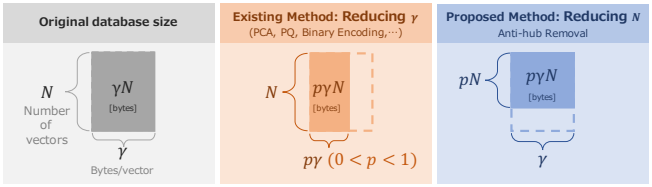
nearest neighbor search, memory efficiency, hubness

ACM Reference Format:
Kimihiro Tanaka, Yusuke Matsui, and Shin'ichi Satoh. 2021. Efficient Nearest Neighbor Search by Removing Anti-hub. In *Proceedings of the 2021 International Conference on Multimedia Retrieval (ICMR '21), August 21–24, 2021, Taipei, Taiwan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3460426.3463622

1 INTRODUCTION

A tremendously large amount of data is available on the Internet. Nearest neighbor search (NN search) is the algorithm to extract similar items to a given query from database vectors. NN search is a fundamental operation to interact with such large-scale data [12].

However, there are two severe problems in NN search. Firstly, the calculation cost is tremendous. NN search calculates the distance between the query and all database vectors. When a database is composed of N D-dimensional vectors, the dominant cost to find similar items is the computation of distances, $O(ND)$. If N is billion-scale, this cost is intractable. Secondly, the memory consumption is too massive. We usually need to store all of the database vectors on-memory to achieve a fast search. When each vector consumes γ bytes to be stored, totally γN bytes are needed for all vectors in the database. For example, the Deep1B dataset [39] consists of 10^9 96-dimensional vectors using 4-byte floats. This means that 384 GB of memory is required to store this dataset. Considering that typical personal computers have only 8-32 GB of memory, this size ($N = 10^9$, $\gamma = 384$) is an unacceptable value.



(a) Original Database (b) Existing Method (c) Proposed Method

Figure 1: Each data size of original database, an existing memory-saving method, and our proposed method.

For the high calculation cost problem, approximate nearest neighbor search algorithms (ANN) have been extensively studied, such as graph schemes [22, 23] and tree-based algorithms [3, 25]. However, these ANNs do not contribute to the problem of memory usage.

As solutions to the problem of memory usage (the usage of γN bytes), two approaches can be considered: 1) reduction of γ, the size of each vector, and 2) reduction of N, the number of vectors. Various approaches have been discussed for the former, including dimensionality reduction, product quantization [17], and binary encoding [38]. While these methods reduce the value of γ and make memory consumption much smaller, excessive compression naturally deteriorates the search accuracy.

This paper proposes the latter approach, which has not been discussed so much: reduction of N by subsampling database vectors. Under the assumption that the underlying distribution of the database vectors and the query vectors are similar, we found that some vectors have a low possibility to be the nearest neighbors for any query. We further observed that such unnecessary vectors do not contribute much to the overall search accuracy. To characterize such vectors **without queries (only with database vectors)**, we revisited an existing statistical measure, *hubness* [29, 30]. We empirically found that the unnecessary vectors have significantly lower hubness scores. Such low-hubness vectors are called anti-hubs in the data mining community [31]. By simply removing them from the database beforehand, we can save memory usage, maintaining its original search accuracy.

Fig. 1 visualizes the idea behind this approach. Fig. 1a shows the original database. To reduce its size (γN), we usually apply γ reduction methods as Fig. 1b. Here, we propose the reduction of

ICMR '21, August 21–24, 2021, Taipei, Taiwan

ACM ISBN 978-1-4503-8463-6/21/08…$15.00
https://doi.org/10.1145/3460426.3463622

N as Fig. 1c shows. These two approaches are similar in terms of information loss. For example, when γ is reduced by PCA [34], this means that the amount of information of each vector is decreased. On the other hand, when we remove some vectors from the original database, the information of eliminated vectors is reduced to zero while that of the remaining vectors is fully saved. In either case, we lose information to a certain extent in order to make the size of the database smaller. Therefore, this new approach of reducing N should be considered as well as the reduction of γ, and its validity should be judged by the effectiveness in the actual application to nearest neighbor search.

Besides hubness, we also tried basic outlier detection algorithms [20, 32] as the indicator of unnecessity in NN search, and showed the high performance of hubness. Also, we applied our proposed method to four million-scale datasets and compared it with various γ reduction methods by partially replacing it with our N reduction. This results in showing the superiority of our methods. Furthermore, we revealed that our method could be efficiently approximated, which enabled us to make a data reduction method for billion-scale datasets. Combined with IVFPQ + HNSW, which is the current best ANN framework for billion-scale data [2, 8], our data reduction method achieved a higher recall rate under the fixed memory consumption and the same search time on the Deep1B [39].

Our contributions are summarized as follows:

- We newly proposed a method that eliminates unnecessary vectors called anti-hubs from a database in nearest neighbor search to save the memory consumption.
- Our proposed method can be combined with any search algorithms because it is a preprocessing.
- We can use the proposed method with vector compression methods as a complementary memory-saving process, improving the search accuracy.
- Our approach can be applied even to billion-scale datasets by approximating the reduction process.

2 RELATED WORK

As mentioned, there are two approaches to reduce the memory usage, γN: 1) the reduction of γ, vector compression, and 2) the reduction of N, the removal of unnecessary vectors. This section introduces some γ reduction methods and outlier detection algorithms as another option for the detection of unnecessary data points in NN search.

2.1 Reduction of γ

The reduction of γ means the vector compression. In other words, it is to make the size of each vector smaller. We categorize vector compression into three methods: 1) dimensionality reduction, 2) product quantization, 3) binary encoding. In this section, we explain them in brief. In the experiment section (Sec. 5.6), we compared proposed anti-hub removal with these γ reduction algorithms.

Dimensionality Reduction. The most basic vector compression method is dimensionality reduction. Today there are some linear data reduction methods such as Principal Component Analysis (PCA) [34], Linear Discriminant Analysis (LDA) [37], and Canonical Correlation Analysis (CCA) [40]. In their methods, PCA [34] is the most fundamental dimensionality reduction algorithm. In PCA, the dimensionality of the data is reduced, maximizing the variance of the data for the retention of the representation ability. When PCA reduces the dimensionality of 4-byte float vectors from d to $\frac{d}{4}$, e.g., the value of γ gets compressed from $4d$ bytes to d bytes.

While many nonlinear dimensionality reduction algorithms like Locally Linear Embedding (LLE)[33] or t-SNE [21] have been developed, they are not used to process large-scale datasets because of their high computation cost. In our experiments, we used PCA as the dimensionality reduction method.

Product Quantization. Product quantization (PQ) [17] is a robust and straightforward method to compress vectors. In PQ, each vector is split into subvectors, and each subvector is quantized. Because each vector is quantized, the memory consumption gets so tiny compared with the original one. For example, let us assume a database consisting of 1,000,000,000 128-dimensional vectors (4-byte float). In this case, the value of γ of an original vector is 512 bytes. Hence, it requires 512GB of memory to store the whole database. After PQ with eight subvectors and an 8-bit quantizer is applied, however, the value of γ is surprisingly shortened to 8 bytes because each quantized subvector is represented as an 8-bit code. This means that the total memory cost is only 8GB.

Although PQ is extended in many ways [1, 9, 19, 24, 27, 41, 42], in our million-scale evaluation we used the original PQ implemented in the Faiss library [16]. In the billion-scale experiment, we used an extension of inverted file system with product quantization (IVFPQ) [2, 8]. IVFPQ framework is a combination of an inverted index and product quantization and the only option to handle billion-scale data efficiently.

Binary Encoding. The binary encoding methods convert a real-valued vector to a short binary vector [4, 6, 7, 14]. Because a supervised signal such as labels is not available in our setting, we make use of unsupervised approaches such as Locality-Sensitive Hashing (LSH) [15] and Iterative Quantization (ITQ) [10]. By binary encoding, e.g., a binarized vector of 64 bits consumes only $\gamma = 8$ bytes. Thanks to this compression, we can handle large-scale data more efficiently. As LSH, similar vectors are supposed to be encoded to the same hash values. Therefore we can calculate the distance between a query and database vectors by simple Hamming distance.

In our experiment, we used a random rotation + binarization algorithm [7] as a binary encoding algorithm.

2.2 Reduction of N

So far, reducing the number of vectors in a database has not been regarded as a method to save memory usage. This paper proposed the detection and removal of unnecessary vectors based on hubness [29, 30]. However, it is natural for us to associate outlier to unnecessity. Therefore, we introduce the essential outlier detection here and compare them with hubness later (Sec. 5.5).

Outlier Detection. Outliers in computer science are the data points that differ from other members of a group. Their detection (outlier detection) is widely used in many fields like financial markets, system diagnosis, user-action sequences [13].

A traditional outlier detection method is knn-based one [32]. In this method, the outlier score of each point is defined by the distance to its kth NN. Related to this knn-based (in other words, proximity-based method), many methods, including Local Outlier Factor (LOF) [5], and connectivity-based outlier factor (COF) [36] have been proposed.

As another type of outlier detection method, ABOD [20] is an angle-based method. In ABOD, the outlier score of a point $\boldsymbol{x}$ is calculated using the variance of angle among $\boldsymbol{x}$ and two other points in the dataset. However, this algorithm needs $O(N^3)$ time complexity, which is realistically intractable. To avoid this problem, FastABOD, the approximated version of ABOD, is also proposed [20]. In FastABOD, only kNNs of each point are used to compute the mentioned variance, and calculation time is drastically shortened.

Besides these methods, there are many types of outliers detection algorithms. See the PyOD library for their details [1].

As mentioned, outliers detection may be another option of N reduction. However, the effectiveness of subsampling outliers in NN search has not been revealed theoretically or empirically. Then in Sec. 5.5 we actually compared our proposed method with introduced typical knn-based outliers detection [32] and FastABOD [20].

3 WHICH VECTORS ARE UNNECESSARY?

First, let us define N D-dimensional database vectors as $\mathcal{X} = \{\boldsymbol{x}_n\}_{n=1}^N \subset \mathbb{R}^D$. Given a query vector $\boldsymbol{q} \in \mathbb{R}^D$, our objective is to find the nearest vector to $\boldsymbol{q}$ from $\mathcal{X}$:

$$\mathrm{NN}(\boldsymbol{q}) = \underset{\boldsymbol{x}\in\mathcal{X}}{\operatorname{argmin}} \|\boldsymbol{q} - \boldsymbol{x}\|_2. \tag{1}$$

The 2nd NN, 3rd NN, etc., are defined in the same manner. Because the direct computation of Eq. 1 is slow, several ANN methods have been proposed for searching approximately but efficiently.

Our approach is a simple preprocessing; we remove unnecessary vectors from $\mathcal{X}$. Any ANN method can then be applied to the reduced data. Of course, the memory cost goes down after the data reduction. If we can remove unnecessary data successfully, we expect that the decrease in accuracy will be moderate. The problem here is how to identify such unnecessary data without unseen queries.

First, we introduce a statistical measure for database vectors, hubness [30]. Then, we illustrate that "unnecessary" vectors have distinctive hubness scores relative to the actual nearest neighbors.

3.1 Hubness

Hubness is literally a measure of how much of a "hub" the point is [30]. In detail, the hubness of a point $\boldsymbol{x}$ (represented by $h_k(\boldsymbol{x})$) is defined as the number of vectors whose k-NNs contain $\boldsymbol{x}$. It is mathematically defined as follows [30].

Given a dataset $\{\boldsymbol{x}_n\}_{n=1}^N$, let us first define an indicator function:

$$p_{n,k}(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \text{ is among the } k \text{ nearest neighbors of } \boldsymbol{x}_n \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

[1] https://pyod.readthedocs.io/en/latest/index.html

The hubness of the point $\boldsymbol{x}$ is defined as:

$$h_k(\boldsymbol{x}) = \sum_{n=1}^{N} p_{n,k}(\boldsymbol{x}) \tag{3}$$

The hubness score depends only on the dataset $\{\boldsymbol{x}_n\}_{n=1}^N$ itself (and a parameter, k), meaning that the score can be computed offline before the search step.

Historically, hubness has been mainly applied to supervised learning tasks. In the image recognition, e.g., a data point with a high hubness score (simply called a *hub*) is regarded as a distractor [29]. This is because hubs can be similar to any kind of image regardless of their labels. This means that *data with a low hubness score are important.*

Meanwhile, in our ANN problem, we found that *data with a high hubness score are important.* This is because only the distance measure itself is important in the ANN problem. A point with a high hubness score is likely to be contained in the k-NNs from the queries. This setting is totally different from the ones of supervised learning, such as image recognition, where similar vectors with different labels should be penalized.

In other words, a point with a low hubness score (called *anti-hub* in the data mining community [31]) is not important in our ANN problem. Such anti-hubs do not contribute to the final search score. Our solution is to remove such anti-hubs from the database beforehand. Again, we also examined outlier detection as the identification of unnecessary points. See Sec. 5.5 for more details.

3.2 Exploratory Experiment with Deep1M

This subsection illustrates a motivating example with the Deep1M dataset (The details of this dataset are described in Sec.5). This example illustrates the intuition that, although hubness scores can be computed before the search, they are suitable measures to identify unimportant vectors for unseen queries.

First, let us discuss the nearest neighbor vector for each query. Let us form the nearest neighbor set $\mathcal{X}_{\mathrm{NN}} \subseteq \mathcal{X}$:

$$\mathcal{X}_{\mathrm{NN}} = \bigcup_{\boldsymbol{q}\in Q} \mathrm{NN}(\boldsymbol{q}), \tag{4}$$

where $Q \subset \mathbb{R}^D$ is a set of queries. This neighbor set $\mathcal{X}_{\mathrm{NN}}$ is obviously important because it contains the ground-truth results for the search task. Next, let $\overline{\mathcal{X}_{\mathrm{NN}}}$ be the remaining vectors, i.e., $\overline{\mathcal{X}_{\mathrm{NN}}} = \mathcal{X} \setminus \mathcal{X}_{\mathrm{NN}}$. Since $\overline{\mathcal{X}_{\mathrm{NN}}}$ does not contain any ground-truths for queries, we can suppose this set is not necessarily important.

Here, we empirically found an interesting relationship; the items in $\mathcal{X}_{\mathrm{NN}}$ have higher hubness scores than those of items in $\overline{\mathcal{X}_{\mathrm{NN}}}$. This result is surprising because, although hubness scores are computed simply using the database vectors, they could be used to predict the importance of database items for unseen queries.

Figure 2 shows the results. Let us first compute the hubness for each vector in the neighbor set: $h_k(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathcal{X}_{\mathrm{NN}}$. Notice that hubness is calculated with all vectors in the database $\mathcal{X}$, instead of with only $\mathcal{X}_{\mathrm{NN}}$. The histogram is shown as red bars in Fig. 2. Next, we randomly select $|\mathcal{X}_{\mathrm{NN}}|$ vectors from $\overline{\mathcal{X}_{\mathrm{NN}}}$ to make the comparison fair, then we compute $h_k(\boldsymbol{x})$ for each item. The result is shown as blue bars in Fig. 2. These histograms clearly show that the vectors in $\mathcal{X}_{\mathrm{NN}}$ have higher hubness scores than those of $\overline{\mathcal{X}_{\mathrm{NN}}}$.

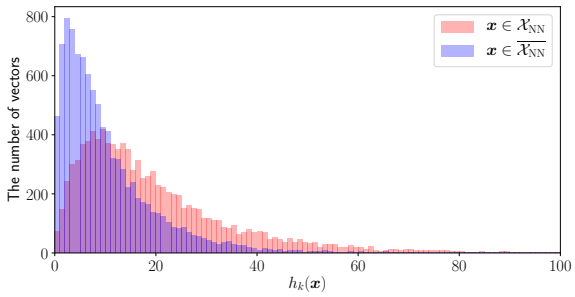


Figure 2: Distribution of hubness scores in $x \in \mathcal{X}_{NN}$ and $x \in \overline{\mathcal{X}_{NN}}$. The hubness calculation set $k = 10$.

As is obvious from the definition, vectors with a high hubness score are likely to be NNs of other vectors in $\mathcal{X}$. Fig. 2 illustrates that such hub vectors tend to be NNs of the unseen queries as well.

4 PROPOSED METHOD

Section 3.2 reveals that a database vector with a high hubness score tends to be a nearest neighbor to the query vector. With this insight in hand, we devised a data reduction scheme that can keep search accuracy as high as possible while saving memory.

4.1 Reduction of Anti-Hubs

Formally, our task is to replace the original $\mathcal{X}$ by a reduced set $\mathcal{X}^* \subseteq \mathcal{X}$. Let us sort the database vectors $\mathcal{X} = \{\boldsymbol{x}_n\}_{n=1}^N$ in descending order of their hubness score, i.e., $h_k(\boldsymbol{x}_n)$ for each $\boldsymbol{x}_n$. We then take the first T vectors from the sorted vectors. This operation is defined as a function f:

$$\mathcal{X}^* = f(\mathcal{X}, T, k), \tag{5}$$

where k is a hyper parameter to compute the hubness score (Eq. 3). The data size after the reduction, T, is an input parameter which users provide. Note that $|\mathcal{X}^*| = T$. As we can compute $\mathcal{X}^*$ offline, we do not need to know anything else about $\mathcal{X}$ when reducing the data size. Note that, if it is obvious from the context, we will simply omit k and write Eq. 5 as $\mathcal{X}^* = f(\mathcal{X}, T)$. A previous study also focused on anti-hubs and applied them to outliers detection [31]. The key here is to re-formulate it for the ANN search problem, e.g., explicitly removing them to get $\mathcal{X}^*$ with a parameter T, which decides the database size.

The computational cost of this reduction method is summarized as follows. After creating the distance matrix of the N D-dimensional database vectors at the cost of $O(DN^2)$, the k nearest items are selected for every vector by using the partial sort algorithm, which takes $O(N^2 \log k)$. Next, hubness is calculated by counting the occurrences of each vector in the partially sorted items. This takes $O(kN)$. Finally, the hubness scores for each vector (an array with N elements) are partially sorted in descending order based on their hubness and T vectors are kept as the reduced database: $O(N \log T)$. The sum of the above operations is the total cost. Suppose k and $\log T$ are much smaller than N, and $\log k$ is much smaller than D; the dominant cost is in creating the distance table: $O(DN^2)$. For example, the computation of Eq. 5 ($N = 10^6$, $D = 96$, $T = N/2$, and $k = 16$) took 227 seconds using our multi-thread implementation with 96 threads.

After the reduced set $\mathcal{X}^*$ is obtained, any ANN method can be run over it. This anti-hub removal is a N reduction method, so we can combine it with various γ reduction methods to save memory.

With this approach, the search with queries near anti-hubs may fail. As mentioned, however, we assume that the distribution of database and queries are the same. Under this assumption, queries tend to exist around hubs. Hence, the mentioned failure is rare and does not affect the overall search accuracy so much.

4.2 Approximation

As mentioned in Sec. 4.1, the dominant computational cost of the hubness calculation is $O(DN^2)$. In the case of million-scale datasets, this cost doesn't matter. When the scale of the dataset is enlarged to a billion, however, it is realistically impossible to finish the hubness calculation with a reasonable amount of time and memory (it will take years, and several hundred gigabytes of memory will be needed). To avoid this problem, we approximate the algorithm of the hubness calculation (Eq. 5) and reduce its computational cost. The idea here is divide-and-conquer, which is simple, yet works well. This approximate process is as follows.

First, k-means clustering is executed on the original dataset $\mathcal{X}$, and $\mathcal{X}$ is divided into C clusters: $\mathcal{X} = \bigcup_{c=1}^{C} \mathcal{X}_c$. We can treat each cluster as a subset. Next, for each subset $\mathcal{X}_c$, Eq. 5 is applied. The size of the subset is reduced from $|\mathcal{X}_c|$ to $\frac{|\mathcal{X}_c|}{|\mathcal{X}|}T$. Finally, all of the reduced subsets are gathered to form the final output. This process is considered to be an approximation of Eq. 5, where the hubness score is approximately computed for each subset. Here, we represent this approximate computing process by $\hat{f}$, which is defined as:

$$\mathcal{X}^* = \hat{f}(\mathcal{X}, T, C, k) = \bigcup_{c=1}^{C} f\left(\mathcal{X}_c, \frac{|\mathcal{X}_c|}{|\mathcal{X}|}T, k\right) \tag{6}$$

Usually, the hyper parameter k is set to 16. Unless we add any change in this setting, we write $\hat{f}(\mathcal{X}, T, C)$ in place of $\hat{f}(\mathcal{X}, T, C, k)$.

The main computational cost is again the creation of the distance table for each subset: $O(\frac{DN^2}{C})$, which is much smaller than the original cost, $O(DN^2)$. For example, the calculation of hubness and reduction of anti-hubs of the Deep1B dataset in this approximation process with $C = 1000$ took about 70 hours with multi-threading (96 threads). In more detail, clustering took only 40 minutes and the rest of the time is consumed by the calculation of hubness. As the algorithm is easy to be parallelized, the main part of this reduction method can also be run on GPUs. When we used four Tesla V100 GPUs (16GB), the whole computing time was shortened to about 5 hours. We would like to emphasize that the proposed data reduction is offline preprocessing. This means that its computational cost does not have any effect on the runtime of the search.

We can compute hubness and reduce the data with this approximate approach because what we want is not the hubness value itself, but rather the order of the database vectors based on hubness. We expect the order based on the hubness score (Eq. 5) is similar to the order computed from each subset (Eq. 6).

In our experiments on billion-scale datasets (described below), we computed hubness with this approximate process. In Sec. 5.8, we validate this approximation on the Deep1M dataset and show that

Dataset	$\lvert\mathcal{X}\rvert$	$\lvert\mathcal{Q}\rvert$	D
Deep1B [39]	1,000,000,000	10,000	96
Deep1M [39]	1,000,000	10,000	96
SIFT1M [18]	1,000,000	10,000	128
GIST1M [18]	1,000,000	1,000	960
GloVe [28]	200,000	2,000	50

Table 1: Statistics of each dataset

it performs comparably to the original hubness. Accordingly, we decided to use this approximation to handle billion-scale datasets.

5 EXPERIMENTS

We evaluated our approach by comparing it with other N reduction methods and γ reduction methods under the fixed memory consumption. To compute approximate Hubness scores in Sec. 5.8 and 5.9, we used four NVIDIA Tesla V100 GPUs. We ran all experiments on a server with a 3.6 GHz Intel Xeon CPU (24 cores, 96 threads) and 192 GiB of RAM for other parts. For the data reduction part, we implemented our approach in Python with multi-threading. We employed the authors' implementation (C++ code with a Python wrapper) with a single thread mode for a fair comparison for the search part. All source codes will be publicly available[2].

Unless otherwise noted, the hyperparameter k is set to 16. This k is the only hyper-parameter that users need to decide, and we thoroughly analyzed the choice of k. The conclusion is that a fixed value, $k = 16$, works pretty well in practice.

In Sec. 5.5, 5.6, and 5.7, hubness are computed exactly (Eq. 5), and in Sec. 5.8 and 5.9, the approximate hubness is calculated (Eq. 6).

5.1 Dataset

We made use of the following four million-scale datasets and one billion-scale dataset;

- **Deep1B** [39] consists of 96-dimensional deep features extracted from the last FC layer of GoogLeNet [35]. This dataset has 1,000,000,000 base vectors and 10,000 queries.
- **Deep1M** is a subset (the first one million items) of the Deep1B. It provides 1,000,000 base and 10,000 query vectors.
- **SIFT1M** [18] consists of 128-dimensional SIFT features extracted from several images. There are 1,000,000 base and 10,000 query vectors available.
- **GIST1M** [18] is composed of 960-dimensional GIST features. These features are also extracted from some images. It provides 1,000,000 base 1,000 query vectors.
- **GloVe** [28] consists of 400,000 50-dimensional word vectors extracted from the Wikipedia 2014 + Gigaword 5 corpus. From this dataset, we used random 200,000 vectors as the database and 2,000 vectors as queries.

The statistics of the datasets are summarized in Tbl. 1.

5.2 Data Subsampling Methods

To reduce the number of vectors in a database (N), we used four methods as follows:

- **Anti-hub removal** is the proposed method in that vectors of low hubness scores are removed as anti-hubs.
- **Random reduction** is the operation of $f_{\text{rand}}(\mathcal{X}, T)$, which means taking T vectors randomly from $\mathcal{X}$. This random reduction is used as a baseline.
- **knn-based outlier detection** [32] is a typical distance-based outlier detection method. The outlier score of each data point is defined as the distance to its kth NN. In our experiment, we set $k = 16$.
- **FastABOD** [20] is an angle-based outliers detection method. The variance of angles among a vector and its two NNs is used to calculate the outlier score of the vector. Here, we used 16 NNs of each vector in making triplets.

5.3 Vector Compression Methods

To change γ, or the data size of each vector, we used the following vector compression methods.

- **PCA** is used as a dimensionality reduction method. We change the value of γ by setting the dimensionality to which PCA projects the vectors.
- **PQ** is implemented in the faiss library [16], so we make use of it. We operate the memory usage by m, the number of subvectors in PQ. With 8-bit subquantizers, PQ compresses each vector to m bytes.
- **BE** indicates the binary encoding of vectors with random rotations [7][10]. In BE, each vector in the database is transformed to a b-bit vector, and we can approximately compute the distance between two vectors by the hamming distance in binary space. Thanks to this encoding, we can reduce γ to $b/8$ bytes. We also used the faiss library to compute BE. Note that we also tried ITQ [10], an extension of basic binary encoding, but the result does not differ. Therefore, we use the basic one in the experiment.

5.4 Notation

In our experiment, various data reduction methods, and their combination are used. Therefore, we use the following notations.

- "**AHR**α" ($\alpha \in [0, 1]$) with a database $\mathcal{X}$ means $f(\mathcal{X}, \alpha|\mathcal{X}|)$. In detail, anti-hubs are removed, and the size of the reduced database is $\alpha|\mathcal{X}|$. This is our proposed method.
- "**RR**α" is a random reduction. This term means $f_{\text{rand}}(\mathcal{X}, \alpha|\mathcal{X}|)$.
- "**KNNOD**α" reduce the size of a database to $\alpha|\mathcal{X}|$ by knn-based outlier detection [32].
- "**FABOD**α" means the data reduction by FastABOD [20]. α is the same as the case in "AHRα."
- "**PCA**d'" indicates the dimensionality reduction by PCA [34]. After PCA, the dimensionality becomes d'.
- "**PQ**m" is a γ reduction method by PQ [17]. m specifies the number of subvectors.
- "**BE**b" computes the binary encoding [7], and each encoded vector is b bits.

We also uses a combination of methods such as "AHR$\frac{1}{2}$PCA$\frac{d}{2}$". In such cases, methods are executed in the written order. In this case, e.g., we compute two data reduction methods with a N d-dimensional database $\mathcal{X}$. Firstly, the half of the vectors are removed

[2] https://github.com/naaktslaktauge/antihub-removal

as anti-hubs, and we get $f(\mathcal{X}, \frac{|\mathcal{X}|}{2})$. After that, PCA halves d, or the dimensionality of $f(\mathcal{X}, \frac{|\mathcal{X}|}{2})$. As a result, the size of the processed database is $\frac{1}{4}$ of that of original $\mathcal{X}$.

5.5 Comparison with Other N Reduction Methods

In this section, we compare anti-hub removal with other N reduction methods. In detail, we compared four methods: 1) AHRT (our proposed method), 2) RRT, 3) KNNODT, 4) FABODT, for $T \in \{\frac{N}{10}, \frac{2N}{10}, \frac{3N}{10}, ..., \frac{10N}{10}\}$. After the data reduction, a pure NN search is conducted, and the search accuracy (recall@1) is calculated.

As fig. 3 shows the result, our proposed method (anti-hub removal, AHR) works best for any dataset and any data size. After this section, we use only anti-hub removal as the N reduction method.

5.6 Comparison with Other γ Reduction Methods

Here, we made a comparison between anti-hub removal and γ removal methods. For each dataset and each γ compression, we make two databases **whose memory consumption are equal**; 1) γ reduction only, 2) the combination of γ reduction and anti-hub removal. These two databases are made by following methods:

(1) **PCA**d' **vs AHR**$\frac{1}{2}$**PCA**$2d'$ with $d' \in \{5, 10, 20\}$.
(2) **PQ**m **vs AHR**$\frac{1}{2}$**PQ**$2m$ with $m \in \{4, 8, 16\}$ for Deep1M, SIFT1M, GIST1M, and $m \in \{1, 5, 25\}$ for GloVe.
(3) **BE**b **vs AHR**$\frac{1}{2}$**BE**$2b$ with $b \in \{D/2, D, 2D, 4D\}$. D specifies the original dimensionality.

For each database, recall@1 as a result of NN search is computed. Repeatedly, this experiment is a comparison of search accuracy under the fixed memory usage.

Fig. 4 shows the result. In any case, our proposed method combined with the γ reduction method outperforms the γ reduction alone. This result suggests that directly compressing vectors by PQ, PCA, etc., is not optimal for reducing memory consumption. A more effective way is to reduce the number of vectors by anti-hub-based sampling first and then compress the remaining vectors.

5.7 Application to Image Retrieval

We qualitatively showed the practicality of our method in the context of an image search. We used the training subset of the Google-Landmarks Dataset [26]. In this dataset, each image depicts one landmark. Here, all images are converted to deep image retrieval features (DIR features) [11]. This dataset contains 1,219,242 2048-dimensional vectors. We randomly chose 10k vectors as queries and used the rest as the database.

Two NN searches were conducted on different databases: 1) PCA32AHR$\frac{1}{2}$, 2) PCA16. Notice that the sizes of both databases are the same, so **this experiment is also a comparison under the fixed memory consumption**.

The results of the two searches are shown in Fig. 5, including a sample query. In the figure, the left search results are closer to the query. The search with our data reduction (PCA32AHR$\frac{1}{2}$) keeps the correct images of the same landmark. On the other hand, only PCA (PCA16) method suggests incorrect images that depict different landmarks from the query because the excessive γ reduction by PCA deteriorates the search accuracy. From this result, we can say that our method can be applied to actual image retrieval tasks.

5.8 Validation of Approximation Process

Next, we showed that the search with the data reduction using the approximate hubness (Eq. 6) has comparable accuracy to that of data reduction using the original hubness (Eq. 5).

Fig. 6 shows the relationship between C in computing the approximate hubness and recall@1 of the query search. When $C = 1$, the search is identical to that of the original reduction: $\hat{f}(\mathcal{X}, T, 1) = f(\mathcal{X}, T)$. Here, we used Deep1M, SIFT1M, GIST1M, GloVe. The parameters T were set to $\frac{|\mathcal{X}|}{2}$. While the runtime of computing hubness becomes shorter with increasing C, the accuracy decreased. However, even when the approximate hubness was calculated with $C = 1024$ subsets, the accuracy decreased only 0.05 (0.812 → 0.764) on average, while runtime was 1000 times shorter. This result indicates that our approximation process works sufficiently well.

5.9 Billion-Scale Evaluation

Because of its huge data size, information retrieval on a billion-scale dataset needs strong approximation to suppress memory consumption and accelerate the search; this causes a decrease in search accuracy [18]. Also, it is hard to manage such massive datasets. We evaluated our data reduction method on the current largest dataset, Deep1B [39]. We found that:

- By combining the IVFPQ+HNSW approach, we achieved the highest accuracy under a fixed memory requirement.
- Unlike data compression schemes such as PCA [34] and PQ [17] where the original data need to be transformed, our approach is sub-sampling, which is simple and intuitive. This makes it easy to handle a large-scale dataset.

In the experiment, we set C to 100 considering the tradeoff between search accuracy and hubness calculation speed.

5.9.1 Settings. The current best algorithm for billion-scale indexing and searching is based on IVFPQ with HNSW. This algorithm is the same as IVFPQ, except that the coarse quantizer is replaced by HNSW [23]. We will denote this algorithm as IVFPQ + HNSW [2, 8]. By adjusting its parameters, IVFPQ + HNSW achieved the best tradeoff curve between search speed and accuracy.

The dominant part of the memory consumption of IVFPQ + HNSW for a dataset $\mathcal{X}$ is summarized as follows (unit: byte)[3]:

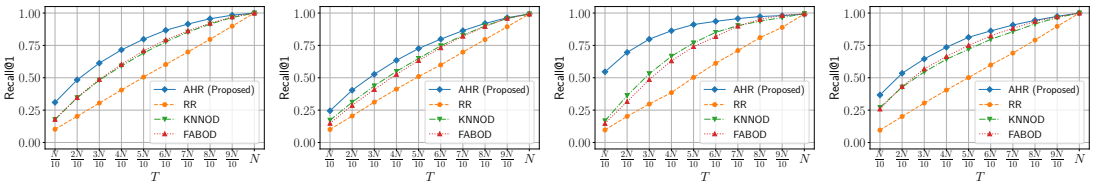
$$(M_{\text{pq}} + 4)|\mathcal{X}|. \tag{7}$$

Here, M_{pq} indicates the number of subvectors in product quantization. For each vector, we need to store a four-byte integer for its identifier. Note that we suppose 8 bits are used for each subquantizer. We ignore the memory usage of the coarse quantizer because it is hugely smaller compared with that of product quantization.

We reduced the memory consumption in three ways:

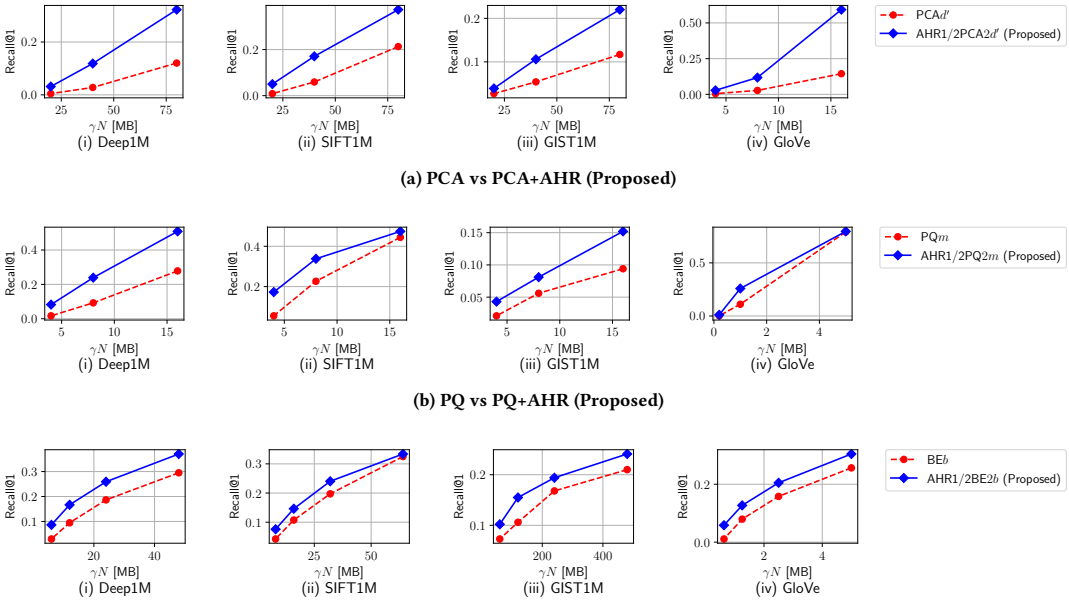
(1) proposed reduction method based on hubness ($\hat{f}$ in Eq. 6)
(2) random reduction method (f_{rand})
(3) changing the parameter: M_{pq}

[3] https://github.com/facebookresearch/faiss/wiki/Faiss-indexes



(a) Deep1M (b) SIFT1M (c) GIST1M (d) GloVe

Figure 3: Accuracy of the nearest neighbor search for the reduced dataset.



(a) PCA vs PCA+AHR (Proposed)

(b) PQ vs PQ+AHR (Proposed)

(c) BE vs BE+AHR (Proposed)

Figure 4: Comparison of each γ reduction method and its combination with anti-hub removal for three methods / four datasets.

(a) Query Image

(b) Search Result of PCA32AHR1/2 (Proposed)

(c) Search Result of PCA16

Figure 5: Query image, and search result with PCA32AHR$\frac{1}{2}$ (proposed) and PCA16. In search results, a left image is closer to the query.

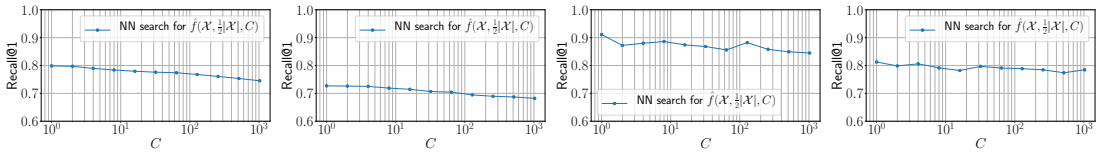
We controlled the memory consumption of the dataset by varying a parameter $\alpha \in \{0.125, 0.25, 0.5, 1\}$. Considering $|\mathcal{X}| = 10^9$ for the Deep1B dataset, we determined the memory cost be $16\alpha + 4$ [GB] for each method described above. The details are as follows.

(1) Proposed: The dataset was reduced by $\mathcal{X}^* \leftarrow \hat{f}(\mathcal{X}, \frac{4\alpha+1}{5}|\mathcal{X}|, 100)$. Thus, $|\mathcal{X}^*| = \frac{4\alpha+1}{5} \times 10^9$. The parameter is: $M_{\text{pq}} = 16$. This results in $(16+4)(\frac{4\alpha+1}{5} \times 10^9) = (16\alpha+4) \times 10^9$ bytes.

(2) Random: The dataset was reduced by $\mathcal{X}^* \leftarrow f_{\text{rand}}(\mathcal{X}, \frac{4\alpha+1}{5}|\mathcal{X}|)$. Thus, $|\mathcal{X}^*| = \frac{4\alpha+1}{5} \times 10^9$. The parameter is: $M_{\text{pq}} = 16$. Again, this results in $(16+4)(\frac{4\alpha+1}{5} \times 10^9) = (16\alpha+4) \times 10^9$ bytes.

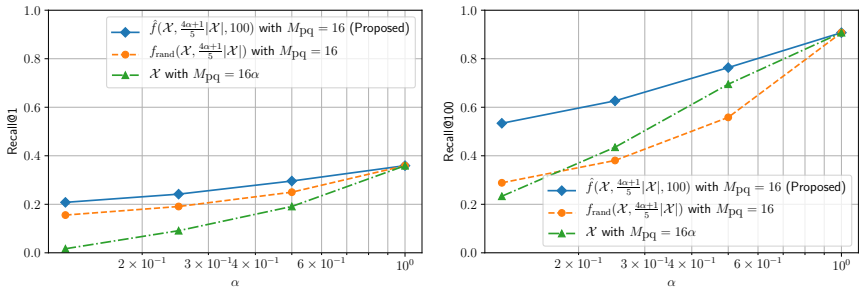
(3) Changing params: The dataset size equaled the original size: $|\mathcal{X}| = 10^9$. This results in $(16\alpha+4) \times 10^9$ bytes with $M_{\text{pq}} = 16\alpha$.

Note that we used the IVFPQ + HNSW implementation in faiss [16].



(a) Deep1M (b) SIFT1M (c) GIST1M (d) GloVe

Figure 6: NN search for $\hat{f}(\mathcal{X}, \frac{1}{2}|\mathcal{X}|, C)$ with various C.



(a) Recall@1 (b) Recall@100

Figure 7: Accuracy of IVFPQ + HNSW for Deep1B reduced in three ways: (1) $\hat{f}(\mathcal{X}, \frac{4\alpha+1}{5}|\mathcal{X}|, 100)$ (proposed), (2) $f_{\mathrm{rand}}(\mathcal{X}, \frac{4\alpha+1}{5}|\mathcal{X}|)$, (3) changing parameters of IVFPQ+HNSW for $\mathcal{X}$ with $M_{\mathrm{pq}} = 16\alpha$. The data size of each method is equal for each α.

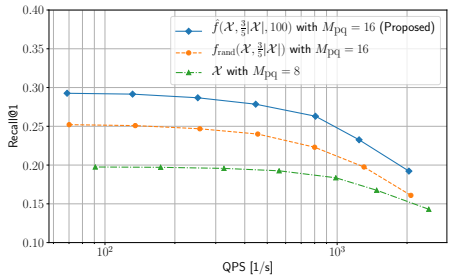


Figure 8: Recall@1-QPS of IVFPQ + HNSW for Deep1B reduced in three ways with $\alpha = 0.5$ and various n_p: (1) $\hat{f}(\mathcal{X}, \frac{3}{5}|\mathcal{X}|, 100)$ with $M_{\mathrm{pq}} = 16$ (proposed), (2) $f_{\mathrm{rand}}(\mathcal{X}, \frac{3}{5}|\mathcal{X}|)$ with $M_{\mathrm{pq}} = 16$, (3) IVFPQ+HNSW for $\mathcal{X}$ with $M_{\mathrm{pq}} = 8$.

The search space was divided into $\sqrt{|\mathcal{X}|}$ cells, and the search was executed in $n_p = |\mathcal{X}|/100$ cells near a query. The number of links from each vector for HNSW was set to 16. For each α, we evaluated these three approaches in terms of recall@1 and recall@100.

5.9.2 Results. Fig. 7 shows the results for each method. As shown, our method works best in terms of both recall@1 and recall@100 for any α. It especially had a much higher recall@100. If the ground-truth vectors had been removed from the database by our method, its recall@100 would not be much higher than recall@1. Hence, we conclude that our method removed truly meaningless data only.

IVFPQ + HNSW is the best ANN method for billion-scale searches, and IVFPQ + HNSW + Anti-hub reduction achieved better performance under the assumption of equal memory usage.

5.9.3 Comparison under the same memory, same speed. We further show that, **given fixed memory consumption and target runtime, our approach achieves higher accuracy than other methods.** Fig. 8 shows Recall@1-QPS (query per second) of three methods in Sec 5.9.1. Here, we equalized the memory consumption as 12 GB for each method by setting $\alpha = 0.5$. Each dot is plotted by changing the query parameter n_p, the number of cells used for the search. We examined $n_p \in \{1, 2, 4, 8, 16, 32, 64\}$. The bigger n_p gets, the slower the search goes (QPS gets smaller) as the search area gets expanded. The result shows that the proposed method consistently achieved the highest accuracy for a given QPS. This means that our proposed method achieves better search accuracy under fixed memory usage and fixed runtime.

6 CONCLUSION

We proposed a new data reduction method based on hubness for memory saving nearest neighbor searches. As our method is preprocessing, it can be applied to any dataset and any ANN method. Our proposed method achieved higher search accuracy under the fixed memory usage by combining with existing vector compression methods, compared with the vector compression alone. Furthermore, we developed an approximate algorithm for calculating hubness, making it possible to apply our method to billion-scale datasets. Combined with the best ANN algorithm, our method achieved much higher accuracy under the assumption of equal memory consumption and fixed runtime.

ACKNOWLEDGMENTS

This work was supported by JST, PRESTO Grant Number JPMJPR1936.

REFERENCES

[1] A. Babenko and V. Lempitsky. 2014. Additive Quantization for Extreme Vector Compression. In *Proc. IEEE CVPR*.

[2] D. Baranchuk, A. Babenko, and Y. Malkov. 2018. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. In *Proc. ECCV*.

[3] E. Bernhardsson. [n.d.]. Annoy. https://github.com/spotify/annoy

[4] P. Brasnett and M. Bober. 2008. Fast and robust image identification. In *ICPR*.

[5] M. M. Breunig, H. Kriegel, R. T. Ng, and J. Sander. 2000. LOF: Identifying Density-Based Local Outliers. *SIGMOD Rec.* 29, 2 (2000), 93–104.

[6] M. M. Bronstein, A. M. Bronstein, F. Michel, and N. Paragios. 2010. Data fusion through cross-modality metric learning using similarity-sensitive hashing. In *Proc. IEEE CVPR*.

[7] M S. Charikar. 2002. Similarity Estimation Techniques from Rounding Algorithms. In *Proc. STOC*.

[8] M. Douze, A. Sablayrolles, and H. Jégou. 2018. Link and Code: Fast Indexing with Graphs and Compact Regression Codes. In *Proc. IEEE CVPR*.

[9] T. Ge, K. He, Q. Ke, and J. Sun. 2014. Optimized Product Quantization. *IEEE TPAMI* 36, 4 (2014), 744–755.

[10] Y Gong, Sv Lazebnik, A Gordo, and F Perronnin. 2012. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE TPAMI* 35, 12 (2012), 2916–2929.

[11] A. Gordo, J. Almazán, J. Revaud, and D. Larlus. 2016. Deep image retrieval: Learning global representations for image search. In *Proc. ECCV*.

[12] G. Guðmundsson, B. Jónsson, L. Amsaleg, and M. J. Franklin. 2018. Prototyping a Web-scale multimedia retrieval service using Spark. *ACM TOMM* 14, 3s (2018), 1–24.

[13] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han. 2014. Outlier Detection for Temporal Data: A Survey. *IEEE TKDE* 26, 9 (2014), 2250–2267.

[14] J. He, S. Chang, R. Radhakrishnan, and C. Bauer. 2011. Compact hashing with joint optimization of search accuracy and time. In *Proc. IEEE CVPR*.

[15] P. Indyk and R. Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality *(Proc. STOC)*.

[16] J. Johnson, M. Douze, and H. Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* (2019).

[17] H. Jégou, M. Douze, and C. Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE TPAMI* 33, 1 (2011), 117–128.

[18] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *Proc. IEEE ICASSP*.

[19] B. Klein and L. Wolf. 2019. End-To-End Supervised Product Quantization for Image Search and Retrieval. In *Proc. IEEE CVPR*.

[20] H. Kriegel, M. Schubert, and A. Zimek. 2008. Angle-Based Outlier Detection in High-Dimensional Data. In *Proc. SIGKDD*.

[21] L. Maaten and G. Hinton. 2008. Visualizing data using t-SNE. *JMLR* 9, Nov (2008), 2579–2605.

[22] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61 – 68.

[23] Y. A. Malkov and D. A. Yashunin. 2018. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *IEEE TPAMI* (2018).

[24] Y. Matsui, R. Hinami, and S. Satoh. 2018. Reconfigurable Inverted Index. In *Proc. ACMMM*.

[25] M. Muja and D. G Lowe. 2009. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP* 2, 331-340 (2009), 2.

[26] H. Noh, A. Araujo, J. Sim, T. Weyand, and B. Han. 2017. Large-Scale Image Retrieval with Attentive Deep Local Features. In *Proc. IEEE ICCV*.

[27] M. Norouzi and D. J. Fleet. 2013. Cartesian K-Means. In *Proc. IEEE CVPR*.

[28] J. Pennington, R. Socher, and C. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proc. EMNLP*.

[29] M. Radovanović, A. Nanopoulos, and M. Ivanović. 2009. Nearest neighbors in high-dimensional data: The emergence and influence of hubs. In *Proc. ICML*.

[30] M. Radovanović, A. Nanopoulos, and M. Ivanović. 2010. Hubs in space: Popular nearest neighbors in high-dimensional data. *Journal of Machine Learning Research* 11, Sep (2010), 2487–2531.

[31] M. Radovanović, A. Nanopoulos, and M. Ivanović. 2015. Reverse Nearest Neighbors in Unsupervised Distance-Based Outlier Detection. *IEEE TKDE* 27, 5 (2015), 1369–1382.

[32] S. Ramaswamy, R. Rastogi, and K. Shim. 2000. Efficient Algorithms for Mining Outliers from Large Data Sets. *SIGMOD Rec.* 29, 2 (2000), 427–438.

[33] L. Saul and S. Roweis. 2001. An introduction to locally linear embedding. *JMLR* 7 (2001).

[34] J. Shlens. 2014. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100* (2014).

[35] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. Going deeper with convolutions. In *Proc. IEEE CVPR*.

[36] J. Tang, Z. Chen, A. W. Fu, and D. W. Cheung. 2002. Enhancing Effectiveness of Outlier Detections for Low Density Patterns. In *Proc. PAKDD*.

[37] A. Tharwat, T. Gaber, A. Ibrahim, and Aboul E. Hassanien. 2017. Linear discriminant analysis: A detailed tutorial. *Ai Communications* 30 (05 2017), 169–190,.

[38] J. Wang, T. Zhang, j. song, N. Sebe, and H. T. Shen. 2018. A Survey on Learning to Hash. *IEEE TPAMI* 40, 4 (2018), 769–790.

[39] A. B. Yandex and V. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *Proc. IEEE CVPR*.

[40] X. Yang, L. Weifeng, W. Liu, and D. Tao. 2019. A Survey on Canonical Correlation Analysis. *IEEE TKDE* (2019), 1–1.

[41] T. Zhang, C. Du, and J. Wang. 2014. Composite Quantization for Approximate Nearest Neighbor Search. In *Proc. ICML*.

[42] T. Zhang, Guo-Jun Qi, Jinhui Tang, and J. Wang. 2015. Sparse composite quantization. In *Proc. IEEE CVPR*.