International Journal of Multimedia Information Retrieval (2022) 11:159–170
https://doi.org/10.1007/s13735-022-00233-w

REGULAR PAPER

Siamese coding network and pair similarity prediction for near-duplicate image detection

Marco Fisichella[1]

Received: 9 December 2021 / Revised: 11 March 2022 / Accepted: 25 March 2022 / Published online: 12 April 2022
© The Author(s) 2022

Abstract

Near-duplicate detection in a dataset involves finding the elements that are closest to a new query element according to a given similarity function and proximity threshold. The brute force approach is very computationally intensive as it evaluates the similarity between the queried item and all items in the dataset. The potential application domain is an image sharing website that checks for plagiarism or piracy every time a new image is uploaded. Among the various approaches, near-duplicate detection was effectively addressed by SimPair LSH (Fisichella et al., in Decker, Lhotská, Link, Spies, Wagner (eds) Database and expert systems applications, Springer, 2014). As the name suggests, SimPair LSH uses locality sensitive hashing (LSH) and computes and stores in advance a small set of near-duplicate pairs present in the dataset and uses them to reduce the candidate set returned for a given query using the Triangle inequality. We develop an algorithm that predicts how the candidate set will be reduced. We also develop a new efficient method for near-duplicate image detection using a deep Siamese coding neural network that is able to extract effective features from images useful for building LSH indices. Extensive experiments on two benchmark datasets confirm the effectiveness of our deep Siamese coding network and prediction algorithm.

Keywords Indexing methods · Deep features extraction · Near-duplicate image detection · Locality sensitive hashing · High-dimensional datasets

1 Introduction

In the last two decades, with the proliferation of digital devices, especially mobile phones, the amount of digital images has increased tremendously. Near-duplicates are altered versions of an original image after various possible operations: Compression, cropping, geometric manipulation, contamination by noise, blurring, etc. This may result from illegal activities such as piracy or plagiarism and is a waste of network resources.

The problem of near-duplicate detection can be formulated as follows: Given an image query, find all the images in the collection that are most similar. Images are usually represented by a set of features stored as a feature vector. Each vector, also known as the embedding of the image, is used as a multidimensional vector representation to compute the similarity between two images. The similarity is expressed using a metric defined over the feature space.

The problem of near-duplicate detection has applications in several areas, such as textual retrieval [5,6,10], natural language text retrieval [9], and in this work, image retrieval. In general, this has been studied for a long time, but it is still considered a difficult task. The main reason for this is the well-known *course of dimensionality* [4]. It has been shown that the computational complexity of near-duplicate detection increases exponentially with the number of point dimensions [2]. There are two major challenges in this task: (i) extracting effective image features to improve the recognition accuracy; (ii) improving the recognition efficiency by reducing the similarity computations between the elements in the dataset.

Among the various approaches, near-duplicate detection was effectively addressed by SimPair LSH [11]. SimPair LSH builds on locality sensitive hashing (LSH) and computes and stores in advance a small set of near-duplicate pairs present in the dataset and uses them to reduce the candidate set returned for a given query using the Triangle inequality.

✉ Marco Fisichella
mfisichella@L3S.de

[1] L3S Research Center, Leibniz University, Appelstr. 9a, Hannover 30167, Germany

The main contributions of this work are summarized below:

- We present an algorithm that predicts a lower bound on the number of elements that will be removed from the candidate set by SimPair LSH. This can be used to predict whether the computational cost imposed by SimPair LSH is worth the number of elements removed.
- We describe an efficient method for maintaining and updating information about similar pairs.
- We develop a new efficient method for detecting near-duplicate images using a deep Siamese coding neural network, which is able to extract effective features from images useful for building LSH indices.
- We perform extensive experiments on two benchmark datasets that confirm the effectiveness of our deep Siamese coding network and prediction algorithm.

The rest of this article is as follows. Section 2 first introduces Locality Sensitive Hashing. Section 3 describes related work, and sect. 4 recalls the basic behavior of SimPair LSH, as it will be used as a starting point in the rest of the article. Section 5 proposes an algorithm to predict a lower bound on the number of elements pruned by SimPair LSH. Section 6 describes how efficiently the SimPair LSH algorithm keeps information about similar pairs in memory. Since both our proposed SimPair LSH method and the original LSH work with feature vectors extracted from objects, Sect. 7 presents the features extracted using a Siamese network for images. In addition, Sect. 8 describes the three methods used to extract features from images and create their vector representations: (i) dense color histograms; (ii) color descriptor SIFT; (iii) indexing construction from Siamese network. Section 9 evaluates our work on different real datasets and under different evaluation criteria. Finally, Sect. 10 provides general comments and conclusions.

2 Preliminaries: locality sensitive hashing

Locality sensitive hashing (LSH) [13,17] was proposed by Indyk and Motwani and has applications in various fields, including multimedia near-duplicate detection (e.g., [7,24]). LSH was first applied to indexing high-dimensional points for the Hamming distance [13] and later extended to the L_p distance [8], where L_2 is the Euclidean distance used in this work.

LSH uses certain hash functions to map each multidimensional point into a scalar. The hash functions used have the property that similar points have a higher probability of being mapped together than dissimilar points. When LSH is used to index a group of points to speed up the similarity search, the procedure is as follows: (i) randomly and uniformly select k hash functions h from an LSH hash function family H and create L hash tables, hereafter called *buckets*; (ii) create an index (a hash table) by grouping all points $\boldsymbol{p}$ in dataset P into different buckets based on their hash values; (iii) when a query point $\boldsymbol{q}$ arrives, use the same set of hash functions to map $\boldsymbol{q}$ into L buckets, one from each hash table; (iv) retrieve all points $\boldsymbol{p}$ from the L buckets, collect them in a candidate set C, and remove duplicate points in C; (v) for each point $\boldsymbol{p}$ in C, compute its distance from $\boldsymbol{q}$ and output the points that are similar to $\boldsymbol{q}$.

In LSH, the probability that two points p_1 and p_2 are hashed into the same bucket is proportional to their distance c and can be computed as follows:

$$p(c) = Pr[h(p_1) = h(p_2)] = \int_0^r \left(\frac{1}{c}\right) f\left(\frac{t}{c}\right)\left(1 - \frac{t}{r}\right) dt \tag{1}$$

where $f(t)$ is the probability density function of the absolute value of the normal distribution. We can now use $p(c)$ to compute the collision probability, i.e., the success probability, under H:

$$P(c) = Pr[H(p_1) = H(p_2)] = 1 - (1 - p(c)^k)^L \tag{2}$$

3 Related work

Since its proposal, LSH has been extended in several directions, as reported in the survey on Locality Sensitive Hashing Algorithms and their Applications by the authors in [18].

Multi-Probe-LSH was introduced by [22] and the authors have experimentally shown that it significantly reduces the space cost with the same search quality and similar time efficiency compared to LSH. The key idea of Multi-Probe LSH is that the algorithm not only searches for the near-duplicates in the buckets where the query point $\boldsymbol{q}$ is hashed, but also searches the buckets where the near-duplicates have a slightly lower probability of occurring. The advantage is that each hash table can be better utilized since more than one bucket of a hash table is checked, reducing the number of hash tables. However, multi-probe LSH does not provide the important search quality guarantee as LSH.

In [2], the authors note that despite decades of research, current solutions still suffer from the "curse of dimensionality," i.e., either an exponential amount of space or query time is required in dimensionality d to guarantee an accurate result. For a sufficiently large dimensionality, current solutions offer little improvement over a linear brute force search of the entire dataset, both in theory and in practice. To overcome this limitation, Indyk et al. [3] developed a two-stage hashing algorithm in one of their recent works. The outer

hash table divides the datasets into ranges of bounded diameter. Then, for each range, they build the inner hash table using the center of the smallest enclosing ball of points in the range as the center. The authors claim that they achieve better query time and smaller space requirement with this approach than with their previous work [17].

Boundary-expanding locality sensitive hashing [25] focuses on the problem of nearest neighbors hashed into different buckets because located on the boundaries. The research challenge is addressed by expanding the boundaries of each bucket and increasing the probability that neighbors collide.

QALSH [16] implements special query-oriented hash functions that consider the location of the query with respect to the buckets: If the query is near the bucket boundaries, its near neighbors may be split into another bucket. Therefore, QALSH considers the query point as a link between buckets. By using B+-trees for each hash function, the method is able to make range queries more efficient and reduce loop time. An extension of QALSH is proposed by [16], where the authors suggest a heuristic with two-level indexing and KD-tree structure to speed up the search process.

I-LSH [20] explores an incremental way for radius expansion, where the search radius is expanded in the projections to capture near neighbors in the proximity buckets of the query point. By incrementally increasing the radius, I-LSH can prevent reading unnecessary buckets, resulting in fewer disk I/O operations. However, this results in a higher computational cost.

PM-LSH [27] propose to improve query processing time by using PM-trees as the indexing structure for the data. For more accurate distance estimation leading to better accuracy, the method uses adjustable confidence intervals.

A new approach, namely *R2LSH*, is proposed by the authors in [21], where one-dimensional projections are replaced by two-dimensional projections. During the index creation phase, B+-trees are created for each two-dimensional projection. In the second phase of query processing, R2LSH employs a query-centric ball to explore the query neighborhood. This results in fewer I/O operations.

Finally, it is important to note that SimPair LSH is orthogonal to LSH and other LSH variants described above and, more importantly, can be applied in these scenarios by leveraging the improvements obtained in this work.

4 SimPair LSH

In this section, we discuss the behavior of SimPair LSH, since it will be used as a starting point in the rest of the paper. In what follows, the term *LSH* refers to the original LSH indexing method, unless otherwise stated.

4.1 Problem definition (incremental range search)

Near–duplicate Given a point $\boldsymbol{q} \in \mathbb{R}^d$, any point $\boldsymbol{p} \in \mathbb{R}^d$ such that $d(\boldsymbol{q}, \boldsymbol{p}) < \tau$ is a near-duplicate of $\boldsymbol{q}$, where $\tau \in \mathbb{R}$ is a similarity threshold. Among the available distance functions d that can be used, Euclidean distance was chosen in this work since it is widely used in various applications. However, SimPair LSH can be easily extended to other distance functions, e.g., L1 and Hamming distance, since LSH can be applied in these cases as well.

Incremental range search Given a point $\boldsymbol{q} \in \mathbb{R}^d$ and a set P of n points $\boldsymbol{p}_i \in \mathbb{R}^d$, find all near-duplicates of $\boldsymbol{q}$ in P according to a given distance function $d(\cdot, \cdot)$ and a given similarity threshold τ before $\boldsymbol{q}$ is inserted into P.

4.2 The SimPair LSH algorithm

The main intuition behind SimPair LSH lies in how LSH works. Given a query point $\boldsymbol{q}$ and a set of points $\boldsymbol{p}$, LSH fills a candidate set C for $\boldsymbol{q}$ with *all* the points $\boldsymbol{p}$ stored in the same buckets where $\boldsymbol{q}$ was hashed. The near-duplicates of $\boldsymbol{q}$ are then found by comparing them to all points in C. This approach suffers from two facts. First, a large number of hash tables must be created to increase the probability that all near-duplicates of $\boldsymbol{q}$ are contained in C. This can lead to a higher number of points in C, which in turn means a higher number of comparisons. Second, for high-dimensional points, the number of operations required to compare two points increases. SimPair LSH speeds up the search by precomputing a certain number of pairwise similar points in the dataset and storing them in memory. These are used at query time to prune the candidate set C, resulting in fewer comparisons.

Formally, the SimPair LSH algorithm works as follows. Given a set P of n points $\boldsymbol{p}_i \in \mathbb{R}^d$, SimPair LSH creates L buckets as in LSH. Moreover, for a given similarity threshold θ, the set *SP* of similar pairs $(\boldsymbol{p}_1, \boldsymbol{p}_2) : d(\boldsymbol{p}_1, \boldsymbol{p}_2) < \theta$ is formed and stored along with the computed distances $d(\boldsymbol{p}_1, \boldsymbol{p}_2)$. Whenever a query point $\boldsymbol{q} \in \mathbb{R}^d$, SimPair LSH retrieves all points in the buckets to which $\boldsymbol{q}$ is hashed. Let this set of points be the candidate set C. Instead of linearly searching all points $\boldsymbol{p}$ in C and computing their distances to $\boldsymbol{q}$ as LSH does, SimPair LSH checks the precomputed similar pair set *SP* for each distance computation $d(\boldsymbol{q}, \boldsymbol{p})$. Depending on the value of $d(\boldsymbol{q}, \boldsymbol{p})$ and on the value of a given similarity threshold τ, SimPair LSH can behave in 2 different ways:

- If $d(\boldsymbol{q}, \boldsymbol{p}) <= \tau$, then SimPair LSH searches *SP* for all points $\boldsymbol{p}\prime : d(\boldsymbol{p}, \boldsymbol{p}\prime) \leq \tau - d(\boldsymbol{q}, \boldsymbol{p})$. It is checked whether

$\boldsymbol{p\prime}$ is in the candidate set C or not: if so, then it marks $\boldsymbol{p\prime}$ as a near-duplicate of $\boldsymbol{q}$ without computing the distance $d(\boldsymbol{p\prime}, \boldsymbol{q})$.
- If $d(\boldsymbol{q}, \boldsymbol{p}) > \tau$, SimPair LSH searches in *SP* for all the points $\boldsymbol{p\prime} : d(\boldsymbol{p}, \boldsymbol{p\prime}) < d(\boldsymbol{q}, \boldsymbol{p}) - \tau$. It checks whether $\boldsymbol{p\prime}$ is in the candidate set C or not. If yes, then it removes $\boldsymbol{p\prime}$ from C without the distance computation.

For the detailed description, see Algorithm 1.

Algorithm 1 SimPair LSH

Require: A set P with n d-dimensional points; L in-memory hash tables created by LSH; a set *SP* storing all similar pairs in P whose pair-wise distances are smaller than θ; a distance threshold τ defining near neighbors; and a query point q.
Ensure: All near neighbors of q in P.

```
1: Check the L buckets q hashed to and retrieve all the points in those
   buckets as in LSH;
2: Put all the points into a candidate set C;
3: for each point p in C do
4:    Compute the distance between q and p, i.e., d(q, p);
5:    if d(q, p) <= τ then
6:       Output p as a near neighbor of q;
7:       Search in SP for all the points p′ which satisfies d(p, p′) <
   τ − d(q, p);
8:       for each point p′ found in SP do
9:          Check if p′ in C or not;
10:         if found then
11:            Output p′ as a near-neighbor of q and remove it from
   C;
12:         end if
13:      end for
14:   end if
15:   if d(q, p) > τ then
16:      Search in SP for all the points p′ which satisfies d(p, p′) <
   d(q, p) − τ;
17:      for each point p′ found in SP do
18:         Check if p′ in C or not;
19:         if found then
20:            Remove p′ from C;
21:         end if
22:      end for
23:   end if
24: end for
```

The algorithm that constructs the LSH buckets is the same one used in the original LSH.

4.3 Algorithm effectiveness

The advantage of SimPair LSH over LSH is that the points in the candidate set returned by LSH are pruned by checking the list of similar pairs *SP* without distance calculations. Therefore, it is important to analyze the number of omissions generated by SimPair. To get the advantage, SimPair LSH needs to search *SP* and C for the points to prune, which may take time, although hash indexes can be created to speed up each search to $O(1)$ time. Next, we analyze the factors that affect the gain and cost.

Analysis of pruning To generate a prune from a point p in C, SimPair must first find a point $p\prime$ that is close enough to p from *SP*, where close enough or not close enough depends on $|d(q, p) - \tau|$. If $|d(q, p) - \tau|$ is large, SimPair LSH has a higher chance of finding a $p\prime$. Another factor that can affect the chance of finding $p\prime$ from *SP* is the size of *SP*. It is clear that a large set of *SP* increases the chance of finding $p\prime$ of p.

Finding $p\prime$ of p does not necessarily lead to a prune. The condition for a prune to occur is that $p\prime$ occurs in C. According to the property of LSH hash functions, points near q have a higher probability of appearing in C. In other words, $d(q, p\prime)$ determines the chance of generating a prune. Although $d(q, p\prime)$ is not precisely known, a bound on this distance can be derived from $d(q, p)$ and the threshold $|d(q, p) - \tau|$ for "close enough."

Cost analysis To achieve pruning, SimPair LSH must incur certain costs, including time and space costs. The time cost mainly comes from the searches: find the points that are close enough to p in *SP* and check whether these points are included in C or not. By constructing hash indices for *SP*, searching for p in *SP* takes only $O(1)$ time; constructing hash indices for *SP* also takes $O(1)$ time for each object. When a candidate set C of points is retrieved for query q, all points in the dataset belonging to C are marked in both LSH and SimPair LSH; This is possible because each point in the dataset has a Boolean attribute that indicates whether or not the point is included in C. The purpose of this attribute is to remove duplicate points when C is created. Duplicates may appear in C because a point may appear in multiple LSH hash buckets. Note that after the search is complete, the Boolean attributes must be cleared (for both LSH and SimPair LSH), which takes $O(|C|)$ time if all points in C are also maintained in a linked list. Pruning requires another Boolean attribute for each point, indicating whether the point has been pruned or not. With the Boolean attributes, searching for $p\prime$ in C takes $O(1)$ time. The time cost is mainly due to the searching for $p\prime$ in C, since for each p there may be multiple $p\prime$ and thus multiple searches in C.

In addition to the time cost, SimPair LSH also incurs an additional space cost for storing *SP* compared to LSH. This cost is limited by the available memory. In our approach, we always limit the size of *SP* based on two constraints: (i) the similarity threshold θ (for the similar point pairs stored in *SP*) is limited to the range $(0, \tau]$; (ii) the size of *SP* must not exceed a constant fraction of the index size (e.g., 10%).

5 Pruning prediction

In this section, we propose an algorithm to predict a lower bound on the number of elements pruned by SimPair LSH compared to LSH. As we saw in Sect. 3, SimPair LSH requires maintenance and access to SP. This leads to a computational cost that could be compensated by pruning elements in the candidate set. Finally, by predicting pruning in advance, we can know whether the overhead is worth it.

5.1 The intuition

According to the pruning analysis presented in [11], a lower bound on the number of prunes given a query $\boldsymbol{q}$ can be estimated. The key idea is as follows: take some sample points $\boldsymbol{p}$ from C and compute $d(\boldsymbol{q}, \boldsymbol{p})$ for each $\boldsymbol{p}$. Based on the sample, estimate the distribution of the different $d(\boldsymbol{q}, \boldsymbol{p})$ for all $\boldsymbol{p}$ in C. From $d(\boldsymbol{q}, \boldsymbol{p})$ we can derive an upper bound of $d(\boldsymbol{q}, \boldsymbol{p'})$ according to the triangle inequality. Thus, we can estimate a lower bound on the probability that $\boldsymbol{p'}$ occurs in C, and correspondingly a lower bound on the number of prunes.

Again, using the small fraction of sample points from C, SimPair LSH can check *SP* and find out if there is any point $\boldsymbol{p'}$ that is close enough to $\boldsymbol{p}$ such that $d(\boldsymbol{p'}, \boldsymbol{p}) < |d(\boldsymbol{q}, \boldsymbol{p}) - \tau|$. Since the distance $d(\boldsymbol{q}, \boldsymbol{p})$ is known, an upper bound of $d(\boldsymbol{q}, \boldsymbol{p'})$ can be derived according to the triangle inequality. Knowing $d(\boldsymbol{q}, \boldsymbol{p'})$, one can know the probability that $\boldsymbol{p'}$ occurs in C, leading to a prune.

5.2 The pruning prediction algorithm

The algorithm for predicting a lower bound on the number of prunes is described in Algorithm 16. Let us first consider the probability that two points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ are hashed into the same bucket. Such a probability is proportional to its distance $d(\boldsymbol{p}_1, \boldsymbol{p}_2)$ by Eq. (1), which will be simply denoted as d in the rest of the section. Second, in our algorithm, the entire range of distances is divided into several intervals. Given a distance value d, we can then use the function $I(d)$ to determine which interval d falls into. The counter *Count*[] is a histogram for storing the number of points in a given interval. The interval is determined by the collision probabilities of the pairwise distances. For a fixed parameter r in Eq. (1), the collision probability $P(d)$ decreases monotonically with $d = ||\boldsymbol{p}_1 - \boldsymbol{p}_2||_l$, where l in our case where we consider the *Gaussian* distribution, is 2. The optimal value for r depends on the dataset and the query point. However, in [8] it was suggested that $r = 4$ gives good results and therefore we currently use the value $r = 4$ in our implementation.

With this setting of the hash function, there is not much difference for the intervals d within the range [0, 1] in terms of the collision probability of the hash function. Thus, we use fewer intervals. In contrast, the hash collision probabilities

Algorithm 2 Prediction of the number of prunes.

Require: A set C with n d-dimensional points $\boldsymbol{p}$; a distance function $d(\cdot, \cdot)$; a distance threshold τ defining near–duplicates; a query point $\boldsymbol{q}$; a distance interval function $I(d)$; the set of similar point pairs *SP*.
Ensure: Number of prunes *PruneNumber*.
1: Construct a sample set S by selecting s points from C uniformly at random;
2: **for** each point $\boldsymbol{p}$ in S **do**
3: Compute the distance $d(\boldsymbol{q}, \boldsymbol{p})$;
4: Search for $\boldsymbol{p'}$ in *SP* where $d(\boldsymbol{p'}, \boldsymbol{p}) < |d(\boldsymbol{q}, \boldsymbol{p}) - \tau|$;
5: **for** each $\boldsymbol{p'}$ found **do**
6: increment the counter $Count[I(d(\boldsymbol{q}, \boldsymbol{p}) + |d(\boldsymbol{q}, \boldsymbol{p}) - \tau|)]$ by 1;
7: **end for**
8: **end for**
9: Scale up the nonzero elements in the counter by a factor of $|C|/|S|$ and store them back to Count[];
10: PruneNumber = 0;
11: **for** each nonzero element in Count[i] **do**
12: Let d_i be the maximum distance of interval i;
13: Let $P(d_i)$ be the probability that 2 points with distance d_i are hashed to the same value, according to Equation 1;
14: $PruneNumber$ += $Count[i] \cdot P(d_i)$;
15: **end for**
16: Output PruneNumber;

differ significantly for distances within the range [1, 2.5]. We use more intervals for this range of distances.

The policy we use to divide a distance range into intervals is defined as follows. First, set the number of intervals (e.g., 100); assign one interval for range $[0, x_1]$ and one for range $[x_2, \infty]$ within which the hash collision probabilities are similar; assign the remaining intervals to range $[x_1, x_2]$ by dividing the range evenly. Since the function $I(d)$ is determined only by the hash function, cutting the intervals can be done offline before processing the dataset.

5.3 Sampling accuracy

SimPair LSH takes only a small fraction (e.g., 10%) of the points from C. If you increase the number of points in the sample whose distances from $\boldsymbol{q}$ are within an interval, and estimate the value in C, some errors arise.

Theorem 1 *A uniform random sample gives an unbiased estimate for the number of points with certain property, and the relative accuracy is inversely proportional to the number of points, and proportional to the sample size and to the true number being estimated. The standard deviation of the ratio between the estimate and the true value is* $\sqrt{\frac{n}{R}\left(\frac{1}{x} - \frac{1}{n}\right)}$, *where n is the total number of points to be sampled, R is the sample size and x is the true value to be estimated, i.e., the number of points with the property.*

Proof Let us assume that R different points are randomly taken into the sample, and each point with the property has a probability $\frac{x}{n}$ to stay in the sample. Let X_i be an indicator

random variable indicating if the i-th point being sampled is with the property or not. That is,

$$X_i = \begin{cases} 1, & \text{if the sampled point has the property} \\ 0, & \text{otherwise} \end{cases}$$

It can be easily shown that $Pr(X_i = 1) = \frac{x}{n}$ and $Pr(X_i = 0) = 1 - \frac{x}{n}$. Given the observed value $Y = \frac{n}{R}\sum_{i=1}^{R} X_i$, since $E[Y] = \frac{n}{R}\sum_{i=1}^{R} E[X_i] = x$ then it is possible to deduce that Y is an unbiased estimate. Thus, we can conclude that:

$$VAR[Y] = \frac{n^2}{R^2} VAR\left[\sum_{i=1}^{R} X_i\right] = \frac{n}{R} x \left(1 - \frac{x}{n}\right)$$

□

6 Similar pair maintenance and updating

Since the advantage of SimPair LSH is that information about similar pairs (i.e., *SP*) is stored in memory, it is important to manage that properly and efficiently.

6.1 Maintenance

Data structure The set *SP* can be implemented as a two-dimensional linked list. The first dimension is a list of points; the near-duplicates of each point $\boldsymbol{q}$ in the first dimension list are stored in another linked list (i.e., the second dimension), ordered by distances from $\boldsymbol{q}$. Then, a hash index is created over the linked list of the first dimension to speed up lookup operations.

Limit the size of SP Since the total number of all similar pairs for a dataset of n objects can be $O(n^2)$, the question is how to bound the size of *SP*. To this end, we set θ (the similarity threshold for the similar point pairs stored in *SP*) to τ (the similarity threshold for the similarity search). However, if the dataset is very large, *SP* may be too large to fit in memory. Therefore, we impose a second constraint on the size of *SP*: it cannot be larger than a constant fraction of the index size (e.g., 10%). To satisfy the last constraint, we can decrease the value of θ within the range $(0, \tau]$. Clearly, a larger value of θ generates a larger amount of *SP*, which increases the probability of finding $\boldsymbol{p\prime}$ of $\boldsymbol{p}$ in C, triggering a prune. Note that these operations do not require real-time updates.

6.2 Update

Point insertions In a continuous-query scenario, each point $\boldsymbol{q}$ triggers a query before being inserted into the database. That is, all near-duplicates of each newly added point must be found before the new point updates *SP*. So to get the list of similar pairs, we only need to insert the near-duplicates of $\boldsymbol{q}$ that we just found into *SP*. To insert the near-duplicates of $\boldsymbol{q}$ into *SP*, we first sort the near-duplicates by their distances from $\boldsymbol{q}$ and store them in a linked list. Then, we insert $\boldsymbol{q}$ and the linked list into *SP* and update the hash index of *SP* in the meantime. Also, we need to search for each near-duplicate of $\boldsymbol{q}$ and insert $\boldsymbol{q}$ into the corresponding linked lists of its near-duplicates. Within the linked list where $\boldsymbol{q}$ is to be inserted, we use a linear search to find the correct position for $\boldsymbol{q}$ based on the distance between $\boldsymbol{q}$ and the near-duplicate. Note that we could have also created indexes for the linked lists of the second dimension, but this would have increased the space cost. Moreover, updates to *SP* in real time are not necessary, unlike query answering. We can cache the new similar pairs and insert them into *SP* when the data arrival rate is lower.

Point deletions When a point $\boldsymbol{q}$ needs to be deleted from the dataset, we need to update *SP*. First, we search for all near-duplicates of $\boldsymbol{q}$ in *SP* and remove $\boldsymbol{q}$ from each of the linked lists of the second dimension of near-duplicates. Then we need to remove the linked second dimension list of $\boldsymbol{q}$ from *SP*.

Buffering SP updates The data arrival rate is indeed irregular, i.e., there may be a large number of queries at one time and very few queries at another time. As mentioned earlier, the updates of *SP* (inserting a point and decreasing the size) can be buffered and processed at a later time when the processor is not overloaded due to the irregular data arrival rate. This buffering mechanism guarantees a real-time response to the similarity search. Note that deleting a point cannot be buffered, as this may lead to inconsistent results.

7 Siamese network

Both our proposed method SimPair LSH and the original LSH work with feature vectors extracted from objects. In this section, we present the features extracted using a Siamese network for images.

7.1 Network structure

The architecture of the Siamese neural network is shown in Fig. 1 and is designed for input image pairs. The specific architecture consists of two symmetric CNNs with identical architectures and parameters. Each CNN follows the AlexNet architecture with eight layers; the first five are convolutional layers and the last three are fully connected layers. The last layer, namely the FC8 layer, is a dense layer with $d = 510$ dimensions and is used to obtain the feature vector representation for the input image, as described in Sect. 8.3. Thus,

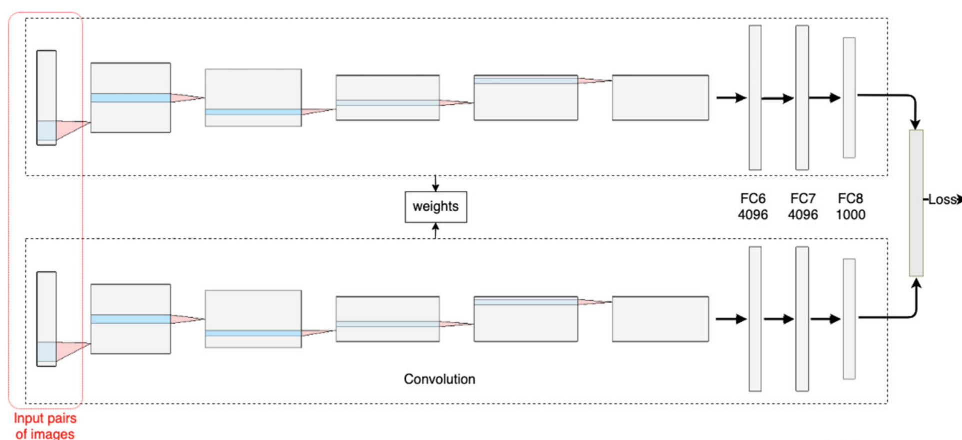


Fig. 1 The Siamese network

this workflow generates two different representations of the dense layer, i.e., the feature vectors, of length d for each image of the input pair. The two feature vectors are then used to compute a loss, which is used in backpropagation for the learning process described below. Although the two feature vectors d are learned differently for the two input images, the particular architecture of the Siamese network enforces that the weights of the two separate AlexNet branches are the same. In Fig. 1, we indicate the weight coupling by the central box containing *weights*, which represents a common set of weights.

7.2 Network training

Each CNN of the Siamese network follows the AlexNet architecture and its parameters are the initial values of the original AlexNet trained on 1000 classes of the ImageNet dataset. We exclude the top of the network and add our own dense layer FC8 with d neurons. We freeze the weights of all pre-trained layers of the model up to layer FC7. This is important so as not to affect the weights that the model has already learned.

The last layer of AlexNet, i.e., FC8, is a dense layer with $d = 510$ dimensions. Each of the two CNNs outputs two separate vectors of length d from FC8, which are used as representations for the input images i_i and i_j. These vectors are denoted as o_i and o_j in this paper, respectively, for each input image of the input pair, as described in Sect. 8.3. We train the Siamese network by determining the Euclidean distance between the two output image representations o_i and o_j. We use the standard formulation for contrastive loss [15] as follows:

$$Loss(o_i, o_j) = \begin{cases} (ED(o_i, o_j))^2 & if\ [o_i, o_j] \in NN \\ (max\{0, 1.0 - ED(o_i, o_j)\})^2 & if\ [o_i, o_j] \in DP \end{cases} \quad (3)$$

where we denote by $ED(o_i, o_j)$ the Euclidean distance between two image embeddings o_i and o_j. NN denotes a set of image pairs each containing two similar images, the near neighbors; in such a case, the loss is the square of the Euclidean distance, i.e., the deviation from the expected value. For dissimilar images contained in the dissimilar pairs set DP, we expect the distance to be as far as possible, ideally at a distance of 1.0 or more. In Sect. 9.3, the creation of the near-neighbor NN-set is illustrated. The Siamese network is trained simply by inputting pairs of images and backpropagating the losses through the layers.

8 Embedding

In this section, we will describe the three methods used to extract features from images and create their vector representations: (i) dense color histograms; (ii) color descriptor SIFT; (iii) indexing construction from Siamese network.

8.1 Dense color histogram

Each image within the dataset is converted to a 510-dimensional vector using the standard HSV histogram method [14]. At the beginning, the color image is divided into three different color channels, each with a length of 170. Then for each channel the histogram is calculated and normalized. The last step is to concatenate each histogram into a compact fixed length feature vector. In the end, each element of the vector represents the percentage of pixels within a given HSV interval.

8.2 Color descriptor SIFT

The descriptor SIFT is based on the extraction of scale-invariant keypoints [26] through each of the three color channels of the image. It is robust to image transformations since it describes local spatial information in the image

matrix. We used a powerful keypoint extraction method using the Harris affine point detector [23]. The color descriptor SIFT is created by concatenating the SIFT descriptors computed for each channel. The result is a vector of length 510.

8.3 Indexes Construction from the Siamese network

The Siamese network is used as a method for learning the representation of the input images: The input image dataset $I = \{i_1, i_2, \ldots, i_n\}$ is transformed into a dataset in a vector space $\mathbb{R}^d$, where d is set as a hyperparameter by the user. Formally:

$$\{i_1, i_2, \ldots, i_n\} \xrightarrow[\text{learn}]{\text{representation}} \{o_1, o_2, \ldots, o_n\} \tag{4}$$

where $o_i \in \mathbb{R}^d$ is a vector space representation of the image i_i. In principle, it is desirable that the output dataset, denoted O, retains the similarity property of the image dataset in the input. In particular, it is desirable that pairs of images that both belong to the same class and are nearly identical are, on average, closer to each other than pairs of dissimilar images that belong to different classes. For this purpose, the features for the images are extracted from the last dense layer FC8 of the trained Siamese network: We choose a branch of the network consisting of an AlexNet and pass each image of the initial image dataset through it, then extract the vector of length 1000 from FC8. The extracted image vector representations are indexed using LSH-based indexes construction. Given an image query i_q, a set of candidate images in the vicinity, denoted as set C, is retrieved based on the constructed indexes. Specifically, let o_q and o_i be the FC8 layer feature vectors of the query image i_q and the ith candidate image i_c^C. The distance between i_q and i_c^C is the Euclidean distance $||o_q - o_i||_2$. If the distance between o_q and o_i is small, the probability that the two images are near neighbors is high. The top k near neighbor images are retrieved. The input of our Siamese network is similar and dissimilar image pairs and their vector representations are suitable for finding near duplicates of images.

9 Experiments and evaluations

We evaluated our work on different real datasets and under different evaluation criteria. In the following sections, we describe the datasets, the criteria, and results of our evaluation.

Experiments were performed on a 2.6 GHz 6-core Intel Core i7 computer, 18 GB DDR4 memory running macOS Catalina 10.15.7. Both data points and LSH indexes were loaded into main memory. The source code of the original LSH was obtained from E2LSH [1] and used without any modification.

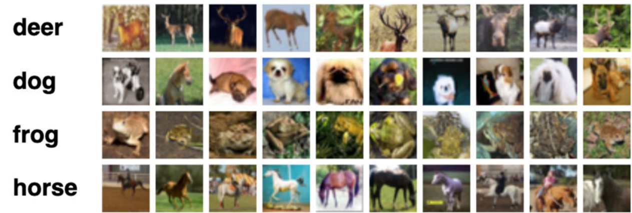


Fig. 2 Near neighbor examples in CIFAR-10

9.1 Datasets

We test the effectiveness of SimPair LSH on the two datasets summarized below.

9.2 Network structure

- *CIFAR-10*: It contains 60,000 color images of 32x32 pixels evenly divided into 10 classes. The classes are completely mutually exclusive [19]. Images of the same class are treated as near neighbors; in Fig. 2, for example, the near neighbors for four different classes are given per line.
- *Flickr*: We sent 26 random queries to Flickr [12] and retrieved all images within the result set. After removing all results with less than 150 pixels, we obtained about 14,000 images. We then created 3 nearly identical versions for each image using image compression, scaling, noise blurring, and image encoding format conversion. Figure 3 shows the generated near neighbors for an example image. In total, this gave us 56,000 images.

For all the image datasets described above, we used the three embedding methodologies reported in Sect. 8.

9.3 Training dataset construction

The Siamese network depicted in Fig. 1 is trained using two image pairs sets.

- In *CIFAR-10*, images of the same class are treated as near neighbors and inserted into *NN*. To distinguish images of different classes, we choose x random images from one class and pair them with x random images from the

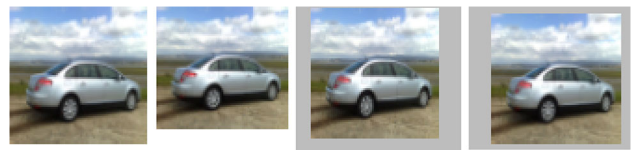


Fig. 3 Near neighbors for an example image in Flickr

remaining classes. The resulting pairs are included in *DP*, the dissimilar pairs set.

- In *Flickr*, each image is already connected to 3 near neighbors. We select y near neighbor image pairs and record them in *NN*. To create *DP*, we select y random images and pair them with random images that do not belong to their near duplicates.

9.4 Evaluation criteria

In this section, we describe the criteria that we use to evaluate the performance of our approach.

Predicted prunes In order to validate the pruning prediction algorithm presented in Sect. 5, we counted the percentage of elements pruned with different values of similar pair threshold τ.

Accuracy For accuracy assessment, we used the mean Relevance Precision (mRP). The search algorithm returns k images, sorted by their Euclidean distance from the query. The mRP is defined as:

$$mRP(k) = \frac{1}{k} \sum_{i=1}^{k} Rel(i) \tag{5}$$

with $Rel(i)$ equal to 1 if the i-th candidate is a near neighbor of the queried image, and 0 otherwise. This is an absolute measure of accuracy. In the CIFAR-10 dataset, we experimented with different values of k. In the Flickr dataset, we fixed k to 4, since for each query there are at most 4 near neighbors, one of which is the exact copy of the query image.

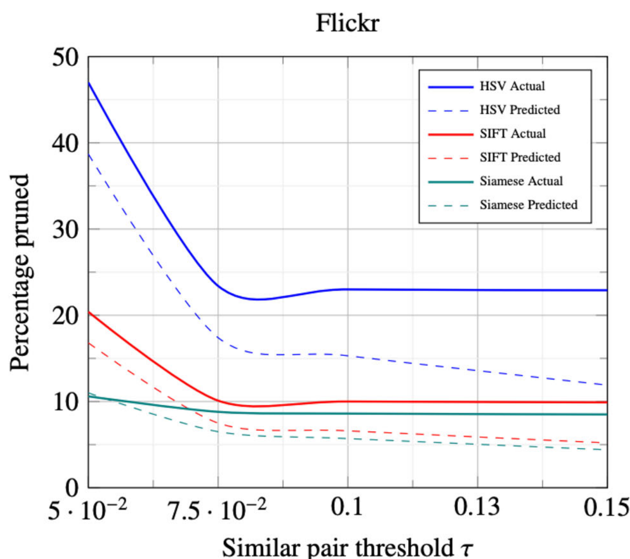
Acceleration factor Given an image query and a dataset with a total number of images given by n, an exhaustive search goes through all n elements to compute and retrieve the near neighbors of the query. Considering another search algorithm that retrieves n_r image candidates, where $n_r \leq n$, the acceleration factor is defined as n/n_r and represents the estimate of the detection efficiency relative to the exhaustive search.

9.5 Results

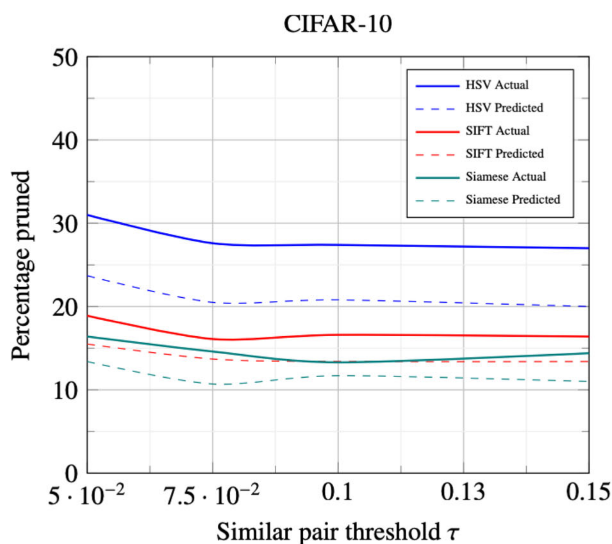
In this section, we report and discuss the results of our evaluation. In order to validate our approach, we randomly selected 100 objects from each dataset as query objects. We tried other values for the number of queries, from 100 to 1000, without noting any change in performances. Thus, we kept it to 100 in the following experiments.

The results presented in the rest of this section represent values averaged over the query objects.

Unless explicitly stated, the results that we report in the rest of this section were achieved setting the parameters as



(a)



(b)

Fig. 4 Avg. percentage of pruned points by similar pair threshold τ per embedding method

follows: distance threshold for near–duplicates $\tau = 0.1$, distance threshold used for filling the similar pair list *SP* $\theta = \tau = 0.1$, number of LSH functions $L = 136$, each with size $k = 12$, and success probability $\lambda = 90\%$.

9.5.1 Pruning prediction

We test our pruning prediction algorithm (Sect. 5) by predicting the number of prunes with different values of similar pair threshold τ.

In Fig. 4, we report the average percentage of pruned elements for different values of the threshold τ for each of

the three embedding methods for both datasets. It can be observed that the pruning prediction (dashed lines) always underestimates the actual number of pruned points (solid lines), thus providing a reliable lower bound. The higher percentage of pruned points at $\tau = 0.05$ is due to the fact that the size of the candidate set is small and the variance of the percentage is higher. Moreover, the difference between the lower bound and the actual number of pruned points is almost constant (between 5 and 10%), especially for low values of τ. This means that the trend of the prediction resembles the real profile of the pruned points: Apart from a scale value (i.e., the difference between prediction and real number), the actual number of pruned points can be estimated fairly accurately. This is less true for higher values of τ, where the difference between the lower bound prediction and the actual number of pruned points varies more. Another consideration relates to the fact that we have a higher percentage of pruned elements with HSV and SIFT than with the Siamese network embedding. This is due to the fact that SimPair LSH using the new embedding gives more accurate results, i.e., fewer false positives than the other two methods, as can be seen in the next Sect. 9.5.2. This analysis motivates us to choose $\tau = 0.1$ for the remaining experiments.

9.5.2 Accuracy and acceleration factor

As an absolute measure of accuracy we used the mean Relevance Precision mRP (Eq. 5), which is a function of the parameter k. If the number of duplicates per query is known, the parameter k can be chosen using a greedy approach that lets the parameter take a value that is an order of magnitude lower than the number of duplicates per query and then increases to the exact number. The more similar the duplicates are to the original images, e.g., due to the specific method used to generate them, the more the k parameter can approach the exact number of neighbors without significantly affecting performance.

In *CIFAR-10* dataset, we experimented with different values of k. Figure 5 shows the values of mRP for different embedding methods as the number of top k returned images changes. It can be seen that the curves are quite stable over k. Our new method based on Siamese network embedding gives more accurate results than the other methods based on HSV and SIFT embeddings. This experiment motivates us to choose $k = 200$ for the following analysis.

LSH can speed up the retrieval of near-duplicate images by increasing the number of hash tables, i.e., increasing the parameters k and L. This has an immediate effect on increasing the acceleration factor. Figure 6 shows the curves of $mRP(200)$ versus the acceleration factors for each embedding method, both when using the original LSH and our proposed SimPair LSH. Each point of the curves, starting

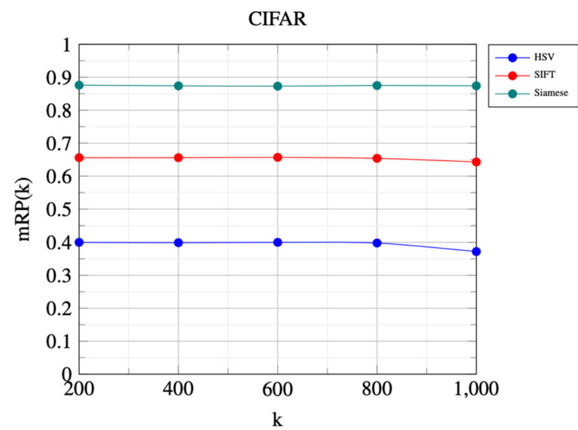


Fig. 5 Mean relevance precision (mRP) with different values of k per embedding method

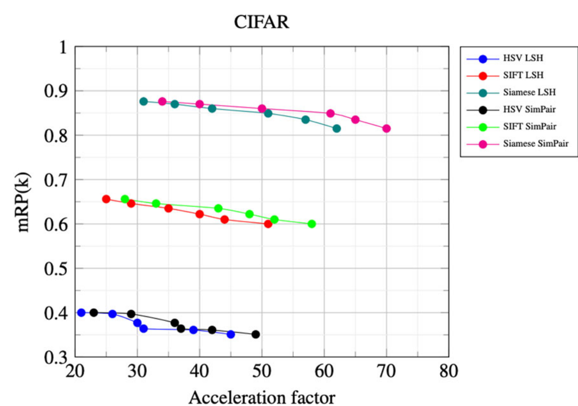


Fig. 6 Mean relevance precision (mRP) with $k = 200$ and acceleration factor for LSH and our proposed SimPair LSH per embedding method

from left to right, corresponds to the following k and L parameter pairs:

k	10	12	14	16	18	20
L	78	136	210	351	561	908

Using the same embedding method, the curves of the original LSH and the SimPair LSH (e.g., *Siamese LSH* and *Siamese SimPair*) obtain the same mRP values corresponding to the same k and L parameters. This is due to the fact that SimPair LSH is based on LSH and cannot improve its mRP, only the acceleration factor. As can be observed, our approach based on SimPair LSH with Siamese network embeddings outperforms all other methods.

In *Flickr* dataset, we fixed k to 4, since for each query there are at most 4 near neighbors, one of which is the exact copy of the query image. The k parameter can be set to the exact number of near neighbors because the method used to generate them produces nearly identical versions through

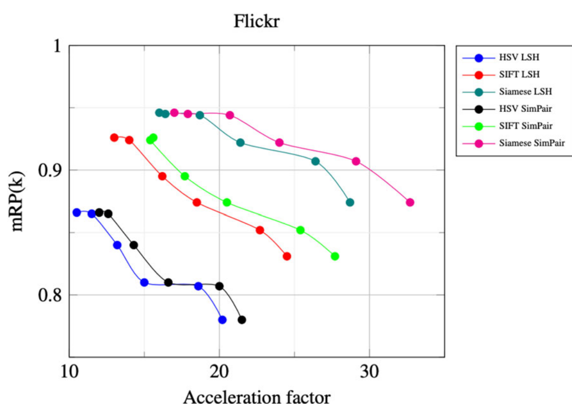


Fig. 7 Mean relevance precision (mRP) with $k = 4$ and acceleration factor for LSH and our proposed SimPair LSH per embedding method

image compression, scaling, noise blurring, and format conversion of the image encoding. Figure 7 reports the values of $mRP(4)$ versus the acceleration factors for each embedding method, using both the original LSH and our proposed SimPair LSH. Each point on the curves applies to the same k and L parameter pairs as adopted for the CIFAR-10 dataset.

In summary, our approach based on SimPair LSH with Siamese network embeddings also performs better than all other methods for Flickr images, as already observed for CIFAR-10. However, for Flickr, the mRPs range from 0.78 to 0.95, while for CIFAR-10 we have a wider range from 0.39 to 0.87. This is due to the different constructions of the datasets. For Flickr, the near duplicates are an extended version of the same image, while for CIFAR-10 we use different images from the same class as near neighbors. Specifically for CIFAR-10, our approach based on SimPair LSH with Siamese network is able to identify image features that are characteristic of the same class, while HSV and SIFT fall short in this regard.

9.5.3 Memory analysis

The amount of memory required by LSH can be derived from L. Since each hash table stores the identifiers of all n points in the dataset, each of which occupies 12 bytes (as implemented by Andoni and Indyk [2]), the LSH space cost is equal to $12nL$.

SimPair LSH incurs an additional space cost for storing SP compared to LSH. This cost is bounded by the available memory. In our approach, we limit the size of SP based on two constraints: (i) the similarity threshold θ (for the similar point pairs stored in SP) was set equal to τ; (ii) the size of SP must not exceed a constant fraction of the index size (i.e., 10%). To limit the size of SP, we considered the lowest value for parameter L, i.e., 78. So for CIFAR-10, we had an upper bound of $SP = 10\% \, (12 * 60,000 * 78)$, which resulted in 5.6 MB. For Flickr, we had an upper bound of $SP = 10\% \, (12 * 56,000 * 78)$, which yielded 5.2 MB.

10 Conclusions

In this paper, we study the problem of range search in an incremental way based on a well-known technique, locality sensitive hashing (LSH). We propose SimPair LSH, a new approach to improve the running time of LSH by exploiting a certain number of existing similar point pairs. SimPair LSH speeds up the search by precomputing a certain number of pairwise similar points in the dataset and storing them in memory. These are used at query time to reduce the candidate set of points C retrieved by LSH, resulting in fewer comparisons.

We also present an algorithm to predict a lower bound on the number of items pruned by SimPair LSH. In our experiments, the pruning prediction algorithm always provides a reliable lower bound for the actual number of pruned points. Moreover, the difference between the lower bound and the actual number of pruned points is almost constant for a subset of setups, which means that the profile of the prediction is similar to the actual profile of pruned points except for one scale value. Finally, we describe the procedure for maintaining and updating the information about similar pairs.

Since both our proposed method SimPair LSH and the original LSH work with feature vectors extracted from objects, we experiment with three methods used to extract features from images and create their vector representations: (i) dense HSV color histograms; (ii) color descriptor SIFT; (iii) indexing construction from Siamese network. In summary, our SimPair LSH-based approach using Siamese network- based feature vectors (namely embeddings) performs better than all other settings on both the Flicker image and CIFAR-10 datasets.

In the future, we plan to investigate other neural network architectures for feature vector extraction.

Acknowledgements This work was partly funded by the SoMeCliCS project under the Volkswagen Stiftung und Niedersächsisches Ministerium für Wissenschaft und Kultur.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the

permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Andoni A (2021) Implementations of LS: E2LSH. http://web.mit.edu/andoni/www/LSH/manual.pdf
2. Andoni A, Indyk P (2008) Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun ACM 51(1)
3. Andoni A, Indyk P, Nguyen HL, Razenshteyn I (2014) Beyond locality-sensitive hashing. In: Proceedings of the twenty-fifth annual ACM-SIAM symposium on discrete algorithms, SODA'14, USA. Society for Industrial and Applied Mathematics, pp 1018–1028
4. Bellman RE (1961) Adaptive control processes—a guided tour
5. Ceroni A, Gadiraju UK, Fisichella M (2015) Improving event detection by automatically assessing validity of event occurrence in text. In: Bailey J, Moffat A, Aggarwal CC, de Rijke M, Kumar R, Murdock V, Sellis TK, Yu JX (eds) Proceedings of the 24th ACM international conference on information and knowledge management, CIKM 2015, Melbourne, VIC, Australia, October 19–23, 2015. ACM, pp 1815–1818
6. Ceroni A, Gadiraju U, Matschke J, Wingert S, Fisichella M (2016) Where the event lies: predicting event occurrence in textual documents. In: Perego R, Sebastiani F, Aslam JA, Ruthven I, Zobel J (eds) Proceedings of the 39th international ACM SIGIR conference on research and development in information retrieval, SIGIR 2016, Pisa, Italy, July 17–21, 2016. ACM, pp 1157–1160
7. Chum O, Philbin J, Isard M, Zisserman A (2007) Scalable near identical image and shot detection. In: CIVR
8. Datar M, Immorlica N, Indyk P, Mirrokni VS (2004) Locality-sensitive hashing scheme based on p-stable distributions. In: SCG
9. Fisichella M (2021) Unified approach to retrospective event detection for event- based epidemic intelligence. Int J Digit Libr 22(4):339–364
10. Fisichella M, Ceroni A (2021) Event detection in wikipedia edit history improved by documents web based automatic assessment. Big Data Cogn Comput 5(3):34
11. Fisichella M, Ceroni A, Deng F, Nejdl W, (2014) Predicting pair similarities for near-duplicate detection in high dimensional spaces. In: Decker H, Lhotská L, Link S, Spies M, Wagner RR (eds) Database and expert systems applications—25th international conference, DEXA 2014, Munich, Germany, September 1–4, 2014. Proceedings, Part II, vol 8645. Lecture notes in computer science. Springer, Berlin, pp 59–73
12. Flickr (2021) https://flickr.com/
13. Gionis A, Indyk P, Motwani R (1999) Similarity search in high dimensions via hashing. In: VLDB
14. Gonzalez RC, Woods RE (2006) Digital image processing, 3rd edn
15. Hadsell R, Chopra S, LeCun Y (2006) Dimensionality reduction by learning an invariant mapping. In: 2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06), vol 2, pp 1735–1742
16. Huang Q, Feng J, Zhang Y, Fang Q, Ng W (2015) Query-aware locality-sensitive hashing for approximate nearest neighbor search. Proc VLDB Endow 9(1):1–12
17. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: STOC
18. Jafari O, Maurya P, Nagarkar P, Islam KM, Crushev C (2021) A survey on locality sensitive hashing algorithms and their applications. CoRR. arXiv:abs/2102.08942
19. Krizhevsky A, Nair V, Hinton G (2009) The cifar-10 dataset. Technical report
20. Liu W, Wang H, Zhang Y, Wang W, Qin L (2019) I-LSH: I/O efficient c-approximate nearest neighbor search in high-dimensional space. In: 35th IEEE international conference on data engineering, ICDE 2019, Macao, China, April 8–11, 2019. IEEE, pp 1670–1673
21. Lu K, Kudo M (2020) R2LSH: nearest neighbor search scheme based on two-dimensional projected spaces. In: 36th IEEE international conference on data engineering, ICDE 2020, Dallas, TX, USA, April 20–24, 2020. IEEE, pp 1045–1056
22. Lv Q, Josephson W, Wang Z, Charikar M, Li K (2007) Multi-probe LSH: efficient indexing for high-dimensional similarity search. In: VLDB
23. Mikolajczyk K, Tuytelaars T, Schmid C, Zisserman A, Matas J, Schaffalitzky F, Kadir T, Gool LV (2005) A comparison of affine region detectors. Int J Comput Vis 65(1–2):43–72
24. Teixeira T, Teodoro G, Valle E, Saltz JH (2013) Scalable locality-sensitive hashing for similarity search in high-dimensional, large-scale multimedia datasets. CoRR. arXiv:abs/1310.4136
25. Wang Q, Guo Z, Liu G, Guo J (2012) Boundary-expanding locality sensitive hashing. In: 8th International symposium on Chinese spoken language processing, ISCSLP 2012, Kowloon Tong, China, December 5–8, 2012. IEEE, pp 358–362
26. Zhang J, Marszalek M, Lazebnik S, Schmid C (2006) Local features and kernels for classification of texture and object categories: a comprehensive study. In: 2006 Conference on computer vision and pattern recognition workshop (CVPRW'06), pp 13–13
27. Zheng B, Zhao X, Weng L, Hung NQV, Liu H, Jensen CS (2020) PM-LSH: a fast and accurate LSH framework for high-dimensional approximate NN search. Proc VLDB Endow 13(5):643–655

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.