Information Systems 91 (2020) 101510

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

An efficient algorithm for approximated self-similarity joins in metric spaces

Sebastián Ferrada [a,*], Benjamin Bustos [a], Nora Reyes [b]

[a] Millennium Institute for Foundational Research on Data, Department of Computer Science, Universidad de Chile, Beauchef 851, Santiago, Chile
[b] Departamento de Informática, Universidad Nacional de San Luis, Ejército de Los Andes 950, San Luis, Argentina

ARTICLE INFO

Article history:
Received 11 September 2019
Received in revised form 28 January 2020
Accepted 19 February 2020
Available online 24 February 2020
Recommended by Dennis Shasha

Keywords:
Similarity joins
kNN
Approximated nearest neighbors
Algorithms
Metric spaces

ABSTRACT

Similarity join is a key operation in metric databases. It retrieves all pairs of elements that are similar. Solving such a problem usually requires comparing every pair of objects of the datasets, even when indexing and ad hoc algorithms are used. We propose a simple and efficient algorithm for the computation of the approximated k nearest neighbor self-similarity join. This algorithm computes $\Theta(n^{3/2})$ distances and it is empirically shown that it reaches an empirical precision of 46% in real-world datasets. We provide a comparison to other common techniques such as Quickjoin and Locality-Sensitive Hashing and argue that our proposal has a better execution time and average precision.

1. Introduction

The self-similarity join in metric spaces is a relevant operation in many domains such as, multimedia retrieval [1,2], the creation of similarity graphs [3,4], pattern recognition [5], and even others outside the computer sciences such as protein similarity computation [6]. Given a set of objects $\mathbb{D} \subseteq \mathbb{U}$, the self-similarity join $\mathbb{D} \bowtie_s \mathbb{D}$ is defined as the problem of finding all pairs (x, y) such that $x, y \in \mathbb{D}$, $x \neq y$ and x *is similar to* y using similarity criteria s. A naïve implementation for solving this problem, called a nested-loop, can be as costly as $O(n^2)$ similarity comparisons, where $|\mathbb{D}| = n$. Similarity is often measured with (metric) distance functions. The pair of the universe of objects and a metric function $(\mathbb{U}, \delta)$ is called a metric space.

The notion of similarity is diverse and depends on the context of the problem, the nature of the objects and the application to develop. In this paper we focus on metric-based similarity, this is, where dissimilarity is measured using metric distance functions calculated among the objects. Similarity criteria can be range-based or nearest neighbor-based. The first one retrieves all pair of objects with a distance below a threshold. The second one, retrieves a fixed number of objects that are the closest to every object. We argue that for a user to define a *distance threshold* can be odd, since it will greatly depend on the user's knowledge of the metric space and its the distance distribution. On the other hand, to set a number of similar objects can be more easily expressible and does not require a deep understanding of the metric space. For instance, *retrieve all images that are at a distance lower than three* could be more abstract and less intuitive than *retrieve the three images most similar to this one*.

The properties of the metric distances allow us to prune elements from comparison when computing a self-similarity join. The pruning can be achieved, for instance, by building indices over the objects and discard complete regions of the space, thus saving computation time. However, there are interesting scenarios where building an index is impractical. For example, while resolving similarity queries in image databases after applying a semantic filter: here, the execution plan for such a query should be to first retrieve all the images that satisfy the filter and then compute the self-similarity join among such images. In this case, building an index is pointless since it will not be used again unless we are computing the exact same query, which is the case for most queries of this kind.

A current algorithm for range-based self-similarity join that has proven to be effective is Quickjoin [7]. This algorithm recursively divides the space until the formed groups are small enough to make a nested loop. It also keeps window partitions to check if relevant pairs are sorted into different regions. The complexity of Quickjoin depends on the fraction of elements that lie in these window partitions. The bigger the fraction, the bigger the chance the algorithm needs to compute a quadratic amount of distances. As far as we know, it is not available a generalization of this algorithm for kNN-based self-similarity joins. [8] proposes to run

* Corresponding author.
E-mail addresses: sferrada@dcc.uchile.cl (S. Ferrada), bebustos@dcc.uchile.cl (B. Bustos), nreyes@unsl.edu.ar (N. Reyes).

https://doi.org/10.1016/j.is.2020.101510
0306-4379/

Quickjoin two times in order to find the k closest pairs of objects in a set, which is not what is required for this problem.

In this paper we present a simple algorithm that computes an approximation of the k nearest neighbor self-similarity join problem in metric spaces. Our algorithm is a generalization of a previous work that presented an heuristic for solving the 1NN problem [9]. The algorithm presented here computes $\Theta(n^{3/2})$ distances even in the worst case and gives an approximation with up to 46% of precision for 1NN self-similarity queries and 39% for 16NN. We show how the algorithm performs with three different real datasets: the English dictionary, very high-dimensional deep-feature vectors, and a massive collection of classic visual descriptors.

The rest of the paper is organized as follows: Section 2 formalizes the definitions and notions needed to follow the work. Later, Section 3 reviews the related work on similarity joins. Section 4 describes the proposed algorithm, along with its parameters, complexity and implementation. Section 5 shows an experimental evaluation of the algorithm. Section 6 concludes the paper with final remarks and discussion of future work.

2. Preliminaries

The concepts, definitions and principles regarding similarity search and similarity join that are necessary to follow the work are presented in this section.

We focus on similarity joins in metric spaces. A metric space is a pair $(\mathbb{U}, \delta)$, where $\mathbb{U}$ is the universe of objects, and $\delta : \mathbb{U} \times \mathbb{U} \rightarrow \mathbb{R}_0^+$ is a metric distance function, i.e., is symmetric ($\delta(x, y) = \delta(y, x) \forall x, y \in \mathbb{U}$), provides that two objects at distance 0 from each other are identical ($\delta(x, y) = 0 \Rightarrow x = y$), and holds the triangle inequality ($\forall x, y, z \in \mathbb{U}, \delta(x, y) \leq \delta(x, z) + \delta(z, y)$). Metric distances are commonly used as similarity measures, where two objects that have a low distance between them have a higher similarity than those further apart.

There are mainly two types of similarity queries in metric spaces $(\mathbb{U}, \delta)$, given a query object $q \in \mathbb{U}$ and a database $\mathbb{D} \subseteq \mathbb{U}$. Range queries aim to find all objects in the database that are at distance at least ε from the query point:

$$Q_\varepsilon(\mathbb{D}, q) = \{r \in \mathbb{D}, \delta(r, q) \leq \varepsilon\}.$$

Nearest neighbor queries aim to find a subset of k objects that have the lowest distance to q, i.e.

$$Q_k(\mathbb{D}, q) = X \subseteq \mathbb{D}, |X| = k, \forall r \in X, r' \in \mathbb{D} - X : \delta(r, q) \leq \delta(r', q).$$

Along the same lines, the database operation of similarity join can be defined between two given databases S and T:

$$S \bowtie T = \{(s, t) | s \in S, t \in T, s \neq t \wedge s \text{ is similar to } t\}.$$

Here, the predicate *is similar to* implies that there is a metric distance involved and a required kind of similarity query, whether range or nearest neighbor queries.

A special case of similarity join occurs when $S = T$ and it is called self-similarity join and the definition is the same than in a regular similarity join.

The complexity of the algorithms that solve this join is usually measured in terms of the number of distances that must be computed. The brute-force algorithm that resolves a similarity join is called Nested-Loop and computes the distances from all the elements of T to all the elements of S, thus computing a total of $|T| \cdot |S|$ distances. If the join is a self-similarity join, the fact that metric distances are symmetric can be exploited to reuse previously computed distances, in which case, the Nested-Loop will compute $\binom{|T|}{2}$ distances. Further pruning in the distance computation can be made taking advantage of the triangle inequality: if $\delta(x, y)$ and $\delta(y, z)$ are known, we can skip the computing of $\delta(x, z)$ if $\delta(x, y) + \delta(y, z) > r$ where $r = \varepsilon$ in range queries, or is the current maximum minimum distance found in kNN queries.

For the sake of computational speed, several approximated techniques have been developed. We call the result of an approximated similarity query as Q_a. From here on, we focus on kNN queries, therefore $Q_a(\mathbb{D}, k, q)$ is an approximation of $Q_k(\mathbb{D}, q)$. The correctness of such a technique can be measured in terms of its precision, and the distance difference between the actual kth nearest neighbors and the reported ones. Precision is defined as the ratio of correctly found neighbors and the number of desired neighbors:

$$\text{prec}(\mathbb{D}, k, q) = \frac{|Q_k(\mathbb{D}, q) \cap Q_a(\mathbb{D}, k, q)|}{k}.$$

Distance difference is measured with respect to the ith nearest neighbor, for $1 \leq i \leq k$. Usually the sum of all ratios is used:

$$\text{diff}_i(\mathbb{D}, q) = \frac{\delta(q, Q_a(\mathbb{D}, k, q)_i)}{\delta(q, Q_k(\mathbb{D}, q)_i)}.$$

3. Related work

During the past years, several algorithms, heuristics, and data structures for solving the various types of similarity joins have been proposed. In this section, we review the related literature.

Gionis et al. [10] proposed the Locality-Sensitive Hashing (LSH). This method hashes all objects using functions that ensure that objects that are closer in the metric space have a higher collision probability than those further apart. A kNN query (x, k) is processed by hashing x and retrieving the elements that collide with it. For the purposes of LSH, Datar et al. [11] propose to use random hash functions taken from a p-stable probability distribution in order to mimic the behavior of an L_p distance. LSH can be used to implement a kNN self-similarity by using all objects in the dataset as query object for the kNN search.

Böhm and Krebs [12] identified the kNN join as an important database primitive for implementing data mining methods. They proposed an algorithm for computing the kNN join using the so-called MuX index. Their algorithm and methods are focused on solving the kNN join on vector spaces, aiming at minimizing CPU and I/O costs.

Jacox and Samet [7] proposed the Quickjoin algorithm for similarity joins. It divides the space in ball cuts using data points as pivots. Given two pivots, it uses one as a center and the distance between them to define a radius. Then, it splits the data into the vectors inside and outside the ball, then proceeds recursively until the groups are small enough to perform a nested loop. It keeps window partitions in the frontier of the ball in case there are pairs relevant for the result where each vector ends in different partitions. Quickjoin is shown to have a complexity of $O(n(1 + w)^{\lceil \log n \rceil})$, where w is the average of the fraction of elements lying within the window partitions. Hence, it has a quadratic worst case. Quickjoin was intended for range-based similarity joins, however, Fredriksson and Braithwaite propose several extensions to Quickjoin [8], including an algorithm for kNN similarity join which runs Quickjoin two times over the data and returns the correct results given a set of assumptions: a correct parameter for the termination of the recursion, the join being a self-join, and balanced partitioning. However, the definition of kNN similarity join presented in [8] is in fact a k-Distance join: it is intended to compute the k pairs of objects in the entire dataset that are the closest in the defined space. Instead, we propose that for each element of the dataset we find its k nearest neighbors.

Yu et al. [13] propose a dynamic method so called *kNNJoin+* for kNN similarity join. This method has the ability to deal with

data updates, and it focuses mainly with high-dimensional data. It also allows one computing reverse kNN queries.

Yao et al. [14] proposed kNN and kNN-join algorithms that can be implemented using SQL primitives. Their algorithms do not depend on the distance function defined by the user. This allows them to take advantage of the query optimizer that can produce an efficient query plan. Their proposal support exact and approximate similarity searches and kNN-joins. Their work only considers vector spaces with L_p norms.

Lu et al. [15] propose an algorithm for computing a kNN join using MapReduce. It divides the objects into groups, choosing the center from one dataset and use them to partition the other dataset. The division into groups is done using a Voronoi partition. At each partition, the algorithm keeps track of the closest objects to the center of the partition and it stores the maximum and minimum distance from the objects to their respective partition center. Each of these groups is processed by a reducer for computing the kNN join. The algorithm may replicate objects in several groups for obtaining the exact answer. Other methods for computing similarity joins based on MapReduce have been proposed by Silva and Reed [16], Wang et al. [17], Song et al. [18], Rong et al. [19], Chen et al. [20], Cech et al. [21], and Moutafis et al. [22].

Pearson and Silva [23] propose an algorithm based on similarity searches for the case of similarity joins. The algorithm is an extension of the eD-Index [24], and it supports a join operation over two individually indexed datasets. The authors also proposes how to fine tune the parameters of the eD-Index. The main idea of the algorithm is to index each of the dataset independently with a D-index. Both indexes share the same structure, use the same number of levels, and the same pivots for each level. Then, they use the similarity join algorithm proposed for the self-join over the D-index. The original algorithm is modified so that each pair given as part of the result contains an object from a different dataset.

Chen et al. [25] propose index structures for range joins in uncertain metric data. Given two sets $\mathbb{U}$ and $\mathbb{V}$ of uncertain objects, a distance function d, a parameter r and a probability threshold θ, a probabilistic range join returns all pairs of uncertain objects $(u, v) \in \mathbb{U} \times \mathbb{V}$ such that $Pr(d(u, v) \leq r) \geq \theta$. For computing the probabilistic range joins, they define two index structures for secondary memory, the so-called "uncertain pivot B+ -tree" (UPB-tree) and the "uncertain pivot B+ -forest" (UPBforest). Both indexes are based on the B+ -tree.

Finally, in our previous work [9] we propose an heuristic for computing approximated 1NN self-similarity join. The algorithm consists in selecting $\sqrt{n}$ centers, form $\sqrt{n}$ groups of size $O(\sqrt{n})$, and finally it performs a nested loop on each group separately. The algorithm computes $\Theta(n^{3/2})$ distances and has an average precision of 31%, where 80% of the answers lie within the actual 10NN.

4. The algorithm

The algorithm here presented is a generalization of our previous work [9], extending self-similarity joins from 1NN to kNN and improving its performance. Given a dataset $\mathbb{D}$, with $|\mathbb{D}| = n$, and a metric function δ, the algorithm selects $\sqrt{n}$ objects as centers, where each center defines a group. Then, the algorithm distributes the remaining objects into the groups such that each object is in the group with the closest center, with respect to distance δ. Groups have a maximum size of $c\sqrt{n}$ objects, where c is a parameter. Given the size restriction, if the group where an object *should* be sorted into is already full, then it is sorted into the next closest group. The criteria for selecting the centers and the algorithm to form the groups can be re-defined by the users. When grouping is finished, the algorithm searches for the k nearest neighbors of each object from a fixed set of candidates: the group of the current object and another group that is the closest to the object.

Algorithm 1: Algorithm for approximated kNN self-similarity join.

```
Data: Data, a set of objects; c an integer; k the number of
      neighbors to find
Result: result, a set of pairs of objects
1  centers ← select_centers(Data);
2  groups ← partition(Data, centers, c);
3  result ← φ ;
4  for group ∈ groups do
5      for e ∈ group do
6          do
7              target ← group ∪ next_closest_group(e);
8          while |target|< k;
9          partial ← get_kNN(k, e, target);
10         result ← result ∪ partial ;
11     end
12 end
13 return result
```

Algorithm 1 presents the pseudo-code for the proposed algorithm. Routines `select_centers` and `partition` choose the $\sqrt{n}$ centers and assign the remaining objects into the groups respectively. For each element in a group, the target set is formed to contain the suitable kNN candidates. The `next_closest_group` subroutine finds a group, different from the current one, that is closest to the element e. This means, obtain the group G such that $\delta(G.center, e) - G.radius$ is minimum. The `get_kNN` routine computes the distance between an object e and all elements in *target*, keeping the approximated k nearest neighbors of e.

For the purposes of this work, centers are chosen at random. There are techniques for choosing evenly distributed objects in metric spaces [26] that can be applied at this point; however, the maximum distance of the metric space must be known beforehand, a parameter must be fine-tuned, and it is not provided with a way to select the number of objects chosen, it relies in the intrinsic dimensionality of the space. Yianilos, along with the definition of the VP-trees, describes a way to choose elements as far apart as possible considering the mean and standard deviation of the distance distribution of a subset of the elements [27]; however it requires further parameters to select a subsample of the objects to compute the mean of the distances among them and maximizing the standard deviation.

We propose that the partition process, given a set of centers, tries to assign each object into the group of the closest center if it has space; if not, the object will be assigned to the second-closest group, if it is also full the algorithm will attempt with the next closest, and so on until the object is assigned. The maximum size of the groups is $c\sqrt{n}$, where $c \in O(1)$ is a parameter, usually lower than 10. Fig. 1 shows a 2D example of how the partition of the elements can change as the value of c increases: 5 thousand randomly-generated points are distributed into 71 groups, shown with different colors. Above each graph, the respective value of c is presented. It can be seen that when $c = 1$ the groups do not have clear boundaries separating each other, and as c increases to 2 and 3, the groups seem better-defined and well-separated from the others. Using higher values of c do not show much change in the partition results; however, in higher dimensions it can have an impact. Other partition criteria can be used, as long as the number of centers and maximum size of groups remain as $O(\sqrt{n})$. We attempted to assign each object to the group that needed to grow the least in order to include it; however we found that the

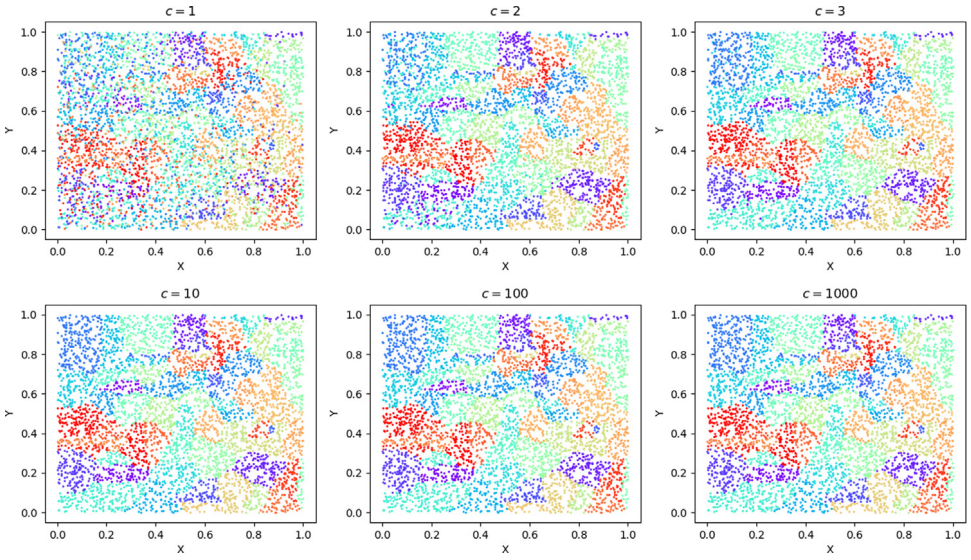

Fig. 1. Example results of the group partition algorithm.

Algorithm 2: Proposed partition strategy

Data: *data*, the set of elements; *centers*, the list of centers; c the constant for max size of groups
Result: *groups*, a list of sets that partition the data

```
groups[i] ← {centers[i], ∀i, 0 ≤ i < |centers|};
maxSize ← c√|data|;
for obj_i ∈ data do
    D ← φ ;
    for center_j ∈ centers do
        D ← D ∪ δ(obj_i, center_j);
    end
    sort(D[i]);
    bestGroup ← the group of the center in D[0];
    repeat
        if not isFull(bestGroup) then
            bestGroup ← bestGroup ∪ {obj_i};
        else
            bestGroup ← the group of the center in next(D);
        end
    until obj_i is added to a group;
end
```

first group to accept an object is filled until its maximum capacity before allowing other group to enlarge; therefore, it causes that the groups become less compact, which is a consequence of the curse of dimensionality (variance of the distances diminishes with dimension increasing). Algorithm 2 presents the pseudo code for the partition process that uses the first described criteria. We improve the efficacy of the algorithm by computing exactly the kNN of each center, since the distances from all objects to all centers are known after this step.

4.1. Complexity analysis

In this section, we discuss the complexity of the algorithm in terms of the number of distances it computes. We propose that given a reasonable constraint on the number k of nearest neighbors, our algorithm computes $\Theta(n^{3/2})$ distances. We also show that, unlike other algorithms [28], the constant factor of the complexity does not depend exponentially on the dimension of the space.

Theorem 4.1. *The proposed algorithm for approximated kNN self-similarity join computes $\Theta(n^{3/2})$ distances if $k \leq c\sqrt{n}$, where $c \in O(1)$.*

Proof. The random selection of $\sqrt{n}$ centers does not involve distance computations. In the partition phase, we compute the distance between all the centers and all the remaining objects, being in total $(n-\sqrt{n})\cdot\sqrt{n} = n^{3/2} - n = \Theta(n^{3/2})$ distances. In the next phase we compute the distance between each object q and all other objects in its own group and the next group closest to q which are called the target set of q. The combination of those two groups has size of at least $c\sqrt{n}$ and at most $2c\sqrt{n}$ which are by definition larger than k. A sequential scan is performed for each object against its target set. Therefore, computing n sequential scans requires at least $n \cdot c\sqrt{n}$ distances, and $n \cdot 2c\sqrt{n}$ distances at most, which is $\Theta(n^{3/2})$ complexity, and proves the result. □

4.2. Implementation

The algorithm is implemented in Python and is publicly available.[1] The module contains a main function called `self_sim_join` that receives a matrix with the data, in the case of vector objects, or a filename where to read in the case of strings. The method also receives the parameters c and k. The function first calls to a sub-routine for the random selection of the $\sqrt{n}$ centers which are then separated from the rest of the data. Later, the groups are formed using the aforementioned criteria computing the distances between the centers and all other objects. For optimization and precision purposes, it stores relevant similarity relations among the centers and the rest of the data. Nearest neighbor candidates are stored in priority queues, one per each object, so whenever a closer object is found it replaces the current candidate at the furthest distance. In the final stage, for each group and for each element in the group a sequential search is performed: the element is compared against all others in its own group and all elements in the one other group that is closer; in the rare case that both groups combined do not have k objects, further groups are added until there are at least k. As per before, every time a better candidate for nearest neighbor is found in the sequential scan the object is added to the queue, as the further candidate is removed. Finally, the set of priority queues is returned as the result of the join.

[1] https://github.com/scferrada/self-sim-join.

Table 1
Average precision of the algorithm for the different datasets, values of c and values of k.

Dataset	c	$k = 1$	$k = 4$	$k = 8$	$k = 16$
DECAF	1	18.15%	17.22%	16.32%	15.48%
	2	31.75%	30.26%	28.65%	27.25%
	3	37.94%	35.69%	34.26%	32.38%
	10	44.31%	41.41%	39.14%	38.19%
	100	44.80%	41.93%	40.31%	37.79%
	1000	45.16%	41.92%	41.21%	38.49%
HOG	1	20.18%	18.90%	18.06%	17.16%
	2	36.80%	34.32%	32.80%	31.10%
	3	42.18%	39.53%	37.89%	35.29%
	10	45.77%	42.77%	41.29%	38.94%
	100	45.74%	42.68%	40.97%	38.81%
	1000	46.00%	42.84%	41.06%	39.03%
STRINGS	1	15.54%	15.86%	15.08%	13.24%
	2	23.66%	24.55%	23.08%	20.11%
	3	24.68%	26.29%	25.46%	22.56%
	10	25.07%	25.87%	25.06%	22.92%
	100	25.82%	26.71%	25.25%	22.45%
	1000	24.64%	26.92%	24.05%	22.16%

5. Evaluation

In this section we describe the experimental design used to test the algorithm. We present and discuss the results obtained. We finish by comparing the algorithm to others.

5.1. Experimental settings

We tested our algorithm on three real-world datasets:

- **STRINGS**: 46,801 words from the English dictionary, using edit distance, and considering only words with 4 or more letters.
- **HOG**: 928,276 visual feature vectors of 72 dimensions, taken from the IMGpedia dataset [4]. We use Manhattan distance to compare.
- **DECAF**: 39,327 deep-visual features DeCAF7 [29] of 4092 dimensions compared with Manhattan distance.

We run the algorithm 100 times on each dataset, computing 1, 2, 8 and 16 nearest neighbors self-similarity join. We use six values of c: 1, 2, 3, 10, 100 and 1000. All experiments were executed on a machine with Debian 4.1.1, a 2.2 GHz 24-core Intel® Xeon® processor, and 120 GB of RAM.

5.2. Average precision

Table 1 shows the average precision of the algorithm for each dataset and value of k and c. The average precision of the algorithm increases as the value of c increases. The biggest increment occurs between $c = 1$ and $c = 2$ where the precision increases about 14% in the vector spaces and about 8% in strings. When $c = 3$ the average precision increases only around a 5.5% versus $c = 2$ in the vector spaces, and only 1% in strings. For higher values of c, average precision also increases but variance increases as well. This increase in variance explains why in some cases the average precision using $c = 100$ or 1000 is not much better than with $c = 10$. Thus, we propose that an optimal value for c is between 2 and 10.

The average precision also decreases as the number of requested nearest neighbors increases. This is because having more elements to find increases the chances for the algorithm of finding wrong ones. For $c = 3$ we report a difference of almost 7% of precision in the HOG dataset between finding 1NN and 16NN; this difference is of 5.5% in the DECAF dataset, and 2% in the STRINGS dataset. For higher values of c this trend is also present with the precision dropping between 1 and 2 percent when k increases from 8 to 16.

The difference between the precision of the algorithm in the vector spaces can be explained by the dimensionality of the data. As HOG is a 72-dimensional space and DECAF a 4098-dimensional space, the distribution of distances is greatly different between both datasets, presenting a low mean and high variance the former and a high mean and a low variance the latter. That behavior is known as the *curse of dimensionality*. As the dimension grows the quality of the results might become less satisfactory. The issue of the quality of a kNN query result in very high-dimensional spaces is discussed by Hinneburg et al. [30] where they define a quality metric and compare different metric distances, concluding that Manhattan distance usually retrieves better quality results than other L_p distances.

The average precision of the algorithm using the STRINGS dataset, with edit distance is worst than when using the other two datasets. This is mostly because when employing discrete distances it becomes odd to talk about nearest neighbors since more than k elements can be at an exact given distance d. In such a case, which elements should be retrieved? Are those elements chosen by the brute-force algorithm aligned with those chosen for our algorithm? The answers to these questions depend on assumptions that must be made beforehand, redefining the nearest neighbor search. Hence, the results over the vector spaces with continuous distances are better than in string spaces with discrete distances. This kind of behavior also explains that in the STRINGS dataset the precision increases from $k = 1$ to $k = 4$.

5.3. Precision distribution

We present the distribution of the number of correctly found nearest neighbors per object. To obtain this, we used 100 runs of the algorithm using DECAF dataset and compute the number of correct matches per object in the search for the 16 nearest neighbors. Fig. 2 shows histograms that display the percentage of elements that have i correctly found neighbors, being i a bin of the histogram. A histogram of the ground truth would show a single bar on the sixteenth bin with length 1. We see that the distributions are single-peaked and skewed right. The figure also presents an outlier peak in the last bin (the elements for which the algorithm finds all correct 16NN). When $c = 1$ the results tend to be poor, being most of the elements matched with none of its actual 16 nearest neighbors. When c increases, the results improve remarkably: the elements with 0 matches decrease considerably and the distribution becomes more uniform. With higher values of c we see further improvement, however the marginal increase in precision is not substantial when increasing the group size given that the distributions for $c = 10$,100 and 1000 are similar. In general, around a 7% of the objects are always matched with none of their actual 16NN.

5.4. Execution time and distance calculations

Fig. 3 presents the execution time for the algorithm in the different experimental settings. Times are provided as an average and a variance, since multiple runs of the experiments are considered. It can be seen that the variance of the execution time increases greatly with c due to the potential formation of a few massive groups and many small groups, where the search in big groups dominates the time. Since computing distances is more expensive as the dimension of the space increases, DECAF average times are higher than HOG times for higher values of c. When considering low values of c time depends mostly on the size of the dataset and the dimensionality of the objects; however, as

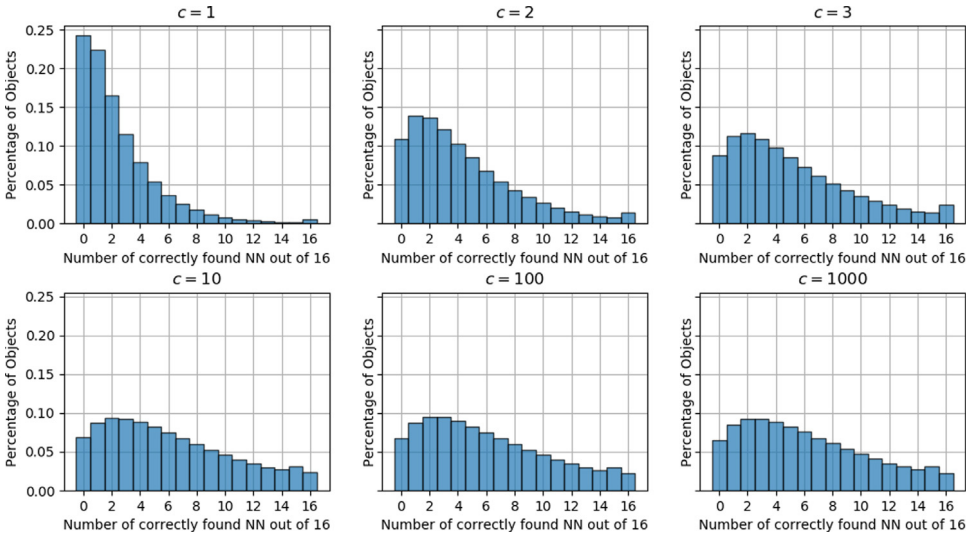

Fig. 2. Distribution of the number of correctly found 16-NN using DECAF dataset.

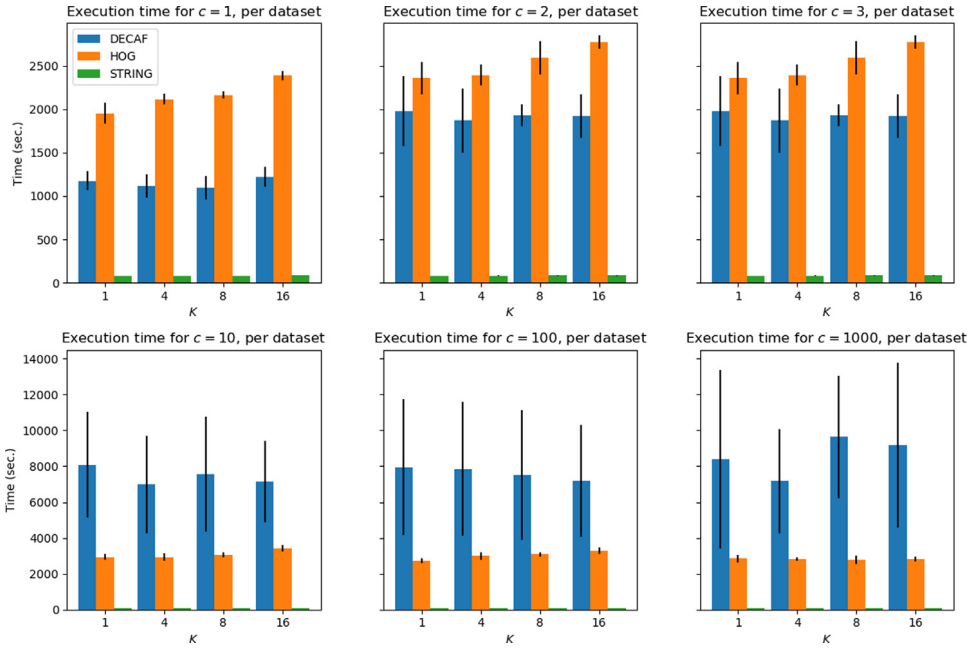

Fig. 3. Execution time for each dataset, c and k.

c increases the dependency of execution time on the number of objects becomes less clear. The number of required nearest neighbors does not seem to correlate with the final execution time when k is much lesser than $\sqrt{n}$.

As per the number of computed distances, Fig. 4 shows the values in logarithmic scale for each dataset and value of c. In all datasets, the average number of distances increases with c, as well as the variance. The last bars of the graph present the distances computed by a brute force approach. In DECAF dataset, the increase is more remarkable due to the high dimension of the space. Average number of distances in DECAF space is even greater than the one of STRINGS, despite of being a smaller set.

When comparing the number of computed distances with the execution time, we can conclude that distance computations are more time consuming when the dimension of the space gets higher, thus dimension has a greater impact on computation time even than the size of the dataset: the algorithms takes much more time in DECAF than in HOG, when DECAF is much smaller than HOG. Finally, when comparing with a brute force approach we can see that our algorithm takes up to 3 less orders of magnitude.

5.5. Center selection strategies

Results in previous sections consider a random selection of centers. Here we present results using a more sophisticated technique. In Section 4 two approaches where discussed: the automatic pivot selection of Brisaboa et al. [26] and the sampling technique of Yianilos [27]. The former technique cannot be applied to this algorithm since it does not contemplate a way to define the number of pivots to be returned. The latter approach requires to set a sample size, among the objects of the sample it selects those that have the greatest spread w.r.t. all other objects.

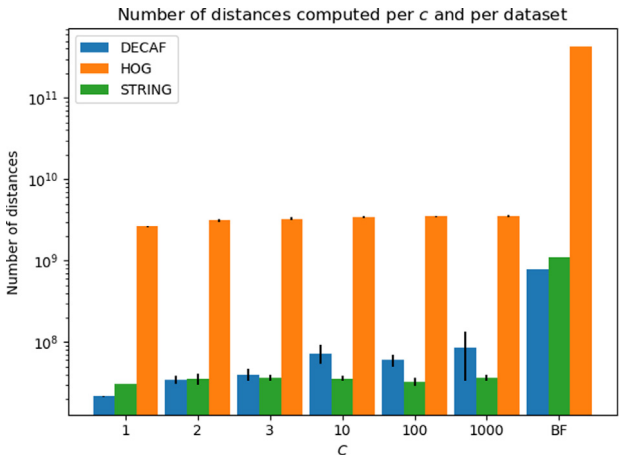

Fig. 4. Computed distances per c value and per dataset.

Algorithm 3 presents the pseudo-code of the strategy. It gets $c\sqrt{n}$ random objects to be candidate centers (original work proposes constant size samples). Each of the candidates is compared with another random sample of the same size as previous sample, the median and standard deviation of the distances is computed. Using a max-heap, we keep the objects that have the maximum standard deviation and return them as centers. Fig. 5 presents how 5 thousand random points distribute into 71 groups with this strategy for the different values of c, it can be seen that well-formed groups appear with $c = 10$ and are mostly unchanged with higher values.

Algorithm 3: Center Selection Strategy based on Yianilos [27]

Data: *data*, the set of elements; c the constant for max size of groups
Result: *centers*, a list of the objects chosen as centers
1 $centers \leftarrow \phi$;
2 $candidates \leftarrow Heap(\{(\perp, 0)\ \forall i \leq c\sqrt{|data|}\})$;
3 $N \leftarrow c\sqrt{|data|}$;
4 $sample \leftarrow choose\_random\_noreplace(data, N)$;
5 **for** $obj_i \in sample$ **do**
6 $D \leftarrow choose\_random\_noreplace(data, N)$;
7 $dist \leftarrow distance(obj_i, D)$;
8 $\mu \leftarrow median(dist)$;
9 $\sigma \leftarrow stdv(dist - \mu)$;
10 **if** $\sigma > candidates.peek_1$ **then**
11 $candidates.pop - push((obj_i, \sigma))$;
12 **end**
13 **end**
14 $centers \leftarrow candidates$;
15 **return** *centers*;

We tested our algorithm selecting $\sqrt{n}$ objects with Yianilos technique. We run the algorithm several times using the HOG dataset, $k = 4$ and $c = 1, 2, 3, 10, 100, 1000$. Results can be found in Table 2, where it can be seen that the achieved average precision of the algorithm is consistently worst using vp-tree-like center selection than with a random selection, even with high values of c where, as it can be seen in Fig. 5, groups are well-delimited the precision drops to ∼12%. Further experimentation can be done: using increasingly larger samples of the data could yield better results. Nonetheless, we explain the poor performance of Yianilos technique by the reasoning that is given in their work: vantage points are meant to be close to the *corner* of the space so those points are more useful to divide the space evenly, which is a different task than to group objects closer together.

Table 2
Average precision of the algorithm using vp-tree-like center selection.

Dataset	k	$c = 1$	$c = 2$	$c = 3$	$c = 10$	$c = 100$	$c = 1000$
HOG	4	18.98%	31.29%	31.25%	11.94%	12.04%	12.08%

5.6. Comparison with other similarity join algorithms

We found that, for range-based similarity joins, Quickjoin is widely used [31,32]. Quickjoin was extended to support k-distance joins [8], i.e., to obtain the k minimum distances between the elements of a dataset. We propose another extension that builds upon the k-distance join and computes a kNN join. As it is defined in [8], we run Quickjoin twice. The first time, we perform a range-based similarity join with $\varepsilon = 0$ where we collect the k nearest neighbors found in that setting, which is an approximation of the real answer. The second run uses the maximum distance found for a candidate nearest neighbor as the value for ε, the value can be decreased over time, whenever the maximum candidate distance decreases. Using such an algorithm carries two problems regarding efficiency and efficacy. The first run of Quickjoin does not necessarily computes at least k distances per element, which can affect the precision by not getting a more fit boundary. Since the maximum distance found in the first run is usually near the average distance of the space (especially in very high-dimensional spaces due to the curse of dimensionality), the window partitions of Quickjoin contain a high fraction of the objects, which implies that the time it would take will be closer to quadratic.

We use this extension to Quickjoin to compute kNN self-similarity joins. Quickjoin requires a parameter c that represents the minimum number of objects in a partition so it can be processed with a nested loop. We run the algorithm 100 times using the DECAF dataset, $k = 4$ and c as 0.1% of the data. We use only this scenario due to time and memory constraints: The execution of this scenario takes 7 days using multiple threads, and with higher values of k the algorithm finishes for lack of memory (this happens mostly because of the window partitions replicating the data). This modified version of Quickjoin reaches a 29.49% average precision, compared with 35.69% of our algorithm in the same scenario. Our algorithm is also faster, it takes 47.5 min in average to complete one run over the DECAF dataset, whereas modified Quickjoin takes around 19 h (again, this is due the size of the window partitions, which increases greatly the number of required recursions).

LSH requires several parameters: The number of hashtables to be used, the number of hash functions of each table, and the parameters for the hash functions. We followed the suggestion of Datar et al. [11], and use 30 tables with 10 hash functions, and for each function we draw a standard Cauchy distribution parameter and a uniformly distributed parameter. Using this setting with the DECAF dataset gives a poor performance of the algorithm in terms of average precision, reaching only a 8.15% for $k = 1$ and 5.62% for $k = 16$. We argue that this poor precision has to do with the parameter selection, which requires to be fine-tuned, but it is mostly due to the high dimension of the space, which implies that the *locality* is not well captured by the proposed hash functions.

6. Conclusions

We presented an heuristic to solve the approximated kNN self-similarity join in metric spaces. We proved it requires $\Theta(n^{3/2})$ distance computations. We provided an open implementation and tested it using three real world datasets, where we found it can reach up to 46% precision. We present performance metrics, namely the execution time and the number of computed

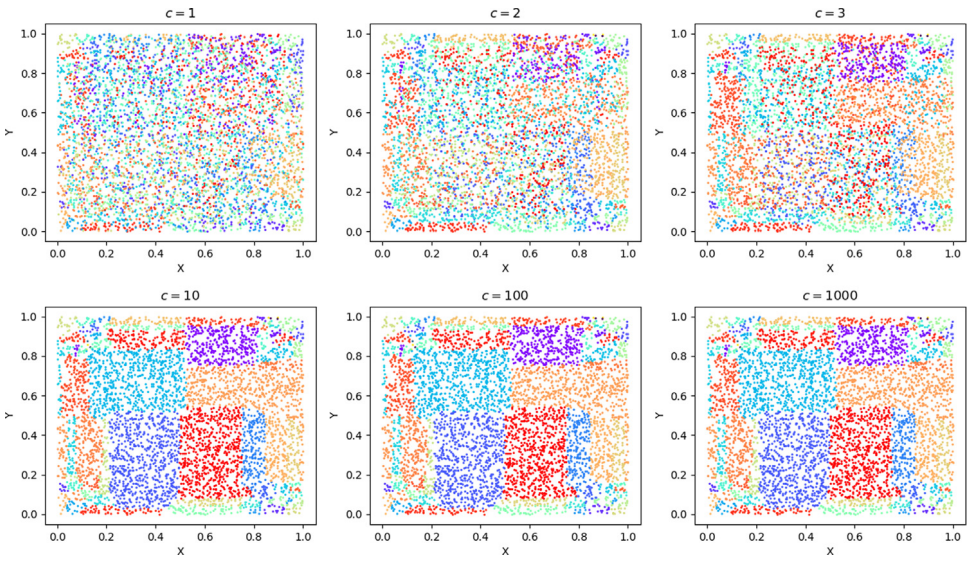

Fig. 5. Partition of random 2D points using center selection of Yianilos [27].

distances, and we discuss how the size and dimension of the datasets affects the performance of the algorithm. We discuss how the algorithm can be compared to a modified version of Quickjoin, that supports *k*NN self-similarity joins, and to LSH and we provide scenarios where our algorithm outperforms them in terms of execution time and average precision. The algorithm is also tested using a different center selection technique, which was found to worsen its efficacy.

The algorithm here described can prove useful for cases where building and storing an index is purposeless and/or there are time restrictions to get a fast answer. An example of such a case is multimedia similarity search on the Web, where queries tend to be unique.

This work can be further improved by using a center selection algorithm that allows the formation of less overlapping groups. It can also be included a multiple association of objects to groups, allowing objects to belong to more than one group under a set of conditions that respect the complexity of the algorithm. The definition of a better *k*NN self-similarity join Quickjoin-based algorithm can be helpful to have a more fair comparison to our algorithm or even to further improve the state-of-the-art in terms of precision and execution time.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partly funded by the Millennium Institute for Foundational Research on Data, Chile and CONICYT-PFCHA, Argentina grant 2017-21170616.

References

[1] G. Giacinto, A nearest-neighbor approach to relevance feedback in content based image retrieval, in: Proc. 6th ACM International Conference on Image and Video Retrieval, ACM, 2007, pp. 456–463.

[2] H. Li, X. Zhang, S. Wang, Reduce pruning cost to accelerate multimedia kNN search over MBRs based index structures, in: 3rd International Conference on Multimedia Information Networking and Security, 2011, pp. 55–59.

[3] R. Guerraoui, A. Kermarrec, O. Ruas, F. Taïani, Fingerprinting big data: The case of KNN graph construction, in: Proc. 35th International Conference on Data Engineering, IEEE, 2019, pp. 1738–1741.

[4] S. Ferrada, B. Bustos, A. Hogan, IMGpedia: A linked dataset with content-based analysis of wikimedia images, in: Proc. 16th International Semantic Web Conference, Springer, 2017, pp. 84–93.

[5] C. Böhm, F. Krebs, Supporting KDD applications by the k-nearest neighbor join, in: Proc. International Conference on Database and Expert Systems Applications, Springer, 2003, pp. 504–516.

[6] R. Apweiler, A. Bairoch, C.H. Wu, W.C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M.J. Martin, D.A. Natale, C. O'Donovan, N. Redaschi, L.L. Yeh, UniProt: the Universal Protein knowledgebase, Nucleic Acids Res. 32 (2004) D115–D119.

[7] E.H. Jacox, H. Samet, Metric space similarity joins, ACM Trans. Database Syst. 33 (2) (2008) 7.

[8] K. Fredriksson, B. Braithwaite, Quicker similarity joins in metric spaces, in: Proc. International Conference on Similarity Search and Applications, Springer, 2013, pp. 127–140.

[9] S. Ferrada, B. Bustos, N. Reyes, A simple, efficient, parallelizable algorithm for approximated nearest neighbors, in: Proc. 12th Alberto Mendelzon International Workshop on Foundations of Data Management, 2018, URL http://ceur-ws.org/Vol-2100/paper3.pdf.

[10] A. Gionis, P. Indyk, R. Motwani, Similarity search in high dimensions via hashing, in: Proc. 25th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., 1999, pp. 518–529.

[11] M. Datar, N. Immorlica, P. Indyk, V.S. Mirrokni, Locality-sensitive hashing scheme based on p-stable distributions, in: Proc. 20th Annual Symposium on Computational Geometry, ACM, 2004, pp. 253–262.

[12] C. Böhm, F. Krebs, The *k*-nearest neighbour join: Turbo charging the KDD process, Knowl. Inf. Syst. 6 (6) (2004) 728–749.

[13] C. Yu, R. Zhang, Y. Huang, H. Xiong, High-dimensional kNN joins with incremental updates, GeoInformatica 14 (1) (2009) 55.

[14] B. Yao, F. Li, P. Kumar, K nearest neighbor queries and kNN-Joins in large relational databases (almost) for free, in: Proc. 26th International Conference on Data Engineering, IEEE, 2010, pp. 4–15.

[15] W. Lu, Y. Shen, S. Chen, B.C. Ooi, Efficient processing of K nearest neighbor joins using MapReduce, Proc. VLDB Endow. 5 (10) (2012) 1016–1027.

[16] Y.N. Silva, J.M. Reed, Exploiting MapReduce-based similarity joins, in: Proc. ACM SIGMOD International Conference on Management of Data, ACM, 2012, pp. 693–696.

[17] Y. Wang, A. Metwally, S. Parthasarathy, Scalable all-pairs similarity search in metric spaces, in: Proc. 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2013, pp. 829–837.

[18] G. Song, J. Rochas, L.E. Beze, F. Huet, F. Magoules, K nearest neighbour joins for big data on MapReduce: A theoretical and experimental analysis, IEEE Trans. Knowl. Data Eng. 28 (9) (2016) 2376–2392.

[19] C. Rong, C. Lin, Y.N. Silva, J. Wang, W. Lu, X. Du, Fast and scalable distributed set similarity joins for big data analytics, in: 33rd IEEE International Conference on Data Engineering, 2017, pp. 1059–1070.

[20] G. Chen, K. Yang, L. Chen, Y. Gao, B. Zheng, C. Chen, Metric similarity joins using MapReduce, IEEE Trans. Knowl. Data Eng. 29 (3) (2017) 656–669.

[21] P. Cech, J. Marousek, J. Lokoc, Y.N. Silva, J. Starks, Comparing MapReduce-based k-NN similarity joins on Hadoop for high-dimensional data, in: Proc. 13th International Conference on Advanced Data Mining and Applications, 2017, pp. 63–75.
[22] P. Moutafis, G. Mavrommatis, M. Vassilakopoulos, S. Sioutas, Efficient processing of all-k-nearest-neighbor queries in the MapReduce programming framework, Data Knowl. Eng. 121 (2019) 42–70.
[23] S.S. Pearson, Y.N. Silva, Index-based R-S similarity joins, in: Similarity Search and Applications, Springer, 2014, pp. 106–112.
[24] V. Dohnal, C. Gennaro, P. Zezula, Similarity join in metric spaces using ed-index, in: Proc. International Conference on Database and Expert Systems Applications, Springer, 2003, pp. 484–493.
[25] L. Chen, Y. Gao, A. Zhong, C.S. Jensen, G. Chen, B. Zheng, Indexing metric uncertain data for range queries and range joins, VLDB J. 26 (4) (2017) 585–610.
[26] N.R. Brisaboa, A. Farina, O. Pedreira, N. Reyes, Similarity search using sparse pivots for efficient multimedia information retrieval, in: Proc. 8th IEEE International Symposium on Multimedia, IEEE, 2006, pp. 881–888.
[27] P.N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in: Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 1993, pp. 311–321.
[28] E. Chávez, G. Navarro, R. Baeza-Yates, J.L. Marroquín, Searching in metric spaces, ACM Comput. Surv. 33 (3) (2001) 273–321.
[29] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, T. Darrell, DeCAF: A deep convolutional activation feature for generic visual recognition, in: Proc. 31st International Conference on Machine Learning, JMLR.org, 2014, pp. 647–655.
[30] A. Hinneburg, C.C. Aggarwal, D.A. Keim, What is the nearest neighbor in high dimensional spaces? in: Proc. 26th International Conference on Very Large Databases, Morgan Kaufmann Publishers Inc., 2000, pp. 506–515.
[31] Y.N. Silva, S.S. Pearson, J.A. Cheney, Database similarity join for metric spaces, in: Proc. International Conference on Similarity Search and Applications, Springer, 2013, pp. 266–279.
[32] Y.N. Silva, J.M. Reed, L.M. Tsosie, MapReduce-based similarity join for metric spaces, in: Proc. 1st International Workshop on Cloud Intelligence, ACM, 2012, pp. 3:1–3:8.