Effective Proximity Retrieval by Ordering Permutations

EDGAR CHAVEZ, *Universidad Michoacana, Mexico*
KARINA FIGUEROA, *Universidad Michoacana, Mexico, and Universidad de Chile, Chile*
GONZALO NAVARRO, *Universidad de Chile, Chile*

Supported by CONACyT (first and second author) and Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile (second and third author). A preliminary partial version of this paper appeared in *Proc. 4th Mexican International Conference on Artificial Intelligence (MICAI 2005)*, pp. 405–414, LNAI Series vol. 3789.

Abstract

We introduce a new probabilistic proximity search algorithm for range and K-nearest neighbor (K-NN) searching in both coordinate and metric spaces. Although there exist solutions for these problems, they boil down to a linear scan when the space is intrinsically high-dimensional, as is the case in many pattern recognition tasks. This, for example, renders the K-NN approach to classification rather slow in large databases.

Our novel idea is to predict closeness between elements according to how they order their distances towards a distinguished set of anchor objects. Each element in the space sorts the anchor objects from closest to farthest to it, and the similarity between orders turns out to be an excellent predictor of the closeness between the corresponding elements.

We present extensive experiments comparing our method against state-of-the-art exact and approximate techniques, both in synthetic and real, metric and non-metric databases, measuring both CPU time and distance computations. The experiments demonstrate that our technique almost always improves upon the performance of alternative techniques, in some cases by a wide margin.

I. Introduction

The classical Pattern Recognition process has three main stages: segmentation, feature extraction, and classification [29]. Segmentation consists of extracting the individual objects from the digitalized data. Feature extraction consists in mapping the digital objects onto a (usually high-dimensional) vector space, where each coordinate represents the degree of presence of a certain feature in the object. Classification consists of assigning each object to one out of a set of predefined classes of objects. This model encompasses concrete pattern recognition tasks such as speech recognition, speaker identification, signature matching, handwriting recognition, face recognition, biometric identification, and so on [22].

Feature extraction converts the original classification problem into a geometric problem. Objects in the same class tend to be spatially close if the features are selected properly. The most popular classification techniques, such as support vector machines, neural networks, or K nearest neighbors, are defined in terms of geometry. Among those, K nearest neighbors (K-NN) classifiers are attractive because the training is implicit.

The K-NN approach translates the problem of classification into a *proximity search* problem (find the K representative objects closest to a new given element) in a high-dimensional feature space. Unfortunately, current methods for proximity searching suffer from the so-called *curse of dimensionality* [16]: Any method for proximity searching, no matter how well it works in low dimensionalities, ends up scanning the whole set of objects in high dimensionalities. Dimensionality reduction techniques are effective and well-known, but they pose an extra overhead on the system when the data is *intrinsically* high-dimensional, and the classification accuracy will drop if the distances in the lower dimensional space are not well preserved. That is, the data will be miss-classified when using a K-NN approach in a mapped space distorting the original distances.

To avoid mapping onto a lower dimensional space, an abstract metric could be defined among objects (e.g., the edit distance or dynamic time warping to match sequences) and can be transparently used as a black box in a K-NN classifier. In some cases this is preferred over either mapping onto a vector space (to classify with a neural network) or defining a suitable kernel function (to classify with a kernel-based support vector machine).

In the so-called *metric spaces*, intrinsic dimensionality can be defined in many ways, for example as the minimum dimensionality of a vector space onto which the metric space objects can be mapped without distorting much their pairwise distances. High-dimensional metric spaces have a concentrated histogram of distances, and just as on high-dimensional vector spaces, no proximity search algorithm can avoid comparing the query against all of the database.

Apart from classification, there are many other application areas for proximity searching: searching for similar objects in multimedia databases, searching for similar documents in information retrieval, searching for similar biological sequences in computational biology, data prediction, correction, or compression in signal processing, and so on. In all cases, the general model is that of a black-box database of objects that can be preprocessed so as to answer proximity queries against new objects that are given later. The only tool to obtain information from the objects is the computation of their distance towards other objects. The curse of dimensionality shows up in all these applications as well, in many cases rendering index-based methods as bad as a linear scan over the database or even worse.

Such a linear scan does not scale well when the set of objects to search is large or the distance function is computationally expensive. Different relaxations on the precision of the result have been proposed in order to obtain a computationally feasible solution in those cases. This is called *inexact proximity searching*, as opposed to the classical *exact proximity searching*. Inexact proximity searching is reasonable in many applications because the feature-extraction or the metric-space modelizations already involve an approximation to reality, and therefore a second approximation at search time is usually acceptable.

In the literature we find basically two alternatives for inexact proximity searching. A first one uses a distance relaxation parameter: It is ensured that the distance to the nearest neighbor answer they find is at most $1+\varepsilon$ times the distance to the true nearest neighbor. This corresponds to approximation algorithms in the usual algorithmic sense, and is considered in depth in [41], [16], [18]. A second alternative takes a probabilistic approach, ensuring that the answer of the algorithm is correct with high probability. This corresponds to probabilistic algorithms in the usual algorithmic sense. A generic method to convert exact into probabilistic algorithms is studied in [14], [10].

In this paper we present a new probabilistic proximity search algorithm for metric spaces (which include vector spaces as a particular case). The central idea is to predict the closeness between any two objects in a metric space by comparing the way these two objects order their distances towards a set of anchor objects called *permutants*. The index does not store any actual distance, but just permutations of the anchor objects as perceived by each database element.

We show that the similarity among permutations is a remarkably good predictor of the proximity among the corresponding objects. Thus, the database can be traversed from the permutation most to the least similar to the permutation of the query object, and we expect to find early most of the relevant answers.

The probabilistic algorithm that results from traversing a given percentage of the database and returning the closest elements seen up to then, is extremely efficient and outperforms any existing alternative we are aware of. This is remarkable because there already exist very successful probabilistic techniques. We also tested our technique over non-metric databases, using quasi-distances where the triangle inequality does not hold, and found that the retrieval effectiveness is comparable to that on metric databases.

II. Basic Concepts and Related Work

A. Basic Terminology

Formally, the proximity searching problem may be stated as follows: There is a universe $\mathbb{X}$ of *objects*, and a nonnegative *distance function* $d : \mathbb{X} \times \mathbb{X} \longrightarrow \mathbb{R}^+$ defined among them. The distance satisfies the axioms that make the set a *metric space*: reflexivity ($d(x,x) = 0$), strict positiveness ($x \neq y \Rightarrow d(x,y) > 0$), symmetry ($d(x,y) = d(y,x)$) and triangle inequality ($d(x,z) \leq d(x,y) + d(y,z)$). This distance is assumed to be expensive to compute (think, for instance, in comparing two fingerprints). We have a finite *database* $\mathbb{U} \subseteq \mathbb{X}$, of size n, which is a subset of the universe of objects. The goal is to preprocess the database $\mathbb{U}$ to efficiently answer (i.e., with as few distance computations as possible) *range queries* and *K-nearest neighbor (K-NN)* queries. Range queries are expressed as (q,r) (a point in $\mathbb{X}$ and a tolerance radius), which should retrieve all the database points at distance r or less from q, i.e., $\{u \in \mathbb{U},\ d(u,q) \leq r\}$. On the other hand, K-nearest neighbor queries retrieve the K elements of $\mathbb{U}$ that are closest to q.

Most of the existing approaches to solve the search problem are *exact algorithms* which retrieve exactly the elements of $\mathbb{U}$ as specified above. In [16], [27], [38], [44] most of those approaches are surveyed and explained in detail. It is usually easier to design range search algorithms, and then apply standard techniques to derive K-NN search algorithms from those.

B. Inexact Proximity Searching

In this work we are interested in inexact algorithms, which relax the condition of delivering the exact solution. This relaxation uses, in addition to the query, a *precision* parameter ε to control how far away (in some sense) can the outcome of the query be from the correct result.

Approximation algorithms are surveyed in depth in [41]. An example is [4], which proposes a data structure for vector spaces under Minkowski metrics L_s. The structure, called the BBD-tree, is inspired in kd-trees and can be used to find "$(1+\varepsilon)$ nearest neighbors": instead of finding u such that $d(u,q) \leq d(v,q)\ \forall v \in \mathbb{U}$, they find u^* such that $d(u^*,q) \leq (1+\varepsilon)d(v,q)\ \forall v \in \mathbb{U}$.

The essential idea behind this algorithm is to locate the query q in a cell (each leaf in the tree is associated with a cell in the decomposition). Every point inside that cell is processed so as to obtain the nearest neighbor u of q within the cell. The search continues with neighboring cells and stops when the radius of a ball centered at q and intersecting any cell not yet considered exceeds $d(q,u)/(1+\varepsilon)$. The query time is $O(\lceil 1+6D/\varepsilon\rceil^D D\log n)$, where D is the dimensionality of the space.

Probabilistic algorithms have been proposed both for vector spaces [4], [43], [41], [23] and for general metric spaces [20], [18], [14], [10]. We survey a few of them.

In [43], the data structure is a standard kd-tree. The author uses “aggressive pruning” to improve the performance. The idea is to increase the number of branches pruned at the expense of losing some candidate points in the process. This is done in a controlled way, so the probability of success is always known. The data structure is useful for finding limited-radius nearest neighbors, that is, nearest neighbors within a fixed distance to the query.

In [23] the distance between two vectors is approximated by a convex combination of a *shape* measure of the vectors and their magnitudes. The shape measure has some resemblances with our technique, as they sort the coordinates of vectors by increasing value. Yet, our method applies to the more general metric spaces, and does not use any equivalent to the magnitudes.

In [20], the author chooses a “training set” of queries and builds a data structure able to answer correctly only queries belonging to the training set. The idea is that this setup is enough to answer correctly, with high probability, an arbitrary query. Under some probabilistic assumptions on the distribution of the queries, it is shown that the probability of not finding the nearest neighbor is $O((\log n)^2/k)$, where k can be made arbitrarily large at the expense of $O(kn\alpha)$ space and $O(k\alpha\log n)$ expected search time. Here α is the logarithm of the ratio between the farthest and the nearest pairs of points in the union of $\mathbb{U}$ and the training set.

In [10], the authors use a technique to obtain probabilistic algorithms that is relevant to this work. They use different techniques to *sort the database* according to some *promise value*. As they traverse the database in such order, they obtain more and more relevant answers to the query. In other words, given a limited amount of work allowed, the algorithm finds each correct answer with some probability, and it can improve the answer incrementally if more work is allowed. A good database ordering is one that obtains most of the relevant answers by traversing a small fraction of the database. Thus, the problem of finding a good probabilistic search algorithm translates into finding a good ordering of the database given a query q. Our contribution in this paper falls within this general approach.

Finally, there are approaches that combine approximation and probabilistic techniques, such as the PAC (probably approximately correct) method [17]. This is also the case of [14], which present a general method based on stretching the triangle inequality.

C. Indexing

All metric space search algorithms rely on an *index*, that is, a data structure that maintains some information on the database in order to save some distance evaluations at search time. There exist two main types of data organizations [16], which we cover next.

1) Pivoting Schemes: A *pivot* is a distinguished database element, whose distance to some other elements is precomputed and stored in an index. Imagine that we have precomputed $d(p,u)$ for some pivot p and every $u \in \mathbb{U}$. At search time, for a range query with radius r, we compute $d(p,q)$. Then, by the triangle inequality, $d(q,u) \geq |d(p,q) - d(p,u)|$, so if $|d(p,q) - d(p,u)| > r$ we know that $d(q,u) > r$, thus u can be filtered out without need of computing distance $d(q,u)$.

The most basic pivoting scheme chooses k pivots $p_1 \ldots p_k$ and computes all the distances $d(p_i,u)$, $u \in \mathbb{U}$, into a table of kn entries. Then, at query time, all the k distances $d(p_i,q)$ are computed and every element u such that $D(q,u) = \max_{i=1 \ldots k} |d(p_i,q) - d(p_i,u)| > r$ is discarded. Finally, q is compared against the elements not discarded.

As k grows, we have to pay more comparisons against pivots, but $D(q,u)$ becomes closer to $d(q,u)$ and more elements may be discarded. It can be shown that there is an optimum number of pivots k^*, which grows fast with the dimensionality and becomes quickly unreachable because of memory limitations. In all but the easiest metric spaces, one simply uses as many pivots as memory permits. There exist many variations over the basic idea, including different ways to store the table of kn entries to reduce extra CPU time, e.g. [13], [11], [32], [5], [12].

Several tree data structures are built on the same pivoting concept, e.g. [42], [9], [30]. In most of them, a pivot p is chosen as the root of a tree, and its subtrees correspond to ranges of $d(p,u)$ values, being recursively structured. In some cases the exact distances $d(p,u)$ are not stored, just the range can be inferred from the subtree the element u is in. Albeit this reduces the accuracy of the index, the tree usually takes $O(n)$ space instead of the $O(kn)$ needed with k pivots.

Moreover, every internal node is a partial pivot (which knows distances to its subtree elements only), so we actually have many more pivots (albeit local and with coarse data). Finally, the trees can be traversed using sublinear extra CPU time.

Different tree variants arise according to the tree arities, the way the ranges of distances are chosen (trying to balance the tree or not), how local are the pivots (different nodes can share pivots, which do not belong anymore to the subtree), the number of pivots per node, and so on. Very little is known about which is best. For example, the golden rule of preferring balanced trees, which works well for exact searching, becomes a poorer choice against unbalancing as the dimensionality increases. For very high dimensional data a good structure is almost a linked list (i.e., a degenerate tree) [15]. Also, little is known about how to choose the pivots.

2) Local Partitioning Schemes: Another scheme builds on the idea of dividing the database into spatially compact groups, meaning that the elements in each group are close to each other. A representative is chosen from each group, so that comparing q against the representative has good chances of discarding the whole group without further comparisons. Usually these schemes are hierarchical, so that groups are recursively divided into subgroups.

Two main ways exist to define the groups. One can define "centers" with a covering radius, so that all elements in its group are within the covering radius distance to the center, e.g. [19]. If a group has center c and covering radius r_c then, if $d(q, c) > r + r_c$, the whole group can be discarded. The geometric shape of the above scheme corresponds to a ball centered around c.

In the second approach, e.g. [8], [33], a set of centers is chosen and every other point is added to the group of its closest center. At query time, if q is closest to center c_i, and $d(q, c_j) - r > d(q, c_i) + r$, then we can discard the whole group of c_j. The geometric shape in this approach corresponds to a Dirichlet domain of the space (a generalization of the Voronoi diagram for metric spaces), without overlaps between groups.

III. An Effective Index based on Ordering Permutations

Since the objects in the metric space are seen as black boxes from which we can only compute their distances toward other objects, all indexes in the literature are bound to store distance information. Actually, the most information an index can store is the $n \times n$ matrix of all the distances among objects in $\mathbb{U}$. This is actually what algorithm AESA [40], a pivot-based scheme,

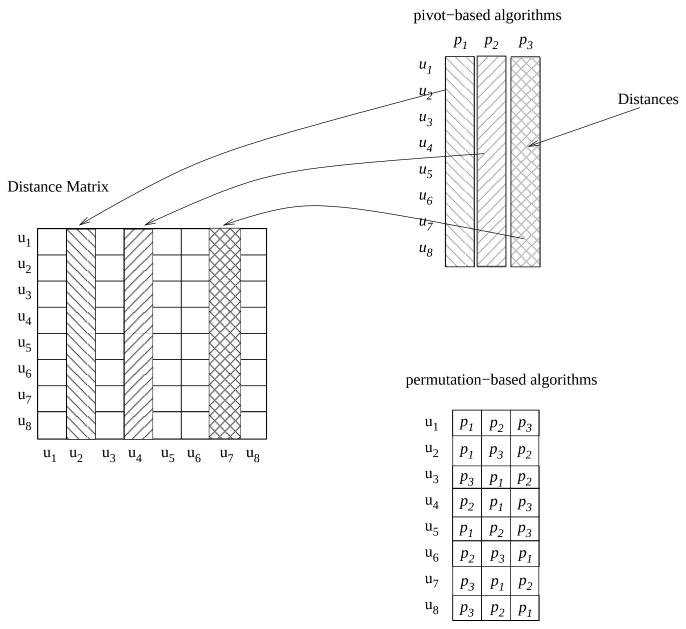


Fig. 1. On the left, the matrix of all distances in $\mathbb{U}$. On the right, on top, a pivot-based algorithm chooses some columns of the distance matrix. On the bottom right, our algorithm only records the order of the pivots, from closest to farthest to the element. Actually, only the permutation is stored, so for example the second row is stored as 1,3,2.

stores as its index. This makes AESA an unbeatable exact algorithm, yet usually impractical because of its high storage consumption.

The design of metric space indexes can be regarded as a quest to store the most useful data from the distance matrix within bounded space. Pivot-based indexes store k columns from the full distance matrix, that is, for each element they store its distances to k fixed pivots. Clustering algorithms store only some of the smallest distances in the matrix, that is, for each cluster center they store the distances to the elements in that cluster. Some algorithms do not store the actual distances but just a range containing them, so as to store more distances with less precision.

Within this framework, our approach can be stated as follows: *We choose k columns from the distance matrix and store, for each row, the order in which the columns are read to obtain the distances in increasing order.* Compared to a classical pivot-based scheme, we do not store the exact distances, but just the order in which each database element sees the pivots, from closest to farthest to the element. That is, to each element we associate a *permutation* of the k pivots. Figure 1 illustrates.

Just as two close elements will have similar distances to pivots, close elements will see the pivots in similar order of closeness, and thus will have similar permutations. A difference in the order between two permutations will hint that the corresponding elements are not too close to each other. However, those differences do not permit us to prove how far away from the query is a database element, thus we will obtain a probabilistic algorithm.

A. Overview of Our Method

We need a bit of terminology. Let $\mathbb{P} \subseteq \mathbb{U}$ be a set of distinguished objects from the database, called *permutants*. Each element of the space, $x \in \mathbb{X}$, defines a *permutation* Π_x, where the elements of $\mathbb{P}$ are written in increasing order of distance to x. Ties are broken using any consistent order, for example the order of the elements in $\mathbb{P}$.

Definition 1: Let $\mathbb{P} = \{p_1,\ p_2, \ldots, p_k\}$ and $x \in \mathbb{X}$. Then we define Π_x as a permutation of $(1 \ldots k)$ so that, for all $1 \leq i < k$ it holds either $d(p_{\Pi_x(i)}, x) < d(p_{\Pi_x(i+1)}, x)$, or $d(p_{\Pi_x(i)}, x) = d(p_{\Pi_x(i+1)}, x)$ and $\Pi_x(i) < \Pi_x(i+1)$.

We are now ready to describe the indexing process, the index structure, and the search process.

1) Indexing: Our index will be just the permutations Π_u for every $u \in \mathbb{U}$, with respect to a set of permutants $\mathbb{P} = \{p_1, \ldots, p_k\} \subseteq \mathbb{U}$.

The construction of the index is carried out as follows:

1) We choose a parameter k, which is the number of permutants to use. The larger k, the more effective the index, but it will need more space ($kn\lceil\log_2 k\rceil$ bits) and also sorting the database to traverse it in the desired order will be slower.
2) We choose $\mathbb{P} = \{p_1, \ldots, p_k\}$, a set of k permutants, at random from $\mathbb{U}$. We will show in Section IV that other selection heuristics of linear-time complexity make no difference in the effectiveness of the indexing algorithm.
3) For each $u \in \mathbb{U}$, we compute $d(u, p_i)$ for all $p_i \in \mathbb{P}$, and store permutation Π_u according to Definition 1.

The result is a table of n rows (one per database element) and k columns (one per permutant). Each cell needs $\lceil\log_2 k\rceil$ bits to store one permutation at each row. The indexing cost is kn distance computations plus $O(nk \log k)$ CPU time to sort all the permutations.

2) Searching: At query time we compute Π_q and traverse $\mathbb{U}$ in the order induced by Π_q. In this order an element u will be smaller than an element v if Π_u is more similar to Π_q than Π_v. As we expect that elements with permutations more similar to Π_q will also be spatially closer to q, we will review them earlier.

The search is carried out as follows:

1) We compute $d(q, p_i)$ for all $p_i \in \mathbb{P}$, and compute permutation Π_q according to Definition 1.
2) Given a similarity measure S between permutations, we sort $\mathbb{U}$ according to $S(\Pi_u, \Pi_q)$ (those $u \in \mathbb{U}$ with smaller $S()$ value go first). Given that we will need just a (small) subset of the first elements after this sorting, we have used an incremental sorting method [36], which gives the elements in order as we need them. Other methods such as a full QuickSort or BucketSort were usually inferior.
3) We traverse the sorted elements $u \in \mathbb{U}$ and compute $d(u, q)$ for each such u. For range queries, we report any u such that $d(u, q) \leq r$. For K-NN queries, we remember the K database elements that yielded the smallest $d(q, u)$ values so far.
4) We stop the scanning of $\mathbb{U}$ at some point, and then deliver the result as obtained up to then, hoping that it will be close to the result we would obtain by a full scan.

Say that we are willing to traverse $f \cdot n$ elements of $\mathbb{U}$. The total time complexity of the search process is k distance computations and $O(k \log k)$ CPU time for step 1; $O(kn)$ CPU time to compute the $S()$ values (we see later that the measure S we use can be computed in $O(k)$ time) and $O(n + fn \log n)$ CPU time for the incremental sorting at step 2; and finally $f \cdot n$ further distance computations for step 3. This adds up $O(kn + fn \log n)$ CPU time and $k + fn$ distance computations. We tried some alternatives to avoid computing $S()$ for the whole database, but the result was not practical.

The stopping criterion deserves some discussion. The simplest is to scan a fraction $0 < f < 1$ of the database, so that the amount of work is fixed beforehand and we have no control over the quality of the answer. Alternatively, we could like to fix an expected fraction $0 < p < 1$ of the correct answer retrieved. For K-NN queries, this can be obtained by previously building plots like those in the Appendix with a set of training queries. Those plots depend on the space but not on K. Later, given a K-NN query, we consider in the plot the points below $y = K/n \times 100\%$

in the y-axis. Now we find the point x in the x-axis so that a fraction p of those points are to the left of x. This x value is the fraction of the database we should traverse to obtain on average a fraction p of the K correct nearest neighbors. For range queries the mechanism is similar, using a plot that on the y axis gives the distance to the points found, and $y = r$.

B. *Measuring Similarity between Permutations*

It remains to specify how we measure the difference between two permutations. We use Spearman Rho [24], denoted $S_\rho(\Pi_q, \Pi_u)$, as our similarity measure: We sum the squares of differences in the relative positions of each element in both permutations. That is, for each $p_i \in \mathbb{P}$ we compute its position in Π_u and Π_q, namely $\Pi_u^{-1}(i)$ and $\Pi_q^{-1}(i)$, and sum up the squares of the differences in the positions. A formal definition follows.

Definition 2: Given permutations Π_u and Π_q of $(1 \ldots k)$, Spearman Rho is defined as[1]

$$S_\rho(\Pi_u, \Pi_q) \;=\; \sum_{1 \le i \le k} \left(\Pi_u^{-1}(i) - \Pi_q^{-1}(i)\right)^2 .$$

Let us give an example of $S_\rho(\Pi_q, \Pi_u)$. Let $\Pi_q = 6, 2, 3, 1, 4, 5$ be the permutation of the query, and $\Pi_u = 3, 6, 2, 1, 5, 4$ that of an element u. A particular element p_3 in permutation Π_u is found two positions off with respect to its position in Π_q. The differences between permutations are: $1 - 2, 2 - 3, 3 - 1, 4 - 4, 5 - 6, 6 - 5$, and the sum of their squares is $S_\rho(\Pi_q, \Pi_u) = 8$.

There are other similarity measures between permutations [24], such as Kendall Tau and Spearman Footrule. Kendall Tau is defined as follows: For every pair $p_i, p_j \in \mathbb{P}$, if p_i and p_j are in the same order in Π_u and Π_q, (that is $\Pi_u^{-1}(i) < \Pi_u^{-1}(j) \Leftrightarrow \Pi_q^{-1}(i) < \Pi_q^{-1}(j)$) then $K_{p_i,p_j}(\Pi_u, \Pi_q) = 0$; otherwise it is 1. Kendall Tau is given by $K(\Pi_u, \Pi_q) = \sum_{p_i,p_j \in \mathbb{P}} K_{p_i,p_j}(\Pi_u, \Pi_q)$, which turns out to be equal to the number of exchanges needed by a bubble sort to convert one permutation into the other. The Spearman Footrule between two permutations is

$$F(\Pi_u, \Pi_q) \;=\; \sum_{1 \le i \le k} |\Pi_u^{-1}(i) - \Pi_q^{-1}(i)|.$$

[1]The actual definition in [24] corresponds to $\sqrt{S_\rho(\Pi_q, \Pi_u)}$ in our terminology. We omit the square root because it is monotonous and hence does not affect the ordering.

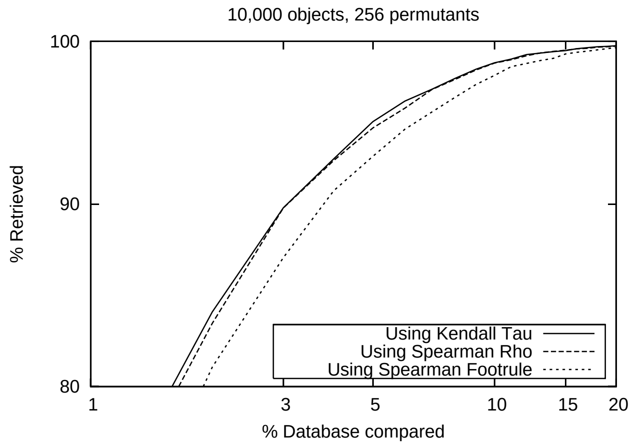


Fig. 2. Using different similarity measures between permutations (log scale). The space is a random uniformly distributed set of 10,000 points in the unitary cube of dimension 128 with Euclidean distance. 256 permutants were used.

In Figure 2 we show that $F()$ is not as good as $S_\rho()$ for our purposes (similar results were obtained in other metric spaces). On the other hand, $K()$ performs similarly to $S_\rho()$, but it is more cumbersome to compute. Thus we stick to Spearman Rho in the sequel.

We promised that S_ρ would be computable in linear time. According to Definition 2, this is easy if we store the *inverse* permutations Π_u^{-1} and Π_q^{-1}. As we prove next, it is enough to invert one of them to compute S_ρ in $O(k)$ time. Therefore we actually use Π_q^{-1} instead of Π_q.

Lemma 1: Definition 2 is equivalent to

$$S_\rho(\Pi_q, \Pi_u) \;=\; \sum_{1 \le j \le k} \left(j - \Pi_q^{-1}(\Pi_u(j))\right)^2 .$$

Proof: It is a matter of calling $j = \Pi_u^{-1}(i)$ and summing in different order. ■

Algorithm 1 gives the complete pseudocode for range searching. It receives the query (q, r) and the fraction of the database $0 < f < 1$ to examine. The permutations Π_u, as well as the sets $\mathbb{U}$ and $\mathbb{P}$, are global variables. The database and the S_ρ values are stored as tuples $\langle u_i, S_\rho(\Pi_{u_i}, \Pi_q)\rangle$ in an array A, which is computed and then partially traversed to retrieve the (approximate) answer. For simplicity we describe the algorithm as fully sorting A, not incrementally.

Algorithm 1 Sort-rangeQuery(q,r,f)

```
1: INPUT: $q$ is a query and $r$ its radius, $f$ is the fraction of the database to traverse.
2: OUTPUT: Reports a subset of those $u \in \mathbb{U}$ that are at distance at most $r$ to $q$.
3: Let $A[1,n]$ be an array of tuples and $\mathbb{U} = \{u_1, \ldots, u_n\}$
4: Compute $\Pi_q^{-1}$
5: for $i \leftarrow 1$ to $n$ do
6:    $A[i] \leftarrow \langle u_i, S_\rho(\Pi_{u_i}, \Pi_q) \rangle$
7: end for
8: SortIncreasing($A$) // by second component of tuples
9: for $i \leftarrow 1$ to $f \cdot n$ do
10:    Let $A[i] = \langle u, s \rangle$
11:    if $d(q,u) \leq r$ then
12:       Report $u$
13:    end if
14: end for
```

IV. Experimental Evaluation

In this section we evaluate and compare the performance of our technique in different metric spaces, such as synthetic vectors on the unitary cube and clustered data (multivariate Gaussian distribution), as well as real-life databases like face images and text documents. We also tested the algorithm in non-metric spaces, where the triangle inequality does not hold. All the experiments reported excellent results for our method. The experiments were carried out on a Intel Xeon workstation with 2.6 GHz CPU and 4GB of RAM with Red Hat Linux, running kernel 2.4.20-9.

A. Unitary Cube

We made some experiments using uniformly distributed sets of 10,000 points in the unitary cube, in 128, 256, 512 and 1024 dimensions, under Euclidean distance. As we can precisely control the dimensionality of the space, we use this experiment to show how the predictive power of permutants varies with the dimensionality, compared with other methods. We tested range queries with a search radius that retrieved on average 0.05% of the database (that is, 5 points). We emphasize that no exact algorithm can avoid a linear scan of the database when we go over dimensionality 30 with uniformly distributed points, only probabilistic algorithms work.

We considered $k = 128$ and $k = 256$ permutants in our experiments. We compare our technique with a standard pivot-based method using the same amount k of pivots, even though

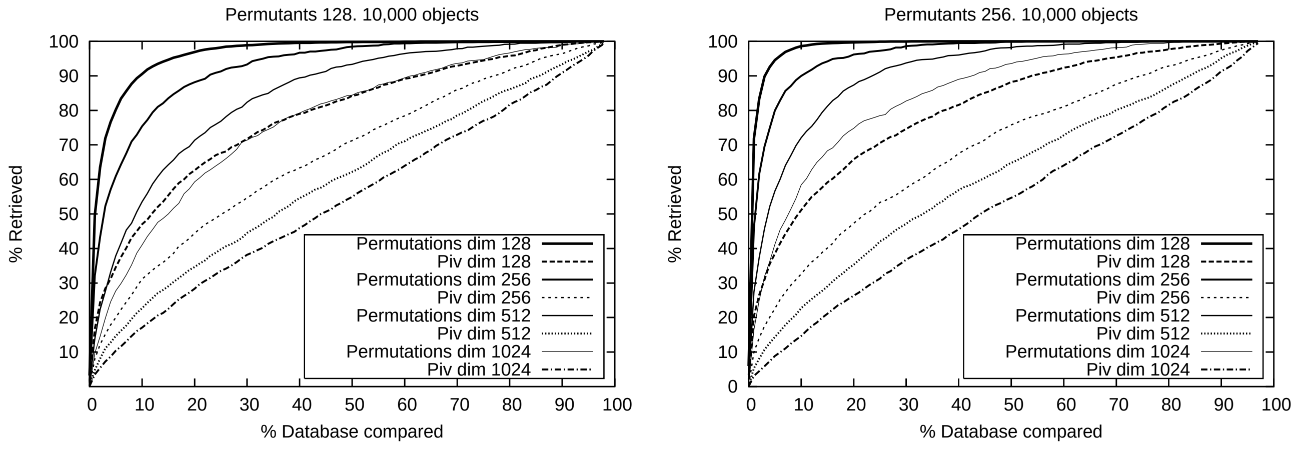


Fig. 3. Performance of ours versus pivot-based probabilistic algorithms in different dimensionalities. On the left we use 128 pivots/permutants, and 256 on the right. Series with the word piv refer to the standard pivot-based algorithm.

this represents at least 4 times the memory we use for our algorithm. If we used the same amount of memory for the two algorithms, the comparison would be even more favorable to us.

The pivot-based probabilistic alternative we tested [10] calculates, for each database element u, estimate $L_\infty(q,u) = \max_{p \in \mathbb{P}} |d(q,p) - d(p,u)|$. The database is then sorted by increasing L_∞ value and compared against the query in this order.

Figure 3 shows the comparison. The x axis represents the percentage of the database examined, and the y axis is the percentage of the actual answer that was retrieved (this estimates the probability of returning a given answer element).

Retrieving 90% of the answer is good enough for most proximity searching applications. With 128 pivots, in dimensionality 128, 60% of the database must be examined to retrieve 90% of the results. For our permutation-based algorithm, with 128 permutants we must examine only 10% of the database to retrieve 90% of the outcome. This raises to 99% if we use 256 permutants. With 256 pivots, instead, one needs to compare 85% of the database to retrieve 99% of the answers.

In general we observe that, as the dimensionality grows, a larger fraction of the database must be examined to obtain a given fraction of the result. This observation is true for the pivot-based algorithm as well as for ours. Yet, the pivot-based algorithm is more affected by dimensionality than ours. Note that an algorithm that traverses the database in random order would achieve a

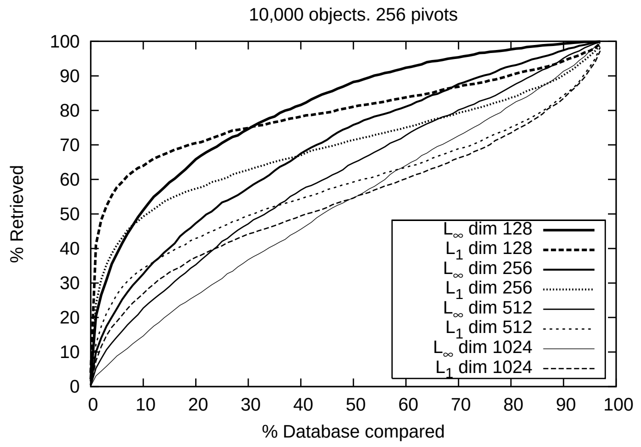


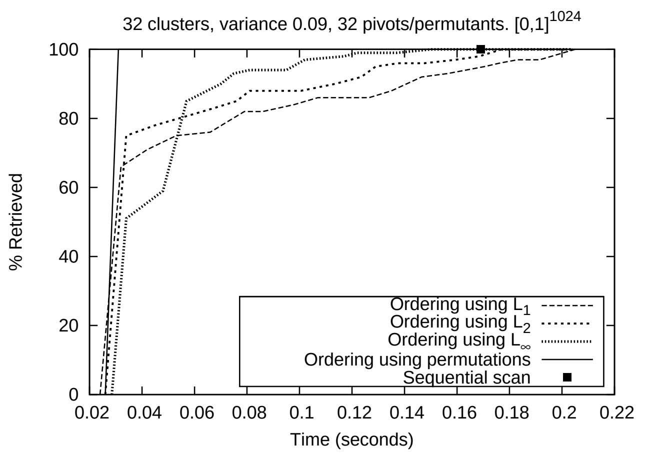
Fig. 4. Comparison between L_1 and L_∞ Minkowski metrics to sort the database with pivot-based algorithms, using 256 pivots.

straight line from the bottom-left to the top-right corner, that is, it needs to examine 90% of the database to obtain 90% of the answer. It can be seen that pivot-based algorithms actually behave almost randomly on very high dimensionalities.

Note that in this synthetic data we may be using more permutants than space coordinates. Since the permutation similarity is more expensive to compute than plain Euclidean distance, this may seem nonsense. We remark that this experiment is just to demonstrate the performance of the technique in terms of distance computations. Real data may have thousands of coordinates or no coordinates at all. We include real CPU times for all the other metric spaces that follow.

One might wonder whether the L_∞ distance used by the pivot-based probabilistic algorithm is a good predictor. Although there are good reasons to use L_∞ [10], one can also argue in favor of L_1: AESA, the best exact algorithm [40], uses L_1 metric as the oracle to select next-best candidates for pruning the database, that is, $L_1(q,u) = \sum_{p \in \mathbb{P}} |d(q,p) - d(p,u)|$. We tested in Figure 4 the L_1 distance to sort the database for the probabilistic algorithm based on pivots, versus the L_∞ choice used above. It can be seen that the results are mixed. In the first part (e.g., scanning less than 20% of the database in dimensionality 128) distance L_1 retrieves a larger percent of the database compared to L_∞. Yet, once a turn point is reached, the result is reversed. The same behavior is observed in all the dimensionalities considered. We emphasize that, anyway, the results are very far from what we obtain with our new technique.

In the Appendix we display the power of the sorting methods using clouds of points. These



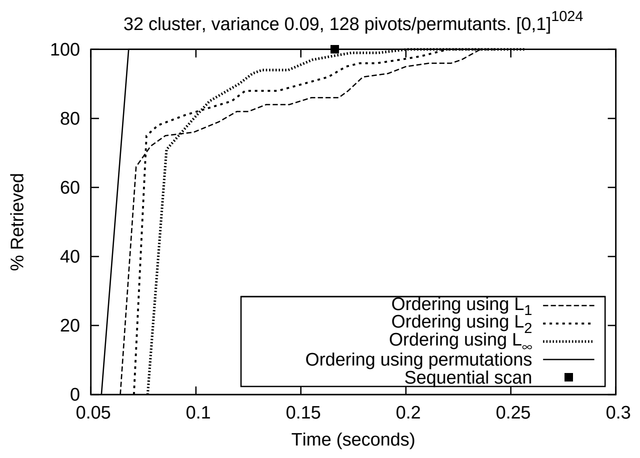


Fig. 5. Retrieving the nearest neighbor on a 1024-dimensional Gaussian space with 32 clusters, using 32 (left) and 128 (right) pivots/permutants. We show the retrieval percentage versus the total time to obtain the results.

clouds show how often our technique put nearest neighbors in the first positions.

B. Gaussian Spaces

Uniformly distributed data is full-dimensional. Real datasets behave more like clustered data, which is easier to index. We tested our algorithm on a Gaussian space. The data was generated for a 1024-dimensional space $[0,1]^{1024}$ with 10,000 points obtained from a multivariate Gaussian distribution with 32 clusters (centers). The variance of the center distribution was 0.09, and the variance inside the clusters was 0.01.

Figure 5 shows experiments of the CPU time needed for retrieving the nearest neighbor using 32 and 128 pivots/permutants. Notice that the ordering using permutations retrieves 100% of the answer faster than the others. On the left, using 32 pivots/permutants, ordering using permutations retrieves 100% of the answer in just 0.03 seconds, while the others require 0.17 seconds.

C. Face Recognition

In many real-world scenarios, objects are modeled as very high-dimensional feature vectors. Spatial access techniques cannot be used efficiently in this case, due to the curse of dimensionality. An alternative is to work without coordinates, using the distance just as a black box, that is, resorting to the metric space model. Yet, in several cases the resulting intrinsic dimensionality is still very high and no exact search method can avoid an exhaustive scan of the database.

In this section we consider the FERET database [37], which consists of 762 grayscale frontal face images of 254 different persons (3 images per person). The pictures are of 128×128 pixels, that is, each face is represented by 16,384 features. The query set has 254 images (1 image per person). To speed up searches, the vectors were transformed by eigenspace methods, which project the input faces onto a 761-components (coordinates) space where the recognition is carried out.

We consider K-NN search, as this is the most frequent query in this application. For the probabilistic algorithms, we measure the number of distance computations performed (averaged over all the queries) until the algorithms obtain the correct K nearest neighbors. We used all the 254 queries for each K value tested.

Since the size of the database allows it, we included AESA [40] in the comparison, as it is considered a baseline to compare exact searching algorithms. AESA uses the entire distance matrix to answer queries, and it is the best exact algorithm. As the distance is Euclidean, we also experiment with a kd-tree [7] as an exact search method that attempts to reduce CPU time.

Figure 6 (top) shows the results, using 64 permutants. It can be seen that the best exact technique (AESA) requires scanning 30%–40% of the database to find the nearest neighbor, and this quickly raises to 80%–90% for larger K. *Kd*-trees need 50% to find the nearest neighbor. Our technique performs better, scanning around 10% of the database on average to find the nearest neighbor, and 30%–40% for 20 nearest neighbors. For the probabilistic algorithm based on pivots we chose the L_1 distance to sort the database. It requires to traverse a larger fraction of the database to achieve the same result of permutants (40%–50% for $K = 20$ neighbors). The results for the L_∞ distance were not included as they are worse than for L_1.

Figure 6 (bottom) shows real CPU times. It can be seen that, although permutations pose a CPU time overhead higher than pivots, the result is still advantageous in terms of CPU time. (Note that AESA is more expensive in practice than a sequential scan.)

Figure 7 displays the results in a form more similar to previous plots. We show the percentage of queries successfully solved (that is, all their K nearest neighbors are found) after traversing a given percentage of the database. We also display the *relative error* ratio between the distance to the K-th nearest neighbor found divided by the distance to the true K-th nearest neighbor

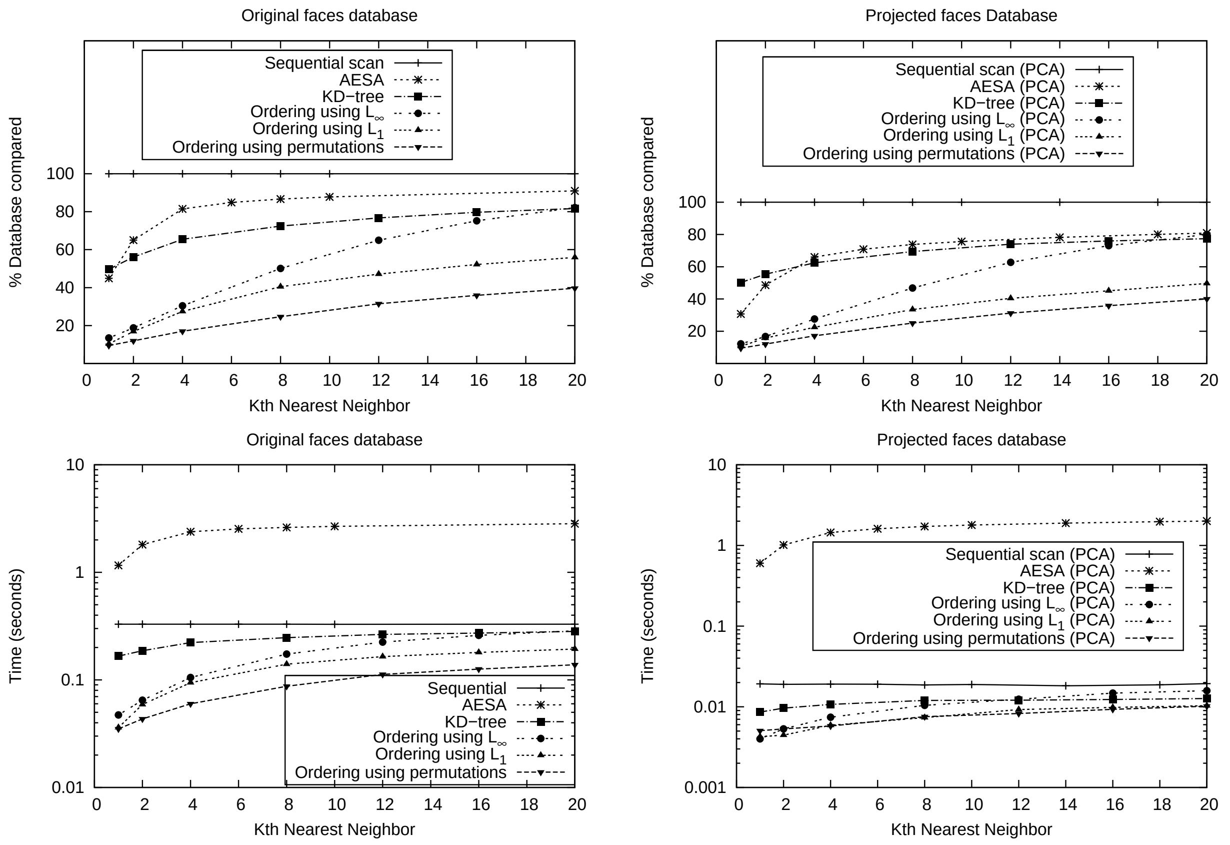


Fig. 6. Comparing techniques over a real database of faces. We show the percentage of the database compared (top) and CPU time (bottom) to find the correct K nearest neighbors, using 64 pivots/permutants. On the left we work with the original space; on the right with the projected space.

(computed only over the unsuccessful queries). It can be seen that, even when the algorithm fails to find the true answer, the approximation it finds is rather good.

Again, in the Appendix we display the power of the sorting methods for this database.

D. Documents

A central problem in Information Retrieval consists in finding documents relevant to a given query. The relevance is measured using a specialized distance definition. Documents are represented as unitary vectors, where every coordinate corresponds to a term, and the value of a document vector along each coordinate is proportional to the weight of the term in that document. The number of different terms in a collection is in the order of hundreds of thousands, resulting in a very high-dimensional vector space with the usual dimensionality curse problems. The distance

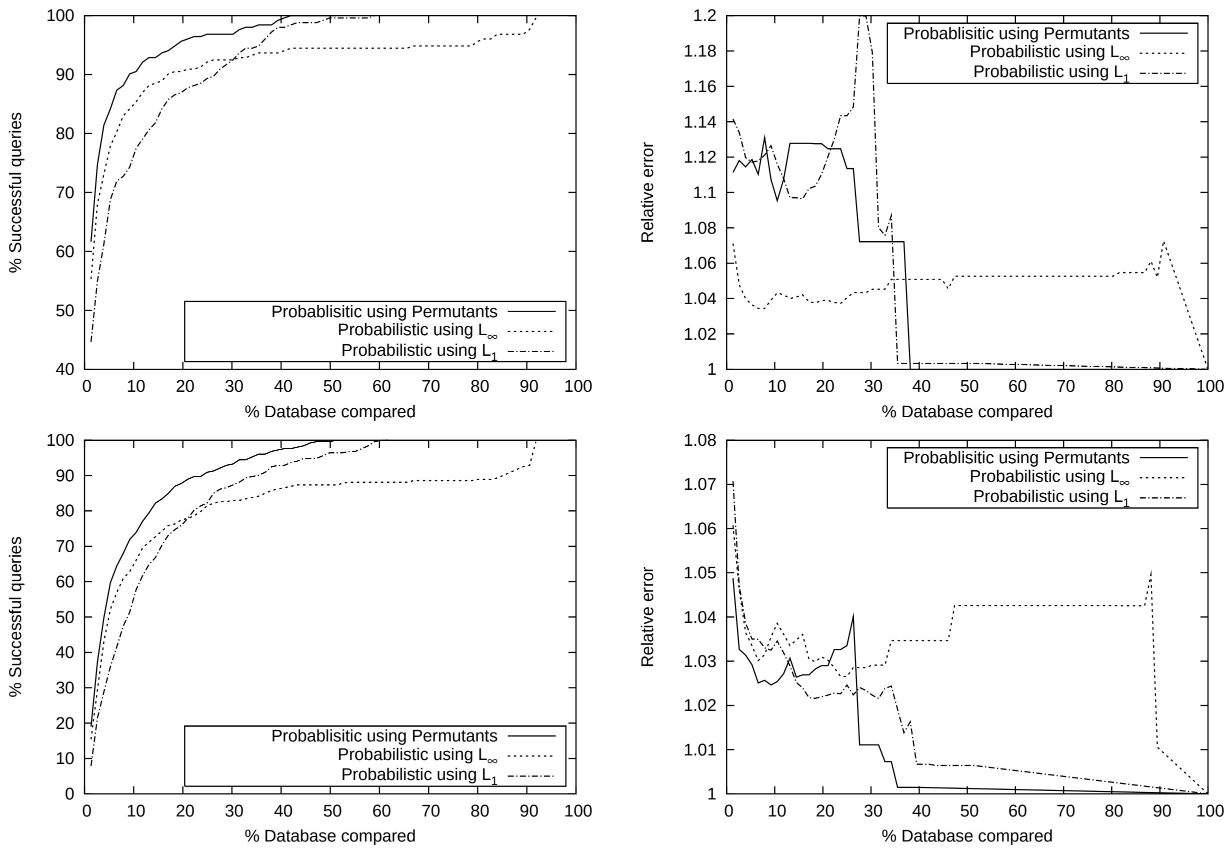


Fig. 7. Comparing techniques over a real database of faces. On top for $K = 2$ and on the bottom for $K = 4$. On the left, percentage of queries where all the K nearest neighbors are correctly found. On the right, relative error for those queries that do not find all the correct neighbors.

between two documents can be taken as the angle between their representing vectors (the cosine of this angle is a similarity measure heavily used in Information Retrieval [6]).

We used a subset of collection TREC-3 [26] to compare the performance of our approach against the best previous results using probabilistic algorithms [10]. The database consists of 24,960 documents. We averaged 1,000 range queries chosen at random, with a radius retrieving on average 0.035% of the database (9 documents). No exact algorithm performs well in this setup: Even AESA needs to compare the query against 60% of the database to solve this query.

The results can be seen in Figure 8, using 128 pivots or permutants. Permutations quickly reach a good percentage of retrieval: We review just 2% of the database to retrieve 95% of the outcome, while the classical pivot-based algorithm (i.e., using L_∞ ordering) needs to review

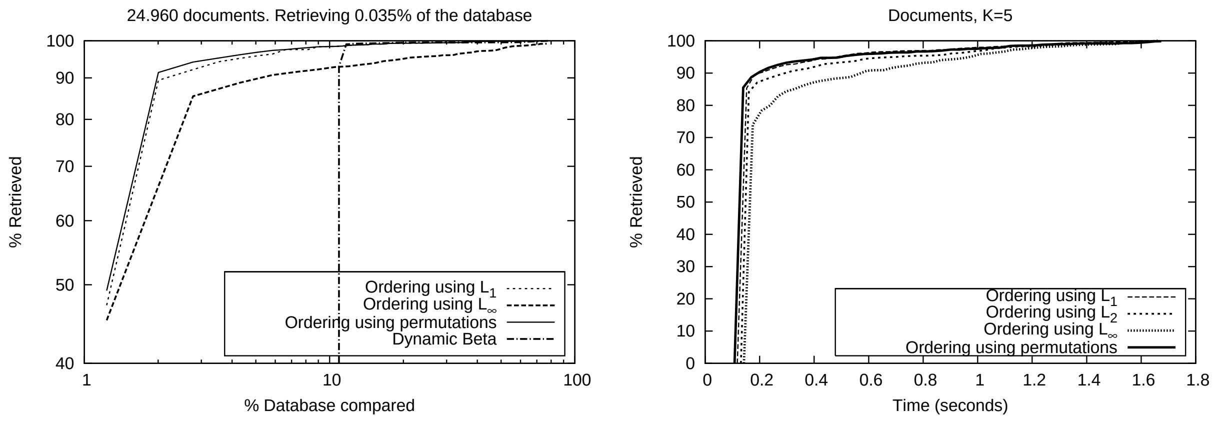


Fig. 8. Comparing our technique with others in a real database of documents. On the left, retrieval performance, on the right, the CPU time compared against retrieval performance for all the probabilistic algorithms.

almost 20% of the database to achieve the same retrieval performance. A pivot-based algorithm using L_1 (not tried before as far as we know) performs almost as well as permutations. Finally, in [10] a method called *Dynamic Beta* is proposed, which needs to review about 10% of the database to reach the same retrieval performance. We note that Dynamic Beta, after paying that 10% of comparison, surpasses by far the pivot-based method, and from then on it becomes similar to permutations.

Figure 8 (right) shows the result of a 5-NN query, this time focusing on CPU times. Again using permutations is (slightly) faster than the others.

We again display the power of the sorting methods on this database using clouds of points in the Appendix.

E. Non-Metric Databases

There are several real-life applications where similarity searching has to be carried out over a space that is not even metric, i.e., where the triangle inequality does not hold. In this case exact proximity search algorithms are useless in general, as there is no way to prove that an element is sufficiently far away from the query q. A probabilistic algorithm, instead, has a chance of still proposing an appealing order to traverse the database. A variation of this idea, forging a monotonous transformation of the database, is indeed used in [39] as a good alternative to search in non-metric databases.

In particular, our probabilistic algorithm does not make use of the triangle inequality, as it never discards an element; it just hints which are the most promising candidates to consider first. As such, it can be used on non-metric databases.

We apply our K-NN algorithm over a couple of non-metric spaces, in order to demonstrate its suitability. The first space is a synthetic uniform vector space just as those in Section IV-A using, instead of Euclidean distance, a so-called *fractional norm* L_p with $0 < p < 1$:

$$L_p((x_1,\ldots,x_D),(y_1,\ldots,y_D)) \;=\; \left(\sum_{1\le i\le D} |x_i - y_i|^p\right)^{\frac{1}{p}} .$$

Fractional norms are sometimes preferred over the usual Minkowski norms L_1, L_2 or L_∞, because they lead to lower intrinsic dimensionality [2], [1], [21], [28]. (Please do not confuse this norm, that is used as the d distance in the metric space, with the L_1 and L_∞ norms explained in Section IV-A to sort the database. These are independent.)

Figure 9 compares the performance of our ordering based on permutations with those based on L_1 and L_∞ as in previous sections. It can be seen that permutations achieve the best result, followed by L_1. The problem is easier as p grows and the space gets closer to be metric.

The second space is that of sequences using *normalized edit distance (NED)* [31], [3]. The usual edit distance (which is a metric) favors short sequences over long ones, given the same fraction of similarity between the two sequences. The NED counterweights this bias by dividing the cost of a sequence of operations by the length of that sequence. The result is not anymore a metric, but it works better in several applications.

Figure 10 shows the results over 40,000 words from a dictionary using this distance, for a range search with radius 1. In this case, the permutations and the L_1 orderings yield similar results, superior to those of L_∞.

F. Selecting Permutants

Permutants are central to our method. Hence, it is worthy to investigate the role of permutant selection. We tested heuristics based on selecting permutants with minimum or maximum Spearman Rho in the set: We start with a set with only one element, and the next permutant

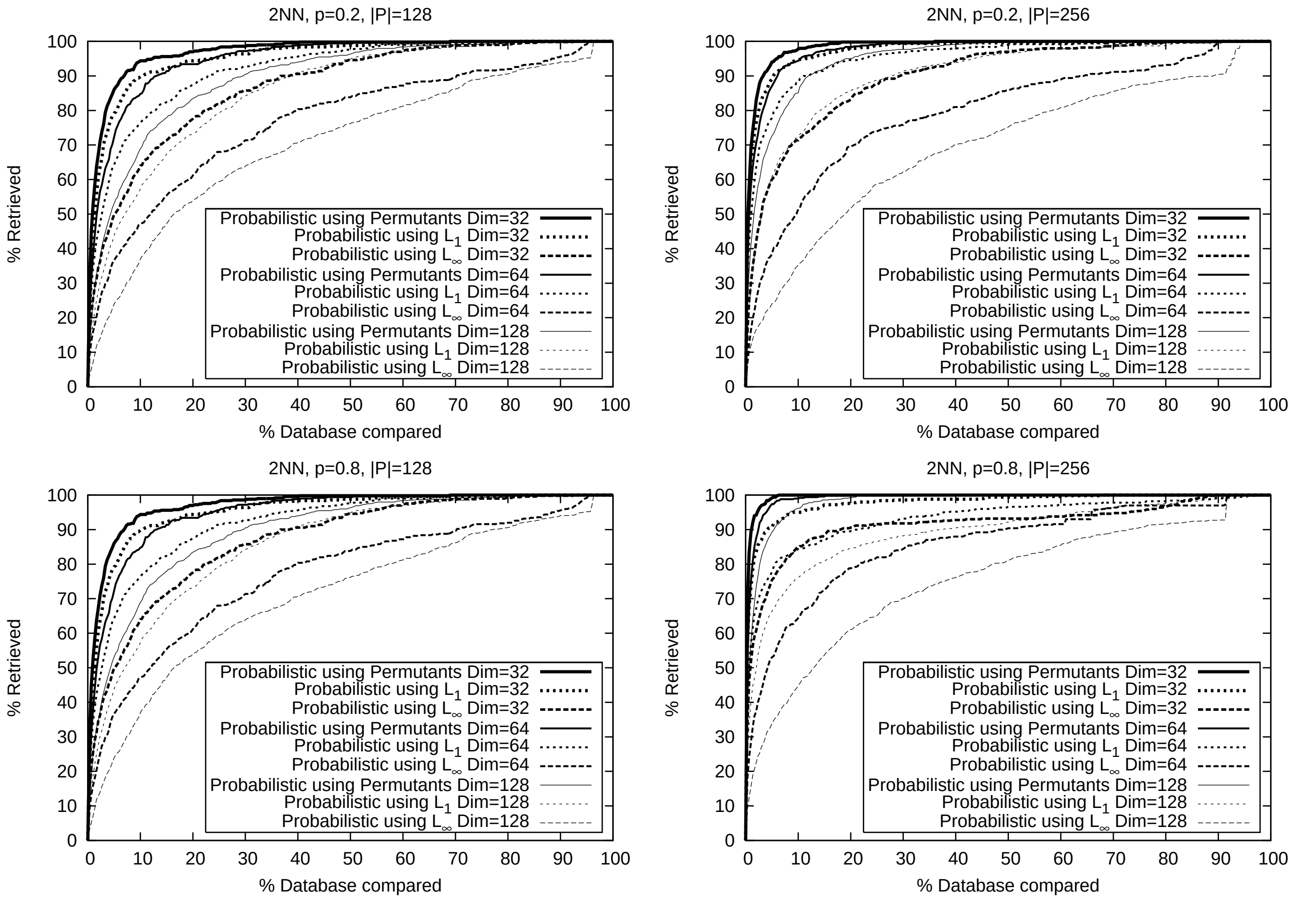


Fig. 9. Comparing our technique with others in uniformly distributed vector spaces using L_p distance (non-metric, $p < 1$), to retrieve two nearest neighbors. On top $p = 0.2$ and on the bottom $p = 0.8$; using 128 pivots on the left and 256 on the right.

will be selected minimizing (or maximizing) the sum $p = \min_{u_i \in \mathbb{U}} \sum_{p_j \in \mathbb{P}} S_p(u_i, p_j)$. This type of heuristic has been successful to choose pivots [10]. Its complexity is $O(k^3 n)$.

We show experiments in Figure 11, for uniformly distributed data (top) and Gaussian data (bottom), with the setup of previous sections. As can be seen, no significant improvement is obtained with the different heuristics. In some cases random selection is even better than the alternatives. Other experiments, choosing artificially the permutants as the centers used to generate the Gaussian data, failed as well.

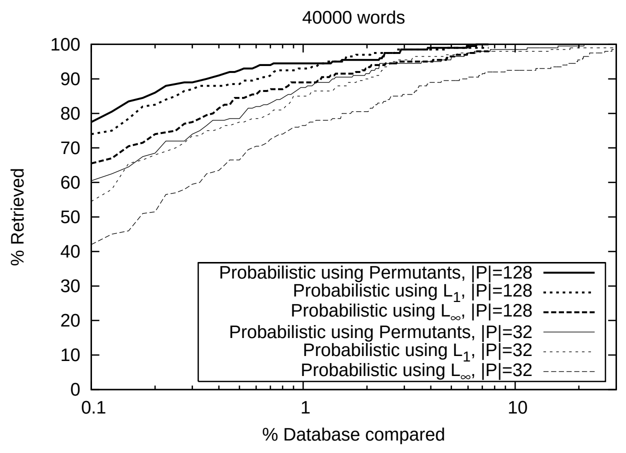


Fig. 10. Comparing our technique with others in a space of strings using normalized edit distance. Range search with radius 1.

V. Conclusion and Future Work

We have presented a new method for probabilistic proximity searching in metric spaces. It is based on comparing the proximity ordering towards a set of distinguished objects (called permutants). We show that this ordering is a very good predictor of the relevance of points to the query. This leads to a very strong probabilistic proximity search algorithm, which needs to scan just a small fraction of the database to obtain most of the relevant answers. Our technique is by far better than any other existing proposal we are aware of.

Our proposal is very simple to implement and has immediate applications to many pattern recognition problems, as well as in other areas that use proximity searching and can tolerate (very good) approximations to the exact solutions to proximity queries. One application we have pursued was to use our technique as an oracle to choose the pivots in AESA, the best exact proximity search algorithm: We use S_ρ instead of L_1 [25]. The result, iAESA, achieves an interesting reduction over an algorithm that had standed out as unbeatable for 20 years. Another idea we are pursuing is to use our algorithm to build approximate K-NN graphs, which are useful for many applications including proximity searching [34]. Our preliminary results indicate that we obtain almost always the correct K-NN graph at very low cost compared to exact construction algorithms such as [35].

On the other hand, several aspects of our technique deserve more research. One challenge is

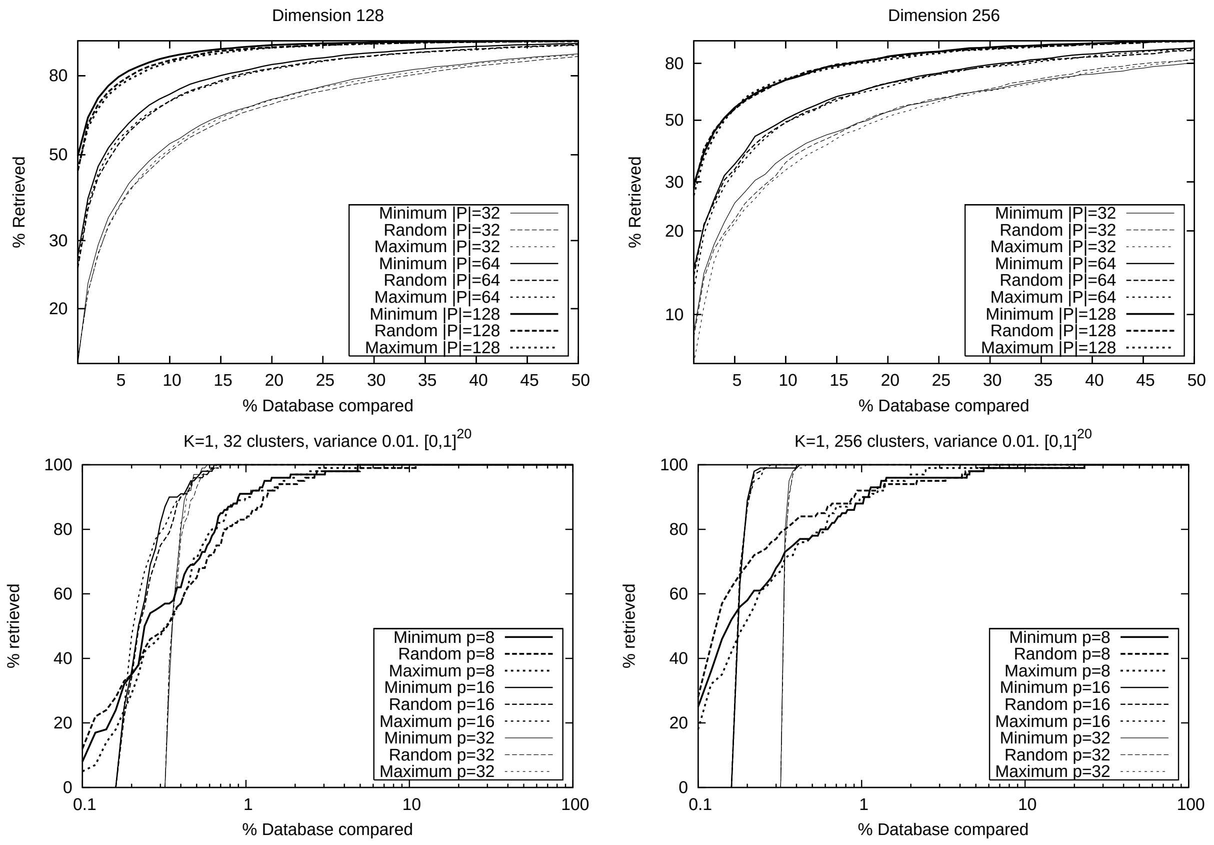


Fig. 11. Different heuristics to select permutants. On top, range searching that retrieves 0.05% of the database on uniform data. On the bottom, 1-NN on clustered data.

to reduce CPU times. Although we have shown that permutants obtain good CPU times when the distance function is moderately expensive to compute, it might be possible to do better. In particular, our best current solutions still take time proportional to the database size (albeit with a small constant in practice). Another is to devise new methods to determine where to stop the scanning so as to achieve some expected quality in the answer. Our method to do this requires training. Maybe it is possible to use the history of the updates to the answer produced by the current query to predict its future behavior.

References

[1] C. Aggarwal. Re-designing distance functions and distance-based applications for high dimensional data. *ACM SIGMOD*, 30(1):13–18, 2001.

[2] C. Aggarwal, A. Hinneburg, and D. Keim. On the surprising behavior of distance metrics in high dimensional spaces. In *Proc. 8th Intl. Conf. on Database Theory*, LNCS 1973, pages 420–434, 2001.

[3] A. Arslan and O. Egecioglu. Efficient algorithms for normalized edit distance. *J. Discrete Algorithms*, 1(1):3–20, 2000.

[4] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimension. In *Proc. 5th Symposium on Discrete Algorithms (SODA)*, pages 573–583, 1994.

[5] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM)*, LNCS 807, pages 198–212, 1994.

[6] R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley, 1999.

[7] J. Bentley. Multidimensional binary search trees used for associative searching. *Comm. of the ACM*, 18(9):509–517, 1975.

[8] S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Very Large Databases (VLDB)*, pages 574–584, 1995.

[9] W. Burkhard and R. Keller. Some approaches to best-match file searching. *CACM*, 16(4):230–236, 1973.

[10] B. Bustos and G. Navarro. Probabilistic proximity search algorithms based on compact partitions. *J. Discrete Algorithms*, 2(1):115–134, 2003.

[11] E. Chávez and K. Figueroa. Faster proximity searching in metric data. In *Proc. Mexican Intl. Conf. in Artificial Intelligence (MICAI)*, LNAI Series, pages 222–231, 2004.

[12] E. Chávez, J. Marroquín, and R. Baeza-Yates. Spaghettis: an array based algorithm for similarity queries in metric spaces. In *Proc. 6th String Processing and Information Retrieval (SPIRE)*. IEEE Computer Science Society, 1999.

[13] E. Chávez, J.L. Marroquin, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications (MTAP)*, 14(2):113–135, 2001.

[14] E. Chávez and G. Navarro. Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Information Processing Letters*, 85(1):39–46, 2003.

[15] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[16] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[17] P. Ciaccia and M. Patella. PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proc. 16th Intl. Conf. on Data Engineering (ICDE)*, pages 244–255. IEEE Computer Society, 2000.

[18] P. Ciaccia and M. Patella. Searching in metric spaces with user-defined and approximate distances. *ACM TODS*, 27(4):398–437, 2002.

[19] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. 23rd Conf. on Very Large Databases (VLDB)*, pages 426–435, 1997.

[20] K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.

[21] K. Doheryu, R. Adams, and N. Davey. Non-euclidean norms and data normalisation. In *Proc. 12th European Symposium on Artificial Neural Networks*, 2004.

[22] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley, 2nd edition, 1973.

[23] Omer Egecioglu. Parametric approximation algorithms for high-dimensional euclidean similarity. In *Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, volume LNAI 2168, pages 79–90, 2001.

[24] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Mathematics*, 17(1):134–160, 2003.

[25] K. Figueroa, E. Chávez, G. Navarro, and R. Paredes. On the least cost for proximity searching in metric spaces. In *Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA)*, LNCS 4007, pages 279–290, 2006.

[26] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conf. (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[27] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM TODS*, 28(4):517–580, 2003.

[28] P. Howarth and S. Rüger. Fractional distance measures for content-based image retrieval. In *Proc. 27th European Conf. on IR Research (ECIR)*, LNCS 3408, pages 447–456, 2005.

[29] Oxford J. Kittler, NATO ASI, editor. *Computational geometric problems in pattern recognition*. Pattern Recognition Theory and Applications, 1981.

[30] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *Transactions on Software Engineering*, 9(5), 1983.

[31] A. Marzal and E. Vidal. Computation of normalized edit distance and applications. *IEEE TPAMI*, 15(9):926–932, 1993.

[32] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

[33] G. Navarro. Searching in metric spaces by spatial approximation. *Very Large Databases Journal*, 11(1):28–46, 2002.

[34] R. Paredes and E. Chávez. Using the k-nearest neighbor graph for proximity searching in metric spaces. In *Proc. 12th String Processing and Information Retrieval (SPIRE)*, LNCS 3772, pages 127–138, 2005.

[35] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. In *Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA)*, LNCS 4007, pages 85–97, 2006.

[36] R. Paredes and G. Navarro. Optimal incremental sorting. In *Proc. 8th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 171–182. SIAM Press, 2006.

[37] P. Phillips, H. Wechsler, J. Huang, and P. Rauss. The FERET database and evaluation procedure for face recognition algorithms. *Image and Vision Computing Journal*, 16(5):295–306, 1998.

[38] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[39] Tomás Skopal. On fast non-metric similarity search by metric access methods. In *Proc. EDBT*, pages 718–736, 2006.

[40] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[41] D. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, La Jolla, California, July 1996.

[42] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge, ALENEX'99*, Baltimore, MD, 1999.

[43] P. N. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. Technical report, NEC Research Institute, Princeton, NJ, June 1999.

[44] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.

APPENDIX: CLOUDS OF POINTS

A nice way to visualize the power of the different database sorting methods (L_1, L_∞, permutants) advocated in [10] is to draw *clouds of points* as follows. We sort the database with either method and plot, for the i-th point in that order (x axis) the position where the point falls if we sort the database by actual distance to the query (y axis). The more similar the cloud to a line from (0,0) to (100,100), the better the method predicts true proximity. This experiment is independent on the type of query intended, nor its radius r or number of neighbors k to retrieve.

Figure 12 displays the results for uniformly distributed vectors, in dimension 8 (64 pivots) and 128 (256 pivots). We aggregate 150 queries in the clouds. We can see that permutants obtain a better database ordering. is more consistent with the true ordering by distance to the query. This good ordering somewhat blurs for the higher dimension (curse of dimensionality), but permutations still perform consistently better (for example, L_∞ looks basically random).

As we usually are interested only in the closest elements to the query, and wish to traverse a small percentage of the database, we obtain two zooms from the previous plots. A *precision* plot considers only scanning the first 10% of the database, and displays the positions of the elements found in that traversal. A *recall* plot considers, along the whole database traversal, when are the 10% closest elements found. Figure 13 displays those plots for dimension 128 (now aggregating 500 queries). It can be confirmed that the permutants behave better in both precision and recall.

Figure 14 (left) shows the clouds of points for faces, according to Section IV-C, using 64 pivots. Again, permutants deal better with this space, although the difference is not that large. Note that, with L_∞ there is a strange line close to 100% in the y axis. This is a cluster, far away from the query, which is not handled well with this ordering, but rather its elements are spread across all the spectrum in x. Figure 15 shows the corresponding precision and recall plots.

In Figure 14 (right) we show the clouds of points for documents, according to Section IV-D, using 128 pivots. This time the clusters in the space, and how different orderings deal with them, is more apparent. For example, there is a cluster at distance 40%–60% from the query, which permutations leave for the end of the ordering, whereas the other methods put in the range 20%–40% of their traversal. Important data at distance below 30% of the query are considered with L_1 and L_∞ ordering only when traversing 80% of the database, whereas permutants have found almost all of the 40% closest results after traversing 40% of the database (same with the closest 20%). The precision and recall plots in Figure 16 show that permutants are by far superior in both aspects.

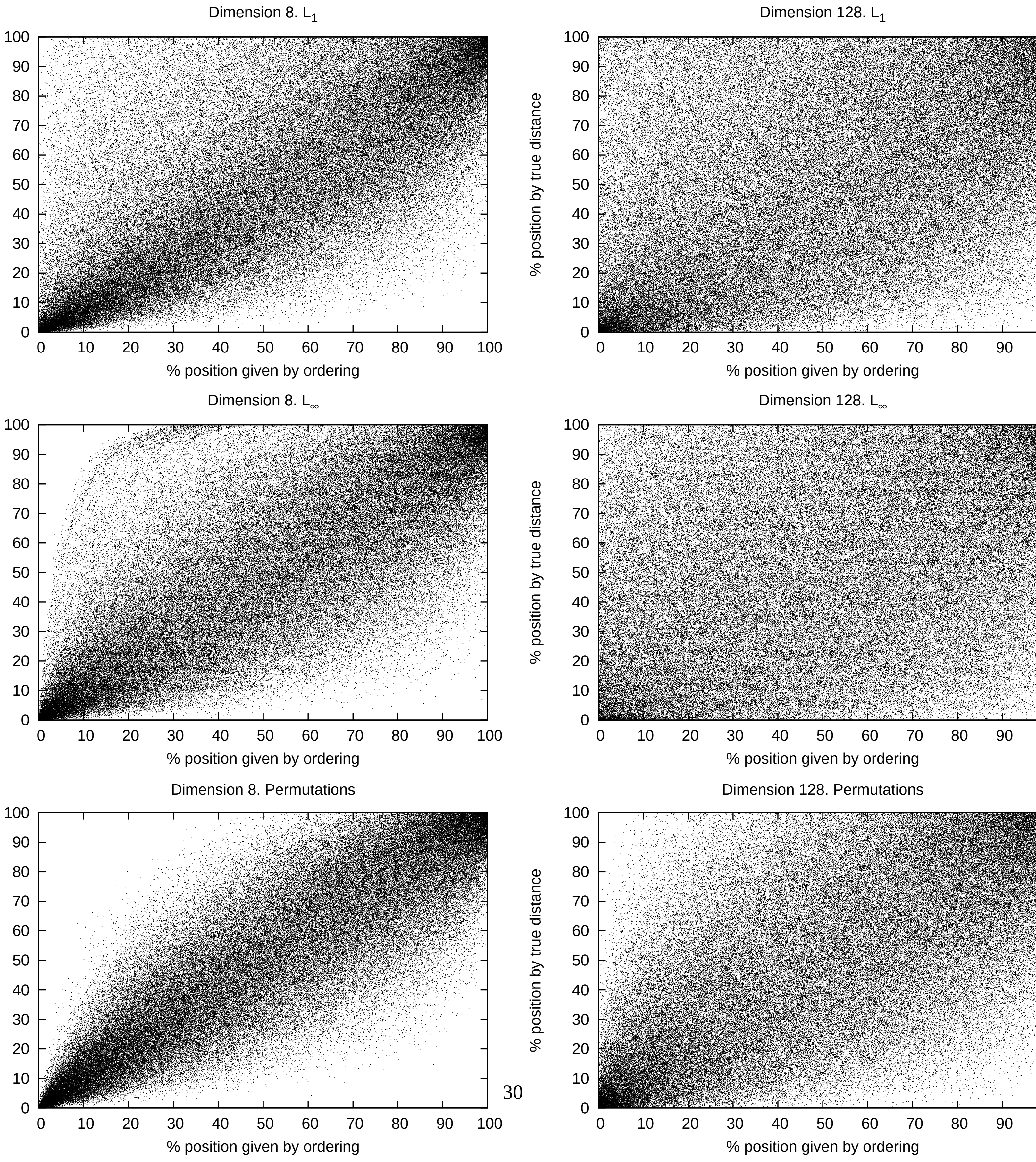


Fig. 12. Clouds of points for uniformly distributed points in dimension 8 and using 64 pivots (left) and dimension 128 using 256 pivots (right). From top to bottom, L_1, L_∞, and permutations.

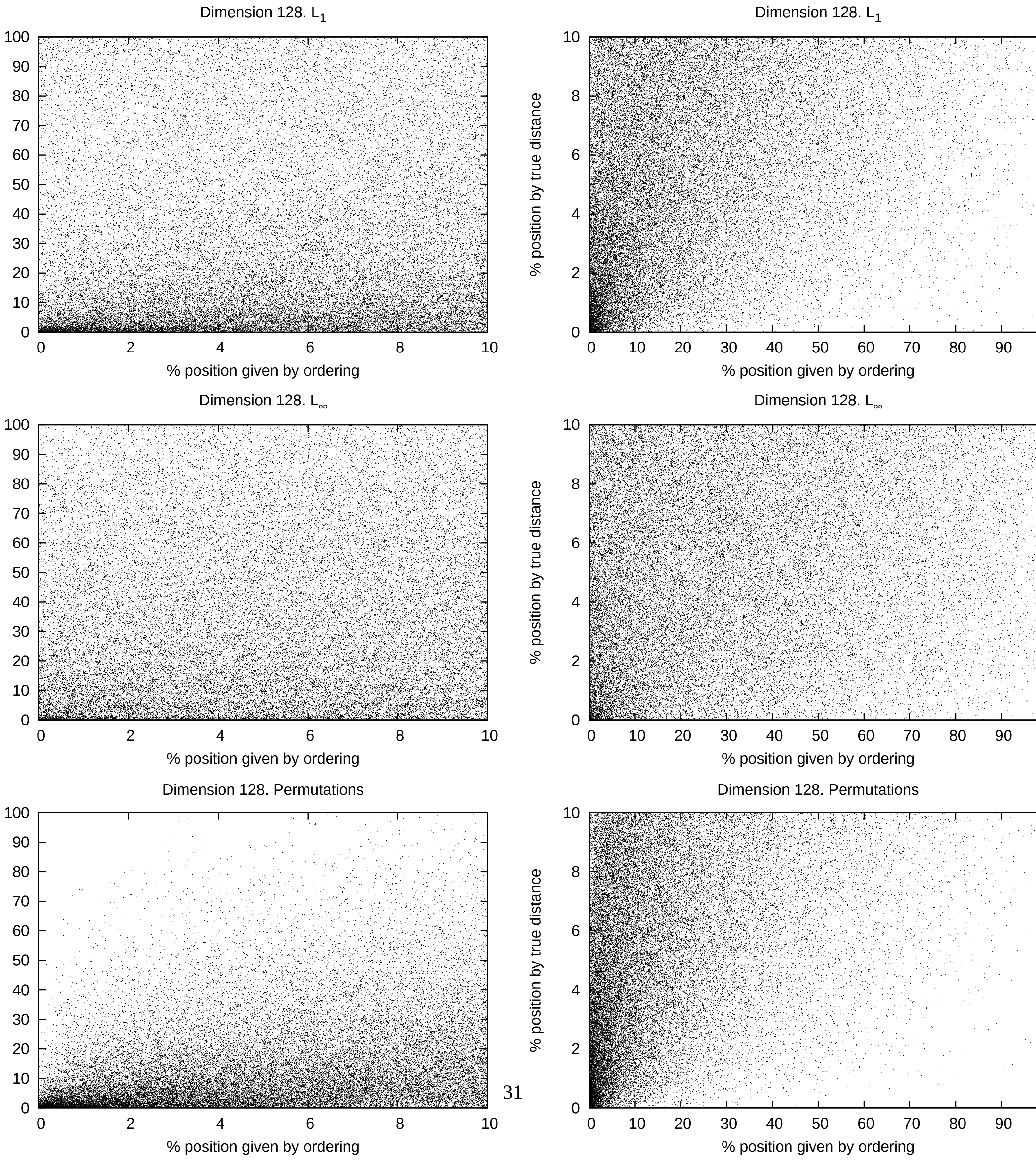


Fig. 13. Zooms for the clouds of points for uniformly distributed points in dimension 128 and 256 pivots. Precision plots on the left and recall plots on the right. From top to bottom, L_1, L_∞, and permutations.

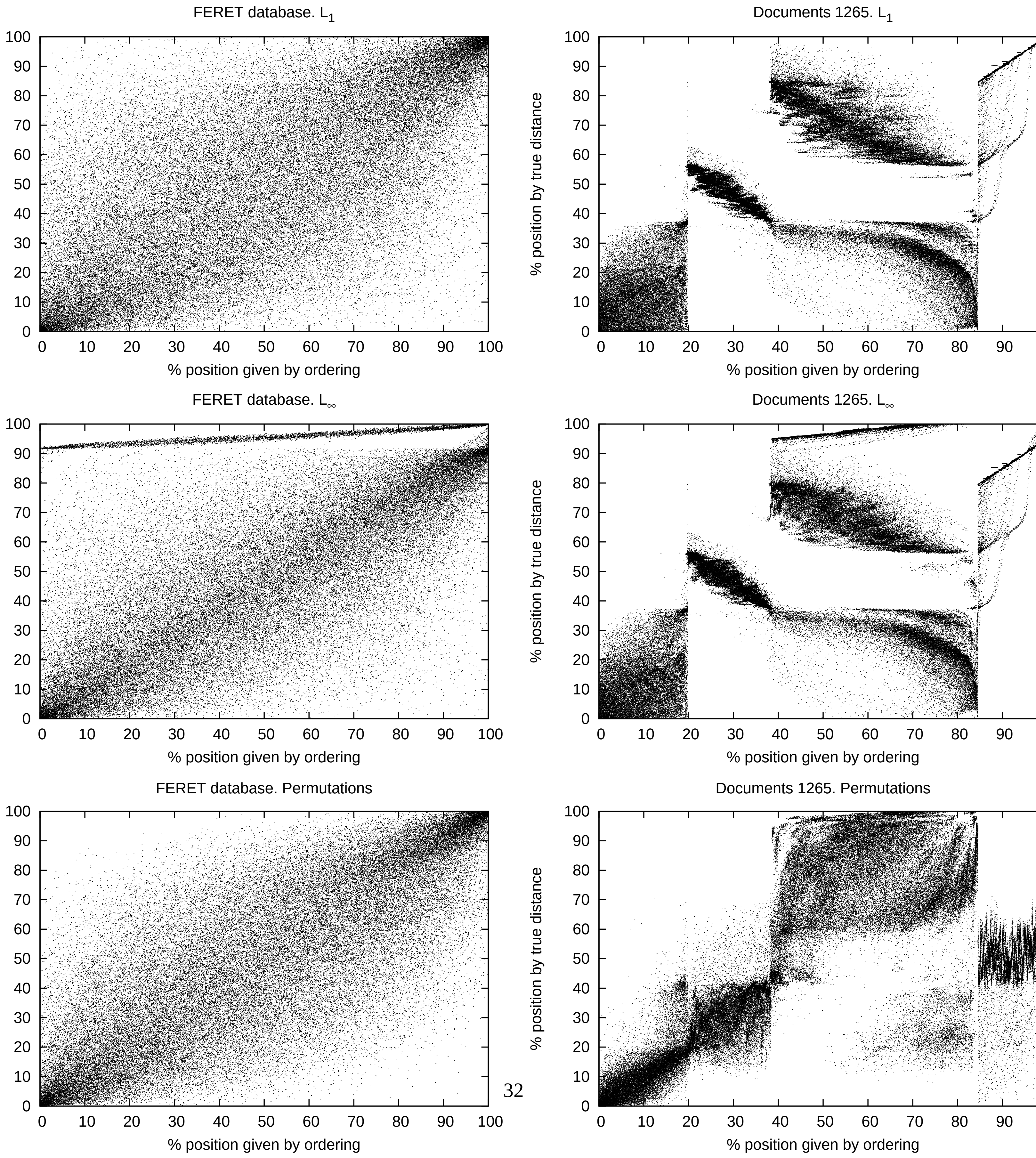


Fig. 14. Clouds of points for the faces database, using 64 pivots (left) and for the documents database, using 128 pivots (right). From top to bottom, L_1, L_∞, and permutations.

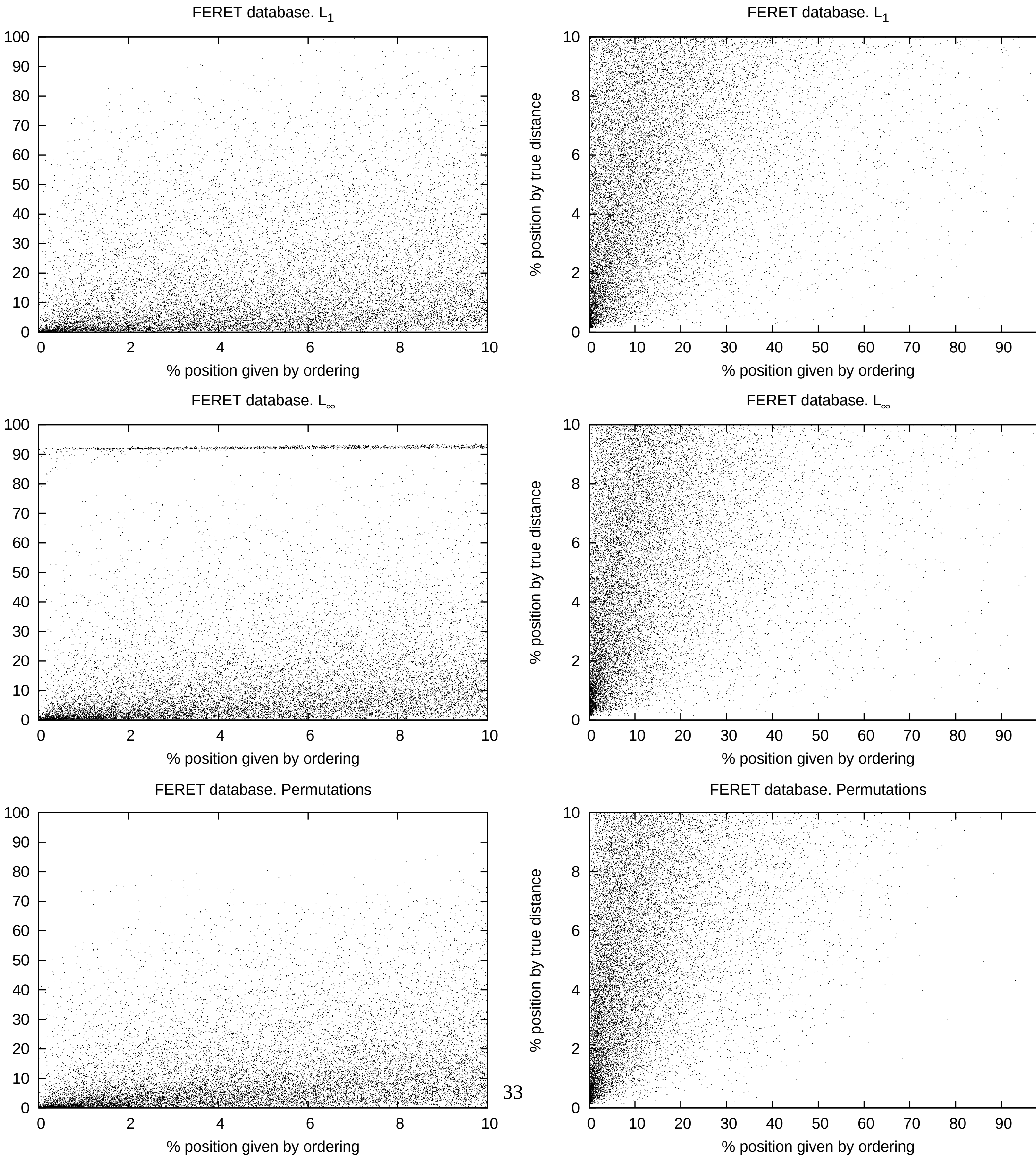


Fig. 15. Precision (left) and recall (right) zooms of the clouds of points for the faces database, using 64 pivots. From top to bottom, L_1, L_∞, and permutations.

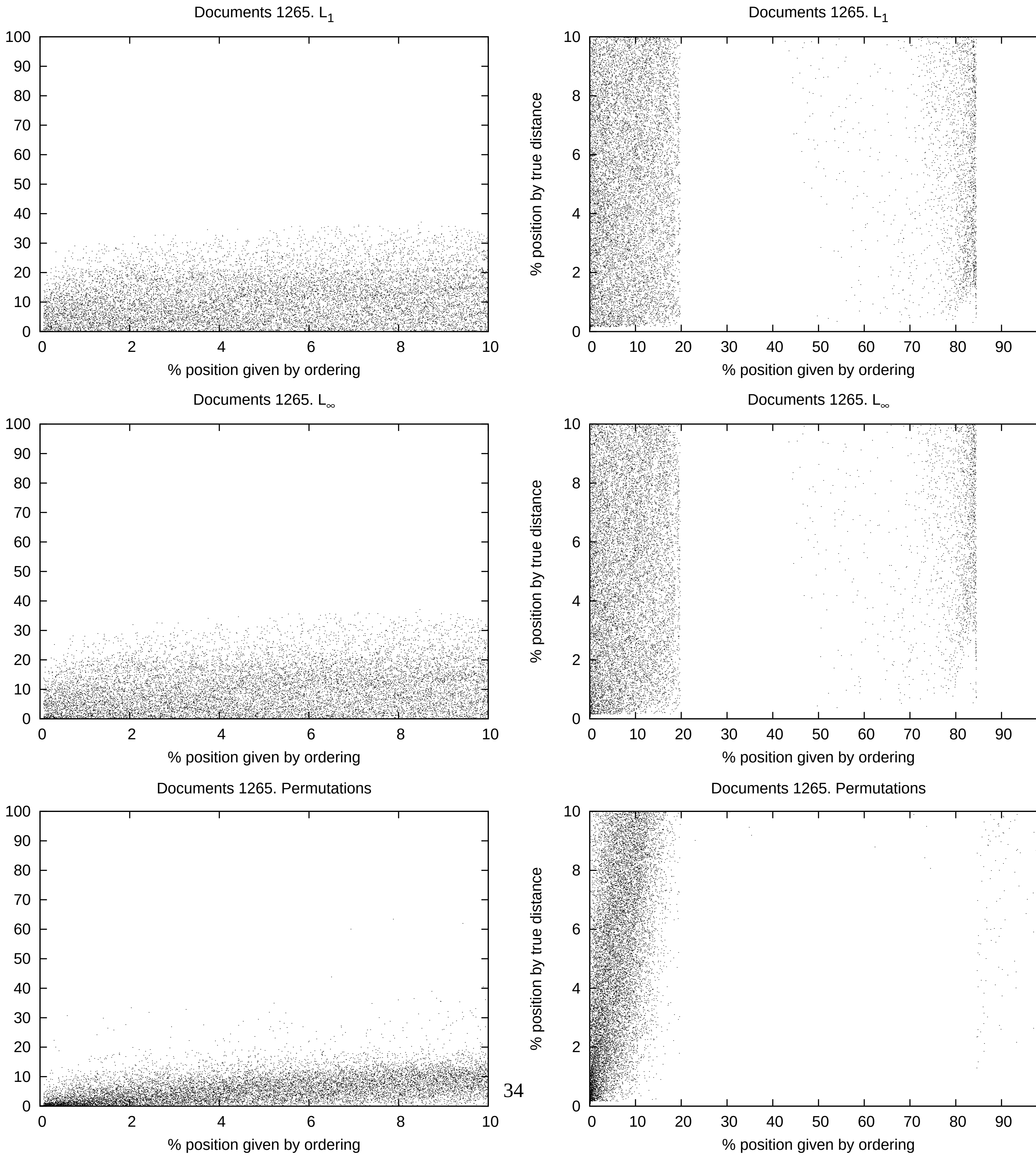


Fig. 16. Precision (left) and recall (right) zooms of the clouds of points for the documents database, using 128 pivots. From top to bottom, L_1, L_∞, and permutations.