Boston University Computer Science Tech. Report No. 2005-010, March 21, 2005.
To appear in Proceedings of ACM International Conference on Management of Data (SIGMOD), June 2005.

Query-Sensitive Embeddings

Vassilis Athitsos **Marios Hadjieleftheriou** **George Kollios** **Stan Sclaroff**

Computer Science Department
Boston University
111 Cummington Street
Boston, MA 02215, USA
{athitsos,marioh,gkollios,sclaroff}@cs.bu.edu

ABSTRACT

A common problem in many types of databases is retrieving the most similar matches to a query object. Finding those matches in a large database can be too slow to be practical, especially in domains where objects are compared using computationally expensive similarity (or distance) measures. This paper proposes a novel method for approximate nearest neighbor retrieval in such spaces. Our method is embedding-based, meaning that it constructs a function that maps objects into a real vector space. The mapping preserves a large amount of the proximity structure of the original space, and it can be used to rapidly obtain a short list of likely matches to the query. The main novelty of our method is that it constructs, together with the embedding, a query-sensitive distance measure that should be used when measuring distances in the vector space. The term "query-sensitive" means that the distance measure changes depending on the current query object. We report experiments with an image database of handwritten digits, and a time-series database. In both cases, the proposed method outperforms existing state-of-the-art embedding methods, meaning that it provides significantly better trade-offs between efficiency and retrieval accuracy.

1. INTRODUCTION

Many important applications require identifying, in a large database, the most similar matches to a query object. For example, a common way of estimating the properties of a biological sequence (like a protein, or DNA sequence) is by identifying its closest matches in a large database of known sequences. As another example, nearest neighbor classification is a widely used pattern recognition technique, in which we classify an object by assigning to it the class of its closest match in a database of training objects.

This work was supported by NSF grants IIS-0308213 and IIS-0133825, and by ONR grant N00014-03-1-0108.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.

Although numerous indexing methods have been proposed for speeding up nearest-neighbor retrieval [6], such methods typically assume that we are operating in a Euclidean space, or a metric space, or a so-called "coordinate space," where each object is represented as a feature vector of fixed dimensions. In many actual applications these assumptions are not obeyed, because we need to use distance measures that are non-Euclidean and even non-metric (meaning that even general indexing methods for metric spaces, like vp-trees and variants [8, 18, 38, 40] cannot be applied), and because the objects are not of fixed dimensionality. Examples of computationally expensive non-Euclidean distance measures include the Kullback-Leibler distance for matching probability distributions, Dynamic Time Warping for matching time series, or the edit distance for matching strings and biological sequences. It is important to design efficient methods for nearest neighbor retrieval in such spaces.

This paper proposes a novel method for approximate nearest neighbor retrieval in such non-Euclidean spaces. Our method is embedding-based, meaning that it constructs a function that maps objects into a real vector space. This mapping preserves a large amount of the proximity structure of the original space, meaning that nearby objects tend to get mapped to nearby vectors. At the same time, measuring distances between vectors (using a weighted L_1 distance measure) can be orders-of-magnitude faster than comparing objects in the original space.

The main novelty of our method is that it constructs, together with the embedding, a "query-sensitive" distance measure that should be used when measuring distances in the vector space. The term "query-sensitive" means that the distance measure changes depending on the current query object. In particular, the weights used for the L_1 distance measure automatically adjust to each query. Using a query-sensitive measure is a natural way to capture the fact that, as described in Sec. 4, given a query object, some coordinates of the embedding are more informative than other coordinates. In general, query-sensitive distance measures provide a solution to an important issue that arises when objects are represented as high-dimensional vectors: the need to identify, for any two objects, the coordinates that are really important for comparing those objects [1].

Our formulation uses a recent technique for constructing embeddings using machine learning, that was introduced in the BoostMap embedding algorithm [2]. The key novelty of the proposed method is that our algorithm produces an embedding *and* a query-sensitive distance measure, with a well-defined mechanism for adjusting the distance measure to each query object. Existing embedding methods, including the original BoostMap algorithm, produce a global distance measure. In the datasets we have experimented with, a query-sensitive distance measure leads to much better retrieval performance.

A secondary contribution in this paper is a method for choosing training data for the learning algorithm. The original BoostMap algorithm uses as training data random triples of objects from the original space. We propose a more selective method for choosing training data, that leads to embeddings that are better optimized for retrieval accuracy. The method we propose is very simple, but it leads to significant improvement in the experimental results.

The proposed method is experimentally compared to the original BoostMap algorithm, as well as to FastMap [12], which is a well-known existing embedding method. Experiments are performed on two datasets: the MNIST database of handwritten digits [22], with Shape Context Distance [4] as the underlying distance measure, and a time-series database [32] with constrained Dynamic Time Warping as the underlying distance measure. In both datasets, the algorithm described in this paper yields superior performance with respect to both the original BoostMap method and FastMap. For a fixed budget of exact distance computations per query, and for different integers k, the new method correctly retrieves all k nearest neighbors for a significantly higher fraction of queries.

2. RELATED WORK

Various methods have been employed for similarity indexing in multi-dimensional datasets, including hashing and tree structures [6, 8, 18, 36, 38]. However, the performance of such methods degrades in high dimensions. This phenomenon is one of the many aspects of the "curse of dimensionality". Another problem with tree-based methods is that they typically rely on Euclidean or metric properties, and those properties do not hold in non-metric spaces.

Approximate nearest neighbor methods have been proposed in [17] and scale better with the number of dimensions. However, those methods are available only for specific sets of metrics, and they are not applicable to arbitrary distance measures. In [13], a randomized procedure is used to create a locality sensitive hashing structure that can report a $(1+\epsilon)$-approximate nearest neighbor with a constant probability. In [40] M-trees are used for approximate similarity retrieval, while [23] proposes clustering the dataset and retrieving only a small number of clusters (which are stored sequentially on disk) to answer each query. In [9, 11, 19] dimensionality reduction techniques are used where lower-bounding rules are ignored when dismissing dimensions and the focus is only on preserving close approximations of distances. In [34] the authors used VA-files [35] to find nearest neighbors by omitting the refinement step of the original exact search algorithm and estimating approximate distances using only the lower and upper bounds computed by the filtering step. Finally, in [30] the authors partition the data space into clusters and then the representatives of each cluster are compressed using quantization techniques. Other similar approaches include [21, 26]. However, all these techniques can be employed mostly for distance functions defined using L_p norms.

Various techniques appeared in the literature for robust evaluation of similarity queries on time-series databases when using non-metric distance functions [20, 32, 37]. These techniques use the filter-and-refine approach, where a computationally efficient approximation of the original distance is utilized in the filtering step. Query speedup is achieved by pruning a large part of the search space at the filter step. Then, the original, accurate but more expensive distance measure is applied to the few remaining candidates, during the refinement step. Usually, the distance approximation function is designed to be metric (even if the original distance is not), so that traditional indexing techniques can be applied to index the database in order to speed up the filtering stage as well. In our experimental evaluation we compare our approach with the technique presented in [32].

Also related to our setting is work on distance-based indexing for string similarity. In [25] special modifications to distance-based indices [8, 18, 38] are proposed for indexing distance functions that are *almost* metric. However, unlike our method, the technique of [25] cannot be applied to general distance functions.

In domains where the distance measure is computationally expensive, significant computational savings can be obtained by constructing a distance-approximating embedding, which maps objects into another space with a more efficient distance measure. A number of methods have been proposed for embedding arbitrary spaces into a Euclidean or pseudo-Euclidean space [2, 7, 12, 16, 24, 28, 33, 39]. Some of these methods, in particular MDS [39], Bourgain embeddings [7, 15], LLE [24] and Isomap [28] are not targeted at speeding up online similarity retrieval, because they still need to evaluate exact distances between the query and most or all database objects. Online queries can be efficiently handled by Lipschitz embeddings [15], FastMap [12], MetricMap [33], SparseMap [16], and BoostMap [2].

Embedding methods designed for speeding up nearest neighbor retrieval [2, 12, 15, 16, 33] have two attractive properties: first, they can compute the embedding of a new query object by comparing that object to a relatively small subset of all database objects; second, they are formulated in a domain-independent way, and they can be applied to any space and distance measure (unlike techniques like [13, 25], for example, whose formulation cannot handle arbitrary spaces). At the same time, when applied to arbitrary spaces, there is no guarantee that these methods will attain some acceptable tradeoff between accuracy and efficiency.

The method proposed in this paper belongs to the same family of approaches as [2, 12, 15, 16, 33]; it tries to solve the same problem (efficient nearest neighbor retrieval), and it can be applied to arbitrary spaces and distance measures. The proposed method can be seen as an extension of BoostMap [2]. The main advantage of BoostMap is that it optimizes a measure of embedding quality that is directly related to how well the embedding preserves the similarity structure of the original space. A secondary contribution of this paper is that it shows how to reformulate this measure of embedding quality so that it is more tightly related to the task of nearest neighbor retrieval. The main contribution consists of showing how to extend the BoostMap algorithm so that it produces, together with the embedding, a query-sensitive distance measure. Given a query object, the query-sensitive distance measure assigns higher weights to the embedding coordinates that are important for that query.

Query-sensitive distance measures have been used in [10, 14] to improve the classification accuracy of nearest neighbor classifiers. In these methods, it is assumed that an initial global (query-insensitive) distance measure is available. Given a query object, the initial distance measure is iteratively refined. In contrast, in this paper we formulate a method that constructs an embedding and a query-sensitive distance measure for speeding up nearest neighbor retrieval. Given a query object, the query-sensitive distance measure is constructed in a non-iterative way, and no initial distance measure is given to our algorithm.

3. BACKGROUND

We use X to denote a set of objects, and $D_X(x_1, x_2)$ to denote a distance measure between objects $x_1, x_2 \in X$. For example, X can be a set of images of handwritten digits (Fig. 3), and D_X can be shape context matching as defined in [5]. However, any X and D_X can be plugged into the formulations described in this paper.

First, we will define some simple embeddings, and then we will briefly describe the association between embeddings and classifiers that was introduced in [2].

3.1 Some Simple Embeddings

An embedding $F : X \to \mathbb{R}^d$ is a function that maps any object $x \in X$ into a d-dimensional vector $F(x) \in \mathbb{R}^d$. Distances in $\mathbb{R}^d$ are measured using the Euclidean (L_2) metric, or some other L_p metric. It is assumed that measuring a single L_p distance between two vectors is significantly faster than measuring a single distance D_X between two objects of X. This assumption is obeyed in the example datasets we used in our experiments. For example, with our PC we can measure close to a million L_1 distances between high-dimensional vectors in $\mathbb{R}^{100}$ in one second, whereas only 15 shape context distances can be evaluated per second.

A simple way to define one-dimensional (1D) embeddings is using prototypes [15]. In particular, given an object $r \in X$, we can define an embedding $F^r : X \to \mathbb{R}$ as follows:

$$F^r(x) = D_X(x, r) . \tag{1}$$

The prototype r that is used to define F^r is typically called a *reference object* or a *vantage object* [15]. The intuition behind embeddings of type F^r is simple: if two objects x_1 and x_2 are very similar to each other, we expect their distances to r, i.e., $D_X(x_1, r)$ and $D_X(x_2, r)$ to also be similar. Therefore, F^r is expected to map similar objects to nearby points on the real line.

Another family of simple, 1D embeddings is proposed in [12] and used as building blocks for FastMap. The idea is to choose two objects $x_1, x_2 \in X$, called pivot objects, and then, given an arbitrary $x \in X$, to define the embedding F^{x_1,x_2} of x to be the *projection* of x onto the "line" $\overline{x_1 x_2}$:

$$F^{x_1,x_2}(x) = \frac{D_X(x, x_1)^2 + D_X(x_1, x_2)^2 - D_X(x, x_2)^2}{2D_X(x_1, x_2)} . \tag{2}$$

The reader can find in [12] an intuitive geometric interpretation of this equation, based on the Pythagorean theorem.

These simple 1D embeddings can be used as building blocks for constructing higher-dimensional embeddings. Embeddings of type F^{x_1,x_2} are used to construct FastMap [12]. Embeddings of type F^r can be combined to form *Lipschitz* embeddings [15].

3.2 Associating Embeddings with Classifiers

Let $F : X \to \mathbb{R}^d$ be a d-dimensional embedding. F acts as a classifier for the following binary classification problem: given three objects $q, a, b \in X$, is q closer to a or to b? If we know F, but we do not know the exact distances $D_X(q, a)$ and $D_X(q, b)$, we can provide an answer by simply checking if $F(q)$ is closer to $F(a)$ or to $F(b)$. If that answer is wrong, we say that embedding F *fails* on triple (q, a, b). If the answer is correct, we say that embedding F *succeeds* on triple (q, a, b). If F succeeds on all triples, then F can be used to correctly identify the true nearest neighbors for all queries.

Simple, 1D embeddings, like the ones we defined above, are expected to act as *weak classifiers* [2, 27], i.e., they will probably have a high error rate, but at the same time they should provide answers that are, on average, more accurate than a random guess, which would have an error rate of 50%. In other words, we expect that a 1D embedding will fail on many triples (q, a, b), but it will succeed on more than half of all possible triples.

The key insight in [2] is that, by associating embeddings with classifiers, we can reduce the problem of embedding construction to the problem of combining many weak classifiers into a strong classifier. The latter problem of combining weak classifiers has been extensively studied in the machine learning community, and a well-known and widely used solution to that problem is the AdaBoost algorithm [27]. The BoostMap algorithm [2] essentially uses AdaBoost to construct a high-dimensional embedding out of 1D embeddings of type F^r and F^{x_1,x_2}. The BoostMap algorithm uses a training set S of triples (q, a, b), picked randomly from the available training objects, with the constraint that q is closer to a than to b. The algorithm constructs an embedding $F : X \to \mathbb{R}^d$ in a way that minimizes the fraction of triples $(q, a, b) \in S$ on which the embedding F fails, i.e., triples (q, a, b) for which F maps q closer to b than to a.

4. MOTIVATION FOR QUERY-SENSITIVE DISTANCE MEASURES

In the experiments reported in [2], it is shown that it is often beneficial to generate, using BoostMap, a high-dimensional embedding, with over 100 dimensions. As pointed out in [1], finding nearest neighbors in a high-dimensional space raises the following issues:

- **Lack of contrasting:** Two high-dimensional objects are unlikely to be very similar in all the dimensions.
- **Statistical sensitivity:** The data is rarely uniformly distributed, and for a pair of objects there may be only relatively few coordinates that are statistically significant for comparing those objects.

Figure 1 illustrates the problem of statistical sensitivity. In that toy example, we define a three-dimensional embedding of the 2D plane using three reference objects. For some query objects, sometimes a single coordinate is sufficient for getting near-perfect retrieval results. In particular, if for a given query object q there is a reference object r really close to q, then using the 1D embedding F^r by itself might give more accurate results than using the high-dimensional embedding. Figure 1 does not illustrate the problem of lack of contrasting, but that problem can also be present if the original distance measure D_X is not metric: then, it is possible

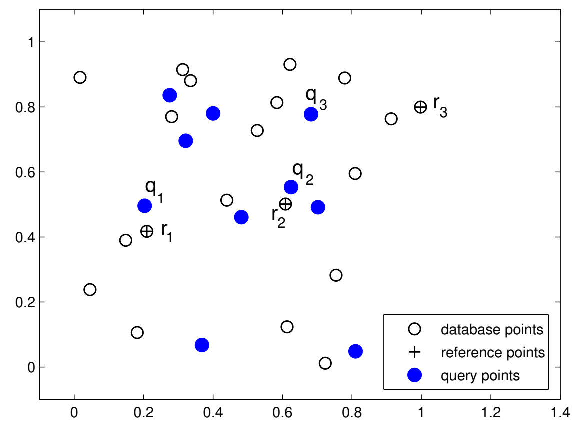

Figure 1: A toy example illustrating the use of query-sensitive embeddings. Our space is the set of points in the unit square $[0,1] \times [0,1]$. There are twenty database objects, three of which (indicated as r_1, r_2, r_3) are selected as reference objects. Using these reference objects, we define embedding $F(x) = (F^{r_1}(x), F^{r_2}(x), F^{r_3}(x))$, and we use the L_1 distance to compare the embeddings of two objects. There are ten query objects, three of which are marked as q_1, q_2, q_3. F fails on 23.5% of the 3800 triples (q, a, b) we can form by picking q from the query objects, and the pair a, b from the database objects. In contrast, the 1D embeddings $F^{r_1}, F^{r_2}, F^{r_3}$ fail respectively on $39.2\%, 36.4\%$, and 26.6% of the triples. However, if we restrict our attention to triples (q, a, b) where $q = q_1$, F^{r_1} does better than F: F^{r_1} fails on 5.8% of those triples, whereas F fails on 11.6% of those triples. Similarly, for $q = q_2$ and $q = q_3$ respectively, F^{r_2} and F^{r_3} are more accurate than F. Therefore, for query objects q_1, q_2, q_3, it would be beneficial to use a query-sensitive weighted L_1 measure, that would respectively use only the first, second, and third coordinate of F.

for two objects x_1 and x_2 to be very close to each other, but have very different distances $D_X(x_1, r)$ and $D_X(x_2, r)$ to a reference object r.

To address these problems, we propose to construct, together with the embedding, a *query-sensitive* distance measure. By "query-sensitive" we mean that the weights used for the weighted L_1 distance (used to measure distances between embeddings of objects) will not be fixed; instead, they will depend on the query object. Figure 1 illustrates how using a query-sensitive distance measure can give better retrieval accuracy. Overall, a query-sensitive distance measure provides a principled way to address the problems described in [1], by putting more emphasis on coordinates that are more important for a particular query.

5. CONSTRUCTING AN EMBEDDING AND A QUERY-SENSITIVE DISTANCE MEASURE

At a high level, our method constructs an embedding using the following steps:

1. We start by specifying a large family of 1D embeddings, using well-known definitions from prior embedding methods.

2. We use 1D embeddings to define binary classifiers, which estimate for object triples (q, a, b) if q is closer to a or to b. These classifiers are expected to be pretty inaccurate, but still better than a random classifier (which would just guess randomly all the time, and therefore would have a 50% error rate).

3. We run AdaBoost to combine many classifiers into a single classifier H, which we expect to be significantly more accurate than the simple classifiers associated with 1D embeddings.

4. We use H to define a d-dimensional embedding F_{out}, and a query-sensitive weighted L_1 distance measure D_{out}. It is shown that H is mathematically equivalent to the combination of F_{out} and D_{out}: if, for three objects q, a, $b \in X$, H predicts that q is closer to a than it is to b, then, under distance measure D_{out}, $F_{\text{out}}(q)$ is closer to $F_{\text{out}}(a)$ than it is to $F_{\text{out}}(b)$.

The main difference between the proposed approach and the original BoostMap algorithm is introduced in the second step. Using 1D embeddings, we will define binary classifiers of a different type than what was used in [2]. We will then show how using those classifiers as building blocks results in constructing a query-sensitive distance measure. These steps are explained in detail in the remainder of this section.

5.1 Defining Query-Sensitive Classifiers from 1D Embeddings

As described earlier, every embedding F corresponds to a classifier that classifies triples (q, a, b) of objects in X. Formally, we can say that a triple (q, a, b) is of type 1 if q is closer to a than to b, type 0 if q is equally close to a and b, and type -1 if q is closer to b than to a. Given embedding F, and a distance measure D for comparing vectors, we can define the classifier $\tilde{F}$ associated with embedding F as follows:

$$\tilde{F}(q, a, b) = D(F(q), F(b)) - D(F(q), F(a)) \ . \qquad (3)$$

The sign of $\tilde{F}(q, a, b)$ is an estimate of whether triple (q, a, b) is of type 1, 0, or -1. We should note that, if F is a 1D embedding, then $\tilde{F}(q, a, b) = |F(q) - F(b)| - |F(q) - F(a)|$.

Sometimes, $\tilde{F}$ may do a really good job on triples (q, a, b) when q is in a specific region, but at the same time it may be beneficial to ignore $\tilde{F}$ when q is outside that region. For example, suppose that we have an embedding F^r defined using reference object r. If $q = r$, then $\tilde{F}^r$ will classify correctly all triples (q, a, b), where a and b are any two objects of space X. If $q \neq r$, we still expect that, the closer q is to r, the more accurate $\tilde{F}^r$ will be on triples (q, a, b). Figure 1 illustrates such cases.

In [2], the weak classifiers that are used by AdaBoost are of type $\tilde{F}$, with F being a 1D embedding. We propose to use a different type of classifier, that can explicitly model the fact that any 1D embedding F is more useful in some regions of the space and less useful in other regions.

In particular, given a 1D embedding F, we need a function $S(q)$ (which we call a *splitter*), that will estimate, given a query q, whether classifier $\tilde{F}$ is useful or not. More formally,

Given: $(o_1, y_1), \ldots, (o_t, y_t)$; $o_i \in \mathcal{G}, y_i \in \{-1, 1\}$.
Initialize $w_{i,1} = \frac{1}{t}$, for $i = 1, \ldots, t$.
For $j = 1, \ldots, J$:

1. Train weak learner using training weights $w_{i,j}$.
2. Get weak classifier $h_j : \mathcal{G} \to \mathbb{R}$.
3. Choose $\alpha_j \in \mathbb{R}$.
4. Set training weights $w_{i,j+1}$ for the next round as follows:
$$w_{i,j+1} = \frac{w_{i,j} \exp(-\alpha_j y_i h_j(x_i))}{z_j} \ . \quad (6)$$
where z_j is a normalization factor (chosen so that $\sum_{i=1}^{t} w_{i,j+1} = 1$).

Output the final classifier:
$$H(x) = \sum_{j=1}^{J} \alpha_j h_j(x). \quad (7)$$

Figure 2: The AdaBoost algorithm. This description is largely copied from [27].

if X is the original space, we use the term *splitter* to denote any function mapping X to the binary set $\{0, 1\}$. We can readily define splitters using 1D embeddings. Given a 1D embedding $F : X \to \mathbb{R}$, and a subset $V \subset \mathbb{R}$, we can define a splitter $S_{F,V} : X \to \{0, 1\}$ as follows:
$$S_{F,V}(q) = \begin{cases} 1 & \text{if } F(q) \in V \ . \\ 0 & \text{otherwise} \ . \end{cases} \quad (4)$$

Now, suppose we have a subset $V \subset \mathbb{R}$ and a 1D embedding $F : X \to \mathbb{R}$. We define a *query-sensitive classifier* $\tilde{Q}_{F,V} : X^3 \to \mathbb{R}$, as follows:
$$\tilde{Q}_{F,V}(q, a, b) = S_{F,V}(q)\tilde{F}(q, a, b) \ . \quad (5)$$

At an intuitive level, $\tilde{F}$ is by itself a classifier of triples (q, a, b). $\tilde{Q}_{F,V}$ is a cropped version of $\tilde{F}$, that gives 0 (i.e., a neutral result) whenever $F(q) \notin V$. For example, if $F = F^r$ for some reference object r, and $V = [0, \tau]$ for some positive threshold τ, splitter $S_{F,V}(q)$ accepts object q if it is within distance τ of reference object r. Therefore, the query-sensitive classifier $\tilde{Q}_{F,V}$ will apply $\tilde{F}$ only if q is sufficiently close to r. By choosing τ in an appropriate way, we can capture the fact that $\tilde{F}$ should only be applied to objects within a specified distance from reference object r.

5.2 Overview of the Training Algorithm

The AdaBoost algorithm (taken, with minor modifications, from [27]) is shown in Figure 2. AdaBoost assumes that we have a "weak learner" module, which we can call at each round to obtain a new weak classifier. The goal is to construct a strong classifier that achieves much higher accuracy than the individual weak classifiers.

The AdaBoost algorithm simply determines the appropriate weight for each weak classifier, and then adjusts the training weights. The training weights are adjusted so that training objects that are misclassified by the chosen weak classifier h_j get more weight for the next round. Because of the training weights, the weak learner is biased towards returning a classifier that tends to correct mistakes of previously chosen classifiers. Overall, weak classifiers are chosen and weighted so that they complement each other. The ability of AdaBoost to construct highly accurate classifiers by combining many relatively inaccurate weak classifiers has been demonstrated in numerous applications (for example, in [29, 31]).

In our case, the AdaBoost algorithm is adapted to the problem of constructing an embedding and a query sensitive distance measure. We adapt AdaBoost to this problem as follows:

- Each training object o_i is a triple (q_i, a_i, b_i) of objects in X. Because of that, we refer to o_i not as a training object, but as a training *triple*. The set $\mathcal{G}$ from which training triples are picked can be the entire X^3 (the set of all triples we can form by objects from X), or a more restricted subset of X^3, as discussed in Sec. 6.
- The i-th training triple (q_i, a_i, b_i) is associated with a class label y_i, which is 1 if q_i is closer to a_i and -1 if q_i is closer to b_i.
- Each weak classifier h_j is a query-sensitive classifier $\tilde{Q}_{F,V}$, where F is a one-dimensional embedding and V is an interval of $\mathbb{R}$.

Also, we pass to AdaBoost some additional arguments:

- A set $C \subset X$ of candidate objects. Elements of C will be used as reference objects and pivot objects to define 1D embeddings of type F^r and F^{x_1,x_2}.
- A matrix of distances between any two objects in C, and a matrix of distances from each $c \in C$ to each q_i, a_i and b_i appearing in one of the training triples.

To fully specify the training algorithm, we need to specify what we do for steps 1, 2 and 3 of the algorithm shown in Figure 2. In simple terms, this is how those steps are implemented at each training round j:

- We construct a large set of classifiers $\tilde{Q}_{F,V}$ by choosing randomly different 1D embeddings F and different ranges $V \subset \mathbb{R}$.
- We choose, among that large set of classifiers, the one that is the "best" at the current round, and we assign a weight to that classifier, using a method suggested in [27]. Evaluating how good a classifier is at a particular training round is related to how well that classifier performs on a training set of triples of objects.

In the next few paragraphs we will discuss how each of those operations is done, i.e., how we construct a large set of weak classifiers at each training round, and how we choose the best one out of them.

5.3 Forming Weak Classifiers

The weak classifiers considered by AdaBoost are classifiers $\tilde{Q}_{F,V}$ as defined in Eq. 5, where F is some 1D embedding defined using reference objects or pivot objects from the set C of candidate objects. To pick a range V for $\tilde{Q}_{F,V}$, we simply compute the values $F(x)$ for every object appearing in a training triple (q_i, a_i, b_i), and set V to be a random interval of $\mathbb{R}$ containing some of those values. We form many such ranges V for each F, and for each range we measure the training error, i.e., the classification error of classifier $\tilde{Q}_{F,V}$, on the training triples. When we measure the training error, we weigh each training triple o_i by the current weight $w_{i,j}$ of that triple in training round j. Therefore, the error of $\tilde{Q}_{F,V}$ will be different at each training round.

At training round j we choose, randomly, a large number of 1D embeddings. For each selected 1D embedding F, we find the range $V_{F,j}$ that achieves the lowest training error at round j. The next classifier will be chosen among the classifiers $\tilde{Q}_{F,V_{F,j}}$.

Now we are ready to specify how to implement steps $1-3$ in Figure 2, for each training round $j = 1, \ldots, J$. Step 1 consists of evaluating each $\tilde{Q}_{F,V_{F,j}}$, so that we can choose the best weak classifier to add to the strong classifier that is being assembled. The function $Z_j(\tilde{Q}, \alpha)$ gives a measure of how useful it would be to choose $h_j = \tilde{Q}$ and $\alpha_j = \alpha$ at training round j:

$$Z_j(\tilde{Q}, \alpha) = \sum_{i=1}^{t} (w_{i,j} \exp(-\alpha y_i \tilde{Q}(q_i, a_i, b_i))) \ . \tag{8}$$

The full details of the significance of Z_j can be found in [27]. Here it suffices to say that if $Z_j(\tilde{Q}, \alpha) < 1$ then choosing $h_j = \tilde{Q}$ and $\alpha_j = \alpha$ is overall beneficial, and is expected to reduce the training error. Given the choice between two weighted classifiers αh and $\alpha' h'$, we should choose the weighted classifier that gives the lowest Z_j value. Given h_j, we should choose α_j to be the α that minimizes $Z_j(h_j, \alpha)$.

Based on the above considerations, in step 1 we find the optimal α for each weak classifier $\tilde{Q}_{F,V_{F,j}}$. Then, in steps 2 and 3 we set h_j and α_j respectively to be the weak classifier and weight that yielded the lowest overall value of Z_j.

5.4 Training Output: Embedding and Distance

The output of the training stage is a classifier H of the following form:

$$H = \sum_{j=1}^{J} \alpha_j \tilde{Q}_{F'_j, V_j} \ . \tag{9}$$

Each $\tilde{Q}_{F'_j, V_j}$ is associated with a 1D embedding F'_j. Classifier H has been trained to estimate, for triples of objects (q, a, b), if q is closer to a or to b. However, our goal is to actually construct not just a classifier of triples of objects, but an embedding. Here we discuss how to define such an embedding F_{out}, and an associated distance measure D_{out} to be used to compare vectors.

A particular 1D embedding F can be equal to multiple F'_j's occurring in the definition of classifier H. We construct the set $\mathbb{F}$ of all unique 1D embeddings used in H, as $\mathbb{F} = \bigcup_{j=1}^{J} \{F'_j\}$, and we denote the elements of $\mathbb{F}$ as $F_1, \ldots, F_d$.

The embedding $F_{\text{out}} : X \rightarrow \mathbb{R}^d$ is defined as $F_{\text{out}}(x) = (F_1(x), \ldots, F_d(x))$. Obviously, it is a d-dimensional embedding.

Before defining distance measure D_{out}, we first need to define an auxiliary function $A_i(q)$, which assigns a weight to the i-th coordinate, for $i = 1, \ldots, d$:

$$A_i(q) = \sum_{j:((j \in \{1,\ldots,J\}) \wedge (F_i = F'_j) \wedge (F_i(q) \in V_j))} \alpha_j \ . \tag{10}$$

In words, given object q, for coordinate i, we go through all weak classifiers $\tilde{Q}_{F'_j, V_j}$ that make up H. For each such classifier, we check if the splitter $S_{F'_j, V_j}$ accepts q (i.e., we check if $F'_j(q) \in V_j$), and we also check if $F'_j = F_i$. If those conditions are satisfied, we add the weight α_j to $A_i(q)$.

Let $F_{\text{out}}(q) = (q_1, ..., q_d)$, and let x be some other object in X, with $F_{\text{out}}(x) = (x_1, ..., x_d)$. We define distance D_{out} as follows:

$$D_{\text{out}}((q_1, ..., q_d), (x_1, ..., x_d)) = \sum_{i=1}^{d} (A_i(q) |q_i - x_i|) \ . \tag{11}$$

$D_{\text{out}}(v_1, v_2)$ (where v_1, v_2 are d-dimensional vectors) is like a weighted L_1 measure on $\mathbb{R}^d$, but the weights depend on v_1. Therefore $D_{\text{out}}(v_1, v_2)$ is not symmetric, and not a metric. We say that $D_{\text{out}}(v_1, v_2)$ is a *query-sensitive* distance measure, since v_1 is typically the embedding of a query, and v_2 is the embedding of a database object that we want to compare to the query.

It is important to note that the way we defined F_{out} and D_{out}, if we apply Eq. 3 to obtain a classifier $\tilde{F}_{\text{out}}$ from F_{out} (with D set to D_{out}), then $\tilde{F}_{\text{out}} = H$. In words, the classifier corresponding to embedding F_{out} is equal to the output of AdaBoost. Here are the main steps of the proof:

PROPOSITION 1. $\tilde{F}_{\text{out}} = H$.

Proof:

$$\begin{aligned}
&\tilde{F}_{\text{out}}(q, a, b) = \\
&D_{\text{out}}(F_{\text{out}}(q), F_{\text{out}}(b)) - D_{\text{out}}(F_{\text{out}}(q), F_{\text{out}}(a)) = \\
&\sum_{i=1}^{d} (A_i(q)|F_i(q) - F_i(b)| - A_i(q)|F_i(q) - F_i(a)|) = \\
&\sum_{i=1}^{d} (A_i(q)(|F_i(q) - F_i(b)| - |F_i(q) - F_i(a)|)) = \\
&\sum_{j=1}^{J} (\alpha_j S_{F'_j, V_j}(q)(|F'_j(q) - F'_j(b)| - |F'_j(q) - F'_j(a)|)) = \\
&\sum_{j=1}^{J} (\alpha_j S_{F'_j, V_j}(q) \tilde{F}'_j(q, a, b)) = \\
&\sum_{j=1}^{J} (\alpha_j \tilde{Q}_{F'_j, V_j}(q, a, b)) = H(q, a, b) \ . \qquad \square
\end{aligned}$$

This equivalence is important, because it shows that the quantity optimized by the training algorithm (i.e., classification error on triples of objects) is not only a property of the classifier H constructed by AdaBoost, but it is also a property of the embedding F_{out}, when coupled with distance measure D_{out}. We should emphasize that this equivalence between classifier H and embedding F_{out} relies on the way we define D_{out}. If, for example we had defined D_{out} as a Euclidean (L_2) distance, or as a query-insensitive L_1 distance, then the equivalence would no longer hold.

6. CHOOSING TRAINING TRIPLES

In the original BoostMap algorithm [2], training triples are chosen at random. By using a random training set of triples, BoostMap tries to preserve the entire similarity structure of the original space X. This means that the resulting embedding is equally optimized for nearest neighbor queries, farthest neighbor queries, or median neighbor queries. In cases where we only care about nearest neighbor queries, we would actually prefer an embedding that gave more accurate results for such queries, even if such an embedding did not preserve other aspects of the similarity structure of X, like farthest-neighbor information.

If we want to construct an embedding for the purpose of answering nearest neighbor queries, then we can construct training triples in a more selective manner. The main idea is that, given a training object q_i, the types of triples (q_i, a_i, b_i) that are related to k-nearest neighbor retrieval accuracy are triples in which a_i is one of the k nearest neighbors of q_i, and b_i is *not* one of the k nearest neighbors of q_i. As long as the embedding does not fail on such triples, the embedding will correctly identify the set of k nearest neighbors of q_i.

Based on the above considerations, given a parameter k_1 and given a set X_{tr} of training objects (typically X_{tr} is a subset of the set of database objects) we propose the following heuristic for choosing the i-th training triple (q_i, a_i, b_i):

1. Choose a random training object $q_i \in X_{\text{tr}}$.
2. Choose a random integer k' in $1, \ldots, k_1$.
3. Choose a_i to be the k'-nearest neighbor of q_i in X_{tr}.
4. Reset k' to a random integer between $k_1 + 1$ and $|X_{\text{tr}}|$.
5. Choose b_i to be the k'-nearest neighbor of q_i in X_{tr}.

The value of parameter k_1 should be based on the maximum number $k_{\max}$ of nearest neighbors that we may want to retrieve for an object. For example, if we want to retrieve up to 50 nearest neighbors per query ($k_{\max} = 50$), and if X_{tr} contains about one tenth of the database, then we should set $k_1 = 5$, so that for every q_i the corresponding a_i is likely to be one of the 50 nearest neighbors of q_i.

By choosing training triples this way, the training algorithm concentrates on building an embedding that, for any query object q, tends to map q closer to q's $k_{\max}$ nearest neighbors than to objects not included in q's $k_{\max}$ nearest neighbors. In practice, essentially the algorithm focuses on training triples (q, a, b) such that a is one of the nearest neighbors of q, that we would like to retrieve, and b is an object that is so far from q that we explicitly do *not* want to retrieve it as a match for q. An embedding that, given q, fails on many such triples (q, a, b), will fail to preserve the fact that a is one of the nearest neighbors of q. By using such triples for training, the learning algorithm will try to minimize the frequency with which the output embedding fails on such triples.

7. COMPLEXITY

At each training round we evaluate a number of weak classifiers by measuring their performance on t training triples, in order to choose the best weak classifier. If m weak classifiers are evaluated at each round, the computational time per training round is $O(mt)$. In contrast, FastMap [12], SparseMap [16], and MetricMap [33] do not require training at all.

Before we even start the training algorithm, we need to compute distances D_X from every object in C (the set of objects that we use to form 1D embeddings) to every object in C and to every object in X_{tr} (the set of objects from which we form training triples). We also need all distances between pairs of objects in X_{tr}. Computing all those distances can sometimes be the most computationally expensive part of the algorithm, depending on the complexity of computing D_X.

If time and memory resources are not limited, then we can set both C and X_{tr} equal to the entire database. Otherwise, we need to create C and X_{tr} by sampling randomly from the database. If (as in our experiments) C and X_{tr} have an equal number of elements, then the number of distances that we need to precompute is quadratic to $|C|$. In the experiments we report some results using relatively small values for $|C|$. We will see that, although larger values of $|C|$ clearly improve embedding quality, we can get reasonable results (better than, say, using FastMap) even with a small $|C|$, thus keeping the number of precomputed distances manageable.

We should emphasize that both the cost of precomputing distances and the cost of the training algorithm are *one-time preprocessing costs*. In many applications, spending the extra hours or days needed for this type of preprocessing is an acceptable cost, as long as it results in a higher-quality embedding, i.e., an embedding that leads to faster retrieval without sacrificing retrieval accuracy.

With respect to the *online retrieval cost*, computing the d-dimensional embedding of a query object takes $O(d)$ time and requires $O(d)$ evaluations of D_X. Comparing the embedding of the query to the embeddings of n database objects takes time $O(dn)$. For a fixed d, these costs are similar to those of FastMap [12], SparseMap [16], and MetricMap [33].

Compared to the original BoostMap algorithm, the proposed method has similar complexity both for the preprocessing steps and the online retrieval.

7.1 Dynamic Datasets

In our discussion so far we have assumed that the database is static. In some applications, however, we may need to add or remove objects online. As long as the underlying distribution of database objects is not altered, adding and removing objects is pretty straightforward. When adding an object x we need to compute its embedding $F_{\text{out}}(x)$. If F_{out} is d-dimensional, computing $F_{\text{out}}(x)$ requires computing at most $2d$ distances D_X between x and database objects.

If the underlying distribution of database objects changes significantly because of additions and removals, we may have to create a new embedding. A way to check whether the distribution of database objects has changed significantly is by measuring, at regular intervals, the error of the current embedding F_{out}, i.e., the classification error of $\tilde{F}_{\text{out}}$ on triples of objects picked (from the current database distribution) the same way we would choose training triples. When that error increases above some threshold, we can reuse our algorithm to construct a new embedding.

8. EMBEDDING APPLICATION: FILTER-AND-REFINE RETRIEVAL

In applications where we are interested in retrieving the k nearest neighbors for a query object q, a d-dimensional embedding F can be used in a filter-and-refine framework [15], as follows: first, we perform an offline preprocessing step, in which we compute and store vector $F(x)$ for every database object x. Then, given a previously unseen query object q, we perform the following three steps:

- Embedding step: compute $F(q)$, by measuring the

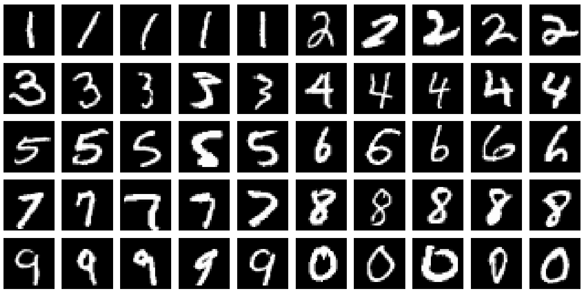

Figure 3: Some examples from the MNIST database of images of handwritten digits.

distances between q and the reference objects and/or pivot objects used to define F.

- Filter step: Find the database objects whose associated vectors are the p most similar vectors to $F(q)$.

- Refine step: sort those p candidates by evaluating the exact distance D_X between q and each candidate.

The assumption is that distance measure D_X is computationally expensive and evaluating distances between vectors is much faster. The filter step discards most database objects by measuring distances between vectors. The refine step applies D_X only to the top p candidates. This is much more efficient than brute-force retrieval, in which we compute D_X between q and the entire database.

To optimize filter-and-refine retrieval, we have to choose p, and often we also need to choose d, which is the dimensionality of the embedding. As p increases, we are more likely to include the true k nearest neighbors in the top p candidates found at the filter step, but we also need to evaluate more distances D_X at the refine step. Overall, we trade accuracy for efficiency. Similarly, as d (the dimensionality of the embedding) increases, computing the embedding for the query object becomes more expensive, but we may also get more accurate results in the filter step (since each additional dimension has been added by the training algorithm in order to improve the classification error of the embedding), and thus we may be able to decrease p. The best choice of p and d will depend on domain-specific parameters like k (i.e., how many of the nearest neighbors of an object we want to retrieve), the time it takes to compute the distance D_X, the time it takes to compare d-dimensional vectors, and the desired retrieval accuracy (i.e., how often we are willing to miss some of the true k nearest neighbors).

We should also note that, as d increases, the filter step also becomes more expensive, because we need to compare vectors of increasingly high dimensionality. However, in our experiments so far, with embeddings of up to 1,000 dimensions, the filter step always takes negligible time; retrieval time is dominated by the few exact distance computations we need to perform at the embedding step and the refine step.

In cases (not encountered in our experiments) when the filter step takes up a significant part of retrieval time, one can apply indexing techniques [6, 17, 36] to speed up filtering. We should keep in mind that in the filter step we are finding nearest neighbors in a real vector space, and many indexing methods are applicable in such a setting. One of the advantages of using embeddings is exactly the fact that we map arbitrary spaces to well-known real vector spaces, for which many tools are available.

9. EXPERIMENTS

We compared the proposed method to the original BoostMap method [2] and FastMap [12]. We used two different datasets: the MNIST dataset of handwritten digits [22], with the Shape Context Distance [4] as the exact distance measure, and a time series database [32] with constrained Dynamic Time Warping [32] as the exact distance measure.

The MNIST dataset contains images of isolated handwritten digits (numbers from 0 to 9). It consists of a training set of 60,000 images, which we used as the database, and a test set (disjoint from the training set) of 10,000 images that we used as query objects. The subjects who produced the test images were not used in producing the training images. The Shape Context Distance is introduced in [4]. To compute that distance, 100 shape context features are extracted from each image. Two images are aligned by doing bipartite matching between their features (which involves the computationally expensive Hungarian algorithm). The final distance is a weighted sum of three terms: the cost of matching shape context features, the cost of the alignment, and the intensity-level differences between image subwindows centered at matching feature locations. A 3-nearest-neighbor classifier using Shape Context matching gave state-of-the-art classification accuracy on the MNIST database, with an error rate of only 0.63%.

The second dataset that we tried was the time-series dataset used in [32]. To generate that dataset, various real datasets were used as seeds for generating a large number of time-series that are variations of the original sequences. Multiple copies of every real sequence were constructed by incorporating small variations in the original patterns as well as additions of random compression and decompression in time. The final dataset contains a "database" set of 32,768 sequences, and a "query" set of 50 sequences. Sequences are multi-dimensional, with an average size of 500 points each. The series were normalized by subtracting the average value in each dimension. Exact distances were measured using constrained Dynamic Time Warping, with a warping length $\delta = 10\%$ of the total length of the shortest sequence under comparison as described in [32].

To compare different embedding methods, we used each of those methods to build embeddings of various dimensions (the dimensionality ranged from 1 to 600). Then, for each embedding method, for each k and accuracy percentage B, we found the optimal parameters (i.e., number of dimensions of the embedding, and parameter p specifying the number of database objects to be retained after the filter step) under which we would successfully retrieve *all* k true nearest neighbors for a percentage of query objects equal to B, while minimizing the total number of exact distance computations per query object.

For the time series dataset, we performed an initial evaluation on the 50 queries used as a test set in [32]. Our method achieved a speed-up factor of 51.2, using filter-and-refine retrieval, with a 150-dimensional embedding and with parameter p set to 443. With those settings, the true nearest neighbor was retrieved correctly for each of the 50 queries. The indexing method in [32] reports a speed-up of approximately a factor of 5, while retrieving correctly the true nearest neighbor for all 50 queries, and measured on the same set of 50 queries that we used.

However, to get a clearer picture of performance, we decided to use a larger set of queries. To achieve that, we merged the query set and the database, and from the merged set we chose (randomly) a new set of 1,000 queries, with the remaining 31,818 objects used as the database for those queries. We found that performance on the new set of queries was not as good as on the initial set of 50 queries; on the new set of queries, a speedup factor of 50 was obtained only if we allowed the true nearest neighbor to be missed for 10% of the query objects. At the same time, even on the new set of queries, our method achieved significant speedups for k-nearest neighbor retrieval with different accuracy percentages and different values of k. The results reported in the remainder of this section for the time series dataset are with respect to the set of 1,000 queries.

In Figures 4 and 5, and in Table 1, we compare the proposed method (denoted as Se-QS) to the original BoostMap method (denoted as Ra-QI) and FastMap. We also show results for two intermediate methods, which incorporate only one of our two modifications to the original BoostMap algorithm. To denote each method, and its relation to the other methods, we use the following abbreviations:

Ra: Training triples are chosen entirely randomly from the set of all possible triples, as in the original BoostMap method.

Se: Training triples are chosen selectively, from a restricted set of possible triples, using the method we propose in Sec. 6.

QI: A query-insensitive distance measure D_{out} is used at the filter step, as in the original BoostMap method.

QS: A query-sensitive distance measure D_{out} is used at the filter step, as proposed in this paper.

Based on these abbreviations, Ra-QI denotes the original BoostMap algorithm, and Se-QS denotes the modified BoostMap algorithm we propose in this paper. Ra-QS and Se-QI add to the original BoostMap only one of the two changes that we propose in this paper (either the method for building a query-sensitive distance measure or the method for choosing training triples).

The optimal number of exact distance computations (i.e., corresponding to optimal settings for the dimensionality of the embedding and the parameter p) is shown for different values of k, from 1 to 50, and different percentages of accuracy (i.e., 90%, 95%, and 99%), in Figure 4 for the MNIST dataset and Figure 5 for the time series dataset. To avoid cluttering the figures, we omit from them the Ra-QS method, which, overall, gave pretty similar performance to the Se-QI version. In Table 1 we show, for selected accuracy percentages and values of k, the number of exact distance computations required by FastMap, the original BoostMap method, the proposed method, and both intermediate methods Se-QI and Ra-QS.

In all cases (except for results on 100% accuracy, which are dominated by the single query giving the worst results), query-sensitive embeddings lead to better performance than embeddings using a global L_1 distance measure. In some cases, query-sensitive embeddings achieve performance that

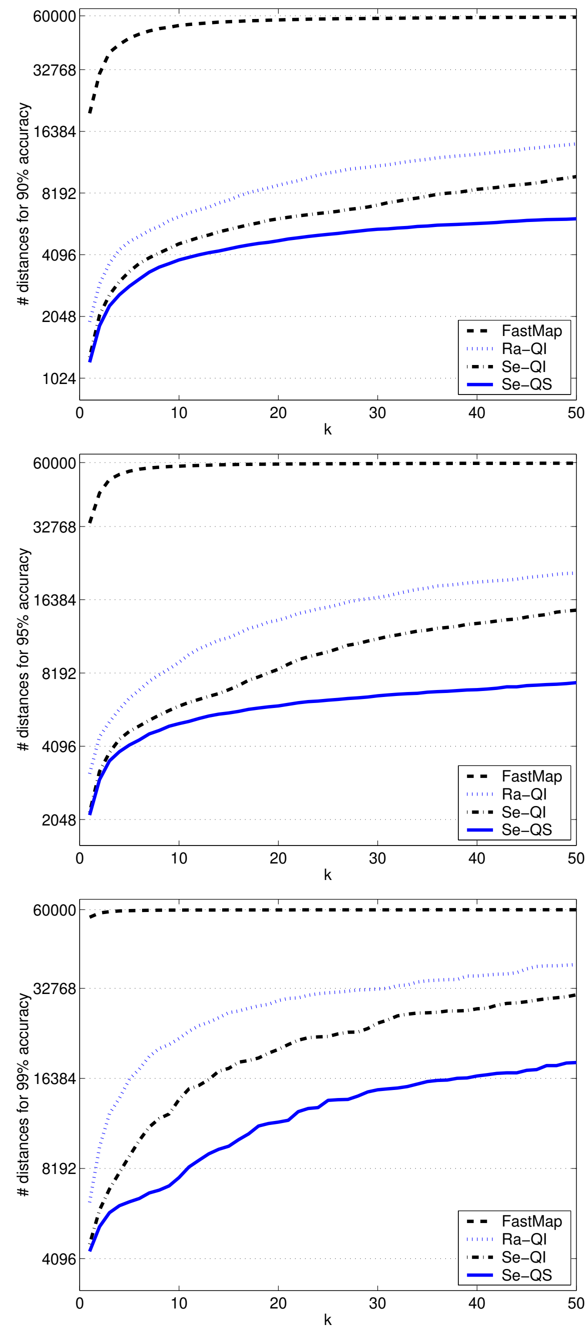

Figure 4: Comparing FastMap, the original BoostMap method (denoted as Ra-QI), the proposed method (denoted as Se-QS), and intermediate method Se-QI (which incorporates our method of choosing training triples, but still constructs a query-insensitive embedding) on the MNIST database of handwritten digits, using the Shape Context Distance as the exact distance measure. We show the number of exact distance computations needed by each method to achieve retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 10,000 query objects that make up the test set of the MNIST database.

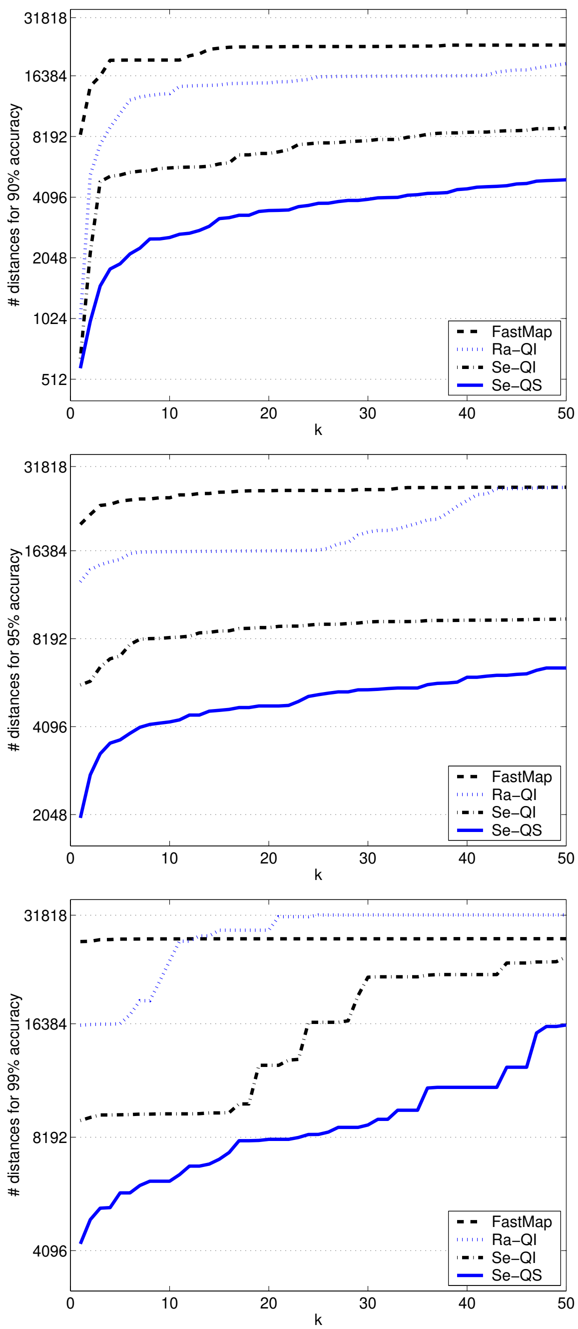

Figure 5: Comparing FastMap, the original BoostMap method (denoted as Ra-QI), the proposed method (denoted as Se-QS), and intermediate method Se-QI (which incorporates our method of choosing training triples, but still constructs a query-insensitive embedding) on the time series database, using constrained Dynamic Time Warping as the exact distance measure. We show the number of exact distance computations needed by each method to achieve retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 1,000 query objects that we use as a test set.

MNIST Database with Shape Context						
k	pct	FastMap	Ra-QI	Ra-QS	Se-QI	Se-QS
1	90	20059	1930	1824	1296	1223
1	95	33858	3161	2789	2190	2135
1	99	56619	6315	5141	4577	4329
1	100	59996	55019	40479	40946	13406
10	90	53852	6280	5233	4631	3866
10	95	58009	9059	6584	5988	5072
10	99	59800	22266	11802	13932	7642
10	100	60000	55019	58677	56936	52066
50	90	59102	14232	9134	9856	6139
50	95	59644	21085	12767	14848	7477
50	99	59980	39311	25878	31176	18510
50	100	60000	59840	59974	59735	59941

Time Series Dataset with Constrained DTW						
k	pct	FastMap	Ra-QI	Ra-QS	Se-QI	Se-QS
1	90	8357	1018	898	649	580
1	95	20176	12851	6484	5691	1995
1	99	27082	16236	9743	9072	4269
1	100	27547	16426	13922	9562	6965
10	90	19613	13364	6521	5721	2582
10	95	24888	16270	9346	8262	4251
10	99	27531	24052	13070	9448	6260
10	100	27623	31818	24730	27267	17627
50	90	23289	18821	9757	9043	4997
50	95	27041	26985	12821	9571	6504
50	99	27564	31818	19357	24672	16265
50	100	27742	31818	26748	27267	26883

Table 1: Comparison of FastMap, the original BoostMap (denoted as Ra-QI), the proposed method (denoted as Se-QS), and the two intermediate methods Ra-QS and Se-QI, on the MNIST dataset based on 10,000 query objects and the time series dataset based on 1,000 query objects. For different values of k, and different percentages of accuracy (shown in the "pct" column), we show the number of exact distance computations required by each embedding method, assuming that we have set the two parameters of that method (dimensionality of embedding and number p of matches to keep after the filter step) to the optimal values that minimize the number of exact distance computations. For comparison, brute force search would require 60000 exact distance computations in the MNIST dataset and 31818 exact distance computations in the time series dataset.

is two or three times as fast for a fixed error rate. Similarly, in all cases, except results on 100% accuracy, the method we introduced in this paper for choosing training triples leads to better performance than training an embedding with randomly chosen triples. Overall, the proposed method, which combines both a query-sensitive distance measure and the new way of choosing training triples, significantly outperforms both FastMap and the original BoostMap method.

To train embeddings, for the original BoostMap, the proposed method, and intermediate methods Ra-QS and Se-QI, we always used a training set of 300,000 triples, generated from a set X_{tr} of 5,000 database objects. The set C of candidate objects also consisted of 5,000 database objects. Query objects from the test set were not used in any part of the training algorithm. Parameter m, the number of weak classifiers to evaluate at each training round, was set to 2,000. Parameter k_1, used in choosing training triples, was set to 5 for the MNIST dataset and to 9 for the time series dataset,

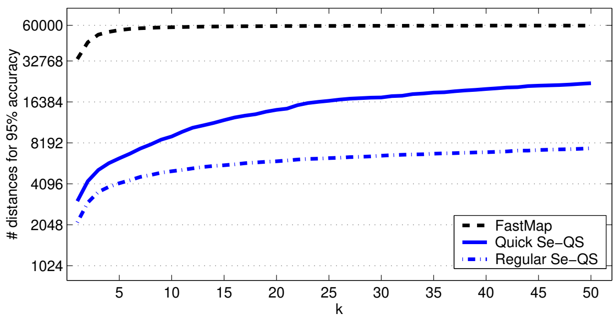

Figure 6: Results using an embedding produced using the Se-QS method, but with sets C and $X_{\rm tr}$ including only 200 objects, and using only 10,000 training triples, so that the preprocessing cost of the Se-QS method becomes very small. We compare those results (denoted as "Quick Se-QS") to the results for FastMap and method "Se-QS" as shown in Fig. 4. "Regular Se-QS" denotes the results using the Se-QS method with $|C|$ and $|X_{\rm tr}|$ equal to 5,000, and with 300,000 training triples. For different values of k, the figure shows the number of exact distance computations required by each embedding method, in order to retrieve the true k nearest neighbors for 95% of the 10,000 queries, for the MNIST dataset.

following the guidelines suggested in Sec. 6 for $k_{\rm max} = 50$. This way, embeddings were optimized for retrieval of up to 50 nearest neighbors per query. In each dataset, we constructed a FastMap embedding by running the FastMap algorithm on a subset of the database, containing 5,000 objects.

We also ran an experiment on the MNIST dataset, in which we used the proposed method, i.e., method Se-QS, but with relatively small sizes for sets C and $X_{\rm tr}$ used in the training algorithm, and with fewer training triples. Both $|C|$ and $|X_{\rm tr}|$ were equal to 200 in this experiment, and we only used 10,000 training triples. Using these settings, the preprocessing cost of the algorithm becomes much smaller: both the number of distances D_X that we need to precompute is smaller (80,000 distances as opposed to 50,000,000 distances in the previous experiments) and the running time for the learning algorithm was much shorter (about 20 minutes, as opposed to about 10 hours using 300,000 triples). Fig. 6 shows the results we obtain with these settings, for 95% retrieval accuracy, and compares those results to what we get using FastMap and using the Se-QS method with $|C|$ and $X_{\rm tr}$ equal to 5,000, and with 300,000 training triples. We see that we still get results that are better than FastMap, and that are useful overall results. Although spending more time on preprocessing improves retrieval efficiency (for fixed accuracy), we can construct useful embeddings even when the time available for preprocessing is limited.

On average, computing exact Shape Context distances can be done at the rate of 15 distances per second, and computing constrained Dynamic Time Warping distances can be done at the rate of about 60 distances per second, on an Opteron 2.2GHz processor. To obtain the corresponding processing times per query for each setting shown in Figs. 4, 5, 6 and Table 1, one simply needs to divide the number of exact distance computations by 15 for Shape Context and by 60 for Dynamic Time Warping. Exact distance computations almost completely determined the processing time per query; the rest of the calculations took a fraction of a second for each query.

10. DISCUSSION

The experimental results reported in this paper provide a quantitative comparison of the proposed algorithm to the original BoostMap method [2] and FastMap [12], on two different datasets that use non-Euclidean, non-metric distance measures: the MNIST dataset of handwritten digits using the Shape Context distance as the underlying distance measure, and a time series dataset using constrained Dynamic Time Warping as the underlying distance measure. The experiments demonstrate that the proposed method gives, for most settings, significantly better results than the original BoostMap method or FastMap.

The main difference of the proposed method with respect to existing embedding methods is that it constructs a query-sensitive distance measure. Such a distance measure captures the fact that different embedding coordinates are important for different queries, and thus leads to improved retrieval accuracy at the filter step of filter-and-refine retrieval.

We should stress that both the Shape Context distance measure and Dynamic Time Warping are non-metric, because they do not obey the triangle inequality. This means that even general indexing tools like M-trees [40], designed for metric spaces, cannot be applied in these datasets. Many other commonly used distance measures, like the Kullback-Leibler distance, or the chamfer distance [3] are also non-metric. Embeddings are the only family of methods that we are aware of that is not domain-specific and that can be applied for efficient retrieval in such spaces. Like other embedding methods, our method is general, and can be applied to arbitrary spaces.

We believe that query-sensitive distance measures may prove useful in other settings, in addition to embedding-based nearest neighbor retrieval. A common problem in data mining, clustering, and pattern recognition applications, is how to construct a meaningful distance measure for comparing high-dimensional vectors. We are interested in exploring whether our algorithm for learning a query-sensitive distance measure can offer advantages in such applications.

11. REFERENCES

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

[2] V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. BoostMap: A method for efficient approximate similarity rankings. In *CVPR*, 2004.

[3] H. Barrow, J. Tenenbaum, R. Bolles, and H. Wolf. Parametric correspondence and chamfer matching: Two new techniques for image matching. In *IJCAI*, pages 659–663, 1977.

[4] S. Belongie, J. Malik, and J. Puzicha. Matching shapes. In *ICCV*, volume 1, pages 454–461, 2001.

[5] S. Belongie, J. Malik, and J. Puzicha. Shape matching and object recognition using shape contexts. *PAMI*, 24(4):509–522, 2002.

[6] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[7] J. Bourgain. On Lipschitz embeddings of finite metric spaces in Hilbert space. *Israel Journal of Mathematics*, 52:46–52, 1985.

[8] T. Bozkaya and Z. Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Database Syst.*, 24(3):361–404, 1999.

[9] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In *VLDB*, pages 89–100, 2000.

[10] C. Domeniconi, J. Peng, and D. Gunopulos. Locally adaptive metric nearest-neighbor classification. *PAMI*, 24(9):1281–1285, 2002.

[11] Ö. Egecioglu and H. Ferhatosmanoglu. Dimensionality reduction and similarity distance computation by inner product approximations. In *International Conference on Information and Knowledge Management*, pages 219–226, 2000.

[12] C. Faloutsos and K. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *ACM SIGMOD*, pages 163–174, 1995.

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[14] T. Hastie and R. Tibshirani. Discriminant adaptive nearest-neighbor classification. *PAMI*, 18(6):607–616, 1996.

[15] G. Hjaltason and H. Samet. Properties of embedding methods for similarity searching in metric spaces. *PAMI*, 25(5):530–549, 2003.

[16] G. Hristescu and M. Farach-Colton. Cluster-preserving embedding of proteins. Technical Report 99-50, CS Department, Rutgers University, 1999.

[17] P. Indyk. *High-dimensional Computational Geometry.* PhD thesis, Stanford University, 2000.

[18] C. T. Jr., A. Traina, B. Seeger, and C. Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *7th International Conference on Extending Database Technology (EDBT)*, pages 51–65, 2000.

[19] K. V. R. Kanth, D. Agrawal, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *ACM SIGMOD International Conference on Management of Data*, pages 166–176, 1998.

[20] E. Keogh. Exact indexing of dynamic time warping. In *VLDB*, pages 406–417, 2002.

[21] N. Koudas, B. C. Ooi, H. T. Shen, and A. K. H. Tung. Ldc: Enabling search by partial distance in a hyper-dimensional space. In *ICDE*, pages 6–17, 2004.

[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[23] C. Li, E. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *IEEE TKDE*, 14(4):792–808, 2002.

[24] S. Roweis and L. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290:2323–2326, 2000.

[25] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Özsoyoglu. Distance based indexing for string proximity search. In *ICDE*, pages 125–136, 2003.

[26] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *VLDB*, pages 516–526, 2000.

[27] R. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

[28] J. Tenenbaum, V. d. Silva, and J. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, 2000.

[29] K. Tieu and P. Viola. Boosting image retrieval. In *CVPR*, pages 228–235, 2000.

[30] E. Tuncel, H. Ferhatosmanoglu, and K. Rose. Vq-index: An index structure for similarity searching in multimedia databases. In *Proc. of ACM Multimedia*, pages 543–552, 2002.

[31] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, volume 1, pages 511–518, 2001.

[32] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *Proc. of ACM SIGKDD*, pages 216–225, 2003.

[33] X. Wang, J. Wang, K. Lin, D. Shasha, B. Shapiro, and K. Zhang. An index structure for data mining and clustering. *Knowledge and Information Systems*, 2(2):161–184, 2000.

[34] R. Weber and K. Bohm. Trading quality for time with nearest-neighbor search. In *International Conference on Extending Database Technology: Advances in Database Technology*, pages 21–35, 2000.

[35] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[36] D. White and R. Jain. Similarity indexing: Algorithms and performance. In *Storage and Retrieval for Image and Video Databases (SPIE)*, pages 62–73, 1996.

[37] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *Proc. of ICDE*, pages 201–208, 1998.

[38] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, 1993.

[39] F. Young and R. Hamer. *Multidimensional Scaling: History, Theory and Applications.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1987.

[40] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate similarity retrieval with M-trees. *The VLDB Journal*, 4:275–293, 1998.