Learning Embeddings for Fast Approximate Nearest Neighbor Retrieval

Vassilis Athitsos, Jonathan Alon, Stan Sclaroff, and George Kollios*
Computer Science Department
Boston University
111 Cummington Street
Boston, MA 02215
email: {athitsos, jalon, sclaroff, gkollios}@cs.bu.edu

Abstract

We present an embedding method that can significantly reduce nearest neighbor retrieval time when the underlying distance measure is computationally expensive. Database and query objects are embedded into a Euclidean space, in which similarities can be rapidly measured using a weighted Manhattan distance. Embedding construction is formulated as a machine learning task, where AdaBoost is used to combine many simple, 1D embeddings into a multidimensional embedding that preserves a significant amount of the proximity structure in the original space. Performance is evaluated in a hand pose estimation system, and a dynamic gesture recognition system, where the proposed method is used to retrieve approximate nearest neighbors under expensive similarity measures. In both systems, BoostMap significantly increases efficiency, with minimal losses in accuracy. Moreover, the experiments indicate that BoostMap compares favorably with existing embedding methods that have been employed in computer vision and database applications, i.e. FastMap and Lipschitz embeddings.

1 Introduction

Many important applications require efficient nearest neighbor retrieval in non-Euclidean, and often non-metric spaces. Finding nearest neighbors efficiently in such spaces can be challenging, because the underlying distance measures can take time superlinear to the length of the data, and also because most indexing methods are not applicable in such spaces. For example, most tree-based and hash-based indexing methods typically assume that objects live in a Euclidean space, or at least a so-called ”coordinate-space”, where each object is represented as a feature vector of fixed dimensions. There is a

[1]This research was funded in part by the U.S. National Science Foundation, under grants IIS-0208876, IIS-0308213, IIS-0329009, and CNS-0202067, and the U.S. Office of Naval Research, under grant N00014-03-1-0108.

wide range of non-Euclidean spaces that violate those assumptions. Some examples of such spaces are proteins and DNA in biology, time series data in various fields, and edge images in computer vision.

Euclidean embeddings (like Bourgain embeddings [17] and FastMap [8]) provide an alternative for indexing non-Euclidean spaces. Using embeddings, we associate each object with a Euclidean vector, so that distances between objects are related to distances between the vectors associated with those objects. Database objects are embedded offline. Given a query object q, its embedding $F(q)$ is computed efficiently online, by measuring distances between q and a small number of database objects. To retrieve the nearest neighbors of q, we first find a small set of candidate matches using distances in the Euclidean space, and then we refine those results by measuring distances in the original space. Euclidean embeddings can significantly improve retrieval time in domains where evaluating the distance measure in the original space is computationally expensive.

This chapter presents BoostMap, a machine learning method for constructing Euclidean embeddings. The algorithm is domain-independent and can be applied to *arbitrary* distance measures, metric or non-metric. With respect to existing embedding methods for efficient approximate nearest neighbor methods, BoostMap has the following advantages:

- Embedding construction explicitly optimizes a quantitative measure of how well the embedding preserves similarity rankings. Existing methods (like Bourgain embeddings [11] and FastMap [8]) typically use random choices and heuristics, and do not attempt to optimize some measure of embedding quality.
- Our optimization method does not make any assumptions about the original distance measure. For example, no Euclidean or metric properties are required.

Embeddings are seen as classifiers, which estimate for any three objects a, b, c if a is closer to b or to c. Starting with a large family of simple, one-dimensional (1D) embeddings, we use AdaBoost [20] to combine those embeddings into a single, high-dimensional embedding that can give highly accurate similarity rankings.

2 Related Work

Various methods have been employed for similarity indexing in multi-dimensional datasets, including hashing and tree structures [29]. 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 cannot be applied to arbitrary spaces.

Approximate nearest neighbor methods have been proposed in [12] 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 [9], 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 [32] M-trees are used for approximate similarity retrieval, while [16] proposes clustering the dataset and retrieving only a small number of clusters (which are stored sequentially on disk) to answer each query. In [13, 7, 4] dimensionality reduction techniques are used where lower-bounding rules are ignored when dismissing dimensions and the focus is on preserving close approximations of distances only. In [27] the authors used VA-files [28] 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 [23] the authors partition the data space into clusters and then the representatives of each cluster are compressed using quantization techniques. Other similar approaches include [15, 19]. 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 [30, 25, 14]. These techniques use the filter and refine approach where an approximation of the original distance that can be computed efficiently is utilized in the filtering step. Query speedup is achieved by pruning a large part of the search space before the original, accurate but more expensive distance measure needs to be applied on few remaining candidates, during the refinement step. Usually, the distance approximation function is designed to be metric (even if the original distance is not), such that traditional indexing techniques can be applied to index the database in order to speed up the filtering stage as well.

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 metric spaces into a Euclidean or pseudo-Euclidean space [3, 8, 11, 18, 22, 26, 31]. Some of these methods, in particular MDS [31], Bourgain embeddings [3, 10], LLE [18] and Isomap [22] 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 handled by Lipschitz embeddings [10], FastMap [8], MetricMap [26] and SparseMap [11], which can readily compute the embedding of the query, measuring only a small number of exact distances in the process. These four methods are the most related to our approach. The goal of BoostMap is to achieve better indexing performance in domains where those four methods are applicable.

3 Background on Embeddings

Let X be a set of objects, and $D_X(x_1, x_2)$ be a distance measure between objects $x_1, x_2 \in X$. D_X can be metric or non-metric. A Euclidean embedding $F : X \to \mathbb{R}^d$ is a function that maps objects from X into the d-dimensional Euclidean space $\mathbb{R}^d$, where distance is measured using a measure $D_{\mathbb{R}^d}$. $D_{\mathbb{R}^d}$ is typically an L_p or weighted L_p norm. Given X and D_X, our goal is to construct an embedding F that can be used for efficient and accurate approximate k-nearest neighbor retrieval, for previously unseen query objects, and for different values of k.

In this section we describe some existing methods for constructing Euclidean embeddings. We briefly go over Lipschitz embeddings [10], Bourgain embeddings [3, 10],

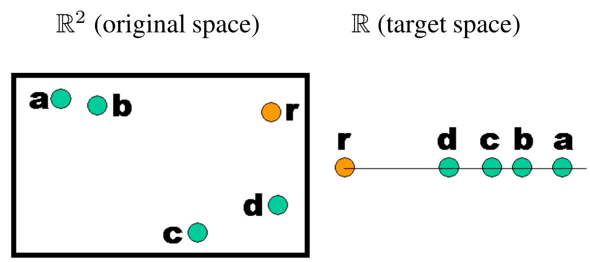

Figure 1: An embedding F^r of five 2D points (shown on the left) into the real line (shown on the right), using r as the reference object. The target of each 2D point on the line is labeled with the same letter as the 2D point. The classifier $\tilde{F}^r$ (Equation 7) classifies correctly 46 out of the 60 triples we can form from these five objects (assuming no object occurs twice in a triple). Examples of misclassified triples are: (b, a, c), (c, b, d), (d, b, r). For example, b is closer to a than it is to c, but $F^r(b)$ is closer to $F^r(c)$ than it is to $F^r(a)$.

FastMap [8] and MetricMap [26]. All these methods, with the exception of Bourgain embeddings, can be used for efficient approximate nearest neighbor retrieval. Although Bourgain embeddings require too many distance computations in the original space X in order to embed the query, there is a heuristic approximation of Bourgain embeddings called SparseMap [11] that can also be used for efficient retrieval.

3.1 Lipschitz Embeddings

We can extend D_X to define the distance between elements of X and subsets of X. Let $x \in X$ and $R \subset X$. Then,

$$D_X(x, R) = \min_{r \in R} D_X(x, r) \; . \tag{1}$$

Given a subset $R \subset X$, a simple one-dimensional Euclidean embedding $F^R : X \rightarrow \mathbb{R}$ can be defined as follows:

$$F^R(x) = D_X(x, R) \; . \tag{2}$$

The set R that is used to define F^R is called a *reference set*. In many cases R can consist of a single object r, which is typically called a *reference object* or a *vantage object* [10]. In that case, we denote the embedding as F^r:

$$F^r(x) = D_X(x, r) \; . \tag{3}$$

If D_X obeys the triangle inequality, F^R intuitively maps nearby points in X to nearby points on the real line $\mathbb{R}$. In many cases D_X may violate the triangle inequality for some triples of objects (an example is the chamfer distance [2]), but F^R may still map nearby points in X to nearby points in $\mathbb{R}$, at least most of the time [1]. On the other hand, distant objects may also map to nearby points (Figure 1).

In order to make it less likely for distant objects to map to nearby points, we can define a multidimensional embedding $F : X \to \mathbb{R}^k$, by choosing k different reference sets $R_1, ..., R_k$:

$$F(x) = (F^{R_1}(x), ..., F^{R_k}(x)) \ . \tag{4}$$

These embeddings are called *Lipschitz embeddings* [3, 11, 10]. Bourgain embeddings [3, 10] are a special type of Lipschitz embeddings. For a finite space X containing $|X|$ objects, we choose $\lfloor \log |X| \rfloor^2$ reference sets. In particular, for each $i = 1, ..., \lfloor \log |X| \rfloor$ we choose $\lfloor log|X| \rfloor$ reference sets, each with 2^i elements. The elements of each set are picked randomly. Bourgain embeddings are optimal in some sense: using a measure of embedding quality called *distortion*, if D_X is a metric, Bourgain embeddings achieve $O(\log(|X|))$ distortion, and there exist metric spaces X for which no embedding can achieve lower distortion [10, 17]. However, we should emphasize that if D_X is non-metric, then Bourgain embeddings can have distortion higher than $O(\log(|X|))$.

A weakness of Bourgain embeddings is that, in order to compute the embedding of an object, we have to compute its distances D_X to almost all objects in X. This happens because some of the reference sets contain at least half of the objects in X. In database applications, computing all those distances is exactly what we want to avoid. SparseMap [11] is a heuristic simplification of Bourgain embeddings, in which the embedding of an object can be computed by measuring only $O(\log^2 |X|)$ distances. The penalty for this heuristic is that SparseMap no longer guarantees $O(\log(|X|))$ distortion for metric spaces.

Another way to speed up retrieval using a Bourgain embedding is to define this embedding using a relatively small random subset $X' \subset X$. That is, we choose $\lfloor \log |X'| \rfloor^2$ reference sets, which are subsets of X'. Then, to embed any object of X we only need to compute its distances to all objects of X'. We use this method to produce Bourgain embeddings of different dimensions in the experiments we describe in this chapter. We should note that, if we use this method, the optimality of the embedding only holds for objects in X', and there is no guarantee about the distortion attained for objects of the larger set X. We should also note that, in general, defining an embedding using a smaller set X' can in principle also be applied to Isomap [22], LLE [18] and even MDS [31], so that it takes less time to embed new objects.

The theoretical optimality of Bourgain embeddings with respect to distortion does not mean that Bourgain embeddings actually outperform other methods in practice. Bourgain embeddings have a worst-case bound on distortion, but that bound is very loose, and in actual applications the quality of embeddings is often much better, both for Bourgain embeddings and for embeddings produced using other methods.

A simple and attractive alternative to Bourgain embeddings is to simply use Lipschitz embeddings in which all reference sets are singleton, as in Eq. 3. In that case, if we have a d-dimensional embedding, in order to compute the embedding of a previously unseen object we only need to compute its distance to d reference objects.

3.2 FastMap and MetricMap

A family of simple, one-dimensional embeddings is proposed in [8] and used as building blocks for FastMap. The idea is to choose two objects $x_1, x_2 \in X$, called, pivot

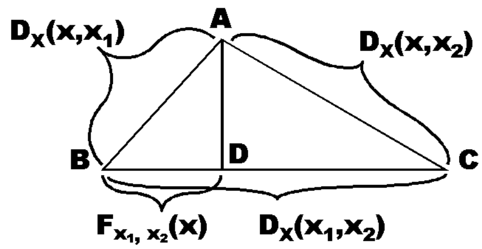

Figure 2: Computing $F^{x_1,x_2}(x)$, as defined in Equation 5: we construct a triangle ABC so that the sides AB, AC, BC have lengths $D_X(x, x_1)$, $D_X(x, x_2)$ and $D_X(x_1, x_2)$ respectively. We draw from A a line perpendicular to BC, and D is the intersection of that line with BC. The length of the line segment BD is equal to $F^{x_1,x_2}(x)$.

objects, and then, given an arbitrary $x \in X$, define the embedding F^{x_1,x_2} of x to be the *projection* of x onto the "line" x_1x_2. As illustrated in Figure 2, the projection can be defined by treating the distances between x, x_1, and x_2 as specifying the sides of a triangle in R^2:

$$F^{x_1,x_2}(x) = \frac{D_X(x, x_1)^2 + D_X(x_1, x_2)^2 - D_X(x, x_2)^2}{2D_X(x_1, x_2)} \,. \tag{5}$$

If X is Euclidean, then F^{x_1,x_2} will map nearby points in X to nearby points in $\mathbb{R}$. In practice, even if X is non-Euclidean, $F^(x_1, x_2)$ often still preserves some of the proximity structure of X.

FastMap [8] uses multiple pairs of pivot objects to project a finite set X into $\mathbb{R}^k$ using only $O(kn)$ evaluations of D_X. The first pair of pivot objects (x_1, x_2) is chosen using a heuristic that tends to pick points that are far from each other. Then, the rest of the distances between objects in X are "updated", so that they correspond to projections into the "hyperplane" perpendicular to the "line" x_1x_2. Those projections are computed again by treating distances between objects in X as Euclidean distances in some $\mathbb{R}^m$. After distances are updated, FastMap is recursively applied again to choose a next pair of pivot objects and apply another round of distance updates. Although FastMap treats X as a Euclidean space, the resulting embeddings can be useful even when X is non-Euclidean, or even non-metric. We have seen that in our own experiments (Section 6).

MetricMap [26] is an extension of FastMap, that maps X into a a pseudo-Euclidean space. The experiments in [26] report that MetricMap tends to do better than FastMap when X is non-Euclidean. So far we have no conclusive experimental comparisons between MetricMap and our method, partly because some details of the MetricMap algorithm have not been fully specified (as pointed out in [10]), and therefore we could not be sure how close our MetricMap implementation was to the implementation evaluated in [26].

3.3 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 Euclidean embedding F can be used in a filter-and-refine framework [10], as follows:

- Offline preprocessing step: compute and store vector $F(x)$ for every database object x.
- Embedding step: given a query object q, compute $F(q)$. Typically this involves computing distances D_X between q and a small number of objects of X.
- Filter step: find the database objects whose vectors are the p most similar vectors to $F(q)$. This step involves measuring distances in $\mathbb{R}^d$.
- 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 in Euclidean space is much faster. The filter step discards most database objects by comparing Euclidean 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 get 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 increases, computing $F(q)$ becomes more expensive (because we need to measure distances to more objects of X), and measuring distances between vectors in $\mathbb{R}^d$ also becomes more expensive. At the same time, we may get more accurate results in the filter step, and we may be able to decrease p. The best choice of p and d will depend on domain-specific parameters like k, 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).

4 Associating Embeddings with Classifiers

In this section we define a quantitative measure of embedding quality, that is directly related to how well an embedding preserves the similarity structure of the original space. The BoostMap learning algorithm will then be shown to directly optimize this quantitative measure.

As previously, X is a set of objects, and $D_X(x_1, x_2)$ is a distance measure between objects $x_1, x_2 \in X$. Let (q, x_1, x_2) be a triple of objects in X. We define the *proximity order* $P_X(q, x_1, x_2)$ to be a function that outputs whether q is closer to x_1 or to x_2:

$$P_X(q, x_1, x_2) = \begin{cases} 1 & \text{if } D_X(q, x_1) < D_X(q, x_2) \text{ .} \\ 0 & \text{if } D_X(q, x_1) = D_X(q, x_2) \text{ .} \\ -1 & \text{if } D_X(q, x_1) > D_X(q, x_2) \text{ .} \end{cases} \tag{6}$$

If F maps space X into $\mathbb{R}^d$ (with associated distance measure $D_{\mathbb{R}^d}$), then F can be used to define a *proximity classifier* $\tilde{F}$ that estimates, for any triple (q, x_1, x_2), whether q is closer to x_1 or to x_2, simply by checking whether $F(q)$ is closer to $F(x_1)$ or to $F(x_2)$.

$$\tilde{F}(q, x_1, x_2) = D_{\mathbb{R}^d}(F(q), F(x_2)) - D_{\mathbb{R}^d}(F(q), F(x_1)) \, . \tag{7}$$

If we define $\mathrm{sign}(x)$ to be 1 for $x > 0$, 0 for $x = 0$, and -1 for $x < 0$, then $\mathrm{sign}(\tilde{F}(q, x_1, x_2))$ is an estimate of $P_X(q, x_1, x_2)$.

We define the classification error $G(\tilde{F}, q, x_1, x_2)$ of applying $\tilde{F}$ on a particular triple (q, x_1, x_2) as:

$$G(\tilde{F}, q, x_1, x_2) = \frac{|P_X(q, x_1, x_2) - \mathrm{sign}(\tilde{F}(q, x_1, x_2))|}{2} \, . \tag{8}$$

Finally, the overall classification error $G(\tilde{F})$ is defined to be the expected value of $G(\tilde{F}, q, x_1, x_2)$, over X^3, i.e. the set of triples of objects of X. If X contains a finite number of objects, we get:

$$G(\tilde{F}) = \frac{\sum_{(q, x_1, x_2) \in X^3} G(\tilde{F}, q, x_1, x_2)}{|X|^3} \, . \tag{9}$$

Using the definitions in this section, our problem definition is very simple: we want to construct an embedding $F_{\text{out}} : X \rightarrow \mathbb{R}^d$ in a way that minimizes $G(\tilde{F}_{\text{out}})$. If an embedding F has error rate $G(\tilde{F}) = 0$, then F perfectly preserves nearest neighbor structure, meaning that for any $x_1, x_2 \in X$, and any integer $k > 0$, x_1 is the k-th nearest neighbor of x_2 in X if and only if $F(x_1)$ is the k-th nearest neighbor of $F(x_2)$ in the set $F(X)$. Overall, the lower the error rate $G(\tilde{F})$ is, the better the embedding F is in terms of preserving the similarity structure of X.

We will address the problem of minimizing $G(\tilde{F}_{\text{out}})$ as a problem of combining classifiers. As building blocks we will use a family of simple, 1D embeddings. Then, we will apply AdaBoost to combine many 1D embeddings into a high-dimensional embedding F_{out} with low error rate.

5 Constructing Embeddings via AdaBoost

The 1D embeddings that we will use as building blocks in our algorithm will be of two types: embeddings of type F^r as defined in Eq. 3, and embeddings of type F^{x_1, x_2}, as defined in Eq. 5. Each 1D embedding F corresponds to a binary classifier $\tilde{F}$. These classifiers estimate, for triples (q, x_1, x_2) of objects in X, if q is closer to x_1 or x_2. If F is a 1D embedding, we expect $\tilde{F}$ to behave as a *weak classifier* [20], meaning that it will have a high error rate, but it should still do better than a random classifier. We want to combine many 1D embeddings into a multidimensional embedding that behaves as a *strong classifier*, i.e. that has relatively high accuracy. To choose which 1D embeddings to use, and how to combine them, we use the AdaBoost framework [20].

5.1 Overview of the Training Algorithm

The training algorithm for BoostMap is an adaptation of AdaBoost to the problem of embedding construction. The inputs to the training algorithm are the following:

- A training set $T = ((q_1, a_1, b_1), ..., (q_t, a_t, b_t))$ of t triples of objects from X.
- A set of labels $Y = (y_1, ..., y_t)$, where $y_i \in \{-1, 1\}$ is the class label of (q_i, a_i, b_i). If $D_X(q_i, a_i) < D_X(q_i, b_i)$ then y_i = 1, else $y_i = -1$. The training set includes no triples where q_i is equally far from a_i and b_i.
- A set $C \subset X$ of candidate objects. Elements of C can be used to define 1D embeddings.
- A matrix of distances from each $c \in C$ to each q_i, a_i and b_i included in one of the training triples in T.

The training algorithm combines many classifiers $\tilde{F}_j$ associated with 1D embeddings F_j, into a classifier $H = \sum_{j=1}^{d} \alpha_j \tilde{F}_j$. The classifiers $\tilde{F}_j$ and weights α_j are chosen so as to minimize the classification error of H. Once we get the classifier H, its components $\tilde{F}_j$ are used to define a high-dimensional embedding $F = (F_1, ..., F_d)$, and the weights α_j are used to define a weighted L_1 distance, that we will denote as $D_{\mathbb{R}^d}$, on $\mathbb{R}^d$. We are then ready to use F and $D_{\mathbb{R}^d}$ to embed objects into $\mathbb{R}^d$ and compute approximate similarity rankings.

Training is done in a sequence of rounds. At each round, the algorithm either modifies the weight of an already chosen classifier, or selects a new classifier. Before we describe the algorithm in detail, here is an intuitive, high-level description of what takes place at each round:

1. Go through the classifiers $\tilde{F}_j$ that have already been chosen, and try to identify a weight α_j that, if modified, decreases the training error. If such an α_j is found, modify it accordingly.
2. If no weights were modified, consider a set of of classifiers that have not been chosen yet. Identify, among those classifiers, the classifier $\tilde{F}$ which is the best at correcting the mistakes of the classifiers that have already been chosen.
3. Add that classifier $\tilde{F}$ to the set of chosen classifiers, and compute its weight. The weight that is chosen is the one that maximizes the corrective effect of $\tilde{F}$ on the output of the previously chosen classifiers.

Intuitively, weak classifiers are chosen and weighted so that they complement each other. Even when individual classifiers are highly inaccurate, the combined classifier can have very high accuracy, as evidenced in several applications of AdaBoost (for example in [24]).

Trying to modify the weight of an already chosen classifier before adding in a new classifier is a heuristic that reduces the number of classifiers that we need in order to achieve a given classification accuracy. Since each classifier corresponds to a dimension in the embedding, this heuristic leads to lower-dimensional embeddings, which reduce database storage requirements and retrieval time.

5.2 The Training Algorithm in Detail

This subsection, together with the original AdaBoost reference [20], provides enough information to allow implementation of BoostMap, and it can be skipped if the reader is more interested in a high-level description of our method.

The training algorithm performs a sequence of training rounds. At the j-th round, it maintains a weight $w_{i,j}$ for each of the t triples (q_i, a_i, b_i) of the training set, so that $\sum_{i=1}^{t} w_{i,j} = 1$. For the first round, each $w_{i,1}$ is set to $\frac{1}{t}$.

At the j-th round, we try to modify the weight of an already chosen classifier or add a new classifier, in a way that improves the overall training error. A key measure, that is used to evaluate the effect of choosing classifier $\tilde{F}$ with weight α, is the function Z_j:

$$Z_j(\tilde{F}, \alpha) = \sum_{i=1}^{t} (w_{i,j} \exp(-\alpha y_i \tilde{F}(q_i, a_i, b_i))) \ . \tag{10}$$

The full details of the significance of Z_j can be found in [20]. Here it suffices to say that $Z_j(\tilde{F}, \alpha)$ is a measure of the benefit we obtain by adding $\tilde{F}$ with weight α to the list of chosen classifiers. The benefit increases as $Z_j(\tilde{F}, \alpha)$ decreases. If $Z_j(\tilde{F}, \alpha) > 1$, then adding $\tilde{F}$ with weight α is actually expected to increase classification error.

A frequent operation during training is identifying the pair $(\tilde{F}, \alpha)$ that minimizes $Z_j(\tilde{F}, \alpha)$. For that operation we will use the shorthand $Z_{\min}$, defined as follows:

$$Z_{\min}(B, j) = \mathrm{argmin}_{(\tilde{F}, \alpha) \in B \times \mathbb{R}} Z_j(\tilde{F}, \alpha) \ . \tag{11}$$

In the above equation, B is a set of classifiers.

At training round j, the training algorithm goes through the following steps:

1. Let B_j be the set of classifiers chosen so far. Set $(\tilde{F}, \alpha) = Z_{\min}(B_j, j)$. If $Z_j(\tilde{F}, \alpha) < .9999$ then modify the current weight of $\tilde{F}$, by adding α to it, and proceed to the next round. We use .9999 as a threshold, instead of 1, to avoid minor modifications with insignificant numerical impact.

2. Construct a set of 1D embeddings $\mathbb{F}_{j1} = \{F^r \mid r \in C\}$ where F^r is defined in Eq. 3, and C is the set of candidate objects that is one of the inputs to the training algorithm (Sec. 5.1).

3. For a fixed number m, choose randomly a set C_j of m pairs of elements of C, and construct a set of embeddings $\mathbb{F}_{j2} = \{F^{x_1,x_2} \mid (x_1, x_2) \in C_j\}$, where F^{x_1,x_2} is as defined in Eq. 5.

4. Define $\mathbb{F}_j = \mathbb{F}_{j1} \cup \mathbb{F}_{j2}$. We set $\tilde{\mathbb{F}}_J = \{\tilde{F} \mid F \in \mathbb{F}_j\}$.

5. Set $(\tilde{F}, \alpha) = Z_{\min}(\tilde{\mathbb{F}}_j, j)$.

6. Add $\tilde{F}$ to the set of chosen classifiers, with weight α.

7. Set training weights $w_{i,j+1}$ as follows:

$$w_{i,j+1} = \frac{w_{i,j} \exp(-\alpha y_i \tilde{F}(q_i, a_i, b_i))}{Z_j(\tilde{F}, \alpha)} \ . \tag{12}$$

Intuitively, the more $\alpha\tilde{F}(q_i, a_i, b_i)$ disagrees with class label y_i, the more $w_{i,j+1}$ increases with respect to $w_{i,j}$. This way triples that get misclassified by many of the already chosen classifiers will carry a lot of weight and will influence the choice of classifiers in the next rounds.

The algorithm can terminate when we have chosen a desired number of classifiers, or when, at a given round j, no combination of $\tilde{F}$ and α makes $Z_j(\tilde{F}, \alpha) < 1$.

5.3 Training Output: Embedding and Distance

The output of the training stage is a classifier $H = \sum_{j=1}^{d} \alpha_j \tilde{F}_j$, where each $\tilde{F}_j$ is associated with a 1D embedding F_j. The final output of BoostMap is an embedding $F_{\text{out}} : X \to \mathbb{R}^d$ and a weighted Manhattan (L_1) distance $D_{\mathbb{R}^d} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$:

$$F_{\text{out}}(x) = (F_1(x), ..., F_d(x)) \ . \tag{13}$$

$$D_{\mathbb{R}^d}((u_1, ..., u_d), (v_1, ..., v_d)) = \sum_{j=1}^{d} (\alpha_j |u_j - v_j|) \ . \tag{14}$$

It is important to note (and easy to check) that, the way we define F_{out} and $D_{\mathbb{R}^d}$, if we apply Equation 7 to obtain a classifier $\tilde{F}_{\text{out}}$ from F_{out}, then $\tilde{F}_{\text{out}} = H$, i.e. $\tilde{F}_{\text{out}}$ is equal to the output of AdaBoost. This means that the output of AdaBoost, which is a classifier, is mathematically equivalent to the embedding F_{out}: given a triple (q, a, b), both the embedding and the classifier give the exact same answer as to whether q is closer to a or to b. If AdaBoost has been successful in learning a good classifier, the embedding F_{out} inherits the properties of that classifier, with respect to preserving the proximity order of triples.

Also, we should note that this equivalence between classifier and embedding relies on the way we define $D_{\mathbb{R}^d}$. For example, if $D_{\mathbb{R}^d}$ were defined without using weights α_j, or if $D_{\mathbb{R}^d}$ were defined as an L_2 metric, the equivalence would not hold.

5.4 Complexity

If C is the set of candidate objects, and n is the number of database objects, we need to compute $|C|n$ distances D_X to learn the embedding and compute the embeddings of all database objects. At each training round, we evaluate classifiers defined using $|C|$ reference objects and m pivot pairs. Therefore, the computational time per training round is $O((|C| + m)t)$, where t is the number of training triples. In our experiments we always set $m = |C|$.

Computing the d-dimensional embedding of a query object takes $O(d)$ time and requires $O(d)$ evaluations of D_X. Overall, query processing time is not worse than that of FastMap [8], SparseMap [11], and MetricMap [26].

6 Experiments

We used two datasets to compare BoostMap to FastMap [8] and Bourgain embeddings [3, 11]: a database of hand images, and an ASL (American Sign Language) database,

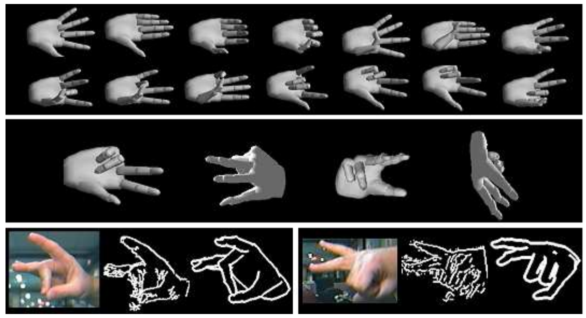

Figure 3: Top: 14 of the 26 hand shapes used to generate the hand database. Middle: four of the 4128 3D orientations of a hand shape. Bottom: for two test images we see, from left to right: the original hand image, the extracted edge image that was used as a query, and a correct match (noise-free computer-generated edge image) retrieved from the database.

containing video sequences of ASL signs. In both datasets the test queries were not part of the database, and not used in the training.

The hand database contains 107,328 hand images, generated using computer graphics. 26 hand shapes were used to generate those images. Each shape was rendered under 4128 different 3D orientations (Figure 3). As queries we used 703 real images of hands. Given a query, we consider a database image to be correct if it shows the same hand shape as the query, in a 3D orientation within 30 degrees of the 3D orientation of the query [1]. The queries were manually annotated with their shape and 3D orientation. For each query there are about 25-35 correct matches among the 107,328 database images. Similarity between hand images is evaluated using the symmetric chamfer distance [2], applied to edge images. Evaluating the exact chamfer distance between a query and the entire database takes about 260 seconds.

The ASL database contains 880 gray-scale video sequences. Each video sequence depicts a sign, as signed by one of three native ASL signers (Figure 4). As queries we used 180 video sequences of ASL signs, signed by a single signer who was not included in the database. Given a query, we consider a database sequence to be a correct match if it is labeled with the same sign as the query. For each query, there are exactly 20 correct matches in the database. Similarity between video sequences is measured as follows: first, we use the similarity measure proposed in [6], which is based on optical flow, as a measure of similarity between single frames. Then, we use Dynamic Time Warping [5] to compute the optimal time alignment and the overall matching cost between the two sequences. Evaluating the exact distance between the query and the entire database takes about six minutes.

In all experiments, the training set for BoostMap was 200,000 triples. For the hand database, the size of C (from Sec. 5.2) was 1000 elements, and the elements of C were chosen randomly at each step from among 3282 objects, i.e. C was different at each training round (a slight deviation from the description in Sec. 5), to speed up training time. For the ASL database, the size of C was 587 elements. The objects used to define FastMap and Bourgain embeddings were also chosen from the same 3282 and

Figure 4: Four sample frames from the video sequences in the ASL database.

587 objects respectively. Also, in all experiments, we set $m = |C|$, where m is the number of embeddings based on pivot pairs that we consider at each training round. Learning a 256-dimensional BoostMap embedding of the hand database took about two days, using a 1.2GHz Athlon processor.

To evaluate the accuracy of the approximate similarity ranking for a query, we used two measures: exact nearest neighbor rank (ENN rank) and highest ranking correct match rank (HRCM rank). The ENN rank is computed as follows: let b be the database object that is the nearest neighbor to the query q under the exact distance D_X. Then, the ENN rank for that query in a given embedding is the rank of b in the similarity ranking that we get using the embedding. The HRCM rank for a query in an embedding is the best rank among all correct matches for that query, based on the similarity ranking we get with that embedding. In a perfect recognition system, the HRCM rank would be 1 for all queries. Figures 5, 6, 7, and 8 show the median ENN ranks and median HRCM ranks for each dataset, for different dimensions of BoostMap, FastMap and Bourgain embeddings. For the hand database, BoostMap gives significantly better results than the other two methods, for 16 or more dimensions. In the ASL database, BoostMap does either as well as FastMap, or better than FastMap, in all dimensions. In both datasets, Bourgain embeddings overall do worse than BoostMap and FastMap.

With respect to Bourgain embeddings, we should mention that they are not quite appropriate for online queries, because they require evaluating too many distances in order to produce the embedding of a query. SparseMap [11] was formulated as a heuristic approximation of Bourgain embeddings, that is appropriate for online queries. We have not implemented SparseMap but, based on its formulation, it would be a surprising result if SparseMap achieved higher accuracy than Bourgain embeddings.

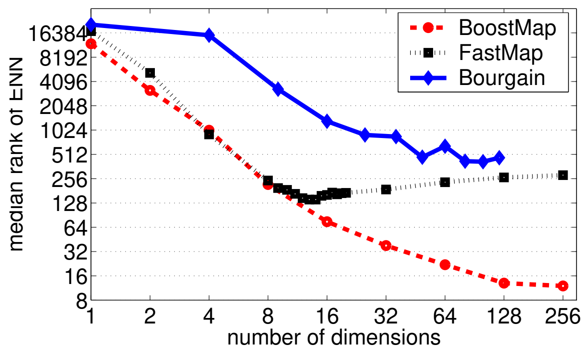

Figure 5: Median rank of exact nearest neighbor (ENN), versus number of dimensions, in approximate similarity rankings obtained using three different methods, for 703 queries to the hand database.

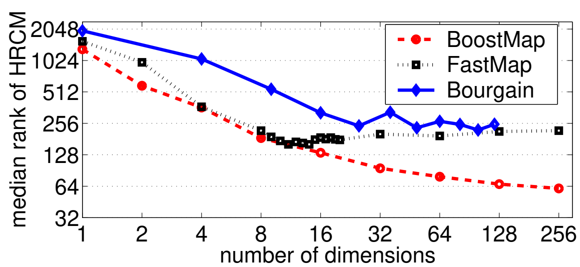

Figure 6: Median rank of highest ranking correct match (HRCM), versus number of dimensions, in approximate similarity rankings obtained using three different methods, for 703 queries to the hand database. For comparison, the median HRCM rank for the exact distance was 21.

6.1 Filter-and-refine Experiments

As described in Sec. 3.3, we can use an embedding to perform filter-and-refine retrieval of nearest neighbors. The usefulness of an embedding in filter-and-refine retrieval depends on two questions: how often we successfully identify the nearest neighbors of a query, and how much the overall retrieval time is.

For both BoostMap and FastMap, we found the optimal combination of d (dimensionality of the embedding) and p (the number of candidate matches retained after the filter step) that would allow 1-nearest-neighbor retrieval to be correct 95% or 100% of the time, while minimizing retrieval time. Table 1 shows the optimal values of p and d, and the associated computational savings over standard nearest-neighbor retrieval, in which we evaluate the exact distance between the query and each database object. In both datasets, the bulk of retrieval time is spent computing exact distances in the original space. The time spent in computing distances in the Euclidean space is negligible, even for a 256-dimensional embedding. For the hand database, BoostMap leads

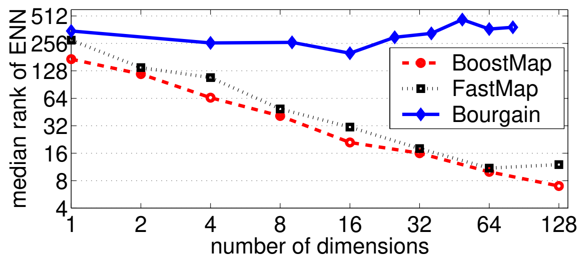

Figure 7: Median rank of exact nearest neighbor (ENN), versus number of dimensions, in approximate similarity rankings obtained using three different methods, for 180 queries to the ASL database.

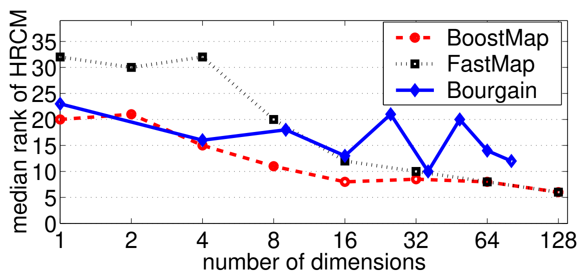

Figure 8: Median rank of highest ranking correct match (HRCM), versus number of dimensions, in approximate similarity rankings obtained using three different methods, for 180 queries to the ASL database. For comparison, the median HRCM rank for the exact distance was 3.

to significantly faster retrieval, because we need to compute far fewer exact distances in the refine step, while achieving the same error rate as FastMap.

7 Discussion and Future Work

With respect to existing embedding methods, the main advantage of BoostMap is that it is formulated as a classifier-combination problem, that can take advantage of powerful machine learning techniques to assemble a high-accuracy embedding from many simple, 1D embeddings. The main disadvantage of our method, at least in the current implementation, is the running time of the training algorithm. However, in many applications, trading training time for embedding accuracy would be a desirable tradeoff. At the same time, we are interested in exploring ways to improve training time.

A possible extension of BoostMap is to use it to approximate not the actual distance between objects, but a hidden state space distance. For example, in our hand image dataset, what we are really interested in is not retrieving images that are similar with respect to the chamfer distance, but images that actually have the same hand pose.

ENN retrieval accuracy and efficiency for hand database					
Method	BoostMap		FastMap		Exact D_X
ENN-accuracy	95%	100%	95%	100%	100%
Best d	256	256	13	10	N/A
Best p	406	3850	3838	17498	N/A
D_X # per query	823	4267	3864	17518	107328
seconds per query	2.3	10.6	9.4	42.4	260

ENN retrieval accuracy and efficiency for ASL database					
Method	BoostMap		FastMap		Exact D_X
ENN-accuracy	95%	100%	95%	100%	100%
Best d	64	64	64	32	N/A
Best p	129	255	141	334	N/A
D_X # per query	249	375	269	398	880
seconds per query	103	155	111	164	363

Table 1: Comparison of BoostMap, FastMap and using brute-force search, for the purpose of retrieving the exact nearest neighbors successfully for 95% or 100% of the queries, using filter-and-refine retrieval. The letter d is the dimensionality of the embedding. The letter p stands for the number of top matches that we keep from the filter step (i.e. using the embeddings). D_X # per query is the total number of D_X computations needed per query, in order to embed the query and rank the top p candidates. The exact D_X column shows the results for brute-force search, in which we not use a filter step, and we simply evaluate D_X distances between the query and all database images.

We can modify the training labels Y provided to the training algorithm, so that instead of describing proximity with respect to the chamfer distance, they describe proximity with respect to actual hand pose. The resulting similarity rankings may be worse approximations of the chamfer distance rankings, but they may be better approximations of the actual pose-based rankings. A similar idea has been applied in [21], although in the context of a different approximate nearest neighbor framework.

References

[1] V. Athitsos and S. Sclaroff. Estimating hand pose from a cluttered image. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 432–439, 2003.

[2] H.G. Barrow, J.M. Tenenbaum, R.C. Bolles, and H.C. Wolf. Parametric correspondence and chamfer matching: Two new techniques for image matching. In *International Joint Conference on Artificial Intelligence*, pages 659–663, 1977.

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

[4] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In *International Conference on Very Large Data Bases*, pages 89–100, 2000.

[5] T.J. Darrell, I.A. Essa, and A.P. Pentland. Task-specific gesture analysis in real-time using interpolated views. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(12), 1996.

[6] A.A. Efros, A.C. Berg, G. Mori, and J. Malik. Recognizing action at a distance. In *IEEE International Conference on Computer Vision*, pages 726–733, 2003.

[7] 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.

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

[9] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *International Conference on Very Large Databases*, pages 518–529, 1999.

[10] G.R. Hjaltason and H. Samet. Properties of embedding methods for similarity searching in metric spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(5):530–549, 2003.

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

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

[13] 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.

[14] Eamonn Keogh. Exact indexing of dynamic time warping. In *International Conference on Very Large Data Bases*, pages 406–417, 2002.

[15] 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 *IEEE International Conference on Data Engineearing*, pages 6–17, 2004.

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

[17] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. In *IEEE Symposium on Foundations of Computer Science*, pages 577–591, 1994.

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

[19] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *International Conference on Very Large Data Bases*, pages 516–526, 2000.

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

[21] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *IEEE International Conference on Computer Vision*, pages 750–757, 2003.

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

[23] 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.

[24] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages 511–518, 2001.

[25] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E.J. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 216–225, 2003.

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

[27] 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.

[28] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *International Conference on Very Large Data Bases*, pages 194–205, 1998.

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

[30] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *IEEE International Conference on Data Engineering*, pages 201–208, 1998.

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

[32] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate similarity retrieval with m-trees. *International Journal on Very Large Data Bases*, 4:275–293, 1998.