Boston University Computer Science Tech. Report No. 2003-023, November 21, 2003.

Submitted to Computer Vision and Pattern Recognition, 2004.

BoostMap: A Method for Efficient Approximate Similarity Rankings

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

This paper introduces BoostMap, a method that can significantly reduce retrieval time in image and video database systems that employ computationally expensive distance measures, metric or non-metric. 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 image and video 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 Bourgain embeddings.

1 Introduction

Content-based image and video retrieval is important for interactive applications, where users want to identify content of interest in large databases [10]. Identifying nearest neighbors in a large collection of objects can also be used as a tool for clustering or nearest neighbor-based object recognition [1, 3, 19]. Depending on the number of objects and the computational complexity of evaluating the distance between pairs of objects, identifying the k nearest neighbors can be too inefficient for practical applications. Measuring distances can be expensive because of high-dimensional feature vectors [13] or because the distance measure takes super-linear time with respect to the number of dimensions [2, 3].

This paper presents BoostMap, an efficient method for obtaining rankings of all database objects in approximate order of similarity to the query. The query still needs to be compared to all database objects, but comparisons are done after the query and the database objects have been embedded to a Euclidean space, where distances can be measured rapidly using a weighted Manhattan (L_1) distance. In many applications ([3], for example) where evaluating distances is the computational bottleneck, substituting the original distances with L_1 distances can lead to orders-of-magnitude improvements in efficiency.

The main novelty in this paper is looking at embedding construction from a machine learning perspective. Embeddings are seen as classifiers, which estimate for three objects a, b, c if a is closer to b or to c. The error rate of an embedding on this task is closely related to how well it preserves similarity rankings. Formulating embedding construction as a classification problem allows us to use powerful machine learning tools to obtain an embedding that is explicitly optimized for similarity ranking approximation. In particular, starting with a large family of simple, one-dimensional (1D) embeddings, we use AdaBoost [18] to combine those embeddings into a single, high-dimensional embedding that can give highly accurate similarity rankings.

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 subset of database objects. In the case of nearest-neighbor queries, the most similar matches obtained using the embedding can be reranked using the original distance measure, to improve accuracy, in a filter-and-refine framework [11]. Overall, the original distance measure is applied only between the query and a small number of database objects.

2 Related Work

Various methods have been employed for similarity indexing in image and video databases, including hashing

[1]This research was funded in part through grants from the U.S. National Science Foundation, and the U.S. Office of Naval Research.

and tree structures [24]. However, the performance of such methods degrades in high dimensions, a phenomenon well-known as the curse of dimensionality. Furthermore, tree-based methods typically rely on Euclidean or metric properties, and cannot be applied to arbitrary non-metric spaces. An other class of approaches, probabilistic approximate nearest neighbor methods, have been proposed in [14, 19]. However, those methods work for specific sets of metrics, and they are not applicable to arbitrary distance measures.

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 [4, 9, 12, 17, 20, 23, 25]. Some of these methods, in particular MDS [25], LLE [17] and Isomap [20] are not applicable for 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 [11], Bourgain embeddings [4, 11], FastMap [9], MetricMap [23] and SparseMap [12], 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.

Image and video database systems have made use of Lipschitz embeddings [1, 5, 6] and FastMap [15, 16], to map objects into a low-dimensional Euclidean space that is more manageable for tasks like online retrieval, data visualization, or classifier training. The goal of our method is to improve embedding accuracy in such applications.

3 Problem Definition

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 $F : X \rightarrow \mathbb{R}^d$ 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.

In this paper, we are interested in constructing an embedding F that, given a query object q, can provide good approximate similarity rankings of database objects, i.e. rankings of database objects in order of decreasing similarity (increasing distance) to the query. To make the problem definition precise, and specify the quantity that our method tries to optimize, we introduce in this section a quantitative measure, that can be used to evaluate how "good" an embedding is in providing approximate similarity rankings.

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:

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

If F maps space X (with associated distance measure D_X) into $\mathbb{R}^d$ (with associated distance measure $D_{\mathbb{R}^d}$), then F can be used to define a *proximity classifier* $\bar{F}$ that estimates P_X using $P_{\mathbb{R}^d}$, i.e. the proximity order function of $\mathbb{R}^d$ with distance $D_{\mathbb{R}^d}$:

$$\bar{F}(q, x_1, x_2) = P_{\mathbb{R}^d}(F(q), F(x_1), F(x_2)) . \quad (2)$$

$\bar{F}$ outputs one of three possible values. Alternatively, we can define a continuous-output classifier $\tilde{F}(q, x_1, x_2)$, that simply outputs the difference between the distances from $F(q)$ to $F(x_2)$ and to $F(x_1)$:

$$\tilde{F}(q, x_1, x_2) = D_{\mathbb{R}^d}(q, x_2) - D_{\mathbb{R}^d}(q, x_1) . \quad (3)$$

$\bar{F}$ can be seen as a discretization of $\tilde{F}$, such that $\bar{F}$ outputs 1, 0 or -1 if $\tilde{F}$ outputs respectively a value that is greater than, equal to, or less than zero. Figure 1 shows a simple embedding example (that uses definitions from Sec. 4), and some misclassified triples.

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

$$G(\bar{F}, q, x_1, x_2) = \frac{|P_X(q, x_1, x_2) - \bar{F}(q, x_1, x_2)|}{2} . \quad (4)$$

Finally, the overall classification error $G(\bar{F})$ is defined to be the expected value of $G(\bar{F}, q, x_1, x_2)$, over all triples of objects in X:

$$G(\bar{F}) = \frac{\sum_{(q,x_1,x_2)\in X^3} G(\bar{F}, q, x_1, x_2)}{|X|^3} . \quad (5)$$

If $G(\bar{F}) = 0$ then we consider that F perfectly preserves the proximity structure of X. In that case, if x is the k-nearest neighbor of q in X, $F(x)$ is the k-nearest neighbor of $F(q)$ in $F(X)$, for any value of k.

Overall, the classification error $G(\bar{F})$ is a quantitative measure of how well F preserves the proximity structure of X, and how closely the approximate similarity rankings obtained in $F(X)$ will resemble the exact similarity rankings obtained in X. Using the definitions in this section, our problem definition is very simple: we want to construct an embedding $F : X \rightarrow \mathbb{R}^d$ in a way that minimizes $G(\bar{F})$.

We will address this problem as a problem of combining classifiers. In Sec. 4 we will identify a family of simple, 1D embeddings. Each such embedding F' is expected to preserve at least a small amount of the proximity structure of X, meaning that $G(\bar{F'})$ is expected to be less than 0.5,

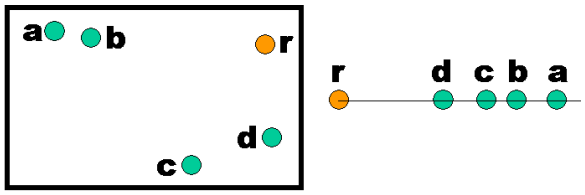

Figure 1: An embedding F_r of five 2D points into the real line, using r as the reference object. The target of each 2D point on the line is labeled with the same letter as the 2D point. The classifier $\bar{F}_r$ (Eq. 2) classifies correctly 46 out of the 60 triples we can form from these five objects (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 $F_r(a)$.

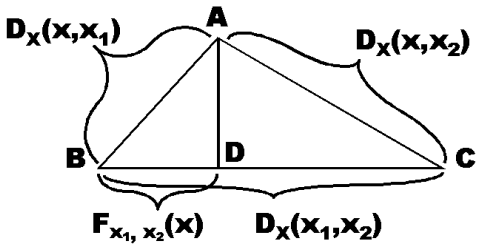

Figure 2: Computing $F_{x_1,x_2}(x)$, as defined in Eq. 7: we construct a triangle ABC so that the sides AB, AC, BC have lengths $D_X(x, x_1), D_X(x, x_2)$ and $D_X(x_1, x_2)$ respectively. We draw from A a line perpendicular to BC, and D is the intersection of that line with BC. The length of the line segment BD is equal to $F_{x_1,x_2}(x)$.

which would be the error rate of a random classifier. Then, in Sec. 5 we will apply AdaBoost to combine many 1D embeddings into a high-dimensional embedding F with low error rate $G(\bar{F})$.

4 Some Simple 1D Embeddings

Given an object $r \in X$, a simple 1D Euclidean embedding F_r can be defined as follows:

$$F_r(x) = D_X(x, r) \; . \tag{6}$$

The object r that is used to define F_r is typically called a *reference object* or a *vantage object*.

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).

Another family of simple, 1D embeddings, is proposed in [9] 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" $\overline{x_1 x_2}$. 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 $\mathbb{R}^2$, and applying the Pythagorean theorem:

$$F_{x_1,x_2}(x) = \frac{D_X(x, x_1)^2 + D_X(x_1, x_2)^2 - D_X(x, x_2)^2}{2D_X(x_1, x_2)} \; . \tag{7}$$

The objects x_1, x_2 that are used to define F_{x_1,x_2} are called *pivot objects*. 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.

If the space X contains $|X|$ objects, then each object can be used as a reference object, and each pair of objects can be used as a pair of pivot objects. Therefore, the number of possible 1D embeddings we can define on X using the definitions of this section is quadratic to $|X|$. Sec. 5 describes how to selectively combine these embeddings into a single, high-dimensional embedding.

5 Constructing Embeddings via AdaBoost

Now we have identified a large family of 1D embeddings. Every such 1D embedding F' is defined using either a reference object, or a pair of pivot objects. Each F' corresponds to a proximity classifier $\bar{F}'$ (Eq. 2) and an associated continuous-output 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. In general, we expect $\bar{F}'$ and $\tilde{F}'$ to behave as *weak classifiers* [18], meaning that they will have a high error rate, but they 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 [18].

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 q_i is closer to a_i than it is to 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, and selects a set of classifiers and associated weights. At each training 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.

Choosing classifiers and weights so that they correct mistakes of already chosen classifiers is a key component of the AdaBoost algorithm. This way, weak classifiers are chosen and weighted so that they complement each other, and even when each individual classifier is highly inaccurate, the combined classifier can have very high accuracy, as evidenced in several applications of AdaBoost in computer vision [21, 22].

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. Essentially we are forcing the training algorithm to try to first minimize error using the classifiers that it has already chosen, before it considers adding in new classifiers. 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

Now that we have given a high-level overview of the training algorithm, we proceed to provide the actual details. This subsection, together with the original AdaBoost reference [18], provides enough information to allow implementation of BoostMap, and it can be skipped if the reader is more interested in the high-level aspects. We should also mention that the training algorithm described in this subsection is a simple adaptation of AdaBoost, and is not in itself a contribution of this paper. The main contribution in this paper is rather the formulation of embedding construction as a machine learning task, in a way that allows the application of powerful machine learning tools like AdaBoost in order to improve embedding accuracy.

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:

$$Z_j(\tilde{F}', \alpha) = \sum_{i=1}^{t} (w_{i,j} \exp(-\alpha y_i \tilde{F}'(q_i, a_i, b_i))) . \quad (8)$$

The full details of the significance of Z_j can be found in [18]. Here it suffices to say that if $Z_j(\tilde{F}', \alpha) < 1$ then adding classifier $\tilde{F}'$ with weight α to the list of chosen classifiers is overall beneficial, and is expected to reduce the training 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:

$$Z_{\min}(B, j) = \text{argmin}_{(\tilde{F}', \alpha) \in B \times \mathbb{R}} Z_j(\tilde{F}', \alpha) . \quad (9)$$

In the above equation, B is a set of candidate embeddings. $Z_{\min}$ returns a pair $(\tilde{F}', \alpha)$ that minimizes $Z_j(\tilde{F}', \alpha)$.

At each round, 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 should note that if $Z_j(\tilde{F}', \alpha) < 1$ then the weight modification is expected to decrease the training error. We use a lower threshold (.9999) 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. 6, 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 $C_j = \{(x_{1,1}, x_{1,2}), ..., (x_{m,1}, x_{m,2})\}$ 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. 7.

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:

$$w_{i,j+1} = \frac{w_{i,j} \exp(-\alpha y_i \tilde{F}'(q_i, a_i, b_i))}{Z_j(\tilde{F}', \alpha)} . \quad (10)$$

Intuitively, the more $\alpha\tilde{F}'(q_i, a_i, b_i)$ disagrees with 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.

5.3 Training Output: Embedding and Distance

The output of the training stage is a continuous-output classifier $H = \sum_{j=1}^{d} \alpha_j \tilde{F}'_j$, where each $\tilde{F}'_j$ is associated with a 1D embedding F'_j. From H we can obtain a $\{-1, 0, -1\}$-valued classifier H' by converting positive values to 1 and negative values to -1. H' takes as input a triple (q, x_1, x_2) of objects in X and provides an estimate of the proximity order for that triple.

The final output of BoostMap is an embedding $F : X \to \mathbb{R}^d$ and a distance $D_{\mathbb{R}^d} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$:

$$F(x) = (F'_1(x), ..., F'_d(x)) . \quad (11)$$

$$D_{\mathbb{R}^d}((u_1, ..., u_d), (v_1, ..., v_d)) = \sum_{j=1}^{d} (\alpha_j |u_j - v_j|) . \quad (12)$$

$D_{\mathbb{R}^d}$ is a weighted Manhattan (L_1) distance. Note that $D_{\mathbb{R}^d}$ is usually not a metric, because some of the α_j's can be negative. The important thing is that $D_{\mathbb{R}^d}(u, v)$ can be evaluated very efficiently, in $O(d)$ time.

6 Complexity

If C is the set of candidate objects (one of the inputs to the training algorithm), 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. In addition, at each training round, we evaluate

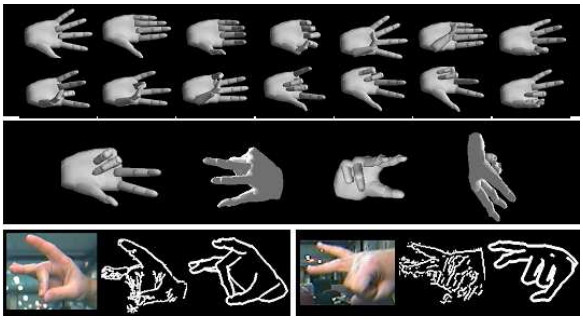

Figure 3: Top: 14 of the 26 hand shapes used to generate the hand database. Middle: four of the 4128 3D orientations of a hand shape. Bottom: for two test images we see, from left to right: the original hand image, the extracted edge image that was used as a query, and a correct match (noise-free computer-generated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

1D embeddings defined using $|C|$ reference objects and m pivot pairs. Each embedding is evaluated on each of the t triples in the training set. Therefore, the computational time per training round is $O((|C| + m)t)$. In our experiments we always set $m = |C|$.

Computing the d-dimensional embedding of a query object, that has not been seen before, takes $O(d)$ time and requires $O(d)$ evaluations of D_X. Computing the $D_{\mathbb{R}^d}$ distances between the embedding of the query and the embeddings of all n database objects takes $O(dn)$ time. Overall, query processing time is not worse than that of FastMap, SparseMap, MetricMap, and Bourgain embeddings.

7 Experiments

We used two datasets to compare BoostMap to FastMap [9] and Bourgain embeddings [4, 12]: a database of hand images, and an ASL (American Sign Language) database, containing video sequences of ASL signs. In both datasets BoostMap was trained using a subset of the database, and the test queries were not parts 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. The 30-degree threshold was chosen based on how much human subjects tended to disagree with each other in visually estimating the 3D orientation of a hand [1]. The queries were manually annotated with their shape and 3D orientation. For each query there are about 25-35 correct matches among

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

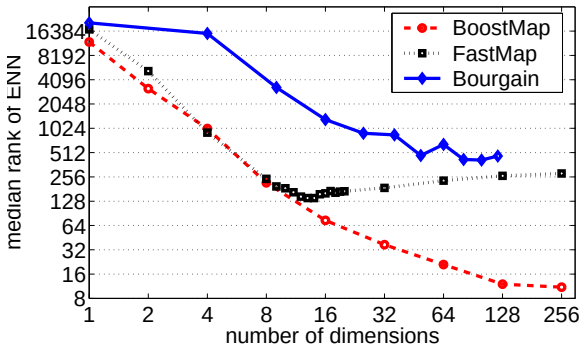

Figure 5: Median rank of exact nearest neighbor (ENN), versus number of dimensions, in approximate similarity rankings obtained using three different methods, for 703 queries to the hand database.

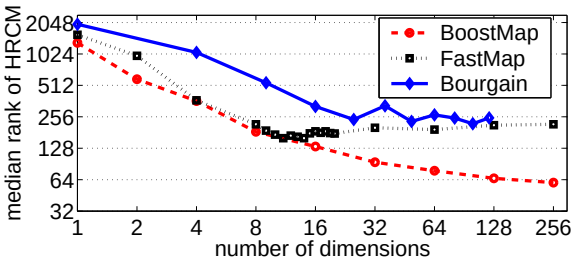

Figure 6: Median rank of highest ranking correct match (HRCM), versus number of dimensions, in approximate similarity rankings obtained using three different methods, for 703 queries to the hand database. For comparison, the median HRCM rank for the exact distance was 21.

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 purpose of similarity queries in the hand database is to obtain estimates of the 3D hand pose in the query image.

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, that were 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. None of the 20 correct matches was signed by the signer that signed in the query. Similarity between video sequences is measured as follows: first, we use the similarity measure proposed in [8], which is based on optical flow, as a measure of similarity between single frames. Then, we use Dynamic Time Warping [7] 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. The purpose of similarity queries in the ASL database is to recognize the sign depicted in the query video sequence.

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 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.

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

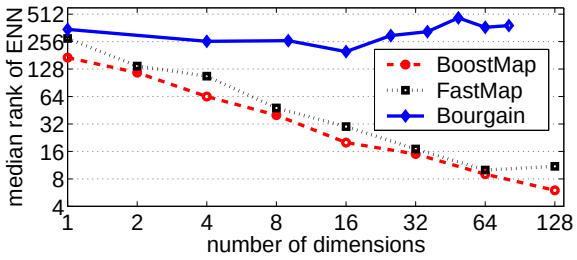

Figure 7: Median rank of exact nearest neighbor (ENN), versus number of dimensions, in approximate similarity rankings obtained using three different methods, for 180 queries to the ASL database.

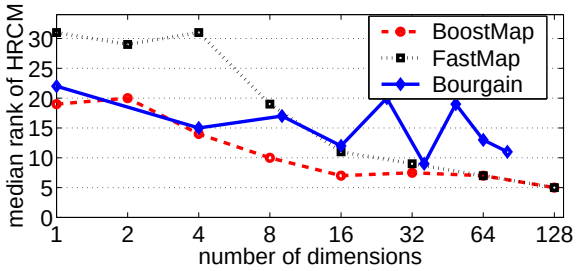

Figure 8: Median rank of highest ranking correct match (HRCM), versus number of dimensions, in approximate similarity rankings obtained using three different methods, for 180 queries to the ASL database. For comparison, the median HRCM rank for the exact distance was 3.

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 at least as well as FastMap in all dimensions, and better than FastMap for some dimensions. In both datasets, Bourgain embeddings give the worst accuracy in most dimensions.

In applications where we are interested in retrieving the k nearest neighbors or k correct matches, BoostMap can be used in a filter-and-refine framework [11], where we use the embedding-based approximate similarity rankings to select p candidates, and then we sort those candidates applying the exact distance D_X. The best choice of p and d, where d is the dimensionality of the embedding, will depend on domain-specific parameters like k, the time it takes to compute the distance D_X, the time it takes to compute the weighted L_1 distance between d-dimensional vectors, and the desired retrieval accuracy.

In our two datasets the main computational bottleneck

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	405	3850	3937	16840	N/A
D_X # per query	822	4267	3960	10850	107328
seconds per query	2	10.4	9.6	40.8	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	259	375	279	398	880
seconds per query	107	155	115	164	363

Table 1: Comparison of BoostMap, FastMap and using only exact distances, for the purpose of retrieving the exact nearest neighbors successfully for 95% or 100% of the queries. The letter d is the dimensionality of the embedding. The letter p stands for number of top matches to keep from the approximate ranking, on which the exact distance is evaluated. 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 standard nearest-neighbor retrieval, i.e. by evaluating D_X distances between the query and all database images. The time per query is in seconds, on a 1.2GHz Athlon processor.

is the computation of the exact distances. To illustrate the computational advantage of using BoostMap on these domains, we evaluated the optimal d and p that would allow nearest-neighbor retrieval to be correct 95% or 100% of the time, while minimizing the number of times we need to compute the exact distance. Table 1 shows the optimal values for BoostMap and FastMap in the two datasets, and the associated computational savings over standard nearest-neighbor retrieval, where we evaluate the distance between the query and each database object.

We should note that the distance measures that we used in the two datasets are both non-metric, because they do not obey the triangle inequality.

Although we have not implemented SparseMap [12] so far, we should mention that SparseMap was formulated as a heuristic approximation of Bourgain embeddings, so it would be a surprising result if SparseMap achieved higher accuracy than Bourgain embeddings in our datasets.

8 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 al-

gorithm. However, in many applications, trading training time for embedding accuracy would be a desirable trade-off. At the same time, we are interested in exploring ways to improve training time.

It is important to compare BoostMap to more existing methods (we hope to have a MetricMap implementation soon), and in different datasets, to gain intuition on the relative advantages and disadvantages of each method. A particularly interesting dataset is the MNIST database of handwritten digits, using shape context as the distance measure [3]. The shape context achieves a very low error rate on that data, but using the publicly available implementation it takes several hours to compute distances between a test image and 10,000 training images.

It is interesting to note that BoostMap is not optimized explicitly for nearest-neighbor retrieval, but instead it tries to approximate the similarity ranking of the entire database given a query. Although we have not carried out any evaluation yet, we expect BoostMap to also give good results for k-farthest-neighbor or median-neighbor queries, which can also be useful in some applications. At the same time, by choosing more specialized training triples (q, a, b), such that a is a k-nearest neighbor of q and b is not, we may be able to guide BoostMap towards optimizing k-nearest neighbor queries for a specific value of k.

Another 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. 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 [19], 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 *CVPR*, volume 1, 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 *IJCAI*, pages 659–663, 1977.

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

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

[5] Y. Chang, C. Hu, and M. Turk. Manifold of facial expression. In *IEEE International Workshop on Analysis and Modeling of Faces and Gestures*, pages 28–35, 2003.

[6] S.S. Cheung and A. Zakhor. Fast similarity search on video signatures. In *ICIP*, 2003.

[7] T.J. Darrell, I.A. Essa, and A.P. Pentland. Task-specific gesture analysis in real-time using interpolated views. *PAMI*, 18(12), 1996.

[8] A.A. Efros, A.C. Berg, G. Mori, and J. Malik. Recognizing action at a distance. In *ICCV*, pages 726–733, 2003.

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

[10] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: The QBIC system. *IEEE Computer*, 28(9), 1995.

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

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

[13] J. Huang, S. Kumar, M. Mitra, W. Zhu, and R. Zabih. Image indexing using color correlograms. In *CVPR*, pages 762–768, 1999.

[14] P. Indyk. *High-dimensional Computational Geometry*. PhD thesis, MIT, 2000.

[15] V. Kobla and D.S. Doerman. Extraction of features for indexing MPEG-compressed video. In *IEEE Workshop on Multimedia Signal Processing*, pages 337–342, 1997.

[16] E.G.M. Petrakis, C. Faloutsos, and K.I Lin. ImageMap: An image indexing method based on spatial similarity. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):979–987, 2002.

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

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

[19] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *ICCV*, pages 750–757, 2003.

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

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

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

[23] 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.

[24] 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.

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