Boston University Computer Science Technical Report No. 2004-014, April 12, 2004.

Learning Euclidean Embeddings for Indexing and Classification

Vassilis Athitsos, Joni Alon, Stan Sclaroff, and George Kollios*

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

ABSTRACT

BoostMap is a recently proposed method for efficient approximate nearest neighbor retrieval in arbitrary non- Euclidean spaces with computationally expensive and possibly non-metric distance measures. Database and query objects are embedded into a Euclidean space, in which similarities can be rapidly measured using a weighted Manhattan distance. The key idea is formulating embedding construction as a machine learning task, where AdaBoost is used to combine simple, 1D embeddings into a multidimensional embedding that preserves a large amount of the proximity structure of the original space. This paper demonstrates that, using the machine learning formulation of BoostMap, we can optimize embeddings for indexing and classification, in ways that are not possible with existing alternatives for constructive embeddings, and without additional costs in retrieval time. First, we show how to construct embeddings that are query-sensitive, in the sense that they yield a different distance measure for different queries, so as to improve nearest neighbor retrieval accuracy for each query. Second, we show how to optimize embeddings for nearest neighbor classification tasks, by tuning them to approximate a parameter space distance measure, instead of the original feature-based distance measure.

1. INTRODUCTION

Many important database applications require representing and indexing data that belong to a non-Euclidean, and often non-metric space. Some examples are proteins and DNA in biology, time series data in various fields, and edge images in computer vision. Indexing such data can be challenging, because the underlying distance measures can take time superlinear to the length of the data, and also because many common tree-based and hash-based indexing methods typically work in a Euclidean space, or at least a so-called "coordinate-space", where each object is represented as a feature vector of fixed dimensions.

Euclidean embeddings (like Bourgain embeddings [17] and FastMap [10]) 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 Euclidean distances between the mappings of those objects. Indexing can then be done in the Euclidean space, and a refinement of the retrieved results can then be performed 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.

BoostMap [2] is a recently introduced method for embedding arbitrary (metric or non-metric) spaces into Euclidean spaces. The main difference between BoostMap and other existing methods is that BoostMap treats embeddings as classifiers, and constructs them using machine learning. In particular, given three objects a, b and c in the original space X, an embedding F can be used to make an educated guess as to whether a is closer to b or to c. The guess is simply that a is closer to b than it is to c if $F(a)$ is closer to $F(b)$ than it is to $F(c)$. If using some embedding F we can make the right guess for all triples, then that embedding perfectly preserves k-nearest-neighbor structure, for any value of k. Overall, we want to construct embeddings that make wrong guesses on as few triples as possible. The classification error of an embedding is the fraction of triples on which the embedding makes a wrong guess.

In this paper, we describe three extensions of BoostMap, that can be used to improve the quality of the embedding, when the application is approximate nearest neighbor retrieval or efficient nearest neighbor classification:

- We show how to construct *query-sensitive* embeddings, in which the weighted L_1 distance used in the Euclidean space depends on the query. In a high-dimensional embedding, using a query-sensitive distance measure provides an elegant way to capture the fact that different coordinates are important in different regions of the space.
- In cases where the ultimate goal is *classification* of the query based on its k nearest neighbors, we show how to create embeddings that are explicitly optimized for classification accuracy, as opposed to being optimized for preserving distances or nearest neighbors.
- We describe an improved method for selecting the training set used by BoostMap. In the original formulation,

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

the training set was chosen at random.

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 [12]. Overall, the original distance measure is applied only between the query and a small number of database objects.

We also describe some preliminary experiments, in which we compare our method to FastMap [10], using as a dataset the MNIST database of handwritten digits [16], and using the chamfer distance as the distance measure in the original space. Our original BoostMap formulation leads to signicantly more efficient retrieval than FastMap. The three extensions introduced in this paper, i.e. query-sensitive embeddings, optimization of classification accuracy, and better choice of training set, lead to additional gains in efficiency and classification accuracy. We are working on evaluating BoostMap on more datasets, with different similarity measure, in order to get a clearer picture of its performance vs. other existing embedding methods.

2. RELATED WORK

Various methods have been employed for similarity indexing in multi-dimensional datasets, including hashing and tree structures [27]. However, the performance of such methods degrades in high dimensions. This phenomenon is one of the many aspects of the "curse of dimensionality" problem. Another problem with tree-based methods is that they typically rely on Euclidean or metric properties, and cannot be applied to arbitrary non-metric spaces. Approximate nearest neighbor methods have been proposed in [14, 22] 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 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 [7, 10, 13, 20, 23, 26, 28]. Some of these methods, in particular MDS [28], Bourgain embeddings [7, 12], LLE [20] and Isomap [23] 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 [12], FastMap [10], MetricMap [26] and SparseMap [13], 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.

Various database systems have made use of Lipschitz embeddings [4, 8, 9] and FastMap [15, 19], 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. Given X and D_X, our goal is to construct an embedding F that, given a query object q, can provide accurate approximate similarity rankings of database objects, i.e. rankings of database objects in order of decreasing similarity (increasing distance) to the query.

3.1 Variations of the Problem

Depending on the domain and application, there are different variations of the general goal, which is to provide accurate approximate similarity rankings . In this paper we will explicitly address three different versions of this goal:

- **Version 1:** We want to rank *all* database objects in approximate (but as accurate as possible) order of similarity to the query object. In this variant, we care not only about identifying the nearest neighbors of the query, but also the farthest neighbors, and in general we want to get an approximate rank for each database object.
- **Version 2:** We want to approximately (but as accurately as possible) identify the k nearest neighbors of the query object, where the value of k is much smaller than the size of the database.
- **Version 3:** We want to *classify* the query object using k-nearest neighbor classification, and we want to construct an embedding and a weighted L_1 distance that are optimized for classification accuracy.

BoostMap can be used to address all three versions. In the BoostMap framework, every embedding F defines a classifier $\tilde{F}$ which, given triples of objects (q, x_1, x_2) of X, provides an estimate of whether q is more similar to x_1 or to x_2. The way we will customize BoostMap to address each of the three versions is by choosing an appropriate training set of triples of objects, and by using an appropriate definiition of what is "similar". After we make these choices, we use the same algorithm in all three cases.

3.2 Formal Definitions

In order to specify the quantity that the BoostMap algorithm 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. Let D be a distance measure on X. For the first two variations of our problem statement, $D = D_X$, but for the third variation we will use an alternative distance measure, that depends on class labels (Section 9). 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:

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

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 classi-*

fier $\tilde{F}$ that estimates P_X using $P_{\mathbb{R}^d}$, i.e. the proximity order function of $\mathbb{R}^d$ with distance $D_{\mathbb{R}^d}$:

$$\tilde{F}(q, x_1, x_2) = D_{\mathbb{R}^d}(F(q), F(x_2)) - D_{\mathbb{R}^d}(F(q), F(x_1)) \ . \quad (2)$$

If we define $\text{sign}(x)$ to be 1 for $x > 0$, 0 for $x = 0$, and -1 for $x < 0$, then $\text{sign}(\tilde{F}(q, x_1, x_2))$ 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:

$$G(\tilde{F}, q, x_1, x_2) = \frac{|P_X(q, x_1, x_2) - \text{sign}(\tilde{F}(q, x_1, x_2))|}{2} \ . \quad (3)$$

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 all triples of objects in X:

$$G(\tilde{F}) = \frac{\sum_{(q, x_1, x_2) \in X^3} G(\tilde{F}, q, x_1, x_2)}{|X|^3} \ . \quad (4)$$

If $G(\tilde{F}) = 0$ 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 $G(\tilde{F})$ 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 $G(\tilde{F})$.

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 $G(\tilde{F}')$ is expected to be less than 0.5, which would be the error rate of a random classifier. Then, in Sec. 7 we will apply AdaBoost to combine many 1D embeddings into a high-dimensional embedding F with low error rate $G(\tilde{F})$.

4. BACKGROUND ON EMBEDDINGS

In this section we describe some existing methods for constructing Euclidean embeddings. We briefly go over Lipschitz embeddings [12], Bourgain embeddings [7, 12], FastMap [10] 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 [13] that can also be used for efficient retrieval.

4.1 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,

$$D_X(x, R) = \min_{r \in R} D_X(x, r) \ . \quad (5)$$

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

$$F^R(x) = D_X(x, R) \ . \quad (6)$$

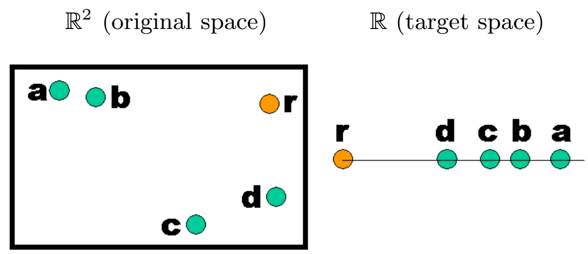

Figure 1: An embedding F^r of five 2D points (shown on the left) into the real line (shown on the right), using r as the reference object. The target of each 2D point on the line is labeled with the same letter as the 2D point. The classifier $\tilde{F}_r$ (Equation 2) classifies correctly 46 out of 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 $F^r(a)$.

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* [12]. In that case, we denote the embedding as F^r.

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 [5]), but F^R may still map nearby points in X to nearby points in $\mathbb{R}$, at least most of the time [4]. 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 $F : X \rightarrow \mathbb{R}^k$, by choosing k different reference sets $R_1, ..., R_k$:

$$F(x) = (F^{R_1}(x), ..., F^{R_k}(x)) \ . \quad (7)$$

These embeddings are called *Lipschitz embeddings* [7, 13, 12]. Bourgain embeddings [7, 12] 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*, Bourgain embeddings achieve $O(|X|)$ distortion, and there exist spaces X for which no better distortion can be achieved. More details can be found in [12, 18].

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, and in database applications computing those distances is exactly what we want to avoid. SparseMap [13] is a heuristic simplification of Bourgain embeddings, in which the embedding of an object can be computed by measuring only $O(\log^2 n)$ distances.

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

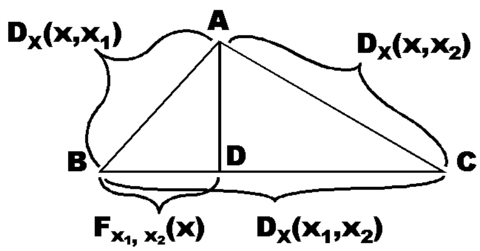

Figure 2: Computing $F^{x_1,x_2}(x)$, as defined in Equation 8: we construct a triangle ABC so that the sides AB, AC, BC have lengths $D_X(x,x_1), D_X(x,x_2)$ and $D_X(x_1,x_2)$ respectively. We draw from A a line perpendicular to BC, and D is the intersection of that line with BC. The length of the line segment BD is equal to $F^{x_1,x_2}(x)$.

objects of X'. We used this method to produce Bourgain embeddings of different dimensions in the experiments we describe in [2]. 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 [23], LLE [20] and even MDS [28], 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. In the experiments described in [2], BoostMap outperformed Bourgain embeddings significantly.

A simple and attractive alternative to Bourgain embeddings is to simply use Lipschitz embeddings in which all reference sets are singleton. 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.

4.2 FastMap and MetricMap

A family of simple, one-dimensional embeddings, is proposed in [10] 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 $x \in X$, 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:

$$F^{x_1,x_2}(x) = \frac{D_X(x,x_1)^2 + D_X(x_1,x_2)^2 - D_X(x,x_2)^2}{2D_X(x_1,x_2)} . \quad (8)$$

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 [10] 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 $\mathbb{R}^{\geqslant}$. 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 11).

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 [12]), and therefore we could not be sure how close our MetricMap implementation was to the implementation evaluated in [26].

4.3 Embedding Application: Filter-and-refine Retrieval

In applications where we are interested in retrieving the k nearest neighbors or k correct matches for a query object q, a d-dimensional Euclidean embedding F can be used in a filter-and-refine framework [12], as follows:

- Offline preprocessing step: compute and store vector $F(x)$ for every database object x.
- Filter step: given a query object q, compute $F(q)$, and find the database objects whose 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 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, comparing Euclidean vectors becomes more expensive, but we may also 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).

5. MOTIVATION FOR BOOSTMAP

Equations 6 and 8 define a family of one-dimensional embeddings. Given a space of objects X, each object $r \in X$

can define a 1D embedding, using Equation 6 with $R = \{r\}$. Each pair of objects can also define a 1D embedding, using Equation 8. Therefore, given n objects of X, the number of 1D embeddings we can construct using those objects is $O(n^2)$.

Intuitively, we expect such 1D embeddings to map nearby objects to nearby points on the line, but at the same time they will frequently map pairs of distant objects into pairs of nearby points. In order to make it more likely for distant objects to map to distant Euclidean points, we need to construct high-dimensional embeddings. Both Lipschitz embeddings and FastMap are methods for constructing a single, high-dimensional embedding, using simple 1D embeddings as a building block.

In Lipschitz embeddings, we need to choose objects for each reference set. Those objects can be chosen at random, or using some geometric heuristics, like picking objects so that they are far from each other [12], or picking reference objects so as to minimize stress or distortion [3, 13]. In FastMap, we choose pivot pairs using heuristics inspired from Euclidean geometry.

Compared to Lipschitz embeddings and FastMap, BoostMap has two important differences:

- The algorithm produces an embedding explicitly optimized for approximating *rank information*, in the form of approximating the proximity order of triples. This is in contrast to FastMap and Bourgain embeddings, where no quantity is explicitly optimized, and Lipschitz embedding variations that minimize stress or distortion, since optimizing those quantities is not equivalent to directly optimizing for ranking accuracy.

- The optimization method that is used is AdaBoost. The main advantages of Adaboost are its efficiency, and its good generalization properties (validated both in theory and in practice), which make AdaBoost significantly resistant to overfitting [21]. Previous approaches [3, 13] have used simple greedy optimization, which is not as powerful.

In short, BoostMap optimizes what we really want to optimize, and it uses a very powerful optimization method.

6. OVERVIEW OF BOOSTMAP

At a high level, the main points in our formulation are the following:

1. We start with a large family of 1D embeddings. As described in previous sections, this large family can be obtained by defining 1D embeddings based on reference objects and pairs of pivot objects.

2. We convert each 1D embedding into a binary classifier, using Equation 2. These classifiers operate on triples of objects, and they are expected to be pretty inaccurate, but still better than a random classifier (which 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 weighted L_1 distance measure $D_{\mathbb{R}^d}$. It is shown that H is equivalent to the combination of F_{out} and $D_{\mathbb{R}^d}$: 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 $D_{\mathbb{R}^d}$, $F_{\text{out}}(q)$ is closer to $F_{\text{out}}(a)$ than it is to $F_{\text{out}}(b)$.

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 $h_j : \mathcal{X} \rightarrow \mathbb{R}$.

3. Choose $\alpha_j \in \mathbb{R}$.

4. Set training weights $w_{i,j+1}$ for the next round as follows:

$$w_{i,j+1} = \frac{w_{i,j} \exp(-\alpha_j y_i h_j(x_i))}{z_j} . \tag{9}$$

where z_j is a normalization factor (chosen so that $\sum_{i=1}^{t} w_{i,j+1} = 1$).

Output the final classifier:

$$H(x) = \text{sign}\left(\sum_{j=1}^{J} \alpha_j h_j(x)\right) . \tag{10}$$

Figure 3: The AdaBoost algorithm. This description is largely copied from [21].

The key idea is establishing a duality between embeddings and binary classifiers. This duality allows us to convert 1D embeddings to classifiers, combine those classifiers using AdaBoost, and convert the combined classifier into a high-dimensional embedding.

7. CONSTRUCTING EMBEDDINGS VIA ADABOOST

The AdaBoost algorithm is shown in Figure 3. 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.

At an intuitive level, in any training round, the highest training weights correspond to objects that have been misclassified by many of the previously chosen weak classifiers. 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 using highly inaccurate weak classifiers has been demonstrated in numerous applications (for example, in [24, 25]).

In the remainder of this section we will describe how we use AdaBoost to construct an embedding, and how exactly we implement steps 1-4 of the main loop shown in Figure 3.

7.1 Adaptation of AdaBoost

The training algorithm for BoostMap follows the AdaBoost algorithm, as described in Figure 3. The goal of BoostMap is to learn an embedding from an arbitrary space X to d-dimensional Euclidean space $\mathbb{R}^d$. AdaBoost is adapted to the problem of embedding construction 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 Section 10.
- The i-th training triple (q_i, a_i, b_i) is associated with a class label y_i. For BoostMap, $y_i = P_X(q_i, a_i, b_i)$, i.e. y_i is the proximity order of triple (q_i, a_i, b_i), as defined in Equation 1.
- Each weak classifier h_j corresponds to a 1D embedding F from X to $\mathbb{R}$. In particular, $h_j = \tilde{F}$ for some 1D embedding F, where $\tilde{F}$ is defined in Equation 2.

Also, we pass to AdaBoost some additional arguments:

- A set $C \subset X$ of candidate objects. Elements of C will be as reference objects and pivot objects to define 1D embeddings.
- A matrix of distances from each $c \in C$ to each $c \in C$ and to each q_i, a_i and b_i included in one of the training triples in T.

7.2 Evaluating Weak Classifiers

At training round j, given training weights $w_{i,j}$, the weak learner is called to provide us with a weak classifier h_j. In our implementation, the weak learner simply evaluates many possible classifiers, and many possible weights for each of those classifiers, and tries to find the best classifier-weight combination.

We will define two alternative ways to evaluate a classifier h at training round j. The first way is the training error Λ:

$$\Lambda_j(h) = \sum_{i=1}^{t} w_{i,j} G(h, q_i, a_i, b_i) \ , \tag{11}$$

where $G(h, q_i, a_i, b_i)$ is the error of h on the i-th training triple, as defined in Equation 4. Note that this training error is weighted based on $w_{i,j}$, and therefore $\Lambda_j(h)$ will vary with j, i.e. with each training round.

A second way to evaluate a classifier h is suggested in [21]. The function $Z_j(h, \alpha)$ gives a measure of how useful it would be to choose $h_j = h$ and $\alpha_j = \alpha$ at training round j:

$$Z_j(h, \alpha) = \sum_{i=1}^{t} (w_{i,j} \exp(-\alpha y_i h(q_i, a_i, b_i))) \ . \tag{12}$$

The full details of the significance of Z_j can be found in [21]. Here it suffices to say that if $Z_j(\tilde{F}, \alpha) < 1$ then choosing $h_j = h$ 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 $Z_j(h_j, \alpha)$.

Finding the optimal α for a given classifier h, and the Z_j value attained using that α are very common operation in our algorithm, so we will define shorthands for it:

$$A_{\min}(h, j, l) = \mathrm{argmin}_{\alpha \in [l, \infty)} Z_j(h, \alpha) \ . \tag{13}$$

$$Z_{\min}(h, j, l) = \min_{\alpha \in [l, \infty)} Z_j(h, \alpha) \ . \tag{14}$$

In the above equation, j specifies the training round, and l specifies a minimum value for α. $A_{\min}(h, j, l)$ returns the α that minimizes $Z_j(h, \alpha)$, subject to the constraint that $\alpha \geq l$. Argument l will be used to ensure that no classifier has a negative weight. In Section 7.4 we will use classifier weights to define a weighted L_1 distance measure $D_{\mathbb{R}^d}$ in $\mathbb{R}^d$, and non-negative weights ensure that $D_{\mathbb{R}^d}$ is a metric.

7.3 Training Algorithm

At the end of the j-th round, the algorithm has assembled an intermediate classifier $H_j = \sum_{i=1}^{j} \alpha_i h_i$. At a high level, H_j is obtained from H_{j-1} by performing one of the following operations:

- Remove one of the already chosen weak classifiers.
- Modify the weight of an already chosen weak classifier.
- Add in a new weak classifier.

First we check whether a removal or a weight modification would improve the strong classifier. If this fails, we add in a new classifier. Removals and weight modifications that improve the strong classifier are given preference over adding in a new classifier because they do not increase the complexity of the strong classifier.

It is possible that some weak classifier occurs multiple times in H_j, i.e. that there exist $i, g < j$ such that $h_i = h_g$. Without loss of generality we assume that we also have an alternative representation of H_j, such that $H_j = \sum_{i=1}^{K_j} \alpha'_i h'_i$, such that if $g \neq i$ then $h'_g \neq h'_i$. K_j is simply the number of unique weak classifiers occurring in H_j.

Our exact implementation of steps 1-4 from Figure 3 is as follows:

1. Let $z = \min_{c=1,\ldots,K_{j-1}} Z_j(h'_c, \alpha'_c)$.
2. If $z < 1$:
   - Set $g = \mathrm{argmin}_{c=1,\ldots,K_{j-1}} Z_j(h'_c, \alpha'_c)$.
   - Set $h_j = h'_g, \alpha_j = -\alpha'_g$.
   - Go to step 11.

   Comment: If $z < 1$, we effectively remove h'_g from the strong classifier.
3. Let $z = \min_{c=1,\ldots,K_{j-1}} Z_{\min}(h'_c, j, -\alpha'_c)$.
4. If $z < .9999$:
   - Set $g = \mathrm{argmin}_{c=1,\ldots,K_{j-1}} Z_{\min}(h'_c, j, -\alpha'_c)$.
   - Set $h_j = h'_g$.
   - Set $\alpha_j = A_{\min}(h'_g, j, -\alpha'_g)$.
   - Go to step 11.

Comments: Here we modify the weight of h'_g, by adding α_j to it. The third arguments used when calling $Z_{\min}$ and $A_{\min}$ ensure that $\alpha_j \geq -\alpha'_g$, so that $\alpha_j + \alpha'_g$ (which will be the new weight of h'_g in H_j) is guaranteed to be non-negative. Also, note that we check if $z < .9999$. In principle, if $z < 1$ then this weight modification is beneficial. By using .9999 as a threshold we avoid minor weight modifications with insignificant numerical impact on the accuracy of the strong classifier.

5. Choose randomly M_1 reference objects $r_1, \ldots, r_{M_1}$ from the set C of candidate objects. Construct a set $\mathbb{F}_{j1} = \{F^{r_i} | i = 1, \ldots, M_1$ of M_1 1D embeddings using those reference objects, as described in Section 4.1.

6. 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 Equation 8.

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

8. Evaluate $\Lambda_j(h)$ for each $h \in \tilde{\mathbb{F}}_J$, and define a set $\mathbb{H}_j$ that includes the M_2 classifiers in $\tilde{\mathbb{F}}_J$ with the smallest $\Lambda_j(h)$.

9. Set $h_j = \text{argmin}_{h \in \mathbb{H}_j} Z_{\min}(h, (h, j, 0)$.

10. Set $\alpha_j = A_{\min}(h_j, j, 0)$.

    Comment: The third argument to $Z_{\min}$ and $A_{\min}$ in the last two steps is 0. This constrains α_j to be non-negative.

11. Set $z_j = Z_j(h_j, \alpha_j)$.

12. Set training weights $w_{i,j+1}$ for the next round using Equation 9.

In step 8, using a small M_2 reduces training time, because it lets us evaluate $A_{\min}$ only for M_2 classifiers. In general, evaluating the weighted training error Λ_j for a classifier h is faster (by a factor of five to ten in our experiments) than evaluating $A_{\min}$, because in $A_{\min}$ we need to search for the optimal value α that minimizes $Z_j(h, \alpha)$. If we do not care about speed, we should set $M_2 = M_1$ and $M_1 = |C|$.

The algorithm can terminate when we have chosen a desired number of classifiers, or when, at a given round j, we get $z_j \geq 1$, meaning that we have failed to find a weak classifier that would be beneficial to add to the strong classifier.

7.4 Training Output: Embedding and Distance

The output of the training stage is a continuous-output classifier $H = \sum_{c=1}^{d} \alpha_c \tilde{F}_c$, where each $\tilde{F}_c$ is associated with a 1D embedding F_c. This classifier 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 a Euclidean embedding. Here we discuss how to define such an embedding, so that the embedding is as accurate as the classifier H in estimating for any triple (q, a, b) if q is closer to a or to b.

Without loss of generality, we assume that if $c \neq j$ then $\tilde{F}_c \neq \tilde{F}_j$ (otherwise we add α_j to α_c and remove $\tilde{F}_j$ from H.) Given H, we define an embedding $F_{\text{out}} : X \to \mathbb{R}^d$ and a distance $D_{\mathbb{R}^d} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$:

$$F_{\text{out}}(x) = (F'_1(x), ..., F'_d(x)) \ . \tag{15}$$

$$D_{\mathbb{R}^d}((u_1, ..., u_d), (v_1, ..., v_d)) = \sum_{c=1}^{d} (\alpha_c |u_c - v_c|) \ . \tag{16}$$

$D_{\mathbb{R}^d}$ is a weighted Manhattan (L_1) distance measure. $D_{\mathbb{R}^d}$ is a metric, because the training algorithm ensured that all α_c's are non-negative. We should note that, in the implementation used in [2], we had allowed weights to also be negative. By ensuring that $D_{\mathbb{R}^d}$ is a metric, we can apply to the resulting embedding any additional indexing, clustering and visualization tools that are available for metric spaces.

It is important to note that the way we defined F_{out} and $D_{\mathbb{R}^d}$, if we apply Equation 2 to obtain a classifier $\tilde{F}_{\text{out}}$ from F_{out}, then $\tilde{F}_{\text{out}} = H_1$, i.e. the output of AdaBoost. The proof is straightforward:

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

Proof:

$$\begin{aligned}
\tilde{F}_{\text{out}}(q, a, b) &= D_{\mathbb{R}^d}(F_{\text{out}}(q), F_{\text{out}}(b)) - D_{\mathbb{R}^d}(F_{\text{out}}(q), F_{\text{out}}(a)) \\
&= \sum_{c=1}^{d} (\alpha_c \|F_c(q) - F_c(b)\| - \alpha_c \|F_c(q) - F_c(a)\|) \\
&= \sum_{c=1}^{d} (\alpha_c (\|F_c(q) - F_c(b)\| - \|F_c(q) - F_c(a)\|)) \\
&= \sum_{c=1}^{d} (\alpha_c \tilde{F}_c(q, a, b)) = H(q, a, b) \ . \qquad \square
\end{aligned}$$

This equivalence is important, because it shows that the quantity optimized by the training algorithm is exactly the quantity that we set out to optimize in our problem definition, i.e. the classification error $G(\tilde{F_{\text{out}}})$ on triples of objects.

We should note that this equivalence between classifier H and embedding F_{out} relies on the way we define $D_{\mathbb{R}^d}$. If, for example we had defined $D_{\mathbb{R}^d}$ as an L_2 distance, or an unweighted L_1 distance, then the equivalence would no longer hold.

One may ask the following question: what if we had a very accurate classifier H, but H did not correspond to an embedding. Could we use H directly? To answer this question, we should keep in mind that our final goal is a method for producing efficient rankings of all objects in a database, in approximate order of similarity to a query object q. Any classifier H that estimates whether q is closer to a or to b defines (given a query object q) a *partial* order of database objects, based on their estimated similarity to q. However, it is mathematically possible to design classifiers of triples of objects that do not define a *total* order for every q. By proving that classifier H is mathematically equivalent to a Euclidean embedding F_{out}, we guarantee that H defines a total order of database objects based on their similarity to query object q, and therefore H always gives well-defined similarity rankings.

7.5 Complexity

Before we start the training algorithm, we need to compute a matrix of distances from each $c \in C$ to each $c \in C$ and to each q_i, a_i and b_i included in one of the training triples in T. This can often be the most computationally expensive part of the algorithm, depending on the complexity of computing D_X. In addition, at each training round we evaluate M_1 classifiers. Therefore, the computational time per training round is $O(M_1 t)$, where t is the number of training triples. In contrast, FastMap [10], SparseMap [13], and MetricMap [26] do not require training at all.

Computing the d-dimensional embedding of n database objects takes time $O(dn)$. 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 [10], SparseMap [13], and MetricMap [26].

In the experiments, as well as in [2], we see that BoostMap often yields significantly higher-dimensional embeddings than FastMap. In that case, embedding the query object and doing comparisons in Euclidean space is slower for BoostMap. At the same time, in filter-and-refine experiments, BoostMap actually leads to much faster retrieval than FastMap; the additional cost of comparing high-dimensional Euclidean vectors is negligible compared to the savings we get in the refine step, using the superior quality of BoostMap embeddings.

8. QUERY-SENSITIVE EMBEDDINGS

It is often beneficial to generate, using BoostMap, a high-dimensional embedding, with over 100 dimensions. Such high-dimensional embeddings incur the additional cost of comparing high-dimensional Euclidean vectors. At the same time, the additional accuracy we gain in high dimensions can be desirable, and it can even lead to faster overall retrieval by reducing p in a filter-and-refine implementation, as described in Section 4.3. This effect is demonstrated in experiments with BoostMap, both in [2] and in this paper.

However, even though producing a high-dimensional embedding can be beneficial, finding nearest neighbors in high dimensions also poses the following problems, as pointed out in [1]:

- **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 in comparing those objects.
- **Skew magnification:** Many attributes may be correlated with each other.

BoostMap produces a high-dimensional embedding that preserves more of the proximity structure of the original space, as compared to lower-dimensional embeddings. In that sense, distances measured in high dimensions are more meaningful than distances measured in low dimensions, since our goal is accurate similarity rankings with respect to distances in the original non-Euclidean space. At the same time, the three problems outlined above are still present, in the sense that we can achieve even better accuracy by addressing those problems in our formulation.

To address those problems, we extend the BoostMap algorithm so that it produces a *query-sensitive* distance measure. By "query-sensitive" we mean that the weights used for the weighted L_1 distance will not be fixed, as defined in Equation 16. Instead, they will depend on the query object. An automatically chosen query-sensitive distance measure provides a principled way to address the three problems described in [1], by putting more emphasis on coordinates that are more important for a particular query, and at the same time setting each weight while taking into account the effects of all other weights.

8.1 Learning a Query-Sensitive Classifier

Learning a query-sensitive distance measure is still done within the framework of AdaBoost, using a simplified version of the alternating decision-tree algorithm, described in [11]. As described earlier, every 1D embedding F corresponds to a classifier $\tilde{F}$, that classifiers triples (q, a, b) of objects in X. The key idea in defining query-sensitive distance measures is that $\tilde{F}$ may do a good job only when q is in a specific region, and it is actually beneficial to ignore $\tilde{F}$ when q is outside that region. In order to do that, we need another classifier $S(q)$ (which we call a *splitter*), that will estimate, given a query q, whether $\tilde{F}$ is useful or not.

More formally, if X is the original space, suppose we have a splitter $S : X \to \{0, 1\}$ and a 1D embedding $F : X \to \mathbb{R}$. We define a *query-sensitive classifier* $\tilde{Q}_{S,F} : X^3 \to \mathbb{R}$, as follows:

$$\tilde{Q}_{S,F}(q, a, b) = S(q)\tilde{F}(q, a, b) \ . \tag{17}$$

We say that the splitter S *accepts* q if $S(q) = 1$, and S *rejects* q if $S(q) = 0$.

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:

$$S_{F,V}(q) = \begin{cases} 1 & \text{if } F(q) \in V \ . \\ 0 & \text{otherwise} \ . \end{cases} \tag{18}$$

We will use the notation $\tilde{Q}_{F_1,V,F_2}$ for the query-sensitive classifier that is based on $S_{F_1,V}$ and F_2:

$$\tilde{Q}_{F_1,V,F_2}(q, a, b) = S_{F_1,V}(q)\tilde{F}(q, a, b) \ . \tag{19}$$

Suppose that the algorithm described in Section 7 has produced a d-dimensional embedding $F_{\text{out}} : X \to \mathbb{R}^d$, such that $F_{\text{out}}(x) = (F_1(x), \dots, F_d(x))$. We will introduce a second training phase, that starts after F_{out} has been produced, and adds a query-sensitive component to $\tilde{F}_out$. This query-sensitive component will be a combination of query-sensitive classifiers $\tilde{Q}_{F_c,V,F_g}$, where F_c and F_g are parts of F_{out}, and $V \subset \mathbb{R}$.

Let J be the number of training rounds it took to produce classifier H as described in Section 7, and let $H = \sum_{c=1}^{d} \alpha_c \tilde{F}_c$. For training round $j > J$ (i.e. for a training round of the second training phase, that builds the query-sensitive component), we perform the following steps:

1. For each $c = 1, \dots, d$, pick g randomly from $1, \dots, d$, so that with 0.5 probability $g = c$. Define $\Gamma_c = F_g$.
2. For each $c = 1, \dots, d$, define a set $\mathbb{V}_c$ of subsets of $\mathbb{R}$, such that each $V \in \mathbb{V}_c$ is of the form $\mathbb{R}$, $(-\infty, t)$, (t, ∞), (t_1, t_2), $(-\infty, t_1) \cup (t_2, \infty)$ (where t, t_1, t_2 are real numbers).

   Comment: Each $V \in \mathbb{V}_c$ will be combined with Γ_c to define a splitter. Choosing sets V can be done by looking at the set of values $\{\Gamma_c(q_i) : i = 1, \dots, t\}$, where q_i is the first object of the i-th training triple, and picking thresholds t, t_1, t_2 randomly from those values.
3. For each $c = 1, \dots, d$, set:
   $V_c = \text{argmin}_{V \in \mathbb{V}}(Z_{\min}(\tilde{Q}_{\Gamma_c,V,F_c}, j, 0))$.

Comment: here, given 1D embeddings Γ_c and F_c, we find the range $V_c \in \mathbb{V}$ that leads to the best classifier $\tilde{Q}_{\Gamma_c, V, F_c}$ (i.e. the classifier that attains the lowest Z_j value).

4. Set $g = \text{argmin}_{c=1,\ldots,d}(Z_{\min}(\tilde{Q}_{\Gamma_c, V_c, F_c}, j, 0))$.

   Comment: here we find the $g \in \{1, \ldots, d\}$ for which the corresponding classifier $\tilde{Q}_{\Gamma_g, V_g, F_g}$ is the best.

5. Set $h_j = \tilde{Q}_{\Gamma_g, V_g, F_g}$.

6. Set $\alpha_j = A_{\min}(h_j, j, 0)$.

   Comment: Note that the third argument to $A_{\min}$ and $Z_{\min}$ in all steps has been 0. This constraints α_j to be non-negative.

7. Set $z_j = Z_j(h_j, \alpha_j)$.

8. Set training weights $w_{i,j+1}$ for the next round using Equation 9.

   Comment: the last three steps are identical to the last three steps of the algorithm in Section 7.3.

Note that the first round of the second training phase, which is training round $J + 1$ overall, uses training weights $w_{i,J+1}$, as they were set by the last training round (round J) of the first training phase (which was described in Section 7).

If H was the output of the first training phase, and if the second training phase was executed in training rounds $J + 1, \ldots, J_2$, we write the output H_2 of the second training phase as follows:

$$H_2 = H + \sum_{c=J+1}^{J_2} \alpha_c h_c \ . \tag{20}$$

We expect H_2 to be more accurate than H, because it includes query-sensitive classifiers, each of which is focused on a specific region of possible queries. In particular, for each classifier $\tilde{F}_c$ used in H, the second training phase tries to identify subsets of queries where $\tilde{F}_c$ is particularly useful, and increases the weight of $\tilde{F}_c$ for those queries.

8.2 Defining a Query-Sensitive Embedding

In Section 7.4 we used H to define an embedding F_{out}, that maps objects of X into $\mathbb{R}^d$, and a distance $D_{\mathbb{R}^d}$, such that $\tilde{F}_{\text{out}}$ was equivalent to H. Now that we have constructed H_2, we also to define an embedding and a distance based on H_2. The embedding associated with H_2 is still F_{out}, since H_2 only uses 1D embeddings that also occur in H. On the other hand, we cannot define a global distance measure in $\mathbb{R}^d$ anymore that would make $\tilde{F}_{\text{out}}$ equivalent to H_2. To achieve this equivalence between $\tilde{F}_{\text{out}}$ and H_2, we define a query-sensitive distance D_q, that depends on the query object q. First, we define an auxiliary function $A_c(q)$, which assigns a weight to the c-th coordinate, for $c = 1, \ldots, d$:

$$A_c(q) = \alpha_c + \sum_{g: g \in \{J+1,\ldots,J_2\} \wedge h_g = \tilde{Q}_{S,F_c} \wedge (S(q)=1)} \alpha_g \ . \tag{21}$$

In words, for coordinate c, we go through all the query-sensitive weak classifiers that were chosen at the second training phase. Each such query-sensitive classifier h_g can be written as $\tilde{Q}_{S,F}$. We check if the splitter S accepts q, and if $F = F_c$. If those conditions are satisfied, we add the weight α_g to α_c.

If $F_{\text{out}}(q) = (q_1, ..., q_d)$, and x is some other object in X, with $F_{\text{out}}(x) = (x_1, ..., x_d)$, we define query-sensitive distance D_q as follows:

$$D_q((q_1, ..., q_d), (x_1, ..., x_d)) = \sum_{c=1}^{d} (A_c(q)|q_c - x_c|) \ . \tag{22}$$

Now, using this distance D_q, with a slight modification of Equation 2, we can define $\tilde{F}_{\text{out},2}$ in such a way that $\tilde{F}_{\text{out},2} = H_2$:

$$\begin{aligned} \tilde{F}_{\text{out},2}(q, x_1, x_2) \ &= \ D_q(F_{\text{out}}(q), F_{\text{out}}(x_2)) - \\ &\quad\ D_q(F_{\text{out}}(q), F_{\text{out}}(x_1)) \ . \end{aligned} \tag{23}$$

The only difference from Equation 2 is that here we use the query-sensitive distance measure D_q, as opposed to a global distance measure $D_{\mathbb{R}^d}$.

We omit the proof that $\tilde{F}_{\text{out},2} = H_2$, it is pretty straightforward and follows the same steps as the proof of Proposition 1. The fact that $\tilde{F}_{\text{out}}$ and H_2 establishes that, if the query-sensitive classifier H_2 is more accurate than classifier H, then we using the query-sensitive distance D_q will lead to more accurate results. This is demonstrated in our experiments.

8.3 Complexity of Query-Sensitive Embeddings

To learn a query-sensitive embedding we have to perform additional training using AdaBoost. The actual number of classifiers $\tilde{Q}$ we can define using embeddings in H_1 can be quite large, since we can form such a classifier for each pair of embeddings occurring in H_1 and each choice of a range R. We can keep training time manageable because of two reasons:

- We noted that, in early implementations, AdaBoost tended to choose query sensitive classifiers $\tilde{(Q)}_{F,F,R}$, i.e. classifiers that used the same 1D embedding twice F. The interpretation of that is that $F(q)$ tends to contain significant information about whether $\tilde{F}$ is useful on triples of type (q, a, b). Based on that observation, in our current implementation we have AdaBoost consider all classifiers $\tilde{(Q)}_{S,F}$ in which S is defined using F, and an equal number of classifiers (chosen randomly) where S is *not* defined using F.

- For each pair of 1D embeddings F_1 and F_2, we use that pair to define a large number of $\tilde{(Q)}_{S,F_2}$ classifiers, by defining a splitter S based on F_1 and any of a large number of possible ranges R. However, the training errors of all those classifiers $\tilde{(Q)}_{S,F_2}$ are related, since the only thing that is different among those classifiers is the range $R \in \mathbb{R}$ of the splitter S. If r is the number of ranges we are willing to consider, and t is the number of training triples used in AdaBoost, we can measure the training errors of all ranges in time $O(t + r)$, as opposed to the time $O(tr)$ it would take if we evaluated all those errors independently of each other. Based on that, at each training round, given F_1 and F_2 we can quickly evaluate a large number of ranges and choose the range that has the smallest training error.

At retrieval time, given a query, using a query-sensitive distance measure incurs negligible additional cost over using a query-insensitive distance measure. The only additional computation we need is in order to compute the $A_c(q)$ values, i.e. the query-specific weights of each coordinate j. This can be done by scanning the classifier H_2 once. The size of classifier H_2 in practice is comparable to the dimensionality of the embedding. The total cost of this scanning is marginal compared to the cost of evaluating distances between the embedding of q and the embedding of each database object.

9. OPTIMIZING EMBEDDINGS FOR CLASSIFICATION

When we have a database of objects in some non-Euclidean or even non-metric space X, Euclidean embeddings can be used for indexing, in order to efficiently identify the nearest neighbors of a query in the original space X. However, in many applications, our ultimate goal is not retrieving nearest neighbors, but actually *classifying* the query, using the known class information of the query's nearest neighbors. This section describes how we can optimize embeddings directly for classification accuracy.

9.1 Hidden Parameter Space

As elsewhere in this paper, let X be an arbitrary space, in which we define a (possibly non-metric) distance D_X. Here we also assume that there is an additional distance defined on X, which we denote as Φ_X. We will call D_X the *feature space distance* and we will call Φ_X the *hidden parameter space distance*, or simply the parameter space distance.

Our experimental dataset, the MNIST database, provides an example. The MNIST database consists of 60,000 training images and 10,000 test images of handwritten digits. Some of those images are shown in Figure 4. Each image shows one of the 10 possible digits, from 0 to 9. One can define various distance measures on this space of hand images, like the non-metric *chamfer distance* [5], or *shape context* [6]. Given a query image q , we want to find the nearest neighbor (or k-nearest neighbors, for some k) of the query, and using the class labels of those neighbors we want to classify the query.

In this case, the feature space distance D_X is a distance measure like the chamfer distance or shape context, that depends on object features. Given two objects, we can always observe those features, and therefore we can always evaluate D_X. The hidden parameters in this case are the class labels of the objects, which we cannot directly observe but we want to estimate. The hidden parameters are known for the database objects (the training images), but not for the query objects. We define the hidden parameter space distance Φ_X between two objects $x, y \in X$ to be 0 if those two objects have the same class labels (i.e. are pictures of the same digit), and 1 if the two objects have different class labels.

There are also domains where Φ_X is not a binary distance. For example, a dataset in which we evaluated the original BoostMap algorithm consisted of hand images [2]. In those images, we used the chamfer distance as the feature space distance, but the goal was to actually estimate the hand pose in the query image. Hand pose is a continuous space, defined by articulated joint angles and global 3D orientation of the hand. In this case one can define a distance Φ_X between hand poses. Since hand poses are not known for the query images, they are hidden parameters.

When our goal is to estimate the hidden parameters of the query object, the only use of feature space distance measure D_X is that lets us perform this estimation, using nearest-neighbor classification. Suppose now that we map database objects into a Euclidean space, using BoostMap for example, and we have a choice of two distance measures, D_1 and D_2, to use in the Euclidean space. For illustration purposes, let's make an extreme assumption that D_1 perfectly preserves distances D_X, and D_2 is a bad approximation of D_X, but it leads to higher classification accuracy than D_X (and therefore than D_1). In that case, if classification is our goal, D_2 would be preferable over D_1.

9.2 Tuning BoostMap for Classification

Euclidean embeddings, like FastMap, MetricMap, Lipschitz embeddings, and BoostMap, provide a Euclidean substitute for the feature space distance D_X, which itself is a substitute for the hidden parameter space distance Φ_X, which cannot be evaluated exactly. Therefore, the distance measure used in the Euclidean space is two levels of approximation away from Φ_X, which is the measure we really want to estimate. However, since BoostMap is actually trained using machine learning, we can easily modify it so that it is directly optimized for classification, i.e. for recovering Φ_X.

Optimizing BoostMap for classification accuracy is pretty straightforward. As described in the overview of the training algorithm, each training triple (q_i, a_i, b_i) has a class label $y_i = P_X(q, x_1, x_2)$. Note that the definition of P_X in Equation 1 is based on an underlying distance measure D between objects of X. If our goal is approximating D_X, then we set $D = D_X$. If our goal is nearest-neighbor classification, we use $D = \Phi_X$.

If we have a training triple (q_i, a_i, b_i) where q is closer to b using measure D_X but closer to a using measure Φ_X, then defining class label y_i using Φ_X will steer the training algorithm towards trying to produce an embedding F_{out} and a distance D_q such that $D_q(F_{\text{out}}(q), F_{\text{out}}(a)) < D_q(F_{\text{out}}(q), F_{\text{out}}(b))$. Overall, the training algorithm will try to map objects from the same class closer to each other than to objects of other classes.

10. CHOOSING TRAINING TRIPLES

In the original implementation and experimental evaluation of BoostMap in [2], training triples were chosen at random. With 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 an object q, we should form a triple (q, a, b) where both a and b are relatively close to q.

In our experiments with the MNIST database, having in mind that we also wanted to optimize embeddings for classification accuracy, we choose training triples as follows: first, we specify the desired number t of training triples to produce, and an integer k' that specifies up to how "far" a_i and b_i can be from q_i in each triple (q_i, a_i, b_i). Then, we choose the i-th training triple (q_i, a_i, b_i) as follows:

1. Choose a random training object q_i.
2. Choose an integer k in $1, \ldots, k'$.
3. Choose a_i to be the k-nearest neibhor of q_i among all training objects for which $\Phi(a_i, q_i) = 0$. This way a_i has the same class label as q_i.
4. Choose a number r in $9k, \cdots, 9k + 9$. Note that 10 is the number of classes in the MNIST database, and 9 is the number of classes that are different than the class of q_i.
5. Choose b_i to be the r-nearest neighbor of q_i among all training objects whose class label is different than the class label of q_i.

Essentially, each training triple contains an object q, one of its nearest neighbors among objects of the same class as q, and one of the nearest neighbors among objects of all classes different than the class of q. If we did not care about classification, we could simply have chosen random triples such that a and b are among the k' nearest neighbors of q.

The reason we choose r to be roughly 9 times bigger than k is that, with that choice, it is reasonable to assume that most of the times $D_X(q_i, a_i)$ will be smaller than $D_X(q_i, b_i)$. In general, if M is the number of classes, even if D_X carries zero information about Φ_X, we still expect that on average the k-nearest neighbor of q among same-class objects will have the same rank as the $k(M-1)$-nearest neighbor of q among objects that belong to different classes. For this assumption to hold, we just need to have the same number of objects in each class, and D_X to be not worse than a random distance measure for k-nearest neighbor classification. These are very weak assumptions. At the same time, if we learn an embedding that, for a large percentage of q objects and k values, maps q closer to its k-nearest neighbor among same-class objects than to the $k(M-1)$-nearest neighbor among objects of different classes, then we expect that embedding to lead to high nearest-neighbor classification accuracy. So, setting $r = M - 1$ we expect the weak classifiers to be better than random classifiers, and the accuracy of the strong classifier on triples is related to k-nn classification accuracy on query objects using the embedding.

If our goal is not classification, but simply to provide accurate indexing for nearest neighbor retrieval, the method outlined above for choosing training triples would still be useful. In each triple (q, a, b), a and b are both relatively close to q, and therefore the training set of triples biases the training algorithm to focus on preserving k-nearest-neighbor structure, for small values of k, as opposed to preserving similarity structure in general.

11. EXPERIMENTS

We compared BoostMap to FastMap [10] on the MNIST dataset of handwritten digits, which is described in [16], and is publicly available on the web. This dataset consists of 60,000 training images, which we used as our database, and 10,000 images, which we used as queries. Some of those images can be seen in Figure 4. We used the symmetric chamfer distance [5] as the underlying distance measure. The chamfer distance is non-metric, because it violates the triangle inequality.

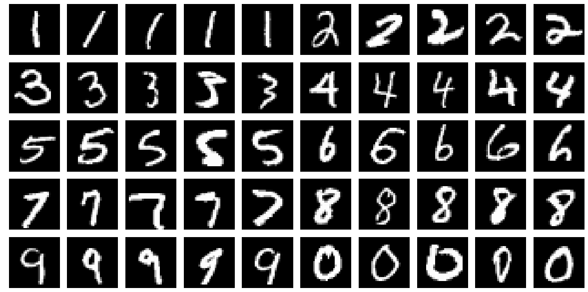

Figure 4: Some examples from the MNIST database of handwritten images.

For BoostMap, we always used 200,000 triples for training. The objects appearing in those triples came from a set of 5000 database objects. The size of C, the set of candidate objects defined in Section 7.1, was also 5000. We used $M_1 = 1000$, and $M_2 = 200$. FastMap was run on a distance matrix produced using 10,000 training objects.

For BoostMap we have tested different variants, in order to evaluate the advantages of the three extensions introduced in this paper: choosing training triples in a selective way, vs. choosing them randomly, using a query-sensitive distance measure vs. using a global distance measure in Euclidean space, and optimizing BoostMap for classification vs. optimizing BoostMap for preserving proximity structure. To denote each of these variants, we use the following abbreviations:

Fe Embedding optimized for approximating feature-space distances D_X, as opposed to **Pa**.

Pa Embedding optimized for approximating parameter-space distances Φ_X, i.e. for classification accuracy, as opposed to **Fe**.

QI Query-insensitive distance measure $D_{\mathbb{R}^d}$ is used, as defined in Section 7.4. The alternative to **QI** is **QS**.

QS Query-sensitive distance measure D_q is used, as opposed to **QI**.

Ra Training triples are chosen entirely randomly from the set of all possible triples, as opposed to **Se**.

Se Training triples are chosen selectively, from a restricted set of possible triples, as described in Section 10. The alternative to **Se** is **Ra**.

For example, BoostMap-Fe-Se-QS means that the embedding was optimized for approximating feature-space distances, we chose training triples selectively, and we used a query-sensitive distance measure.

11.1 Measures of Embedding Accuracy

To evaluate the accuracy of the approximate similarity ranking for a query, we use a measure that we call exact k-nearest neighbor rank (ENN-k rank), defined as follows: given query object q, and integer k, let $b_1, \ldots, b_k$ be the

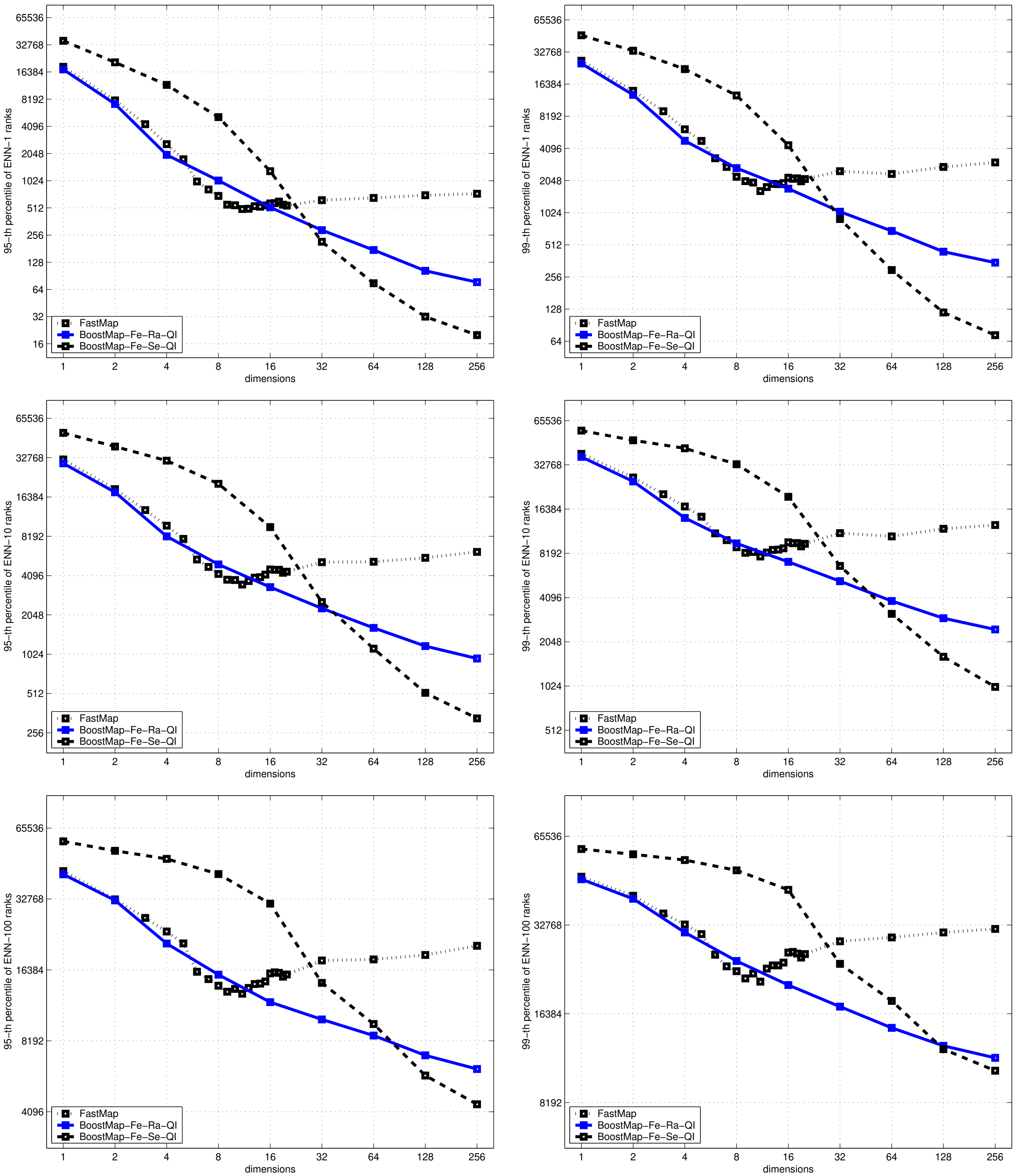

Figure 5: Plots of the 95th and 99th percentile of ENN-k ranks attained by FastMap, BoostMap-Fe-Ra-QI, and BoostMap-Fe-Se-QI, for different embedding dimensions, for k = 1, 10, 100.

k-nearest neighbors of q in the database, under the exact distance D_X. Given an embedding F, the rank of any b_i under the embedding is defined to be one plus the number of database objects that F maps closer to $F(q)$ than $F(b_i)$ is to $F(q)$. Then, the ENN-k rank for q under embedding F is the worst rank attained by any object in $b_1, \ldots, b_k$.

For example, for $k = 10$, an ENN-10 rank of 150 using BoostMap and 16 dimensions means that all 10 exact k-nearest neighbors of q were within the 150 nearest neighbors of the query, as computed using a 16-dimensional BoostMap embedding. Using filter-and-refine retrieval, if we keep 150 or more candidates after the filter step, we will successfully identify all 10 nearest neighbors of q at the refine step.

If we have a set of queries, then we can look at different percentiles of the ENN-k ranks attained for those queries. For example, given embedding F, a value of 3400 for the 95th percentile of ENN-10 ranks means that, for 95% of the 10,000 query objects, the ENN-10 rank was 3400 or less.

Another measure of accuracy for an embedding F is simply the k-nearest neighbor classification error using F. Given query object q, we identify the k-nearest neighbors of $F(q)$ in the embedding of the database. Each of those k objects gives a vote for its class label. The class label that receives the most votes is assigned to q. If two or more classes receive the same number of votes, we find, for each class y among those classes, the database object $x_{y,q}$ that belongs to class y and is the nearest to q, and we choose the class y for which the corresponding $x_{y,q}$ is the closest to q. If there is still a tie, we break it by choosing at random.

11.2 BoostMap vs. FastMap

Figure 5 shows the 95th and 99th percentiles of ENN-k attained by FastMap, BoostMap-Fe-Ra-QI, and BoostMap-Fe-Se-QI, for different embedding dimensions, for k = 1, 10, 100. We note that in fewer than 16 dimensions, FastMap sometimes gives the best results. From 16 dimensions and on, BoostMap-Fe-Ra-QI (which is essentially the BoostMap variant described in [2]) gives better results than FastMap. BoostMap-Fe-Se-QI does worse for lower dimensions, but at 256 dimensions it gives the best results in all cases. These facts also hold for other values of k (up to 100) and other percentiles (from 80% to 99%) that we have checked.

The results demonstrate that, in a filter-and-refine framework, using BoostMap-Fe-Ra-QI we typically need to keep significantly fewer candidate matches after the filter step, and overall we can find the correct top k nearest neighbors evaluating far fewer exact distances D_X, compared to using FastMap. As k and the percentile increase, at some point it becomes beneficial to use BoostMap-Fe-Se-QI.

For example, if we want to retrieve the true 10 nearest neighbors ($k = 10$) for 98% of the query objects, these are the optimal results we get for the three different methods:

FastMap: We get the best result for 11 dimensions, and keeping 5523 database objects after the filter step. We need to compute 22 distances D_X to embed each query, and 5523 distances D_X to find the 10 nearest neighbors. In total, we compute 5545 D_X distances.

BoostMap-Fe-Ra-QI: We get the best result for 256 dimensions, and keeping 1698 database objects after the filter step. We need to compute at most 512 distances D_X to embed each query (for some dimensions we need one D_X evaluation, for some dimensions we need two D_X evaluations), and 1698 distances D_X to find the 10 nearest neighbors. In total, we compute at most 2210 D_X distances.

BoostMap-Fe-Se-QI: We get the best result for 256 dimensions, and keeping 637 database objects after the filter step. We need to compute at most 512 distances D_X to embed each query, and 637 distances D_X to find the 10 nearest neighbors. In total, we compute at most 1149 D_X distances.

		BoostMap-Fe-QI		BoostMap-Fe-QS	
k	percentile	random	selective	random	selective
1	95	77	20	38	20
1	99	349	73	136	62
10	95	949	330	502	273
10	99	2483	1010	1302	675
100	95	6220	4406	3773	3215
100	99	11617	10508	7437	7333

Table 1: Comparison of the two methods for choosing training triples: sampling them from the set of all possible triples vs. choosing them from a selective subset of triples. For 256-dimensional embeddings, we show the 95th and 99-th percentiles of ENN-k ranks, for k = 1, 10, 100.

In domains where computing D_X distances is the computational bottleneck, and computing distances in 256-dimensional Euclidean space is relatively fast, results like the above mean that BoostMap leads to significantly more efficient filter-and-refine retrieval.

At this point, we have only trained 256-dimensional query-sensitive embeddings, so we do not have enough data to include query-sensitive embeddings to the plots of Figure 5. Later in this section we show that using query-sensitive embeddings further improves embedding quality.

11.3 Random vs. Selective Training Triples

Figure 5 shows the ENN-k ranks attained by BoostMap-Fe-Se-QI vs. BoostMap-Fe-Ra-QI for different percentiles. We note that, for lower dimensions, choosing training triples from a restricted set seems to lead to less accurate embeddings. On the other hand, at 256 dimensions, choosing triples selectively leads to more accurate embeddings.

One possible interpretation of these results is that, by choosing triples selectively, the training algorithm optimizes the embedding so that it is highly accurate on those triples, but not necessarily on other triples. If each training triple (q, a, b) is such that a_i and b_i are close to q_i, the training will not consider triples (q, a, b') where b is farther away from q (for example, cases where b' is not in the 1000 nearest neighbors of q).

For example, suppose that we want to retrieve the 10 nearest neighbors $a_1, \ldots, a_{10}$ of q in the original space X with distance measure D_X. An ideal embedding F_{ideal} would map q closer to those 10 neighbors than to any other object. The ENN-10 rank that is achieved by an embedding F for object q increases because of objects b_i such that $F(q)$ is closer to b_i than it is to one of the ten nearest neighbors a_i. Choosing training triples (q, a, b) so that b is, say, within the 1000 nearest neighbors of q, we make the implicit assumption that objects outside the 1000 nearest neighbors of q will not cause

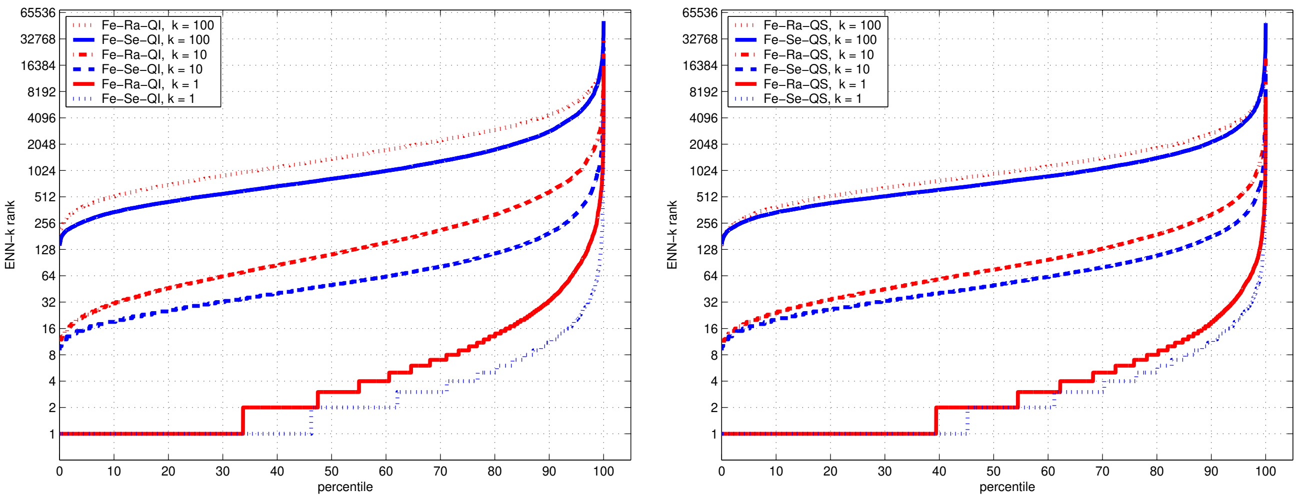

Figure 6: Comparison of the two methods for choosing training triples: sampling them from the set of all possible triples vs. choosing them from a selective subset of triples. We plot ENN-k ranks vs. percentile, for 256-dimensional embeddings, for $k = 1$, 10, 100. On the left we show query-insensitive embeddings, on the right we show query-sensitive embeddings. In all cases, and for all percentiles, choosing triples selectively leads to better results.

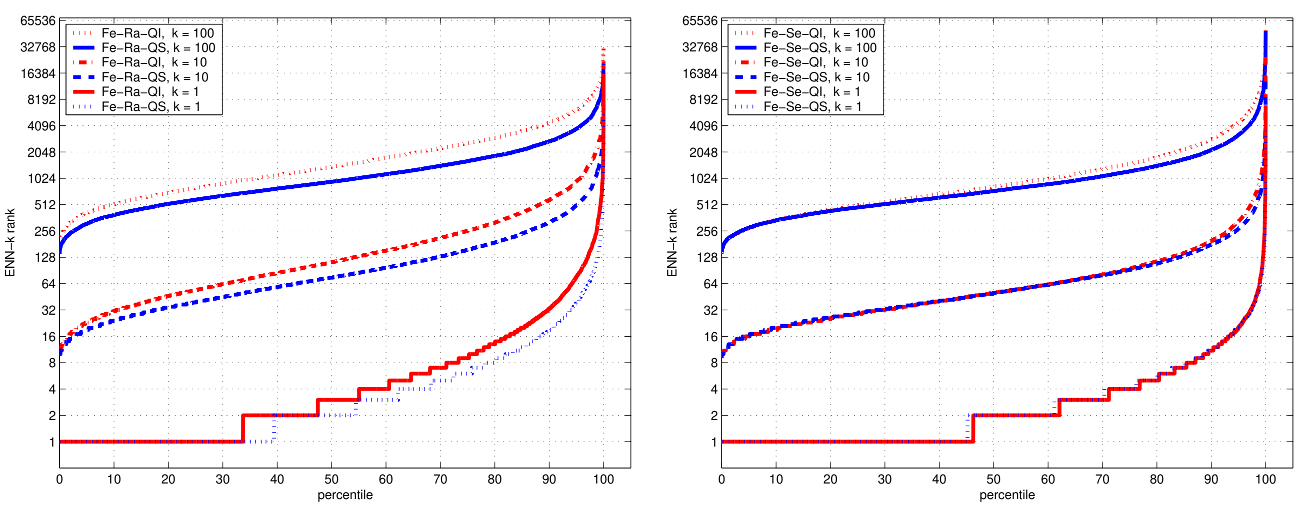

Figure 7: Comparison of query-insensitive versus query-sensitive embeddings. We plot ENN-k ranks vs. percentile, for 256-dimensional embeddings, for $k = 1$, 10, 100. On the left we show embeddings learned using random training triples, on the right we show embeddings learned using selective training triples. In most cases query-sensitive embeddings give similar or better results, compared to query-insensitive embeddings, when all other settings are fixed.

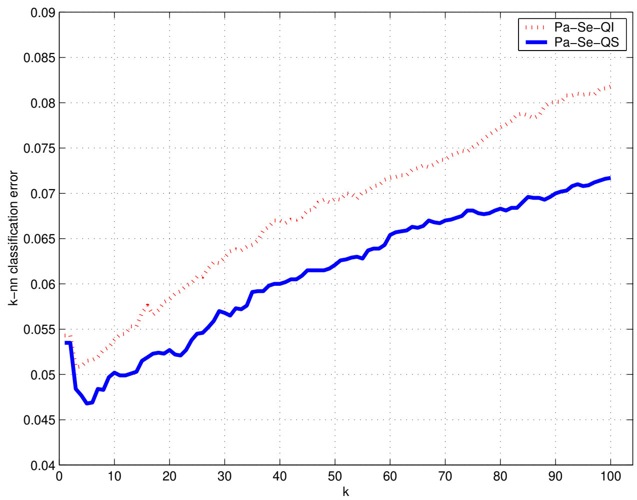

Figure 8: K-nearest neighbor classification error using query-insensitive and query-sensitive embeddings. For $k = 1, \ldots, 100$ we show the corresponding k-nn error rate.

problems, i.e. we expect that the embedding F will not map q closer to any of those distant objects than to q's 10 nearest neighbors. That's why we want the training algorithm to focus more on triples where b is close to q.

This assumption is obviously violated in lower-dimensional embeddings, which are not very accurate and they can map distant objects close to each other. In those cases, choosing random triples forces the training algorithm to try to preserve the overall proximity structure of the space, whereas choosing triples so that a and b are close to q means that the training algorithm does not penalize choices that map distant objects close to each other.

In high dimensions, it is much more rare for distant objects to map close to each other, and then the main source of indexing errors is objects that are somewhat close to q. Training BoostMap with selective triples optimizes the embedding so as to avoid that type of indexing errors, so overall we get higher embedding quality.

At this point, this interpretation is just a hypothesis. We need additional experiments, in which we vary the parameter k' (defined in Section 10) that specifies how close a and b are to q for each training triple. If larger k' values lead to higher accuracy in lower dimensions, that would provide supporting evidence for our interpretation. In the experiments reported here, we used $k' = 4$.

To demonstrate the advantages of choosing triples selectively for high-dimensional embeddings, we compare the two methods of choosing training triples in Figure 6 and Table 1. The results demonstrate that, in 256 dimensions, choosing triples selectively leads to better embedding quality.

11.4 Query-Sensitive vs. Query-Insensitive Embeddings

To evaluate the benefits of query-sensitive embeddings (i.e. BoostMap embeddings that use query-sensitive distance measures) we trained, for different settings, query-sensitive 256-dimensional embeddings. Figure 7 and Table 2 compare query sensitive embeddings to query-insensitive

		BoostMap-Fe-Ra		BoostMap-Fe-Se	
k	percentile	random	selective	random	selective
1	95	77	38	20	20
1	99	349	136	73	62
10	95	949	502	330	273
10	99	2483	1302	1010	675
100	95	6220	3773	4406	3215
100	99	11617	7437	10508	7333

Table 2: Comparison of the two methods for choosing training triples: sampling them from the set of all possible triples vs. choosing them from a selective subset of triples. For 256-dimensional embeddings, we show the 95th and 99-th percentiles of ENN-k ranks, for $k = 1$, 10, 100.

embeddings, based on different percentiles of ENN-k ranks. We see that, in most cases, the query-sensitive variants give similar or better results than the query-insensitive variants, and in some cases the results are significantly better.

We also evaluate query-sensitive embeddings based on the k-nn classification error rate attained using embeddings optimized for classification (parameter-space embeddings). Figure 8 shows the corresponding results for 256-dimensional parameter-space embeddings. For all values of k that we tested, the query-sensitive embedding had lower error rate than the query-insensitive embedding.

11.5 Parameter-Space vs. Feature-Space Embeddings

As discussed in Section 9, we expect parameter-space embeddings to be worse than feature-space embeddings with respect to preserving the feature-space distance D_X, but at the same time we expect parameter-space embeddings to give higher classification accuracy than feature-space embeddings. Figure 9 shows percentiles of ENN-k ranks, and Figure 10 shows the k-nn error rates achieved for different values of k, for feature-space and parameter-space embeddings. These results agree with our expectations.

Figure 11 and Table 3 compare the classification error rates achieved using the original chamfer distance and using a 256-dimensional parameter-space query-sensitive embedding. It is interesting to note that for most values of k the embedding actually achieves a lower error rate than the original distance measure. The chamfer distance achieves the best overall error rate, but it is only marginally better than the best error rate achieved using the embedding: for $k = 5$, the chamfer distance misclassified 463 images and the embedding misclassified 468 images, out of 10,000 objects.

In domains where we get such results, we actually do not need to apply filter-and-refine retrieval in order to do k-nn classification, since we get equally good results using nearest neighbors in the Euclidean space. When computing distances in the original space is the computational bottleneck, using a parameter-space embedding can speed up recognition significantly, since we only need to compute a few hundreds of distances in the original space, in order to compute the embedding of the query object.

FastMap, MetricMap, SparseMap, Bourgain embeddings, and the original formulation of BoostMap, usually provide approximations of an original distance measure, that lead to more efficient distance computations but less accuracy. Us-

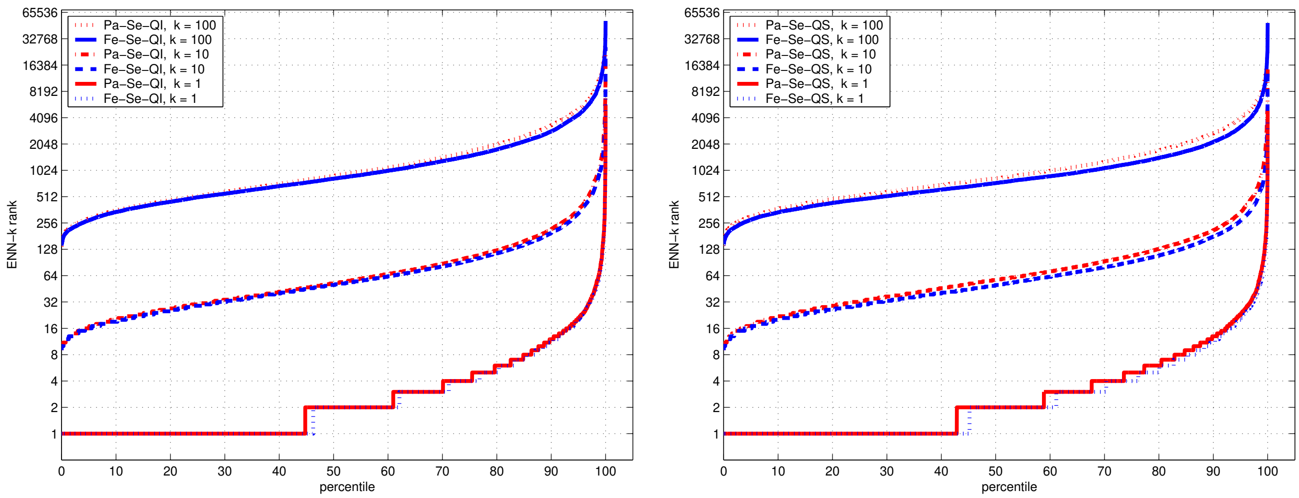

Figure 9: Comparison of feature-space versus parameter-space embeddings, with respect to ENN-k ranks. We plot ENN-k ranks vs. percentile, for 256-dimensional embeddings, for $k = 1$, 10, 100. On the left we show query-insensitive embeddings, on the right we show query-sensitive embeddings. Feature-space embeddings give somewhat better results for ENN-k ranks.

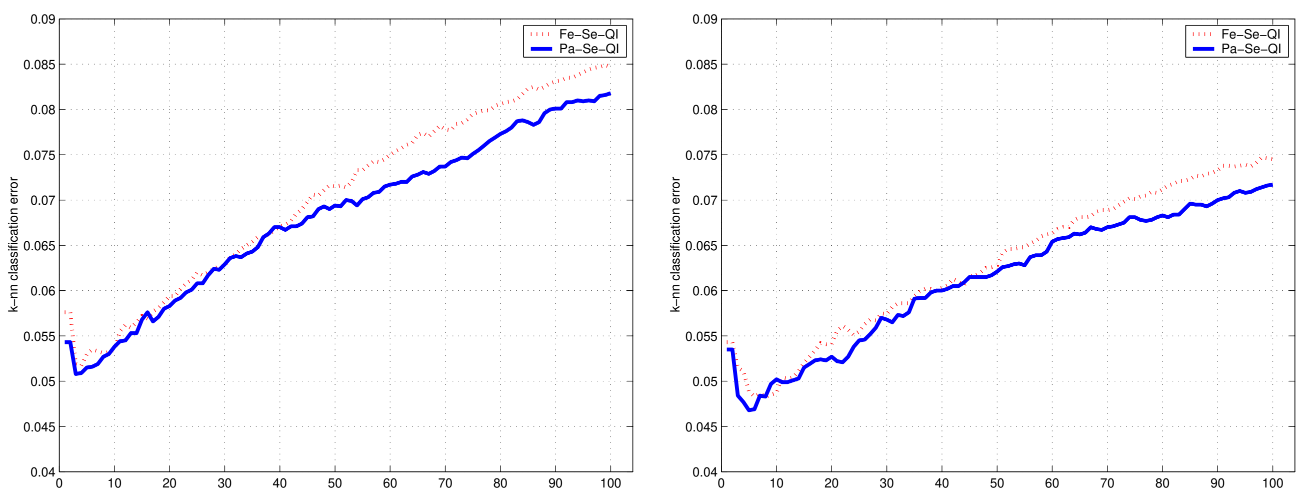

Figure 10: Comparison of feature-space versus parameter-space embeddings, with respect to k-nn classification error rate. We plot k-nn error rate versus k. On the left we show query-insensitive embeddings, on the right we show query-sensitive embeddings. Parameter-space embeddings give lower error rates.

ing parameter-space BoostMap embeddings it is possible in some domains to obtain both faster distance measures and higher classification accuracy. For the MNIST database and using the chamfer distance, we get a Euclidean approximation of the chamfer distance that achieves similar accuracy. It will be interesting to see if we get similar results in other domains.

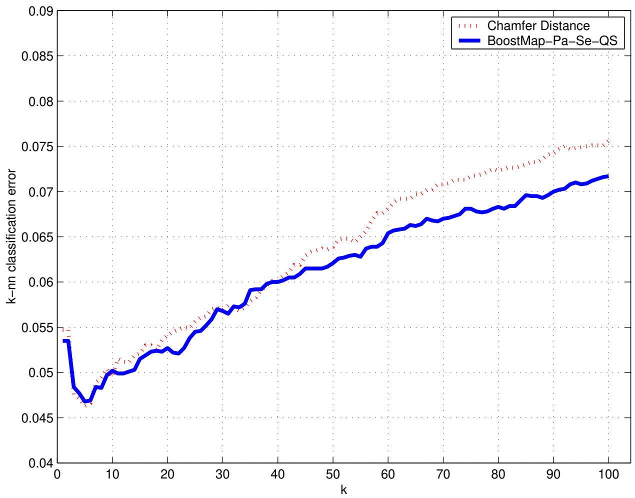

Figure 11: Comparison of k-nn error rates using original distance measure, and using a 256-dimensional parameter-space query-sensitive embedding. We plot k-nn error rate vs. k.

	chamfer distance	BoostMap-Pa-Se-QS
1-nn	0.0547	0.0535
Best k-nn	0.0463 (k = 5)	0.0468 (k = 5)

Table 3: Comparison of k-nn error rates using original distance measure, and using a 256-dimensional parameter-space query-sensitive embedding. We show the error rates for 1-nn, and for the value of k that achieved the lowest k-nn error rate (the best k equals 5 in both cases).

12. DISCUSSION

The experimental results reported in this paper provide a thorough evaluation of different BoostMap variants on the MNIST dataset using the chamfer distance as the underlying distance measure. However, experiments with more datasets and comparisons with more existing methods are needed in order to have a clear picture of the relative advantages and disadvantages of BoostMap.

The main disadvantage of BoostMap is the running time of the training algorithm. At the same time, in [2] and in this paper we have successfully completed the training for large datasets and with computationally expensive distance measures, so we expect BoostMap to be applicable in a wide range of domains. Furthermore, in many applications, the training time can be an acceptable cost as long as it leads to a higher-quality embedding, and significantly faster nearest neighbor retrieval and k-nn classification.

The main advantage of BoostMap is that it is formulated as a classifier-combination problem, so that we can take advantage of powerful machine learning techniques to construct a highly accurate embedding from many simple, 1D embeddings. Our problem definition, that treats embeddings as classifiers, leads to an embedding construction method that can be applied in any space, metric or non-metric, without assuming any property of the space, except for expecting 1D embeddings to behave as weak classifiers. In contrast, FastMap makes strong Euclidean assumptions that, in our experiments, are useful only for low-dimensional embeddings. Bourgain embeddings make weaker assumptions, but they still assume that the underlying space is metric.

The machine-learning formulation also allows us to define query-sensitive embeddings and parameter-space embeddings, which are shown in the experiments to improve overall embedding quality for both nearest neighbor retrieval and nearest-neighbor classification accuracy. There are no obvious modifications to FastMap, Bourgain embeddings, and other related methods, that could yield query-sensitive or parameter-space embeddings. Posing embedding construction as minimization of a well-defined classification error provides us with great flexibility in deciding exactly what we want to optimize for, and how to achieve that optimization.

13. 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] V. Athitsos and S. Sclaroff. Database indexing methods for 3D hand pose estimation. In *Gesture Workshop*, pages 288–299. Springer-Verlag Heidelberg, 2003.

[4] V. Athitsos and S. Sclaroff. Estimating hand pose from a cluttered image. In *CVPR*, volume 1, pages 432–439, 2003.

[5] 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.

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

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

[8] 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.

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

[10] 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.

[11] Y. Freund and L. Mason. The alternating decision tree learning algorithm,. In *International Conference on Machine Learning*, pages 124–133, 1999.

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

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

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

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

[16] 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.

[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] 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.

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

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

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

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

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

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

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

[26] 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.

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

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