LEARNING EMBEDDINGS FOR INDEXING, RETRIEVAL, AND CLASSIFICATION, WITH APPLICATIONS TO OBJECT AND SHAPE RECOGNITION IN IMAGE DATABASES

VASSILIS ATHITSOS

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

BOSTON UNIVERSITY

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

LEARNING EMBEDDINGS FOR INDEXING, RETRIEVAL, AND CLASSIFICATION, WITH APPLICATIONS TO OBJECT AND SHAPE RECOGNITION IN IMAGE DATABASES

by

VASSILIS ATHITSOS

B.S., University of Chicago, 1995,
M.S., University of Chicago, 1997

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2006

Approved by

First Reader

Stan Sclaroff, PhD
Associate Professor of Computer Science

Second Reader

George Kollios
Assistant Professor of Computer Science

Third Reader

Margrit Betke
Associate Professor of Computer Science

Acknowledgments

First of all I thank my advisor, Stan Sclaroff, for his guidance and patience throughout my PhD studies, and especially for giving me the freedom to pursue my own interests and learn from my own mistakes. My thesis co-advisor, George Kollios, has offered me invaluable help in many areas, and most importantly should be credited with introducing me to embedding-based nearest neighbor retrieval, which turned out to be my thesis topic. I thank Professor Margrit Betke for participating in my committee, and for providing many helpful comments that helped me improve the quality of the document. I am also grateful to my other committee members, Professor Bill Freeman and Professor Tom Huang, for taking time out of their extremely busy schedules to go over my thesis and participate in my defense.

I have worked closely with many members of the IVC group, and I want to thank everyone in the group for providing me with such an enjoyable and stimulating working environment. I also want to thank the tens of students, professors, and staff that I have interacted with in the Computer Science Department, for all the times they have helped and supported me.

My friends in Boston and outside Boston have made a big difference in my life throughout my PhD studies. There is not enough space and time to list everyone who deserves to be mentioned here. However, I should especially thank Nenad Dedic, the Fishers in Morrison, IL, and Ella Averbukh, who have been an important part of my life.

I am grateful to my mother, father, and brother, for their unwavering support during my studies. I am also grateful to Viki and my little nephew, who are relative newcomers in my life but have made a big difference in helping me go through the last few months.

Finally, I want to thank Joni Alon, my good friend and primary research collaborator over the last several years. The core contributions of this thesis have evolved through the numerous discussions that we have had. This thesis, and my entire research agenda, would have been very differen without Joni.

LEARNING EMBEDDINGS FOR INDEXING, RETRIEVAL, AND CLASSIFICATION, WITH APPLICATIONS TO OBJECT AND SHAPE RECOGNITION IN IMAGE DATABASES

(Order No.)

VASSILIS ATHITSOS

Boston University, Graduate School of Arts and Sciences, 2006

Major Professor: Stan Sclaroff, Department of Computer Science

ABSTRACT

Nearest neighbor retrieval is the task of identifying, given a database of objects and a query object, the objects in the database that are the most similar to the query. Retrieving nearest neighbors is a necessary component of many practical applications, in fields as diverse as computer vision, pattern recognition, multimedia databases, bioinformatics, and computer networks. At the same time, finding nearest neighbors accurately and efficiently can be challenging, especially when the database contains a large number of objects, and when the underlying distance measure is computationally expensive. This thesis proposes new methods for improving the efficiency and accuracy of nearest neighbor retrieval and classification in spaces with computationally expensive distance measures. The proposed methods are domain-independent, and can be applied in arbitrary spaces, including non-Euclidean and non-metric spaces. In this thesis particular emphasis is given to computer vision applications related to object and shape recognition, where expensive non-Euclidean distance measures are often needed to achieve high accuracy.

The first contribution of this thesis is the BoostMap algorithm for embedding arbitrary spaces into a vector space with a computationally efficient distance measure. Using this

approach, an approximate set of nearest neighbors can be retrieved efficiently - often orders of magnitude faster than retrieval using the exact distance measure in the original space. The BoostMap algorithm has two key distinguishing features with respect to existing embedding methods. First, embedding construction explicitly maximizes the amount of nearest neighbor information preserved by the embedding. Second, embedding construction is treated as a machine learning problem, in contrast to existing methods that are based on geometric considerations.

The second contribution is a method for constructing query-sensitive distance measures for the purposes of nearest neighbor retrieval and classification. In high-dimensional spaces, query-sensitive distance measures allow for automatic selection of the dimensions that are the most informative for each specific query object. It is shown theoretically and experimentally that query-sensitivity increases the modeling power of embeddings, allowing embeddings to capture a larger amount of the nearest neighbor structure of the original space.

The third contribution is a method for speeding up nearest neighbor classification by combining multiple embedding-based nearest neighbor classifiers in a cascade. In a cascade, computationally efficient classifiers are used to quickly classify easy cases, and classifiers that are more computationally expensive and also more accurate are only applied to objects that are harder to classify. An interesting property of the proposed cascade method is that, under certain conditions, classification time actually decreases as the size of the database increases, a behavior that is in stark contrast to the behavior of typical nearest neighbor classification systems.

The proposed methods are evaluated experimentally in several different applications: hand shape recognition, off-line character recognition, online character recognition, and efficient retrieval of time series. In all datasets, the proposed methods lead to significant improvements in accuracy and efficiency compared to existing state-of-the-art methods. In some datasets, the general-purpose methods introduced in this thesis even outperform domain-specific methods that have been custom-designed for such datasets.

Contents

1 Introduction **1**

1.1 Nearest Neighbor Retrieval: Issues and Applications 1

1.2 Nearest Neighbor Classification 4

1.3 Computationally Expensive Distance Measures 7

1.4 Main Contributions 14

1.5 Overview of Thesis 19

2 Background **20**

2.1 Some Basic Definitions and Notation 20

2.2 Measures of Embedding Quality 23

2.3 Embedding Methods for Indexing 24

2.4 Embedding Application: Filter-and-Refine Retrieval 28

3 Related Work **31**

3.1 Indexing Methods for Vector Spaces 32

3.2 Indexing Methods for Non-Vector Spaces 34

3.3 Methods for Efficient Nearest Neighbor Classification 38

3.4 Summary of Related Work 39

4 BoostMap: A Machine Learning Method For Embedding Construction **41**

4.1 Associating Embeddings with Classifiers 42

4.2 Reducing Embedding Construction to a Boosting Problem 48

4.3 The Embedding Construction Algorithm 50

4.4 Properties of BoostMap Embeddings 59

4.5 Summary of the BoostMap method 63

5 Query-Sensitive Embeddings **65**

5.1 Some Additional Related Work 66

5.2 Motivation for Query-Sensitive Distance Measures 67

5.3 Constructing a Query-Sensitive Embedding 71

5.4 Properties and Discussion of the Method 78

5.5 Summary of Query-Sensitive Embeddings 79

6 Efficient Nearest Neighbor Classification Using Cascades of Approximate Classifiers **81**

6.1 Some Additional Related Work 82

6.2 Optimizing for Classification Accuracy 83

6.3 Overview of Cascades of Classifiers 85

6.4 Constructing a Cascade of Approximate Nearest Neighbor Classifiers 88

6.5 Discussion of the Cascade Method 93

7 Experiments **96**

7.1 Datasets 96

7.2 Evaluation Methodology and Parameter Choices 102

7.3 Methods Used for Comparison Purposes 104

7.4 Evaluation of the Original BoostMap Method 106

7.5 Evaluation of Query-Sensitive Embeddings 108

7.6 Experiments on Nearest Neighbor Classification 127

7.7 Summary of Experimental Results 136

8 Discussion and Conclusions **140**

8.1 Discussion of Contributions 140

8.2 Broader Issues and Future Work 143

8.3 Conclusions 147

References **149**

List of Tables

2.1 Table of the main symbols used throughout this thesis, part 1. 21

2.2 Table of the main symbols used throughout this thesis, part 2. 22

3.1 A list of methods for efficient nearest neighbor netrieval, and some of their key characteristics. 40

7.1 Comparison of BoostMap, FastMap, random reference objects (RRO), random line projections (RLP), and VP-trees, on the ASL handshape dataset. 113

7.2 Comparison of BoostMap, FastMap, random reference objects (RRO), random line projections (RLP), and VP-trees, on the MNIST dataset. 113

7.3 Comparison of BoostMap, FastMap, random reference objects (RRO), random line projections (RLP), and VP-trees, on the UNIPEN dataset. 114

7.4 Comparison of BoostMap, FastMap, random reference objects (RRO), random line projections (RLP), and VP-trees, on the time series dataset. . . . 114

7.5 Comparison of Ra-QI, Ra-QS, Se-QI, and Se-QS on the MNIST dataset based on 10,000 query objects and the time series dataset based on 1,000 query objects. 121

7.6 Comparison of Se-QS, FastMap, random reference objects (RRO), random line projections (RLP), and VP-trees, on the MNIST dataset based on 10,000 query objects and the time series dataset based on 1,000 query objects. . . 125

7.7 The sequences $\mathbb{P}$ of filter-and-refine processes that were passed as input to Algorithm 3 for the MNIST and UNIPEN datasets. 131

7.8 Speeds and error rates achieved by different approximate and exact nearest neighbor classification methods on the MNIST dataset, using 10,000 test objects and 20,000 database objects. 134

7.9 Speeds and error rates achieved by different approximate and exact nearest neighbor classification methods on the UNIPEN dataset. 139

List of Figures

1·1 Handwritten digit recognition using a nearest neighbor classifier and the MNIST database of 60,000 training images. 4

1·2 Estimating 3D hand pose using nearest neighbors. 5

1·3 An illustration of how simple linear-time distance measures can fail to capture intuitive notions of edge image similarity. 8

1·4 An illustration of the chamfer distance and the Hausdorff distance. 10

1·5 An illustration of the need for string alignment in order to compute a meaningful distance between strings. 11

1·6 Four sample frames from video sequences of native sign language speakers communicating in ASL. Computationally expensive distance measures can be useful at various stages in an automated ASL recognition system. 12

2·1 A Lipschitz embedding of the plane into the real line. 25

2·2 Computing a "line-projection" 1D embedding. 27

4·1 An example of an embedding and its associated classifier. 44

4·2 The AdaBoost algorithm. 50

5·1 A toy example illustrating the use of query-sensitive embeddings. 70

7·1 The 20 handshapes used in the ASL handshape dataset. 97

7·2 Examples of different appearance of a fixed 3D hand shape, obtaining by altering camera viewpoint and image plane rotation. 97

7·3 Example images from the MNIST dataset of handwritten digits. 100

7·4 Example of a normalized Unipen digit. 101

7·5 Comparing methods BoostMap, FastMap, RRO, RLP, and VP-trees, on the ASL handshape dataset, using the chamfer distance as the exact distance measure. 109
7·6 Comparing methods BoostMap, FastMap, RRO, RLP, and VP-trees, on the MNIST dataset, using shape context matching as the exact distance measure. 110
7·7 Comparing methods BoostMap, FastMap, RRO, RLP, and VP-trees, on the UNIPEN dataset, using DTW as the exact distance measure. 111
7·8 Comparing methods BoostMap, FastMap, RRO, RLP, and VP-trees, on the time series database, using constrained DTW as the exact distance measure. 112
7·9 Comparing BoostMap as described in this thesis to a modified version of BoostMap, where training triples are chosen randomly. Here we show results on the ASL handshape dataset, using the chamfer distance as the exact distance measure. 115
7·10 Comparing BoostMap as described in this thesis to a modified version of BoostMap, where training triples are chosen randomly. Here we show results on the MNIST dataset, using shape context matching as the exact distance measure. 116
7·11 Comparing methods BoostMap, FastMap, RRO, RLP, and VP-trees, on the UNIPEN dataset, using DTW as the exact distance measure. 117
7·12 Comparing BoostMap as described in this thesis to a modified version of BoostMap, where training triples are chosen randomly. Here we show results on the time series database, using constrained DTW as the exact distance measure. 118
7·13 Comparing methods Ra-QI, Ra-QS, Se-QI and Se-QS on the MNIST dataset, using shape context matching as the exact distance measure. 119
7·14 Comparing methods Ra-QI, Ra-QS, Se-QI and Se-QS on the time series dataset, using constrained Dynamic Time Warping as the exact distance measure. 120

7·15 Comparing methods Se-QS, FastMap, random reference objects, random line projections, and VP-trees, on the MNIST dataset, using shape context matching as the exact distance measure. 123
7·16 Comparing methods Se-QS, FastMap, random reference objects, random line projections, and VP-trees, on the time series database, using constrained Dynamic Time Warping as the exact distance measure. 124
7·17 Comparing classification accuracy vs. efficiency trade-offs achieved by the BoostMap, RRO, and RLP methods on the ASL handshape dataset. 128
7·18 Comparing classification accuracy vs. efficiency trade-offs achieved by the BoostMap, RRO, and RLP methods on the MNIST dataset. 129
7·19 Error rates attained using BoostMap and BoostMap-C, without a cascade, vs. number of exact distance evaluations per test object, on the MNIST dataset. 130
7·20 Error rates attained by cascade classifiers vs. average number of exact distance evaluations per test object, for the MNIST dataset. 132
7·21 Comparing classification accuracy vs. efficiency trade-offs achieved by the BoostMap, RRO, and RLP methods on the UNIPEN dataset. 137
7·22 Error rates attained using BoostMap and BoostMap-C, without a cascade, vs. number of exact distance evaluations per test object, on the UNIPEN dataset. 138
7·23 Error rates attained by cascade classifiers vs. average number of exact distance evaluations per test object, for the UNIPEN dataset 138

List of Abbreviations

1D	....	One-dimensional
2D	....	Two-dimensional
3D	....	Three-dimensional
AESA	....	Approximating and Eliminating Search Algorithm
ASL	....	American Sign Language
Boost-NN	....	Boosted Nearest Neighbors
CNN	....	Condensed Nearest Neighbor
CPU	....	Central Processing Unit
CSDTW	....	Cluster Generative Statistical Dynamic Time Warping
DNA	....	Deoxyribonucleic Acid
DP	....	Dynamic Programming
DTW	....	Dynamic Time Warping
EMD	....	Earth Movers Distance
KL	....	Kullback-Leibler
k-nn	....	k-nearest neighbors
LAESA	....	Linear Approximating and Eliminating Search Algorithm
LLE	....	Locally Linear Embedding
LSH	....	Locality Sensitive Hashing
MDS	....	Multidimensional Scaling

MNIST	Modified NIST
NIST	National Institute of Standards and Technology
MVP	Multiple Vantage Point
OCR	Optical Character Recognition
PCA	Principal Component Analysis
RLP	Random Line Projections
RRO	Random Reference Objects
QI	Query-Inensitive
QS	Query-Sensitive
SC	Shape Context
VA-File	Vector-Approximation File
VP	Vantage Point

Chapter 1

Introduction

This thesis proposes novel methods for improving the accuracy and efficiency of nearest neighbor retrieval and classification in spaces with computationally expensive distance measures. Before describing those methods in detail, we first need to take a look at the problem we are trying to solve. In this chapter we briefly introduce the concepts of nearest neighbor retrieval and classification, and we outline some of the numerous applications of nearest neighbor methods. We highlight application areas that require the use of computationally expensive distance measures, and can thus benefit from the work presented in this thesis. We also provide a brief overview of the main contributions of this thesis.

1.1 Nearest Neighbor Retrieval: Issues and Applications

In recent years, the capacity of digital storage systems has increased dramatically. The ability to create large databases containing vast amounts of data has been exploited in a wide range of domains and applications. Internet search engines store enormous amounts of Web content in large databases, and help Internet users quickly identify content of interest. Large databases of biological data are used to store the wealth of information obtained by analyzing biological structures and processes in different species [Boeckmann et al., 2003]. In computer vision, large image databases have been used for object and shape recognition, in applications as diverse as face recognition [Phillips et al., 2003], body pose estimation [Shakhnarovich et al., 2003], and optical character recognition [Belongie et al., 2002].

In many applications the primary purpose of a database is to provide specific information on a need-to-know basis. The information that we need to extract depends on the specific task that we must perform at the current moment, and that task is not known to the database system in advance. For example, we may want to find web pages similar to a particular page, proteins that are similar to a particular protein, or "mugshots" of a particular person. Naturally, the database system cannot know in advance which web page, protein, or person the user will be interested in. An important issue that arises in such applications is that, as the amount of stored data gets bigger, it becomes increasingly challenging to extract specific pieces of information from the database. The specific data that we are interested in at a particular moment is typically a small fraction of the entire database. The larger the database, the deeper the few items of interest are "buried" inside the database, and the harder we have to look to identify those items.

The three example tasks mentioned above, i.e., finding web pages, proteins, and mugshots of interest, are cases where we want to perform nearest neighbor retrieval: we have a database of "objects," there is a particular query object that we are interested in, and we want to retrieve the database objects that are the most similar to the query object. Two key measures of performance that we use to evaluate a retrieval system are accuracy and speed. Naturally, we want the retrieved objects to indeed be the database objects that are the most similar to the query, with as few omissions as possible and as few inclusions of unrelated objects as possible. At the same time, we want retrieval to be fast and efficient, in order to minimize both the time the user has to wait and the required CPU processing time.

Nearest neighbor retrieval has many uses in a wide range of domains and applications, beyond the example applications in Web browsing, biological databases and face recognition that we have already mentioned. Some additional examples include:

- Optimizing network usage in peer-to-peer computer networks. When a host requests to download some specific content, network performance is optimized if we can identify nearby network locations (using some measure of network proximity) that can provide that content [Hildrum et al., 2002].

- Content-based access in large multimedia databases. Users of such databases often need to identify and efficiently retrieve database entries that are similar to a particular document [White and Jain, 1996, Shen et al., 2005].

- In the medical domain, improving diagnosis and prognosis. Given a new case, it is often beneficial to identify the most similar cases in a database of case histories, in order to better analyze the current symptoms and evaluate different treatment options [Borst et al., 2000].

- Analyzing and predicting time series data, such as stock prices, weather and climate data, or trajectories of moving objects. An important tool in processing time series data is comparing that data with similar items in databases of known examples [Keogh, 2002, Vlachos et al., 2003].

- Clustering applications. Many widely used clustering methods, such as K-means clustering and agglomerative clustering, require large numbers of nearest neighbor retrieval operations, in order to decide what cluster each object should be assigned to, or what clusters should be merged together [Everitt et al., 2001].

- Visualization applications. When visualizing high-dimensional or non-Euclidean data, it is often important to identify the nearest neighbors of each object and to try to display objects in such a way that they are shown close to their nearest neighbors [Vlachos et al., 2002].

A general application of nearest neighbor retrieval that is of particular interest to the fields of computer vision and pattern recognition is nearest neighbor classification. While the methods proposed in this thesis can be used in many of the above retrieval applications, the main motivation for developing these methods has been to improve the accuracy and efficiency of nearest neighbor classifiers that use computationally expensive distance measures. We will now proceed to take a closer look at the topic of nearest neighbor classification.

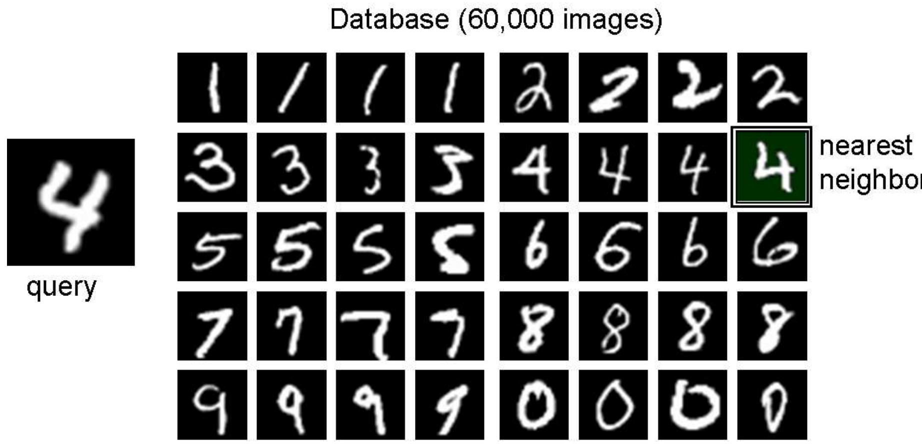

Figure 1·1: Handwritten digit recognition using a nearest neighbor classifier and the MNIST database of 60,000 training images [LeCun et al., 1998, Belongie et al., 2002, Athitsos et al., 2005a]. Given a query image that we want to classify, the system retrieves the nearest neighbor of the query in the database, and assigns the class of the retrieved nearest neighbor to the query image.

1.2 Nearest Neighbor Classification

Suppose we want to build a system that can recognize images of isolated digits, from 0 to 9, as shown in Figure 1·1. A straightforward approach is to use a nearest neighbor classifier. To design such a classifier, we first create a database of training objects, i.e., a database of images of digits in this example. For each training image we need to also store its class label, i.e., we must specify which digit is actually displayed in that image. Finally, we need to specify a distance measure that can be used for evaluating similarity between images. Then, given a test image to recognize, the system simply finds the nearest neighbor of that image in the database, and classifies the test image as belonging to the same class as its nearest neighbor. This process is illustrated in Figure 1·1. Alternatively, the system can retrieve the k nearest neighbors (for some parameter k) and classify the test image as belonging to the class that is the most common among those k nearest neighbors.

As another example, suppose we want to build a system that takes as input the image

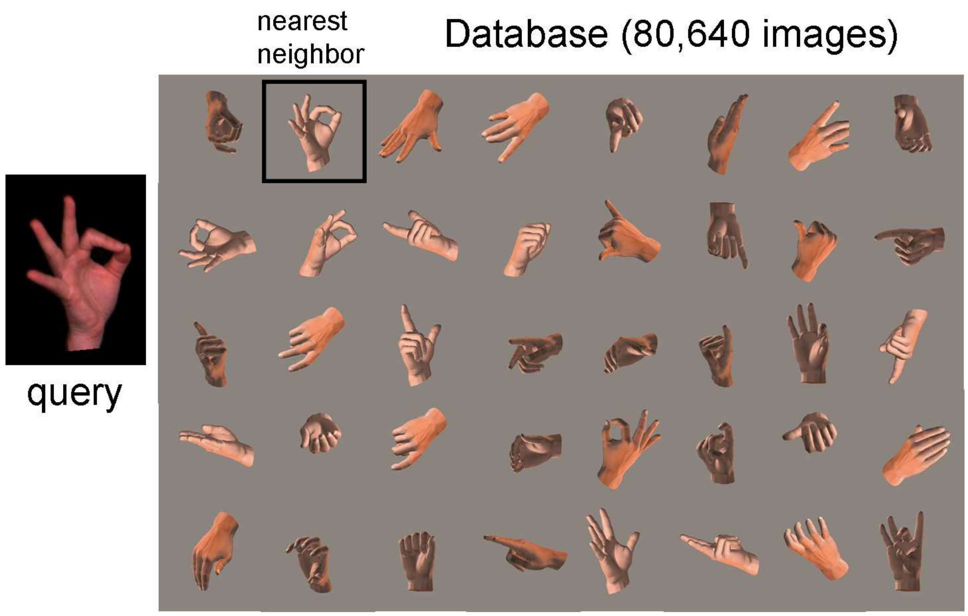

Figure 1·2: Estimating 3D hand pose using nearest neighbors. Given a query image of a hand, the system finds the most similar images in a large database of computer-generated images of hands. The 3D hand poses of the retrieved images are output by the system as plausible estimates for the 3D hand pose of the query. The figure displays a query image, the nearest neighbor of the query in the database, and some other, randomly selected database images.

of a hand and produces as output the most plausible estimates of the 3D hand pose, i.e. estimates of the joint angles and the 3D orientation of the hand. This problem can also be tackled by finding the nearest neighbors of the test image in a database, as described in [Athitsos and Sclaroff, 2003, Athitsos et al., 2004] and illustrated in Figure 1·2. We build, using computer graphics, a large database containing pictures of hands with different joint angle configurations and 3D orientations. For each picture, we also store the 3D hand pose that was used to create that picture. Then, given a test image, the system finds the nearest neighbors of that image in the database, and uses the hand poses corresponding to the nearest neighbors to generate estimates of the hand pose in the test image.

Nearest neighbor classification is a popular technique in computer vision and pattern recognition. One of the main attractions of nearest neighbor classifiers is their simplicity. All we need to do to design such a classifier is provide a database of training objects and specify a distance measure. At the same time, nearest neighbor classifiers can be very powerful and have some very desirable properties:

- Nearest neighbor classifiers can easily be applied to problems with an arbitrary number of classes, such as the problem of recognizing the faces of hundreds or thousands of individuals, or the problem of estimating the pose of an articulated object. Many popular methods, such as AdaBoost [Schapire and Singer, 1999] and support vector machines [Vapnik, 1995], are not well-suited for such problems because they do not scale well with the number of classes.

- Even in problems with a small number of classes, such as optical character recognition, nearest neighbor classifiers can be very powerful, because of their ability to model complex, non-parametric distributions.

- A well-known theoretical property of k-nearest neighbor classifiers is that their classification accuracy becomes asymptomatically optimal as the training size approaches infinity [Duda et al., 2001].

In practice, nearest neighbor classifiers are often more accurate than other, significantly more complicated classification methods. As an example, nearest neighbor classification using shape context matching as a distance measure produced a lower error rate than a large number of competing methods for the problem of handwritten digit recognition, as measured on the popular MNIST dataset [Belongie et al., 2002]. At the same time, nearest neighbor classifiers are often impractical for real applications, because they are too computationally expensive. The handwritten digit recognition system in [Belongie et al., 2002] showcases that problem: classifying a single object takes over 20 minutes on a modern PC using an optimized C++ implementation. The dilemma with nearest neighbor classifiers is that, as the number of available training objects increases, classification accuracy improves, but processing time increases as well because it takes longer to find nearest neighbors.

This problem is exacerbated in domains where we need to use computationally expensive distance measures. In the experiments section we will apply the domain-independent methods proposed in this thesis to speed up the handwritten digit recognition system of [Belongie et al., 2002], from over 20 minutes per test image to about 5 seconds per test image with virtually no loss of accuracy.

The main focus of this thesis is on improving nearest neighbor retrieval and classification performance in spaces with computationally expensive distance measures. In the next section we provide some additional motivation for our methods, by illustrating several examples where we need to use expensive distance measures, in order to capture intuitive notions of similarity that more efficient distace measures fail to capture. We also discuss the problems that arise when using such measures.

1.3 Computationally Expensive Distance Measures

Determining whether a distance measure is computationally expensive or not is to a large extent a subjective decision, and also depends on application-specific settings and parameters. At the same time, we can provide a rule of thumb that frequently agrees with our intuitive judgement of when a distance measure is expensive. Our rule of thumb is the following: we consider a distance measure to be computationally expensive when measuring a single distance between two objects takes time that is superlinear to the length of these objects. A family of measures that are *not* computationally expensive, according to this rule of thumb, are the L_p metrics defined on any real vector space $\mathbb{R}^d$. The most common L_p metrics are the L_2 metric, which is the Euclidean distance, and the L_1 metric, which is often called the Manhattan distance. Evaluating L_p distances in $\mathbb{R}^d$ takes $O(d)$ time, which is linear to the length of the objects. Naturally, if the dimensionality d is very large, then our rule of thumb breaks down and L_p start taking "too long" to compute. However, for all practical purposes within the scope of this thesis, L_p distances are considered to be efficient alternatives, compared to the various computationally expensive distance measures that we need to use.

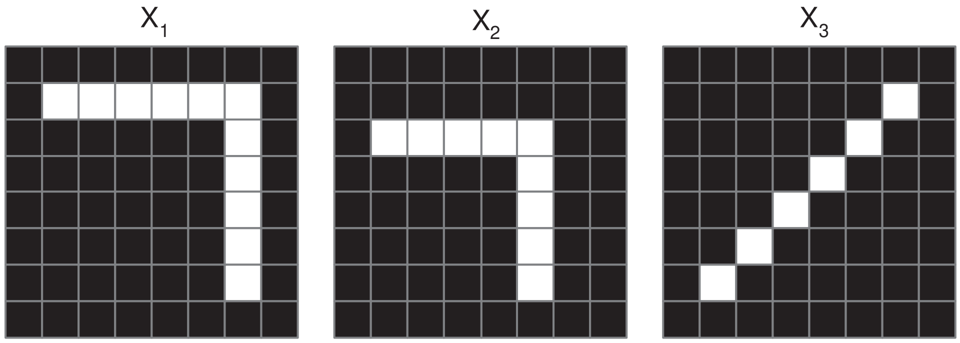

Figure 1·3: An illustration of how simple linear-time distance measures can fail to capture intuitive notions of edge image similarity. Images X_1, X_2, and X_3 are 8×8 edge images, where background pixels contain value 0 and are shown as black, and edge pixels contain value 1 and are shown as white. To a human, X_1 and X_2 are more similar to each other than to X_3, because in X_1 and X_2 the edge pixels are closer to each other and the overall shapes are much more similar. Using any L_p metric (for $0 < p < \infty$), however, X_1 and X_2 are closer to X_3 than to each other. For example, for the Manhattan metric L_1, we get: $L_1(X_1, X_2) = 20, L_1(X_1, X_3) = 15, L_1(X_2, X_3) = 13$,.

As a first example of a space where it is beneficial to use a computationally expensive distance measure, consider the space of binary edge images. One way to represent a binary edge image is as a binary vector, with one dimension per image pixel. In each dimension we store value 1 if the corresponding pixel is an edge pixel, and value 0 otherwise. Matching edge images can be useful for various applications, such as optical character recognition (Figure 1·1), and hand pose estimation (Figure 1·2). One possible distance measure we can use for edge images is the Euclidean distance between the binary vector representations of those images. However, the problem with the Euclidean distance is that it fails to capture our intuitive notions of when two edge images should be considered similar, as illustrated in Figure 1·3. In that figure, we see that we often want to consider two edge images similar (images X_1 and X_2 in the figure) even when there is zero overlap between the edge pixels in one image and the edge pixels in the other image.

Two alternative distance measures we can use for comparing edge images are the *chamfer distance* [Barrow et al., 1977] and the *Hausdorff distance* [Huttenlocher et al., 1993]

(Figure 1·4). The directed chamfer distance from edge image A to edge image B is the average distance from each edge pixel in A to its nearest edge pixel in B. The undirected chamfer distance, which is often referred to simply as chamfer distance, is the sum of the two directed distances, from A to B and from B to A. The Hausdorff distance is the maximum distance between an edge pixel in one image and its nearest edge pixel in the other image. Figure 1·4 illustrates how these distances are computed.

Compared to L_p distances, the chamfer distance and the Hausdorff distance are much closer to our intuitive notions of similarity between edge images. Even when the edge pixels from the two images do not overlap, it is still possible for the two images to have a small distance to each other, as long as edge pixels in one image are close to edge pixels in the other image. For example, for the images shown in Figure 1·3, using the Euclidean distance to measure distances between pixel locations, the chamfer distances are: 2.04 between X_1 and X_2, 4.52 between X_1 and X_3, and 3.56 between X_2 and X_3. Consequently, in contrast to L_p measures, the chamfer distance captures our intuition that X_1 is more similar to X_2 than to X_3. However, the chamfer distance and the Hausdorff distance take superlinear time to compute: they require $O(d \log d)$ time for images with at most d edge pixels [Huttenlocher et al., 1993]. The reason is that, for both distance measures, we need to find for each edge pixel x in one image the closest edge pixel in the other image. That operation takes $O(\log d)$ time for a single edge pixel x, using a two-dimensional version of binary search.

We should note that the chamfer distance and the Hausdorff distance can be computed in an alternative way, that takes $O(d)$ time, if we have precomputed, for each edge image, the so-called *distance transform* of that image [Breu et al., 1995]. The distance transform stores in every image pixel the distance of that pixel to its nearest edge pixel. However, since edge images are typically sparse, computing and storing distance transforms for a large database of edge images can increase the memory and disk storage requirements of that database by orders of magnitude, and thus using distance transforms is often impractical.

Another example case where it is beneficial to use a computationally expensive distance

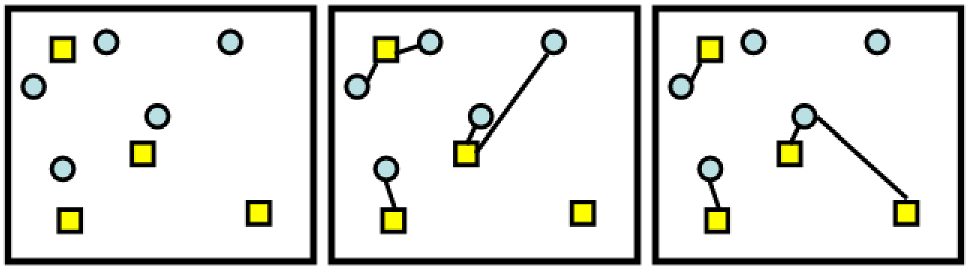

Figure 1·4: An illustration of the chamfer distance and the Hausdorff distance. The left image shows two sets of points. Points in the first set are shown as circles, points in the second set are shown as squares. Each set of points could be, for example, the set of edge pixels in one image. The middle image shows a link between each circle and its closest square. The circle-to-square directed chamfer distance between the set of circles and the set of squares is the average distance between a circle and its closest square, so it is the average length of the links shown in the middle image. The right image shows a link between each square and its closest circle. The square-to-circle directed chamfer distance is the average length of those links. The (undirected) chamfer distance between squares and circles is the sum of the two directed distances. The Hausdorff distance between squares and circles is simply the length of the longest link in the middle and right images.

measure is the space of strings. A naive way to measure distances between two strings is to simply compare the letter in each position of the first string with the letter in the same position of the second string. As Figure 1·5 illustrates, this simple way of measuring distances again fails to capture our intuitive notion of when two strings are similar. A more meaningful, and popular, measure of the distance between two strings is the *edit distance* [Levenshtein, 1966], which counts the minimum amount of insertions, deletions and letter changes that are needed to convert one string into the other string (Figure 1·5). This distance is computed using dynamic programming and takes time $O(d^2)$ for strings of at most d letters. Essentially the distance is computed by finding an optimal alignment between the two strings. A popular application of the edit distance is the UNIX/LINUX *diff* utility, that finds the minimal set of changes that can be applied to convert one file to another file.

The Smith-Waterman algorithm is a variant of the edit distance that is used for match-

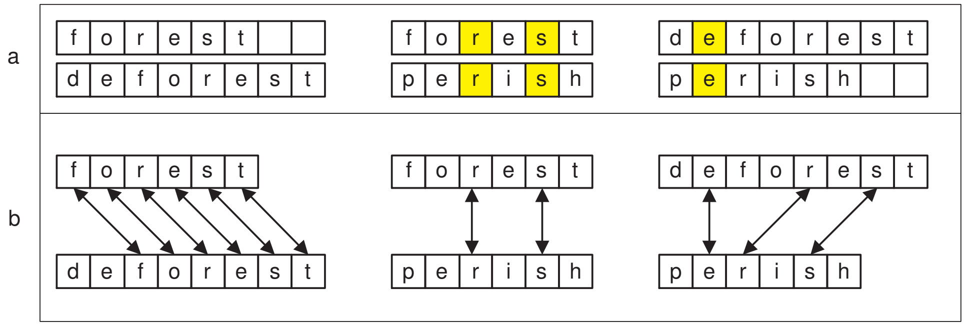

Figure 1·5: An illustration of the need for string alignment in order to compute a meaningful distance between strings. The words "forest" and "deforest" are much more similar to each other than to the word "perish." However, a naive linear-time distance measure, that simply compares letters in the same position, would output that "forest" and "deforest" are more similar to "perish" than to each other, because, as shown in (a), for "forest" and "deforest" there is no position where they have the same letter, whereas for both words there are positions where they have the same letter as the word "perish." The edit distance is a quadratic-time distance measure that finds an optimal correspondence between letters in the two words, as shown in (b), and considers the distance between strings to be the minimum number of letter changes, insertions, and deletions that are needed to convert one string to the other. Using the edit distance as D, $D(\text{forest}, \text{deforest}) = 2$, $D(\text{forest}, \text{perish}) = 4$, and $D(\text{deforest}, \text{perish}) = 5$. For similar reasons, meaningful distance measures for proteins and dynamic gestures also depend on finding an optimal alignment between objects, and thus take superlinear time.

ing proteins and DNA sequences [Smith and Waterman, 1981]. Dynamic programming is also used in computing Dynamic Time Warping (DTW) [Kruskall and Liberman, 1983], which is a distance measure between time series. DTW is frequently used for recognizing gestures and time series (Figure 1·6). Distance measures that are defined based on dynamic programming also include Dynamic Space-Time Warping [Alon et al., 2005c], and the shape context matching method described in [Thayananthan et al., 2003].

A third example of a computationally expensive distance measure is bipartite matching [Kuhn, 1955]. Suppose that we want to compare two images, and from each image we have extracted a set of d features. Furthermore, suppose that we have defined an auxiliary dis-

Figure 1·6: Four sample frames from video sequences of native sign language speakers communicating in ASL. Computationally expensive distance measures can be useful at various stages in an automated ASL recognition system. Recognizing hand pose, which is important for discriminating between different signs, can be achieved using a distance measure like the chamfer distance, that aligns hand features from the image with hand features from database images. Recognizing an actual ASL utterance (word or phrase), represented as a time series of extracted features, can be done using Dynamic Time Warping, a distance measure that uses Dynamic Programming to align two sequences with each other.

tance measure for comparing features with each other. In bipartite matching, the distance between the two feature sets is the minimum sum of distances between corresponding features, over all possible one-to-one correspondences we can define between features in one set and features in the other set. The optimal set of one-to-one correspondences can be found using the Hungarian algorithm, which takes time $O(d^3)$ for sets of d features [Kuhn, 1955].

In all three examples we have listed so far, the computationally expensive distance measures we have considered are computed by finding an optimal alignment, i.e., by finding optimal correspondences (one-to-one, or many-to-many) between components of the two objects. Searching to find an optimal alignment takes time that is superlinear to the length of the objects and that leads to distance measures that are computationally expensive. In contrast, when we measure L_p distances between vectors in $\mathbb{R}^d$, we align those vectors in a trivial way: the i-th coordinate of one vector simply corresponds to the i-th coordinate of any other vector. If $j \neq i$, the difference between the i-th coordinate of one vector and the j-th coordinate of another vector does not affect the distance between the two vectors. The fact that vectors have a fixed 1-1 correspondence between their coordinates allows us to compute L_p distances in linear time.

Computationally expensive distance measures can also be encountered in domains where aligning objects is not an issue. An example of such a domain is a peer-to-peer network [Hildrum et al., 2002]. Intuitively, the distance between two network nodes is a number that describes how efficient it is to move content from one node to the other. Efficiency is mainly measured in terms of bandwidth and round-trip time. If a node Q in the network needs to download some content, it is beneficial to identify, among all nodes that store that content, the node that is the "closest" to Q, so as to maximize network efficiency. The naive, brute-force way of finding the closest node is for Q to simply communicate with all nodes and measure bandwidth and round-trip time. However, these measurements would generate a large amount of network traffic and would defeat the purpose of maximizing network efficiency. Furthermore, such measurements would also take "too long" to complete in many practical scenarios.

We should note that distances between network nodes also deviate from the rule of thumb we proposed in the beginning of this section, which stated that expensive distance measures are the ones that take time superlinear to the length of the objects. That rule of thumb refers to distances that are evaluated algorithmically, and objects that are elements of an abstract space and can be specified with a set of numbers or symbols. In contrast, network distances are physical, not algorithmic, measurements, and the network nodes are actual physical objects. Actually, the methods proposed in this thesis can be applied to map network nodes into vectors, and to approximate network distances with distances between vectors, thus addressing both the problem of how to represent network nodes symbolically/numerically, and the problem of how to approximate network distances efficiently and without generating large amounts of network traffic.

Some additional examples of computationally expensive distance measures that we can mention briefly are shape context matching [Belongie et al., 2002] for comparing edge images, the Earth Mover's Distance (EMD) [Rubner et al., 1998] and the Kullback-Leibler (KL) distance [Cover and Thomas, 1991] for comparing distributions, and Dynamic Space Time Warping [Alon et al., 2005c] for comparing dynamic gestures. Overall, using computationally expensive distance measures in a large database makes nearest neighbor retrieval challenging. Developing appropriate indexing methods can significantly expand the range of practical applications where such measures can be employed, and this has been a primary motivation behind the work described in this thesis.

1.4 Main Contributions

In this section we briefly go over the main contributions of this thesis. In order to put these contributions in context, we first take a look at the issues that arise when we want to perform nearest neigbhor retrieval in spaces with computationally expensive distance measures.

1.4.1 Overview of the Problem

The most straightforward algorithm for nearest neighbor retrieval is brute-force search: we simply measure the distance between the query object and each database object. The time complexity of brute-force search is linear to two quantities: the number of database objects, and the average time it takes to measure the distance between two objects. Clearly, as the number of database objects increases, brute-force search can become computationally demanding, or even impractical for particular applications. As discussed in the previous section, this problem is exacerbated in domains where evaluating the distance between two objects is computationally expensive.

What complicates matters is that computationally expensive distance measures are typically not L_p measures, and the majority of existing indexing tools are applicable only to L_p measures [Böhm et al., 2001, White and Jain, 1996]. Furthermore, while several metric indexing methods have been proposed that can, in theory, be used with arbitrary distance measures, many popular distance measures violate the triangle inequality, and are thus non-metric. Examples of non-metric measures are the chamfer distance [Barrow et al., 1977], Dynamic Time Warping [Kruskall and Liberman, 1983], shape context matching [Belongie et al., 2002], and the Kullback-Leibler (KL) distance [Cover and Thomas, 1991]. Applying metric methods to such measures is heuristic, and there is a need for principled indexing methods that do not rely on metric assumptions. The focus of this thesis is on designing principled and general methods for efficient nearest neighbor retrieval in non-Euclidean and non-metric spaces with computationally expensive distance measures.

1.4.2 The Contributions

This thesis proposes indexing methods that can be applied in general metric and non-metric spaces with computationally expensive distance measures and achieve state-of-the-art performance in several experimental datasets. The proposed methods do not rely on any geometric assumptions and directly maximize the amount of nearest neighbor information preserved by the indexing structure, in both metric and non-metric spaces.

More specifically, in this thesis we propose methods for defining, and optimizing, novel types of embeddings for approximating computationally expensive distance measures. Embeddings are a general family of methods that can be used for efficient nearest neighbor retrieval in arbitrary spaces. Embeddings are simply functions that map an original space and distance measure to a target space and target distance measure. Typically the target space is the d-dimensional real vector space $\mathbb{R}^d$, and the target distance measure is the Euclidean distance or some alternative L_p metric. Our goal is to construct embeddings such that the target distance measure is significantly more efficient computationally than the original distance measure. If an embedding F satisfies that requirement, then given a database $\mathbb{U}$ and a query object Q, instead of performing brute-force search for the nearest neighbors of Q in $\mathbb{U}$, it is much more efficient to perform brute-force search for the nearest neighbors of $F(Q)$ in $F(\mathbb{U})$. Furthermore, if the target space is a real vector space, we can make the search for the nearest neighbors of $F(Q)$ even more efficient, by taking advantage of the numerous indexing tools that have been developed for vector spaces with L_p metrics.

In order for embedding-based retrieval and classification to be useful, a necessary condition is that the nearest neighbors of $F(Q)$ are, at least in most cases, the mappings of the nearest neighbors of Q. In other words, embeddings must preserve a large amount of the nearest-neighbor structure of the original space. At the same time, embeddings are used as components of retrieval and classification processes that must be as computationally as efficient as possible. In this thesis, we introduce two novel methods for the problem of maximizing the amount of nearest neighbor structure preserved by an embedding: the *BoostMap* method, which is a machine learning method for optimizing embeddings, and *query-sensitive embeddings*, which automatically identify, for each query, the embedding coordinates that are the most informative for that query. We also introduce a novel method for speeding up nearest neighbor classification, by designing a cascade of approximate nearest neighbor classifiers. We now proceed to briefly describe each of these three contributions.

BoostMap

BoostMap is a general method for constructing and optimizing embeddings for the purpose of nearest neighbor retrieval, that is applicable to arbitrary spaces and distance measures. One key differentiating feature of BoostMap with respect to other embedding methods is that it optimizes a direct measure of how well the embedding preserves the nearest neighbor structure of the original space. The optimization criterion is valid in any space and does not rely on any Euclidean or metric assumptions. Consequently, the embedding optimization method can be used in both metric and non-metric spaces. From another perspective, BoostMap is different than existing embedding methods because its formulation is based on machine learning. In the BoostMap method, embeddings are treated as binary classifiers, and embedding optimization is performed using the popular boosting methodology from machine learning. De-emphasizing reliance on geometric properties and formulating embedding construction as a machine learning problem is an important step towards creating principled indexing methods for non-Euclidean and non-metric spaces whose geometry is only poorly understood.

The key intuition behind the BoostMap method is that any embedding F can be treated as a binary classifier that predicts, for any three objects X, A, B, if X is closer to A or to B, by simply checking if $F(X)$ is closer to $F(A)$ or to $F(B)$. If F never makes any mistakes, then F perfectly preserves nearest neighbor structure. We show that the error rate of F on a specific set of triples (X, A, B) is a direct measure of the the amount of nearest neighbor structure preserved by F. The building blocks that we use for embedding construction are simple, one-dimensional (1D) embeddings that have been previously proposed and used in the literature [Faloutsos and Lin, 1995, Hjaltason and Samet, 2003a]. We treat these 1D embeddings as weak classifiers, and we use AdaBoost [Schapire and Singer, 1999] to combine many such weak classifiers into a strong classifier. We show that the strong classifier constructed by AdaBoost naturally corresponds to a multidimensional embedding.

Query-Sensitive Embeddings

Query-sensitive embeddings are a novel type of embeddings that can be used to improve the accuracy and efficiency of embedding-based nearest neighbor retrieval. What differentiates query-sensitive embeddings from previously proposed types of embeddings [Hjaltason and Samet, 2003a, Roweis and Saul, 2000, Tenenbaum et al., 2000, Young and Hamer, 1987] is the distance measure that is used in the target space of embedding. As is typical in embedding methods, query-sensitive embeddings map objects into a vector space with a weighted L_p distance measure. What is different in query-sensitive embeddings is that the weights of this L_p measure are not fixed, but depend on the query object. A query-sensitive distance measure improves embedding quality, by providing a natural way to identify, for each query object, the embedding dimensions that are the most useful for retrieving the nearest neighbors of that query.

Identifying the most informative dimensions is an important issue that arises when objects are represented as high-dimensional vectors [Aggarwal, 2001]. For the purposes of efficient retrieval, query-sensitive embeddings are able to preserve a larger amount of nearest neighbor information compared to their query-insensitive counterparts. This additional modeling power translates in practice to significantly better trade-offs between retrieval accuracy and efficiency.

Cascades of Approximate Nearest Neighbor Classifiers

An exact nearest neighbor classifier classifies a test object based on the class labels of its true nearest neighbors. However, in many practical applications, particularly when non-metric distance measures are used, the only method that guarantees retrieval of the true nearest neighbors is brute-force search. An approximate nearest neighbor classifier classifies a test subject based on the class labels of the approximate nearest neighbors retrieved using some indexing scheme. In our method, using BoostMap embeddings, we construct a sequence of approximate nearest neighbor classifiers. The first approximation in that sequence classifies objects relatively fast, but also has a relatively high error rate. Each successive approximation in the sequence is slower and more accurate than the previous one. These approximations are combined in a cascade structure, whereby easy cases are classified by earlier classifiers, and harder cases are passed on to the slower but more accurate classifiers. An interesting empirical property of classification using such cascades is that, in our experiments, both classification time and error rate actually decrease as the size of the database increases. This behavior is in stark contrast to the behavior of typical nearest neighbor classification systems, where classification time increases with the size of the database.

1.5 Overview of Thesis

Chapter 2 establishes some standard terminology and notation, defines basic concepts, and offers a brief survey of existing embedding methods that can be used for speeding up nearest neighbor retrieval. Chapter 3 takes a look at existing methods for efficient nearest neighbor retrieval and classification and outlines the main differentiating features and advantages of the methods proposed in this thesis.

Chapters 4, 5 and 6 described the three main contributions of this thesis. Chapter 4 describes the BoostMap method for embedding construction. We show that constructing embeddings for efficient retrieval can be framed as a machine learning problem, and we propose a learning-based embedding construction algorithm that directly maximizes the amount of nearest neighbor structure preserved by the embedding. Chapter 5 defines query-sensitive embeddings, discusses the properties of those embeddings, and describes how to apply the BoostMap method for producing such embeddings. Chapter 6 discusses cascades of approximate nearest neighbor classifiers, and describes how to construct and use such cascades in order to achieve efficient classification.

In Chapter 7, the proposed methods are evaluated experimentally on several different applications: 3D hand pose estimation, off-line character recognition, online character recognition, and efficient retrieval of time series. In all data sets, the proposed methods lead to significant improvements in accuracy and efficiency compared to existing state of the art methods. In some datasets, the general-purpose methods introduced in this thesis even outperform domain-specific methods that were specifically designed to work well with such datasets.

Chapter 2

Background

This chapter provides some background on embedding-based methods for efficient nearest neighbor retrieval in non-Euclidean spaces. Section 2.1 provides a formal definition of embeddings and establishes notation for some of the key concepts. Section 2.2 discusses various measures of embedding quality that have been previously used in the literature. Section 2.3 reviews existing methods for constructing embeddings that can be used for an efficient retrieval. Finally, Section 2.4 describes the filter-and-refine retrieval framework, a popular retrieval framework that embedding methods are typically used in conjunction with. For reference, a list of symbols that are used throughout the thesis is provided in Tables 2.1, 2.2.

2.1 Some Basic Definitions and Notation

Let $\mathbb{X}$ be a space of objects, and D be a distance measure in $\mathbb{X}$. Distance D can be metric or non-metric. Let database $\mathbb{U}$ be a finite subset of $\mathbb{X}$, containing $|U|$ objects. Let $Q \in \mathbb{X}$ be a query object, and suppose we want to find the k nearest neighbors of Q in $\mathbb{U}$. The brute-force method is to measure the distance $D(Q, U)$ between the query and every $U \in \mathbb{U}$. Obviously, brute-force search requires $|U|$ distance evaluations.

If $|U|$ is large and D is computationally expensive to compute, brute-force search can be too slow to be practical. A way to speed up retrieval is to construct an embedding that maps objects into another space, where distances can be computed more efficiently. Typically we construct an embedding $F : \mathbb{X} \rightarrow \mathbb{R}^d$ into the d-dimensional real vector space $\mathbb{R}^d$, where distances are measured using a weighted Minkowski (L_p) metric like the Euclidean (L_2) distance or the Manhattan (L_1) distance.

Symbol	Meaning
A	object of space $\mathbb{X}$.
$A_i(Q)$	query-sensitive weight of the i-th embedding dimension, for query Q.
$A_{\min}(h, j, l)$	value of $\alpha \geq l$ that minimizes $Z_j(h, \alpha)$.
B	object of space $\mathbb{X}$.
c	variable ranging between 1 and K_{j-1}.
C	object in $\mathbb{X}$, in Chapter 4. class label, in Chapter 6.
c_1	$\frac{1}{c_1}$ is the minimum scaling of the original distance measure by the embedding.
c_2	maximum scaling of the original distance measure by the embedding.
$\mathbb{C}$	candidate objects used by the training algorithm.
$\mathbb{C}_j$	candidate pairs of pivot objects at training round j.
d	number of dimensions of the embedding target space.
D	distance measure in the original space $\mathbb{X}$ of the embedding.
e	error threshold in the cascade construction algorithm.
f	aggregate function, computes the aggregate of a set of features, in alignment-based embeddings.
F	embedding from original space $\mathbb{X}$ to a vector space.
F_{out}	embedding constructed by the training algorithm.
$\tilde{F}$	classifier of triples of objects defined using embedding F.
F'	one-dimensional embedding.
$\mathbb{F}_j$	set of candidate 1D embeddings to try at training round j.
$\mathbb{F}_{j1}$	set of reference-object embeddings to try at training round j.
$\mathbb{F}_{j2}$	set of line projection embeddings to try at training round j.
g	index used to specify weak classifiers selected before training round j.
$G(\tilde{F}, X, A, B)$	measure of the classification error of $\tilde{F}$ on triple (X, A, B).
$\mathcal{G}$	set of training objects for AdaBoost.
H	strong classifier produced by AdaBoost.
H_j	strong classifier assembled by AdaBoost after the first j training rounds.
$\mathbb{H}_j$	set of classifiers that pass the training error test at training round j.
h	weak classifier.
h_i	weak classifier for feature selection in alignment-based embeddings.
h_j	weak classifier selected by AdaBoost at training round j.
h'_j	j-th unique weak classifier appearing in strong classifier H.
J	total number of training rounds performed by AdaBoost.
k	number of nearest neighbors to retrieve, especially for k-nearest neighbor classification.
$k_{\max}$	maximum number of nearest neighbors we want to retrieve in an application.
$K(Q, P_i)$	measure of confidence in the classification of query Q.
K_j	number of unique weak classifers selected during the first j training rounds.
l	specifies a minimum value for the weight of a weak classifier.
m	dimensionality of Euclidean space that FastMap assumes the original space is equal to, in Chapter 2. index of coordinates in Chapter 5.
o_i	i-th training object in AdaBoost.
p	parameter for the refine step of filter-and-refine retrieval.
P	reference object, element of $\mathbb{X}$.
$\mathbb{P}$	reference set, subset of $\mathbb{X}$.

Table 2.1: Table of the main symbols used throughout this thesis, part 1.

Symbol	Meaning
Q	the query object, element of $\mathbb{X}$.
$\mathbb{R}$	the set of real numbers.
s	number of classifiers in a cascade of approximate nearest neighbor classifiers.
$S(Q)$	splitter: a function mapping each query Q to either 0 or 1.
t_i	threshold for the i-th step in the cascade.
$\mathbb{T}$	training set, used for selecting training triples.
u	vector in $\mathbb{R}^d$.
$\mathbb{U}$	database, subset of $\mathbb{X}$.
v	vector in $\mathbb{R}^d$.
V	set used to define a splitter.
$w_{i,j}$	weight of i-th training object during j-th training round.
W	normalization factor for making an embedding contractive.
$W(Q)$	normalization factor for making a query-sensitive embedding contractive.
$W(A, B)$	alignment of objects A and B: set of pairs of correspondences between features of A and features of B.
X	element of $\mathbb{X}$.
$\mathbb{X}$	original space of an embedding.
Y	set of all class labels for a particular classification problem.
$y(X)$	class label of X.
z_j	constant used for normalizing training weights, equal to $Z_j(h_j, \alpha_j)$.
$Z_j(h, \alpha)$	a measure of how useful it is to set $h_j = h, \alpha_j = \alpha$, at the j-th training round of AdaBoost.
$Z_{\min}(h, j, l)$	a measure of how useful it is to set $h_j = h$, at the j-th training round of AdaBoost.
$Z_{\max}$	we stop AdaBoost when $z_j \geq Z_{\max}$.
α_j	weight of weak classifier h_j.
α'_j	training weight for unique classifier h'_j, also weight of j-th coordinate or L_1 distance measure.
β	number of training triples used in the training algorithm for embedding construction.
γ	number of classifiers to consider at each training round.
Γ	query-sensitive classifier of triples of objects.
Δ	distance in the target space $\mathbb{R}^d$ of the embedding.
δ	number of classifiers selected based on training error at each training round of AdaBoost.
$\Lambda_j(h)$	training error of weak classifier h at round j.
τ	threshold used to define a splitter.

Table 2.2: Table of the main symbols used throughout this thesis, part 2.

Let Δ denote the distance measure used in $\mathbb{R}^d$. In addition to the notation $F : \mathbb{X} \rightarrow \mathbb{R}^d$, we will also frequently use the notation $F : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$, to emphasize that an embedding F is not fully specified without Δ. The embedding construction algorithms proposed in chapters 4 and 5 optimize simultaneously both an embedding F and the corresponding distance measure Δ.

2.2 Measures of Embedding Quality

An embedding $F : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$ is called *isometric* when it perfectly preserves distances, i.e., when it holds for all $X_1, X_2 \in \mathbb{X}$ that

$$D(X_1, X_2) = \Delta(F(X_1), F(X_2)) \ . \tag{2.1}$$

An embedding F is called *proximity preserving* when it perfectly preserves proximity relations between triples of objects, i.e., when it holds for all $X_1, X_2, X_3 \in \mathbb{X}$ that

$$\begin{aligned} & D(X_1, X_2) \leq D(X_1, X_3) \Leftrightarrow \\ & \Delta(F(X_1), F(X_2)) \leq \Delta(F(X_1), F(X_3)) \ . \end{aligned} \tag{2.2}$$

In most cases it is impossible to find isometric or proximity-preserving embeddings from a non-Euclidean space $\mathbb{X}$ to $\mathbb{R}^d$, regardless of the value of d. In such cases, it is typically desired that the distances $\Delta(F(X_1), F(X_2))$ in $\mathbb{R}^d$ closely approximate distances $D(X_1, X_2)$ in $\mathbb{X}$. Different measures can be used to evaluate how good those approximations are. The most commonly used such measures are *distortion* and *stress*. We will define those measures following (up to some changes in notation) the description in [Hjaltason and Samet, 2003a].

The distortion of embedding F is defined as the lowest value $c_1 c_2$ that guarantees that

$$\frac{1}{c_1} \cdot D(X_1, X_2) \leq \Delta(X_1, X_2) \leq c_2 \cdot D(X_1, X_2) \ , \tag{2.3}$$

for all pairs of objects $X_1, X_2 \in \mathbb{X}$. Note that F is isometric if and only if the distortion is equal to 1.

Whereas the distortion measures the worst-case deviation of distances, stress is a measure of the average deviation. One common definition of stress is:

$$\frac{\sum_{X_1,X_2} (\Delta(F(X_1), F(X_2)) - D(X_1, X_2))^2}{\sum_{X_1,X_2} D(X_1, X_2)^2} \ . \tag{2.4}$$

Clearly, if an embedding F is isometric then it has stress equal to 0.

2.3 Embedding Methods for Indexing

Several existing embedding methods can be used for speeding up nearest neighbor retrieval. In this section we briefly go over Lipschitz embeddings, SparseMap, FastMap, and MetricMap. The reader is referred to [Hjaltason and Samet, 2003a] for a thorough analysis of these methods.

2.3.1 Lipschitz Embeddings

Given any space $\mathbb{X}$ with a distance measure D, we can extend the definition of D so as to define a distance between elements of $\mathbb{X}$ and subsets of $\mathbb{X}$. Let $X \in \mathbb{X}$ and $\mathbb{P} \subset \mathbb{X}$. Then,

$$D(X, \mathbb{P}) = \min_{P \in \mathbb{P}} D(X, P) \ . \tag{2.5}$$

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

$$F^{\mathbb{P}}(X) = D(X, \mathbb{P}) \ . \tag{2.6}$$

The set $\mathbb{P}$ that is used to define $F^{\mathbb{P}}$ is called a *reference set*. In many cases $\mathbb{P}$ can consist of a single object P, which is typically called a *reference object* or a *vantage object*. In that case, we denote the embedding as F^P:

$$F^P(X) = D(X, P) \ . \tag{2.7}$$

If D obeys the triangle inequality, F^P maps nearby points in $\mathbb{X}$ to nearby points on the real line $\mathbb{R}$. In quantitative terms, it can be shown using the triangle inequality that,

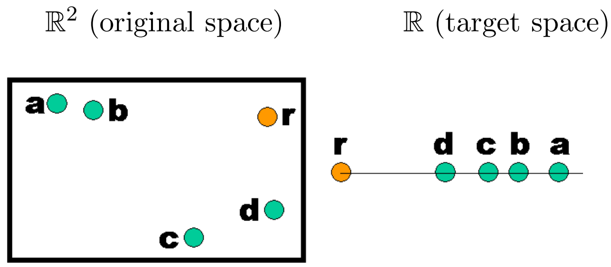

Figure 2·1: An embedding F^r of five 2D points (shown on the left) into the real line (shown on the right), using r as the reference object. The target of each 2D point on the line is labeled with the same letter as the 2D point. We see that F maps all pairs of points closer to each other than they are in the original space. As a result, nearby points in $\mathbb{R}^2$, like a and b, are always mapped to nearby points in $\mathbb{R}$. At the same time, points that are relatively far from each other in $\mathbb{R}^2$ may also be mapped to nearby points in $\mathbb{R}$, as is the case for points b and c.

for any $X_1, X_2 \in \mathbb{X}$, $|F^P(X_1) - F^P(X_2)| \leq D(X_1, X_2)$. In many cases D may violate the triangle inequality for some triples of objects (an example is the chamfer distance [Barrow et al., 1977]), but F^P may still map nearby points in $\mathbb{X}$ to nearby points in $\mathbb{R}$, at least most of the time. On the other hand, distant objects may also map to nearby points, if they happen to have similar distances to the reference set $\mathbb{P}$ (Figure 2·1).

In order to reduce the likelihood that distant objects get mapped to nearby points, we can define a multidimensional embedding $F : X \to \mathbb{R}^d$, by choosing d different reference sets $\mathbb{P}_1, ..., \mathbb{P}_d$:

$$F(x) = (F^{\mathbb{P}_1}(x), ..., F^{\mathbb{P}_d}(x)) \ . \tag{2.8}$$

These embeddings are called *Lipschitz embeddings* [Bourgain, 1985, Hristescu and Farach-Colton, 1999, Hjaltason and Samet, 2003a]. Bourgain embeddings [Bourgain, 1985, Hjaltason and Samet, 2003a] are a special type of Lipschitz embeddings. Given a database $\mathbb{U} \subset \mathbb{X}$, such that $\mathbb{U}$ contains n objects, we choose $\lfloor \log n \rfloor^2$ reference sets. In particular, for each $i = 1, ..., \lfloor \log n \rfloor$ we choose $\lfloor \log n \rfloor$ reference sets, each with 2^i elements. The elements of each reference set are picked randomly. Bourgain embeddings are optimal in some sense: on the task of embedding metric spaces to a Euclidean space, Bourgain embeddings achieve $O(\log n)$ distortion, and there exist finite spaces $\mathbb{U}$ for which no better distortion can be achieved. More details can be found in [Hjaltason and Samet, 2003a] and [Linial et al., 1994].

A weakness of Bourgain embeddings is that, in order to compute the embedding of a query object Q, distances D between Q and almost all objects in database $\mathbb{U}$ must be computed. The reason for that is that there are $\lfloor \log n \rfloor$ reference sets containing $2^{\lfloor \log n \rfloor}$ objects each, and $2^{\lfloor \log n \rfloor} \geq \frac{n}{2}$. Computing distances between the query object and the majority of database objects is exactly what we want to avoid, in order to achieve better efficiency than brute-force search. Consequently, Bourgain embeddings, at least in their original formulation, are not useful for speeding up nearest neighbor retrieval. SparseMap [Hristescu and Farach-Colton, 1999] 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 to database objects. Since SparseMap can compute the embedding of a query by comparing the query to only a small fraction of the database, SparseMap can be used for speeding up nearest neighbor retrieval.

It is important to note that Bourgain embeddings can also be applied in non-metric spaces, but no theoretical bounds on distortion can be provided for Bourgain embeddings of arbitrary non-metric spaces.

A simple and attractive alternative to Bourgain embeddings is to simply use Lipschitz embeddings in which all reference sets are singleton. In particular, if $P_1, \ldots, P_d \in \mathbb{U}$, we can define an embedding $F(X)$ as follows:

$$F(X) = (D(X, P_1), \ldots, D(X, P_d)) \ . \tag{2.9}$$

With this type of embeddings, in order to compute a d-dimensional embedding of a previously unseen object Q we only need to compute the distance between Q and the d reference objects $P_1, \ldots, P_d$.

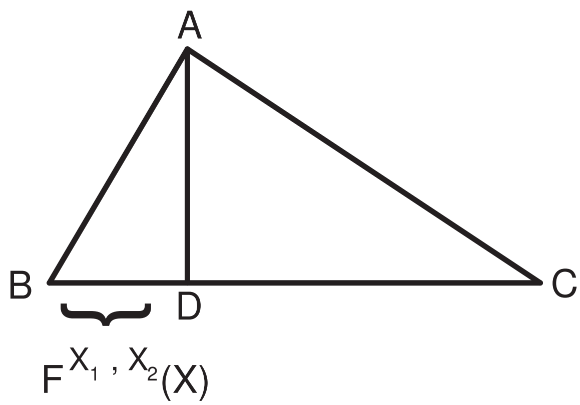

Figure 2·2: Computing $F^{X_1,X_2}(X)$, as defined in Equation 2.10: we construct a triangle ABC so that the sides AB, AC, BC have lengths $D(X, X_1), D(X, X_2)$ and $D(X_1, X_2)$ respectively. We draw from A a line perpendicular to BC, and D is the intersection of that line with BC. The length of the line segment BD is equal to $F^{X_1,X_2}(X)$.

2.3.2 FastMap and MetricMap

A family of simple one-dimensional embeddings is proposed in [Faloutsos and Lin, 1995] and used as building blocks for FastMap. The idea is to choose two objects $X_1, X_2 \in \mathbb{X}$, called *pivot objects*, and then, given an arbitrary $X \in \mathbb{X}$, 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 2·2, the projection can be defined by treating the distances between X, X_1, and X_2 as specifying the sides of a triangle in $\mathbb{R}^2$:

$$F^{X_1,X_2}(X) = \frac{D(X, X_1)^2 + D(X_1, X_2)^2 - D(X, X_2)^2}{2D(X_1, X_2)} \,. \tag{2.10}$$

If $\mathbb{X}$ is Euclidean, then F^{X_1,X_2} corresponds indeed to a projection of $\mathbb{X}$ to the line passing through X_1 and X_2. Therefore, If $\mathbb{X}$ is Euclidean, F^{X_1,X_2} maps nearby points in $\mathbb{X}$ to nearby points in $\mathbb{R}$. In practice, even if $\mathbb{X}$ is non-Euclidean, F^{X_1,X_2} often still preserves some of the proximity structure of $\mathbb{X}$.

FastMap [Faloutsos and Lin, 1995] uses multiple pairs of pivot objects to project a database $\mathbb{U} \subset \mathbb{X}$ into $\mathbb{R}^d$. 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 $\mathbb{U}$ are "updated," so that they correspond to projections into the "hyperplane" perpendicular to the line $X_1 X_2$. Those projections are computed again by treating distances between objects in $\mathbb{U}$ as Euclidean distances in some space $\mathbb{R}^m$ of unspecified dimensionality m. After distances are updated, FastMap is recursively applied again to choose a next pair of pivot objects and apply another round of distance updates. Computing the embedding of the entire database requires measuring only $O(d|\mathbb{U}|)$ distances between pairs of objects. Given a previously unseen query $X \in (\mathbb{X} - \mathbb{U})$, computing the embedding of Q requires measuring only $2d$ distances. Although FastMap treats $\mathbb{X}$ as a Euclidean space, the resulting embeddings can be useful even when $\mathbb{X}$ is non-Euclidean, or even non-metric. We have seen that in our own experiments (Chapter 7).

MetricMap [Wang et al., 2000] is an extension of FastMap, that maps $\mathbb{X}$ into a a pseudo-Euclidean space. The experiments in [Wang et al., 2000] report that MetricMap tends to do better than FastMap when $\mathbb{X}$ is not Euclidean.

2.4 Embedding Application: Filter-and-Refine Retrieval

Suppose that, using some embedding method, we have constructed an embedding F from a space $\mathbb{X}$ with a computationally expensive distance measure D to vector space $\mathbb{R}^d$ with distance measure $\Delta = L_p$ for some value of p. The question is: how can we use F to speed up retrieval of the k-nearest neighbors of some query object Q? A popular and commonly used answer to this question is the filter-and-refine framework [Hjaltason and Samet, 2003a], in which retrieval is done as follows:

- Offline preprocessing step: compute and store vector $F(U)$ for every database object $U \in \mathbb{U}$.

- Online retrieval, given a query object Q, number k of nearest neighbors to retrieve, and a free parameter p:

– Embedding step: compute $F(Q)$.

– Filter step: rank all database objects in increasing order of the distance of their embeddings from $F(Q)$.

– Refine step: rerank the p highest-ranked database objects by evaluating the exact distance D between those objects and Q.

– Output: return the k highest-ranked database objects.

The filter step provides an approximate preliminary ranking of database objects by comparing d-dimensional vectors using distance measure Δ. The refine step applies D only to the top p candidates. Assuming that Δ is significantly more efficient than D, filter-and-refine retrieval is much more efficient than brute-force retrieval, in which we compute distance D between Q and every database object.

It is important to note that, in the general case, filter-and-refine retrieval does not guarantee that the correct k nearest neighbors will be retrieved. If X is one of the k nearest neighbors of Q, and $p \geq k$, X will be retrieved if and only if it is included in the p objects that survive the filter step. In other words, X will be correctly retrieved if and only if $F(X)$ is one of the p nearest neighbors of $F(Q)$ among the embeddings of all database objects. Clearly, parameter p provides a trade-off between retrieval accuracy and efficiency. Given a query, as p increases it becomes increasingly likely that the true k nearest neighbors of the query will be included in the top p candidates found at the filter step. At the same time, as p increases, more exact distances D must be computed at the refine step.

In an implementation of embedding-based filter-and-refine retrieval, two parameters must be specified: parameter p, which is the number of exact distances to compute at the refine step, and parameter d, which is the dimensionality of the embedding. As d increases, the embedding step becomes more expensive, since typically embedding a query object requires $O(d)$ distances. Comparing d-dimensional vectors also becomes more expensive. At the same time, using a higher-dimensional embedding may yield more accurate results in the filter step, thus allowing a smaller value for p. The best choice of p and d will depend on domain-specific parameters like the desired number k of nearest neighbors, the time it takes to compute a single distance D, the time it takes to compare d-dimensional vectors, and the desired retrieval accuracy, i.e. how often we are willing to miss some of the true k nearest neighbors.

We should emphasize that filter-and-refine retrieval is a general framework, and a variety of different methods can be used for the filtering step, including methods that are not embedding-based. In principle, for filtering we can use any method that can efficiently eliminate database objects from the nearest neighbor search without needing to measure exact distances. In the next chapter, where we overview existing methods for nearest neighbor retrieval, we will provide more examples of methods that can be used for the filtering step.

Chapter 3

Related Work

Various methods have been employed for speeding up nearest neighbor retrieval in high-dimensional and non-Euclidean spaces. The reader can refer to [Böhm et al., 2001, Hjaltason and Samet, 20 Hjaltason and Samet, 2003b, White and Jain, 1996] for comprehensive reviews of existing nearest neighbor methods. Existing methods can be classified into different categories, according to the following criteria:

- Assumptions made about the space and distance measure. A method may be applicable to Euclidean spaces, vector spaces with general L_p metrics, metric spaces, arbitrary spaces with arbitrary (even nonmetric) distance measures, or specific spaces and distance measures.

- Exact or approximate results. Some methods guarantee retrieval of the true nearest neighbors, while other methods only provide approximate results.

- Pruning-based or approximation-based retrieval. Pruning-based methods use pruning strategies to eliminate from consideration large portions of the database during the search for nearest neighbors. Such methods can achieve retrieval time complexity that is sublinear to the number of database objects. Approximation-based methods, on the other hand, achieve computational savings by computing a fast approximate distance measure, or lower/upper bounds of the distance between objects, instead of computing the original, computationally expensive distance measure. The complexity of these methods is linear to the number of database objects, as in brute-force search. However, using fast approximate estimates of distances can lead to speed-ups of orders of magnitude in practice, compared to brute-force search.

We will first go over indexing methods that are only applicable in vector spaces, then we will survey methods that are applicable in non-vector spaces, and finally we will briefly discuss methods that can be used for speeding up nearest neighbor classification.

3.1 Indexing Methods for Vector Spaces

A large amount of work focuses on efficient nearest neighbor retrieval in multidimensional vector spaces. Methods that provide exact results are proposed in [Weber et al., 1998, Sakurai et al., 2000, Chakrabarti and Mehrotra, 2000]. In [Weber et al., 1998] exact retrieval is performed efficiently using an indexing structure called "vector-approximation files", or "VA-files" in short. The data space is divided into 2^b rectangular cells, where b is a user-specified number of bits to describe each cell. Every object is represented simply by the bits that describe what cell it belongs to. At the filtering step, upper and lower bounds can be derived for the distance between the query and each object using the bits that represent each database object. An alternative method, A-trees, is proposed in [Sakurai et al., 2000], where a tree indexing structure stores bounding rectangle information about subregions of the database and individual data points. In [Chakrabarti and Mehrotra, 2000], clustering is used to identify subsets of the database where the objects are correlated. PCA is applied to each of the clusters separately, and a separate indexing structure is built for each cluster. By applying dimensionality reduction separately to each cluster, this method achieves higher efficiency than methods that apply global dimensionality reduction.

Approximate methods are proposed in [Li et al., 2002, Egecioglu and Ferhatosmanoglu, 2000, Kanth et al., 1998, Weber and Böhm, 2000, Koudas et al., 2004, Tuncel et al., 2002]. Clindex [Li et al., 2002] is a method that clusters the data and then builds an index structure that identifies the clusters of interest given a query object. In [Egecioglu and Ferhatosmanoglu, 2000] inner products of high-dimensional vectors are efficiently approximated using a relatively low-dimensional representation of each object. In [Kanth et al., 1998] an efficient method is proposed for nearest neighbor retrieval in dynamic datasets. In particular, an approximation of the singular value decomposition of the database is efficiently recomputed, thus addressing the limitation of other dimensionality reduction methods that assume that the database is known in advance. A modified version of VA-files is described in [Weber and Böhm, 2000], that derives approximate lower bounds for the distance between the query and database objects. Compared to the original VA-files, this method sacrifices the guarantee of always producing correct retrieval results, but achieves significant improvements in retrieval efficiency. In [Koudas et al., 2004] each object is represented with a set of bits, and subsets of those bits are used to prune out large portions of the database given a query object. The binary representation is constructed by clustering the database objects and then comparing each object in each dimension with each cluster center. An alternative to VA-files is proposed in [Tuncel et al., 2002], where vector quantization is used instead of the fixed grid-based partition used in VA-files.

Particular mention should be made to Locality Sensitive Hashing (LSH) [Gionis et al., 1999], an approximate nearest neighbor method for vector spaces that has been shown theoretically to scale well with the number of dimensions and has produced good results in practice, even in very high-dimensional spaces [Frome et al., 2004, Grauman and Darrell, 2004, Shakhnarovich et al., 2003]. LSH constructs multiple hashing functions, where each hashing function maps objects into binary strings. Ideally, given a query object Q and one of its near neighbors A (where "near neighbor" is simply an object within some pre-specified distance from the query), the desired behavior of a hashing function is to map both Q and A to the same binary string. We say that two objects "collide" under a hashing function when that function maps them to the same binary string. Given Q and an object B that is not a near neighbor of Q, ideally we want Q and B not to collide. In LSH, after the query is mapped to a binary string by each hashing function, exact distances are computed between the query and all database objects that collide with the query under at least one hashing function. By appropriately choosing the length of the binary strings and the number of hashing functions to construct, the probability that the query will collide with one of its near neighbors under at least one hashing function becomes relatively high, while the overall number of database objects colliding with the query under at least one hashing function remains relatively low. Consequently, by only evaluating exact distances between the query and objects that the query collides with, a near neighbor can be retrieved with high probability, without needing to consider the vast majority of database objects.

For the purposes of this thesis, the key limitation of all the methods we have discussed in this section is that they are mainly applicable to vector spaces with L_p metrics, and they are not designed for arbitrary distance measures. Such methods cannot be applied to the non-vector spaces induced by computationally expensive distance measures such as the measures described in Section 1.3. In contrast, the methods proposed in this thesis can be applied to arbitrary spaces with computationally expensive distance measures.

3.2 Indexing Methods for Non-Vector Spaces

A more general class of spaces, that includes real vector spaces with L_p metrics, is the class of metric spaces, i.e., spaces with a metric distance measure. A distance measure D is metric if it is symmetric ($D(A, B) = 0$ iff $A = B$), reflexive ($D(A, B) = D(B, A)$) and obeys the triangle inequality ($D(A, X) + D(X, B) \geq D(A, B)$). Examples of metric distance measures are the edit distance for strings [Levenshtein, 1966], the Hausdorff distance for edge images [Huttenlocher et al., 1993], bipartite matching for sets [Kuhn, 1955], or the Earth Mover's Distance (EMD) [Rubner et al., 1998] for distributions with equal mass. A number of nearest neighbor methods have been designed that are applicable to arbitrary metric spaces. We will discuss some of those methods in the next paragraphs; the reader is referred to [Hjaltason and Samet, 2003b] for a comprehensive survey of such methods.

VP-trees [Yianilos, 1993] hierarchically partition the database into a tree structure by partitioning, at each node, the set of objects based on whether they are closer than a threshold to a specific object, called a *pivot* object. A similar structure, called metric trees, has been proposed independently in [Uhlman, 1991]. MVP-trees [Bozkaya and Özsoyoglu, 1999] are an extension of VP-trees, where multiple pivot points are used at each node. M-trees [Ciaccia et al., 1997] are a variant of metric trees explicitly designed for dynamic databases. Slim-trees [Jr. et al., 2000] further improve on M-trees by minimizing the overlap between nodes. While the above methods are exact, an approximate variant of M-trees is proposed in [Zezula et al., 1998], that can achieve significant additional speed-ups by sacrificing the guarantee of always retrieving the true nearest neighbors.

The above tree-based methods for nearest neighbor retrieval in metric spaces are pruning based. An alternative class of methods are approximation-based methods, where every database object is considered during the search for the nearest neighbor of a particular query object, but most database objects are eliminated using efficiently computed approximations or bounds of the true distance between those objects and the query. AESA [Vidal, 1994] and LAESA [Micó and Vidal, 1994] compute the exact distance between the query and a small set of database objects and then use the triangle inequality to establish lower bounds of the distance between the query and the rest of the database objects.

In domains where the distance measure is computationally expensive, significant computational savings can be obtained by constructing a distance-approximating embedding that maps objects into another space with a more efficient distance measure. A number of methods have been proposed for embedding arbitrary spaces into a Euclidean or pseudo-Euclidean space [Bourgain, 1985, Faloutsos and Lin, 1995, Hristescu and Farach-Colton, 1999, Roweis and Saul, 2000, Tenenbaum et al., 2000, Wang et al., 2000, Young and Hamer, 1987]. Some of these methods, in particular MDS [Young and Hamer, 1987], Bourgain embeddings [Bourgain, 1985, Hjaltason and Samet, 2003a], LLE [Roweis and Saul, 2000] and Isomap [Tenenbaum et al., 2000] are not targeted at speeding up nearest neighbor retrieval for previously unseen query objects, because to embed the query those methods still need to evaluate exact distances between the query and most or all database objects. Online queries can be efficiently handled by Lipschitz embeddings [Hjaltason and Samet, 2003a], FastMap [Faloutsos and Lin, 1995], MetricMap [Wang et al., 2000], and SparseMap [Hristescu and Farach-Colton, 19 as described in Chapter 2.

Embedding methods are typically used within the filter-and-refine retrieval framework [Hjaltason and Samet, 2003a]. In filter-and-refine retrieval a computationally efficient approximation of the original distance (obtained, for example, using embeddings) is utilized in the filtering step, in order to prune a large part of the search space. Then, the original, accurate but more expensive distance measure is applied to the few remaining candidates, during the refine step. Usually, the target distance measure of the embedding is designed to be metric, even if the original distance is not, so that traditional indexing techniques can be applied to index the database in order to speed up the filtering stage as well. If the distance measure in the original space is metric, then SparseMap (as modified in [Hjaltason and Samet, 2003a]) and Lipschitz embeddings satisfy a property called *contractiveness*, discussed in Section 4.4.1. Contractive embeddings can be used in a way that guarantees retrieval of the true nearest neighbors after the refine step. FastMap and MetricMap, on the other hand, are not contractive, and thus using those methods within filter-and-refine retrieval there are no guarantees of retrieving the true nearest neighbors.

BoostMap, and its query-sensitive extension, which are two of the three main contributions of this thesis, were introduced in [Athitsos et al., 2004, Athitsos et al., 2005b] and belong to the same family of approaches as the methods in [Faloutsos and Lin, 1995, Hjaltason and Samet, 2003a, Hristescu and Farach-Colton, 1999, Wang et al., 2000]; they are embedding methods designed for achieving efficient nearest neighbor retrieval. Like SparseMap and Lipschitz embeddings, BoostMap is contractive and can be used for exact retrieval in metric spaces. A key difference between BoostMap and existing embedding methods is that, in BoostMap, embedding construction optimizes a direct measure of the amount of nearest neighbor structure preserved by the embedding.

As noted in Section 1.3, there are many situations in pattern recognition where we need to use distance measures that are non-metric. Methods that are designed for general metric spaces can still be applied when the distance measure is non-metric. However, since the metric assumptions are violated, applications of metric methods to non-metric spaces are heuristic. Methods that are exact for metric spaces become approximate in non-metric spaces. No theoretical guarantees of accuracy can be made for arbitrary non-metric spaces, since in arbitrary non-metric spaces the distances between the query and database objects do not have to satisfy any constraints. Like other existing methods, BoostMap can offer no guarantees of performance in arbitrary non-metric spaces. However, a fundamental difference between BoostMap and other nearest neighbor methods is that the embedding optimization criterion used in the BoostMap algorithm is equally valid for arbitrary spaces, including metric and non-metric spaces. Therefore, even in non-metric spaces, the algorithm maximizes the amount of nearest neighbor structure preserved by the embedding that is being constructed.

In several domains where BoostMap is applicable, domain-specific methods have been proposed for speeding up similarity queries. For time series databases, various techniques have appeared in the literature for robust evaluation of similarity queries when using non-metric distance functions [Keogh, 2002, Vlachos et al., 2003, Yi et al., 1998]. These techniques use the filter-and-refine approach, and use efficient distance approximations in the filter step.

One of the distance measures that we applied BoostMap to in the experiments is shape context matching [Belongie et al., 2002]. Several methods have been proposed for speeding up similarity matching and classification using shape context. In [Mori et al., 2001], efficient retrieval is attained by pruning based on comparisons of a small subset of shape context features, and also using vector quantization on the space of those features. In [Grauman and Darrell, 2004] shape context features are matched based on the Earth Mover's Distance (EMD). The EMD is efficiently approximated using an embedding, and then Locality Sensitive Hashing is applied on top of the embedding. In [Zhang and Malik, 2003] a discriminative classifier is learned based on correspondences of shape context features between the test object and a small number of prototypes per class.

A fundamental difference between our method and the above-mentioned methods for speeding up time series matching and shape context matching is that our method is domain-independent, and treats the underlying distance measure as a black box. It is natural that a method that uses domain-specific knowledge can sometimes lead to better performance than a domain-independent method. At the same time, our method does outperform some domain-specific methods [Zhang and Malik, 2003, Vlachos et al., 2003] in our experiments with shape context matching and time series, and thus may be a viable alternative in applications where currently domain-specific methods are being used to improve efficiency.

3.3 Methods for Efficient Nearest Neighbor Classification

The third main contribution of this thesis is a method for speeding up nearest neighbor classification using cascades of approximate nearest neighbor classifiers. This method was first introduced in [Athitsos et al., 2005a]. An alternative family of methods that can also be used to improve the efficiency of nearest neighbor classifiers are condensing methods [Devi and Murty, 2002, Gates, 1972, Hart, 1968]. Those methods try to identify and remove training objects whose removal does not negatively affect classification accuracy. In principle, condensing methods are orthogonal to indexing methods, and could be used together to further increase efficiency. However, a weakness of many condensing methods is that, although the objects they remove do not affect classification accuracy as measured on the *training* set, they generalize poorly and they often lead to significantly worse classification accuracy on objects not included in the training set. This is illustrated in our experiments, where we compare our method to the condensing method described in [Hart, 1968].

A recently proposed method for efficient nearest neighbor classification is the Pyramid Match Kernel [Grauman and Darrell, 2005], which is applicable to spaces where objects are represented as sets of features. The Pyramid Match Kernel compares two images in time linear to the number of features, thus providing an efficient alternative to computationally expensive distance measures that take superlinear time. Computational savings are obtained by matching features hierarchically, in a coarse-to-fine manner, using multiple feature histograms of different granularity. One limitation of the Pyramid Match Kernel is that it cannot enforce certain constraints that are often useful to impose on feature correspondences, such as the temporal monotonicity constraint in time series [Kruskall and Liberman, 1983], or ordering constraints in strings and biological sequences. In addition, the embedding-based methods described in this thesis can also be applied in domains where distance measures are not based on feature correspondences, such as peer-to-peer networks. At the same time, methods for efficient approximate feature matching, where applicable, are an alternative to combining indexing methods with computationally expensive distance measures, which is the focus of this thesis.

3.4 Summary of Related Work

Tables 3.1 summarizes the methods for efficient nearest neighbor retrieval that we have gone over in this chapter. Overall, the key differentiating features of the methods proposed in this thesis are the following:

- The proposed methods can be applied to non-vector spaces, whereas the majority of efficient retrieval methods can only be applied to vector spaces.

- Existing methods for non-vector spaces are based on Euclidean or metric assumptions, although these methods can sometimes produce useful results even when heuristically applied in spaces that violate those assumptions. In contrast, the methods proposed in this thesis do not make any assumptions about the geometry of the space.

- With respect to methods for efficient nearest neighbor classification, such methods typically focus on reducing the size of the database by attempting to identify objects that can be discarded without negatively affecting classification accuracy. The methods proposed in this thesis are orthogonal to such methods, and can be applied to further speed up nearest neighbor retrieval in the reduced datasets.

Methods for Efficient Nearest Neighbor Retrieval				
method	Applicability	Assumptions	Exact	Pruning
Weber 1998	vector spaces		yes	no
Sakurai 2000	vector spaces		yes	yes
Chakrabarti 2000	vector spaces		yes	yes
Li 2002	vector spaces		no	yes
Egecioglu 2000	vector spaces		no	no
Kanth 1998	vector spaces		no	no
Weber 2000	vector spaces		no	no
Koudas 2004	vector spaces		no	yes
Tuncel 2002	vector spaces		no	yes
LSH	vector spaces		no	yes
VP-trees	arbitrary spaces		metric	yes
MVP-trees	arbitrary spaces	metric space	metric	yes
M-trees	arbitrary spaces	metric space	metric	yes
Slim-trees	arbitrary spaces	metric space	metric	yes
Zezula 1998	arbitrary spaces	metric space	no	yes
AESA	arbitrary spaces	metric space	metric	no
LAESA	arbitrary spaces	metric space	metric	no
Lipschitz embeddings	arbitrary spaces	metric space	metric	no
FastMap	arbitrary spaces	metric space	no	no
SparseMap	arbitrary spaces	Euclidean space	metric	no
MetricMap	arbitrary spaces	metric space	no	no
BoostMap	arbitrary spaces	none	metric	no
Query-Sensitive BoostMap	arbitrary spaces	none	metric	no

Table 3.1: Methods for efficient nearest neighbor netrieval, and some of the key properties of each method: what spaces it is applicable to, whether it guarantees retrieving the exact nearest neighbors, geometric assumptions that the method is based on, and whether it is pruning-based or approximation-based. Under the "Exact" column, a method denoted as "metric" guarantees retrieval of exact nearest neighbors only in metric spaces.

Chapter 4

BoostMap: A Machine Learning Method For Embedding Construction

As mentioned in Section 2.2, an embedding $F : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$ is called *proximity-preserving* if it holds that $D(X_1, X_2) \leq D(X_1, X_3) \Leftrightarrow \Delta(F(X_1), F(X_2)) \leq \Delta(F(X_1), F(X_3))$ for all $X_1, X_2, X_3 \in \mathbb{X}$. We say that an embedding F *fails* on triple (X_1, X_2, X_3) if $D(X_1, X_2) < D(X_1, X_3)$ and $\Delta(F(X_1), F(X_2)) > \Delta(F(X_1), F(X_3))$, or $D(X_1, X_2) > D(X_1, X_3)$ and $\Delta(F(X_1), F(X_2)) < \Delta(F(X_1), F(X_3))$. In typical cases, as illustrated in our experiments in Chapter 7, embeddings of a non-Euclidean space $\mathbb{X}$ into a vector space $\mathbb{R}^d$ with an L_p distance measure are not proximity-preserving. In the BoostMap method that we describe in this chapter, the goal is to construct an embedding that is as close to being proximity-preserving as possible, i.e., an embedding that fails on as few triples as possible. We will show that, for the purpose of speeding up nearest neighbor retrieval, it is sufficient to limit our attention to triples (X_1, X_2, X_3) of a specific type, as discussed in Section 4.1.3.

The question then becomes how to construct an embedding according to the above criterion, so that it should fail on as few triples as possible. We will provide an answer to that question by making a key observation: deciding, for a triple (X_1, X_2, X_3) if X_1 is closer to X_2 or to X_3 is essentially a binary classification problem, since there are two main options: X_1 is either closer to X_2 or to X_3 (we assume that cases where X_1 is equally far from X_2 and X_3 are rare, and we ignore such cases). Any embedding F defines a binary classifier $\tilde{F}$ that decides if X_1 is closer to X_2 or to X_3 by simply checking if $F(X_1)$ is closer to $F(X_2)$ or to $F(X_3)$. Constructing an embedding F that fails on as few triples as possible is equivalent to constructing the associated classifier $\tilde{F}$ to be as accurate as possible.

The BoostMap algorithm constructs an embedding F by minimizing the error rate of the associated classifier $\tilde{F}$. Embedding F is a multidimensional embedding, and each of its dimensions is a 1D embedding defined using a reference object or a pair of pivot objects, as specified by Equations 2.7 and 2.10. We will show that the task of selecting 1D embeddings for each of the dimensions of F is equivalent to the task of designing an accurate binary classifier of triples (X_1, X_2, X_3) as a linear combination of weak classifiers corresponding to 1D embeddings. The latter task is a natural fit for boosting methods proposed in the machine learning literature, and we use for that task the AdaBoost algorithm [Schapire and Singer, 1999].

4.1 Associating Embeddings with Classifiers

In this section we formally define a classifier $\tilde{F}$ for every embedding F, and we show that any linear combination of such classifiers $\tilde{F}$ also corresponds to an embedding. In addition, we specify a set of triples such that filter-and-refine retrieval accuracy using F is only affected by misclassifications of triples belonging to that set.

4.1.1 From Embeddings to Classifiers

As in previous sections, $\mathbb{X}$ is a space of objects and D is a distance measure defined on $\mathbb{X}$. Now, let (X, A, B) be a triple of objects in $\mathbb{X}$. For that triple, one of the following three cases must be true:

- X is closer to A than to B.
- X is equally far from A and B.
- X is closer to B than to A.

In order to denote, for each triple (X, A, B), which of those three possibilities is true, we define the *proximity order* P of triple (X, A, B) to be a function that outputs 1 when X is closer to A than to B and -1 when X is closer to B than to A. In case of a tie, $P(X, A, B)$ is zero.

$$P(X, A, B) = \begin{cases} 1 & \text{if } D(X, A) < D(X, B) \; . \\ 0 & \text{if } D(X, A) = D(X, B) \; . \\ -1 & \text{if } D(X, A) > D(X, B) \; . \end{cases} \tag{4.1}$$

We consider the case $P(X, A, B) = 0$ (where X has exactly the same distance from A and B) to be a relatively rare, borderline case. In spaces where distances can take any value within some range of real numbers, it is typically unusual for an object to have the exact same distance to two database objects. Based on these considerations, we will largely ignore the case where $P(X, A, B) = 0$ from this point forward. The cases $P(X, A, B) = 1$ and $P(X, A, B) = -1$ are the only two cases that we worry about. Consequently, we consider the task of estimating $P(X, A, B)$ to be a *binary* classification task.

Let F be an embedding that maps (X, D) to $(\mathbb{R}^d, \Delta)$. If we know $F(X), F(A)$, and $F(B)$, we can guess whether X is closer to A or to B, by checking whether $F(X)$ is closer to $F(A)$ or to $F(B)$. More formally, for every embedding F we define a *proximity classifier* $\tilde{F}$, that estimates the proximity order $P(X, A, B)$ as follows:

$$\tilde{F}(X, A, B) = \Delta(F(X), F(B)) - \Delta(F(X), F(A)) \; . \tag{4.2}$$

If we define $\text{sign}(x)$ to be 1 for $x > 0$, 0 for $x = 0$, and -1 for $x < 0$, then $\text{sign}(\tilde{F}(X, A, B))$ is an estimate of $P(X, A, B)$.

Given a set of triples $\mathbb{T} \subseteq \mathbb{X}^3$, the error rate of $\tilde{F}$ on that set is essentially the fraction of triples misclassified by $\tilde{F}$. We will provide a formal definition $G(\tilde{F}, \mathbb{T})$ for the error rate that is just slightly more complicated, because it treats errors where $P(X, A, B) = 0$ or $\tilde{F}(X, A, B) = 0$ as only half-errors. First, as auxiliary notation, we define the classification error $G(\tilde{F}, X, A, B)$ of applying $\tilde{F}$ on a particular triple (X, A, B) as:

$$G(\tilde{F}, X, A, B) = \frac{|P(X, A, B) - \text{sign}(\tilde{F}(X, A, B))|}{2} \; . \tag{4.3}$$

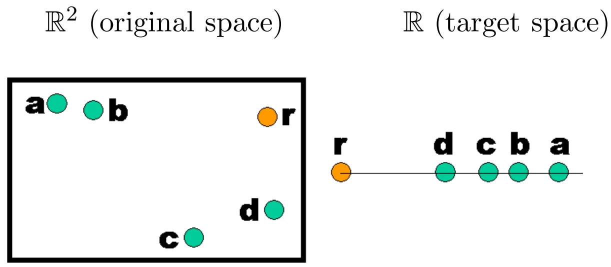

Figure 4·1: An embedding F^P of five 2D points (shown on the left) into the real line (shown on the right), using P as the reference object. The target of each 2D point on the line is labeled with the same letter as the 2D point. The classifier $\tilde{F}^P$ (Equation 4.2) classifies correctly 46 out 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 $F^P(B)$ is closer to $F^P(C)$ than it is to $F^P(A)$.

In the above definition, essentially $G = 0$ if the embedding successfully predicts the proximity order of the triple, and $G = 1$ when the embedding makes the wrong prediction. The only exceptions are cases where X is equally far from A and B or $F(X)$ is equally far from $F(A)$ and $F(B)$, in which case G has a value of 0 if classification is correct, or a value of 0.5 if classification is incorrect.

The overall classification error rate $G(\tilde{F}, \mathbb{T})$ is defined to be the expected value of $G(\tilde{F}, X, A, B)$, over all triples $(X, A, B) \in \mathbb{T}$:

$$G(\tilde{F}, \mathbb{T}) = \frac{\sum_{(X,A,B)\in\mathbb{T}} G(\tilde{F}, X, A, B)}{|\mathbb{T}|} , \tag{4.4}$$

where $|\mathbb{T}|$ denotes the number of triples in $\mathbb{T}$.

Figure 4·1 illustrates an example of a 1D embedding, and triples of objects that are misclassified by the classifier corresponding to that embedding.

4.1.2 From Classifiers to Embeddings

At this point we have established that every embedding $F : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$ corresponds to a binary classifier $\tilde{F}$ of triples of objects, i.e., a classifier that decides for any three objects $X, A, B \in \mathbb{X}$ if X is closer to A or to B. A natural question to ask is whether the converse also holds, i.e., whether it holds that for every binary classifier H of triples $(X, A, B) \in \mathbb{X}^3$ we can find F, d, and Δ such that $F : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$ and $H = \tilde{F}$. The answer to that question is negative, and here is a simple counterexample: suppose we have a classifier H of triples, giving the following answers:

$$H(X, A, B) = 1 \ .$$
$$H(X, B, C) = 1 \ .$$
$$H(X, A, C) = -1 \ .$$

Classifier H predicts that X is closer to A than to B, X is closer to B than to C, and X is closer to C than to A. If classifier H did correspond to an embedding F and a distance measure Δ, so that $H(X, A, B) = \Delta(F(X), F(B)) - \Delta(F(X), F(A))$, then we would arrive at the following contradiction:

$$\Delta(F(X), F(A)) < \Delta(F(X), F(B)) < \Delta(F(X), F(C)) < \Delta(F(X), F(A)) \Rightarrow$$
$$\Delta(F(X), F(A)) < \Delta(F(X), F(A)) \ .$$

Therefore, there can be no F and Δ such that $F : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$ and $H = \tilde{F}$.

At the same time, given a space $\mathbb{X}$ and distance measure D, there exists a family $\mathbb{H}$ of classifiers for which we can easily show that each classifier $H \in \mathbb{H}$ corresponds to an embedding F and a distance measure Δ. We define $\mathbb{H}$ to be the set of all classifiers H that can be written in the following form:

$$H(X, A, B) = \sum_{j=1}^{J} (\alpha_j \tilde{F}_j(X, A, B)) \ , \tag{4.5}$$

where J is any positive integer and each F_j is an embedding mapping $\mathbb{X}$ and D to some real vector space $\mathbb{R}^{d_j}$ and some distance measure Δ_j.

Proposition 1 *If classifier H is of the form of Equation 4.5, then we can construct an embedding F and distance measure Δ such that $F : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$ and $H = \tilde{F}$, for some integer d. Furthermore, if each F_j is a 1D embedding mapping $(\mathbb{X}, D)$ to $(\mathbb{R}, L_p)$, for any $p > 0$, then Δ is a weighted L_1 distance measure.*

Proof: Given that $H(X, A, B) = \sum_{j=1}^{J}(\alpha_j \tilde{F}_j(X, A, B))$, we define F and Δ as follows:

$$F(X) = (F_1(X), \ldots, F_J(X)) \ .$$
$$\Delta(F(X_1), F(X_2)) = \sum_{j=1}^{J}(\alpha_j \Delta_j(F_j(X_1), F_j(X_2))) \ .$$

Embedding F maps $\mathbb{X}$ into a d-dimensional vector space, where $d = \sum_{j=1}^{J} d_j$, and Δ is the sum of individual distances Δ_j that correspond to embeddings F_j.

Given these definitions, the proof that $H = \tilde{F}$ can be obtained in a few simple steps, by starting from the definition of $\tilde{F}$ in Equation 4.2:

$$\begin{aligned}
\tilde{F}(X, A, B) &= \Delta(F(X), F(B)) - \Delta(F(X), F(A)) \\
&= \sum_{j=1}^{J}(\alpha_j \Delta_j(F_j(X), F_j(B))) - \sum_{j=1}^{J}(\alpha_j \Delta_j(F_j(X), F_j(A))) \\
&= \sum_{j=1}^{J}(\alpha_j (\Delta_j(F_j(X), F_j(B)) - \Delta_j(F_j(X), F_j(A)))) \\
&= \sum_{j=1}^{J}(\alpha_j \tilde{F}_j(X, A, B)) = H(X, A, B) \ .
\end{aligned}$$

If each F_j is a 1D embedding mapping $(\mathbb{X}, D)$ to $(\mathbb{R}, L_p)$, for any $p > 0$, then it holds that $\Delta_j(F_j(X_1), F_j(X_2)) = |F_j(X_1) - F_j(X_2)|$. Therefore, $\Delta(F(X_1), F(X_2)) = \sum_{j=1}^{J}(\alpha_j |F_j(X_1) - F_j(X_2)|)$, which means that Δ is a weighted L_1 distance measure, with weights α_j.

$\square$

We have shown that if classifier H is a weighted linear combination of classifiers corresponding to embeddings, then H itself is equivalent to a specific embedding F and a specific distance measure Δ. By the word "equivalent" we mean that, for any (X, A, B) such that X is closer to A than to B, H misclassifies (X, A, B) if and only if F fails on that triple, i.e., if and only if F maps X closer to B than to A according to distance measure Δ.

The significance of this equivalence is that it allows us to map the problem of embedding optimization to the problem of optimizing a weighted linear combination of binary classifiers, which is exactly the problem that boosting methods are designed to solve.

4.1.3 Choosing a Set of Triples

In the BoostMap algorithm, embedding construction is formulated as the problem of constructing an embedding F in such a way that the classification error $G(\tilde{F}, \mathbb{T})$ of the embedding is minimized on a specific set $\mathbb{T}$ of triples. We will now take a look at the issue of how to choose this set $\mathbb{T}$ of triples.

We denote by $\mathbb{X}^3$ the set of all possible triples we can form using objects from $\mathbb{X}$. If $\tilde{F}$ classifies correctly all triples in $\mathbb{X}^3$, then F is proximity-preserving. In that case, if X is the k-nearest neighbor of Q in $\mathbb{X}$, $F(X)$ is the k-nearest neighbor of $F(Q)$ in $F(\mathbb{X})$, for any value of k. Overall, the classification error $G(\tilde{F}, \mathbb{X}^3)$ is a quantitative measure of how closely the approximate similarity rankings obtained in $F(\mathbb{X})$ will resemble the exact similarity rankings obtained in $\mathbb{X}$.

Suppose that we have a finite database $\mathbb{U} \subseteq \mathbb{X}$, and we know that in our application we are only interested in retrieving up to $k_{\max}$ nearest neighbors for each query object $Q \in \mathbb{X}$. An example of such an application is k-nearest neighbor classification, where for every test object we want to retrieve k database objects, so $k_{\max} = k$ in that case. We will denote the set of the $k_{\max}$ nearest neighbors of Q in database $\mathbb{U}$ as $NN(Q, \mathbb{U}, k_{\max})$. In order to achieve perfect retrieval accuracy of up to $k_{\max}$ nearest neighbors using an embedding F, it is sufficient that classifier $\tilde{F}$ be perfect on a restricted set of triples $\mathbb{T}_{k_{\max}}$ defined as follows:

$$\mathbb{T}_{k_{\max}} = \{(Q, A, B) | Q \in \mathbb{X}, A \in NN(Q, \mathbb{U}, k_{\max}), B \in \mathbb{U}\} \ . \tag{4.6}$$

If $\tilde{F}$ makes no mistakes on triples in $\mathbb{T}_{k_{\max}}$ then for any query object Q and any $k \in \{1, \dots, k_{\max}\}$, X is the k-nearest neighbor of Q in $\mathbb{U}$ iff $F(X)$ is the k-nearest neighbor of $F(Q)$ in $F(\mathbb{U})$. Therefore, using F for the filtering step of filter-and-refine retrieval, and setting parameter p of the refine step to 0, which is the setting that minimizes retrieval time, we can always retrieve successfully up to $k_{\max}$ nearest neighbors for any query object. If $\tilde{F}$ misclassifies triples that are not in $\mathbb{T}_{k_{\max}}$, retrieval accuracy is not affected. Based on the these considerations, assuming that we are given parameter $k_{\max}$, the goal of our formulation is to construct an embedding F_{out} in a way that minimizes $G(\tilde{F}_{\text{out}}, \mathbb{T}_{k_{\max}})$.

It is interesting to compare the measure of embedding quality we have proposed, i.e., the classification error on the set $\mathbb{T}_{k_{\max}}$, with the measures of stress and distortion described in Section 2.2. A key difference is that the measure proposed here is fundamentally a local measure: we only care that nearest neighbors remain nearest neighbors under the embedding. The vast majority of triples of objects (X_i, A_i, B_i) are such that neither A_i nor B_i is one of the nearest neighbors of X_i, and therefore we are not concerned about classifying such triples correctly. In contrast, stress and distortion are global measures, that are affected by every pair of objects, although the vast majority of pairs of objects (X_i, A_i) are such that X_i and A_i are not nearest neighbors of each other. Arguably a method that minimizes stress and distortion spends most of its effort on pairs of objects that have no bearing on how well the embedding preserves nearest neighbor structure. In contrast, the measure proposed here is tightly connected with the amount of nearest neighbor structure preserved by the embedding.

4.2 Reducing Embedding Construction to a Boosting Problem

As stated in Section 4.1.3, our goal is to construct an embedding $F_{\text{out}} : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$ in a way that minimizes the classification error of classifier $\tilde{F}_{\text{out}}$ on a specific set of triples. Using the correspondences we have established between embeddings and classifiers, we can reduce the problem of embedding optimization to an equivalent problem of classifier optimization, and in particular to the problem of optimizing a weighted linear combination of binary classifiers. This reduction is based on the following three observations:

- Given a large database $\mathbb{U}$, we can define a large pool of 1D embeddings, either by picking any database object as a reference object and applying Equation 2.7, or by picking any pair of database objects as pivot objects and applying Equation 2.10.

- Since every embedding corresponds to a classifier, having a large pool of 1D embeddings means we also have a large pool of classifiers. These classifiers are based on simple 1D embeddings and we do not expect these classifiers to be very accurate. At the same time, as long as the simple 1D embeddings defined by Eqs. 2.7 and 2.10 have the property of mapping nearby objects to nearby points on the real line, we expect such embeddings to preserve at least a small amount of the structure of the original space. In that case, the classifiers corresponding to such embeddings should behave as *weak classifiers* [Schapire and Singer, 1999], meaning that even if they have a high error rate, they should classify correctly more than half the triples, and thus perform better than a random guess. A simple example is shown in Figure 4·1, where the classifier corresponding to a 1D embedding correctly classifies 46 out of 60 possible triples of objects.

- If we start with weak classifiers corresponding to 1D embeddings, and we manage to linearly combine many such classifiers into an aggregate classifier that achieves a low error rate, then we can use Proposition 1 to convert the aggregate classifier into an embedding. Consequently, the problem that we are trying to solve, i.e., constructing an embedding whose corresponding classifier has a low error rate, can be reduced to the problem of finding a good linear combination of weak classifiers corresponding to 1D embeddings.

Naturally, the problem of linearly combining weak binary classifiers into a strong classifier with low error rate is exactly the problem that classical boosting methods like AdaBoost

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 training round $j = 1, \ldots, J$:

1. Train weak learner using training weights $w_{i,j}$, and obtain weak classifier $h_j : \mathcal{G} \rightarrow \mathbb{R}$, and a corresponding weight $\alpha_j \in \mathbb{R}$.

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

$$w_{i,j+1} = \frac{w_{i,j} \exp(-\alpha_j y_i h_j(o_i))}{z_j} \ . \tag{4.7}$$

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

Output the final classifier:

$$H(x) = \sum_{j=1}^{J} \alpha_j h_j(x). \tag{4.8}$$

Figure 4·2: The AdaBoost algorithm. This description is largely copied from [Schapire and Singer, 1999].

[Schapire and Singer, 1999] and LogitBoost [Friedman et al., 2000] have been designed to solve. Therefore, we can directly use one of these methods to construct an optimized high-dimensional embedding using 1D embeddings as building blocks. In our implementation we have chosen to use the AdaBoost algorithm.

4.3 The Embedding Construction Algorithm

The AdaBoost algorithm is shown in Figure 4·2. The inputs to AdaBoost are a set of objects o_i, together with their corresponding class labels y_i. Labels y_i are equal either to -1 or to 1. The objects o_i should not be confused with the objects of space $\mathbb{X}$. As we will see shortly, actually each o_i corresponds to a triple of objects of $\mathbb{X}$. AdaBoost assumes that we have a “weak learner” module, which we can call at each round to obtain a new weak classifier and a weight for that classifier. The goal is to construct a strong classifier that achieves much higher accuracy than the individual weak classifiers.

Overall, the AdaBoost algorithm performs a number J of training rounds. At each round j, AdaBoost picks a weak classifier h_j, and determines a weight α_j for that classifier, by calling the weak learner. Then, AdaBoost defines new training weights $w_{i,j+1}$ to be used in the next training round. The training weights are adjusted so that training objects that are misclassified by the chosen weak classifier h_j have a higher weight in the next round. The influence that each training object has on the output of the weak learner at round j is proportional to the weight of that object at round j.

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, and the reader can refer to [Tieu and Viola, 2000] and [Viola and Jones, 2001] for two examples.

The BoostMap algorithm is an adaptation of AdaBoost to the problem of embedding construction. The goal of BoostMap is to construct an embedding from an arbitrary space X to d-dimensional Euclidean space $\mathbb{R}^d$. In order to apply AdaBoost to our problem we need to perform some preprocessing before invoking AdaBoost, we need to specify how to implement the first step of the AdaBoost algorithm shown in Figure 4·2, we need to specify how to decide J, i.e., the number of training rounds to perform in AdaBoost, and finally we need to postprocess the output of AdaBoost and convert that output into an embedding. We now proceed to describe in detail how we perform each of these steps.

4.3.1 Inputs and Preprocessing

The inputs to the BoostMap algorithm are the following:

- A database $\mathbb{U}$ of objects in some space $\mathbb{X}$ with distance measure D.
- A positive integer $k_{\max}$ specifying the maximum number of nearest neighbors we will be interested in retrieving using the resulting embedding.

- A set $\mathbb{C} \subset \mathbb{U}$ of candidate reference and pivot objects. Elements of $\mathbb{C}$ will be used to define 1D embeddings.

- A set $\mathbb{T} \subset \mathbb{U}$ of training objects. Elements of $\mathbb{T}$ will be used to form training triples, i.e., the o_i's used by AdaBoost. We should note that we use the term "training objects" for the elements of $\mathbb{T}$, and not for the o_i's, which are typically called "training objects" in existing literature. For the o_i's we will use the term "training triples."

- A matrix of distances from each $X_1 \in \mathbb{C}$ to each $X_2 \in \mathbb{C}$.

- A matrix of distances from each $X_1 \in \mathbb{C}$ to each $X_2 \in \mathbb{T}$.

- A matrix of distances from each $X_1 \in \mathbb{T}$ to each $X_2 \in \mathbb{T}$.

In addition, we need to specify a few parameters that are useful for controlling the running time of the training algorithm. For reference, we provide here the list of all parameters needed by the training algorithm. The role that these parameters play will be fully explained in the description of the training algorithm.

- The size β of the set $\mathcal{G}$ of training triples.

- The number γ of weak classifiers to consider at each training round.

- The number δ of classifiers selected after a quick scan at each training round j. These selected classifiers are then evaluated more thoroughly in order to choose the best weak classifier h_j and weight α_j.

- A parameter $Z_{\max}$ that will be used for deciding when to stop the training algorithm.

The goal of the training algorithm is to construct an embedding F_{out} in a way that minimizes the classification error of the corresponding classifier $\tilde{F}_{\text{out}}$ on $\mathbb{T}_{k_{\max}}$, the set of triples defined in Equation 4.6. As a reminder, a triple (X, A, B) is in $\mathbb{T}_{k_{\max}}$ if X is any object of $\mathbb{X}$, A is one of the $k_{\max}$ nearest neighbors of X in the database $\mathbb{U}$, and B is any database object. Ideally the training set of triples should be obtained by sampling from the set $\mathbb{T}_{k_{\max}}$. In practice, at least with the datasets that we have experimented, it is more convenient to choose training triples (X, A, B) such that X is a database object, as opposed to choosing X from objects that are not in the database, for two reasons: first, we often do not have enough non-database objects to use for training, and second, we want to use the non-database objects that we have for performance evaluation of the system.

An additional consideration is that choosing training triples (X, A, B) without imposing some restrictions on X, A and B increases the memory requirements of the algorithm, because we need to pass as an input a matrix of distances from all objects in $\mathbb{C}$ to all objects used to form training triples. This is the reason we require that, for any training triple (X, A, B), X, A and B must be elements of a smaller set $\mathbb{T}$ that is a subset of database $\mathbb{U}$. Now, in order to simulate sampling from $\mathbb{T}_{k_{\max}}$, given set $\mathbb{T}$ we choose β training triples (X_i, A_i, B_i) as follows:

1. Choose X_i randomly from $\mathbb{T}$.
2. Set $k' = \frac{k_{\max}|\mathbb{T}|}{|\mathbb{U}|}$.
3. Choose A_i randomly from the k' nearest neighbors of X_i in $\mathbb{T} - \{X_i\}$.
4. Choose B_i randomly from $\mathbb{T} - \{X_i, A_i\}$.
5. Set training triple o_i to (X_i, A_i, B_i).
6. Set class label y_i of o_i to the proximity order $P(o_i)$ of the triple, where the proximity order is as defined in Equation 4.1. As a reminder, we always assume that X_i is *not* equally far from A_i and B_i, so that $y_i(o_i)$ equals either -1 or 1. If a triple (X_i, A_i, B_i) happens to be such that X_i is equally far from A and B, the algorithm simply discards that triple.

The formula we use for setting k' makes the training triple selection process simulate uniform sampling from $\mathbb{T}_{k_{\max}}$ as closely as possible, under the constraint that each X_i, A_i and B_i must be an element of $\mathbb{T}$.

We have now specified the inputs and parameters that must be provided to the training algorithm. Next, we will specify how to implement the training algorithm, i.e., how to implement Step 1 of the algorithm (Step 2 is fully specified in Figure 4·2), and how to decide when to stop training.

4.3.2 The Training Algorithm

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 and a weight α_j. In our implementation, the weak learner simply evaluates a large number of weak classifiers, and finds the best classifier and best weight for that classifier. For the purposes of the original BoostMap algorithm described in this chapter, each weak classifier is a classifier $\tilde{F}_i$ where F_i is a 1D embedding. In Chapter 5 we will describe an alternative family of weak classifiers that can also be used with the algorithm described here. In [Athitsos and Sclaroff, 2005] and [Alon et al., 2005b] we have desribed additional alternative families of weak classifiers that can be used within the context of this algorithm.

The number of classifiers to evaluate is specified by parameter γ of the algorithm. Of these γ classifiers, half are reference-object embeddings of the form shown in Equation 2.7 and half are line-projection embeddings of the form shown in Equation 2.10. We will define two alternative ways to evaluate a classifier h at training round j. The first way is the training error Λ:

$$\Lambda_j(h) = \sum_{i=1}^{t} w_{i,j} G(h, X_i, A_i, B_i) \ , \tag{4.9}$$

where $G(h, X_i, A_i, B_i)$ is the error of h on the i-th training triple, as defined in Equation 4.4. Note that this training error is weighted based on $w_{i,j}$, and therefore $\Lambda_j(h)$ will be different at each training round j.

A second way to evaluate a classifier h is suggested in [Schapire and Singer, 1999]. 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:

$$Z_j(h, \alpha) = \sum_{i=1}^{t} (w_{i,j} \exp(-\alpha y_i h(X_i, A_i, B_i))) \ . \tag{4.10}$$

Note that, in the above equation, y_i is in the class label of training triple (X_i, A_i, B_i). When y_i has a different sign than $h(X_i, A_i, B_i)$ then Z_j receives a contribution greater than or equal to $w_{i,j}$ for that training triple, and the contribution increases, in that case, with the absolute value of $h(X_i, A_i, B_i)$. Therefore, the more badly a triple is misclassified by h, the more it contributes to Z_j. The full details of the significance of Z_j can be found in [Schapire and Singer, 1999]. 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 computing the Z_j value attained using that optimal α are very common operations in our algorithm, so we will define specific notation for those operations:

$$A_{\min}(h, j, l) \quad = \quad \text{argmin}_{\alpha \in [l, \infty)} Z_j(h, \alpha) \ . \tag{4.11}$$

$$Z_{\min}(h, j, l) \quad = \quad \min_{\alpha \in [l, \infty)} Z_j(h, \alpha) \ . \tag{4.12}$$

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 5.3.2 we will use classifier weights to define a weighted L_1 distance measure Δ in $\mathbb{R}^d$, and non-negative weights ensure that Δ is a metric, so that we can apply any of the numerous indexing methods available for L_1 metrics to further speed up the filtering step of filter-and-refine retrieval.

Choosing the Next Weak Classifier and Weight

We denote with H_j the classifier assembled by AdaBoost after j training rounds, so that $H_j = \sum_{i=1}^{j} \alpha_i h_i$. We now describe how to obtain H_j given H_{j-1}. At a high level, at training round j, H_j is obtained from H_{j-1} by performing one of the following operations:

- Remove one of the already chosen weak classifiers. To accomplish this, we simply add to the strong classifier the negation of the weak classifier we want to remove.
- Modify the weight of an already chosen weak classifier. To accomplish this, we simply add to the strong classifier the weak classifier we want to modify, with a weight equal to the desired modification.
- 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-1} as a weighted linear combination of unique weak classifiers. We denote that representation as $H_{j-1} = \sum_{i=1}^{K_{j-1}} \alpha'_{i,j-1} h'_{i,j-1}$. By saying that each classifier $h'_{i,j-1}$ is unique we mean that if $g \neq i$ then $h'_{g,j-1} \neq h'_{i,j-1}$. K_{j-1} is simply the number of unique weak classifiers occurring in H_{j-1}. The reason we need subscripts i and $j-1$ to specify a unique classifier and its weight is that the weight of a classifier $h_{i,j-1}$ can be different at different training rounds, since it changes at every training round j where that classifier is selected as h_j.

Our exact implementation of Step 1 of the AdaBoost algorithm of Figure 4·2 is shown in Algorithm 1. In Step 8 of the algorithm, using a small δ reduces training time, because it lets us evaluate $A_{\min}$ only for δ classifiers. In general, evaluating the weighted training error Λ_j for a classifier h is faster (by a factor of ten to twenty 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 $\delta = \gamma$ and $\gamma = |C|$.

Note that, as specified in Step 11 of Algorithm 1, the algorithm terminates when, at a given round j, we get select a new weak classifier h_j for which $Z_j(h_j, \alpha_j) \geq Z_{\max}$, meaning that we have failed to find a weak classifier that would be more than marginally beneficial to add to the strong classifier.

4.3.3 Postprocessing: Defining an Embedding and a Distance Measure

The output of AdaBoost is a strong classifier H. Without loss of generality, we can write H as $H = \sum_{c=1}^{d} \alpha'_c \tilde{F}_c$, where each $\tilde{F}_c$ is associated with a unique 1D embedding F_c. Classifier H has been trained to estimate, for triples of objects (X, A, B), if X is closer to A or to B. However, our final goal is to construct not a classifier, but an embedding. To achieve that we use Proposition 1, to convert H into an embedding $F_{\text{out}} : \mathbb{X} \to \mathbb{R}^d$ and a distance measure Δ, as follows:

$$F_{\text{out}}(x) = (F_1(x), ..., F_d(x)) \ . \tag{4.13}$$

$$\Delta((u_1, ..., u_d), (v_1, ..., v_d)) = \sum_{c=1}^{d} (\alpha'_c |u_c - v_c|) \ . \tag{4.14}$$

Δ is a weighted Manhattan (L_1) distance measure. Δ is a metric, because the training algorithm ensured that all α_c's are non-negative. By ensuring that Δ is a metric, we can apply to the resulting embedding any additional indexing, clustering and visualization tools that are available for L_1 metric spaces.

4.3.4 Using Alternative Families of Weak Classifiers

Steps 5, 6, and 7 of Algorithm 1 are the only steps that depend on the family of weak classifiers that we want to use as building blocks for the strong classifier. In this chapter the weak classifiers are of the form $\tilde{F}$ where F is a 1D reference-object or line-projection embedding. In Chapter 5, we will propose a different family of weak classifiers, that can be used to improve embedding accuracy. We have described additional families of weak classifiers

1. Let $z = \min_{c=1,\ldots,K_{j-1}} Z_j(h'_{c,j-1}, -\alpha'_{c,j-1})$.
2. If $z < 1$:
   - Set $g = \mathrm{argmin}_{c=1,\ldots,K_{j-1}} Z_j(h'_{c,j-1}, -\alpha'_{c,j-1})$.
   - Set $h_j = h'_{g,j-1}, \alpha_j = -\alpha'_{g,j-1}$.
   - Go to Step 12.

   Comment: If $z < 1$, we effectively remove $h'_{g,j-1}$ from the strong classifier.
3. Let $z = \min_{c=1,\ldots,K_{j-1}} Z_{\min}(h'_{c,j-1}, j, -\alpha'_{c,j-1})$.
4. If $z < Z_{\max}$:
   - Set $g = \mathrm{argmin}_{c=1,\ldots,K_{j-1}} Z_{\min}(h'_{c,j-1}, j, -\alpha'_{c,j-1})$.
   - Set $h_j = h'_{g,j-1}$.
   - Set $\alpha_j = A_{\min}(h'_g, j, -\alpha'_{g,j-1})$.
   - Go to Step 12.

   Comments: Here we modify the weight of $h'_{g,j-1}$, by adding α_j to it. The third arguments used when calling $Z_{\min}$ and $A_{\min}$ ensure that $\alpha_j \geq -\alpha'_{g,j-1}$, so that $\alpha_j + \alpha'_{g,j-1}$ is guaranteed to be non-negative. Note that $\alpha_j + \alpha'_{g,j-1}$ will be the new weight $\alpha'_{g,j}$ of h'_g. Also, note that we check if $z < Z_{\max}$. In principle, if $z < 1$ then this weight modification is beneficial. By using $Z_{\max}$ as a threshold we avoid minor weight modifications with insignificant numerical impact on the accuracy of the strong classifier.
5. Choose randomly $\gamma/2$ reference objects $P_1, \ldots, P_{\gamma/2}$ from the set $\mathbb{C}$ of candidate objects. Construct a set $\mathbb{F}_{j1} = \{F^{P_i} | i = 1, \ldots, \gamma/2\}$ of 1D embeddings using those reference objects .
6. Choose randomly a set $\mathbb{C}_j = \{(X_{1,1}, X_{1,2}), ..., (X_{\gamma/2,1}, X_{\gamma/2,2})\}$ of pairs of elements of $\mathbb{C}$, and construct a set of embeddings $\mathbb{F}_{j2} = \{F^{x_1,x_2} \mid (x_1, x_2) \in \mathbb{C}_j\}$, where F^{X_1,X_2} is as defined in Equation 2.10.
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 δ classifiers in $\tilde{\mathbb{F}}_J$ with the smallest $\Lambda_j(h)$.
9. Set $h_j = \mathrm{argmin}_{h \in \mathbb{H}_j} Z_{\min}(h, j, 0)$.
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. If $Z_j(h_j, \alpha_j) \geq Z_{\max}$ then terminate the training algorithm, and output H_{j-1} as the output strong classifier.
12. Set $z_j = Z_j(h_j, \alpha_j)$.
13. Set training weights $w_{i,j+1}$ for the next round using Equation 4.7.
14. Set $j = j + 1$. Go to Step 1.

Algorithm 1: The main loop of the BoostMap training algorithm. Steps 1-10 implement Step 1 of the AdaBoost algorithm shown in Figure 4·2.

that can be used with the BoostMap training algorithm in [Athitsos and Sclaroff, 2005] and [Alon et al., 2005b]. In order to make the training algorithm work with those different families of weak classifiers, the only change we need to make to the training algorithm is modify Steps 5, 6, and 7. The modifications simply need to ensure that, after Step 7, set $\mathbb{F}_j$, which is the pool of classifiers from which the next weak classifier will be chosen, contains only classifiers from the alternative family of weak classifiers that we want to use.

4.4 Properties of BoostMap Embeddings

In this section we take a closer look at some properties of query-sensitive embeddings and the proposed algorithm for constructing such embeddings.

4.4.1 Contractiveness

Contractiveness is an important property of some types of embeddings. When it holds, contractiveness can be used to guarantee that filter-and-refine retrieval will always return the true k-nearest neighbors, for any query [Hjaltason and Samet, 2003a]. An embedding $F : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$ is contractive if for any $X_1, X_2 \in \mathbb{X}$ it holds that $\Delta(F(X_1), F(X_2)) \leq D(X_1, X_2)$. As explained in [Hjaltason and Samet, 2003a], when an embedding is contractive, then the refine step of filter-and-refine retrieval can automatically decide how many exact distance computations it needs to perform, given a query, in order to guarantee correct results.

The output embedding $F_{\text{out}} : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$ of the BoostMap algorithm, constructed as described in Section 4.3.3, can be made contractive by dividing $\Delta(F_{\text{out}}(X_1), F_{\text{out}}(X_2))$ with a normalization term, provided that D is metric. If F_{out} contains no line projection embeddings, the normalization term W that should be used is:

$$W = \sum_{i=1}^{d} \alpha'_i \ , \tag{4.15}$$

where α'_i are the weights used in Equation 4.14 to define Δ.

Proposition 2 *Let* $X_1, X_2 \in \mathbb{X}$. *Suppose that* $\alpha'_i \geq 0$ *for all* i, *and suppose that* D *is metric. If all dimensions of* F_{out} *are reference-object embeddings, then the following property holds:*

$$\frac{1}{W}\Delta(F_{\text{out}}(X_1), F_{\text{out}}(X_2)) \leq D(X_1, X_2) \ . \tag{4.16}$$

Proof: If each dimension of F_{out} is a reference-object embedding, then F_{out} can be represented as $F_{\text{out}} = (F^{P_1}, \ldots, F^{P_d})$, where d is the dimensionality of F_{out} and P_i are reference objects. We will denote $F_{\text{out}}(X_1)$ as $(x_{1,1}, \ldots, x_{1,d})$ and $F_{\text{out}}(X_2)$ as $(x_{2,1}, \ldots, x_{2,d})$. First, based on the triangle inequality, we can easily see that:

$$|x_{1,i} - x_{2,i}| = |D(X_1, P_i) - D(X_2, P_i)| \leq D(X_1, X_2) \ . \tag{4.17}$$

Using this observation, we can complete the proof:

$$\begin{aligned}
\frac{1}{W}\Delta(F(X_1), F(X_2)) &= \frac{1}{W}\sum_{i=1}^{d}(\alpha'_i|x_{1,i} - x_{2,i}|) \\
&\leq \frac{1}{W}\sum_{i=1}^{d}(\alpha'_i D(X_1, X_2)) \\
&= \frac{1}{W}D(X_1, X_2)\sum_{i=1}^{d}\alpha_i \\
&= \frac{1}{W}D(X_1, X_2)W \\
&= D(X_1, X_2) \ .
\end{aligned}$$

□

If F_i, the i-th dimension of F_{out}, is a line-projection embedding, then it is shown in [Hjaltason and Samet, 2003a] that $|F_i(X_1) - F_i(X_2) \leq 3D(X_1, X_2)$. Therefore, if we divide $\Delta(X_1, X_2)$ by $3W$, then F_{out} is contractive even in the case where some of its dimensions are line-projection embeddings.

We should point out that the distance measures D used in the experiments do not obey the triangle inequality, and thus are non-metric. Consequently, contractiveness does not hold in the resulting embeddings, and the system can fail to retrieve the true nearest neighbor(s). No existing domain-independent embedding method [Faloutsos and Lin, 1995, Hristescu and Farach-Colton, 1999, Wang et al., 2000] is contractive in non-metric spaces. Since perfect accuracy is not guaranteed, the goal of embedding optimization in non-metric spaces is to provide as good trade-offs as possible between accuracy and efficiency. This is exactly what our algorithm achieves, by maximizing the amount of nearest neighbor structure preserved by the embedding. We will see in the experiments that, in all the datasets we have tried, embeddings produced using this algorithm achieved better accuracy-efficiency trade-offs than any alternative embedding method we have implemented.

4.4.2 Complexity

At each training round we evaluate a number of weak classifiers by measuring their performance on β training triples, in order to choose the best weak classifier. If γ weak classifiers are evaluated at each round, the computational time per training round is $O(\gamma\delta)$. In contrast, FastMap [Faloutsos and Lin, 1995], SparseMap [Hristescu and Farach-Colton, 1999], and MetricMap [Wang et al., 2000] do not require training at all.

Before we even start the training algorithm, we need to compute three distance matrices to pass as input to BoostMap: distances between objects in $\mathbb{C}$, distances between objects in $\mathbb{T}$, and distances from objects in $\mathbb{C}$ to objects in $\mathbb{T}$. Computing all those distances can sometimes be the most computationally expensive part of the algorithm, depending on the computational complexity of the distance measure D.

If time and memory resources are not limited, then we can set both $\mathbb{C}$ and $\mathbb{T}$ equal to the entire database. Otherwise, we need to create $\mathbb{C}$ and $\mathbb{T}$ by sampling randomly from the database. If (as in our experiments) $\mathbb{C}$ and $\mathbb{T}$ have an equal number of elements, then the number of distances that we need to precompute is quadratic to $|\mathbb{C}|$.

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

With respect to the *online* filter-and-refine retrieval cost, computing the d-dimensional embedding of a query object takes $O(d)$ time and requires between d and $2d$ evaluations of D. The number of distance evaluations depends on the number of line-projection 1D embeddings that are used in the d-dimensional embedding: for each such 1D embedding we need to compute two distances, whereas for each reference-object 1D embedding we only need to compute one distance. Comparing the embedding of the query to the embeddings of n database objects takes time $O(dn)$. For a fixed d, these costs are within a factor of 2 of the corresponding costs using FastMap [Faloutsos and Lin, 1995], SparseMap [Hristescu and Farach-Colton, 1999], and MetricMap [Wang et al., 2000]. To be exact, the number of distance evaluations required by these methods to embed a query is: $2d$ for FastMap, d for SparseMap, and $d + 1$ for MetricMap.

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

In cases (not encountered in our experiments) when the filter step takes up a significant part of retrieval time, one can apply vector indexing techniques [Böhm et al., 2001, Indyk, 2000, White and Jain, 1996] to speed up filtering. We should keep in mind that in the filter step we are finding nearest neighbors in a real vector space, and many indexing methods are applicable in such a setting. One of the advantages of using embeddings is exactly the fact that we map arbitrary spaces to well-understood real vector spaces, for which many tools are available.

4.4.3 Dynamic Datasets

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

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

4.5 Summary of the BoostMap method

In this chapter we have described the BoostMap method for embedding construction. At the core of this method is a formulation that associates embeddings with classifiers. We demonstrate that a direct measure of the amount of nearest neighbor structure preserved by an embedding is equivalent to the classification error rate of the classifier corresponding to that embedding, when that the error rate is measured on a specific set of triples of objects. Associating embedding quality with classification error on a binary classification task allows us to use machine learning methods, namely AdaBoost, for embedding optimization.

While this chapter has described a specific version of the BoostMap method, BoostMap is a very flexible method that can be applied in different ways to produce different types of embeddings. In Chapter 5 we will show how to modify BoostMap so as to produce query-sensitive embeddings, and in Chapter 6 we propose an alternative optimization criterion that is more relevant when our goal is nearest neighbor classification.

Overall, BoostMap is the core contribution of this thesis. The methods proposed in the subsequent chapters demonstrate how to extend BoostMap, or build on top of BoostMap, in ways that achieve even better trade-offs between accuracy and efficiency for the tasks of nearest neighbor retrieval and classification.

Chapter 5

Query-Sensitive Embeddings

In this chapter we present a new type of embeddings: embeddings that use a "query-sensitive" distance measure for the target space of the embedding. This distance measure is used to compare the vectors that the query and database objects are mapped to. The term "query-sensitive" means that the distance measure changes depending on the current query object. More specifically, the query-sensitive distance measure is a weighted L_1 distance measure where the weights automatically adjust to each query.

Existing embedding methods [Athitsos et al., 2004, Faloutsos and Lin, 1995, Hristescu and Farach-Colt Wang et al., 2000] compare vectors using global, query-insensitive distance measures. A query-sensitive distance measure improves embedding quality, by providing a natural way to identify, for each query object, the embedding dimensions that are the most useful for retrieving the nearest neighbors of that query. Identifying the most informative dimensions is an important issue that arises when objects are represented as high-dimensional vectors [Aggarwal, 2001] and addressing this issue can offer significant help in defining meaningful high-dimensional distance measures. As is demonstrated in our experimental results, using a well-chosen set of dimensions for each query is more useful than simply using all dimensions for all queries and assigning a fixed weight to each dimension.

First, we describe the motivation behind using a query-sensitive distance measure for the purposes of nearest neighbor retrieval. Next we illustrate theoretically the additional modeling power that can be obtained from query-sensitivity. Finally we show how to adapt the BoostMap algorithm to the purpose of constructing a query-sensitive distance measure, so as to increase the amount of nearest neighbor structure preserved by the resulting embedding.

5.1 Some Additional Related Work

The goal of query-sensitive embeddings is the same as the goal of the original BoostMap algorithm: we want to achieve efficient nearest neighbor retrieval in spaces with computationally expensive distance measures. Consequently, all the literature on efficient nearest neighbor retrieval surveyed in Chapter 3 is also relevant in the context of query-sensitive embeddings. Here we discuss some additional related work, and in particular we take a look at other existing methods that use non-global distance measures.

In the existing literature, non-global distance measures have been used for the purposes of optimizing the accuracy of nearest neighbor classification. In [Paredes and Vidal, 2000] the distance measure depends not on the query, but on the class of the database object that the query is compared to. Query-sensitive distance measures have been used in [Domeniconi et al., 2002, Hastie and Tibshirani, 1996]. Compared to the query-sensitive methods in [Domeniconi et al., 2002, Hastie and Tibshirani, 1996], the method described in this chapter has two important differences:

- We try to solve a different problem, namely efficient retrieval as opposed to accurate classification.

- In [Domeniconi et al., 2002, Hastie and Tibshirani, 1996], it is assumed that an initial global (query-insensitive) distance measure is available. In contrast, no initial distance measure is given as input to our algorithm.

In particular, in [Domeniconi et al., 2002, Hastie and Tibshirani, 1996], the query-sensitive distance measure is constructed online, while processing the query object, by iteratively refining the initial distance measure based on the objects in the neighborhood of the query. Because of this online iterative refinement, these methods incur additional computational costs compared to alternative query-insensitive methods. In contrast, in our method the query-sensitive distance measure is constructed offline, using training data. Compared to the query-insensitive version of the BoostMap described in the previous chapter, the online computational cost of processing a query using a query-sensitive embedding is virtually the same: some additional terms need to be computed, but those computations take up only a negligible fraction (typically much smaller than 0.001) of the total query processing time.

5.2 Motivation for Query-Sensitive Distance Measures

There exist several methods for constructing high-dimensional embeddings, e.g., FastMap [Faloutsos and Lin, 1995], the BoostMap algorithm [Athitsos et al., 2004], or simply using Equation 2.9 to define an embedding based on distances to multiple reference objects. However, existing methods have limited themselves to the task of assigning a vector of coordinates to each object. These vectors are often compared using the standard Euclidean distance without weights [Faloutsos and Lin, 1995, Hristescu and Farach-Colton, 1999]. BoostMap uses an L_1 metric and assigns weights to each dimension, but its representational power is still equivalent to that of a method using an unweighted distance: any weighted L_1 metric can be isometrically converted to an unweighted L_1 metric, by simple scaling of each dimension.

Our main motivation in designing query-sensitive embeddings has been to enhance the modeling power of embeddings, so that they can encapsulate additional information about the structure of the original space. Allowing the distance measure to be query-sensitive, i.e., allowing the weight assigned to each dimension to depend on the query, provides modeling power that is impossible to achieve with an embedding that utilizes a query-insensitive distance measure. For example, a well-known limitation of Euclidean embeddings (i.e., embeddings that use an L_2 distance measure in the target space) is that there exist finite metric spaces that cannot be isometrically embedded; furthermore, there exist worst cases where no embedding can achieve better than $O(\log |\mathbb{X}|)$ distortion, where $|\mathbb{X}|$ is the number of objects in the finite metric space $\mathbb{X}$ [Hjaltason and Samet, 2003a]. Query-sensitive embeddings trivially overcome that limitation, not only for metric spaces, but also for a large family of non-metric spaces:

Proposition 3 *For any finite space $\mathbb{X}$ with a reflexive distance measure D there exists a query-sensitive isometric embedding to $\mathbb{R}^{|\mathbb{X}|}$, i.e., the real vector space of dimension $|\mathbb{X}|$.*

Proof: We will construct such an embedding F and the associated query-sensitive distance measure Δ. By definition, F is isometric if the following holds: for any $X_i, X_j \in \mathbb{X}$, $D(X_i, X_j) = \Delta(F(X_i), F(X_j))$. Since $\mathbb{X}$ is finite, it can be represented as $X = \{X_1, \ldots, X_{|\mathbb{X}|}\}$. Without loss of generality we can assume that $X_i = X_j$ if and only if $i = j$. We define query-sensitive embedding F as follows:

$$F(X) = (D(X_1, X), D(X_2, X), \ldots, D(X_{|\mathbb{X}|}, X)) \ . \tag{5.1}$$

Now, we define an auxiliary function $S(y) : \mathbb{R} \rightarrow \{0, 1\}$:

$$S(y) = \begin{cases} 1 & \text{if } y = 0 \ . \\ 0 & \text{otherwise} \ . \end{cases} \tag{5.2}$$

Finally, if $u, v \in \mathbb{R}^{|\mathbb{X}|}$, $u = (u_1, \ldots, u_{|\mathbb{X}|})$ and $v = (v_1, \ldots, v_{|\mathbb{X}|})$, we define distance measure $\Delta(u, v)$:

$$\Delta(u, v) = \sum_{m=1}^{|\mathbb{X}|} (S(u_m)|u_m - v_m|) \ . \tag{5.3}$$

We have assumed that distance measure D is reflexive. By definition of reflexivity, $D(X_i, X_m) = 0$ iff $X_i = X_m$, i.e., iff $i = m$. Therefore, the m-th coordinate of $F(X_i)$, which is equal to $D(X_i, X_m)$, is zero iff $i = m$. Using that, we can verify that $\Delta(F(X_i), F(X_j)) = D(X_i, X_j)$, as follows:

$$\begin{aligned} \Delta(F(X_i), F(X_j)) &= \sum_{m=1}^{|\mathbb{X}|} (S(D(X_m, X_i))|D(X_m, X_i) - D(X_m, X_j)|) \\ &= S(D(X_i, X_i))|D(X_i, X_i) - D(X_i, X_j)| \\ &= |0 - D(X_i, X_j)| = D(X_i, X_j) \ . \end{aligned}$$

□

Distance measure Δ is query-sensitive: if Q is the query and X is a database object, the weight assigned to the m-th dimension while calculating $\Delta(Q, X)$ depends on whether the embedding of Q has a zero or non-zero value at that dimension. We should note that the only distance measure property used in this construction is reflexivity. Symmetry and the triangle inequality were not used. Consequently, finite non-metric spaces can also be isometrically embedded using the above construction, as long as the underlying distance measure is reflexive.

This isometric construction is not really useful, as is, in the context of nearest neighbor retrieval, where typically we cannot assume that the query object is identical to a database object, or to a reference object used in the embedding. Furthermore, the proof entails a certain amount of "cheating," since our construction essentially boils down to defining, for each query X_i, a custom-made embedding that uses X_i as a reference object. However, the mere fact that query-sensitive embeddings allow this kind of "cheating" illustrates the additional modeling power we gain by query-sensitivity: no such trick can be used with a query-insensitive L_p metric, except when $p = \infty$.

As a matter of fact, using a pretty similar proof one can show that every finite metric space $\mathbb{X}$ can be isometrically embedded to an L_∞ space of dimensionality $|\mathbb{X}|$. However, the proof that we have provided for query-sensitive embeddings is also applicable to finite non-metric spaces whose distance measures obey reflexivity, such as the chamfer distance [Barrow et al., 1977], Dynamic Time Warping [Kruskall and Liberman, 1983], and shape context matching [Belongie et al., 2002]. Also, it is important to note that the family of query-sensitive L_p measures is much richer than the family of L_∞ measures, because in a query-sensitive measure there is a large number of degrees of freedom to specify (and optimize), i.e., the weight of each dimension for each query object. In contrast, given a vector space, the L_∞ metric is fully specified, there are no degrees of freedom available to specify. Overall, query-sensitive L_p distance measures offer us a large number of degrees of freedom to optimize, much larger than the degrees of freedom in query-insensitive L_p measures with the same value of p, while at the same time they have the power to provide isometries of arbitrary metric spaces and a large family of non-metric spaces.

In more practical terms, a query-sensitive distance measure can capture the fact that, given a query object, using only a well-chosen subset of dimensions may be more useful

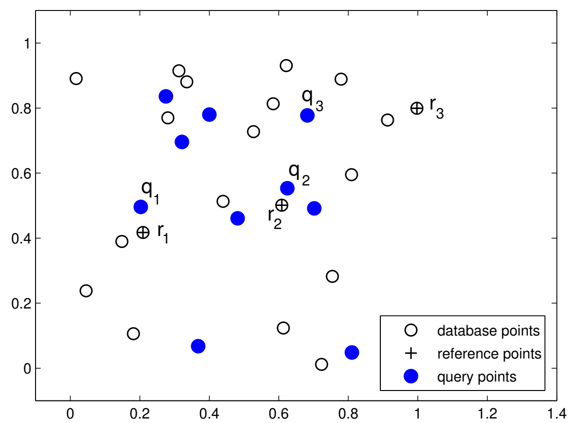

Figure 5·1: A toy example illustrating the use of query-sensitive embeddings. Our space is the set of points in the unit square $[0, 1] \times [0, 1]$. There are twenty database objects, three of which (indicated as r_1, r_2, r_3) are selected as reference objects. There are ten query objects, three of which are marked as q_1, q_2, q_3. As explained in the text, assigning query-sensitive weights to each embedding dimension would improve the accuracy of the embedding for queries q_1, q_2, q_3.

than indiscriminately using all dimensions. Figure 5·1 illustrates a quantitative example. In that toy example, we define a three-dimensional embedding F of the unit square $[0, 1] \times [0, 1]$. There are twenty database objects, three of which (indicated as r_1, r_2, r_3) are selected as reference objects. Using these reference objects, we define embedding $F(x) = (F^{r_1}(x), F^{r_2}(x), F^{r_3}(x))$, and we use the L_1 distance to compare the embeddings of two objects. There are ten query objects, three of which are marked as q_1, q_2, q_3. F fails on 23.5% of the 3800 triples (q, a, b) we can form by picking q from the query objects, and the pair a, b from the database objects. In contrast, the 1D embeddings $F^{r_1}, F^{r_2}, F^{r_3}$ fail respectively on $39.2\%, 36.4\%$, and 26.6% of the triples. However, if we restrict our attention to triples (q, a, b) where $q = q_1$, F^{r_1} does better than F: F^{r_1} fails on 5.8% of those triples, whereas F fails on 11.6% of those triples. Similarly, for $q = q_2$ and $q = q_3$ respectively, F^{r_2} and F^{r_3} are more accurate than F. Therefore, for query objects q_1, q_2, q_3, it would be beneficial to use a query-sensitive weighted L_1 measure, that would respectively use only the first, second, and third dimension of F.

In general, as pointed out in [Aggarwal, 2001], measuring distances between objects that are represented as high-dimensional vectors raises the following issues:

- **Lack of contrasting:** Two high-dimensional objects are unlikely to be very similar in all the dimensions.

- **Statistical sensitivity:** The data is rarely uniformly distributed, and for a pair of objects there may be only relatively few coordinates that are statistically significant for comparing those objects.

The example of Figure 5·1 that we have discussed can be seen as an illustration of the problem of statistical sensitivity, and how that problem can be addressed by utilizing query-sensitivity. Overall, query-sensitive distance measures provide a principled way to address the problems described in [Aggarwal, 2001], by putting more emphasis on dimensions that are more important for a particular query.

5.3 Constructing a Query-Sensitive Embedding

In this section we describe a method for constructing query-sensitive embeddings using a modification of the BoostMap algorithm described in Chapter 4.

5.3.1 Defining Query-Sensitive Classifiers from 1D Embeddings

As described earlier, every embedding F corresponds to a classifier that classifies triples (X, A, B) of objects in $\mathbb{X}$. We recall from Section 4.1.1 that, given embedding F, and a distance measure Δ for comparing vectors, we can define the classifier $\tilde{F}$ associated with embedding F as follows:

$$\tilde{F}(X, A, B) = \Delta(F(X), F(B)) - \Delta(F(X), F(A)) \ .$$

Sometimes, $\tilde{F}$ may do a really good job on triples (X, A, B) when X is in a specific region, but at the same time it may be beneficial to ignore $\tilde{F}$ when X is outside that region.

For example, suppose that we have an embedding F^P defined using reference object P. If $X = P$, then $\tilde{F}^P$ will classify correctly all triples (X, A, B), where A and B are any two objects of space $\mathbb{X}$. If $X \neq P$, we still expect that, the closer X is to P, the more accurate $\tilde{F}^P$ will be on triples (X, A, B). Figure 5·1 illustrates such cases.

In Chapter 4, and in the original description of BoostMap in [Athitsos et al., 2004], the weak classifiers that are used by AdaBoost are of type $\tilde{F}$, with F being a 1D embedding. In this chapter we propose to use a different type of classifier, that can explicitly model the fact that an 1D embedding F can be more useful in some regions of the space and less useful in other regions.

In particular, given a 1D embedding F, we need a function $S(Q)$ (which we call a *splitter*), that will estimate, given a query Q, whether classifier $\tilde{F}$ is useful or not. More formally, if $\mathbb{X}$ is the original space, we use the term *splitter* to denote any function mapping $\mathbb{X}$ to the binary set $\{0, 1\}$. We can readily define splitters using 1D embeddings. Given a 1D embedding $F : \mathbb{X} \rightarrow \mathbb{R}$, and a subset $V \subset \mathbb{R}$, we can define a splitter $S_{F,V} : \mathbb{X} \rightarrow \{0, 1\}$ as follows:

$$S_{F,V}(Q) = \begin{cases} 1 & \text{if } F(Q) \in V \ . \\ 0 & \text{otherwise} \ . \end{cases} \tag{5.4}$$

Now, suppose we have a subset $V \subset \mathbb{R}$ and a 1D embedding $F : \mathbb{X} \rightarrow \mathbb{R}$. We define a *query-sensitive classifier* $\Gamma_{F,V} : \mathbb{X}^3 \rightarrow \mathbb{R}$, as follows:

$$\Gamma_{F,V}(Q, A, B) = S_{F,V}(Q)\tilde{F}(Q, A, B) \ . \tag{5.5}$$

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

In our implementation, the sets V that we use for defining query-sensitive classifiers $\Gamma_{F,V}$ art of two different types:

Type 1: $V = (\tau_1, \tau_2)$, where τ_1, τ_2 can be any real numbers, or $\pm\infty$. In this case, $S_{F,V}$ accepts Q if $F(Q) > \tau_1$ and $F(Q) < \tau_2$.

Type 2: $V = \mathbb{R} - (\tau_1, \tau_2)$, where τ_1, τ_2 can be any real numbers, or $\pm\infty$. In this case, $S_{F,V}$ accepts Q if $F(Q) < \tau_1$ or $F(Q) > \tau_2$.

We should emphasize that different types of V can also be used in our algorithm, and different types of splitters are also possible. For example, instead of having a binary splitter that always outputs 0 or 1, we can alternatively define a "soft" splitter, that outputs a value between 0 and 1. However, in our implementation we have only experimented with binary splitters and the five types of V listed above.

5.3.2 Adapting the BoostMap Training Algorithm

Selecting Weak Classifiers

The key adaptation of the BoostMap algorithm from the previous chapter, that will allow the algorithm to produce query-sensitive embeddings, is that now the weak classifiers considered by AdaBoost are classifiers not of the form $\tilde{F}$, but of the form $\Gamma_{F,V}$ as defined in Eq. 5.5, where F is some 1D embedding defined using reference objects or pivot objects from the set $\mathbb{C}$ of candidate objects. In particular, at training round j we choose, randomly, a large number of 1D embeddings, as in the training algorithm described in Chapter 4. Then, for each selected 1D embedding F, we find the range $V_{F,j}$ that achieves the lowest training error at round j. Algorithm 2 describes the procedure for finding the optimal range $V_{F,j}$ of type 1. The corresponding algorithm for finding the optimal range for type 2 is almost identical, except that the weighted training error corresponding to a range $V_{F,j}$ of type 1 is equal to 1 minus the weighted training error corresponding to range $\mathbb{R} - V_{F,j}$.

The weak classifier selected at training round j is chosen among the classifiers $\Gamma_{F,V_{F,j}}$. In particular, to adapt the BoostMap training algorithm (Algorithm 1 from Section 4.3.2) so that it produces query-sensitive embeddings, we simply need to modify Step 7 of that algorithm as follows:

7. Define $\mathbb{F}_j = \mathbb{F}_{j1} \cup \mathbb{F}_{j2}$. Set $\tilde{\mathbb{F}}_J = \{\Gamma_{F,V_{F,j}} \mid F \in \mathbb{F}_j\}$.

With this modification, the training algorithm produces a strong classifier that can be readily converted into an embedding and a query-sensitive distance measure. We now proceed to discuss how make that conversion.

Training Output: Embedding and Distance

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

$$H = \sum_{j=1}^{J} \alpha_j \Gamma_{F'_j,V_j} \ . \tag{5.6}$$

Each $\Gamma_{F'_j,V_j}$ is associated with a 1D embedding F'_j. As in the original BoostMap algorithm from the previous chapter, we now need to convert classifier H into an embedding and a distance measure.

First, we should note that a particular 1D embedding can be equal to multiple F'_j's occurring in the definition of classifier H. For example, in different training rounds the algorithm may choose choose query-sensitive weak classifiers based on the same 1D embedding, but using a different splitter. We construct the set $\mathbb{F}$ of all unique 1D embeddings used in H, as $\mathbb{F} = \bigcup_{j=1}^{J}\{F'_j\}$, and we denote the elements of $\mathbb{F}$ as $F_1, \ldots, F_d$. Using this notation, the output embedding $F_{\text{out}} : X \to \mathbb{R}^d$ is defined simply as $F_{\text{out}}(X) = (F_1(X), \ldots, F_d(X))$. Obviously, it is a d-dimensional embedding.

Before defining distance measure Δ, we first need to define an auxiliary function $A_i(Q)$, which assigns a weight to the i-th dimension, for $i = 1, \ldots, d$, given a query object Q:

$$A_i(Q) = \sum_{j:((j\in\{1,\ldots,J\})\wedge(F_i=F'_j)\wedge(F_i(Q)\in V_j))} \alpha_j \ . \tag{5.7}$$

In words, given query Q, for dimension i, we go through all weak classifiers $\Gamma_{F'_j,V_j}$ that make up H. For each such classifier, we check if the splitter $S_{F'_j,V_j}$ accepts Q (i.e., we check

input : F: 1D embedding.
$(X_1, A_1, B_1), \ldots, (X_\beta, A_\beta, B_\beta)$: the training triples used in the BoostMap algorithm.
$y_1, \ldots, y_\beta$: the class labels of the training triples.
$w_{1,j}, \ldots, w_{\beta,j}$: the weights of the training triples at training round j.
output : A set $V_{F,j}$.

$((X'_1, A'_1, B'_1), \ldots, (X'_\beta, A'_\beta, B'_\beta)) =$ sorted $((X_1, A_1, B_1), \ldots, (X_\beta, A_\beta, B_\beta))$ in ascending order of $F(X_i)$.
$w'_1, \ldots, w'_\beta =$ training weights of $((X'_1, A'_1, B'_1), \ldots, (X'_\beta, A'_\beta, B'_\beta))$.
$y'_1, \ldots, y'_\beta =$ class labels of $((X'_1, A'_1, B'_1), \ldots, (X'_\beta, A'_\beta, B'_\beta))$.
best_error $= 2$.
best_low $= -1$.
best_high $= -1$.
for $i = 1 : \beta$ **do**
  if $i > 1$ **and** $F(X'_i) == F(X'_{i-1})$ **then**
    continue;
  end
  incorrect $= 0$.
  for $j = i : \beta$ **do**
    if $\tilde{F}(X'_j, A'_j, B'_j) \cdot y'_j < 0$ **then**
      incorrect $=$ incorrect $+ w'_j$.
    end
    if $j < \beta$ **and** $F(X'_i) == F(X'_{i+1})$ **then**
      continue;
    end
    error $= \frac{\beta-(j-i+1)}{2} +$ incorrect.
    if error $<$ best_error **then**
      best_error $=$ error.
      best_low $= i$.
      best_high $= j$.
    end
  end
end
if best_low $== 1$ **then**
  $\tau_1 = -\infty$.
end
else
  $\tau_1 = \frac{F(X'_{\text{best\_low}-1}) + F(X'_{\text{best\_low}})}{2}$.
end
if best_high $== \beta$ **then**
  $\tau_2 = \infty$.
end
else
  $\tau_2 = \frac{F(X'_{\text{best\_high}}) + F(X'_{\text{best\_high}+1})}{2}$.
end
$V_{F,j} = (\tau_1, \tau_2)$.

Algorithm 2: The algorithm for finding, given a 1D embedding F, the range $V_{F,j}$ of type 1 that minimizes the weighted training error of $\Gamma_{F,V_{F,j}}$.

if $F'_j(Q) \in V_j$), and we also check if $F'_j = F_i$, i.e., if F'_j is one of the (possibly multiple) occurrences of F_i in H. If those conditions are satisfied, we add the weight α_j to $A_i(Q)$.

Let $F_{\text{out}}(Q) = (q_1, ..., q_d)$, and let X be some other object in $\mathbb{X}$, with $F_{\text{out}}(X) = (x_1, ..., x_d)$. We define query-sensitive distance measure Δ as follows:

$$\Delta((q_1, ..., q_d), (x_1, ..., x_d)) = \sum_{i=1}^{d} (A_i(q)|q_i - x_i|) \ . \tag{5.8}$$

Distance measure $\Delta(u, v)$ (where u, v are d-dimensional vectors) is like a weighted L_1 measure on $\mathbb{R}^d$, but the weights depend on u. Therefore Δ is not symmetric, and not a metric. We say that Δ is a *query-sensitive* distance measure, because we use Δ in such a way that u is always the embedding of a query, and v is the embedding of a database object that we want to compare to the query.

Procedures for Computing Query-Sensitive Weights

Equation 5.7 defines the query-sensitive weight $A_i(Q)$ assigned to the i-th dimension of embedding F_{out} given a query object Q. A naive algorithmic implementation of Equation 5.7 takes time linear to the number J of training rounds performed by AdaBoost, which would lead to $O(Jd)$ total time for computing the weight corresponding to each of the d dimensions of F_{out}. Here we describe algorithmically how to compute the weights for all dimensions of F_{out} in $O(d)$ total time.

First, after the training algorithm is completed, we perform an offline procedure, in which we identify the unique embeddings F_i used in the strong classifier of Equation 5.6. During this procedure, we also mark in an array called unique, for each F'_j used in the strong classifier, the integer i such that $F'_j = F_i$. This offline procedure consists of the following steps:

1. $d = 1, j = 1$.
2. $F_1 = F'_1$.
3. unique[1] = 1.

4. $j = j + 1$.

5. If $j > J$ exit.

6. already_exists = 0.

7. For $d' = 1 : d$

   - If $F'_j == F_{d'}$

     – already_exists = 1.

     – unique[j] = d'.

     – break;

8. If already_exists == 0

   - d = d + 1.

   - unique[j] = d.

9. Go to Step 4.

Now, during the online application of the system, given a query Q for which we want to retrieve the nearest neighbors, the following procedure computes the query-sensitive weights for all dimensions of F_{out} in $O(J)$ time, and stores those weights in an array called weights, such that weights[i] is $A_i(Q)$:

1. Set weights[1], ..., weights[d] to 0.

2. For $j = 1 : J$

   - If $F'_j(Q) \in V_j$

     – d' = unique[j].

     – weights[d'] = weights[d'] + α_j.

As a reminder, the symbols F'_j, V_j, and α_j used in the above procedure are defined in Equation 5.6.

5.4 Properties and Discussion of the Method

In this section we take a closer look at some properties of query-sensitive embeddings and the proposed algorithm for constructing such embeddings.

5.4.1 Complexity

With respect to the *online* filter-and-refine retrieval cost, computing the d-dimensional embedding of a query object takes $O(d)$ time and requires $O(d)$ evaluations of D_X, exactly as in the original BoostMap algorithm from the previous chapter. The only additional computation we need to perform, with respect to the query-insensitive version of BoostMap, is computing weights $A_i(Q)$ for each dimension i given query object Q. However, computing those weights can easily be done in $O(J)$ time, as discussed in the previous section. In all our experiments this part of the computation is at least three orders of magnitude faster than computing the distances we need to measure in order to produce the embedding of the query. In summary, the additional computational cost of using query-sensitive embeddings for online retrieval is negligible, and for all practical purposes this additional cost can be ignored.

With respect to the *offline* cost of the training algorithm, there is one additional computation we need to perform compared to the query-insensitive version of BoostMap: for each weak classifier $\tilde{F}$ considered at a particular training round j we need to compute an interval $V_{F,j}$ so as to convert $\tilde{F}$ to a query-sensitive classifier $\Gamma_{F,V_{F,j}}$. Our algorithm for finding $V_{F,j}$ takes time linear to the number of training triples and quadratic to the size of $\mathbb{T}$, which is the set from where we pick objects for the training triples. In practice, the query-sensitive training algorithm takes about two-three times longer to run than the query-insensitive version. Naturally, in many applications, the additional training time is an acceptable overhead, since it represents a one-time preprocessing cost, as long as this additional cost leads to better online retrieval and classification performance.

5.4.2 Contractiveness

Similar to query-insensitive BoostMap embeddings, a query-sensitive embedding $F_{\text{out}} : (\mathbb{X}, D) \rightarrow (\mathbb{R}^d, \Delta)$, constructed as described this chapter, can be made contractive by dividing Δ with a query-sensitive normalization term, provided that D is metric.

If F_{out} contains no line projection embeddings, the normalization term $W(Q)$ that should be used given query Q is:

$$W(Q) = \sum_{i=1}^{d} A_i(Q) \ . \tag{5.9}$$

The proof that dividing with this normalization term makes the embedding contractive is almost identical to the proof we provide for query-insensitive embeddings in the previous chapter. Also, following the same reasoning that we used for query-insensitive embeddings, if F uses line-projection 1D embeddings, then to make F_{out} contractive we need to divide Δ by $3W(Q)$.

5.5 Summary of Query-Sensitive Embeddings

In this chapter we have proposed using a novel type of embeddings, *query-sensitive* embeddings, for efficient nearest neighbor retrieval. Query-sensitive embeddings use a weighted L_p distance measure for the target space of the embedding, where the weights of the L_p measure depend on each query. Such a distance measure can capture the important fact that different dimensions of the embedding are important for different queries. This fact cannot be captured by traditional embeddings that use global, query-insensitive distance measures.

We have described how to modify the original BoostMap algorithm of Chapter 4 so as to construct query-sensitive embeddings. In a way, the query-sensitive version of BoostMap illustrates the flexibility of the BoostMap method, and its ability to construct novel types of embeddings that cannot be obtained using alternative methods, like FastMap or Lipschitz embeddings; it is not clear if and how such alternative methods can be modified to become query-sensitive.

An important property of query-sensitive embeddings is that computing the query-sensitive weights given a query takes negligible time. Therefore, during online retrieval, using a query-sensitive distance measure does not introduce any additional run-time costs to filter-and-refine retrieval. On the contrary, as we will see in Chapter 7, by capturing more of the nearest neighbor structure of the original space, query-sensitive embeddings achieve significantly better trade-offs between accuracy and efficiency than their query-insensitive counterparts.

Chapter 6

Efficient Nearest Neighbor Classification Using Cascades of Approximate Classifiers

As mentioned in the introduction, nearest neighbor classifiers are appealing because of their simplicity, ability to model a wide range of complex, non-parametric distributions, and very competitive classification accuracy in many applications. However, finding the nearest neighbors of an object in a large database can take too long, especially in domains that employ computationally expensive distance measures. The embedding methods proposed in the previous chapters can speed up classification by speeding up retrieval of the nearest neighbors.

The key message in this chapter is that nearest neighbor classification is a different problem than nearest neighbor retrieval. In nearest neighbor classification, the ultimate goal is to *classify* the query correctly, not to find the true nearest neighbors of the query. As a consequence, a retrieval result that is considered undesirable for the purposes of finding nearest neighbors may be considered desirable for the purposes of classification. As a simple example, suppose that query object Q belongs to class C_1, but the nearest neighbor $NN(Q, \mathbb{U})$ of Q in the database $\mathbb{U}$ belongs to class $C_2 \neq C_1$. If our goal is to find the true nearest neighbor of Q, then retrieving $NN(Q, \mathbb{U})$ is the desired result. However, if our goal is to classify Q correctly, then we would rather have the system retrieve another object, whose class label is the same as that of Q.

The method proposed in this chapter aims to improve the accuracy and efficiency of nearest neighbor *classification*. We will build the proposed method on top of the Boost-Map algorithm, but we will modify the embedding construction algorithm and describe a different online classification algorithm. The key idea is that we are willing to sacrifice nearest neighbor *retrieval* accuracy, as long as we can improve efficiency and simultaneously improve, or at least minimally affect *classification* accuracy.

The modification of the embedding construction algorithm consists of using a different optimization cost, that is more appropriate for classification accuracy. In particular, given a triple (X, A, B), whether we want the embedding to map X closer to A or to B now depends not on the distances $D(X, A)$ and $D(X, B)$, but on the class labels of X, A, and B. The online classification method that we use is a cascade of approximate nearest neighbor classifiers. Using BoostMap embeddings and the filter-and-refine framework [Hjaltason and Samet, 2003a] we construct a sequence of approximations of the exact nearest neighbor classifier. The first approximation in that sequence is relatively fast, but also has a relatively high classification error rate. Each successive approximation in the sequence is slower and more accurate than the previous one. These approximations are combined in a cascade structure, whereby easy cases are classified by earlier classifiers, and harder cases are passed on to the slower but more accurate classifiers. The key idea is that as soon as we have enough information to classify a query with high confidence, we can just produce the classification, even if we do not have enough information to confidently estimate the true nearest neighbors of the query.

6.1 Some Additional Related Work

Clearly the problem of efficient nearest neighbor classification in spaces with computationally expensive distance measures has a large overlap with the problem of efficient retrieval in such spaces. Methods that accurately and efficiently retrieve nearest neighbors, such as the methods surveyed in Chapter 3, can also be used to speed up nearest neighbor classification. However, the the method proposed in this chapter focuses explicitly on efficient classification; in order to achieve that goal we are willing to sacrifice nearest neighbor *retrieval* accuracy, as long as classification accuracy does not suffer. Methods for efficient retrieval do not take classification accuracy into account, and therefore are not formulated to make trade-offs between retrieval and classification performance.

With respect to existing embedded methods, such as Lipschitz embeddings [Hjaltason and Samet, 2003a FastMap [Faloutsos and Lin, 1995], MetricMap [Wang et al., 2000], SparseMap [Hristescu and Farach-Colt and the two embedding methods described in [Athitsos et al., 2004, Athitsos et al., 2005b] and in Chapters 4 and 5 respectively, the cascade of classifiers described in this chapter is a complementary method, in the sense that it can be applied on top of any of these embedding methods. The key idea is that, instead of using existing embedding methods to produce a single approximate nearest neighbor classifier, we can produce a cascade of such classifiers, in increasing order of computational complexity and accuracy. A classical application of existing embedding methods essentially corresponds to the last and most accurate classifier in the proposed cascade.

Cascades of classifiers have been very popular in recent years [Li and Zhang, 2004, Ong and Bowden, 2004, Viola and Jones, 2001]. However, typically cascades are applied to a binary, unbalanced "detection" problem, where the goal is to determine whether a given image window contains an instance of some object, and we can safely assume that the overwhelming majority of windows do *not* contain such an instance. In contrast, our method produces a cascade of classifiers for a balanced, multiclass problem. In Sections 6.3 and 6.4 of this chapter we discuss in detail the issues that must be addressed in order to apply the cascade framework to such a problem.

6.2 Optimizing for Classification Accuracy

The original BoostMap algorithm aims at preserving similarity rankings, and minimizes the fraction of triples (X, A, B) where X is closer to A than to B, but $F(X)$ is closer to $F(B)$ than to $F(A)$. If our end goal is classification accuracy, then for some triples of objects the optimization criterion of BoostMap can be problematic. For the purposes of illustration, we will use as an example the MNIST dataset of handwritten digits, where each object is an image displaying a number between "0" and "9", and the goal is to recognize the number displayed in each test image.

As a first example where the optimization criterion of BoostMap can be problematic, let X be an image of the digit "2", and A and B be images, respectively, of digits "0" and "1". Furthermore, let A be one of the nearest neighbors of X. For the purposes of accurate nearest neighbor retrieval we want an embedding F that maps X closer to A than to B. However, for the purposes of classification, it is irrelevant whether F maps X closer to A or to B. A more interesting example is the following: suppose that X and B are both images of the digit "2", and A is an image of the digit "0". Furthermore, suppose that A is the nearest neighbor of X among all database objects. For classification purposes, we would actually prefer an embedding that maps X closer to B than to A, although the original BoostMap training algorithm would penalize for that. In both these examples, the optimization criterion of BoostMap is not tightly connected to classification accuracy, and does not capture our intuition of what constitutes desirable embedding behavior.

To address these problems, we propose the following modifications to the process of selecting training triples and measuring the optimization cost:

- Useful training triples are triples (X, A, B) such that A is one of the nearest neighbors of X among objects *of the same class* as X, and B is a database object belonging to some other class.

- The optimization cost that should be minimized is the number of training triples (X, A, B) such that the output embedding F maps X closer to B than to A, regardless of whether X is actually closer to B than to A in terms of distance measure D. Since A is of the same class as X, we want X to be mapped closer to A than to B.

Based on the above guidelines, we design an alternative version of the BoostMap method, called "BoostMap-C", whose optimization criterion is more tightly connected to classification accuracy than the criterion of the original BoostMap method. BoostMap-C is obtained by modifying the process of selecting and labeling training triples that is described in Section 4.3.1. Given a subset $\mathbb{T}$ of the database, and given a parameter k_1, to specify training triple (X_i, A_i, B_i) for BoostMap-C we perform the following steps:

1. Choose X_i randomly from $\mathbb{T}$.

2. Choose A_i randomly from the k_1 nearest neighbors of X_i among all objects in $\mathbb{T}-\{X_i\}$ that have the same class label as X_i.

3. Choose (randomly) a class C different than that of X_i.

4. Set B_i randomly to one of the k_1 nearest neighbors of X among objects of class C in $\mathbb{T}$.

5. Set training triple o_i to (X_i, A_i, B_i).

6. Set class label y_i of o_i to 1, reflecting that we want the output embedding to map X_i closer to A_i than to B_i, since X_i and A_i have the same class label.

The BoostMap-C method tries to construct embeddings that map objects of each class near other objects that belong to the same class, even when such mappings violate the nearest neighbor structure of the original space. By mapping objects close to other objects of the same class we aim to achieve improved nearest neighbor classification accuracy.

We now turn our attention to improving classification efficiency, using cascades of approximate nearest neighbor classifiers. First we provide a brief description of the popular framework of classifier cascades, and then we describe how to apply that framework to embedding-based nearest neighbor classification.

6.3 Overview of Cascades of Classifiers

A cascade of classifiers is essentially a special case of a decision tree. The tree is represented as a sequence of nodes N_i, where $i = 1, \ldots, s$, and node N_1 is the root of the tree. Except for nodes N_1 and N_s, every node N_i has one parent, N_{i-1}, and one child, N_{i+1}. The i-th node of the tree stores a classifier P_i and a decision function $T_i : \mathbb{X} \rightarrow \{0, 1\}$. No decision function is needed for the last node N_s, or, alternatively, we can think that $T_s(Q) = 1$ for all objects Q. Given a query object Q that we want to classify, we apply the following process:

1. Set $i = 1$.

2. If $T_i(Q) = 1$, return $P_i(Q)$.

3. If $i = s$, return $P_s(Q)$.

4. Set $i = i + 1$.

5. Go to Step 2.

Typically, each classifier P_{i+1} is slower and more accurate than the previous classifier P_i. The role of each decision function T_i is to determine, given a query object Q, if we should trust the output of P_i on Q, or we should pass on the query to subsequent classifiers. In designing a cascade, ideally we want the overall classification accuracy of the cascade to be equal to the accuracy of the final classifier P_s. In order to achieve that, we want to design decision functions T_i such that $T_i(Q) = 1 \Rightarrow P_i(Q) = y(Q)$. In other words, T_i should allow Q to be classified by classifier P_i only in cases where $P_i(Q)$ is the correct classification. Intuitively, it is OK if $T_i(Q) = 0$ for some objects Q that are correctly classified by P_i, but it is not OK if $T_i(Q) = 1$ for some objects Q that are incorrectly classified by P_i. Naturally, the decision functions are typically not ideal. Still, the overall design principle is that we want each T_i to allow only a small fraction of objects to get misclassified by P_i.

The computational savings of employing a cascade, as opposed to simply using the final classifier P_s, depend on the ability of functions T_i to direct a large majority of queries to the earlier, more efficient classifiers in the cascade. In that case, the average classification time mainly depends on the speed of those earlier classifiers, and not on the speed of the final classifier P_s. At the same time, as long as we avoid (at least to a large extent) misclassifications by the earlier classifiers, the classification accuracy of the cascade mainly depends on the accuracy of P_s. Overall, the goal of a cascade is to provide the best of both worlds, by combining the speed of computationally efficient classifiers and the accuracy of more complicated and slower classifiers.

Cascades of classifiers are typically applied to unbalanced binary problems where the overwhelming majority of objects belong to one class and a very small fraction of objects belongs to the second class. Detection is the classical setting where cascades have been applied, including face detection [Li and Zhang, 2004, Viola and Jones, 2001] and hand detection [Ong and Bowden, 2004]. In detection tasks the number of positive objects (i.e., faces or hands) present in image is a very small fraction of the number of negative objects, i.e., all image subwindows that do not correspond to a face or a hand. Cascades can significantly speed up such detection tasks by taking advantage of the fact that most of the negative objects can easily be rejected using simple classifiers. Consequently, the cascade methods described in [Li and Zhang, 2004, Ong and Bowden, 2004, Viola and Jones, 2001] allow early classifiers in the cascade to output only negative class labels; positive labels are only output by the last classifier of the cascade.

Our setting, which is nearest neighbor classification in domains with an arbitrary number of classes, differs from the typical cascade setting in two important aspects:

- The classification problem is not constrained to be binary, and actually we typically deal with non-binary classification tasks. In the experiments we apply the cascade method to two datasets where the number of classes is ten. In existing cascade methods [Li and Zhang, 2004, Ong and Bowden, 2004, Viola and Jones, 2001] the decision function is typically a thresholded binary classifier that estimates if the test object is positive or negative. In our setting we need to design decision functions that can be applied in non-binary classification problems.

- The classification problem is not constrained to be unbalanced. In fact, both of the datasets we apply our cascade method to are balanced, meaning that the frequencies of the different classes are pretty similar. Therefore, we need decision functions that can allow any node of the cascade to produce any classification output. This is in contrast to the methods in [Li and Zhang, 2004, Ong and Bowden, 2004, Viola and Jones, 2001], where a positive classification can only be obtained at the last stage of the cascade.

In the next section we proceed to describe how to construct a cascade of classifiers for embedding-based nearest neighbor classification.

6.4 Constructing a Cascade of Approximate Nearest Neighbor Classifiers

Suppose that we have used some embedding construction method, like BoostMap or the BoostMap-C method described earlier in this chapter, and we have obtained an embedding F. We can use embedding F for k-nearest neighbor classification in a filter-and-refine framework [Hjaltason and Samet, 2003a], as described in Section 2.4. Given the final ranking produced by the filter and refine steps, we can classify query Q based on majority voting among the retrieved k nearest neighbors. Note that we can set filter-and-refine parameter p, which specifies the number of objects to retain after the filter step, to 0. In that case no refine step is performed, we simply use the ranking produced by the filter step.

If we apply embedding F as described in the previous paragraph, we need to make a practical choice about the dimensionality d' of the embedding and the number p of distances to evaluate during the refine step. Low values for d' and p lead to faster but potentially less accurate results. On the other end of the spectrum, high values of d' and p lead to slower but potentially more accurate results. The correspondence of this problem to the typical cascade problem is clear: we would like to combine the best of both worlds, i.e., the efficiency obtained using small d' and p values and the accuracy obtained using large d' and p values.

In order to design a cascade method for our problem we need to address three issues:

- How to construct a sequence of classifiers $P_1, \dots, P_s$ to be used in the nodes of the cascade.
- What family of decision functions T_i we want to use.
- How to select an appropriate decision function for each node.

We now proceed to describe how we address each of these three issues.

6.4.1 Constructing a Sequence of Classifiers

Let F be the embedding obtained using BoostMap (or some other embedding method), and let d' be the dimensionality of F. For any $d \in \{1, \ldots, d'\}$, we define embedding F_d to be the embedding consisting of the first d dimensions of F. Given positive integers $d \leq d'$ and p we define filter-and-refine process $F_{d,p}$ to be the the filter-and-refine process that uses F_d as the embedding, and p as the parameter for filter-and-refine retrieval. We also treat process $F_{d,p}$ as a classifier, that assigns a class label to each query Q based on the nearest neighbors of Q that $F_{d,p}$ retrieves. Naturally, given a query object Q, as d and p increase, the approximate similarity ranking obtained using process $F_{d,p}$ will get closer to the correct ranking. For example, if p is equal to the number of training objects, then process $F_{d,p}$ becomes equivalent to brute-force search: at the refine step we simply compare the query object with every database object. On the other hand, small d and p allow the filter-and-refine process $F_{d,p}$ to give results very fast.

With appropriate choices of d_i and p_i we can construct a sequence $\mathbb{P} = (P_1, ..., P_s)$ of s filter-and-refine processes $P_i = F_{d_i,p_i}$, such that each successive process P_i is less efficient and more accurate than the preceding process P_{i-1}. An example of such a sequence $\mathbb{P}$ is given in Table 7.7 of Chapter 7.

We construct such sequences manually, we have found it to be a pretty straightforward process. The main guideline in designing such a sequence is that, given a query Q, each process P_i should be able to reuse all the work performed by previous processes. For example, if $d_1 = 10, d_2 = 20, p_1 = 10, p_2 = 20$, then process P_1 performs some work that is reusable by process P_2, and some work that might not be reusable by P_2. The reusable work is the computation of the first 10 dimensions of $F(Q)$. Process P_2 only needs to compute dimensions $11 - 20$ of $F(Q)$. On the other hand, the ten exact distances computed at the refine step of P_1 might not be reusable for P_2, since we have no guarantee that the ten objects that survived the filter step of P_1 will be included in the twenty objects that survive the filter step of P_2.

A simple way to address the issue of reusability is to make the sequence $\mathbb{P}$ of classifiers consist of two parts. The first part is a sequence where for each P_i we set $p_i = 0$, so that no refine step is performed for that process. For sequences P_i in the first part, the dimensionality of embeddings increases as i increases. Each such P_i simply needs to compute some additional dimensions of the embedding that have not been computed yet. The second part of sequence $\mathbb{P}$ consists of processes P_i that all have the same dimensionality, but different values of p_i, that increase as i increases. Therefore, each process P_i in the second part of $\mathbb{P}$ can reuse the exact distances computed by the refine steps of all previous processes; P_i simply evaluates some additional exact distances and reranks the p_i objects that survived the filter step based on their exact distances to the query. The sequence $\mathbb{P}$ shown in Table 7.7 of Chapter 7 follows these guidelines, so that processes P_1 to P_6 use no refine step, and processes P_7 to P_{16} all have the same dimensionality.

Note that in considering the type of computations that should be reusable by subsequent processes we have ignored the vector distances computed at the filter step of each process, and the sorting of those distances in order to determine the objects that should be evaluated during the refine step. In all datasets we have evaluated so far, computing vector distances and sorting those distances takes negligible time compared to the time spent on evaluating exact distances. That is why exact distance computations are the only type of computation that we require to be reusable by subsequent processes in the cascade.

6.4.2 Specifying a Family of Decision Functions

In general, in order to combine a sequence of classifiers into a cascade structure, we need a way of deciding which objects are safe to be classified by each classifier in the sequence. In other words, given the similarity ranking that P_i produces for query object Q, we want to specify some criteria for deciding whether we can reliably classify Q using that ranking, or whether we need to pass on Q to the next process P_{i+1}. We want these decisions not to hurt overall classification accuracy, so that the cascaded classifier overall is as accurate as the final classifier P_s of the cascade.

In order to define an appropriate family of decision functions for the cascade, we first define a quantity $K(Q, P_i)$, which is a measure of confidence in the classification result for object Q obtained using process P_i. We define $K(Q, P_i)$ as follows: $K(Q, P_i)$ is the highest integer k such that all the k nearest neighbors of Q retrieved using process P_i belong to a single class. For example, if our objects are images of handwritten digits, suppose that according to P_i the 50 nearest neighbors of Q belong to class "1", and the 51st neighbor belongs to class "2". Then, $K(Q, P_i) = 50$. We use quantity $K(Q, P_i)$ to define a criterion $T_i(Q)$ for when P_i should be used to classify Q, as follows: Given a threshold t_i, if $K(Q, P_i) \geq t_i$ then $T_i(Q)$ outputs 1, and Q is classified based on the class of its approximate nearest neighbors retrieved by P_i. Otherwise $T_i(Q)$ outputs 0, and Q is passed on to the next classifier in the cascade.

The intuition behind this criterion is that if, for some test object, process P_i reports that the test object is surrounded by a large number of training objects of a single class, then we can confidently assign that class to the test object, without needing to spend any additional computation. Note that we are not concerned about actually finding the true nearest neighbor of the query: we stop as soon as we have sufficient information about the query's class label.

We should note that, in addition to the measure $K(Q, P_i)$ that we have proposed, there are alternative measures that can also be employed. For example, instead of measuring the number of nearest neighbors that all belong to the same class, we could measure the number of nearest neighbors such that at most k_2 objects do *not* belong to the same class as the other nearest neighbors, for some parameter k_2. As another example, we could use a measure that depends on the distance between Q and its nearest k_3 neighbors, for some parameter k_3, the intuition here being that the closer (i.e., the more similar) Q is to its nearest neighbors the more likely Q is to belong to the same class as those neighbors.

The family of decision functions that we have proposed here satisfies the two key requirements that we listed in Section 6.3: these decision functions can be used in non-binary classification problems, and they allow any cascade node to produce any classification output. Naturally, in order to achieve good results with these decision functions we need to choose appropriate values for the thresholds t_i, that are used to make the decision at each node. We now describe how to learn those thresholds using training data.

6.4.3 Learning the Decision Functions

Given a sequence $\mathbb{P}$ of classifiers that we want to combine into a cascade, and given that we want to use decision functions that check if $K(Q, P_i) \geq t_i$, we need a method for choosing thresholds t_i. We propose to choose those values using a validation set, sampled from the set of database objects. Let $e \geq 0$ be an integer parameter that specifies how many objects from the validation set we are allowed to misclassify by each P_i. Then, for the first process P_1 we can simply set t_1 to the smallest threshold t satisfying the following property: if we use process P_1 to classify all validation objects X satisfying the criterion $K(X, P_1) \geq t$, we misclassify at most e validation objects. For example, if $e = 2$, we can try all thresholds until we find the smallest threshold t such that, if we find all validation objects X with $K(X, P_1) \geq t$ and we label those objects with the label of the t nearest neighbors retrieved using P_1, we misclassify no more than two objects.

After we have determined the right threshold for process P_1, we proceed to select an appropriate threshold for P_2, using the same parameter e, and using only the validation objects X that P_1 does not classify (i.e., for which $K(X, P_1) < t_1$). Proceeding this way recursively we can choose all thresholds t_i. Naturally, the last process P_s in the sequence does not need a threshold, because P_s is the last step in the cascade, and therefore it needs to classify all objects that are passed on to it.

A slight problem with the above procedure for determining thresholds is that there may be some validation objects X that even the final process P_s will misclassify, and for which $K(X, P_i)$ is very high for all processes P_i. In our experiments, such objects were identified in practice. These objects are essentially outliers that look very similar to a large number of objects from another class. These objects are likely to influence the threshold choice t_i

for every P_i, so that t_i is large enough to avoid misclassifying those objects, even though they will end up being misclassified anyway at the final step P_s.

In principle, objects that are misclassified even by the most accurate process in the cascade should not influence the selection of thresholds for earlier processes. Such objects will be misclassified anyway, so whether they get misclassified earlier or later in the cascade is inconsequential with respect to the final error rate. According to this principle, before choosing thresholds for the different processes, we identify all validation objects that the final process P_s misclassifies. We then remove those objects from the validation set, and proceed with threshold selection without taking those objects into account.

The exact algorithm for picking a threshold for each node in a cascade is described in Algorithm 3.

6.5 Discussion of the Cascade Method

Intuitively, in order for a cascade method to produce significant computational savings without losses of accuracy, it has to be the case that a large fraction of test objects are unambiguous and easy to classify, and a small fraction of objects are harder to classify and require more resources for accurate classification. A key component of designing a cascade is defining decision functions that allow most of the easy objects to be classified during the early stages, while allowing hard-to-classify objects to be handled by the later, more accurate stages.

The key difference of the cascade method we have described here from existing cascade methods is that our method is designed for balanced multiclass problems, as opposed to unbalanced binary problems. We have proposed a family of decision functions that is appropriate for this setting and can identify easy-to-classify objects regardless of the class that they belong to. Intuitively, the decision functions try to determine whether the query object lies in a uniform region where all objects have the same class label. Naturally, in order for these functions to work well, it has to be the case that a large fraction of query objects indeed lie in such uniform regions, and it also has to be the case that we

---

input : $\mathbb{P}$: sequence of filter-and-refine processes $P_1, \cdots, P_s$.
s: number of filter-and-refine processes in sequence $\mathbb{P}$.
X_{train}: set of training objects.
$X_{\text{validation}}$: set of validation objects.
B: set of validation objects that are misclassified by process P_s.
e: a parameter specifying how many objects should be misclassified, at most, at each step in the cascade.
output : $t_1, \cdots, t_{s-1}$: The thresholds t_i to be used with each process P_i.

```
X_validation = X_validation − B
for i = 1 : (s − 1) do
    for q ∈ X_validation do
        K_q = K(q, P_i)
        N_q = nearest neighbor of q in X_train according to P_i
        Y_q = class label of q
        C_q = class label of N_q
    end
    t_i = size(X_train) + 1
    for t = 1 : size(X_train) do
        X_1 = {q ∈ X_validation | K_q ≥ t}
        X_2 = {q ∈ X_1 | Y_q ≠ C_q}
        if size(X_2) ≤ e then
            t_i = t
            break;
        end
    end
    X_validation = {q ∈ X_validation | K_q < t_i }
end
```

Algorithm 3: The algorithm for choosing thresholds for a cascade of filter-and-refine processes.

have enough objects in the database to populate those regions. If those assumptions are violated then the method proposed here is not expected to yield significant computational savings, but it is also not expected to be worse than simply using a single filter-and-refine retrieval process. The algorithm that determines the thresholds for the decision functions can be made pretty conservative, by setting parameter $e = 0$, so that we do not allow any node of the cascade (except the last node) to make any mistakes on the training data. In that case, when most objects lie in ambiguous, non-uniform regions, or when we do not have enough database objects to determine whether a region is uniform or not, we expect the decision thresholds to be automatically set in such a way that most objects are simply passed on to the last node of the cascade.

In practice, as we will see in the experimental evaluation in Chapter 7, cascades of approximate nearest neighbor classifiers produce significant computational savings in two of the datasets we have experimented with. Those datasets indeed satisfy the requirement that most objects lie in unambiguous, uniform regions. The decision functions that we have proposed successfully capture that fact and exploit it to improve classification efficiency while sustaining negligible losses in accuracy.

Chapter 7

Experiments

In this chapter we experimentally evaluate the three methods proposed in in the previous chapters: BoostMap, query-sensitive embeddings, and cascades of approximate nearest neighbor classifiers. We compare these methods to several alternative existing methods that can be used to speed up approximate nearest neighbor retrieval and classification. Experiments are performed on four different applications: hand shape classification using a database of hand images, offline handwritten digit recognition using the MNIST database of handwritten digits [LeCun et al., 1998], online handwritten digit recognition using the isolated digits benchmark (category 1a) of the UNIPEN Train-R01/V07 online handwriting database [Guyon et al., 1994], and similarity-based retrieval of time series using the benchmark time series dataset described in [Vlachos et al., 2003]. We describe each of the datasets in detail, and then we evaluate, in separate sections, the performance of the proposed methods on nearest neighbor retrieval and classification tasks. First, we provide results on nearest neighbor retrieval for the query-insensitive and the query-sensitive versions of BoostMap. Finally, we evaluate the performance of BoostMap and cascades of approximate nearest neighbor classifiers on nearest neighbor classification.

7.1 Datasets

In this section we provide details about each of the four datasets we use in the experiments, including the source of each dataset, and the distance measure that is used for measuring distances in each dataset.

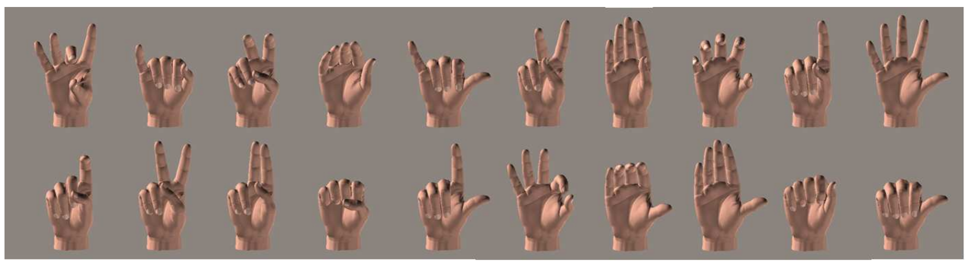

Figure 7·1: The 20 handshapes used in the ASL handshape dataset.

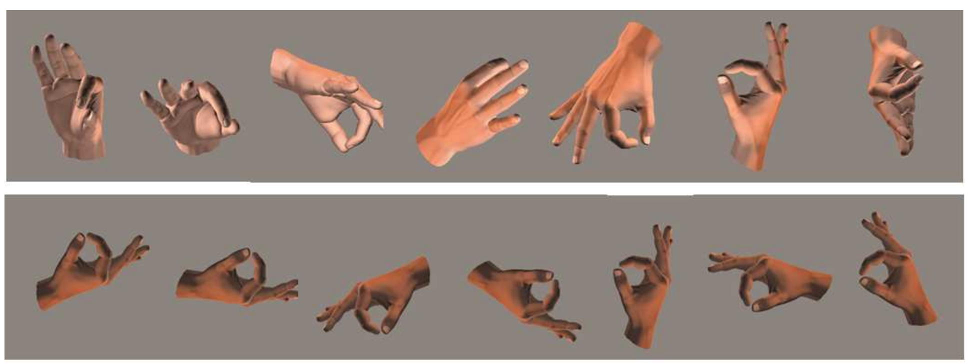

Figure 7·2: Examples of different appearance of a fixed 3D hand shape, obtaining by altering camera viewpoint and image plane rotation. Top: the ASL "F" handshape rendered from seven different camera viewpoints. Bottom: the ASL "F" handshape rendered from a specific camera viewpoint, using seven different image plane rotations.

7.1.1 ASL Handshape Dataset

The ASL handshape dataset consists of a database of $80,640$ synthetic images of hands, generated using the Poser 5 software [Curious Labs, 2002], and a test set of 710 real images of hands, used as queries. Both the database images and the query images display the hand in one of 20 different 3D handshape configurations. Those configurations are shown in Figure 7·1. Those 20 handshapes are all handshapes that are commonly used in American Sign Language (ASL). The target application is a system that can generate a short list of most likely handshapes for each query image, and that can handle query images displaying the hand in arbitrary 3D orientations. Such a system can be used as a plug-in to software tools that are currently used for manually annotating ASL video content [Neidle et al., 2001]. Annotation of large amounts of ASL video content is important both for facilitating the linguistic analysis of ASL, and for developing and training computer algorithms that can be used for automatic processing and interpretation of ASL video.

For each of the 20 different handshapes we synthetically generate a total of $4,032$ database images that correspond to different 3D orientations of the hand. In particular, the 3D orientation depends on the viewpoint, i.e., the camera position on the surface of a viewing sphere centered on the hand, and on the image plane rotation. We sample 84 different viewpoints from the viewing sphere, so that viewpoints are approximately spaced 22.5 degrees apart. We also sample 48 image plane rotations, so that rotations are spaced 7.5 degrees apart. Therefore, the total number of images is $80,640$ images, i.e., 20 handshapes $\times$ 84 viewpoints $\times$ 48 image plane rotations. Figure 7·2 displays example images of a handshape in different viewpoints and different image plane rotations. Each image is normalized to be of size 256×256 pixels, and the hand region in the image is normalized so that the minimum enclosing circle of the hand region is centered at pixel $(128, 128)$, and has radius 120.

The query images are obtained from video sequences of a native ASL signer either performing individual handshapes in isolation or signing in ASL. The hand locations were extracted from those sequences using the method described in [Yuan et al., 2005]. Accurate localization of the hand in such sequences remains a very challenging task, and hand localization fails in more than 50% of the frames. For the purposes of these experiments we only use frames where the hand is localized correctly. The query images are obtained from the original frames by extracting the subwindow corresponding to the hand region, and then performing the same normalization that we perform for database images, so that the image size is 256×256 pixels, and the minimum enclosing circle of the hand region is centered at pixel $(128, 128)$, and has radius 120.

The distance measure that we use to compare images is the chamfer distance [Barrow et al., 1977], which we have described in Section 1.3. The chamfer distance operates on edge images. The synthetic images generated by Poser can be rendered directly as edge images by the software. For the query images we simply apply the Canny edge detector [Canny, 1986]. On an AMD Athlon processor running at 2.0GHz, we can compute on average 715 chamfer distances per second. Consequently, finding the nearest neighbors of each query using brute force search, which requires computing the chamfer distances between the query image and each database image, takes about 112 seconds.

7.1.2 Offline Handwritten Digit Dataset (MNIST)

The offline handwritten digit dataset that we use is the well-known MNIST dataset of handwritten digits [LeCun et al., 1998]. The MNIST contains $60,000$ training images, which we use as the database, and $10,000$ test images, which we use as our set of queries. Each image is a 28x28 image displaying an isolated digit between 0 and 9. Example images are shown in Figure 7·3. The target application for this dataset is automatic recognition of the digit displayed in each test image.

The distance measure that we use in this dataset is shape context matching [Belongie et al., 2002]. As reported in [Belongie et al., 2002], 3-NN classification using shape context matching yields an error rate of 0.63%, when the database is a subset of $20,000$ images from the MNIST training set. In our experiments we have also measured the error rate obtained by using as a database the full MNIST training set of $60,000$ images, and we have found

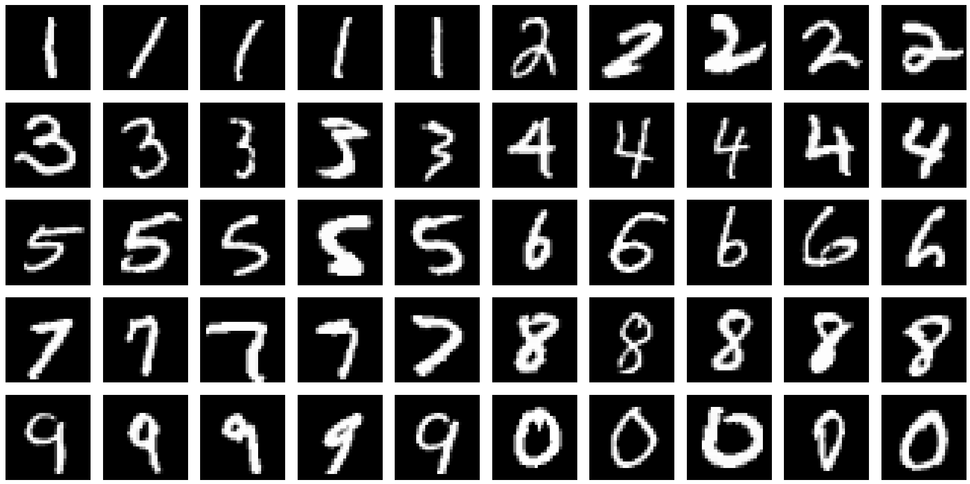

Figure 7·3: Example images from the MNIST dataset of handwritten digits.

that the error decreases from 0.63% to 0.54%. As can be seen on the MNIST web site (`http://yann.lecun.com/exdb/mnist/`), shape context matching outperforms in accuracy a large number of other methods that have been applied to the MNIST dataset.

Using our own heavily optimized C++ implementation of shape context matching, and running on an AMD Opteron 2.2GHz processor, we can compute on average 15 shape context distances per second. As a result, using brute force search to find the nearest neighbors of a query takes on average approximately 22 minutes when using the smaller database of $20,000$ images, and about 66 minutes when using the full database of $60,000$ images.

7.1.3 Online Handwritten Digit Dataset (UNIPEN)

The online handwritten digit dataset that we use is the isolated digits benchmark (category 1a) of the UNIPEN Train-R01/V07 online handwriting database [Guyon et al., 1994], which consists of $15,953$ digit examples. The digits have been randomly and disjointly divided into training and test sets with a 2:1 ratio (or 10,630 : 5,323 examples). We use the training set as our database, and the test set as our set of queries. The target application

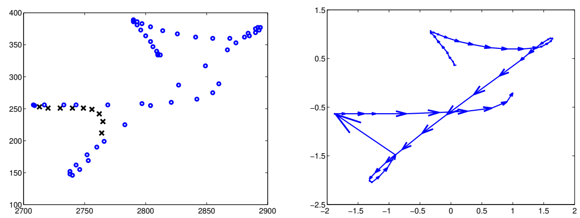

Figure 7·4: Left: Example of a "seven". Circles denote "pen-down" locations, x's denote "pen-up" locations. Right: The same example, after preprocessing.

for this dataset is automatic real-time recognition of the digit corresponding to each query.

Each query and database object in this dataset is preprocessed exactly as described in [Bahlmann and Burkhardt, 2004] Section 2. Each extracted feature is represented by three features: 2D normalized location $(\tilde{x}_i, \tilde{y}_i)$ and the tangent angle θ_i of the line segment between $(\tilde{x}_i, \tilde{y}_i)$ and $(\tilde{x}_{i-1}, \tilde{y}_{i-1})$. Figure 7·4 shows an example digit "seven" before and after preprocessing.

The distance measure D used for classification is Dynamic Time Warping [Kruskall and Liberman, 1983] On an AMD Athlon 2.0GHz processor, we can compute on average 890 DTW distances per second. Therefore, nearest neighbor classification using brute-force search takes about 12 seconds per query. The nearest neighbor error obtained using brute-force search is 2.05%.

7.1.4 Time-series dataset

As our last dataset, we use the time series dataset described in [Vlachos et al., 2003]. To generate that dataset, various real datasets were used as seeds for generating a large number of time-series that are variations of the original sequences. Multiple copies of every real sequence were constructed by incorporating small variations in the original patterns as well as additions of random compression and decompression in time. The final dataset contains a "database" set of 32,768 sequences, and a "query" set of 50 sequences. Sequences are multi-dimensional, with an average size of 500 points each. The series were normalized by subtracting the average value in each dimension.

In order to get a clearer picture of performance, we have mainly used an alternative splitting of the dataset into database objects and query objects, so that we could obtain a significantly larger set of queries. To achieve that, we merged the query set and the database, and from the merged set we chose (randomly) a new set of 1,000 queries, with the remaining 31,818 objects used as the database. Unless indicated otherwise, the results reported for the time series dataset are with respect to the set of 1,000 queries.

Exact distances in this dataset are measured using constrained Dynamic Time Warping, with a warping length $\delta = 10\%$ of the total length of the shortest sequence under comparison as described in [Vlachos et al., 2003]. On average, on an AMD Opteron 2.2GHz processor, we can compute 60 distances per second. Consequently, brute-force retrieval of the nearest neighbors of a query takes on average 530 seconds, i.e., roughly nine minutes.

7.2 Evaluation Methodology and Parameter Choices

Our goal is to evaluate the methods proposed in the thesis on the tasks of approximate nearest neighbor retrieval and classification. In both tasks one of the performance measures is efficiency, which is simply measured by the average time it takes to process a single query. In all experiments with all datasets the entire database is stored in physical memory, so we do not need to worry about disk accesses.

In all our experiments, the time it takes to process a query is dominated by the number of exact distance computations that we need to perform. Other operations, such as the filter step of filter-and-refine retrieval, where we compare the embedding of the query to the embeddings of database objects, take negligible time (less than 0.1 seconds/query for all computations that are not part of measuring an exact distance). Based on that observation, we mainly report processing time per query by providing the average number of exact distances we need to measure per query. To convert the processing time to actual seconds one simply needs to divide the number of exact distances measured per query by the number of exact distances we can compute per second, as reported in Section 7.1 for each dataset.

In evaluating approximate k-nearest neighbor retrieval accuracy, we consider the retrieval result for a query to be correct if and only if *all* k-nearest neighbors of the query have been correctly identified. For example,if we measure accuracy on 50-nearest neighbor retrieval for a particular method and set of parameters, 95% retrieval accuracy means that for 95% of the queries we successfully identify all 50 nearest neighbors.

For the purposes of evaluating embedding methods, including BoostMap and alternative methods, we use those methods for the filtering step of filter-and-refine retrieval. To apply an embedding method in this way we need to specify two parameters: d, which is the dimensionality of the embedding, and p, which specifies the number of objects for which we measure exact distances during the refine step. If we construct an embedding (for example using the BoostMap method) with d' dimensions, and we set parameter d to a value less than d', then we simply use the first d dimensions of that embedding. In all experiments, d and p are selected to be the ones that maximize efficiency given a specific setting for retrieval accuracy. For example, if we want to measure the efficiency of an embedding method for 50-nearest neighbor retrieval with 95% accuracy, we first find, for that method and for many different dimensionalities d, the parameter p that is needed for each dimensionality in order to obtain the desired accuracy. Then, we simply choose the combination of d and p that requires the smallest amount of distance computations per query. We should emphasize that we perform this parameter selection to all embedding methods that we evaluate.

With respect to the additional free parameters that are needed by the BoostMap algorithm, we use the same values in all experiments, except, of course, experiments where we explicitly measure the effect of tuning a specific parameter. There is one additional exception, noted below, where due to the small size of the UNIPEN dataset we use a different value for one parameter. The parameter values that we use are the following:

- $k_{\max} = 50$. Parameter $k_{\max}$ specifies the maximum number of nearest neighbors that we want to retrieve, and is used for choosing training triples.

- $|\mathbb{C}| = |\mathbb{T}| = 5000$. Set $\mathbb{C}$ contains the objects from which reference objects and pivot objects are selected to define 1D embeddings. Set $\mathbb{T}$ contains the objects from which we form training triples. The only dataset where we use different values is the UNIPEN dataset, for which $|\mathbb{C}| = |\mathbb{T}| = 3500$. The reason we use a different value here is that the database of the UNIPEN dataset contains only $10,630$ objects, and (for the purposes of training cascades, as described later in this chapter) we want at least one third of database objects to be included in neither $\mathbb{C}$ nor $\mathbb{T}$.

- $\beta = 300,000$. Parameter β is the number of training triples used by the training algorithm for embedding construction.

- $\gamma = 2000$. Parameter γ is the number of weak classifiers that are evaluated at each training round.

- $\delta = 200$. Parameter δ is the number of weak classifiers that we evaluate using function $Z_{\min}$ at each training round j. The remaining weak classifiers are discarded after measuring their weighted training error Λ_j.

- $Z_{\max} = .9999$. Parameter $Z_{\max}$ is used to decide when to stop the training algorithm.

- $k_1 = 5$. Parameter k_1 is used for choosing training triples for the BoostMap-C algorithm, which, as described in Chapter 6, is the embedding method we use for constructing a cascade of approximate nearest neighbor classifiers.

7.3 Methods Used for Comparison Purposes

In order to better evaluate the performance and competitiveness of the methods proposed in this thesis, we have implemented several alternative methods for speeding up nearest neighbor retrieval and classification. For three of the datasets (MNIST, UNIPEN, and time series) we also compare the proposed methods to domain-specific methods that have been applied to these datasets. We have not implemented ourselves these domain-specific methods, we simply report the results listed in the corresponding publications, as measured on the same data that we evaluate our methods with. In this section we provide a list of all methods that we use for comparison purposes:

- **Embedding methods:**
  - **FastMap.** FastMap is a popular embedding method, introduced in [Faloutsos and Lin, 1995], and briefly described in Section 2.3.2. For each dataset, we have constructed a FastMap embedding by running the FastMap algorithm on a subset of the database, containing 5000 objects (3500 objects for the UNIPEN dataset). The subset used for each dataset is the set $\mathbb{C}$ of candidate objects that we have used for BoostMap.
  - **Random reference objects (RRO)**. In this method we simply construct a multi-dimensional Lipschitz embedding as a concatenation of multiple 1D embeddings, where each 1D embedding is obtained by choosing a random reference object P from the database and applying Equation 2.7.
  - **Random line projections (RLP)**. In this method we construct a multi-dimensional embedding as a concatenation of multiple 1D embeddings, each of which is defined by choosing two random database objects X_1, X_2 as pivot objects and applying Equation 2.10.
- **Other general nearest neighbor retrieval methods:**
  - **VP-trees.** VP-trees are introduced in [Yianilos, 1993] and are a pruning-based method for efficient nearest neighbor retrieval in arbitrary spaces. In metric spaces, VP-trees rely on the triangle inequality to achieve efficient retrieval while always finding the true nearest neighbors. Since the distance measures in our experiments are all non-metric, we modify the VP-tree nearest-neighbor search algorithm using a method similar to [Sahinalp et al., 2003]. The modification guarantees correct retrieval results assuming that the triangle inequality is satisfied up to a constant ζ. Larger values of ζ lead to more accurate results and slower retrieval time.

- **General methods for improving nearest neighbor classification efficiency:**
  - **Condensed Nearest Neighbor (CNN)**. This methods is introduced in [Hart, 1968] and speeds up nearest neighbor classification by removing from the database any object that is not needed for correct classification of other database objects.
- **Domain-specific methods:**
  - **Zhang 2003.** This method, introduced in [Zhang and Malik, 2003], has been applied for efficient offline handwritten digit classification using shape context matching, and has been tested on the MNIST dataset.
  - **CSDTW.** This method is described in [Bahlmann and Burkhardt, 2004]. The CSDTW method has been applied to the problem of efficient online handwritten character classification, and has been tested on the UNIPEN dataset.
  - **Vlachos 2003.** This is a method for efficient similarity-based retrieval of time series, and has been applied to the time series dataset that we use in our experiments, using constrained DTW as the underlying distance measure [Vlachos et al., 2003].

7.4 Evaluation of the Original BoostMap Method

In this section, we evaluate the original BoostMap method, as described in Chapter 4. We have applied BoostMap to all four datasets. We compare the trade-offs of accuracy vs. efficiency obtained using BoostMap to the trade-offs achieved using FastMap, RRO embeddings, RLP embeddings, and VP-trees.

In Figures 7·5, 7·6, 7·7, and 7·8 we compare BoostMap to alternative methods on the task of k-nearest neighbor retrieval, on all four datasets. We provide results for different values of k, between 1 to 50, and different percentages of retrieval accuracy, between 90% and 99%. In Tables 7.1, 7.2, 7.3, and 7.4 we show, for selected accuracy percentages and values of k, the number of exact distance computations required by each method.

The results demonstrate that BoostMap clearly outperforms all other methods in three of the four datasets, namely the ASL handshape, MNIST and UNIPEN datasets. The performance difference between BoostMap and the other methods varies depending on the setting, i.e., the desired accuracy and the number of nearest neighbors to retrieve. In many settings BoostMap achieves retrieval times that are from 50% to over 300% faster than the times attained by the best alternative method. Of all alternative methods, the method that uses random reference objects gives the best performance. The time series dataset is the only dataset where BoostMap is not the best-performing method. In that dataset, using random reference objects provides results that are roughly as good as those of BoostMap for 90% and 95% retrieval accuracy, and results that are better than those of BoostMap for 99% retrieval accuracy.

The results on the time series dataset illustrate one limitation of the training algorithm: since we use AdaBoost as the underlying training method, the classifier that is constructed is not a globally optimal classifier. AdaBoost is essentially a greedy optimization method that finds locally optimal solutions. It is possible in some cases to obtain a better classifier using random choices. At the same time, we will see that the query-sensitive version of BoostMap significantly improves performance on the time series dataset over the query-insensitive version evaluated in this section. The query-sensitive version outperforms alternative methods in all datasets.

It is important to make a couple of remarks about results obtained for 100% retrieval accuracy. We provide such results in several tables, in order to offer a more complete picture of the performance of different methods. At the same time, the reader should bear in mind that those results, for each setting and each method, are completely dominated by the single query for which retrieval is the most inaccurate. Therefore, those results are far more sensitive to outliers than the results shown for other accuracy settings. Furthermore, we should emphasize that obtaining perfect retrieval accuracy on the specific set of queries that we use does not provide any guarantees of perfect retrieval accuracy using a different set of queries.

A second set of experiments we have performed evaluates the usefulness of the method we describe in Chapter 4 for choosing training triples. In that chapter we argue that the proposed method leads to an optimization measure that is tightly connected to the amount of nearest neighbor structure preserved by the embedding. The original implementation of BoostMap, described in [Athitsos et al., 2004], constructed training triples by choosing random objects from the database. In Figures 7·9, 7·10, 7·11, and 7·12, we compare these two different methods of choosing training triples. We see that the method described in this thesis clearly outperforms the method that uses random training triples, for all settings in all four datasets.

7.5 Evaluation of Query-Sensitive Embeddings

In the results reported in the previous section, we see that approximate nearest neighbor retrieval using the query-insensitive version of BoostMap is the most computationally expensive for the MNIST and time series datasets. For example, if we want to obtain 95% retrieval accuracy on 1-nearest neighbor retrieval, processing time per query is 0.63 seconds for the ASL handshape dataset, 146 seconds for the MNIST dataset, 0.16 seconds for the UNIPEN dataset, and 95 seconds for the time series dataset. In this section, we apply the query-sensitive version of BoostMap, as described in Chapter 5, to these two challenging datasets, in order to achieve further performance improvements.

Since we use an adaptation of the BoostMap algorithm to construct query-sensitive embeddings, the most direct way to evaluate the advantages of query-sensitive embeddings is to compare these embeddings to the query-insensitive embeddings produced using the original BoostMap algorithm. In order to get a more comprehensive picture, we use two different methods for choosing training triples. The first method is the one proposed in Section 4.1.3. The second method chooses triples randomly, by picking objects at random from the set $\mathbb{T}$ passed into the training algorithm. This second method was used in our initial implementation of the BoostMap algorithm, as described in [Athitsos et al., 2004].

Overall, then, we evaluate four embedding methods, each of which is characterized

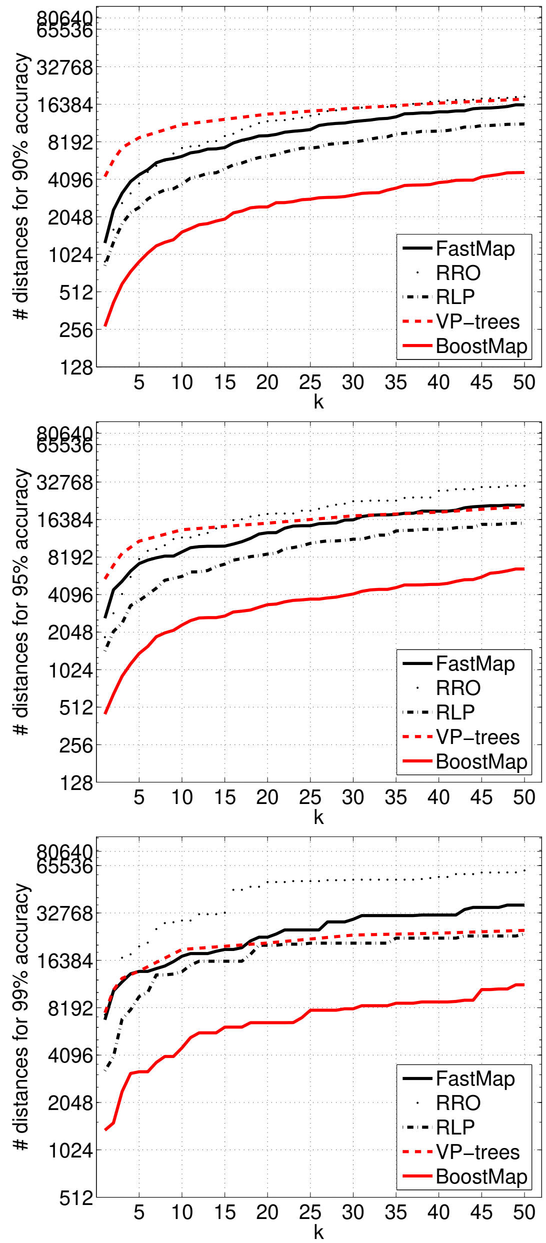

Figure 7·5: Comparing methods BoostMap, FastMap, RRO, RLP, and VP-trees, on the ASL handshape dataset, using the chamfer distance as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 710 query objects that we use as a test set.

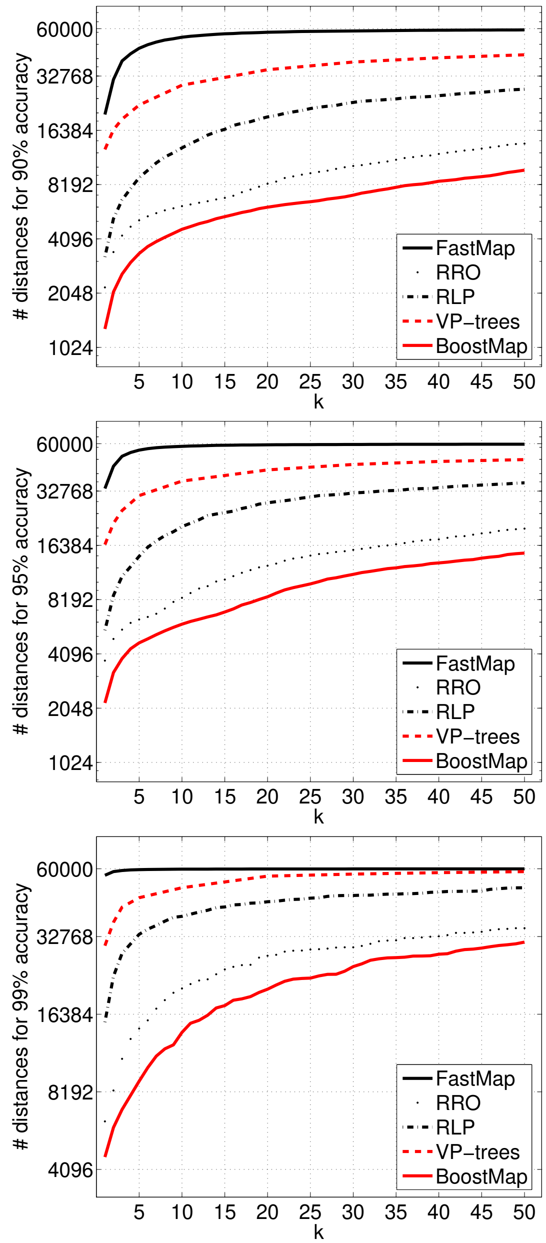

Figure 7·6: Comparing methods BoostMap, FastMap, RRO, RLP, and VP-trees, on the MNIST dataset, using shape context matching as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 10,000 query objects that we use as a test set.

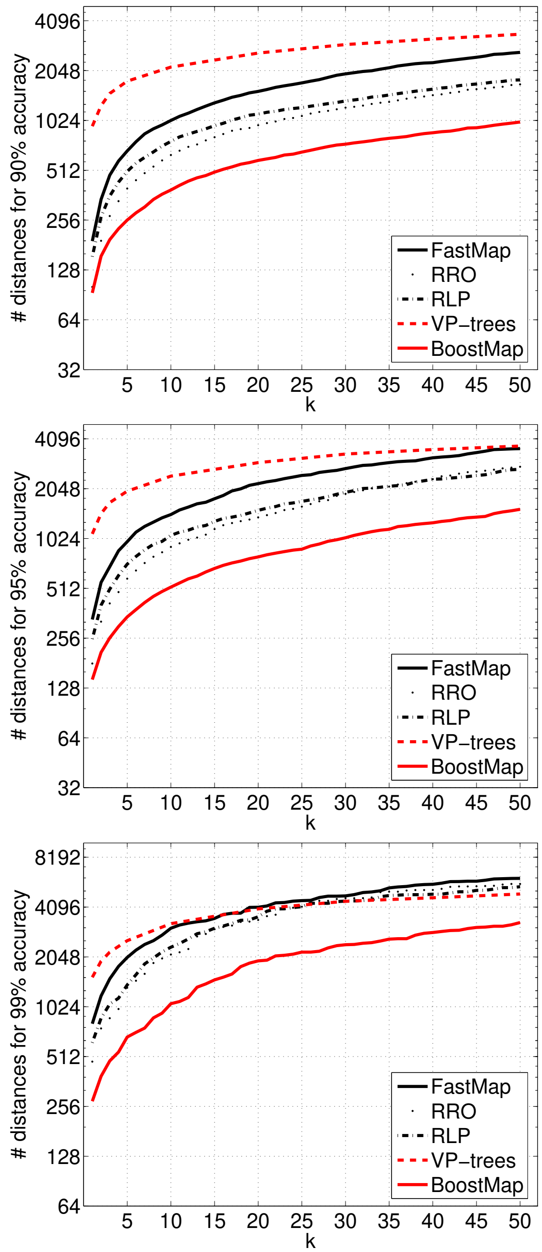

Figure 7·7: Comparing methods BoostMap, FastMap, RRO, RLP, and VP-trees, on the UNIPEN dataset, using DTW as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 10,000 query objects that we use as a test set.

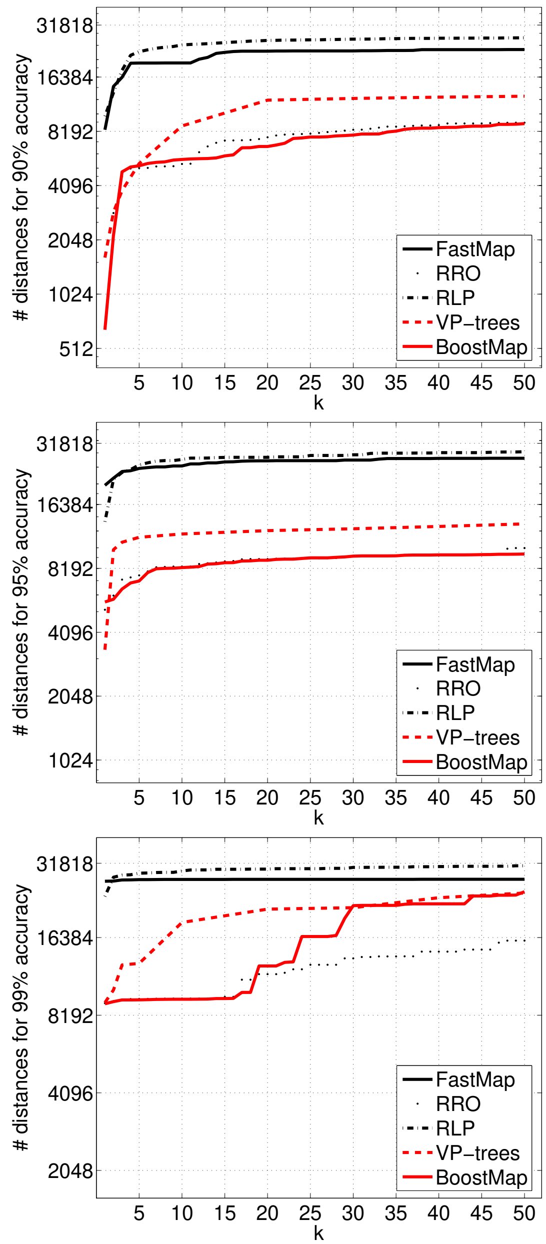

Figure 7·8: Comparing methods BoostMap, FastMap, RRO, RLP, and VP-trees, on the time series database, using constrained DTW as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 1,000 query objects that we use as a test set.

ASL Handshape Dataset with the Chamfer Distance						
k	accuracy	BoostMap	FastMap	RRO	RLP	VP-trees
1	90	**270**	1,259	895	831	4,303
1	95	**450**	2,647	1,866	1,444	5,471
1	99	**1,365**	6,865	7,405	3,259	7,611
1	100	**5,995**	16,242	24,950	13,235	12,012
10	90	**1,544**	6,251	7,301	3,745	11,261
10	95	**2,332**	9,023	11,766	5,736	13,573
10	99	**4,556**	17,402	29,345	13,890	19,198
10	100	**8,978**	47,577	58,031	23,894	27,448
50	90	**4,640**	16,198	18,826	11,381	18,006
50	95	**6,759**	21,373	30,585	15,305	20,892
50	99	**11,458**	36,753	61,009	24,094	25,395
50	100	**16,560**	65,948	59,961	43,363	30,836

Table 7.1: Comparison of BoostMap, FastMap, random reference objects (RRO), random line projections (RLP), and VP-trees, on the ASL handshape dataset. For different values of k, and different percentages of accuracy we show the number of exact distance computations required by each method. For each k and accuracy, we display in bold font the best result.

MNIST Dataset with Shape Context						
k	accuracy	BoostMap	FastMap	RRO	RLP	VP-trees
1	90	**1,296**	20,059	2,203	3,251	12,842
1	95	**2,190**	33,858	3,757	5,538	16,544
1	99	**4,577**	56,619	6,290	15,264	30,201
1	100	40,946	59,996	**37,753**	46,235	57,089
10	90	**4,631**	53,852	6,219	13,094	29,178
10	95	**5,988**	58,009	8,381	20,842	37,261
10	99	**13,932**	59,800	20,617	39,258	50,681
10	100	**56,936**	60,000	59,961	59,237	59,981
50	90	**9,856**	59,102	13,828	27,679	42,996
50	95	**14,848**	59,644	20,287	36,457	49,017
50	99	**31,176**	59,980	35,319	50,664	58,525
50	100	**59,735**	60,000	59,961	60,000	60,000

Table 7.2: Comparison of BoostMap, FastMap, random reference objects (RRO), random line projections (RLP), and VP-trees, on the MNIST dataset. For different values of k, and different percentages of accuracy we show the number of exact distance computations required by each method. For each k and accuracy, we display in bold font the best result.

UNIPEN Dataset with DTW						
k	accuracy	BoostMap	FastMap	RRO	RLP	VP-trees
1	90	**93**	191	101	154	945
1	95	**144**	332	180	253	1,092
1	99	**275**	809	477	621	1,542
1	100	**2,555**	2,685	3,268	2,614	2,791
10	90	**389**	1,024	633	765	2,152
10	95	**521**	1,429	911	1,070	2,439
10	99	**1,069**	3,051	2,118	2,344	3,252
10	100	**5,173**	8,808	7,126	8,932	10,630
50	90	**1,002**	2,639	1,692	1,801	3,397
50	95	**1,538**	3,576	2,778	2,683	3,705
50	99	**3,302**	6,107	5,647	5,441	4,914
50	100	10,417	9,968	10,494	**9,913**	10,630

Table 7.3: Comparison of BoostMap, FastMap, random reference objects (RRO), random line projections (RLP), and VP-trees, on the UNIPEN dataset. For different values of k, and different percentages of accuracy we show the number of exact distance computations required by each method. For each k and accuracy, we display in bold font the best result.

Time Series Dataset with Constrained DTW						
k	accuracy	BoostMap	FastMap	RRO	RLP	VP-trees
1	90	**649**	8,357	678	9,938	1,633
1	95	5,691	20,176	5,229	13,594	**3,388**
1	99	**9,072**	27,082	9,118	23,662	9,166
1	100	**9,562**	27,547	10,134	29,335	16,126
10	90	5,721	19,613	**5,401**	24,686	8,781
10	95	**8,262**	24,888	8,360	26,689	11,896
10	99	**9,448**	27,531	9,504	29,684	18,748
10	100	27,267	27,623	**20,096**	31,456	28,253
50	90	**9,043**	23,289	9,144	27,052	12,828
50	95	**9,571**	27,041	10,217	29,039	13,277
50	99	24,672	27,564	**15,944**	31,127	24,378
50	100	27,267	27,742	**25,832**	31,731	31,818

Table 7.4: Comparison of BoostMap, FastMap, random reference objects (RRO), random line projections (RLP), and VP-trees, on the time series dataset. For different values of k, and different percentages of accuracy we show the number of exact distance computations required by each method. For each k and accuracy, we display in bold font the best result.

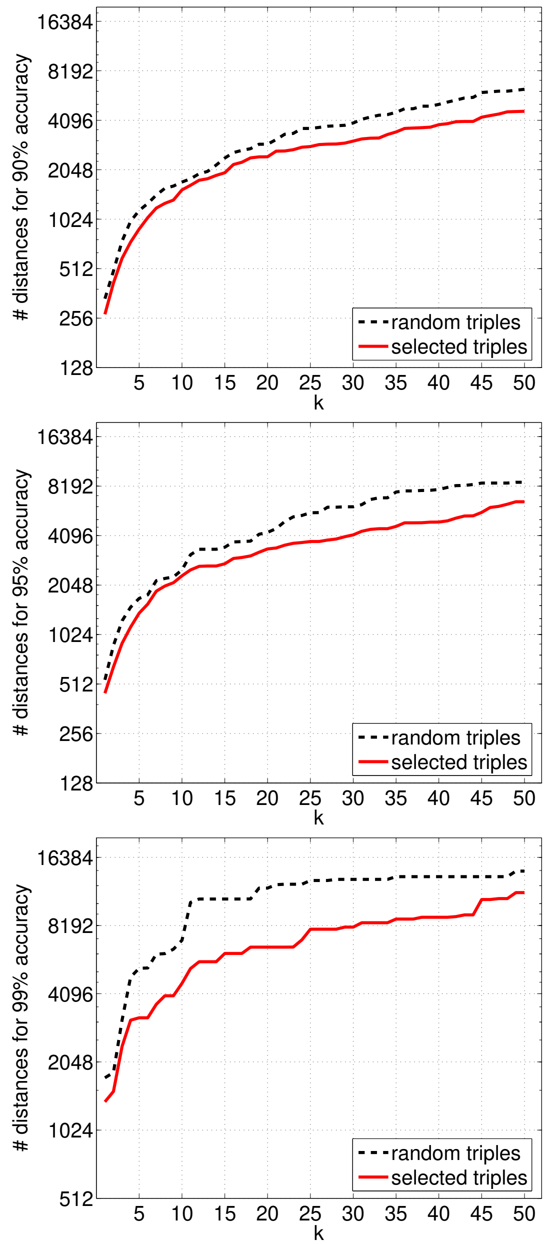

Figure 7·9: Comparing BoostMap as described in this thesis to a modified version of BoostMap, where training triples are chosen randomly. Here we show results on the ASL handshape dataset, using the chamfer distance as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 710 query objects that we use as a test set.

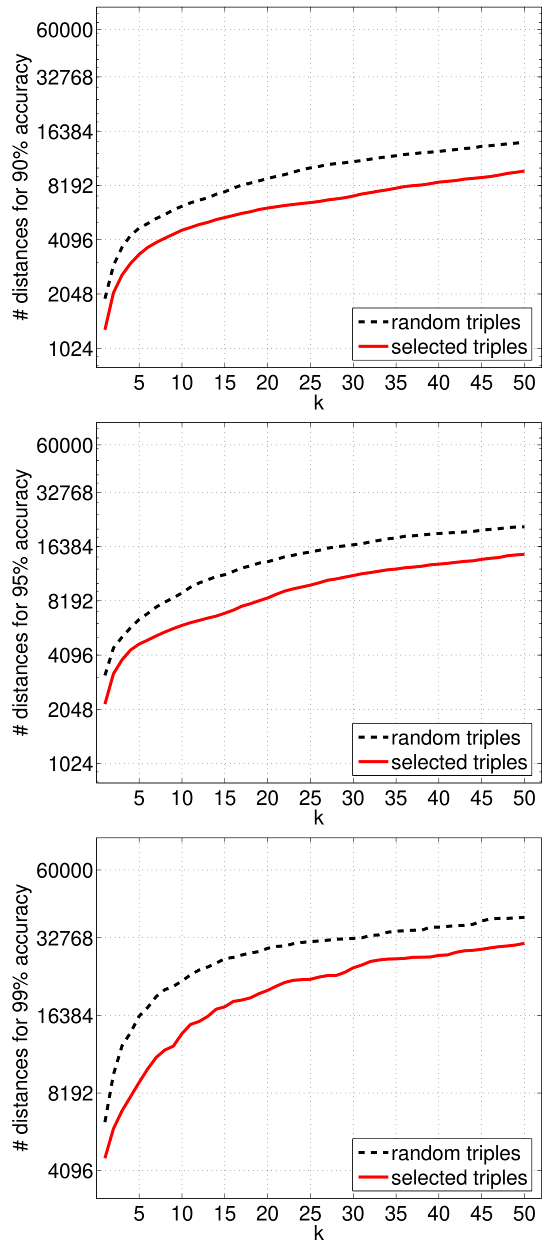

Figure 7·10: Comparing BoostMap as described in this thesis to a modified version of BoostMap, where training triples are chosen randomly. Here we show results on the MNIST dataset, using shape context matching as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 10,000 query objects that we use as a test set.

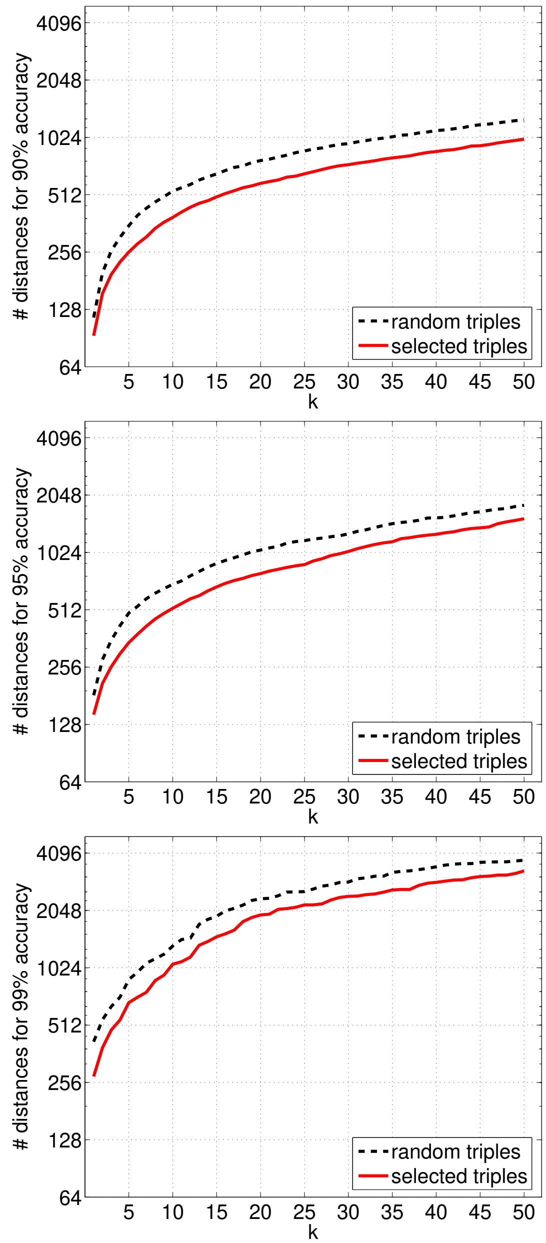

Figure 7·11: Comparing methods BoostMap, FastMap, RRO, RLP, and VP-trees, on the UNIPEN dataset, using DTW as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 10,000 query objects that we use as a test set.

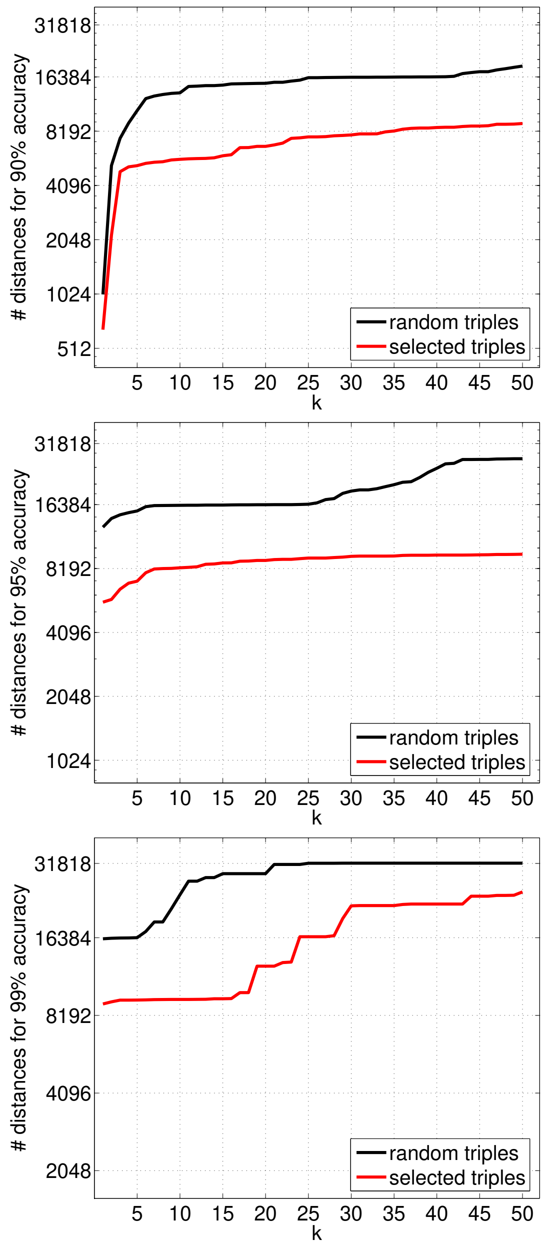

Figure 7·12: Comparing BoostMap as described in this thesis to a modified version of BoostMap, where training triples are chosen randomly. Here we show results on the time series database, using constrained DTW as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 1,000 query objects that we use as a test set.

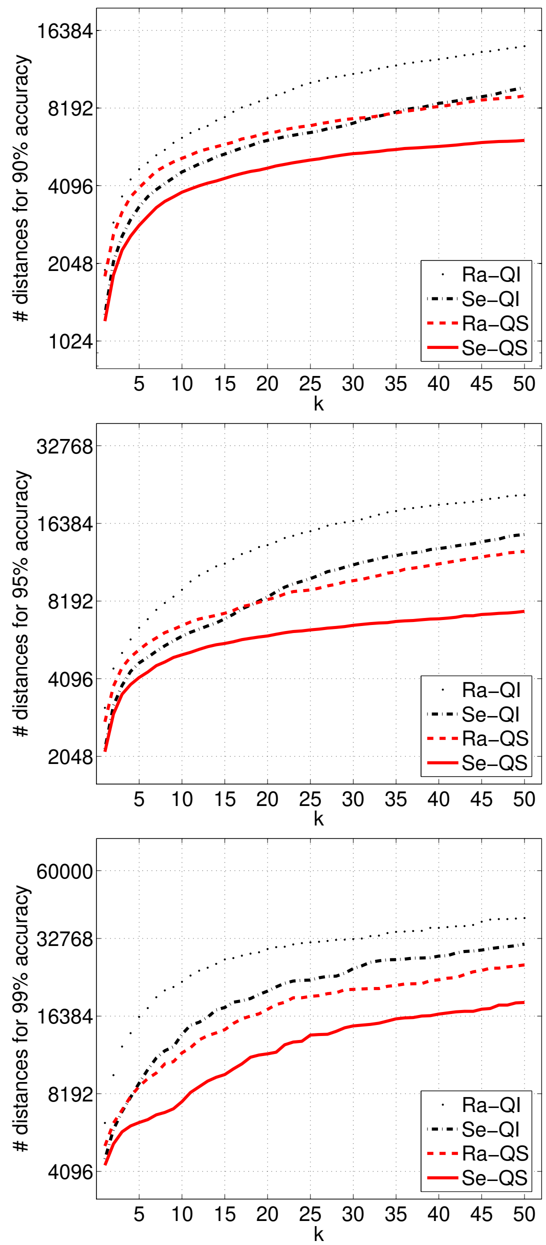

Figure 7·13: Comparing methods Ra-QI, Ra-QS, Se-QI and Se-QS on the MNIST dataset, using shape context matching as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 10,000 query objects.

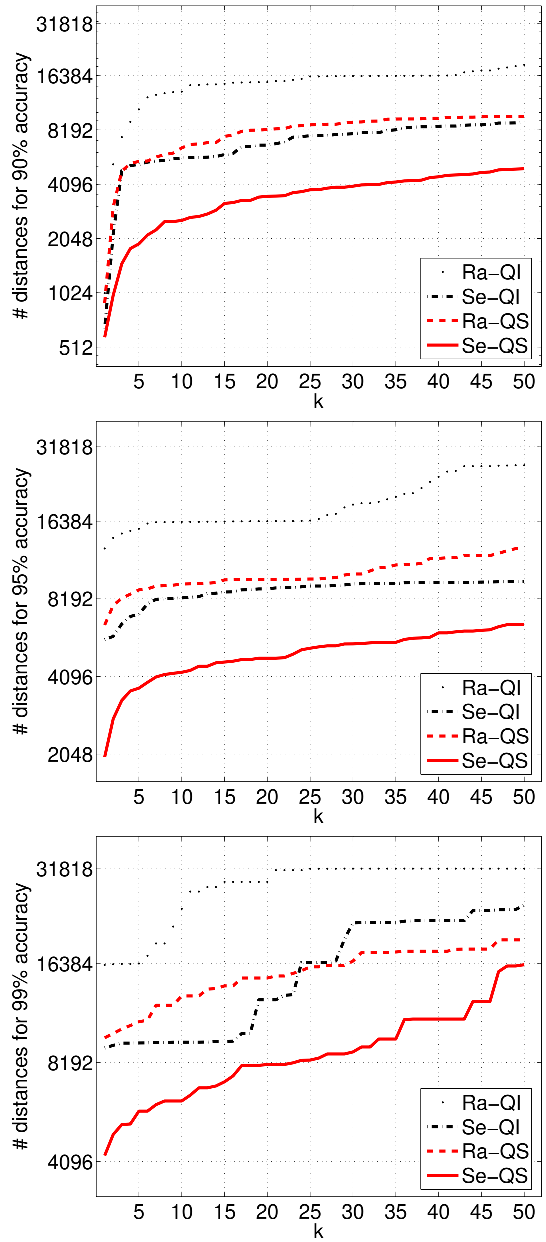

Figure 7·14: Comparing methods Ra-QI, Ra-QS, Se-QI and Se-QS on the time series dataset, using constrained Dynamic Time Warping as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 1,000 query objects.

MNIST Database with Shape Context					
k	accuracy	Ra-QI	Ra-QS	Se-QI	Se-QS
1	90	1,930	1,824	1,296	1,223
1	95	3,161	2,789	2,190	2,135
1	99	6,315	5,141	4,577	4,329
1	100	55,019	40,479	40,946	13,406
10	90	6,280	5,233	4,631	3,866
10	95	9,059	6,584	5,988	5,072
10	99	22,266	11,802	13,932	7,642
10	100	55,019	58,677	56,936	52,066
50	90	14,232	9,134	9,856	6,139
50	95	21,085	12,767	14,848	7,477
50	99	39,311	25,878	31,176	18,510
50	100	59,840	59,974	59,735	59,941

Time Series Dataset with Constrained DTW					
k	accuracy	Ra-QI	Ra-QS	Se-QI	Se-QS
1	90	1,018	898	649	580
1	95	12,851	6,484	5,691	1,995
1	99	16,236	9,743	9,072	4,269
1	100	16,426	13,922	9,562	6,965
10	90	13,364	6,521	5,721	2,582
10	95	16,270	9,346	8,262	4,251
10	99	24,052	13,070	9,448	6,260
10	100	31,818	24,730	27,267	17,627
50	90	18,821	9,757	9,043	4,997
50	95	26,985	12,821	9,571	6,504
50	99	31,818	19,357	24,672	16,265
50	100	31,818	26,748	27,267	26,883

Table 7.5: Comparison of Ra-QI, Ra-QS, Se-QI, and Se-QS on the MNIST dataset based on 10,000 query objects and the time series dataset based on 1,000 query objects. For different values of k, and different percentages of accuracy (shown in the "accuracy" column), we show the number of exact distance computations required by each embedding method. For comparison, brute force search would require 60,000 exact distance computations in the MNIST dataset and 31,818 exact distance computations in the time series dataset.

by whether it is query-sensitive or not, and whether it uses random training triples or not. To denote each method, and its relation to the other methods, we use the following abbreviations:

Ra: Training triples are chosen entirely randomly from the set of all possible triples, as in [Athitsos et al., 2004].

Se: Training triples are chosen selectively, from a restricted set of possible triples, using the method we describe in Section 4.1.3.

QI: A query-insensitive distance measure Δ is constructed, as described in Chapter 4.

QS: A query-sensitive distance measure Δ is constructed, as described in Chapter 5.

Based on these abbreviations, Se-QI denotes the method in Chapter 4, and Se-QS denotes the method described in Chapter 5. Ra-QI and Ra-QS are alternative methods, in which training triples are chosen randomly.

In Figures 7·13 and 7·14, and in Table 7.5, we compare the four different methods on k-nearest neighbor retrieval. The number of exact distance computations required by each method is shown for different values of k, from 1 to 50, and different percentages of accuracy (i.e., 90%, 95%, and 99%), in Figure 7·13 for the MNIST dataset and Figure 7·14 for the time series dataset. In Table 7.5 we show, for selected accuracy percentages and values of k, the number of exact distance computations required by each of the four methods.

The results demonstrate that query-sensitive methods clearly outperform their query-insensitive counterparts, and provide significantly better trade-offs between efficiency and accuracy. The only exception occurs in results on 100% accuracy, which are dominated by the single query giving the worst results. In some cases, query-sensitive embeddings achieve performance that is two or three times as fast for a fixed error rate. As a side note, we can also see that, with the exception of results on 100% accuracy, choosing training triples as described in this thesis leads to better performance than choosing training triples randomly.

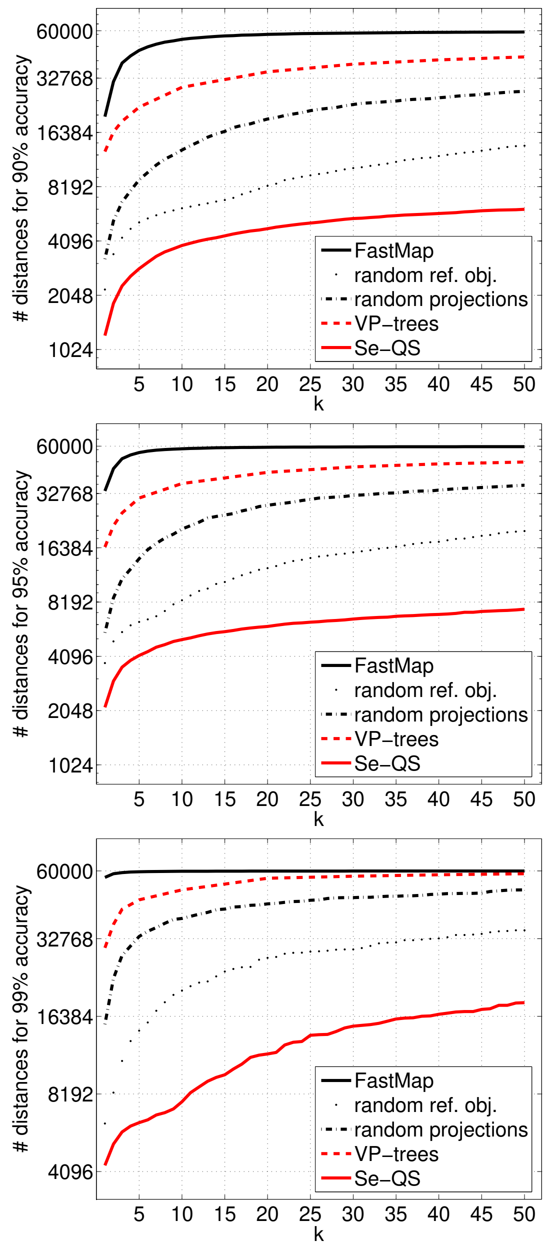

Figure 7·15: Comparing methods Se-QS, FastMap, random reference objects, random line projections, and VP-trees, on the MNIST dataset, using shape context matching as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 10,000 query objects that we use as a test set.

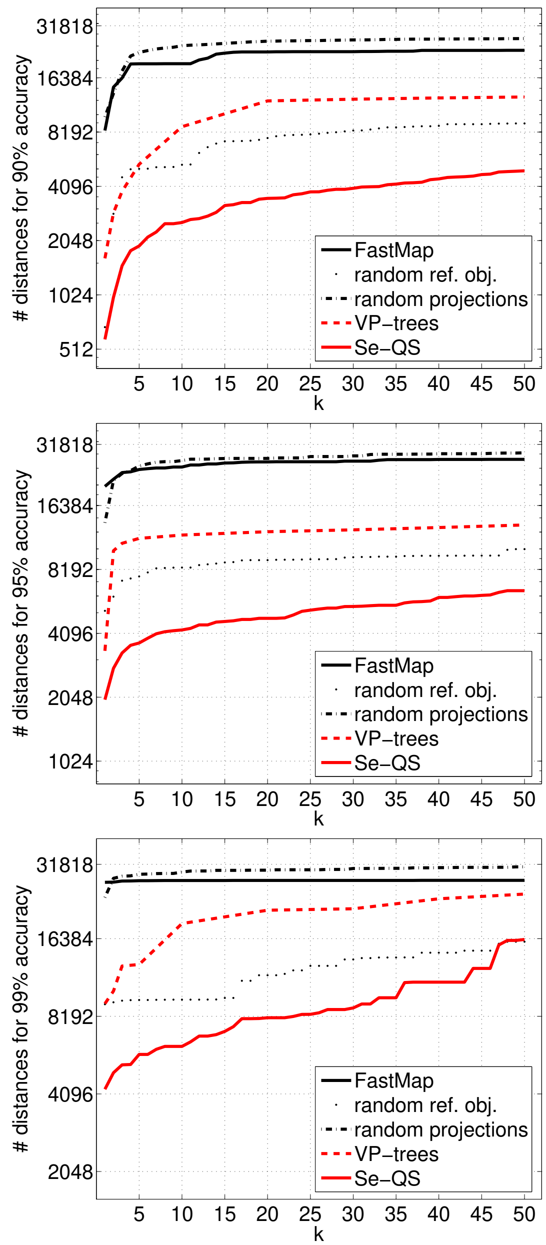

Figure 7·16: Comparing methods Se-QS, FastMap, random reference objects, random line projections, and VP-trees, on the time series database, using constrained Dynamic Time Warping as the exact distance measure. We show the number of exact distance computations needed by each method to achieve correct retrieval of all k nearest neighbors (k ranging from 1 to 50) for 90%, 95%, and 99% of the 1,000 query objects that we use as a test set.

MNIST Database with Shape Context						
k	accuracy	Se-QS	FastMap	RRO	RLP	VP-trees
1	90	**1,223**	20,059	2,203	3,251	12,842
1	95	**2,135**	33,858	3,757	5,538	16,544
1	99	**4,329**	56,619	6,290	15,264	30,201
1	100	**13,406**	59,996	37,753	46,235	57,089
10	90	**3,866**	53,852	6,219	13,094	29,178
10	95	**5,072**	58,009	8,381	20,842	37,261
10	99	**7,642**	59,800	20,617	39,258	50,681
10	100	**52,066**	60,000	59,961	59,237	59,981
50	90	**6,139**	59,102	13,828	27,679	42,996
50	95	**7,477**	59,644	20,287	36,457	49,017
50	99	**18,510**	59,980	35,319	50,664	58,525
50	100	**59,941**	60,000	59,961	60,000	60,000

Time Series Dataset with Constrained DTW						
k	accuracy	Se-QS	FastMap	RRO	RLP	VP-trees
1	90	**580**	8,357	678	9,938	1,633
1	95	**1,995**	20,176	5,229	13,594	3,388
1	99	**4,269**	27,082	9,118	23,662	9,166
1	100	**6,965**	27,547	10,134	29,335	16,126
10	90	**2,582**	19,613	5,401	24,686	8,781
10	95	**4,251**	24,888	8,360	26,689	11,896
10	99	**6,260**	27,531	9,504	29,684	18,748
10	100	**17,627**	27,623	20,096	31,456	28,253
50	90	**4,997**	23,289	9,144	27,052	12,828
50	95	**6,504**	27,041	10,217	29,039	13,277
50	99	16,265	27,564	**15,944**	31,127	24,378
50	100	26,883	27,742	**25,832**	31,731	31,818

Table 7.6: Comparison of Se-QS, FastMap, random reference objects (RRO), random line projections (RLP), and VP-trees, on the MNIST dataset based on 10,000 query objects and the time series dataset based on 1,000 query objects. For different values of k, and different percentages of accuracy (shown in the "accuracy" column), we show the number of exact distance computations required by each method. For comparison, brute force search would require 60000 exact distance computations in the MNIST dataset and 31818 exact distance computations in the time series dataset.

In order to better illustrate the competitiveness of query-sensitive embeddings with respect to existing methods, we directly compare query-sensitive embeddings to the methods used in the previous section, i.e., FastMap, RRO, RLP, and VP-trees. The results we provide are similar to the results from the previous section, except that we replace the results of the original BoostMap algorithm with the results of the query-sensitive version of BoostMap, i.e., version Se-QS. The results can be seen in Figures 7·15 and 7·16, and in Table 7.6.

In these results we see that query-sensitive embeddings clearly outperform the other methods. The only settings where Se-QS does not yield the best result are 50-nearest neighbor retrieval with 99% accuracy 100% accuracy on the time series database. Random reference objects are marginally better for those settings. In many settings our method is faster than any alternative by a factor between 1.45 and 2.7, e.g., for 90%, 95% and 99% accuracy on 1, 10, and 50-nearest neighbor retrieval on the MNIST database.

As a last experiment, we compare query-sensitive embeddings to the method described in [Vlachos et al., 2003] on the time series dataset. To make the comparison fair, in this experiment we use the exact same database and set of queries that was used in the experiments reported in [Vlachos et al., 2003]. In other words, we do *not* use the database of 31818 objects and query set of 1000 object that we use in the other experiments. Instead, we use a different splitting of the objects into database and query object, so that the database contains 32768 objects and the query set contains 50 objects.

On this modified time series dataset, we use a query-sensitive embedding and we set parameters d and p so as to retrieve the 1-nearest neighbor correctly for each of the 50 queries, i.e., so as to achieve 100% accuracy on 1-nearest neighbor retrieval. In particular, we set dimensionality $d = 150$ and filter-and-refine parameter $p = 443$. With these settings, we only need to measure 640 distances per query, and thus we obtain a speed-up factor of 51.2 compared to brute-force search. The indexing method in [Vlachos et al., 2003] is reported to achieve a speed-up of approximately a factor of 5, while retrieving correctly the true nearest neighbor for all 50 queries, and measured on the same set of 50 queries that we have used.

7.6 Experiments on Nearest Neighbor Classification

In this section we evaluate the performance of the proposed methods on the task of efficient nearest neighbor classification. Evaluation is performed on three datasets: the ASL handshape dataset, the MNIST dataset, and the UNIPEN dataset. The reason we do not use the time series dataset is simply that there are no class labels associated with the objects in that dataset. In each dataset we compare BoostMap with the RRO and RLP methods, because, as can be seen in the previous sections, in all three datasets either RRO or RLP was the best-performing retrieval method besides BoostMap. In the MNIST and UNIPEN datasets we also evaluate the cascade method. As we will see shortly, the error rate on the ASL handshape dataset is very high (over half the objects are misclassified), and therefore the ASL handshape dataset violates the fundamental assumption of cascades, i.e., the assumption that most objects are easy to classify.

7.6.1 Classification Experiments on the ASL Handshape Dataset

Classification on the ASL handshape dataset is a very challenging task. Unlike typical handshape recognition settings, which assume that each handshape is always seen in the same 3D orientation, in this dataset the orientation is arbitrary. Our goal is to identify for each query image which of the 20 handshapes it displays. Given the vast difference in appearance between different 3D orientations of the same shape, it is not surprising that exact nearest neighbor classification using brute-force search has a very high error rate of 67%.

Figure 7·17 displays the error rate attained using filter-and-refine retrieval with the BoostMap, RRO, and RLP methods. Overall, BoostMap produces better results than the other two methods. At the cost of 100 exact distance computations, BoostMap attains an error rate of 67%, which essentially equals the error rate of brute-force search. Therefore, using BoostMap we obtain a speed up factor of 800 over brute-force search, with no losses

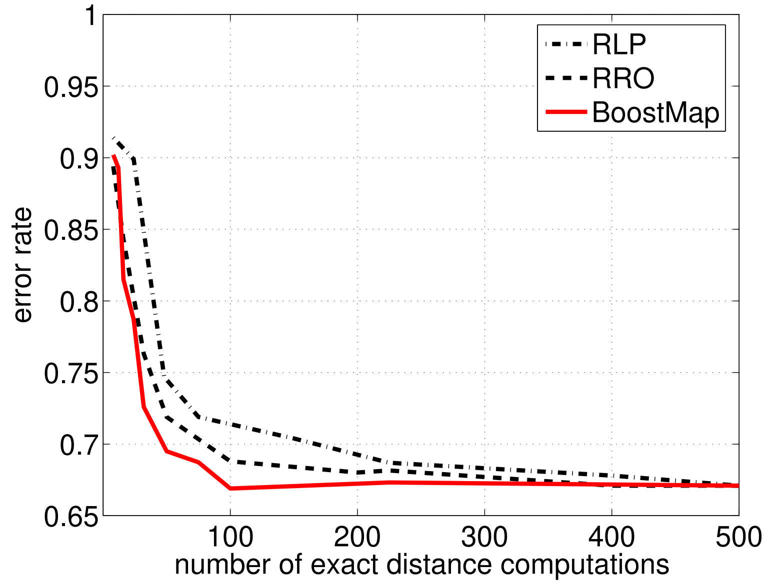

Figure 7·17: Comparing classification accuracy vs. efficiency trade-offs achieved by the BoostMap, RRO, and RLP methods on the ASL handshape dataset. For different numbers of exact distance computations we show the error rate obtained by each method.

in classification accuracy. In terms of actual running time, using BoostMap we can classify about 3.5 queries per second, whereas it takes 112 seconds on average to classify a query using brute-force search.

For a cost of 100 exact distance computations, RRO achieves an error rate of 69%, and RLP achieves an error rate of 70%. RRO and RLP achieve an error rate of 67% at 400 distances and 500 distances respectively. It is important to note that even these very simple methods achieve speed-ups of roughly a factor of 200 over brute-force search.

Overall, it is fair to say that the accuracy we obtain on the ASL handshape dataset is not at the level where it can be useful for actual applications. We should emphasize that this low accuracy is not caused by BoostMap or the other embedding methods, it is inherent in the choice of the underlying distance measure, i.e., the chamfer distance, which produces a high error rate even when using brute-force search. Reliable handshape classification of hand images displaying arbitrary 3D orientations is still an open problem

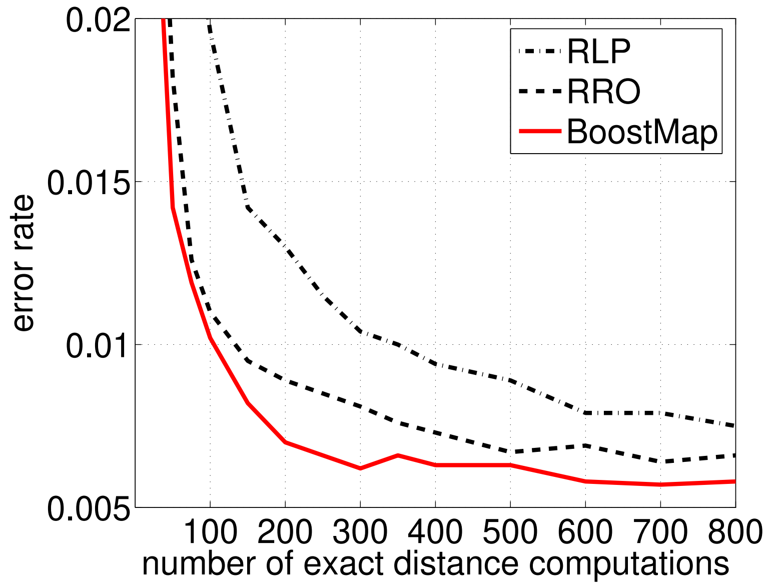

Figure 7·18: Comparing classification accuracy vs. efficiency trade-offs achieved by the BoostMap, RRO, and RLP methods on the MNIST dataset. For different numbers of exact distance computations we show the error rate obtained by each method.

in computer vision.

7.6.2 Classification Experiments on the MNIST Dataset

As a reminder, exact nearest neighbor classification using shape context matching achieves error rates of 0.63% using a database of 20,000 objects and 0.54% using the full MNIST database of 60,000 training objects, with classification time per object equal to about 22 minutes and 66 minutes respectively. Figure 7·18 displays the error rate attained using filter-and-refine retrieval with the BoostMap, RRO, and RLP methods on the MNIST dataset, using 60,000 training objects. BoostMap achieves an error rate of 0.58% at a cost of 800 exact distance computations. At the same cost of 800 exact distance computations, the RRO method obtains an error rate of 0.66% and the RLP method obtains an error rate of 0.75%. Overall, using BoostMap we achieve a speed-up factor of about 75 over brute-force search, while achieving an error rate that is only 0.04% worse than the error rate of brute-force search. Classification time per query reduces from about 66 minutes

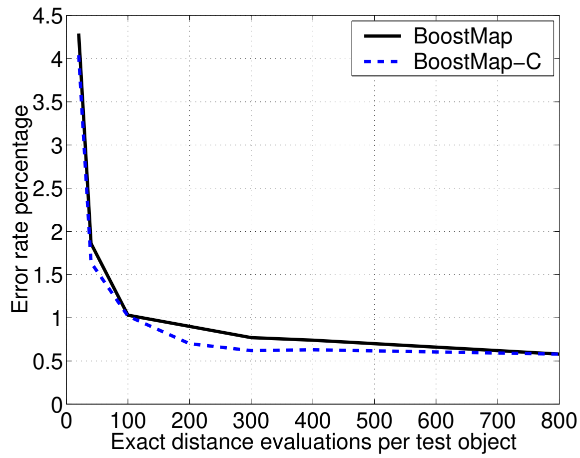

Figure 7·19: Error rates attained using BoostMap and BoostMap-C, without a cascade, vs. number of exact distance evaluations per test object, on the MNIST dataset. 60,000 training objects were used.

seconds per query using brute-force search to 53 seconds per query using BoostMap with 800 exact distance computations.

In constructing embeddings for the MNIST dataset, we have used both the original BoostMap algorithm of Chapter 4 and the modified BoostMap-C version proposed in Chapter 6, where training triples (X_i, A_i, B_i) are labeled not based on whether X_i is closer to A_i or to B_i, but based on whether X_i belongs to the same class as A_i or to the same class as B_i. Using 60,000 training objects, both methods have an error rate of 0.58%. In Fig. 7·19 we plot error rate vs. number of exact distance evaluations per test object. We see that BoostMap-C attains its peak accuracy at around 300 exact distance computations per object, but it takes BoostMap about 800 exact distance computations per object to reach the same accuracy. Overall, BoostMap-C performs slightly better than BoostMap.

We have applied Algorithm 3 to construct a cascade of classifiers, using different values of e, ranging from 0 to 4. The training and validation sets passed to the algorithm are disjoint subsets of the database, containing 20,000 and 10,000 objects respectively. The sequence $\mathbb{P}$ of filter-and-refine processes that is passed as input to Algorithm 3 is shown in Table 7.7. We have constructed that sequence by hand, i.e., we have manually picked d

Filter-and-refine processes for MNIST			
Process	Dimensions (d)	p	Threshold
P_1	10	0	50
P_2	20	0	56
P_3	40	0	51
P_4	60	0	51
P_5	80	0	50
P_6	100	0	41
P_7	100	20	41
P_8	100	40	41
P_9	100	60	17
P_{10}	100	80	20
P_{11}	100	100	20
P_{12}	100	150	24
P_{13}	100	200	11
P_{14}	100	250	4
P_{15}	100	300	5
P_{16}	100	700	NA

Filter-and-refine processes for UNIPEN			
Process	Dimensions (d)	p	Threshold
P_1	5	0	11
P_2	10	0	9
P_3	20	0	5
P_4	30	0	3
P_5	30	10	2
P_6	30	20	0
P_7	30	30	NA

Table 7.7: The sequences $\mathbb{P}$ of filter-and-refine processes that were passed as input to Algorithm 3 for the MNIST and UNIPEN datasets. We used these sequences with both BoostMap and BoostMap-C embeddings. The dimensions column specifies the dimensionality of the embedding, and p is the parameter specifying the number of distances to measure in the refine step. We also show the threshold chosen by the cascade learning algorithm, using embeddings from BoostMap-C and setting $e = 0$. Naturally, no threshold is needed for the final step in the cascade. The fact that the threshold for P_6 in the UNIPEN experiment is 0 means that the algorithm has determined that P_7 is redundant, and uses P_6 as the final process.

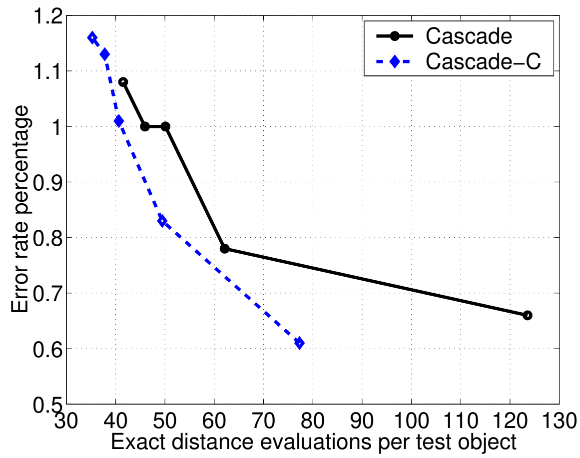

Figure 7·20: Error rates attained by cascade classifiers vs. average number of exact distance evaluations per test object, for the MNIST dataset. Cascade and Cascade-C correspond to the five cascades that were learned, using embeddings constructed with BoostMap and BoostMap-C respectively. For each of these two embedding methods, we obtained five cascades by running Algorithm 3 with parameter e set respectively to $e = 0, 1, 2, 3, 4$. 60,000 training objects were used.

and p for each process. The thresholds are learned automatically by the algorithm. Our guideline in constructing the sequence has been simply to provide an adequate number of steps, ranging from really fast and inaccurate to really slow and accurate, with the constraint that each cascade step should be able to reuse the work done at the previous steps.

Using a database of 20,000 objects, and passing $e = 0$ to Algorithm 3, using BoostMap we obtain a cascade with an error rate of 0.75%, at an average cost of measuring about 149.0 distances per test object, which translates to average classification time of 9.9 seconds per test object. Using the modified algorithm BoostMap-C, the resulting cascade yields an error rate of 0.74%, at an average cost of measuring 92.5 distances per test object, which translates to average classification time of 6.2 seconds per test object. We have evaluated the same cascades using the full MNIST database of 60,000 objects. For each cascade, the thresholds are set to the same values as in the experiments with a database of 20,000 objects. The results, for parameter e ranging from 0 to 4, are shown in Fig. 7·20. For $e = 0$ and using BoostMap we get an error rate of 0.66% at an average cost of 123 exact distance computations per test object. For $e = 0$ and using BoostMap-C we get an error rate of 0.61% at an average cost of only 77.3 distance computations per test object. This is a speed-up of almost three orders of magnitude over brute-force search, which achieves an error rate of 0.54%. As seen in Fig. 7·20, cascades using BoostMap-C achieve better tradeoffs of accuracy versus efficiency compared to cascades using BoostMap.

Note that increasing the database size from 20,000 to 60,000 objects improves both the accuracy and the efficiency of the cascade classifier. This result may seem surprising at first glance, and is in stark contrast to traditional nearest-neighbor methods, where recognition time increases as the database size increases. By taking a closer look at the results we have found that, as the database size increases and the processes P_i and thresholds t_i remain fixed, the quantity $K(Q, P_i)$ tends to increase for most query objects Q, meaning that more objects are classified at earlier steps in the cascade.

In [Zhang and Malik, 2003] a discriminative classifier is trained using shape context features, and achieves an error rate of 2.55% on the MNIST dataset while measuring only exact distances between the test object and 50 prototypes. That method is not a nearest-neighbor method, so after learning the classifier only the 50 prototypes are needed, and the rest of the training set is discarded. Overall, the cost of classifying a test object using the method in [Zhang and Malik, 2003] is the cost of evaluating 50 exact distances. Using BoostMap-C and the full training set of 60,000 objects, with parameter $e = 1$ we obtain a cascade that yields an error rate of 0.83%, while measuring on average 49.5 exact distances per test object. Also, as Fig. 7·20 shows, several cascades obtained using BoostMap and BoostMap-C achieve error rates under 1.2% at an average cost ranging from 35 to 50 exact distance evaluations per test object.

We also compare the cascade method to two additional methods that can be used for speeding up nearest neighbor classification: the well-known condensed nearest neighbor (CNN) method [Hart, 1968] and VP-trees [Yianilos, 1993]. We evaluate these methods

Method	Distances per query object	Speed-up factor	Seconds per query object	Error rate
brute force	20,000	1	1,232	0.63%
VP-trees [Yianilos, 1993]	8,594	2.3	572	0.66%
CNN [Hart, 1968]	1,060	18.9	70.6	2.40%
Zhang [Zhang and Malik, 2003]	50	400	3.3	2.55%
BoostMap	800	25	53.3	0.74%
BoostMap-C	800	25	53.3	0.72%
Cascade	149	134	9.9	0.75%
Cascade-C	93	216	6.2	0.74%

Table 7.8: Speeds and error rates achieved by different methods on the MNIST dataset, using 10,000 test objects and 20,000 database objects. We also show the number of exact shape context distance evaluations per query object for each method.

using the smaller training set of 20,000 objects. Both methods achieve significantly worse tradeoffs between accuracy and efficiency compared to our method. CNN selects 1060 out of the 20,000 training objects, speeding up classification time by approximately a factor of 20. However, the error rate using CNN increases from 0.63% to 2.40%. With VP-trees the error rate is 0.66%, but the attained speed up with respect to brute-force search is only a factor of 2.3; an average of 8594 exact distances need to be measured per test object.

Table 7.8 summarizes the results of all the different methods on the smaller database of 20,000 objects. As we can see from those results, VP-trees, BoostMap and the cascade methods are the only methods that achieve accuracy comparable to brute force search. The speed-up obtained using VP-trees is pretty minor compared to using BoostMap or using a cascade. The cascade methods, and especially Cascade-C, achieve by far the best trade-offs between accuracy and efficiency.

7.6.3 Classification Experiments on the UNIPEN Dataset

Figure 7·21 displays the error rate attained using filter-and-refine retrieval with the BoostMap, RRO, and RLP methods on the UNIPEN dataset. Exact nearest neighbor classification using brute-force search achieves an error rate of 1.90% on this dataset. BoostMap achieves an error rate of 1.95% at a cost of 75 exact distance computations, and an error rate of 1.90 at a cost of 300 distance computations. At a cost of 300 exact distance computations, the RRO method obtains an error rate of 1.99% and the RLP method obtains an error rate of 1.97%. Overall, using BoostMap we achieve a speed-up factor of about 35 over brute-force search, while achieving the same error rate, thus reducing classification time per query from 12 seconds to 0.34 seconds.

In Fig. 7·22 we compare original BoostMap algorithm of Chapter 4 and the modified BoostMap-C version of Chapter 6. We see that BoostMap-C attains its peak accuracy at around 150 exact distance computations per object. Overall, BoostMap performs slightly better than BoostMap-C for costs of up to 50 distances per query, and BoostMap-C performs slightly better than BoostMap for costs of 75 or more distances per query.

We have applied Algorithm 3 to construct a cascade of classifiers, using different values of e, ranging from 0 to 4. The training set passed to the algorithm is the entire database of 10,630 objects, and the validation set is a subset of the database consisting of 3,500 objects. No validation object is used for its own classification while running Algorithm 3. The sequence $\mathbb{P}$ of filter-and-refine processes that is passed as input to Algorithm 3 is shown in Table 7.7. As with the MNIST dataset, we have constructed that sequence by hand, making sure we provide an adequate number of steps, ranging from really fast and inaccurate to really slow and accurate, with the constraint that each cascade step should be able to reuse the work done at the previous steps.

Passing $e = 0$ to Algorithm 3, using BoostMap we obtain an error rate of 2.03%, at an average cost of measuring about 67 distances per test object. Using the modified algorithm BoostMap-C, the resulting cascade yields an error rate of 2.10%, at an average cost of measuring 35 distances per test object. Setting $e = 2$, the resulting BoostMap cascade has an error rate of 2.10%, at a cost of 30 distances per test object, which translates to a processing time of 0.034 seconds per test object. Setting $e = 1$, the resulting BoostMap-C cascade yields an error rate of 2.09%, at an average cost of measuring 32 distances per test object, which translates to average classification time of 0.036 seconds per test object. The cascade results, for both BoostMap and BoostMap-C, for parameter e ranging from 0 to 4, are shown in Fig. 7·23. Overall, the number of distance computations per query does not change as much for BoostMap-C as it changes for BoostMap, as we vary parameter e between 0 and 4. BoostMap tends to produce better or similar accuracy compared to BoostMap-C, for the same number of distance computations. This is not unexpected, given that for all the filter-and-refine processes in the cascade of Table 7.7 BoostMap actually gives better or similar results compared to BoostMap-C, as seen in Figure 7·22.

We should note that the CSDTW method [Bahlmann and Burkhardt, 2004], which has been explicitly designed for classifying time series and in particular for online handwritten character recognition, achieves an error rate of 2.90 on the UNIPEN dataset at a cost equivalent to 150 exact computations of DTW distances. The cascades obtained by BoostMap and BoostMap-C obtain a significantly better error rate of about 2.10%, while the average classification time is about 5 times faster compared to CSDTW, since we only need to measure 30 or 32 exact distances per query on average. The worst case cost for classifying a query is 75 exact distances, which is still 2.5 times faster than the cost of CSDTW. The advantage of CSDTW over our method is that CSDTW requires significantly less memory; in our method, we need to store in memory the embeddings of all database objects, and that takes up about 7MB for a 100-dimensional embedding.

Table 7.9 provides a summary of classification results obtained using different methods, including VP-trees, different embedding methods, CSDTW and cascades. We see that BoostMap and BoostMap-C outperform alternative embedding methods and VP-trees, and cascades provide even better trade-offs between accuracy and efficiency.

7.7 Summary of Experimental Results

For the purposes of nearest neighbor retrieval, the query-insensitive version of BoostMap significantly outperforms all methods that we have evaluated and that are not based on BoostMap, in three of the four datasets we have experimented with. In the time series dataset, RRO performs slightly better. The query-sensitive version of BoostMap, on the

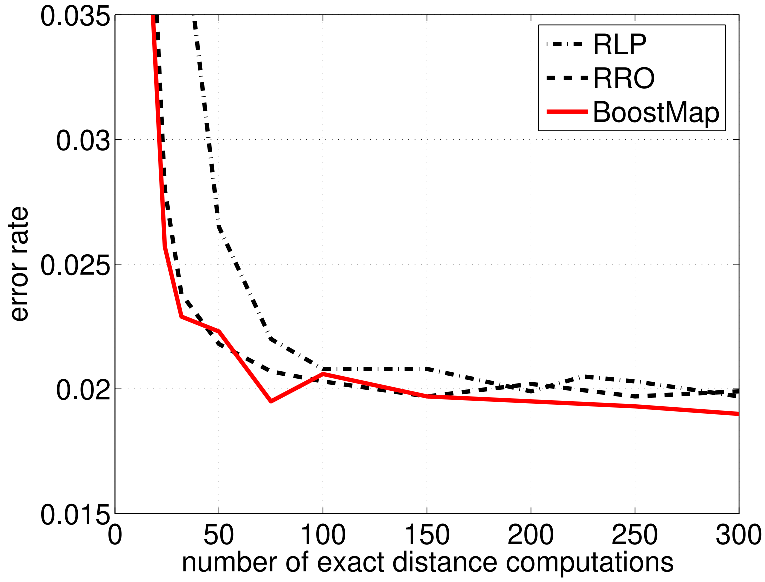

Figure 7·21: Comparing classification accuracy vs. efficiency trade-offs achieved by the BoostMap, RRO, and RLP methods on the UNIPEN dataset. For different numbers of exact distance computations we show the error rate obtained by each method.

other hand, significantly outperforms RRO and other methods on both datasets where we have evaluated query-sensitive embeddings, i.e., both on the time series dataset and the MNIST dataset. Also, in all comparisons we have performed, the query-sensitive version of BoostMap performs better than the query-insensitive version. Overall, the retrieval results obtained using the methods proposed in this thesis are better than the results obtained using any other method we have implemented, in all four datasets.

It is interesting to note that the two relatively well known retrieval methods we have evaluated, i.e., VP-trees and FastMap, actually perform significantly worse than RRO and RLP. We are not aware of any existing work actually using RLP, and our experiments indicate that RLP may be a competitive alternative to FastMap, which uses the same type of 1D embeddings as RLP, but combines those embeddings in a different way than RLP.

For the purposes of classification, we see that BoostMap embeddings perform better than alternative retrieval methods, and offer significant speed-ups over brute-force search,

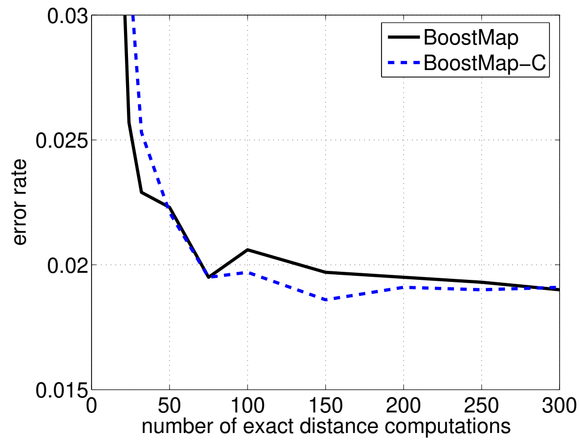

Figure 7·22: Error rates attained using BoostMap and BoostMap-C, without a cascade, vs. number of exact distance evaluations per test object, on the UNIPEN dataset.

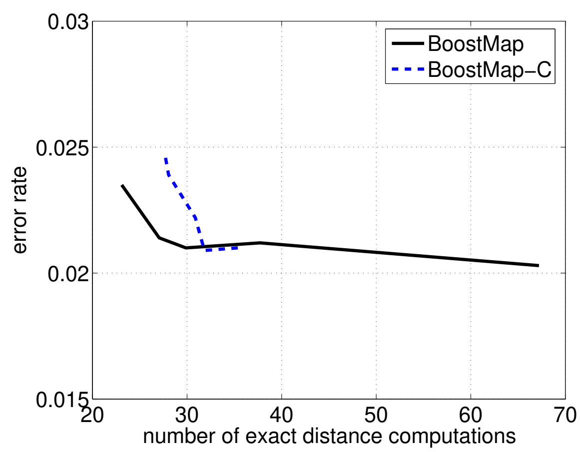

Figure 7·23: Error rates attained by cascade classifiers vs. average number of exact distance evaluations per test object, for the UNIPEN dataset. Cascade and Cascade-C correspond to the five cascades that were learned, using embeddings constructed with BoostMap and BoostMap-C respectively. For each of these two embedding methods, we obtained five cascades by running Algorithm 3 with parameter e set respectively to $e = 0, 1, 2, 3, 4$.

Method	Distances per query object	Speed-up factor	Seconds per query object	Error rate
brute force	10,630	1.0	11.94	1.90%
VP-trees [Yianilos, 1993]	1,899	5.6	2.13	1.90%
CSDTW	150	70.9	0.17	2.90%
RRO	150	70.9	0.17	1.97%
RLP	150	70.9	0.17	2.08%
BoostMap	150	70.9	0.17	1.97%
BoostMap-C	150	70.9	0.17	1.86%
RRO	32	332	0.036	2.38%
RLP	32	332	0.036	3.92%
BoostMap	32	332	0.036	2.29%
BoostMap-C	32	332	0.036	2.53%
Cascade	30	354	0.034	2.10%
Cascade-C	32	332	0.036	2.09%

Table 7.9: Speeds and error rates achieved by different methods on the UNIPEN dataset. To make it easier to compare different methods, for some methods we show multiple results, which correspond to different numbers of exact distance evaluations per query.

with minimal or no loss in classification accuracy. Cascades of approximate nearest neighbor classifiers offer even better tradeoffs between accuracy and efficiency, compared to using a single filter-and-refine process for nearest neighbor classification. In the MNIST and UNIPEN datasets, cascades also outperform in accuracy and processing time two domain-specific methods, namely [Zhang and Malik, 2003] and [Bahlmann and Burkhardt, 2004] that have been previously evaluated on those datasets. The most impressive result of cascades is obtained on the MNIST database, where classification time using the full database of 60,000 objects is reduced from 66 minutes using brute-force search to about 5 seconds using a cascade, while the error rate changes minimally from 0.54% using brute-force search to 0.61% using a cascade. This error rate difference means that the cascade misclassifies seven more objects, out of 10,000, compared to brute-force search, while the cascade is on average about 775 times faster than brute-force search.

Chapter 8

Discussion and Conclusions

In this final chapter of the thesis we summarize the main lessons learned from the work we have described, and we point out open questions and interesting directions for future research.

8.1 Discussion of Contributions

The focus of this thesis has been on designing methods for efficient nearest neighbor retrieval and classification in spaces with computationally expensive distance measures. The key insight is the concept that constructing efficient approximations of computationally expensive distance measures is a machine learning problem. Prior literature has mainly tackled that problem as a geometric problem, where different geometric assumptions like metricity of distance measure (e.g., for Bourgain embeddings) or Euclidean structure (e.g., for Fastmap) have been used to construct embeddings and demonstrate that, when the assumptions hold, the embeddings capture a certain amount of information above the original space. The main two limitations of existing methods have been that:

- Embedding construction is often seen not as an optimization problem, but as a problem of establishing some worst-case bounds (e.g., Bourgain embeddings), or even as a matter of making random choices (e.g., Lipschitz embeddings with random reference objects). Thus, the resulting embeddings are often suboptimal, as our experiments have demonstrated.

- Oftentimes the spaces that we need to index violate the assumptions that existing embedding methods are based on. For example, in all four datasets used in our experiments the original distance measure is non-metric, whereas existing embedding methods assume that the original distance measure is metric (e.g., in Bourgain embeddings, SparseMap) or Euclidean (e.g., in FastMap).

In this thesis, we have proposed novel methods for addressing the above limitations and for improving the overall trade-offs between accuracy and efficiency that can be obtained using embeddings. The main contributions have been the following:

Framing Embedding Construction As a Machine Learning Problem

The foundation of the methods described in this thesis has been the correspondence that we established in Chapter 4 between embeddings and classifiers: the association of every embedding with a corresponding classifier, and the proof that any linear combination of such embedding-based classifiers naturally corresponds to an embedding and a distance measure. By treating embeddings as classifiers and embedding construction as a problem of learning how to estimate the proximity order of triples of objects, we simultaneously address both the issue of how to optimize an embedding and the issue of avoiding assumptions about the geometry of the original space. First, the embedding construction algorithm directly maximizes the amount of nearest neighbor structure preserved by the embedding. Second, the optimization criterion that we use does not rely on any geometric assumptions and is equally principled for Euclidean, metric, and non-metric spaces. Overall, by moving away from geometric considerations and treating embedding construction as a machine learning problem, we provide a promising step forward towards developing a general indexing framework for non-metric spaces whose geometric structure is either poorly understood or "inconvenient" from the point of view of existing indexing methods.

At the same time, a side advantage of treating embeddings as classifiers is that the embedding construction algorithm optimizes not only the embedding itself, by picking an appropriate set of 1D embeddings, but also the distance measure to be used in the target space of the embedding. Existing methods typically take for granted that the target distance measure is a simple, unweighted Euclidean metric. From our formulation it follows naturally that the choice of distance measure makes a difference, and the algorithm constructs a distance measure that optimizes embedding performance.

Query-Sensitive Embeddings

Query-sensitive embeddings take the concept of distance measure optimization one step further, by producing a distance measure that automatically adapts to each query. The ability to adapt the distance measure greatly enhances the modeling power of embeddings, and allows embeddings to capture a larger amount of the structure of the original space, without increasing the computational complexity of online nearest neighbor retrieval. Query-sensitive embeddings combine the efficiency of measuring distances in a vector space with the ability to capture non-metric structure that exists in the original space, such as violations of the triangle inequality or asymmetric distances. Because of its ability to capture non-metric structure, the query-sensitive version of BoostMap overcomes the only geometric limitation of the original BoostMap algorithm, i.e., the constraint that the target space of the embedding has to be an L_1 metric space. It will be interesting to explore possible applications of query-sensitivity in other contexts where we need a meaningful distance measure between high-dimensional vectors, for example for clustering or data mining applications.

Cascades of Approximate Nearest Neighbor Classifiers

The message from the third main contribution in this thesis, i.e., the cascade of approximate nearest neighbor classifiers, is that the task of nearest neighbor classification, although naturally and intricately related to the task of nearest neighbor retrieval, is fundamentally a different task than retrieval. Intuitively, as the size of the database increases, retrieving the true nearest neighbors becomes increasingly harder, because the number of wrong answers that need to be eliminated becomes larger. On the contrary, accurate classification of a query object becomes easier, because a larger number of training objects provides more accurate information about the distribution of different classes.

The proposed cascade method takes advantage of the fundamental differences between the tasks of retrieval and classification, and essentially decouples classification from retrieval, at least for a large majority of test objects that we can confidently classify without needing an accurate estimate of the true nearest neighbors. An intriguing result in our experiments with the MNIST dataset has been that cascades of approximate nearest neighbor classifiers improve in both classification accuracy and classification efficiency as we include more training objects. This empirical behavior is in agreement with our intuition that, as the database becomes larger, while retrieval becomes harder, classification becomes easier. At the same time, an interesting direction for future work is to theoretically analyze this empirical behavior and identify the conditions under which we can expect this behavior to occur.

8.2 Broader Issues and Future Work

Naturally, a large number of open problems remain to be addressed in the field of efficient nearest neighbor retrieval and classification. Here we take a brief look at some issues that have not been addressed in this thesis and that point to interesting directions for future work.

Duality Between Indexing and Classification

A general idea that has inspired to a large extent the work described in this thesis is that indexing and classification can oftentimes be seen as equivalent problems. Finding the nearest neighbors of the query can be seen as a classification problem, where we need to determine, for each database object, whether it is a nearest neighbor of the query or not. At the same time, classifying an object, regardless of the classification method we use, can be seen as a nearest neighbor problem where the database is a set of classes and we want to find the class that is nearest to the object.

Despite the fundamental similarities between the problems of indexing and classification, these problems have typically been studied by different communities, i.e., the database community and the machine learning community, without significant synergy between the communities. The work described in this thesis can be seen as a step towards addressing one half of the equivalence between indexing and classification: for a particular indexing method, i.e., embedding-based filter-and-refine retrieval, we have proposed a mathematical framework for reducing the indexing optimization problem into a classification optimization problem. By drawing from the machine learning arsenal, that has been largely unexploited in existing database indexing methods, we have been able to significantly improve indexing performance, compared to existing state-of-the-art methods.

From the broader perspective of unifying the problems of indexing and classification, it becomes clear that the work in this thesis has only covered a small part of that unification. An obvious question that needs to be explored is whether it is feasible and beneficial to apply machine learning methods for optimizing different types of indexing structures, such as various tree-based structures (e.g., KD-trees, VP-trees) and hashing structures. Furthermore, it is important to explore whether by shifting the emphasis from geometric considerations to machine learning considerations we can extend some of these indexing methods, to either make them principled in spaces where currently those methods are heuristic (e.g., VP-trees in non-metric spaces), or to make them applicable in spaces where currently they cannot be applied (e.g., LSH in arbitrary non-vector spaces).

The other direction of the relation between indexing and classification also needs to be explored, and this is an issue that has not been addressed at all in this thesis. Treating classification as an indexing problem can be advantageous in domains with a large number of classes, such as articulated pose estimation, or biometrics-based identification of a large number of individuals. The majority of machine learning methods scale at least linearly with the number of classes. As a result, the problem of recognizing very large numbers of classes has received very little attention, not because of the lack of applications, but because of the computational complexity of existing solutions. Treating classification as an indexing problem, where we want to accurately and efficiently retrieve for each query the “nearest” class in the space of all classes, may offer significant insights and allow the use or adaptation of existing database techniques in order to achieve classification time that is sublinear to the number of classes. A promising step towards that direction is the Parameter Sensitive Hashing method described in [Shakhnarovich et al., 2003].

Applications to Different Domains

The work described in this thesis has been primarily applied to image databases. At the same time, the formulation of the proposed methods is quite general, and can be applied to problems in many areas of computer science beyond computer vision and pattern recognition. Two domains that we are particularly interested in exploring are peer-to-peer networks and protein databases. It will be interesting to see if applying our methods to these domains can produce results that are competitive, given the availability of several domain-specific methods for these problems.

An interesting question to explore is whether by using domain-specific information we can improve embedding quality and thus achieve better performance in specific applications. The formulation presented in this thesis defines weak classifiers based only on distances between objects. The advantage of this formulation is that it is quite general and can be applied to any arbitrary space, as long as there is a distance measure defined on that space. At the same time, if our goal is to maximize performance in a particular domain, like networks or protein databases, it may be worth defining domain-specific weak classifiers, that can capture additional information compared to classifiers based on distances. In [Alon et al., 2005b] we describe such a domain-specific method, for spaces where computing distances between objects requires establishing correspondences between object features. In that work, we use correspondences between objects to define a richer family of weak classifiers. Applying the BoostMap algorithm on that richer family leads to embeddings that capture more of the structure of the original space and yield better retrieval and classification performance.

The Unsegmented Nearest Neighbor Problem

Going back to computer vision and pattern recognition applications, existing nearest neighbor methods, including the methods proposed in this thesis, have a significant limitation: they assume that the query is an object that is not fundamentally different from database objects. A very common situation where this assumption is violated is the case where:

- the query object is an unsegmented image or video sequence, that contains a pattern of interest for which we want to find the nearest neighbors.
- the database contains segmented patterns.

For example, the query can be an unsegmented image that contains a face, and the database may contain segmented faces. As another example, the query can be an unsegmented image containing a hand, and the database can contain, as in the experiments, segmented hand images. As a third example, the query can be a video sequence of a gesture, in which we do know neither when the gesture begins and ends, nor where the gesturing hands are located in each frame, and the database may contain gestures with additional annotation that specifies gesture time boundaries and 2D hand locations [Alon et al., 2005a, Alon et al., 2005c].

We use the term "unsegmented nearest neighbor problem" to refer to this problem of finding the nearest neighbors of an unsgemented pattern that is only a part of the query object. The dominant paradigm for solving this problem currently involves decoupling segmentation from retrieval and classification, so that first we do our best at segmenting the pattern of interest and then we use the segmented pattern as a query. Clearly, this paradigm inherits the typical limitations of bottom-up methods: mistakes in the low-level task of segmentation cannot be corrected, and high-level information available in the database cannot be used to improve performance in the low-level task. Developing methods that can directly solve the unsegmented nearest neighbor problem would significantly widen the applicability of nearest neighbor methods to computer vision problems, and at the same time would offer a new family of methods for integrating bottom-up and top-down information and treating pattern segmentation and pattern recognition in a unified framework.

8.3 Conclusions

The main topic of this thesis has been efficient nearest neighbor retrieval and classification in spaces with computationally expensive distance measures. We have shown that the problem of designing efficient embedding-based approximations of such measures can be formulated as a machine learning problem. We have demonstrated that this novel formulation has the theoretical advantages of being principled in arbitrary spaces and of explicitly maximizing the amount of nearest neighbor structure captured by the embedding. Taking advantage of the machine learning formulation, we have extended our method to produce query-sensitive embeddings. Query-sensitive embeddings are a novel type of embeddings, and map objects to a vector space with a query-sensitive weighted L_1 measure, where the weights automatically adjust to each query object. We have shown theoretically and experimentally the additional modeling power and improved performance gained by using query-sensitive embeddings. Overall, the BoostMap method is based on machine learning, in contrast to prior indexing methods that are primarily based on geometric considerations. By formulating indexing as a machine learning problems we obtain a method that is free from geometric assumptions, is equally principled in metric and non-metric spaces, and can capture non-metric structure.

With respect to nearest neighbor classification, we have proposed a method for significantly improving efficiency using a cascade of approximate nearest neighbor classifiers. Our method decouples the classification problem from the retrieval problem, and identifies cases where we have sufficient information to provide an accurate classification even though we do not have sufficient information to provide accurate retrieval results. By decoupling classification from retrieval our method achieves significant improvement in efficiency compared to more traditional retrieve-and-then-classify methods.

The key message of this thesis is that indexing in spaces with computationally expensive distance measures can be framed as a machine learning problem. Existing indexing methods mainly rely on geometric properties. Shifting the focus from geometry to machine learning has allowed us to develop principled methods that are applicable in arbitrary spaces, regardless of their geometry. Non-Euclidean and non-metric computationally expensive distance measures are frequently utilized in computer vision, and we have demonstrated that the proposed methods can be successfully applied in a variety of domains to achieve state-of-the-art accuracy and efficiency at the same time.

References

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

[Alon et al., 2005a] Alon, J., Athitsos, V., and Sclaroff, S. (2005a). Accurate and efficient gesture spotting via pruning and subgesture reasoning. In *IEEE Workshop on Human Computer Interaction*, pages 189–198.

[Alon et al., 2005b] Alon, J., Athitsos, V., and Sclaroff, S. (2005b). Online and offline character recognition using alignment to prototypes. In *International Conference on Document Analysis and Recognition*, pages 839–843.

[Alon et al., 2005c] Alon, J., Athitsos, V., Yuan, Q., and Sclaroff, S. (2005c). Simultaneous localization and recognition of dynamic hand gestures. In *IEEE Motion Workshop*, pages 254–260.

[Athitsos et al., 2005a] Athitsos, V., Alon, J., and Sclaroff, S. (2005a). Efficient nearest neighbor classification using a cascade of approximate similarity measures. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 486–493.

[Athitsos et al., 2004] Athitsos, V., Alon, J., Sclaroff, S., and Kollios, G. (2004). BoostMap: A method for efficient approximate similarity rankings. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 268–275.

[Athitsos et al., 2005b] Athitsos, V., Hadjieleftheriou, M., Kollios, G., and Sclaroff, S. (2005b). Query-sensitive embeddings. In *ACM International Conference on Management of Data (SIGMOD)*, pages 706–717.

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

[Athitsos and Sclaroff, 2005] Athitsos, V. and Sclaroff, S. (2005). Boosting nearest neighbor classifiers for multiclass recognition. In *IEEE Workshop on Learning in Computer Vision and Pattern Recognition*.

[Bahlmann and Burkhardt, 2004] Bahlmann, C. and Burkhardt, H. (2004). The writer independent online handwriting recognition system frog on hand and cluster generative statistical dynamic time warping. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(3):299–310.

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

[Belongie et al., 2002] Belongie, S., Malik, J., and Puzicha, J. (2002). Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4):509–522.

[Boeckmann et al., 2003] Boeckmann, B., Bairoch, A., Apweiler, R., Blatter, M. C., Estreicher, A., Gasteiger, E., Martin, M. J., Michoud, K., O'Donovan, C., Phan, I., Pilbout, S., and Schneider, M. (2003). The swiss-prot protein knowledgebase and its supplement trembl in 2003. *Nucleic Acids Research*, 31(1):365–370.

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

[Borst et al., 2000] Borst, F., Thurler, G., Breant, C., Lehner-Godinho, B., Calmy, A., and Meier, C. (2000). Finding similar cases within a hospital information system. *Studies in health technology and informatics*, 77:875–879.

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

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

[Breu et al., 1995] Breu, H., Gil, J., Kirkpatrick, D., and Werman, M. (1995). Linear-time euclidean distance transform algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(5):529–533.

[Canny, 1986] Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698.

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

[Ciaccia et al., 1997] Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *International Conference on Very Large Data Bases*, pages 426–435.

[Cover and Thomas, 1991] Cover, T. M. and Thomas, J. A. (1991). *Elements of information theory*. Wiley-Interscience, New York, NY, USA.

[Curious Labs, 2002] Curious Labs (2002). *Poser 5 Reference Manual*. Curious Labs, Santa Cruz, CA.

[Devi and Murty, 2002] Devi, V. S. and Murty, M. N. (2002). An incremental prototype set building technique. *Pattern Recognition*, 35(2):505–513.

[Domeniconi et al., 2002] Domeniconi, C., Peng, J., and Gunopulos, D. (2002). Locally adaptive metric nearest-neighbor classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(9):1281–1285.

[Duda et al., 2001] Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification*. Wiley-Interscience.

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

[Everitt et al., 2001] Everitt, B. S., Landau, S., and Leese, M. (2001). *Cluster Analysis*. Arnold Publishers, London, England, 4th edition.

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

[Friedman et al., 2000] Friedman, J., Hastie, T., and Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28(2):337–374.

[Frome et al., 2004] Frome, A., Huber, D., Kolluri, R., Bulow, T., and Malik, J. (2004). Recognizing objects in range data using regional point descriptors. In *European Conference on Computer Vision*, volume 3, pages 224–237.

[Gates, 1972] Gates, G. W. (1972). The reduced nearest neighbor rule. *IEEE Transactions on Information Theory*, 18(3):431–433.

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

[Grauman and Darrell, 2005] Grauman, K. and Darrell, T. (2005). The pyramid match kernel: Discriminative classification with sets of image features. In *IEEE International Conference on Computer Vision*, pages 1458–1465.

[Grauman and Darrell, 2004] Grauman, K. and Darrell, T. J. (2004). Fast contour matching using approximate earth mover's distance. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages I: 220–227.

[Guyon et al., 1994] Guyon, I., Schomaker, L., and Plamondon, R. (1994). Unipen project of on-line data exchange and recognizer benchmarks. In *12th International Conference on Pattern Recognition*, pages 29–33.

[Hart, 1968] Hart, P. E. (1968). The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, 14(3):515–516.

[Hastie and Tibshirani, 1996] Hastie, T. and Tibshirani, R. (1996). Discriminant adaptive nearest-neighbor classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(6):607–616.

[Hildrum et al., 2002] Hildrum, K., Kubiatowicz, J. D., Rao, S., and Zhao, B. Y. (2002). Distributed object location in a dynamic network. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 41–52.

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

[Hjaltason and Samet, 2003b] Hjaltason, G. R. and Samet, H. (2003b). Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580.

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

[Huttenlocher et al., 1993] Huttenlocher, D., Klanderman, D., and Rucklige, A. (1993). Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863.

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

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

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

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

[Koudas et al., 2004] Koudas, N., Ooi, B. C., Shen, H. T., and Tung, A. K. H. (2004). LDC: Enabling search by partial distance in a hyper-dimensional space. In *IEEE International Conference on Data Engineearing*, pages 6–17.

[Kruskall and Liberman, 1983] Kruskall, J. B. and Liberman, M. (1983). The symmetric time warping algorithm: From continuous to discrete. In *Time Warps.* Addison-Wesley.

[Kuhn, 1955] Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–87.

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

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics*, 10(8):707–710.

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

[Li and Zhang, 2004] Li, S. Z. and Zhang, Z. Q. (2004). Floatboost learning and statistical face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1112–1123.

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

[Micó and Vidal, 1994] Micó, L. and Vidal, E. (1994). A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17.

[Mori et al., 2001] Mori, G., Belongie, S., and Malik, J. (2001). Shape contexts enable efficient retrieval of similar shapes. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages 723–730.

[Neidle et al., 2001] Neidle, C., Sclaroff, S., and Athitsos, V. (2001). SignStream: A tool for linguistic and computer vision research on visual-gestural language data. *Behavior Research Methods, Instruments and Computers*, 33(3):311–320.

[Ong and Bowden, 2004] Ong, E. J. and Bowden, R. (2004). A boosted classifier tree for hand shape detection. In *Face and Gesture Recognition*, pages 889–894.

[Paredes and Vidal, 2000] Paredes, R. and Vidal, E. (2000). A class-dependent weighted dissimilarity measure for nearest neighbor classification problems. *Pattern Recognition Letters*, 21(12):1027–1036.

[Phillips et al., 2003] Phillips, P., Grother, P., Micheals, R., Blackburn, D., Tabassi, E., and Bone, M. (2003). Face recognition vendor test 2002. Technical Report NIST IR 6965, NIST, Gaithersburg, Maryland, USA.

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

[Rubner et al., 1998] Rubner, Y., Tomasi, C., and Guibas, L. J. (1998). A metric for distributions with applications to image databases. In *IEEE International Conference on Computer Vision*, pages 59–66.

[Sahinalp et al., 2003] Sahinalp, S. C., Tasan, M., Macker, J., and Özsoyoglu, Z. M. (2003). Distance based indexing for string proximity search. In *IEEE International Conference on Data Engineering*, pages 125–136.

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

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

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

[Shen et al., 2005] Shen, H. T., Ooi, B. C., and Zhou, X. (2005). Towards effective indexing for very large video sequence database. In *ACM International Conference on Management of Data (SIGMOD)*, pages 730–741.

[Smith and Waterman, 1981] Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197.

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

[Thayananthan et al., 2003] Thayananthan, A., Stenger, B., Torr, P. H. S., and Cipolla, R. (2003). Shape context and chamfer matching in cluttered scenes. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 127–133.

[Tieu and Viola, 2000] Tieu, K. and Viola, P. (2000). Boosting image retrieval. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 228–235.

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

[Uhlman, 1991] Uhlman, J. (1991). Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179.

[Vapnik, 1995] Vapnik, V. (1995). *The nature of statistical learning theory.* Springer-Verlag New York, Inc.

[Vidal, 1994] Vidal, E. (1994). New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (AESA). *Pattern Recognition Letters*, 15(1):1–7.

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

[Vlachos et al., 2002] Vlachos, M., Domeniconi, C., Gunopulos, D., Kollios, G., and Koudas, N. (2002). Non-linear dimensionality reduction techniques for classification and visualization. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 645–651.

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

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

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

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

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

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

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

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

[Yuan et al., 2005] Yuan, Q., Sclaroff, S., and Athitsos, V. (2005). Automatic 2D hand tracking in video sequences. In *IEEE Workshop on Applications of Computer Vision*, pages 250–256.

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

[Zhang and Malik, 2003] Zhang, H. and Malik, J. (2003). Learning a discriminative classifier using shape context distances. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages 242–247.