UNIVERSITY OF CALIFORNIA
RIVERSIDE

Learning from Time Series in the Resource-Limited Situations

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Lexiang Ye

June 2010

Dissertation Committee:
Dr. Eamonn Keogh, Chairperson
Dr. Christian Shelton
Dr. Stefano Lonardi

Acknowledgements

My greatest gratitude goes to my advisor, Dr. Eamonn Keogh. I will never forget the summer morning three years ago, when I walked into his office and was given a brief introduction to his research on time series data mining. That day was the beginning of an exciting journey in this field. Since then, I have been awed by Eamonn's passion in his research. He is always the most hard-working person amongst the lab. Everyday he tries to improve and innovate through his work. He has been consistently very positive, supportive, and inspirational throughout my work and life here. Eamonn, thank you for guiding me to become a better researcher and a better person. You are the greatest advisor I could have ever met.

I would also like thank, my admirable committee: Dr. Christian Shelton and Dr. Stefano Lonardi. I have taken three courses with Dr. Shelton at the University of California, Riverside. I have really enjoyed his enthusiastic teaching and would like to thank him for providing me with a solid knowledge in AI and machine learning. I greatly appreciate him for always being willing and ready to spend his time discussing research ideas, and for always providing patient guidance and helpful advice. I would also like to thank Dr. Stefano Lonardi. Though I only took a single class with him, I found that it was one of the most important classes I would take as a graduate student. The fundamental knowledge he has given me has been paramount in my research.

In the four years I have spent at the University of California, Riverside, I have met many people and have made great friendships. I owe many thanks to all of these friends, especially those in my lab. I thank Xiaoyue Wang, Dr. Yu Fan, Bilson Jake Libres Campana, Qiang

Zhu, Jing Xu, and Jin-Wien Shieh. I have spent most of my time in the lab and you all have made it a family-like environment for me.

Another person I need to give my special thanks to is Giulia Hoffmann. She helped me with my English writing a lot. She was always very willing to help and patiently and nicely taught me English.

Last, but most importantly, I would like to give my deepest thanks to my parents. Thank you for trusting me to go so far away to realize my dreams. I have always understood your worries and concerns and have appreciated the love and support you have always given me regardless. Though separated by thousands of miles, I have always felt your encourangement and caring in every moment of my time here. Thank you, you have always been and always will be my biggest motivation to achieve my goals and dreams, and beyond.

ABSTRACT OF THE DISSERTATION

Learning from Time Series in the Resource-Limited Situations

by

Lexiang Ye

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2010
Dr. Eamonn Keogh, Chairperson

Many data mining tasks are based on datasets containing sequential characteristics, such as web search queries, medical monitoring data, motion capture records, and astronomical observations. In these and many other applications, a time series is a concise yet expressive representation. A wealth of current data mining research on time series is focused on providing exact solutions in such small datasets. However, advances in storage techniques and the increasing ubiquity of distributed systems make realistic time series datasets orders of magnitude larger than the size that most of those solutions can handle due to computational resource limitations. On the other hand, proposed approximate solutions such as dimensionality reduction and sampling suffer from two drawbacks: they do not adapt to available computational resources and they often require complicated parameter tuning to produce high quality results.

In this dissertation, we discuss anytime/anyspace algorithms as a way to address these issues. Anytime/anyspace algorithms (after a small amount of setup time/space) are algorithms that always have a best-so-far answer available. The quality of these answers improves as more computational time/space is provided. We show that by framing problems as anytime/anyspace algorithms, we can extract the most benefit from the available computational resources and provide high-quality approximate solutions accordingly.

We further argue that it is not always effective and efficient to rely on whole datasets. When the data is noisy, using distinguishing local features rather than global features can mitigate the effect of noise. Moreover, building a concise model based on local features makes the computational time and space much less expensive. We introduce a new time series primitive, time series shapelets, as a distinguishing feature. Informally, shapelets are time series subsequences which are in some sense maximally representative of a class. As we shall show with extensive empirical evaluations in diverse domains, classification algorithms based on the time series shapelet primitives can be interpretable, more accurate, and significantly faster than state-of-the-art classifiers.

Contents

List of Tables **xii**

List of Figures **xiii**

1 Introduction **1**

1.1 Learning from time series while compromising with resources 4

1.1.1 Distance Measure 5

1.1.2 Similarity Join under the Anytime Framework 8

1.1.3 Indexing under the Anyspace Framework 12

1.1.4 Feature Extraction for Time Series Classification 13

1.2 Contributions 16

2 The Asymmetric Approximate Anytime Join **18**

2.1 Introduction 19

2.1.1 Background and Assumptions 21

2.2 The Asymmetric Approximate Anytime Join 24

2.2.1 Augmented Orchard's Algorithm 27

2.3 Experiments and Case Studies 34

2.3.1 A Sanity Check on Random Walk Data 34

2.3.2 Anytime Classification of Batched Data 35

2.3.3 Historical Manuscript Annotation 39

2.3.4 Anytime Anomaly Detection 45

2.4 Conclusions 49

3 Auto-cannibalistic and Anyspace Indexing Algorithms with Applications to Sensor Data Mining 50

3.1 Introduction 51

3.2 Classic Orchard's Algorithm 52

3.3 Anyspace Orchard's Algorithm 56

3.3.1 Truncated Orchard's Algorithm 56

3.3.2 Anyspace Orchard's Algorithm 58

3.3.3 Constructing Anyspace Orchard's 62

3.3.4 Using Anyspace Orchard's Algorithm 66

3.3.5 Optimizations of Anyspace Orchard's Algorithm 69

3.4 Experiments 71

3.5 Experimental Case Studies 74

3.5.1 Insect Monitoring 74

3.5.2 Robotic Sensors 78

3.6 Discussion 81

3.6.1 Related Work 81

3.6.2 Conclusion 82

4 Time Series Shapelets: A Novel Technique that Allows Accurate, Interpretable and Fast Classification 84

4.1 Introduction 85

4.2 Related Work and Background 91

4.2.1 Related Work 91

4.2.2 Notation 93

4.3 Finding the Shapelet 100

4.3.1 Brute-Force Algorithm 100

4.3.2 Subsequence Distance Early Abandon 104

4.3.3 Admissible Entropy Pruning 106

4.3.4 Techniques for Breaking Ties 110

4.4 Shapelets for Classification 112

4.5 Experimental Evaluation 114

4.5.1 Experiment Setup 114

4.5.2 Performance Comparison 116

4.5.3 Projectile Points (Arrowheads) 119

4.5.4 Mining Historical Documents 120

4.5.5 Understanding the Gun/NoGun Problem 123

4.5.6 Wheat Spectrography 125

4.5.7 Coffee Spectrography 127

4.5.8 Mallat 128

4.5.9 Gait Analysis 130

4.5.10 The Effect of Noise on Training and Testing a Shapelet Classifier . . 135

4.6 Conclusions and Future Work 138

5 Conclusions **139**

Bibliography **143**

List of Tables

2.1 An abridged trace of the current classification of the 30 characters of the string “SGT PEPPERS LONELY HEARTS CLUB BAND” vs. the number of distance calculations made 36

3.1 Orchard’s Algorithm Ranked Lists $P[A]$ 53

3.2 Truncated Orchard’s Algorithm 56

3.3 Highly Truncated Orchard’s Algorithm 57

3.4 Anyspace Orchard’s Ranked Lists (I) 62

3.5 Anyspace Orchard’s Ranked Lists (II) 62

3.6 Anyspace Orchard’s Ranked Lists (III) 66

4.1 Symbol table 94

List of Figures

1.1 *Top*) Time series representation of 10-second electrocardiogram records. The abnormal data is recorded from a 72-year-old female. There is a missed-beat subsequence in the range of [2100, 2700]. The normal data is recorded from a 65-year-old male. *Bottom*) Time series representation of the stock market trends. The data starts on February 5, 1971 and ends on July 23, 2009 5

1.2 Time series representation for a *Texas Duran* projectile point outline 6

1.3 *Top*) DTW flexibly warps data points to minimize the distance between two time series. *Bottom*) A case where DTW gives a more intuitive distance measure than Euclidean distance 9

1.4 *Top*) All nearest neighbor similarity join example on a butterfly data set. The similarity measure takes shape, color, and texture into consideration. Data set *A* is for drawings of butterflies and data set *B* is for real pictures. Every butterfly in data set *A* is matched with its nearest neighbor in *B*, while some of the butterflies in *B* can be unmatched 10

1.5 *Left*) Time series representation of five projectile points in the same class. The broken part of the projectile point and its corresponding subsequence in the time series is circled. *Right*) The classification model we used in Chapter 4 only relies on the most distinguishing subsequence (highlighted in red) . . 14

2.1 An abstract illustration of an anytime algorithm. Note that the quality of the solution keeps improving until the time the algorithm is interrupted by the user 23

2.2 *Left*) The triangular inequality is used in Orchard's Algorithm to prune the items in A that cannot possibly be the nearest neighbor of query q. *Right*) Similarly, the triangular inequality is used in Augmented Orchard's Algorithm to prune the items in A that are certain to have a better best-so-far match than the current query q 29

2.3 The log of Q as a function of the number of calls to the Euclidean distance . 34

2.4 *Left*) The normalized value of Q as a function of the number of calls to the Euclidean distance. *Right*) The accuracy of classification, averaged over the 30 unknown letters, as a function of the number of calls to the Euclidean distance 37

2.5 *Left*) The normalized value of Q as a function of the number of calls to the Euclidean distance. *Right*) The accuracy of classification, averaged over the 1,000 star light curves, as a function of the number of calls to the Euclidean distance 38

2.6 Page 967 of “The Theatre of Insects; or, Lesser living Creatures...” [54], published in 1658, showing the “Day” butterfly 39

2.7 We can compare two shapes by converting them to one-dimensional representations and using an appropriate distance measure, in this case Dynamic Time Warping 41

2.8 The normalized value of Q as a function of the number of calls to the Euclidean distance 43

2.9 *Left column*) Eight sample images from the data set, a collection of butterfly images from historical manuscripts. *Center column*) The best matching images after AAAJ has seen 10% of the data. *Right column*) The best matching images after AAAJ has seen all 44

2.10 Various light curve vectors in the data set. (a) and (b) are normal ones. (c) and (d) are outliers 46

2.11 Average nearest neighbor distance with respect to the performed Euclidean distance comparisons. *Top*) A is composed only of normal elements. *Bottom*) A contains one outlier 48

3.1 A small data set A containing 7 items, used as a running example in this chapter 53

3.2 *Left*) Assume we know the pairwise distances between a_i, a_j and $a_{j'}$. A newly arrived query q must be answered. *Right*) After calculating the distance $\textsc{Dist}(q, a_i)$ we can conclude that items with a distance to a_i less than or equal to $2 \times \textsc{Dist}(q, a_i)$(i.e. the gray area) might be the nearest neighbor of q, but everything else, including $a_{j'}$ in this example, can be excluded from consideration 54

3.3 An idealized Anyspace Indexing Algorithm 59

3.4 Assume a_j is the current nearest neighbor of query q, and that $P[a_j]$ was deleted and replaced with the *goto* entry a_i, *Left*) A newly arrived query q must be answered. *Right*) An admissible pruning rule is to exclude items whose distance to a_i is greater than or equal to $\textsc{Dist}(a_i, a_j)+2\times\textsc{Dist}(q, a_j)$. In this example, everything outside the large gray circle can be pruned 70

3.5 The indexing efficiency vs. level of truncation for a synthetic data set 72

3.6 The indexing efficiency vs. level of truncation for the Dodgers data set . . . 73

3.7 The distribution of distances to nearest neighbor for 1,000 insects in our training set. We consider exemplars whose distance to their nearest neighbor is more than the mean plus 4 standard deviations to be suspicious and worthy of follow-up investigation 76

3.8 Indexing efficiency vs. space on the insect monitoring problem 77

3.9 *Top*) The cumulative number of Euclidean distance calculations performed vs. the number of sound events processed. *Bottom*) The size of auto-cannibalistic algorithm vs. the number of sound events processed 78

3.10 *clockwise from top right.* A Sony Aibo robot. An on-board sensor can measure acceleration at 125 Hz. The accelerometer data projected into two dimensions 79

3.11 The distribution of distances to nearest neighbor for 700 tactile events in our training set. We consider exemplars whose distance to their nearest neighbor is more that the mean plus five standard deviations to be suspicious, and worthy the memory required to take a photograph 80

3.12 *Top*) The cumulative number of Euclidean distance calculations performed vs. the number of tactile events processed. *Bottom*) The size of auto-cannibalistic algorithm vs. the number of tactile events processed 81

4.1 Samples of leaves from two species. Note that several leaves have insect-bite damage 86

4.2 A shape can be converted into a one dimensional “time series” representation. The reason for the highlighted section of the time series will be made apparent shortly 87

4.3 Here, the shapelet hinted at in Figure 4.2 (in both cases shown with a bold line) is the subsequence that best discriminates between the two classes . . . 88

4.4 A decision-tree classifier for the leaf problem. The object to be classified has all of its subsequences compared to the shapelet, and if any subsequence is less than (the empirically determined value of) 5.1, it is classified as *Verbena urticifolia* 89

4.5 Illustration of the best matching location in time series T for subsequence S . 96

4.6 The CHECKCANDIDATE() function at the heart of the brute force search algorithm can be regarded as testing to see how mapping all of the time series objects on the number line, based on their SUBSEQUENCEDIST(T, S), separates the two classes 101

4.7 *Left*) Illustration of complete Euclidean distance. *Right*) Illustration of Euclidean distance early abandon 105

4.8 Distance arrangement of the time series objects in one-dimensional representation of best-so-far information gain. The positions of the objects represent their distances to the candidate 107

4.9 The arrangement of first five distances from the time series objects to the candidate 107

4.10 One optimistic prediction of distance distribution based on distances that have already been calculated in Figure 4.9. The dashed objects are in the optimistically assumed placements 108

4.11 The time required to find the best shapelet for increasing large dataset sizes . 117

4.12 The hold-out accuracy for increasing large dataset sizes 117

4.13 The time required to find the best shapelet for increasing lengths of time series objects in the dataset, with various dataset sizes 118

4.14 Examples of the three classes of projectile points in our dataset. The testing dataset includes some broken points and some drawings taken from anthropologists' field notes 119

4.15 *Top*) The dictionary of shapelets together with the thresholds d_{th}. The x-axis shows the position of the shapelets in the original time series. *Bottom*) The decision tree for the 3-class projectile points problem 120

4.16 Examples of the three classes in our dataset. The shields were hand-drawn one to six centuries ago 121

4.17 *Top*) The dictionary of shapelets, together with the thresholds d_{th}. The x-axis shows the position of the shapelets in the original time series. *Bottom*) A decision tree for heraldic shields 122

4.18 The top section of a page of the 1840 text, *A guide to the study of heraldry* [53]. Note that some shields are torn 123

4.19 *Top*) The dictionary of shapelets, with the thresholds d_{th}. The x-axis shows the position of the shapelet in the original time series. *Bottom*) The decision tree for the Gun/NoGun problem 124

4.20 One sample from each of the seven classes in the wheat problem. The objects are separated in the y-axis for visual clarity, as they all have approximately the same mean 125

4.21 *Top*) The dictionary of shapelets, with the thresholds d_{th}. The x-axis shows the position of the shapelet in the original time series. *Bottom*) The decision tree for the wheat spectrography problem 126

4.22 *Top*) The dictionary of shapelets, with the thresholds d_{th}. The x-axis shows the position of the shapelet in the original time series. *Bottom*) The decision tree for the coffee problem 128

4.23 One sample from each of the eight classes in the Mallat problem 129

4.24 *Top*) The dictionary of shapelets, with the thresholds d_{th}. The x-axis shows the position of the shapelet in the original time series. *Bottom*) The decision tree for the Mallat problem 131

4.25 The illustration of a data item of a walking motion. Each data item consists of two time series in the same time interval. Using a sliding window across the data item, we get the pairs of subsequences as the shapelet candidates . . 132

4.26 The accuracy comparison between the shapelet classifier and the rotation invariant nearest neighbor under three segmentation schemes. The shapelet classification method outperforms the nearest neighbor in all three segmentation schemes. However, careful segmentation reduces the accuracy difference between the two methods 134

4.27 *Top*) The dictionary of shapelets, with the thresholds d_{th}. The x-axis shows the position of the shapelet in the original time series. *Bottom*) The decision tree for the 2-class gait analysis problem 134

4.28 Data samples from Lightning EMP dataset (*Left*) and Mallat dataset (*Right*) . 135

4.29 The change of classification accuracy on Gun/noGun dataset as we add noise to the data. *Left*) Noise is added only to the training dataset. *Right*) Noise is added only to the testing dataset. The lighter lines indicate plus/minus one standard deviation 136

Chapter 1

Introduction

Data mining is the process of utilizing sophisticated data analysis methods to discover previously unknown, valid information from large data sets [24]. Data mining is one of the most influential disciplines in computer science developed in the last few decades. With significant advances in machine learning theories and increased industry-driven demands, data mining techniques have pervaded almost every aspect of our lives. Search engines, such as Google and Yahoo!, have greatly reduced a person's time searching for online information [28]. Social networking communities, such as Facebook and Myspace, offer people opportunities to experience a virtual world, where communication distances between people are tremendously shortened [39]. In medicine and genetics, potential diseases can be predicted and prevented more accurately and easily through understanding DNA sequences [57]. In business, companies target consumers with efficient and directed advertising through profile and/or behavioral analysis [72]. In these cases and many others, extensive and profound changes in our daily lives, society, and the economy have been largely attributed to the development of data mining techniques.

The ability to generate, capture, and store data has greatly improved as its necessity has increased [23]. In order to manage increasingly large data in an effective way, data representations have become more and more complex. Data has been represented in individual data to relational tables, progressing to sequential data, and now is evolving into graph data [17]. The growing representational sophistication of the data implies more comprehensive and intrinsic relationships within it, as well as requiring more effort to explore the information hidden within the data. Pushed by data volume demands, storage technology has also advanced at an incredible speed [33]. Data was stored at the gigabyte level until very recently, and now we are starting to discuss data in the terabytes. Following this trend, consumer machines will approach the petabyte range in the very near future [71].

Advances in these data manipulation and storage techniques offer researchers great opportunities to understand data at a profound and comprehensive level, as well as introducing the difficult challenge of how best to utilize the data in a more efficient way. The divergence between our ability to gather, organize, and explore data is becoming larger [11]. The rate of generating and gathering the raw data far exceeds the speed of analyzing and understanding the data. What is more, data can be generated over a long period of time, such as one week, one month, or even much longer. However, people are unwilling to wait the same amount of time for a meaningful result from the gathered data. Therefore, the increased capability of generating and storing raw data makes time a resource-limiting factor for extracting valuable information. An example of a case where time is considered as the limited resource will be discussed in Chapter 2.

Although high-performance computers such as mainframes and supercomputers make it more common to handle data at the terabyte level, low-power/low-memory distributed systems such as smartphones, sensors, and robots are becoming prevalent. There is an obvious difference between high-performance computers and distributed systems in their computing and storage ability. It is difficult to directly transplant techniques and algorithms from machines with high computing and storage abilities to these low-capacity, distributed systems. In these cases, space is the limiting resource and should be taken into consideration when designing data mining techniques. In Chapter 3 we will study a problem where space is the limiting factor.

However, there are other applications where instead of compromising the decision quality with limited resources, we can build a concise model, extract the most useful information, and remove all other redundant data or irrelevant noise. With less data, yet more information, this model saves both time and space. Such a problem will be considered in Chapter 4.

In the rest of this dissertation, we will discuss several data mining approaches which efficiently organize and explore data.

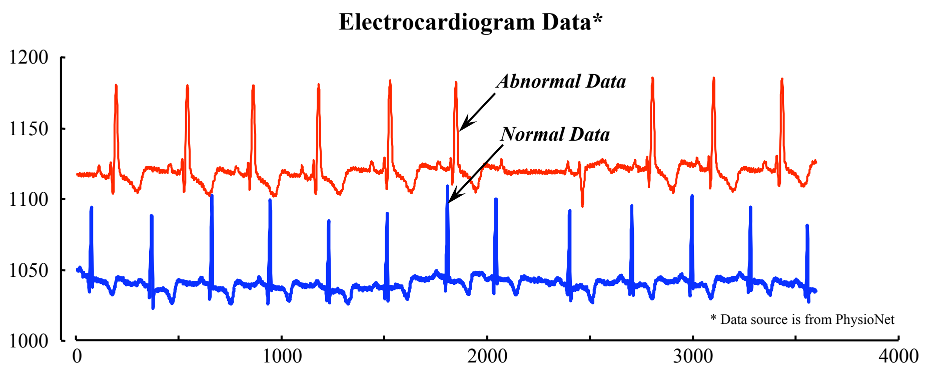
1.1 Learning from time series while compromising with resources

In much of this work, we use time series as the data representation. A time series is a concise yet expressive representation of many real-world phenomena, such as star light curves in astronomy (cf. Section 2.3.4), electrocardiography that records heartbeats over time in medicine, and stock market trends in finance. Figure 1.1 shows the time series representation for electrocardiogram records and stock market trends.

Since much of this work is focused on time series, below we give a formal definition of time series.

Definition 1. *Time Series.* A time series $T = t_1 \ldots t_m$ is an ordered set of m real-valued variables. Data points $t_1 \ldots t_m$ are typically arranged by temporal order and spaced at equal time intervals.

Time series are ubiquitous in a large fraction of data mining domains, such as motion capture [67], meteorology [59], finance [72], handwriting [78], medicine [20], web logs [98], music [37], etc. However, it should be noted that, besides correlating data with time, there are other pseudo time series that consider data points in an ordered sequence. In spectrography [41], data values are ordered by component wavelengths (cf. Section 4.5.6 and 4.5.7); in color histograms [77], the number of pixels are sorted by color component values; in analyzing shapes [79], the order could be the clockwise direction analysis starting from a specific point in the outline. Such a shape time series example is shown in Figure 1.2. Relax-



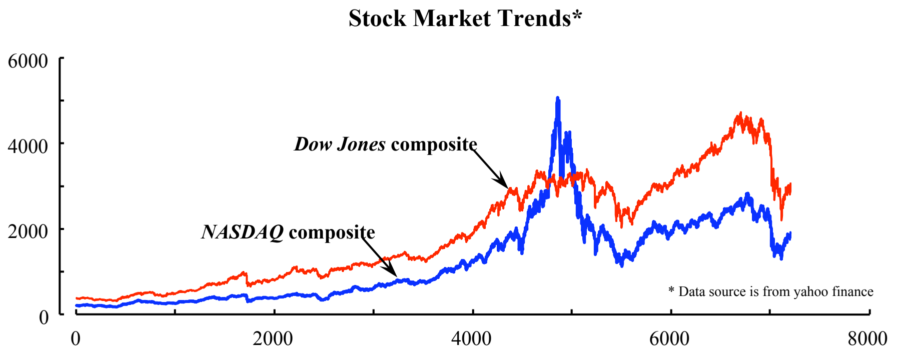


Figure 1.1: *Top*) Time series representation of 10-second electrocardiogram records. The abnormal data is recorded from a 72-year-old female. There is a missed-beat subsequence in the range of [2100, 2700]. The normal data is recorded from a 65-year-old male. *Bottom*) Time series representation of the stock market trends. The data starts on February 5, 1971 and ends on July 23, 2009

ing the time order restriction allows time series to be generally applicable to data not usually considered to be related to time.

1.1.1 Distance Measure

Similarity search is a useful subroutine in many time series applications such as: motif discovery [83], rule discovery [19], anomaly detection [89], and indexing [43]. A distance measure between time series, which intuitively reflects the underlying similarity of data and

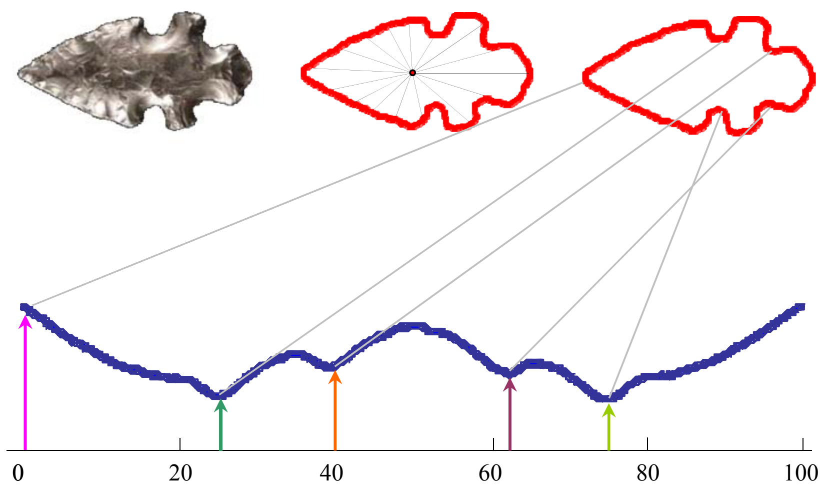


Figure 1.2: Time series representation for a *Texas Duran* projectile point outline

scales well in high-dimensional and large data sets, is very desirable for similarity search. We provide the definition of a distance measure between the time series $\textsc{Dist}(T, R)$.

Definition 2. *Distance between the time series.* $\textsc{Dist}(T, R)$ is a distance function that takes two time series, T and R, as inputs and returns a nonnegative value, d, which is said to be the distance between T and R. We require that the function $\textsc{Dist}$ be symmetrical; that is, $\textsc{Dist}(R, T) = \textsc{Dist}(T, R)$.

There are two common distance measures for time series:

Euclidean Distance

Euclidean distance is the simplest and most generic distance measure [22]. Given two time series $T = t_1 \ldots t_n$ and $R = r_1 \ldots r_n$, the Euclidean distance between them is defined as:

$$\textsc{Dist}_{Euclidean}(T, R) = \sqrt{\sum_{i=1}^{n}(t_i - r_i)^2} \tag{1.1}$$

Informally, Euclidean distance accumulates the distance difference point by point for the whole time series. Euclidean distance is an efficient distance measurement because it is indexable and satisfies the necessary conditions to be *metric*, which is an advantage we will exploit for the approaches discussed in Chapter 2 and Chapter 3.

Dynamic Time Warping (DTW)

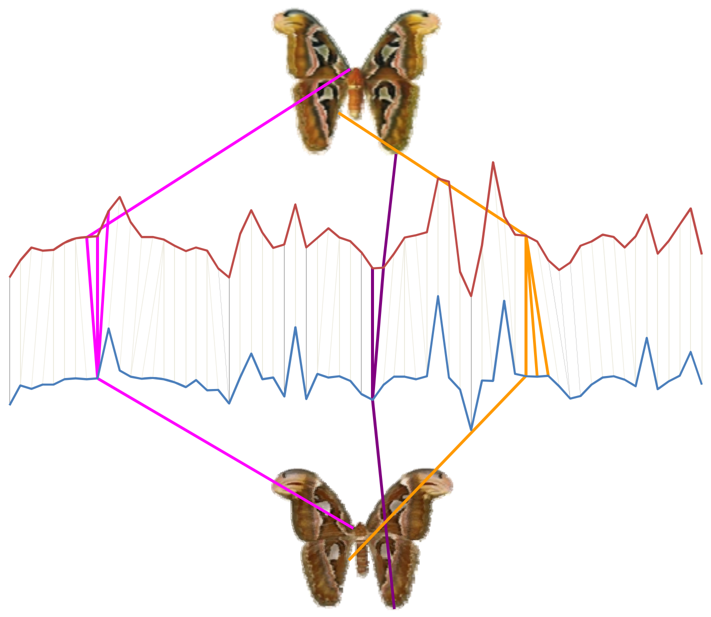
Because of the motivation to handle data that is possibly out-of-phase (such as in the gait analysis problem in motion capture, where different people walk at a different pace), the dynamic time warping (DTW) algorithm was introduced into the data mining community [4] to provide a better distance measurement by flexibly stretching or compressing time series data (see Figure 1.3 for example). DTW finds a minimum distance (optimal match) between time series. In DTW, the one-to-one data point mapping in the Euclidean distance is relaxed. The neighboring data points are warped together to do a one-to-many or many-to-one data point mapping. DTW is a dynamic programming algorithm and the DTW distance can be calculated recursively as

$$\textsc{Dist}_{DTW}(T_i, R_j) = d(t_i, r_j) + \min \begin{cases} \textsc{Dist}_{DTW}(T_{i-1}, R_{j-1}) \\ \textsc{Dist}_{DTW}(T_{i-1}, R_j) \\ \textsc{Dist}_{DTW}(T_i, R_{j-1}) \end{cases} \tag{1.2}$$

Computational efficiency has traditionally been a challenge for DTW-based techniques [43] [45] [94]. Recent research has shown that providing a constrained warping window size not only improves computational efficiency, but potentially increases the accuracy of the similarity measure. Based on the idea of constrained warping, the improved algorithms provide a lower-bound distance [82] as well as the exact indexing time series on DTW [43], both of which tremendously reduce the computational time [9] [62].

1.1.2 Similarity Join under the Anytime Framework

A similarity join is an extremely important primitive for many data mining algorithms, which is achieved by applying similarity searches between two data sets. There are three well-known types of the similarity join. One is the *range join* [56], where the similarity is defined based on the distance threshold ε, and every two objects which have their distance bounded by ε should be combined together. The second is the *closest-point query* [85]. Here, given a number k, the top k pairs of the objects with the smallest distances are returned as the join



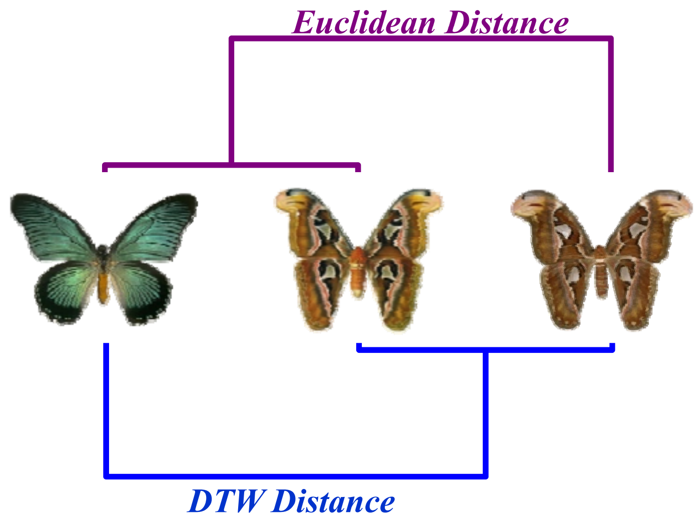


Figure 1.3: *Top*) DTW flexibly warps data points to minimize the distance between two time series. *Bottom*) A case where DTW gives a more intuitive distance measure than Euclidean distance

result. In the third case, the variant similarity join is considered as the *all nearest neighbor join* [13]. For each data item in the first set, the join returns its nearest neighbor in the second

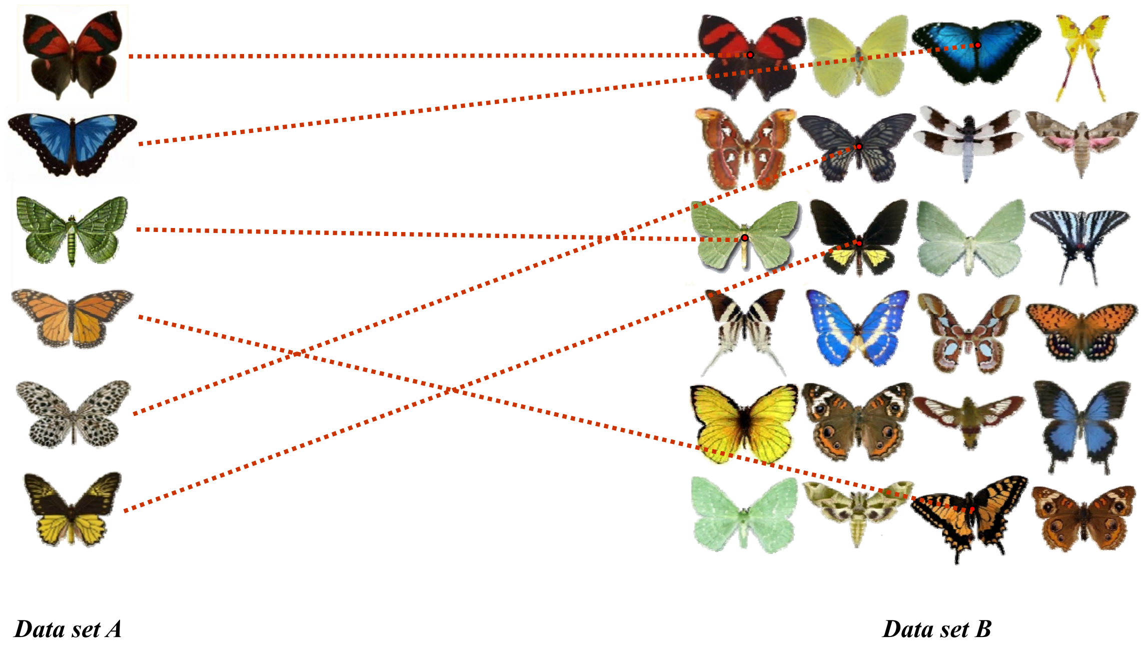


Figure 1.4: *Top*) All nearest neighbor similarity join example on a butterfly data set. The similarity measure takes shape, color, and texture into consideration. Data set A is for drawings of butterflies and data set B is for real pictures. Every butterfly in data set A is matched with its nearest neighbor in B, while some of the butterflies in B can be unmatched

set. We consider the last case of the similarity join in our study, and an example of a butterfly data set (cf. Section 2.3.3) is shown in Figure 1.4.

An effective and efficient similarity join algorithm is very desirable in the data mining community. Similarity joins have a wide variety of applications, such as identifying similar webpages [86], content-based image searching [50], and detecting moving objects of similar trajectories [21]. Furthermore, a similarity join is a primitive for many other data mining techniques, such as clustering [5], pattern recognition and classification [47], and graph-based computational learning [46].

However, as data gathering and data storage capabilities improve at an ever-increasing pace, the rate of data collection sometimes prevents indexing before the similarity join. Internet data, for instance, is such an example: the rate that users from all over the world add new content on the Internet is much faster than the rate any computer can download and index it. In our join problem, we consider an *asymmetric approximate* similarity join. In this context, *asymmetric* means that the sizes of the two data sets are very different,where the second set is many orders of magnitude larger than the first one. *Approximate* indicates that instead of searching for the exact answer in the second set, because of the time limitation, we aim at finding a high-quality answer matched with each object in the first set.

The total time for a similarity join is proportional to the product of the number of objects in each of the two data sets, which might not be affordable in the time-limited case. In Chapter 2, we cast the join algorithm in the anytime framework. Rather than finding perfect answers for some of the items in the first set and making the others randomly paired with data from the second set when the time is up, we consider every item at the same time and spend our time wisely. The algorithm always has the best-so-far answer for each item in the first set. The quality of the answer improves with more computational time.

In Chapter 3, we further show that if a metric distance measurement is used, the first set can be simply indexed and exhibits a large speedup for the anytime algorithm. With the help of the indexing, we can efficiently locate the nearest neighbor in the first set for each item b in the second set. Taking advantage of the triangular inequality of a metric distance measurement, we update the nearest neighbor distance for each item in the first set if b is potentially nearer than the best so far, and we thus avoid many unnecessary distance calculations.

1.1.3 Indexing under the Anyspace Framework

The traditional approach to supporting similarity searches is to use an index structure [2] [27]. For the benefit of fast searching in data sets, we pay the cost of storing additional information in memory other than the data itself to quickly locate a specific data item. As the size of problems in computations becomes greater, the size of the indicies increases at the same time. (Note that an index is not necessarily smaller than the data set itself.) What if an index is too large to fit into the memory? Meanwhile, there are circumstances where intercommunications are implemented between high-performance computers and distributed systems. We would like distributed systems to do searching for data stored in high-performance computers. How can we adapt an index of a large data set into the low-memory distributed systems?

In Chapter 2, we introduce a simple index method for the smaller data set to accelerate similarity search. This indexing method is completely parameter free, simple to implement and scales well for high-dimensional data. However, it has the drawback of a quadratic space requirement for the index, which prohibits the indexing method from scaling well with large data sets. In Chapter 3, we mitigate the problem by casting the indexing method in the anyspace framework, where the index is flexibly built on the available space, and the algorithm keeps the most important part of the index in memory to provide the best searching performance.

We note that the index we used in Chapter 2 has three properties which can be taken advantage of to compress the space requirement. First, the index is a table where each item has its own entry. Second, the indexing structure of each item in a data set is exactly the same. The structure is totally independent of the actual value of the items. Third, items close to each other in original space have redundant data in the index. Making use of these three properties, we develop an algorithm to reorder the entries for each item by their utility for indexing. The anyspace algorithm will truncate the entries with lower utility scores when space is limited.

Under the anyspace framework, we also develop an *auto-cannibalistic* algorithm. The idea of an auto-cannibalistic algorithm is to utilize the space as efficiently as possible. Therefore, the algorithm initially fills up memory with an index. Under certain circumstances, if there are extra space requirements from any other process, the index dynamically deletes part of itself to make space for new data. Under this mechanism, we can always make full use of the available space and thus maximize the overall performance of ongoing tasks.

1.1.4 Feature Extraction for Time Series Classification

In Chapter 2 and Chapter 3, we overcome the similarity join problem and indexing problem using a similar methodology. We build the algorithm or the data structure directly over all of the information from the entire data set. When time or space is limited under real cases, the algorithm tries to minimize the resource usage based on the data similarity. In Chapter 2, when time is limited, the similarity join algorithm skips all the unvisited items and assigns each object to be joined with an approximate item seen thus far; the approximate item is generally very similar to the best answer after doing an exhaustive search. In Chapter 3, when space is a scarce resource, the indexing algorithm removes the most unimportant portion of the data structure. The importance of the entries in the indexing data structure is evaluated by the data redundancy; the more redundancy between the entry and other entries left in the data structure, the less important the entry is and the sooner the entry will be removed.

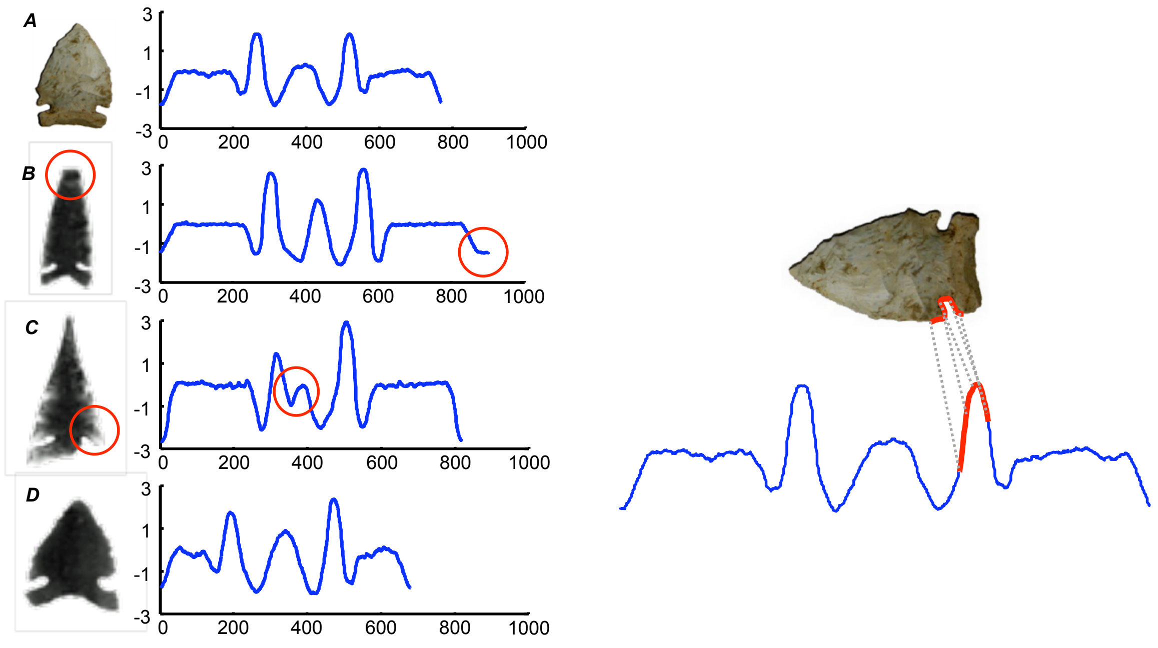


Figure 1.5: *Left*) Time series representation of five projectile points in the same class. The broken part of the projectile point and its corresponding subsequence in the time series is circled. *Right*) The classification model we used in Chapter 4 only relies on the most distinguishing subsequence (highlighted in red)

In Chapter 4, we address the time series classification problem in a different way. Instead of using the entire data set to build the model, we rely only on good features from the data set and make the model as concise and precise as possible. The time series classification problem is a supervised learning problem. Given a training set where each time series is associated with a specific class label, the goal is to build a model and to map the unlabeled, arriving time series to one of the predefined classes through the model. The high dimensionality property of the time series makes many state-of-the-art classification methods ineffective. Extensive experiments show that the simple nearest neighbor classifier is the competitive leading model for time series classification [22]. However, the drawback of the nearest neighbor classifier is especially obvious under limited resources: comparing each target time series with the entire training data set is both time-consuming and space-wasting. To avoid the comparison to all the time series data of one class which would be similar, we use concise features to represent the data of that class, and do the comparison based on the features rather than on the raw data. Figure 1.5, for example, presents four time series of the projectile point in one class on the left. These time series are globally similar due to the visual similarity. If an unknown time series is similar to one of the time series, it is also similar to the others. Comparisons between the unknown time series and all of these four time series are wasteful and should be avoided. Moreover, the time series *B* and *C* contain noise because of the broken part in the original shapes. This noise is a common artifact in real-world problems, which makes the distance measurement non-intuitive and increases the difficulty of the classification. Therefore, the noise in individual time series should be omitted in the general model.

Our model in Chapter 4 is capable of filtering out the noise, removing the data redundancy, and extracting the intrinsic features defined for one class. In our model we view subsequences from time series as features. A good feature is a subsequence that has small distances to all of the instances in one class and much larger distances to the instances in other classes. In Figure 1.5 right, the small notched hafting part is the most distinguishing common subsequence for the class. Thus we make an unknown time series comparison to this subsequence feature to determine whether it belongs to this class. The algorithm in Chapter 4 is trying to use minimal information for its time series classification model. Storing and comparing with the minimum information saves much more space and computing time. Additionally, noise removal while using the minimum information makes the classification model more accurate. Moreover, the subsequence allows for easier interpretation of the data and the classification result.

1.2 Contributions

We can summarize the main contributions of this thesis as follows:

- We provide an anytime framework for the similarity join problem. When time is a limiting resource, the algorithm provides the best possible result to make use of the available computational time. We argue that in many problems, the approximate join result is as useful as the exact join result.

- We cast the indexing problem in an anyspace framework. We show that an efficient indexing method can be rescued from its quadratic space complexity by utilizing the unique properties of the indexing data structure. We successfully applied this anyspace indexing algorithm to sensor data mining. The ability to create an anyspace algorithm also allows us to create auto-cannibalistic algorithms, which dynamically delete parts of the indexing structure to make room for new space usage.

- We introduce time series *shapelets* as a new primitive for time series classification. With the help of the shapelet, which is a distinguishing subsequence, the classification algorithm avoids storing all the training data or complicated models, and thus saves both space and time. Furthermore, the shapelet presents an interpretable classification result and greater accuracy.

Chapter 2

The Asymmetric Approximate Anytime Join

This chapter discusses similarity join problems, where the computational time is especially considered as the limiting factor. It has long been noted that many data mining algorithms can be built on top of join algorithms [5, 46]. This has lead to a wealth of recent work on efficiently supporting such joins with various indexing techniques. However, there are many applications which are characterized by two special conditions, firstly the two data sets to be joined are of radically different sizes, which is a situation we call an asymmetric join. Secondly, the two data sets are not, and possibly can not be indexed for some reason. In such circumstances the time complexity is proportional to the product of the number of objects in each of the two data sets, an untenable proposition in most cases. In this chapter we make two contributions to mitigate this situation. We argue that for many applications, an exact solution to the problem is not required, and we show that by framing the problem as an anytime algorithm we can extract most of the benefit of a join in a small fraction of the time taken by the full algorithm. In situations where the exact answer is required, we show that we can quickly index just the smaller data set on the fly, and greatly speed up the exact computation. We motivate and empirically confirm the utility of our ideas with case studies on problems as diverse as batch classification, anomaly detection, and annotation of historical manuscripts.

2.1 Introduction

Many researchers have noted that many data mining algorithms can be built on top of an approximate join algorithm. This has lead to a wealth of recent work on efficiently supporting such joins with various indexing techniques [6, 13, 84, 97]. However, we argue that while the classic database use of approximate joins for record linkage (entity resolution, duplicate detection, record matching, reduplication, merge/purge processing database hardening etc.) does require a full join, many data mining/information retrieval uses of joins can achieve the same end result with an approximate join. Here approximate does not refer to the distance measure or rule used to link two objects, but rather to the fact that only a small subset of the Cartesian product of the two data sets needs to be examined. While the result will not be the same as that of an exhaustive join, it can often be good enough for the task at hand.

For example, when performing a classic record linkage, if one data set contains "`John Lennon, 9 October 1940`", and the other contains "`John W. Lennon, 09-Oct -40`", it is clear that these correspond to the same person, and an algorithm that failed to link them would be very undesirable. In contrast, for many data mining applications of joins it is not really necessary to find the nearest neighbor, it can suffice to find a near-enough neighbor. Examples of useful tasks that utilize the detection of near-enough neighbors as a subroutine include clustering [25], classification [82], anomaly detection [79] and as we show in Section 2.3.3, historical manuscript annotation. Given this, we show that by framing the problem as an anytime algorithm we can extract most of the benefit of the full join algorithm in only a small fraction of the time that it requires. Anytime algorithms are algorithms that trade execution time for quality of results [32]. In particular, an anytime algorithm always has a best-so-far answer available, and the quality of the answer improves with execution time. The user may examine this answer at any time, and then choose to terminate the algorithm, temporarily suspend the algorithm, or allow the algorithm to run to completion.

Furthermore, we show that although we are explicitly assuming the data is not indexed at query time, we can build an index on the fly for the smaller data set and greatly speed up the process.

The rest of the chapter is organized as follows. The next section offers more background and explicitly states our assumptions. Section 2.2 introduces our algorithm and Section 2.3 offers detailed experiments and case studies. We discuss our results and offer conclusions and directions for future work in Section 2.4.

2.1.1 Background and Assumptions

The *Similarity Join* (SJ) combines two sets of complex objects such that the result contains all pairs of similar objects [6]. It is essentially the classic database join which has been relaxed to allow linkage of two objects that satisfy a similarity criterion. The related *All Nearest Neighbor* (ANN) operation takes as input two sets of multi-dimensional data points and computes for each point in the first set the nearest neighbor in the second set [13]. Note that this definition allows for points in the second set to be unmatched. In this chapter we introduce the *Asymmetric Approximate Anytime Join* (AAAJ) which also allows objects in the second set to be unmatched; however, it differs from the above in several important ways:

- We assume that the second set is many orders of magnitude larger than the first set. In some cases the second set may be considered effectively infinite [1], for example, this may be the set of all images on the internet or some streaming data.
- The sheer size of the second set means that it cannot be indexed, or can only be weakly indexed. For example we cannot index the billions of high dimensional images on the WWW, but we can use Google image search to weakly order images on *size*, *date* of creation or most significantly (cf. Section 2.3.3) *keywords* surrounding them.
- The vast majority of work in this area assumes the distance metric used is the Euclidean distance [6, 13, 84, 97]. However, motivated by several real world problems we need

[1]We call the set of images on the WWW "*effectively infinite*" because it is expanded at a rate faster that the download rate of any one machine.

to be able to support more general measures such as Dynamic Time Warping (DTW) (cf. section 1.1.1), rotation invariant Euclidean distance, or weighted combinations of individual measures such as shape, color and texture similarity.

- Given that the second set may be effectively infinite, we may need to abandon any notion of finding an exact answer; rather we hope to find a high quality answer. In such circumstances we frame the problem as an anytime algorithm.

Note that it is critical to the motivation of this chapter that we assume that the second set is *not* indexed, because there are many excellent techniques for computing all manner of variations of joins when the data *is* indexed [6, 13, 84, 97]. In addition to the reasons noted above, additional reasons why the data might not be indexed include the following:

- The input query could be intermediate results of complex database queries (as noted in [97]), or the incremental results of a directed web crawl.

- The high dimensionality of the data we need to consider. For example, the five data sets considered in [13] have an average dimensionality of 4.8, the data sets considered in [97] are all 2 dimensional and even [84] which is optimized for "*handling high-dimensional data efficiently*" considers at most 64 dimensions. In contrast we need to consider data sets with dimensionalities in the thousands, and at least some of these data sets are not amiable to dimensionality reduction.

At least some anecdotal evidence suggests that many real world data sets are often not indexed. For example Protopapas et al. [58] have billions of star light curves (time series measuring the intensity of a star) which they mine for outliers, however the data is not indexed due to its high dimensionality and the relative cost and difficulty of building an index for a data set that may only be queried a few times a year. Additional examples include NASA Ames, which has archived flight telemetry for one million domestic commercial flights. Dr. Srivastava, the leader of the Intelligent Systems Division notes that linear scans on this data set take two weeks, but the size at dimensionality of the data makes indexing untenable even with state of the art techniques [66]. Given the above, we feel that our assumption that the larger of the data sets is not indexed is a reasonable assumption reflected by many real word scenarios.

The main contribution of this chapter is to show that joins can be cast as *anytime algorithms*. As illustrated in Figure 2.1 anytime algorithms are algorithms that trade execution time for quality of results [32]. In particular, after some small amount of setup-time an anytime algorithm always has a best-so-far answer available, and the quality of the answer improves with execution time.

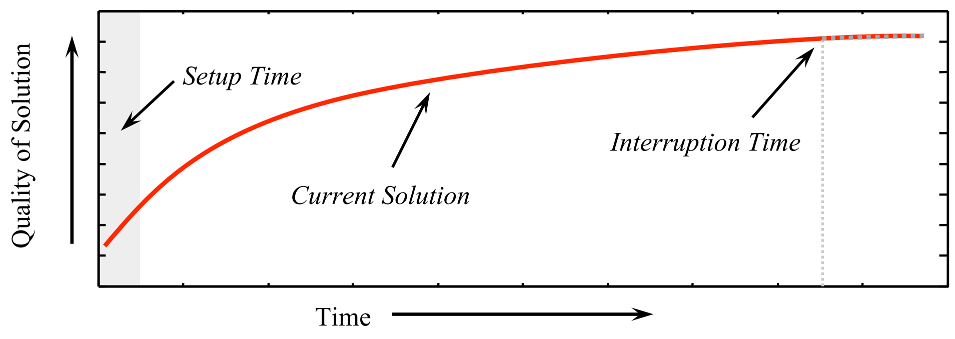


Figure 2.1: An abstract illustration of an anytime algorithm. Note that the quality of the solution keeps improving until the time the algorithm is interrupted by the user

Zilberstein and Russell [99] give a number of desirable properties of anytime algorithms:

- **Interruptability:** After some small amount of setup time, the algorithm can be stopped at anytime and provide an answer.
- **Monotonicity:** the quality of the result is a non-decreasing function of the computation time.
- **Measurable quality:** the quality of an approximate result can be determined.
- **Diminishing returns:** the improvement in solution quality is largest at the early stages of computation, and diminishes over time.
- **Preemptability:** the algorithm can be suspended and resumed with minimal overhead.

As we shall see, we can frame an approximate asymmetric join as any anytime algorithm to achieve all these goals. Due to their applicability to real world problems, there has been increasing interest in anytime algorithms. For example some recent works such as [73] and [82] show how to frame nearest neighbor classification as an anytime algorithm and that Top-k queries can also be calculated in an anytime framework, and [88] shows how Bayesian network structure can be learned in an anytime setting.

2.2 The Asymmetric Approximate Anytime Join

For concreteness of the exposition we start by formalizing the notion of the *All Nearest Neighbor* query.

Definition 3. *All-Nearest Neighbor query.* Given two sets of objects A and B, an *All-Nearest Neighbor query*, denoted as *ANN_query*(A, B), finds for each object $a_i \in A$ an object $b_j \in B$ such that for all other objects $b_k \in B, k \neq j,\ \textsc{Dist}(b_j, a_i) \leq\ \textsc{Dist}(b_k, a_i)$.

Note that in general *ANN_query*$(A, B) \neq$ *ANN_query*(B, A).

We will record the mapping from A to B with a data structure called $mapping$. We can discover the index of the object in B that a_i maps to by accessing $mapping[i].pointer$, and we can discover the distance from a_i to this object with $mapping[i].dist$.

It is useful for evaluating anytime or approximate joins to consider a global measure of how close all the objects in A are to their (current) nearest neighbor. We call this Q, the quality of the join and we measure it as: $Q = \sum_{i=1}^{|A|} mapping[i].dist$.

Given this notation we can show the brute force nested loop algorithm for the *All Nearest Neighbor* (ANN) algorithm in Algorithm 1.

Algorithm 1 $\textsc{BruteForceJoin}(A, B)$

```
1: for $i \leftarrow 1$ to $|A|$ do
2:     $mapping[i].dist \leftarrow \textsc{Dist}(a_i, b_1)$
3:     $mapping[i].pointer \leftarrow 1$
4:     for $j \leftarrow 2$ to $|B|$ do
5:         $d \leftarrow \textsc{Dist}(a_i, b_j)$
6:         if $d < mapping[i].dist$ then
7:             $mapping[i].dist \leftarrow d$
8:             $mapping[i].pointer \leftarrow j$
9:         end if
10:    end for
11: end for
12: return $mapping$
```

Note that lines 1 to 3 are not part of the classic ANN algorithm. They simply map everything in A to the first item in B. However, once this step has been completed, we can continuously measure Q as the algorithm progresses, a fact that will be useful when we consider anytime versions of the algorithm below.

In Algorithm 1 we have A in the outer loop and B in the inner loop, a situation we denote as *BF_AoB* (Brute Force, A over B). We could, however, reverse this situation to have B in the outer loop and A in the inner loop. For a batch algorithm this makes no difference to either the efficiency or outcome of the algorithm. Yet, as we shall see, it can make a big difference when we cast the algorithm in an anytime framework.

Before considering our approach in the next section, we will introduce one more idealized strawman that we can compare to. Both flavors of the algorithms discussed above take a single item from one of the two data sets to be joined and scan it completely against the other data set before considering the next item. Recall, however, that the desirable property of *diminishing returns* would like us to attempt to minimize Q as quickly as possible. For example, assume that we must scan B in sequential order, but we can choose which objects in A to scan across B, and furthermore we can start and stop with different objects from A at any point. Suppose that at a particular point in the algorithms execution we could either scan a_1 across five items in B to reduce its error from 10.0 to 2.5, or we could scan a_2 across ten items in B to reduce its error from 11.0 to 1.0. The former would give us a $rate$ of error reduction of 1.5 = (10.0 2.5) / 5, while the latter choice would give us a *rate* of error reduction of 1 = (11.0 1.0) / 10. In this case, the former choice gives us the faster rate of error reduction and we should choose it. Imagine that we do this for *every* object in A, at *every* step of the algorithm. This would give us the fastest possible rate of error reduction for a join. Of course, we cannot actually compute this on the fly, we have no way of knowing the best choices without actually doing all the calculations. However, we can compute the best choices offline and imagine that such an algorithm exists. Fittingly, we call such an algorithm magic, and can use it as an upper bound for the improvement we can make with our algorithms.

2.2.1 Augmented Orchard's Algorithm

While the statement of the problem at hand explicitly prohibits us from indexing the larger data set B, nothing prevents us from indexing the smaller set A. If A is indexed, then we can simply sequentially pull objects b_j from B and quickly locate those objects in A that are nearer to b_j than to their current best so far.

While there is a plethora of choices for indexing data set A, there are several special considerations which constrain and guide our choice. The overarching requirement is *generality*, we want to apply AAAJ's to very diverse data sets, some of which may be very high dimensional, and some of which, for example strings under the edit distance, may not be amiable to spatial access methods. With these considerations in mind we decided to use a slightly modified version of Orchard's algorithm [55], which requires only that the distance measure used be a metric. Orchard's algorithm is not commonly used because its quadratic space complexity is simply untenable for many applications. However, for most of the practical applications we consider here this is not an issue. For example, assume that we can record both the distance between two objects and each of the values in the real valued vectors with the same number of bits. Further we assume have a feature vector length of n per object. Given this, the data set A itself requires $\mathcal{O}(|A|n)$ space, and Orchard index requires $\mathcal{O}(|A|^2)$ space. Concretely, for the Butterfly example in Section 2.3.3 the space overhead amounts to approximately 0.04$\%$, and for the light curve example the overhead is approximately 2.8$\%$. Because Orchard's algorithm is not widely known we will briefly review it below. We refer the interested reader to [55] for a more detailed treatment.

A Review of Orchard's Algorithm

The basic idea of Orchard's algorithm is to quickly prune non-nearest neighbors based on the triangular inequality. In the preprocessing stage the distance between each two items in the data set A is calculated. As shown in Figure 2.2.*Left*, given a query q, if the distance between q and an item a_i in data set A is already known as d, then those items in data set A whose distance is larger than $2d$ to a_i can be pruned. The distance between these items and q is guaranteed to be larger than d which directly follows the triangular inequality. Therefore, none of them can become the nearest neighbor of q.

Specifically, for every object $a_i \in A$, Orchard's algorithm creates a list $P[a_i].pointer$ which contains pointers to all other objects in A sorted by distance to a_i. I.e., the list stores the index, denoted as $P[a_i].pointer[k]$, of the kth nearest neighbor to a_i within data set A, and the distance $P[a_i].dist[k]$ between a_i and this neighbor. This simple array of lists is all that we require for fast nearest neighbor search in A. The algorithm 2, begins by choosing some random element in A as the tentative nearest neighbor $a_{nn}.loc$, and calculating the distance

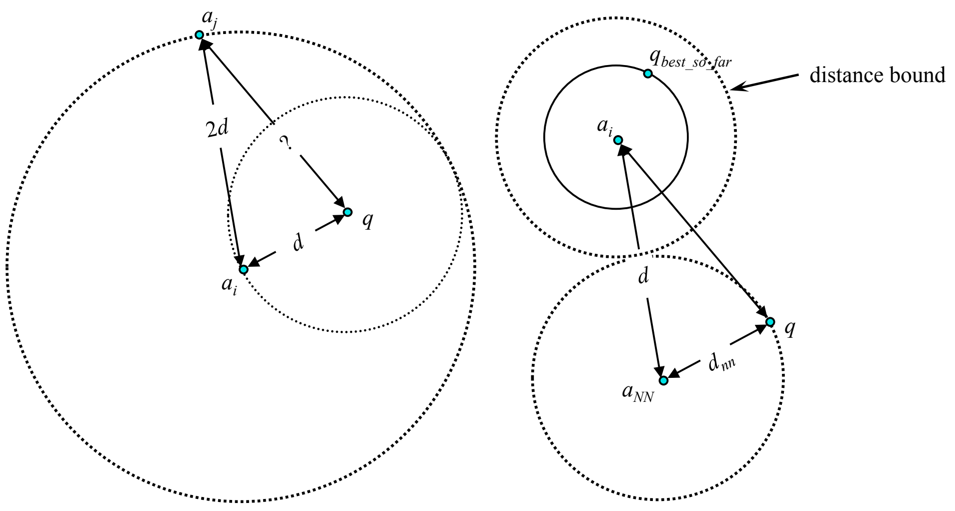


Figure 2.2: *Left*) The triangular inequality is used in Orchard's Algorithm to prune the items in A that cannot possibly be the nearest neighbor of query q. *Right*) Similarly, the triangular inequality is used in Augmented Orchard's Algorithm to prune the items in A that are certain to have a better best-so-far match than the current query q

$nn.dist$ between the query q and that object (lines 1 and 2). Thereafter, the algorithm inspects the objects in list $P[a_{nn}.loc]$ in ascending order until one of three things happen. The end of the list is reached, or the next object on the list has value that is more than twice the current $nn.dist$ (line 4). In either circumstance the algorithm terminates. The third possibility is that the item in the list is closer to the query than the current tentative nearest neighbor (line 7). In that case the algorithm simply jumps to the head of the list associated with this new nearest neighbor to the query and continues from there (lines 8 to 10).

We can see that a great advantage of this algorithm is its simplicity; it can be implemented in a few dozen lines of code. Another important advantage for our purposes is its generality. The function DIST can be any distance metric function. In contrast, most algorithms use to index data in order to speed up joins explicitly exploit properties that may not exist in all data

Algorithm 2 ORCHARDS(A, q)

```
1: $nn.loc \leftarrow$ random integer between 1 and $|A|$
2: $nn.dist \leftarrow$ DIST$(a_{nn.loc}, q)$
3: $index \leftarrow 1$
4: while $P[a_{nn.loc}].dist[index] < 2 \times nn.dist$ and $index < |A|$ do
5:     $node \leftarrow P[a_{nn.loc}].pointer[index]$
6:     $d \leftarrow$ DIST$(a_{node}, q)$
7:     if $d < nn.dist$ then
8:         $nn.dist \leftarrow d$
9:         $nn.loc \leftarrow node$
10:        $index \leftarrow 1$
11:    else
12:        $index \leftarrow index + 1$
13:    end if
14: end while
15: return $nn$
```

sets. For example they may require the data to be real valued [13, 97], or may be unable to handle mixed data types.

Note that this algorithm finds the one nearest neighbor to a query q. However, recall that we have a data structure called mapping which records the nearest item to each object in A that we have seen thus far. So we need to adapt the classic Orchard's algorithm to update not only the nearest neighbor to q in A (if appropriate) but all objects ai such that DIST$(a_i, q) < mapping[i].dist$. We consider a method to adapt the algorithm below.

Augmented Orchard's Algorithm

The results together with the mapping structure built for data set A can be utilized into an extended scheme for approximate joins, which exhibits the properties of an anytime join algorithm too. We term this scheme *Augmented Orchard's Algorithm* (see Algorithm 3).

Algorithm 3 AUGMENTORCHARDS(A, q)

```
1: for $i \leftarrow 1$ to $|A|$ do
2:   Build $P[a_i]$
3:   $mapping[i].dist \leftarrow \textsc{Dist}(a_i, b_1)$
4:   $mapping[i].pointer \leftarrow 1$
5: end for
6: for $i \leftarrow 2$ to $|B|$ do
7:   $nn \leftarrow \textsc{Orchards}(A, b_j)$
8:   if $nn.dist < mapping[nn.loc].dist$ then
9:     $mapping[nn.loc].dist \leftarrow nn.dist$
10:    $mapping[nn.loc].pointer \leftarrow j$
11:  end if
12:  for $index \leftarrow 1$ to $|A| - 1$ do
13:    $node \leftarrow P[a_{nn.loc}].pointer[index]$
14:    $bound \leftarrow |nn.dist - P[a_{nn.loc}].dist[index]|$
15:    if $mapping[node].dist > bound$ then
16:      $d \leftarrow \textsc{Dist}(a_{node}, b_j)$
17:      if $d < mapping[node].dist$ then
18:        $mapping[node].dist \leftarrow d$
19:        $mapping[node].pointer \leftarrow j$
20:      end if
21:    end if
22:  end for
23: end for
24: return $mapping$
```

The algorithm starts with an initialization step (lines 1 to 4) during which the table of sorted neighborhood lists $P[a_i]$ is computed. At this point all elements in A are also mapped to the first element in the larger data set B. To provide for the interruptability property of anytime algorithms, we adopt a B_over_A mapping between the elements of the two sets. The approximate join proceeds as follows: Data set B is sequentially scanned from the second element on, improving the best-so-far match for some of the elements in data set A. For this purpose, we first invoke the classic Orchard's algorithm which finds the nearest neighbor $a_{nn.loc}$ to the current query b_j and also computes the distance between them (i.e. $nn.dist$, line

7). If the query improves on the best-so-far match of $a_{nn.loc}$, then we update the elements nearest neighbor as well as its distance (lines 9 and 10).

In addition to improving the best match for $a_{nn.loc}$, the query example b_j may also turn out to be the best neighbor observed up to this point for some of the other elements in A. However, we do not need to check the distances between all a_i and the query b_j. Instead, we can use the pre-computed distance list $P[a_{nn.loc}]$ and once again apply the triangle inequality to prune many of the distance computations. Concretely, we need to compute the actual distance between b_j and any $a_i \in A$ only if the following holds: $mapping[i].dist > |nn.dist - \textsc{Dist}(a_{nn.loc}, a_i)|$. Figure 2.2.***Right*** gives the intuition behind this pruning criterion. The "distance bound" represents the right term of the above inequality. If it is larger or equal to the bound obtained with the best-so-far query to a_i, then the new query cannot improve the current best match of a_i. Note that all terms in the inequality are already computed as demonstrated on line 14 of the algorithm, so no distance computations are performed for elements that fail the triangle inequality. Finally, it is important to point out that a nave implementation of both Orchard's algorithm and our extension may attempt to compare the query and the same element ai more than once, so we keep a temporary structure that stores all elements compared thus far and the corresponding distances. The memory overhead for bookkeeping the structure is negligible and at the same time allows us to avoid multiple redundant computations.

We conclude this section by sketching the intuition behind the pruning performance of the Augmented Orchard's Algorithm. Choosing a *vantage* (center) point is a common technique in many algorithms that utilize the triangular inequality, such as [8, 65, 95] etc. In all of these works one or more center points are selected and the distance between them and all other elements in the data set are computed. These distances are subsequently used together with the triangular inequality to efficiently find the nearest neighbors for incoming queries. As to what points constitute good centers, i.e. centers with good pruning abilities, depends on several factors [65], e.g. data set distribution, position of the centers among the other points, as well as the position of the query in the given data space. The two common strategies in selecting the centers are random selection [8, 65, 95] or selection based on some heuristic, e.g. maximal remoteness from any possible cluster center [65]. In the Augmented Orchard's Algorithm we follow a different approach. Namely, for every query b_j we select a different center point $a_{nn.loc}$. This is possible as in the initialization step we have computed all pairwise distances among the elements in A. The algorithm also heavily relies on the fact that the $P[a_i]$ lists are sorted in ascending order. The assumption is that b_j may impact the best-so-far match only for elements that are close to it, i.e. its nearest neighbor $a_{nn.loc}$ and the closest elements to that neighbor which are the first elements in the $P[a_{nn.loc}]$ list. All remaining elements in this list are eventually pruned by the triangular inequality on line 15 of the algorithm.

2.3 Experiments and Case Studies

In this section we consider examples of applications of AAAJ for several diverse domains and applications. Note that in every case the experiments are completely reproducible, the data sets may be found at [90].

2.3.1 A Sanity Check on Random Walk Data

We begin with a simple sanity check on synthetic data to normalize our expectations. We constructed a data set of one thousand random walk time series to act as database A, and one million random walk time series to act as database B. All time series are of length 128.

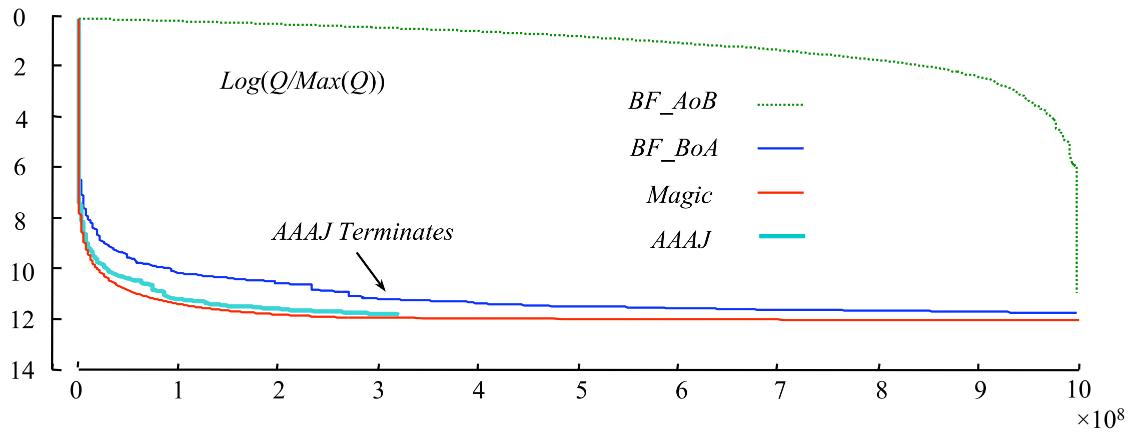


Figure 2.3: The log of Q as a function of the number of calls to the Euclidean distance

Figure 2.3 shows the rate at which the measure Q decreases as a function of the number of Euclidean distance comparisons made. Note that this figure does include the setup time for AAAJ to build the index, but this time is so small relative to the overall time that it cannot be detected in the figure (It can just be seen in Figure 2.5, where $|B|$ is much smaller relative to $|A|$).

We can see that not only does AAAJ terminate 3 times faster than the other algorithms, but the rate at which it minimizes the error is much greater, especially in the beginning. This of course is the desirable *diminishing returns* property for anytime algorithms. Note in particular that the rate of error reduction for AAAJ is very close to *magic*, which is the optimal algorithm possible, given the assumptions stated for it.

2.3.2 Anytime Classification of Batched Data

There has been recent interest in framing the classification problem as an anytime algorithm [73, 88]. The intuition is that in many cases we must classify data without knowing in advance how much time we have available. The AAAJ algorithm allows us to consider an interesting variation of the problem which to our knowledge has not been addressed before. Suppose that instead of been given*one* instance to classify under unknown computational resources we are given a *collection* of unlabeled instances. In this circumstance we can trivially use AAAJ as an anytime classifier.

Suppose we are given k objects to classify, and we intend to classify them using the One Nearest Neighbor (1NN) algorithm with training set B. However, it is possible that we may be interrupted at any time (Paper [73] discusses several real world application domains where this may happen). In such a scenario it would make little sense to use the classic brute force algorithm 1 to classify the data. This algorithm might perfectly classify the first 20%

Table 2.1: An abridged trace of the current classification of the 30 characters of the string "SGT PEPPERS LONELY HEARTS CLUB BAND" vs. the number of distance calculations made

TTT	TTTTTTT	TTTTTT	TTTTTT	TTTT	TTTT	30
TTT	TTITTTT	TTTTTT	TTTTTT	TTTT	TTTT	436
TTT	TTIITTT	TTTTTT	TTTTTT	TTTT	TTTT	437
: : :	: : : : : : :	: : : : : :	: : : : : :	: : :	: : :	: : :
SBT	PSPPERS	LONEOY	REARTF	CLUB	BANG	21166
SBT	PSPPERS	LONEGY	REARTF	CLUB	BANG	**22350***
SBT	PSPPERS	LONEGY	REARTF	CLUN	BANG	23396
: : :	: : : : : : :	: : : : : :	: : : : : :	: : :	: : :	: : :
SGT	PEPPERS	LONELY	HEARTZ	CLUB	HAND	182405
SGT	PEPPERS	LONELY	HEARTS	CLUB	HAND	305794

of the k objects before interruption, but then its overall accuracy, including the default rate on the 80% of the data it could not get to in time, would be very poor. Such a situation can arise in several circumstances, for example robot location algorithms are often based on classifying multiple "clues" about location and combining them into a single best guess, and robot navigation algorithms are typically cast as anytime algorithms [32].

The *Letter* data set has 26 class labels corresponding to the letters from `A` to `Z`. The task here is to recognize letters given 16 features extracted from image data. There are a total of 20,000 objects in the data set. We decided on a 30-letter familiar phrase, and randomly extracted 30 objects from the training set to spell that phrase. We ran AAAJ with the 30 letters corresponding to A, and the 19,970 letters remaining in the training set corresponding to B. Table 2.1 shows a trace of the classification results as the algorithm progresses.

After the first 30 distance comparisons (corresponding to lines 1 to 4 of Algorithm 3), every letter in our phrase points to the first letter of data set B, which happens to be "`T`".

Thereafter, whenever an item in A is joined with a new object in B, its class label is updated. When the algorithm terminates after 305,794 distance calculations, the target phrase is obvious in spite of one misspelling ("HAND" instead of "BAND"). Note, however, that after the AAAJ algorithm has performed just 7.3% of its calculations, its string "sbt psppers lonegy reartf club bang" is close enough to the correct phrase to be autocorrected by Google, or to be recognized by 9 out of 10 (western educated) professors at UCR. Figure 2.4 gives some intuition as to why we can correctly identify the phrase after performing only a small fraction of the calculations. We can see that AAAJ behaves like an ideal anytime algorithms, deriving the greatest changes in the early part of the algorithms run.

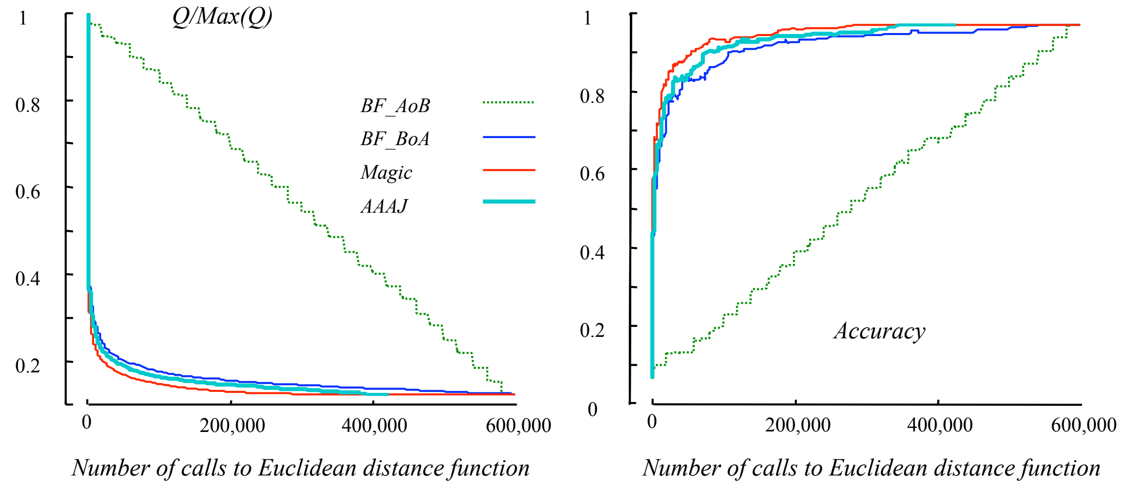


Figure 2.4: *Left*) The normalized value of Q as a function of the number of calls to the Euclidean distance. *Right*) The accuracy of classification, averaged over the 30 unknown letters, as a function of the number of calls to the Euclidean distance

In this experiment the improvement of AAAJ over BF_BoA is clear but not very large. AAAJ terminates with 433,201 calculations (including the setup time for the Orchard's algorithms), but the other algorithms only take 598,801. However, this is because the size of

A was a mere 30, as we shall see in the next experiment the improvement becomes more significant for larger data sets.

In Figure 2.5 we see the results of a similar experiment, this time the data is star light curves (discussed in more detail in Section 2.3.4), with $|A|$ = 500 and $|B|$ = 8,736. Each time series was of length 1,024.

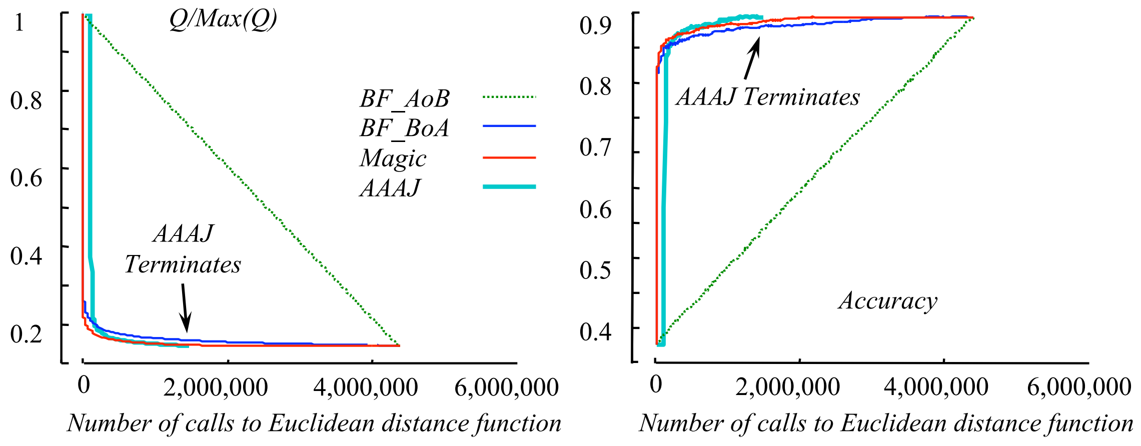
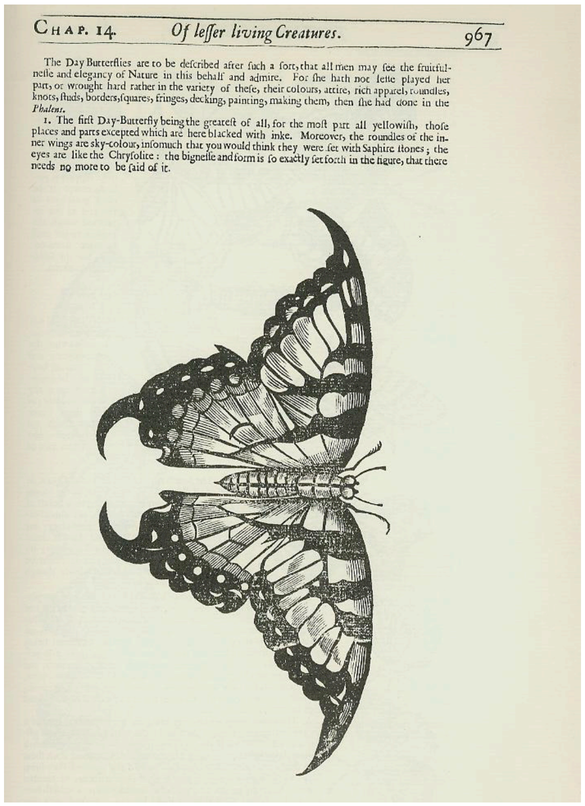


Figure 2.5: *Left*) The normalized value of Q as a function of the number of calls to the Euclidean distance. *Right*) The accuracy of classification, averaged over the 1,000 star light curves, as a function of the number of calls to the Euclidean distance

At this scale it is difficult to see the advantage of the proposed algorithm. However, consider this. After performing 1,000,000 distance calculations the AAAJ algorithm has reduced the normalized value of Q to 0.1640. On the other hand, the BF_BoA algorithm must perform 4,342,001 distance calculations to reduce the normalized Q to the same level. The following observation is worth noting. While all 4 algorithms have identical classification accuracy when they terminate (by definition), the accuracy of AAAJ is actually slightly higher while it is only 80% completed. This is not an error, it is simply the case that this particular run happened to have a few mislabel objects towards the end of data set B, and those objects cause several objects in $|A|$ to be mislabeled.

2.3.3 Historical Manuscript Annotation

Recent initiatives like the Million Book Project and Google Print Library Project have already archived several million books in digital format, and within a few years a significant fraction of worlds books will be online [34]. While the majority of the data will naturally be text, there will also be tens of millions of pages of images. Consider as an example the lithograph of the "Day" butterfly shown in Figure 2.6.



CHAP. 14. *Of leſſer living Creatures.* 967

The Day Butterflies are to be deſcribed after ſuch a ſort, that all men may ſee the fruitful-neſſe and elegancy of Nature in this behalf and admire. For ſhe hath not leſſe played her part, or wrought hard rather in the variety of theſe, their colours, attire, rich apparel, roundles, knots, ſtuds, borders, ſquares, fringes, decking, painting, making them, then ſhe had done in the *Phalens.*

1. The firſt Day-Butterfly being the greateſt of all, for the moſt part all yellowiſh, thoſe places and parts excepted which are here blacked with inke. Moreover, the roundles of the inner wings are sky-colour, inſomuch that you would think they were ſet with Saphire ſtones; the eyes are like the Chryſolite: the bigneſſe and form is ſo exactly ſet forth in the figure, that there needs no more to be ſaid of it.

Figure 2.6: Page 967 of "The Theatre of Insects; or, Lesser living Creatures..." [54], published in 1658, showing the "Day" butterfly

The image was published in 1658, therefore predating the binomial nomenclature of Linnaeus by almost a century. So a modern reader cannot simply find the butterfly species by looking it up on the web or reference book[2]. However, the shape is well defined, and in spite of being in black and white the author's helpful notes tell us that it is "*for the most part yellowish, those places and parts excepted which are here blacked with inke*".

With a combination of information retrieval techniques and human insight we can attempt to discover the true identify of the species illustrated in the book. Note that a query to Google image search for "Butterfly" returns approximately 4.73 million images (on 9/12/2007). A large fraction of these are not images of butterflies, but of butterfly valves, swimmers doing the butterfly stroke, greeting cards etc. Furthermore, of the images actually depicting butterflies, many depict them in complex scenes that are not amiable to state-of-the-art segmentation algorithms. Nevertheless, a surprising fraction of the images can be automatically segmented with simple off the shelf tools (we use Matlab's standard image processing tools with default parameters). As illustrated in Figure 2.7, for those images which can be segmented into a single shape, we can convert the shapes into a one dimensional representation and compare them to our image of interest [44].

While the above considers just one image, there are many online collections of natural history which have hundreds or thousands of such images to annotate. For example, Albertus Seba's 18th century masterwork [64] has more than 500 drawings of butterflies and moths,

[2]There is a "Day moth" (*Apina callisto*), however it looks nothing like the insect in question.

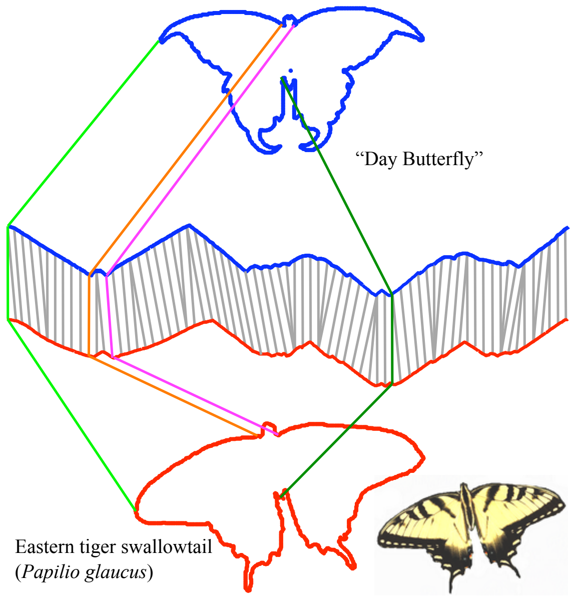


Figure 2.7: We can compare two shapes by converting them to one-dimensional representations and using an appropriate distance measure, in this case Dynamic Time Warping

and there are at least twenty 17th and 18th century works of similar magnitude (see [68], page 86 for a partial list).

We believe that annotating such collections is a perfect application of AAAJ. We have a collection of a few hundred objects, that we want to link to a subset of a huge data set (the images on the internet). An exhaustive join is not tenable, nor is it needed. We dont need to find the exact nearest match to each butterfly, just one that is *near-enough* to likely be the same or related species.

Once we have a near-enough neighbor we can use any meta tags or text surrounding the neighbor to annotate our unknown historical images. The basic framework for annotation of historical manuscripts in this way has been the subject of extensive study by several groups, most notably at the University of Padova[3] [1]. However, the work assumes unlimited computation resources and a high degree of human intervention. Here we attempt to show that AAAJ could allow real time exploration of historical manuscripts. We can simply point to an image and right-click and choose *annotation-search*. At this point the user can provide some text clues to seed the search. In this case we assume the word butterfly is provide to the system, which then issues a Google image search.

While there are still some image processing and web crawling issues to be resolved we consider this problem an ideal one to test the utility of AAAJ, so we conducted the following experiment.

Data set A consists of 35 drawings of butterflies. They are taken from manuscripts as old as 1783 and as recent as 1968. In every case we know the correct species label because it either appears in the text (in some cases in German or French which we had translated by bilingual entomologists) or because the entomologist Dr. Agenor Mafra-Neto was able to unambiguously identify it for us. Data B consist of 35 real photographs of corresponding insects, plus 691 other images of butterflies (including some duplicates of the above) plus 44,215 random shapes. The random shapes come from combining all publicly available shape data sets, and include images of fish, leafs, bones, fruit, chicken parts, arrowheads, tools, algae, and trademark logos. Both data sets are randomly shuffled.

[3]The work at the University of Padova and related projects consider mostly religious texts, in particular illuminated manuscripts. We know of no other research effort that considers historical scientific texts.

While we used Dynamic Time Warping as the distance measure in Figure 2.7, it is increasingly understood that for large data sets the amount of warping allowed should be reduced [60, 66]. For large enough data sets the amount of warping allowed should be zero, which is simply the special case of Euclidean distance.

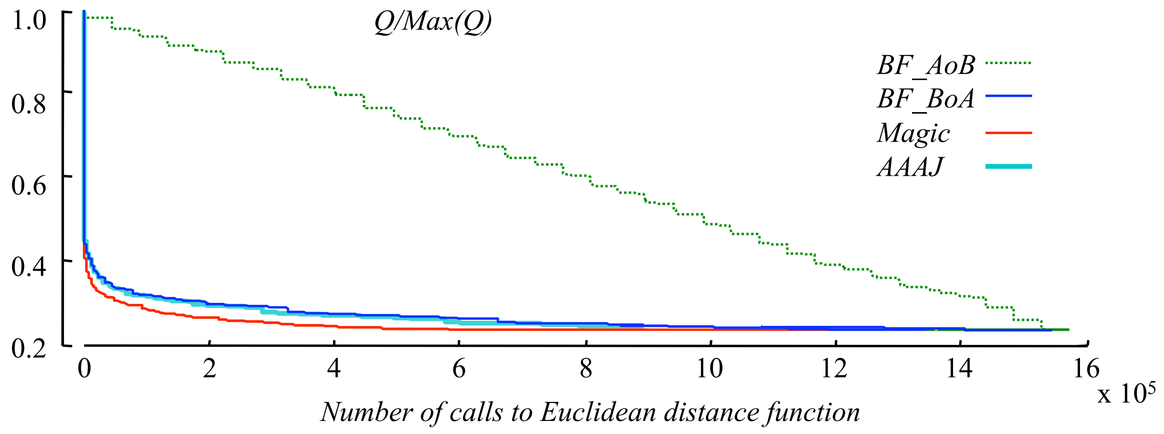


Figure 2.8: The normalized value of Q as a function of the number of calls to the Euclidean distance

Figure 2.8 shows the rate of reduction of Q on the butterfly data set.

While we can see that the rate of error reduction is very fast, it is difficult to get context for the utility of AAAJ from this figure. Consider therefore Figure 2.9. Here we show in the leftmost column the original historical manuscripts (note that at this resolution they are very difficult to differentiate from photographs). After 10% of the eventual time had passed we took a snapshot of the current nearest neighbors of A. Figure 2.9.*Center* shows the top eight, as measured by $mapping[i].dist$. In the rightmost column we show the final result (which is the same for all algorithms). It is difficult to assign a metric to these results, since precision/recall or accuracy cannot easily be modified to give partial credit for discovering a

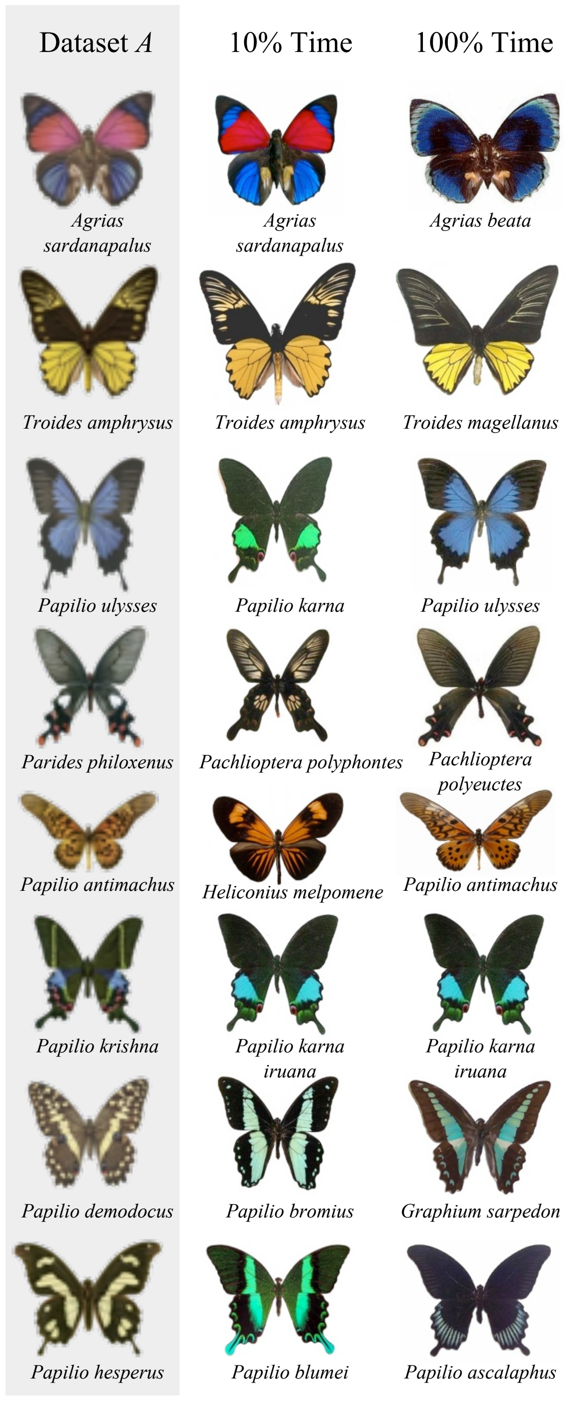


Figure 2.9: *Left column*) Eight sample images from the data set, a collection of butterfly images from historical manuscripts. *Center column*) The best matching images after AAAJ has seen 10% of the data. *Right column*) The best matching images after AAAJ has seen all

related species (or a mimic, or convergently evolved species). However, we can intuitively see that much of the utility of conducting this join is captured in the first 10% of the time.

2.3.4 Anytime Anomaly Detection

In certain domains data observations come gradually over time and are subsequently accumulated in large databases. In some cases the input data rate may be very high or data may come from multiple sources, which makes it hard to guarantee the quality of the stored observations. Still we may need to have some automatic way to efficiently detect whether incoming observations are normal or represent severe outliers, with respect to the data already in the database. A simple means to achieve this is to apply the nearest neighbor rule and to find out whether there is some similar observation stored so far. Depending on the accumulated data set size and the input rate, processing all incoming observation in online manner with the above simple procedure may still be intractable. At the same time it is often undesirable and, as we demonstrate below, unnecessary to wait for the whole process to finish and then run offline some data cleaning procedure. What we can use instead is an anytime method that computes the approximate matches to small subsets of elements in a batch mode, before the next subset of observation has arrived. Below we demonstrate how our AAAJ algorithm can help us with this.

As a concrete motivating example consider star light curves, also known as variable stars. The American Association of Variable Star Observers has a database of over 10.5 million variable star brightness measurements going back over ninety years. Over 400,000 new variable star brightness measurements are added to the database every year by over 700 observers from all over the world [51, 58]. Many of the objects added to the database have errors. The sources of these errors range from human error, to malfunctioning observational equipment, to faults in punch card readers (for attempts to archive decade old observations) [51]. An obvious way to check for errors is to *join* the new tentative collection (A) to the existing database (B). If any objects in A are unusually far from their nearest neighbor in B then we can single them out for closer inspection. The only problem with this idea is that a full join on the 10.5 million object database takes much longer than a day. As we shall see, doing this with AAAJ allows us to spot potentially anomalous records much earlier.

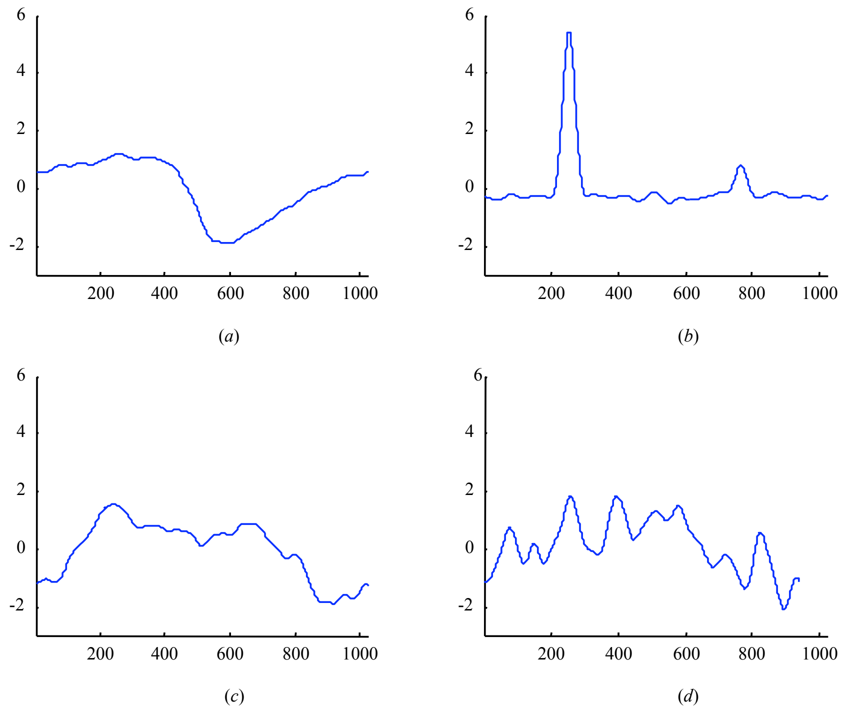


Figure 2.10: Various light curve vectors in the data set. (a) and (b) are normal ones. (c) and (d) are outliers

For this experiment we use a collection of star light curves donated by Dr. Pavlos Protopapas of Time Series Center at Harvard"s Initiative for Innovative Computing. The data set that we have obtained contains around 9,000 such light curves from different star classes. Each example is a time series of size 1,024. Figure 2.10 shows four different light curve shapes in the data. Suppose that a domain expert considers as outlier examples whose nearest neighbor distance is more than a predefined threshold of t standard deviations away from the average nearest neighbor distance [58]. For the light curve data we set $t = 5$ standard deviations, which captures most of the anomalies as annotated by the expert astronomers. Examples (c) and (d) in Figure 2.10 , show two such light curves, while (a) and (b) are less than 5 standard deviations from their nearest neighbors and therefore are treated as normal.

Now assume that the light curve observations are recorded in batch mode 100 at a time. We can apply the AAAJ algorithm to every such set of incoming light curves (representing data set A in the algorithm) and find its approximate match within the database (set B) interrupting the procedure before the next subset of light curves arrives. If the maximal approximate nearest neighbor distance in A is more than the threshold of 5 standard deviations from the average nearest neighbor distance, then before combining A with the rest of the database we remove from it the outliers that fail the threshold.

To demonstrate the above procedure we conduct two experiments. In the first one data set A consists of 100 randomly selected examples among the examples annotated as normal. For the second experiment, we replace one of the normal examples in A (selected at random) with a known outlier. This outlier is not believed to be a recording or transcription error, but

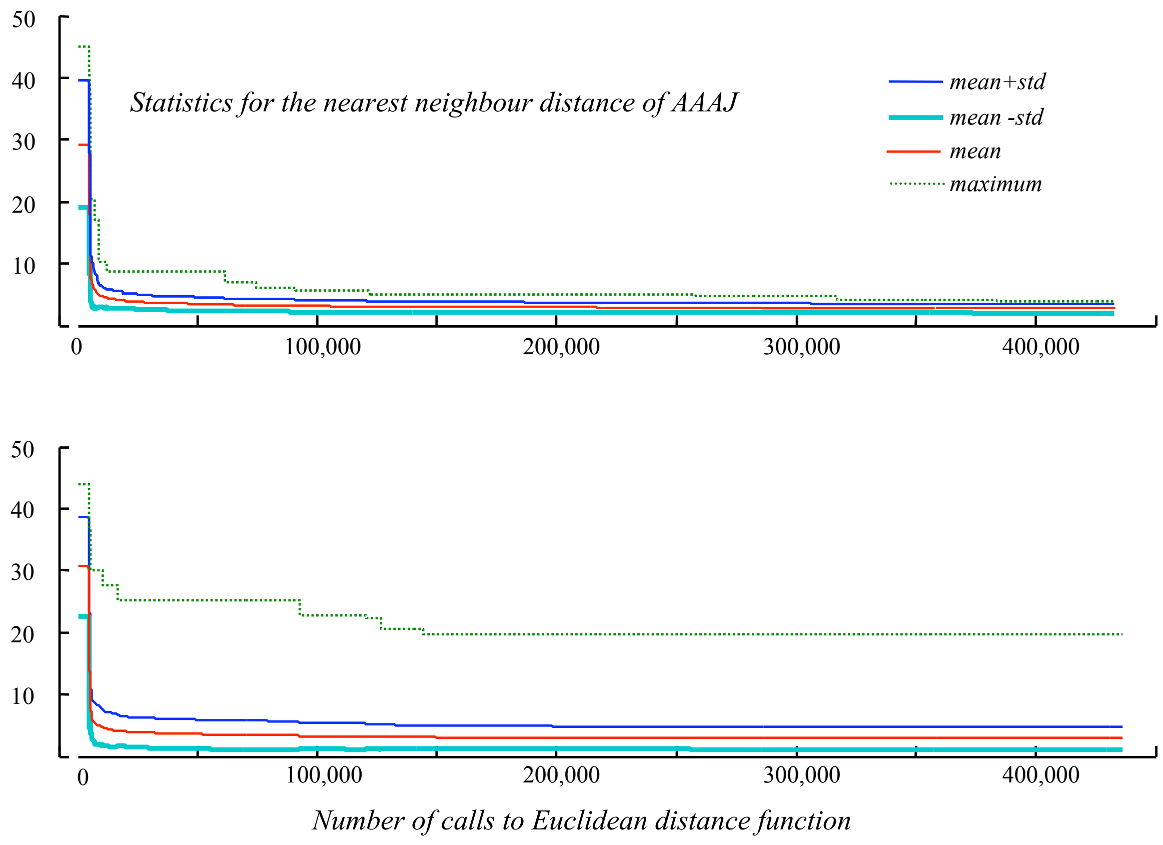


Figure 2.11: Average nearest neighbor distance with respect to the performed Euclidean distance comparisons. *Top*) A is composed only of normal elements. *Bottom*) A contains one outlier

an unusual astronomical object. Figure 2.11 shows the improvement in the average nearest neighbor distance in data set A, again as a function of the performed Euclidean distance computations. The maximal nearest neighbor distance is also presented in the graphs. The top graph corresponds to the experiment where data set A is composed only of normal elements, and the bottom one is for the data set with one outlier. The experiment shows how efficiently the AAAJ algorithm can detect the outliers. The average distance drops quickly after performing only a limited number of computations. After that the mean value stabilizes. At this point we can interrupt the algorithm and compute the deviation for each element in A. The elements that fail the expert provided threshold t are likely to fail it had we performed a full join too. In the second example we are able with high confidence to isolate the outlier in only a few thousand distance computations.

2.4 Conclusions

In this chapter we have argued that for many data mining problems an approximate join may be as useful as an exhaustive join. Given the difficulty of figuring out exactly how approximate is sufficient, we show that we can cast joins as anytime algorithms, in order to use as much computational resources as are available. We demonstrated the utility of our ideas with experiments on diverse domains and problems.

As we have shown for the domains considered we can extract most of the benefit of a join after doing only a small fraction of the work (the diminishing returns requirement of [99]), and we can use internal statistics (as in Section 2.3.4) to achieve the measurable quality requirement [99]. The interruptability and preemptability requirements are trivially true, we can stop the AAAJ algorithm at anytime and we only need to save the relatively tiny mapping data structure if we want to pick up where we stopped at a later date. Finally, with regards to Zilberstein and Russells requirements, the monotonicity property is clearly observed in Figures 2.3, 2.4, 2.5, 2.8, 2.11,

Chapter 3

Auto-cannibalistic and Anyspace Indexing Algorithms with Applications to Sensor Data Mining

This chapter demonstrates indexing algorithms, where the memory for indexing is a limiting or variant factor. Efficient indexing is at the heart of many data mining algorithms. As mentioned in Chapter 2, a simple and extremely effective algorithm for indexing under any metric space was introduced in 1991 by Orchard. Orchard's algorithm has not received much attention in the data mining and database community because of a fatal flaw; it requires quadratic space. In this chapter we show that we can produce a reduced version of Orchard's algorithm that requires much less space, but produces nearly identical speedup. We achieve this by casting the algorithm in an anyspace framework, allowing deployed applications to take as much of an index as their main memory/sensor can afford. As we shall demonstrate, this ability to create an *anyspace* algorithm also allows us to create *auto-cannibalistic* algorithms. Auto-cannibalistic algorithms are algorithms which initially require a certain amount of space to index or classify data, but if unexpected circumstances require them to store additional information, they can dynamically delete parts of themselves to make room for the new data. We demonstrate the utility of auto-cannibalistic algorithms in a fielded project on insect monitoring with low power sensors, and a simple autonomous robot application.

3.1 Introduction

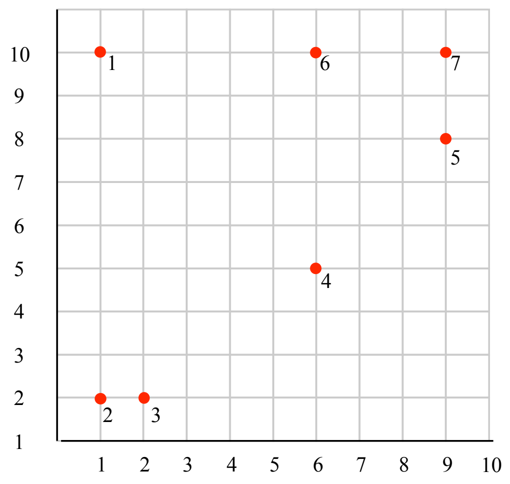
Efficient indexing is at the heart of many data mining algorithms. As we have introduced in Chapter 2, a simple and extremely effective algorithm for indexing under any metric space was proposed in 1991 by Orchard [55]. The algorithm is commonly known as Orchard's algorithm, however Charles Elkan points out that a nearly identical algorithm was proposed by Hodgson in 1988 [25, 35]. While Orchard's algorithm is currently used in some specialized domains such as vector quantization [30] and compression [96], it has not received much attention in the data mining and database community because of a fatal flaw; it requires quadratic space. In this chapter we show that we can produce a reduced version of Orchard's algorithm which requires much less space, but produces nearly identical speedup. We further show that we can cast our ideas in an *anyspace* framework [99], allowing deployed applications to take as much of an index as their main memory/sensor can afford. It is important to note that our reduced space algorithms produce *exactly* the same results as the full algorithm, they simply trade freeing up (a lot of) space for (very little) reduction in speed.

As we shall demonstrate, the ability to cast indexing as an anyspace algorithm allows us to create *auto-cannibalistic* algorithms. Auto-cannibalistic algorithms are algorithms which initially require a certain amount of space to index or classify data, but if unexpected circumstances require them to store additional information, they can dynamically delete ("eat") parts of themselves to make room for the new data. This allows the algorithm to be extremely efficient for a given memory allocation at the beginning of its life, and then gracefully degrade as it encounters outliers which it must store. We demonstrate the utility of auto-cannibalistic algorithms in a fielded project on insect monitoring with low power sensors and show that it can greatly extend the battery life of field deployed sensors.

The rest of the chapter is organized as follows. Section 3.2 reviews the classic Orchard's algorithm and Section 3.3 introduces our extensions and modifications. We conduct a detailed empirical evaluation in Section 3.4, and in Section 3.5 we consider two concrete applications, including one which is already being tested in the field. We conclude with a discussion of related and future work in Section 3.6.

3.2 Classic Orchard's Algorithm

In order to help the reader understand Orchard's algorithm, and our extensions and modifications to it, we will introduce a simple example data set in Figure 3.1 as a running example.



Dataset ***D***

	X	Y
a_1	1	10
a_2	1	2
a_3	2	2
a_4	6	5
a_5	9	8
a_6	6	10
a_7	9	10

Figure 3.1: A small data set A containing 7 items, used as a running example in this chapter

Table 3.1: Orchard's Algorithm Ranked Lists $P[A]$

Item	1st NN {dist}	2nd NN {dist}	3rd NN {dist}	4th NN {dist}	5th NN {dist}	6th NN {dist}
a_1	**6** {5.0}	**4** {7.1}	**2** {8.0}	**7** {8.0}	**3** {8.1}	**5** {8.2}
a_2	**3** {1.0}	**4** {5.8}	**1** {8.0}	**6** {9.4}	**5** {10.0}	**7** {11.3}
a_3	**2** {1.0}	**4** {5.0}	**1** {8.1}	**6** {8.9}	**5** {9.2}	**7** {10.6}
a_4	**5** {4.2}	**3** {5.0}	**6** {5.0}	**2** {5.8}	**7** {5.8}	**1** {7.1}
a_5	**7** {2.0}	**6** {3.6}	**4** {4.2}	**1** {8.2}	**3** {9.2}	**2** {10.0}
a_6	**7** {3.0}	**5** {3.6}	**1** {5.0}	**4** {5.0}	**3** {8.9}	**2** {9.4}
a_7	**5** {2.0}	**6** {3.0}	**4** {5.8}	**1** {8.0}	**3** {10.6}	**2** {11.3}

The preprocessing for Orchard's algorithm requires that we build for each item a_i in our data set A, a sorted list of its neighbors, annotated with the actual distances to ai in ascending order. We denote a sorted list for instance ai as $P[a_i]$, and the full set of these lists for the entire data set A as $P[A]$. Table 3.1 shows this data structure for our running example. Note that even for our small running example, the size of the ranked lists data structure is considerable, and would clearly be untenable for real world problems.

The basic intuition behind Orchard's algorithm is to prune non-nearest neighbors based on the triangular inequality [25]. Suppose we have a data set $A = \{a_1, a_2, \ldots, a_{|A|}\}$, in which we want to find the nearest neighbor of query q. Further suppose ai is the nearest neighbor found in A thus far. For any unseen element a_j, which will not be the nearest neighbor if:

$$\textsc{Dist}(a_j, q) \geq \textsc{Dist}(a_i, q) \tag{3.1}$$

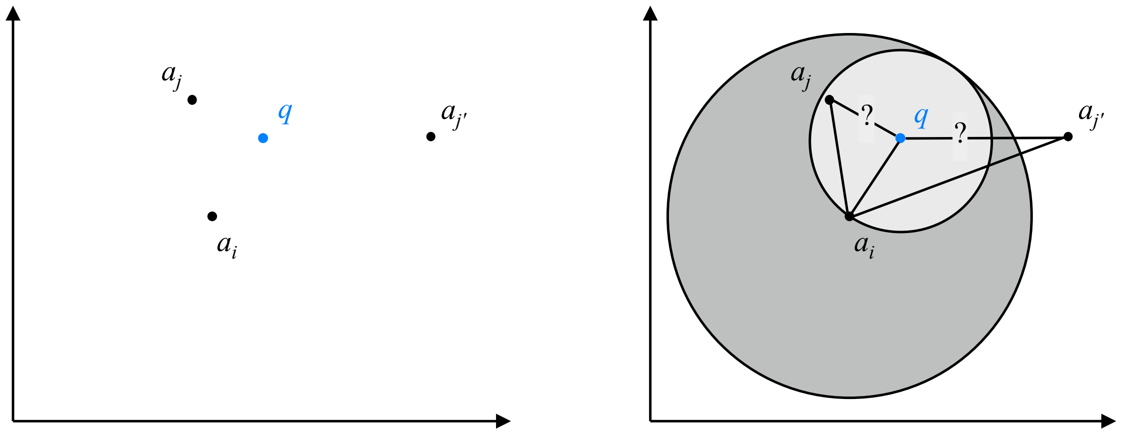


Figure 3.2: *Left*) Assume we know the pairwise distances between a_i, a_j and $a_{j'}$. A newly arrived query q must be answered. *Right*) After calculating the distance $\textsc{Dist}(q, a_i)$ we can conclude that items with a distance to a_i less than or equal to $2 \times \textsc{Dist}(q, a_i)$(i.e. the gray area) might be the nearest neighbor of q, but everything else, including $a_{j'}$ in this example, can be excluded from consideration

Given that we are dealing with a metric space, the principle of triangular inequality [25] illustrated in Figure 3.2 applies here:

$$\textsc{Dist}(a_i, a_j) \leq \textsc{Dist}(a_j, q) + \textsc{Dist}(a_i, q) \tag{3.2}$$

Combining Equation 3.1 and 3.2, we can derive the fact that if:

$$\textsc{Dist}(a_i, a_j) \geq 2 \times \textsc{Dist}(a_i, q) \tag{3.3}$$

is satisfied, a_j can be pruned, since it could not be the nearest neighbor. This is how a combination of the triangular inequality and the information stored in the ranked lists data structure $P[A]$, can allow us to prune some items a_j without the expense of calculating the actual distance between q and a_j.

The Orchard's algorithm has beem outlined in Algorithm 2. Note that as the algorithm is traversing down the lists, it may encounter an item more than once. To avoid redundant calculations, a record is kept of all items encountered thus far, and an $\mathcal{O}(n)$ hash is sufficient to check if we have already calculated the distance to that item.

As the reader can appreciate, the algorithm has the advantages of simplicity, and being completely parameter free. Furthermore as shown both here (cf. Section 3.4) and elsewhere [15, 70, 96], it is a very efficient indexing algorithm. However, while we typically would be willing to spare the quadratic time complexity to build the ranked lists data set, the quadratic space complexity has all but killed interest in this method. In the next section we show how we can achieve the same efficiency in a fraction of the space.

Table 3.2: Truncated Orchard's Algorithm

Item	1^{st} NN {dist}	2^{nd} NN {dist}	3^{rd} NN {dist}	4^{th} NN {dist}	5^{th} NN {dist}	6^{th} NN {dist}
a_1	**6** {5.0}	**4** {7.1}	**2** {8.0}	**7** {8.0}	**3** {8.1}	**5** {8.2}
a_2	**3** {1.0}	**4** {5.8}	**1** {8.0}	**6** {9.4}	**5** {10.0}	**7** {11.3}
a_3						
a_4	**5** {4.2}	**3** {5.0}	**6** {5.0}	**2** {5.8}	**7** {5.8}	**1** {7.1}
a_5	**7** {2.0}	**6** {3.6}	**4** {4.2}	**1** {8.2}	**3** {9.2}	**2** {10.0}
a_6	**7** {3.0}	**5** {3.6}	**1** {5.0}	**4** {5.0}	**3** {8.9}	**2** {9.4}
a_7	**5** {2.0}	**6** {3.0}	**4** {5.8}	**1** {8.0}	**3** {10.6}	**2** {11.3}

3.3 Anyspace Orchard's Algorithm

In this section we begin by giving the intuition behind our idea for an anyspace Orchard's algorithm, and then show concrete approaches to allow us to create such an algorithm.

3.3.1 Truncated Orchard's Algorithm

We motivate our ideas with a simple observation. Note that in Figure 3.1 the two data items a_2 and a_3 are very close together. As a result, their lists of nearest neighbors in Table 3.1 are almost identical. This is a redundancy; most queries that are efficiently pruned by a_2 would also be pruned efficiently by a_3, and vice-versa. We can exploit this redundancy by deleting one entire list, thus saving some space. For the moment let us assume we have randomly chosen to delete the list of a_3, as shown in Table 3.2.

Note that we cannot just delete the entire row. We have a small amount of bookkeeping to do. It is possible that as we are using the index and traversing down the one of the other lists, we will encounter a_3 and find that it is the *best so far*. We should therefore jump to the

Table 3.3: Highly Truncated Orchard's Algorithm

Item	1^{st} NN {dist}	2^{nd} NN {dist}	3^{rd} NN {dist}	4^{th} NN {dist}	5^{th} NN {dist}	6^{th} NN {dist}
a_1	*goto* a_6					
a_2	*goto* a_4					
a_3	*goto* a_2					
a_4	**5** {4.2}	**3** {5.0}	**6** {5.0}	**2** {5.8}	**7** {5.8}	**1** {7.1}
a_5	*goto* a_4					
a_6	*goto* a_7					
a_7	*goto* a_5					

list for a_3 and continue searching, however the list was deleted. To solve this problem we need to place a special "*goto*" pointer which tells the search algorithm that it should continue searching from a_2's list instead.

As the reader will have already guessed, we can iteratively use this idea to delete additional lists, thus saving more space. In the limit, we will be left with a single list as shown in Table 3.3.

Note that whatever algorithm we use to delete lists, we must make sure that we don not end up with cycles. For example if a_2's row says "*goto* a_3" and if a_3's row says "*goto* a_2" we have an infinite loop. Another important observation is that although we should not expect this highly truncated Orchard's algorithm to perform as well as the quadratic space version, we still have not (necessarily) degraded to sequential search. Queries that happen to land near a_4 will be answered quickly, no matter which random starting position we choose.

3.3.2 Anyspace Orchard's Algorithm

As framed in the previous section, we appear to have a solution to the quadratic complexity of Orchard's algorithm. We can simply work out how much memory is available for our application, and delete the necessary number of lists to make our Truncated Orchard's Algorithm fit. However, there may be situations where the amount of memory is variable (we discuss such applications in more detail in Section 3.5). In these applications we may find it useful to delete additional lists on-the-fly, as memory becomes more precious. For example, an autonomous robot could use a 90% Truncated Orchard's Algorithm to efficiently classify the items it sees as *friend*, *foe* or *unknown*. For both the *friend* and *foe* categories, it suffices to count how many it encountered. However, for the *unknown* category we may want the robot to store a picture of the unidentified item for later analysis. In this case, it would be useful to throw out additional lists of the Truncated Orchard's Algorithm in order to make space for the new image. An obvious question is, which lists should we toss out? A random selection would be easy, but this may decrease indexing efficiency greatly. Can we do better than random?

Our solution is to frame the Truncated Orchard's Algorithm as an *anyspace* algorithm [99]. Anyspace algorithms are algorithms that trade space for quality of results. In general, an anyspace algorithm is able to solve the problem at hand with any amount of memory, and the speed at which it can solve the problem improves if more space is made available. In our particular case, we assume we *start* with all the memory of full data structure for the Truncated Orchard's Algorithm, and if we need space to store information about an unexpected event, we "cannibalize" a part of the Truncated Orchard's Algorithm's space to store it. We call such an approach an *Auto-Cannibalistic* algorithm. Figure 3.3 shows an idealized anyspace algorithm.

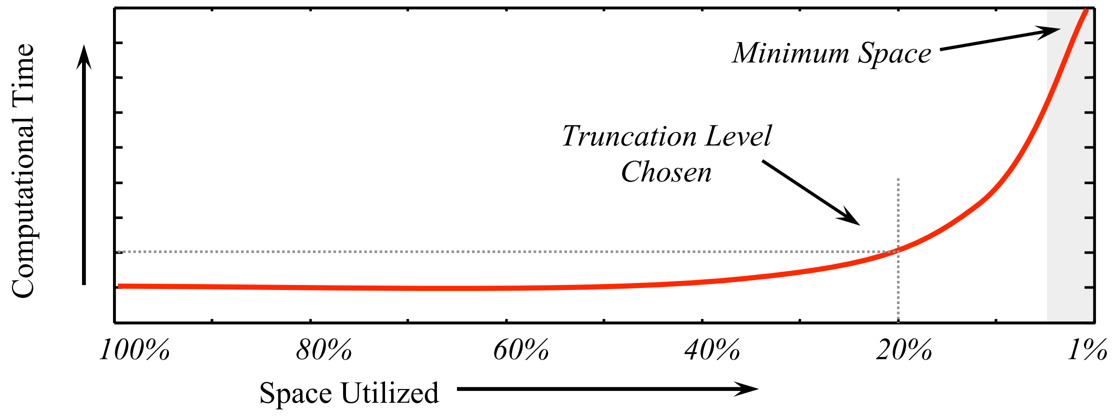


Figure 3.3: An idealized Anyspace Indexing Algorithm

Note that in this hypothetical case we get the best indexing performance if we use all the memory, however we can throw away 80% of the data and the performance only gets twice as bad. We could instead have thrown away 50% of the data with no significant difference in performance. Note that all anyspace algorithms have some absolute minimum amount of memory which they require. In our case this is the $\mathcal{O}(|A|)$ space for the list of items in the data set.

Assume the size of the full data structure is denoted as unity. Then we can denote the size of an anyspace algorithm as S, where $minimum\_space \leq S \leq 1$. In our particular problem we have $\mathcal{O} \leq S \leq \mathcal{O}(|A|^2)$.

The basic framework for using an anyspace algorithm is as follows. We precompute the full space truncated Orchard's Algorithm table and store it in main memory. At some point in the future we expect to get requests for the indexes which are space limited. For example, sensor **A** may need the index, but only have 2MB available. We simply pull off the best 2MB version and give it to sensor **A**. If sensor **B** requests a 3 MB version, we pull off the best 3MB version and give it to **B**.

Note that for any memory size S of the anyspace algorithm, the data structure S is a proper subset of the data structure $S + \epsilon$. That is to say that a larger data structure is always the same as a smaller one, plus some additional data. This is a useful property. First of all it ensures that the size of the full data structure ($S = 1$) is no greater than the original (non anyspace) version (plus a tiny overhead). Thus we have no space overhead for keeping the data in an anyspace format. Second, it allows progressive transmission of the data structure. For example, in our scenario above, if sensor **A** manages to free up some addition memory, and can now devote 2.5MB to indexing the data, we only need to send it the 0.5MB difference.

Anyspace algorithms are rare in machine learning/data mining applications [12, 18], however *anytime* algorithms, which are exact analogues that substitute time instead of space as the critical quality, have seen several data mining applications [69, 73, 74, 88]. Zilberstein and Russell give a number of desirable properties of anytime algorithms, which we can adapt for anyspace algorithms. Below we consider the desirable properties of anyspace algorithms, placing the desirable properties for anytime algorithms in parentheses:

- **Interruptability:** After some small amount of minimum space (*setup time*), the algorithm returns an answer using any additional amount of space (*time*) given.

- **Monotonicity:** the quality of the result is a non-decreasing function of space (*computation time*) used (cf. Figure 3.3).

- **Measurable quality:** the quality of an approximate result can be determined. In our case, this quality is the indexing efficiency, which can be measured by the number of distance calculations required to answer a query.

- **Diminishing returns:** the improvement in solution efficiency (*quality*) is largest at the early stages of computation, and diminishes as more space (*time*) is given (cf. Figure 3.3).

- **Preemptability:** The algorithm can use the space given, or additional space if it become available (*the algorithm can be suspended and resumed*) with minimal overhead.

As we shall see, our anyspace indexing algorithm meets all these properties.

Algorithm 3.4 shows our proposed Anyspace Orchard's algorithm data structure. It differs from the classic data structure (as shown in Table 3.1) in just two ways.

First, the rows are no longer in the original order, but sorted in a best first order. Second, note that in the leftmost column there is a small amount of additional information in the form of a *goto* list. This list tells us what to do if we need to free up some space by deleting lists. We will always delete the lists from the bottom, and replace the entire list by the corresponding *goto* pointer.

Table 3.4: Anyspace Orchard's Ranked Lists (I)

goto list	Item	1st NN {dist}	2nd NN {dist}	3rd NN {dist}	4th NN {dist}	5th NN {dist}	6th NN {dist}
Linear	a_4	**5** {4.2}	**3** {5.0}	**6** {5.0}	**2** {5.8}	**7** {5.8}	**1** {7.1}
goto a_4	a_7	**5** {2.0}	**6** {3.0}	**4** {5.8}	**1** {8.0}	**3** {10.6}	**2** {11.3}
goto a_4	a_3	**2** {1.0}	**4** {5.0}	**1** {8.1}	**6** {8.9}	**5** {9.2}	**7** {10.6}
goto a_4	a_1	**6** {5.0}	**4** {7.1}	**2** {8.0}	**7** {8.0}	**3** {8.1}	**5** {8.2}
goto a_7	a_6	**7** {3.0}	**5** {3.6}	**1** {5.0}	**4** {5.0}	**3** {8.9}	**2** {9.4}
goto a_3	a_2	**3** {1.0}	**4** {5.8}	**1** {8.0}	**6** {9.4}	**5** {10.0}	**7** {11.3}
goto a_7	a_5	**7** {2.0}	**6** {3.6}	**4** {4.2}	**1** {8.2}	**3** {9.2}	**2** {10.0}

Table 3.5: Anyspace Orchard's Ranked Lists (II)

goto list	Item	1st NN {dist}	2nd NN {dist}	3rd NN {dist}	4th NN {dist}	5th NN {dist}	6th NN {dist}
Linear	a_4	**5** {4.2}	**3** {5.0}	**6** {5.0}	**2** {5.8}	**7** {5.8}	**1** {7.1}
goto a_4	a_7	**5** {2.0}	**6** {3.0}	**4** {5.8}	**1** {8.0}	**3** {10.6}	**2** {11.3}
goto a_4	a_3	**2** {1.0}	**4** {5.0}	**1** {8.1}	**6** {8.9}	**5** {9.2}	**7** {10.6}
goto a_4	a_1	**6** {5.0}	**4** {7.1}	**2** {8.0}	**7** {8.0}	**3** {8.1}	**5** {8.2}
	a_6	*goto* a_7					
	a_2	*goto* a_3					
	a_5	*goto* a_7					

As a concrete example, suppose that we must free up approximately 40% of the space used. As shown in Table 3.5, we can achieve this by deleting the last three lists and replacing them with their respective *goto* pointers.

3.3.3 Constructing Anyspace Orchard's

Assume we have truncated Orchard's data structure, T. At one extreme, T has all lists $P[a_i]$ for $1 \leq i \leq |A|$, and is thus the "classic" Orchard's data structure. At the other extreme, it has only a single list, and we can only efficiently answer queries that happen to be near the untruncated item.

Assume that we have a black box function EVALUATEADDITION(T, i) which given T returns the estimated utility of adding list $P[a_i]$. This function estimates the expected number of items that an arbitrary query must be compared to in order to find its nearest neighbor by adding list $P[a_i]$ to the existing table, and returns the highest utility with the smallest comparison number. For the moment we will gloss over the details of this function, except to note that it allows us to create an Anyspace Orchard's Algorithm. The basic idea of the algorithm is to start with an empty set T, and iteratively use function EVALUATEADDITION(T, i) to decide which list to add next. For example, we can see from column 2 of Table 3.1, the lists were added in this order: a_4, a_7, a_3, a_1, a_6, a_2 and a_5.

The formal algorithm 4 is divided into two phases; selection and mapping. The first phase is a simple, greedy-forward selection search. T is initialized as an empty stack in line 1. For each outer iteration, the algorithm tests every item (that was not previously selected) with the utility function (line 7), picks the item with highest utility and pushes it onto T (lines 8 to 10). This procedure continues until all the items are pushed onto T. In essence, this phase sorts all the items by their expected utility for indexing. For instance, in the running example shown in Table 3.1, it first selects a_4, then a_7, then a_3, etc.

The second phase is to create the *goto* list. Our strategy is to start from the bottom of the table, repeatedly selecting the candidate with the lowest utility, and change its *goto* pointer to point at its nearest neighbor with a higher utility.

Algorithm 4 BUILDANYSPACEORCHARDS(A, P)

```
1: $T \leftarrow textNULL$
2: for $i \leftarrow 1$to$|A|$ do
3:     $maxeval \leftarrow$ EVALUATIONADDITION$(T, 1)$
4:     $additem \leftarrow 1$
5:     for $j \leftarrow 2$to$|A|$ do
6:         if $a_j$ is a candidate then
7:             $eval \leftarrow$ EVALUATIONADDITION$(T, j)$
8:             if $eval > maxeval$ then
9:                 $maxeval \leftarrow eval$
10:                $additem \leftarrow j$
11:            end if
12:        end if
13:    end for
14:    PUSH$(T, additem)$
15: end for
16: $P'.sortlists \leftarrow$sort $P$ best first according to $T$
17: for $i \leftarrow |A|$to$2$ do
18:    for $j \leftarrow 1$to$|A| - 1$ do
19:        $index \leftarrow P[a_i].pointer[j]$
20:        if $a_{index}$ appears above $a_i$ in $P'$ then
21:            $P.gotolist[a_i] \leftarrow a_{index}$
22:            Break
23:        end if
24:    end for
25: end for
26: return $P'$
```

To achieve this we scan the sorted Orchard's algorithm table bottom up (as in line 17). For each item a_i under consideration, we scan down its nearest neighbor list $P[a_i].pointer$ in lines 18 and 19. If we find one nearest neighbor aindex that ranks above a_i in sorted Orchard's algorithm table, we assign aindex to the entry of ai in the *goto* list $goto[a_i]$ in line 20 to 22. In the running example, we first consider the item a_5. Following a_5's nearest neighbor list, we check the item a_7, which is on the second line of the sorted Orchard's algorithm table, and thus above a_5. We therefore make $goto[a_5]$ point to a_7. We next consider the item a_2, and so on.

We have yet to explain how our EVALUATIONADDITION function is defined. We propose a simple approach that takes both density and overlap of each item into consideration. Intuitively, we assume the density distribution of the queries to be at least somewhat similar to the density distribution of the data in the index, so we want to reward items for being in a dense part of the space. At the same time, if we have one item from the center of a dense region, then there is little utility in having another item from the same region (overlap), so we want to penalize for this.

Concretely, our algorithm works as follows: the candidate's pool is initialized to include all the items. Given the parameter nearest neighbors number n, we set the item ai maximum utility in the candidate pool with smallest distance between a_i and its nth *valid* nearest neighbor. We then delete ai from candidate pool. In addition, for all the items that on the a_i's nearest neighbor list, ranked from 1 to n, we assign them the minimum utility value (overlap penalty) and they are never again considered to be neighbors of any other items. Suppose we have m items initially, in our approach, we first pick pivot items according its radius to cover n valid nearest neighbors. After that, for those $m\%(n+1)$ items that are uncovered by any pivot item, we pick them in random order. Finally, we pick remaining nearest neighbors items in random order. We did consider several other possibilities, such as leaving-one-out evaluation, measuring the rank correlation, mutual information, or entropy gain between two lists as a measure of redundancy. However either these ideas did not work empirically, or

Table 3.6: Anyspace Orchard's Ranked Lists (III)

goto list	Item	1^{st} NN {dist}	2^{nd} NN {dist}	3^{rd} NN {dist}	4^{th} NN {dist}	5^{th} NN {dist}	6^{th} NN {dist}
Linear	a_4	**5** {4.2}	**3** {5.0}	**6** {5.0}	**2** {5.8}	**7** {5.8}	**1** {7.1}
goto a_4	a_7	**5** {2.0}	**6** {3.0}	**4** {5.8}	**1** {8.0}	**3** {10.6}	**2** {11.3}
	a_3	*goto* a_4					
	a_1	*goto* a_4					
	a_6	*goto* a_7					
	a_2	*goto* a_3					
	a_5	*goto* a_7					

required several parameters to be tuned. As a simple sanity check, we will include empirical comparisons to random, a variant of EVALUATIONADDITION which simply chooses a random list to add.

3.3.4 Using Anyspace Orchard's Algorithm

After constructing the sorted Orchard's algorithm table, it is easy to adapt the Orchard's algorithm search technique (Algorithm 6) to allow it to search in the truncated version. The main task is to decide what we should do if the algorithm indicates a jump to the list of a certain item a_i while that list has already been deleted.

Simply jumping to the list of a_j if $a_j = goto[a_i]$ is not possible, as the list of a_j may also have been deleted. Consider the running example in Table 3.5. If two more lists are deleted, the Orchard's Algorithm table shown in Table 3.6 is produced.

Suppose some query arrives, and the algorithm finds itself needing to jump to the list of a_2. Since the list of a_2 is deleted, it wants to jump to a_3 which the *goto* entry of a_2 indicates.

However, it cannot do so, because the list of a_3 is also deleted. The algorithm should continue the jump action, and see whether the list of $a_4 = goto[a_3]$ is deleted. The general approach to find a valid item to jump to is described in Algorithm 5.

Algorithm 5 FINDGOTO($gotolist, a_i$)

```
1: $item \leftarrow a_i$
2: while 1 do
3:     $item \leftarrow gotolist[item]$
4:     if the list of item is not deleted then
5:         Break
6:     end if
7: end while
8: return item
```

There are two additional things we need to check. One is if the list of a_j has not been visited, and the other is if the distance between q and a_j is smaller than the distance between query q and a_{nn}, which is the item whose list we are currently visiting.

The first test is performed to avoid an infinite loop which makes the algorithm jump back and forth between the ranked lists. The second test is because the spirit of Orchard's algorithm tells us to attempt to jump to the item that is nearer the query than the item being visited. After confirmation of the two questions above, we can safely jump to the list of a_j. Algorithm 6 shows the entire Anyspace Orchard's search algorithm.

First, the algorithm checks the available space and builds the truncated sorted Orchard's algorithm table in lines 1 to 3. One modification is that we add some bookkeeping to the item that best matches query, and the corresponding distance in lines 7 and 8, lines 13 to 15, and lines 20 to 22. The reason is the list of best-match item may have been deleted, thus the search does not necessarily end at the best-match item. Therefore the information of the

Algorithm 6 ANYSPACEORCHARDS(A, q)

```
1: Build $P$
2: $P' \leftarrow$ BUILDANYSPACEORCHARDS$(A, P)$
3: truncate $P'$ from bottom to fit it into memory
4: $nn.loc \leftarrow$ random integer between 1 and $|A|$
5: $nn.dist \leftarrow$ DIST$(ann.loc, q)$
6: $index \leftarrow 1$
7: $best \leftarrow nn$
8: while $P[a_{nn.loc}].dist[index] < 2 \times nn.dist$ AND $index < |A|$ do
9:    $item \leftarrow P[a_{nn.loc}].pointer[index]$
10:   $d \leftarrow$ DIST$(a_{item}, q)$
11:   if $d < nn.dist$ then
12:      if $d < bestdist$ then
13:         $best.dist \leftarrow d$
14:         $best.pos \leftarrow item$
15:      end if
16:      $a_{goto} \leftarrow$ FINDGOTO$(gotolist, a_{index})$
17:      if list of $a_{goto}$ not visited AND DIST$(a_{goto}, q) < nn.dist$ then
18:         $nn.loc \leftarrow goto$
19:         $nn.dist \leftarrow$ DIST$(a_{goto}, q)$
20:         if $nn.dist < bestdist$ then
21:            $best \leftarrow nn$
22:         end if
23:      else
24:         $index \leftarrow index + 1$
25:      end if
26:   else
27:      $index \leftarrow index + 1$
28:   end if
29: end while
30: return $best$
```

best-match item should be stored at the time we compare the item to the query. Another modification is that we find the valid item to jump to in line 16, and test if we should jump to that item in line 17 as discussed above. Apart from these minor changes, the rest of the algorithm is exactly the same as in the classical Orchard's algorithm.

3.3.5 Optimizations of Anyspace Orchard's Algorithm

As described above, the mechanism used to create the Anyspace Orchard's algorithm may be quite slow. In some sense this is not a big issue, since we expect to perform this step offline. Nevertheless, it is reasonable to ask if we can speed up this process.

Note that one cause of its lethargy is the redundant calculations spent in finding the valid *goto* entries. The time will increase as the more lists being deleted. Here we show a simple one-scan strategy to update the entire *goto* list and avoid this overhead, which is described in Algorithm 7. The parameter cut means the number of sorted rank lists can accommodate in the memory.

Algorithm 7 FINDGOTOLIST(P', cut)

```
1: for $i \leftarrow 1$ to $cut$ do
2:     $a_i \leftarrow$ the item on the $i$th row of $P'.sortlists$
3:     $validgotolist[a_i] \leftarrow a_i$
4: end for
5: for $i \leftarrow cut + 1$ to $n$ do
6:     $a_i \leftarrow$ the item on the $i$th row of $P'.sortlists$
7:     $validgotolist[a_i] \leftarrow validgotolist[P'.gotolist[a_i]]$
8: end for
9: return $validgotolist$
```

We initialize the *goto* entry of those items which have not been truncated to point to themselves in lines 1 and 4. This initialization does not affect these items, as they never use *goto* pointers, and makes finding valid *goto* pointers easier for the remaining items. In lines 5 and 8, we consider these truncated items from top to bottom. For each item a_i we are considering, we are sure all the items whose position above it in the table already point to a valid item. Suppose a_k is the item that a_i's original goto pointer points to. In that case, the option is either making the valid a_i's *goto* pointer the same as a_k's *goto* pointer when a_k is truncated, or when a_k is not truncated, the pointer should point to a_k. Once the valid *goto* list is built, we can avoid all the redundant *goto* searches.

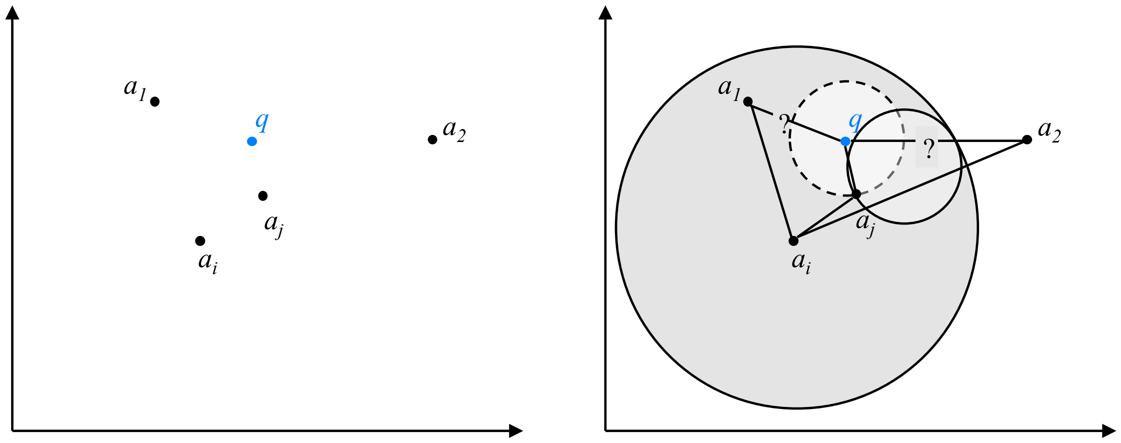


Figure 3.4: Assume a_j is the current nearest neighbor of query q, and that $P[a_j]$ was deleted and replaced with the *goto* entry a_i, *Left*) A newly arrived query q must be answered. *Right*) An admissible pruning rule is to exclude items whose distance to a_i is greater than or equal to $\text{DIST}(a_i, a_j) + 2 \times \text{DIST}(q, a_j)$. In this example, everything outside the large gray circle can be pruned

Another optimization is to narrow down the pruning criteria. We discovered an extra inequality we can exploit using the distance between the query and the best-so-far item. As in Figure 3.4.*Left*, suppose a_j is the best-so-far item while its rank list has been truncated, and a_i is a valid item which a_j's *goto* pointer points to. As shown in Figure 3.4.*Right* we only need to compute the actual distance between q and any $a_k \in A$ only if the following inequality holds: $\text{DIST}(a_i, a_k) < 2 \times \text{DIST}(a_j, query) + \text{DIST}(a_i, a_j)$. Recall that, as shown in Section 3.2, a_k can be pruned if

$$\textsc{Dist}(a_j, a_k) \geq 2 \times \textsc{Dist}(a_j, q) \tag{3.4}$$

However, $\textsc{Dist}(a_j, a_k)$ is not available because $P[a_j]$ was truncated. There is an additional inequality between the three items where we can have the lower bound of the value of $\textsc{Dist}(a_j, a_k)$:

$$\textsc{Dist}(a_j, a_k) \geq \textsc{Dist}(a_i, a_k) - \textsc{Dist}(a_i, a_j) \tag{3.5}$$

Combining Equations 3.4 and 3.5, if we have

$$\textsc{Dist}(a_i, a_k) - \textsc{Dist}(a_i, a_j) \geq 2 \times \textsc{Dist}(a_j, q) \tag{3.6}$$

The item a_k can be admissibly pruned. In our implementation, we can simply replace line 8 in Algorithm 6 with the line below, which is:

```
8': While $P[a_{nn.loc}].dist[index] < 2 \times nn.dist$ AND $index < |A|$ AND $\textsc{Dist}(a_i, a_k) \leq$
    $\textsc{Dist}(a_i, a_j) + 2 \times \textsc{Dist}(a_j, q)$
```

3.4 Experiments

We begin by stating our experimental philosophy. In order to ensure easy replication of our experiment we have placed all data and code at a publicly available website [91].

Recall that our algorithm for constructing truncated Orchard's algorithm has a parameter n. One objective of our experiments is to see how sensitive our algorithm is to the choice of this parameter. A further objective is to test the utility of our EVALUATIONADDTION function. It might be that any function would work well in this context. As a simple baseline comparison we compare against a function that randomly orders the lists for truncation. To understand the algorithm's efficiency we measure the average number of distance calculations needed to answer a one nearest neighbor query. In this, and all subsequent experiments we normalize the range to be between zero and one when creating figures, so a perfect algorithm would have a value near zero, and a sequential scan would have a value of one.

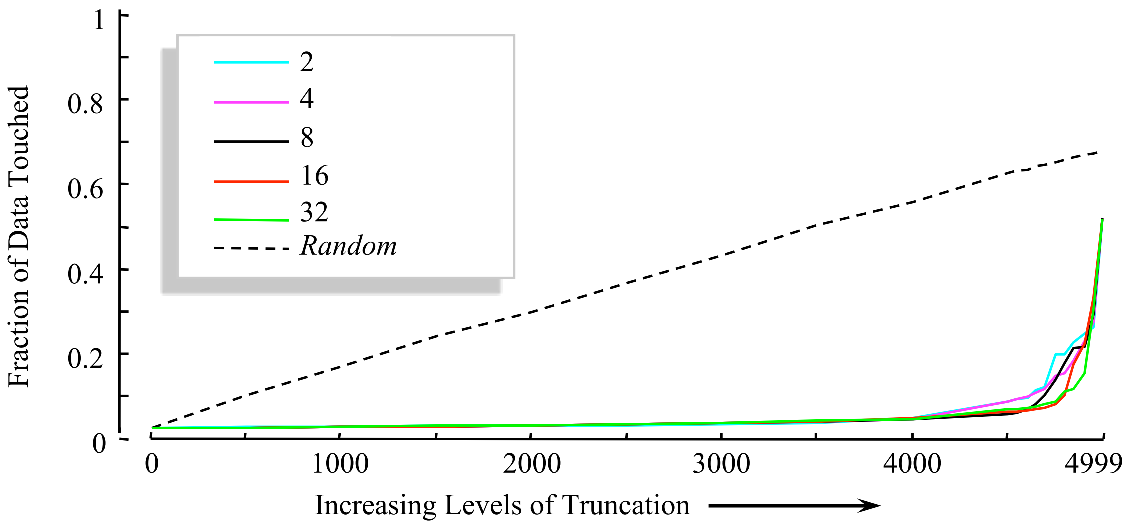


Figure 3.5: The indexing efficiency vs. level of truncation for a synthetic data set

We begin with a simple experiment on a synthetic data set. We created a data set of 5,000 random items from a 2D Gaussian distribution. We created a further 50,000 test examples. We begin by testing the indexing efficiency of the full Orchard's algorithm (i.e with zero truncation) with the test set, and then we truncate a single item and test again. We repeat the process until there is only a single list available to the algorithm, Figure 3.5 shows the experiment on the synthetic data.

The results appear very promising (compare to the idealized case in Figure 3.3). First, it is clear that the choice of parameter n has very little impact on the results. Note that we can truncate 80% of the data without making a significant difference to the efficiency. Thereafter, the efficiency does degrade, but gracefully. Further notice that in contrast to our algorithm, the random approach has linear relationship between size and efficiency. This tells us our EVALUATIONADDITION function does find redundancies in the data to exploit.

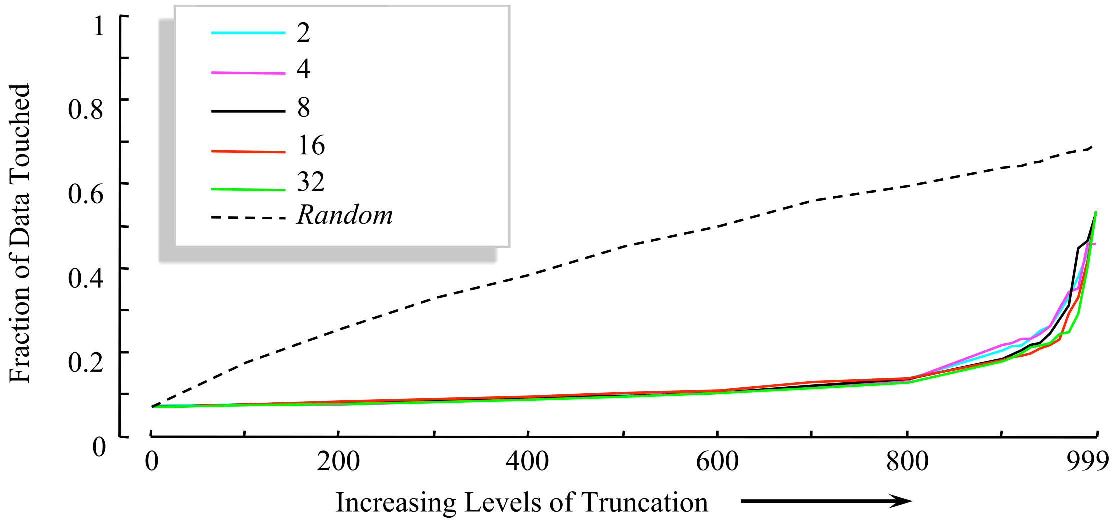


Figure 3.6: The indexing efficiency vs. level of truncation for the Dodgers data set

We next consider a problem of indexing data from a road sensor. This sensor data was collected for the Glendale on-amp at the 5-North freeway in Los Angeles. The observations were taken over 25 weeks, at 5 minute count aggregates. As the location is close to the

Dodgers stadium, it has bursty behavior on days in which a game is played. There are a total of 47497 observations, we randomly choose 1,000 to build our index, and used the rest for testing. Figure 3.6 shows the results.

As before, the performance of our algorithm is exactly we would like to see in an idealized anyspace algorithm, and once again, and our algorithms performance is almost invariant to the choice of parameter n.

Having shown that truncated Orchard's algorithm passes some simple sanity checks, in the next section we consider detailed case studies of problems we can solve using our algorithm.

3.5 Experimental Case Studies

We conclude the experimental section with two detailed case studies of uses for our algorithms.

3.5.1 Insect Monitoring

ISCA Technologies is a Southern California based company that produces devices to monitor and control insect populations in order to mitigate harm to agricultural and human health. They have produced a "smart-trap" device that can be mass produced, and left unattended in the field for long periods to monitor a particular insect of interest.

The system under consideration here is primarily designed to track *Aedes aegypti* (yellow fever mosquito), a mosquito that can spread the dengue fever, yellow fever viruses, and a host of other diseases. In particular, the system needs to classify the sex of the insects and keep a running total of how many of each sex are encountered[1] . In order to only capture *Aedes aegypti*, the trap can be designed specifically for them. For example, carbon dioxide can be used as an attractant (this eliminates most non-mosquito insects). The trap can be placed at a certain height which eliminates low flying insects, and the entrance can be made small enough to prevent larger insects from entering. Nevertheless, as we shall see, non *Aedes aegypti* insects can occasionally enter the traps.

While we have attempted classification of the insects with Bayesian Classifiers, SVMs and decision trees, our current best results come from one Nearest Neighbor classification with a 4-dimensional feature vector extracted from the audio signal. A further advantage of using 1NN is that it allows us to come up with a simple definition of outlier. We empirically noted that on average both males and females tend to be a distance of m to their nearest neighbors. This number has a relatively small standard deviation. We therefore have defined an outlier as a data sample that is more than m plus four standard deviations from its nearest neighbor. Figure 3.7 shows a visual intuition of this.

There are two obvious sources of outliers; non-insect sounds from outside the trapincluding helicopters and farming equipmentand non-*Aedes aegypti* insects that enter the trap. Knowing the true identity of the outliers can be very useful. In the former case, it may be

[1]Recall that only female mosquitoes suck blood from humans and other animals.

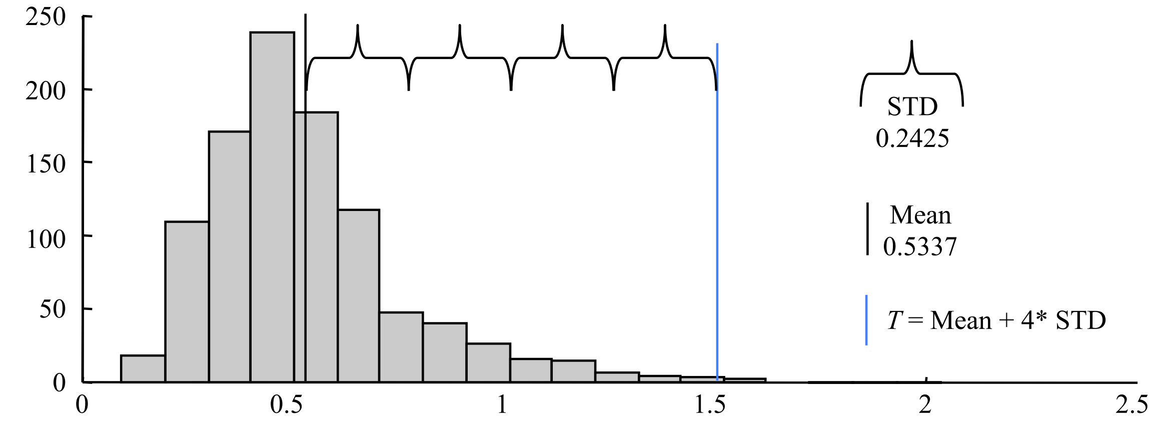


Figure 3.7: The distribution of distances to nearest neighbor for 1,000 insects in our training set. We consider exemplars whose distance to their nearest neighbor is more than the mean plus 4 standard deviations to be suspicious and worthy of follow-up investigation

possible to change the traps location to reduce the number of outlier events caused by the sound of farm machinery. In the latter case, it may be useful know which unexpected insects we have caught. For example, we may have been subject to an unexpected invasion, as in the famous invasion of Glassy-winged Sharpshooters (*Homalodisca coagulata*) which almost devastated the wine industry in Temecula southern Californias Temecula Valley 1997 [10]. However, the low power/low memory requirements of the traps prohibit recording the entire audio stream. Our solution therefore is to use an auto-cannibalistic algorithm which allows efficient indexing to support the one-nearest neighbor classification, and to record a one second snippet of outlier sounds. Each snippet requires the auto-cannibalistic to delete 3 lists from its table.

A classifier was built using 1,000 lab reared insects, with 500 of each sex. The training error suggests that we can achieve over 99% accuracy, however we have yet to confirm this by hand annotation of insects captured in the field. In Figure 3.8 we see a plot showing the indexing efficiency for various levels of truncation.

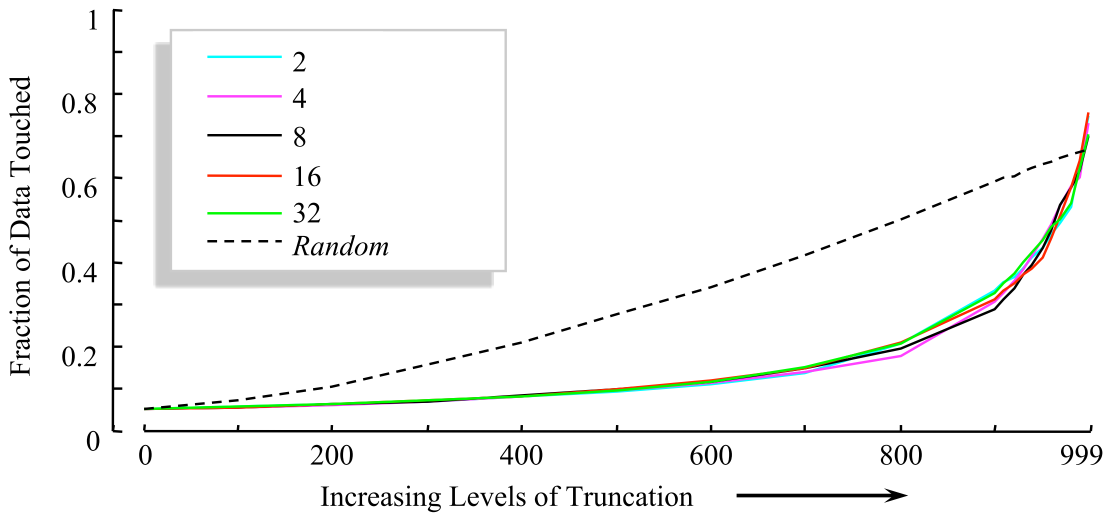


Figure 3.8: Indexing efficiency vs. space on the insect monitoring problem

Note that the application lends itself well to any anyspace framework. Even when we have deleted 25% of the data we can barely detected any change in the indexing efficiency.

Having demonstrated the concept in the lab, we deployed an auto-cannibalistic algorithm in the field. As a baseline comparison we compared to hard-coded truncated Orchard's algorithms where just 5%, 1% and a single list remains. Figure 3.9 shows the results.

The figure shows that the more memory an indexing algorithm has, the more efficiently it can process incoming data. The auto-cannibalistic algorithm starts out with a full Orchard's algorithm in memory and is consequently much more effective than its smaller rivals. Over time, outliers are encountered and must be stored, so the amount of memory available to auto-cannibalistic algorithm decreases, causing it to become less efficient. However it is difficult

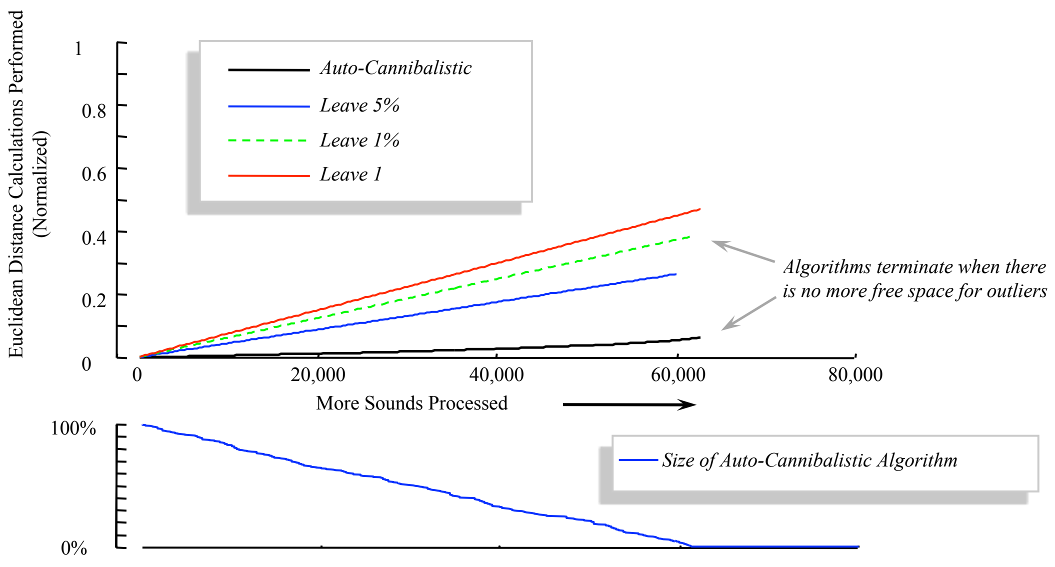


Figure 3.9: *Top*) The cumulative number of Euclidean distance calculations performed vs. the number of sound events processed. *Bottom*) The size of auto-cannibalistic algorithm vs. the number of sound events processed

to detect this for the first 50,000 or so events. As the power required is almost perfectly correlated with the cumulative number of Euclidean distance calculations, which in turn is simply the area under the curves, the auto-cannibalistic algorithm requires less than one tenth the energy of the 5% Orchard's algorithms, and is able to handle 4.99% more outlier events before its memory fills up.

3.5.2 Robotic Sensors

In this section we consider the utility of auto-cannibalistic algorithms in a robotic domain. Note that unlike the insect example in the previous section, this is not a mature fielded product; it is simply a demonstration on a toy problem. In particular, we are not claiming that the approach below for finding unexpected tactical sensations is the best possible approach; it is merely an interesting test bed for a demonstration of our ideas.

The Sony AIBO, shown in Figure 3.10, is a small quadruped robot that comes equipped with a tri-axial accelerometer. This accelerometer measures data at a rate of 125 Hz.

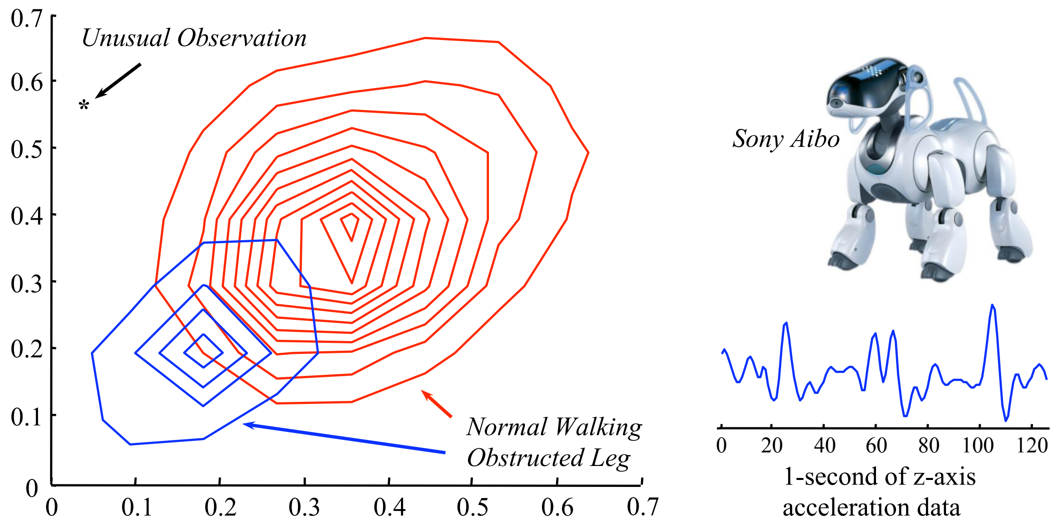


Figure 3.10: *clockwise from top right.* A Sony Aibo robot. An on-board sensor can measure acceleration at 125 Hz. The accelerometer data projected into two dimensions

By examining the sensor traces, we can (perhaps imperfectly) learn about the surface the robot is walking on. For example, in Figure 3.10 we show a two dimensional mapping of the Z-axis time series for both normal unobstructed walking and walking when one leg is obstructed. While the overlap of the two distributions in this figure suggests a high error rate, in three dimensions the separation is better, and we can achieve about 96% accuracy. Naturally it is useful to distinguish between these two situations, as the robot can attempt to free itself or change direction. In addition to merely classifying current state, it may be useful to detect unusual states and photograph them for later analysis. As the AIBO has only 4 megabytes of flash memory on board, memory must be used sparingly. A single compressed image with its 416x320 pixel camera requires about 100k of space.

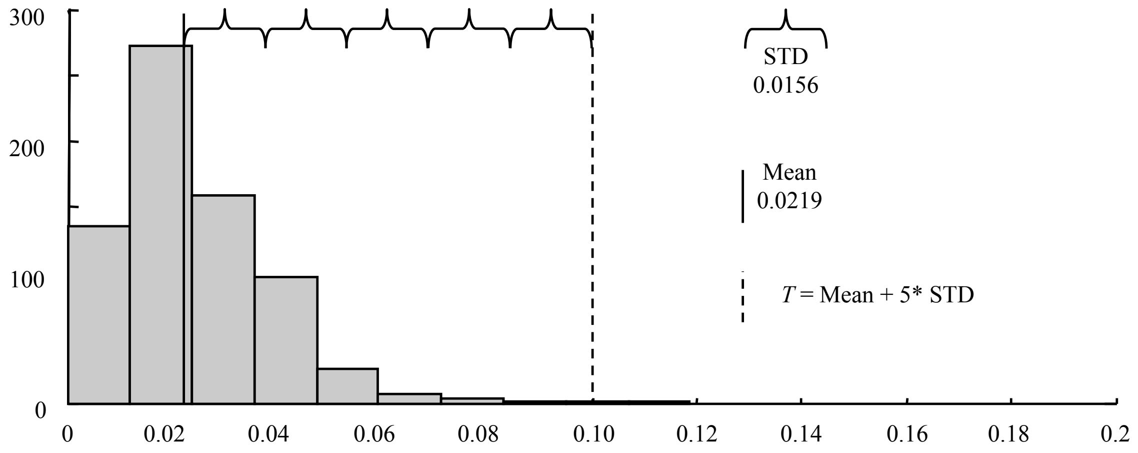


Figure 3.11: The distribution of distances to nearest neighbor for 700 tactile events in our training set. We consider exemplars whose distance to their nearest neighbor is more that the mean plus five standard deviations to be suspicious, and worthy the memory required to take a photograph

We use the same basic framework as in the previous section here. We took 700 training instances and use them to build a 1-nearest neighbor classifier indexed by the truncated Orchard's algorithm. Every time an outlier is detected, and an image must be stored, we must delete an average of 18 lists from our index to make room for it. Figure 3.12 shows the result of the experiment. As before, we compare to hard-coded truncated Orchard's algorithms where just 5%, 1% and a single list remains.

As with the insect example, the truncated Orchard's algorithm requires only a fraction of the energy of the fixed-size indices, and is able to process data until just a single list remains available after dealing with event 1,522.

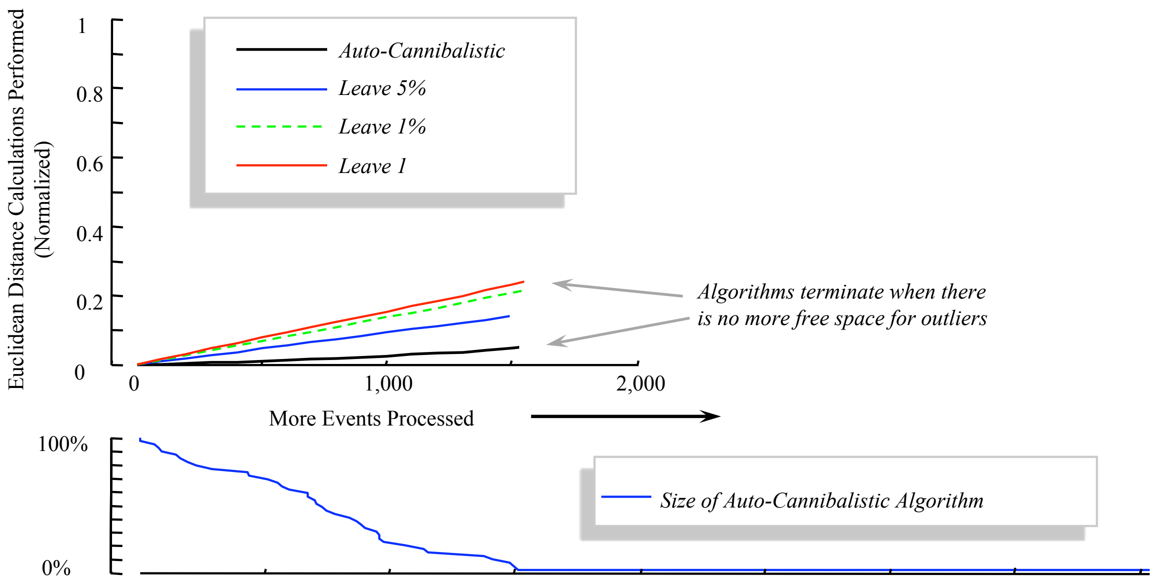


Figure 3.12: *Top*) The cumulative number of Euclidean distance calculations performed vs. the number of tactile events processed. *Bottom*) The size of auto-cannibalistic algorithm vs. the number of tactile events processed

3.6 Discussion

We have introduced a novel indexing method especially for sensor data mining. In this section, we discuss the related work and provide future extensions.

3.6.1 Related Work

Indexing is important for similarity search because it will reduce a large amount of searching time since it can eliminate expensive distance calculations. The problem of indexing under metric distance has been studied intensively in the last decade and many efficient algorithms have been proposed. There are basically two alternative categories:

- **Embedding method:** for the objects in the data set of N dimensions, it created a k-dimension feature vector to represent each object. The distance calculated in the k-dimension feature space provides a lower bound of the actual distance between the objects. If k is considerably smaller than N, and the lower bound is reasonably tight, it can prune a lot of objects with much less distance calculation effort. Examples of this method are in [26, 76].

- **Distance-based method:** typical distance-based method is based on partition. All or some distance between the objects in the data set are precomputed. When a query comes in, we can estimate the majority of distances between the objects and query based on a small fraction of the actual distance we have computed between the objects and the query, and thus prune a lot of non-qualified objects. The vantage-point tree method [95] is an example in this category. The method we proposed in this chapter falls into the second category. The obvious drawback of method in the second category is that the index data structure is fixed, which means, the indexing has a rigid memory requirement. However, under the scenario where main memory is bottleneck, e.g. in the sensor or robot, the algorithms with fixed memory requirement may fail.

3.6.2 Conclusion

In this chapter, our major contribution is that:

- We have shown the Orchard's algorithm may be rescued from its relative obscurity by considering it as an anyspace algorithm and leveraging off of its unique properties to produce efficient sensor mining algorithms.
- We have further shown what we believe is the first example of an auto-cannibalistic algorithm.

Chapter 4

Time Series Shapelets: A Novel Technique that Allows Accurate, Interpretable and Fast Classification

This chapter develops a method of feature extraction which considers local features that distinguish time series in one class from those in other classes. A classification model built based on the features is both time and space efficient.

Classification of time series has been attracting great interest over the past decade. While dozens of techniques have been introduced, recent empirical evidence has strongly suggested that the simple nearest neighbor algorithm is very difficult to beat for most time series problems, especially for large-scale data sets. While this may be considered good news, given the simplicity of implementing the nearest neighbor algorithm, there are some negative consequences of this. First, the nearest neighbor algorithm requires storing and searching the entire data set, resulting in a high time and space complexity that limits its applicability, especially on resource-limited sensors. Second, beyond mere classification accuracy, we often wish to gain some insight into the data and to make the classification result more explainable.

In this chapter we introduce a new time series primitive, *time series shapelets*, which addresses these limitations. Informally, shapelets are time series subsequences which are in some sense maximally representative of a class. As we shall show with extensive empirical evaluations in diverse domains, algorithms based on the time series shapelet primitives can be interpretable, more accurate and significantly faster than state-of-the-art classifiers.

4.1 Introduction

While the last decade has seen a huge interest in time series classification, to date the most accurate and robust method is the simple nearest neighbor algorithm [22, 61, 82]. While the nearest neighbor algorithm has the advantages of simplicity and not requiring extensive parameter tuning, it does have several important disadvantages. Chief among these are its space and time requirements, since during the classification, the algorithm needs the object to be compared with each object in the training set, leading to quadratic time and linear space complexity. Another drawback is the fact that it tells us very limited information about why a particular object is assigned to a particular class.

In this chapter we present a novel time series data mining primitive called *time series shapelets* [93]. Informally, shapelets are time series subsequences which are in some sense maximally representative of a class. While we believe shapelets can have many uses in data mining, one of their obvious implications is their ability to mitigate the two weaknesses of the nearest neighbor algorithm noted above.

Because we are defining and solving a new problem, we will take some time to consider a detailed motivating example. Figure 4.1 shows some examples of leaves from two classes, *Urtica dioica* (stinging nettles) and *Verbena urticifolia*. These two plants are commonly confused, hence the colloquial name "false nettle" for *Verbena urticifolia*.

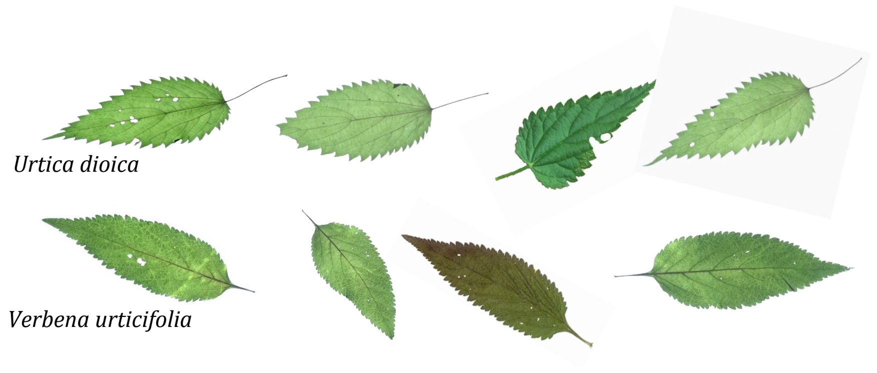


Figure 4.1: Samples of leaves from two species. Note that several leaves have insect-bite damage

Suppose we wish to build a classifier to distinguish these two plants; what features should we use? Since the intra-variability of color and size within each class completely dwarfs the inter-variability between classes, our best hope is based on the shapes of the leaves. However, as we can see in Figure 4.1, their differences in the global shapes are very subtle. Furthermore, it is very common for leaves to have distortions or "occlusions" due to insect damage, which are likely to confuse any global measures of shape. Instead we attempt the following. We first convert each leaf into a one-dimensional representation as shown in Figure 4.2.

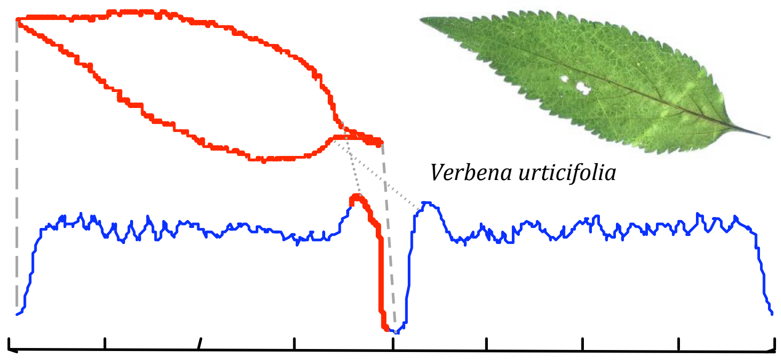


Figure 4.2: A shape can be converted into a one dimensional "time series" representation. The reason for the highlighted section of the time series will be made apparent shortly

Such representations have been successfully used for the classification, clustering and outlier detection of shapes in recent years [44]. However, here we find that using the nearest neighbor classifier with either the (rotation invariant) Euclidean distance or Dynamic Time Warping (DTW) distance does not significantly outperform random guessing. The reason for the poor performance of these otherwise very competitive classifiers seems to be due to the fact that the data is somewhat noisy, and this noise is enough to swamp the subtle differences in the shapes.

Suppose, however, that instead of comparing the *entire* shapes, we only compare a *small* subsection of the shapes from the two classes that is particularly discriminating. We can call such subsections *shapelets*, which invokes the idea of a small "sub-shape." For the moment we ignore the details of how to formally define shapelets, and how to efficiently compute them. In Figure 4.3, we see the shapelet discovered by searching the small dataset shown in Figure 4.1.

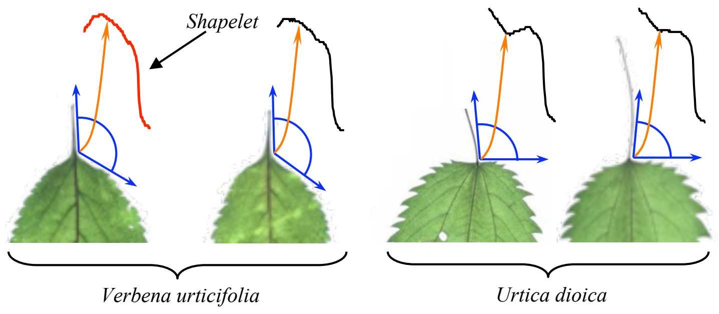


Figure 4.3: Here, the shapelet hinted at in Figure 4.2 (in both cases shown with a bold line) is the subsequence that best discriminates between the two classes

As we can see, the shapelet has "discovered" that the defining difference between the two species is that *Urtica dioica* has a stem that connects to the leaf at almost 90 degrees, whereas the stem of *Verbena urticifolia* connects to the leaf at a much wider angle. Having found the shapelet and recorded its distance to the nearest matching subsequence in all objects in the database, we can build the simple decision-tree classifier shown in Figure 4.4.

The reader will immediately see that this method of classification has many potential advantages over current methods:

- Shapelets can provide interpretable results, which may help domain practitioners better understand their data. For example, in Figure 4.3 we see that the shapelet can be summarized as the following: "*Urtica dioica* has a stem that connects to the leaf at almost 90 degrees." Most other state-of-the-art time series/shape classifiers do not produce interpretable results [22, 42].

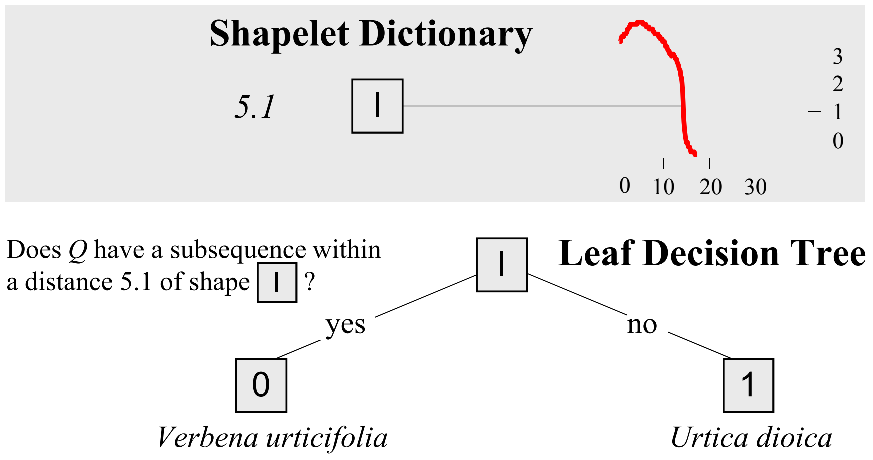


Figure 4.4: A decision-tree classifier for the leaf problem. The object to be classified has all of its subsequences compared to the shapelet, and if any subsequence is less than (the empirically determined value of) 5.1, it is classified as *Verbena urticifolia*

- Shapelets can be significantly more accurate/robust on some datasets. This is because they are local features, whereas most other state-of-the-art time series/shape classifiers consider global features, which can be brittle to even low levels of noise and distortions [22]. In our example, leaves which have insect bite damage are still usually correctly classified.

- Shapelets can be significantly faster at classification than existing state-of-the-art approaches. The classification time is just $\mathcal{O}(ml)$, where m is the length of the query time series and l is the length of the shapelet. In contrast, if we use the best performing global distance measure, rotation invariant DTW distance [44], the time complexity is on the order of $\mathcal{O}(km^3)$, where k is the number of reference objects in the training set[1] . On real-world problems the speed difference can be greater than three orders of magnitude.

- Shapelets save considerable space compared to the state-of-the-art approaches. The space required by the shapelet is $\mathcal{O}(nl)$, where l is the length of the shapelet and n is related to the number of classes of the training set, since shapelets indicate the features of the objects in the entire class. For the example-based approach, the space requirement is $\mathcal{O}(km)$, where k is the number of reference objects and m is the length of reference objects in the training set. Therefore, n is much smaller than k.

The leaf example, while from an important real-world problem in botany, is a contrived and small example to help develop the readers intuitions. However, as we shall show in Section 4.5, we can provide extensive empirical evidence for all of these claims on a vast array of problems in domains as diverse as anthropology, human motion analysis, spectrography, and historical manuscript mining.

The rest of this chapter is organized as follows. In Section 4.2, we review related work and background material. Section 4.3 introduces concrete algorithms to efficiently find shapelets. Section 4.4 discusses the method of classification using shapelets as a tool. In Section 4.5, we perform a comprehensive set of experiments on various problems of different domains. Finally, in Section 4.6 we conclude our work and suggest directions for future work.

[1]There are techniques to mitigate the cubic complexity of rotation invariant DTW, but unlike shapelets, the computational time *is* dependent on the size of the training dataset.

4.2 Related Work and Background

4.2.1 Related Work

Classification for interpretability

Classification has attracted significant attention over the past decade. Among all the time series classification algorithms, the simple nearest neighbor algorithm has been shown to be the most robust and accurate method by experiments on a great variety of datasets [82]. Though there are different representation methods, such as Discrete Fourier Transformation (DFT) [27] and Symbolic Aggregate approXimation (SAX) [49], and different similarity measures, such as Euclidean distance [27] and Dynamic Time Warping (DTW) [4, 43], a recent large scale set of experiments indicates that the difference between different methods and measures diminishes or disappears as the size of the dataset becomes larger [22].

In many real-world applications, classification tasks should focus not only on accuracy, but also on *interpretability*. Rule-based methods, like the decision tree method, are generally more interpretable than instance-based methods. However, it is difficult to find a general method to extract rules from time series datasets.

Previous work uses either global or predefined features to construct the model. In [40], the method classifies and interprets the data via two ways: global feature calculation and event extraction. The parameterized events are clustered in the parameter space and synthetic event attributes are extracted from the clusters. The global features and event attributes combine together to form the classifier. The drawback of this method is that all of the events must be defined by a user on a case-by-case basis. It becomes particularly difficult to determine which events should be used in the clusters when the dataset becomes large, even if you already know the rule of classifying examples from different classes.

The work in [87] also uses the decision tree classifier to explore comprehensibility. However, instead of extracting the common feature from the time series in one class, their method compares test examples with one entire time series at each internal node of the decision tree. The search space of extracting one entire time series is several orders of magnitude smaller than extracting a shapelet. In the meantime, viewing the time series in the global view also significantly reduces both the interpretability and effectivity of the classification, especially when the application contains high-dimensional, large-scale, or noisy datasets.

The closest work is that of [29]. Here the author also attempts to find local patterns in a time series which are predictive of a class. However, the author considers the problem of finding the best such pattern intractable, and thus resorts to examining a single, randomly chosen instance from each class; even then the author only considers a reduced piecewise constant approximation of the data. While the author notes "*it is impossible in practice to consider every such subsignal as a candidate pattern*," this is in fact exactly what we do, aided by eight years of improvements in CPU time and, more importantly, an admissible pruning technique that can prune off more than 99% of the calculations (c.f. Section 4.5). Our work may also be seen as a form of a supervised motif discovery algorithm [14], in that we are looking for repeated patterns (as in motif discovery). However, we are trying to find "motifs" which exist mostly in one class and not the other. So motif discovery can at most be seen as a subroutine in our work.

Similar Problems in Other Domains

In string processing, there is a somewhat similar problem called the *Distinguishing SubString Selection* (DSSS) [31]. The problem is defined as follows: Two sets of strings, one "good" and the other "bad", produce a substring, which is "close" to the strings in the "bad" set, and "away" from those in the "good" set. This problem is simpler than the shapelet discovery problem. First, it is only defined for a finite alphabet of the strings. In most instances in the literature only a binary alphabet is considered. Second, it uses the Hamming metric as the distance measure. Moreover, only two-class problems are considered. Most methods initiate the candidate-distinguishing substring with the inverse of the first string in the "good set," and adjust the candidate substring in all possible ways to move it away from some string in the "good set" if the candidate is too close to that string.

4.2.2 Notation

Table 4.1 summarizes the notation in this chapter; we expand on the definitions below.

We then define the key terms in the paper. For ease of exposition, we consider only a two-class problem. However, extensions to a multiple-class problem are trivial.

Table 4.1: Symbol table

Symbol	**Explanation**
T, R	Time series
S	Subsequence
S_T^l	Set of all subsequences of length l from time series T
m, $\lvert T \rvert$	Length of time series
$\bar{m}$	Average length of time series in a dataset
l, $\lvert S \rvert$	Length of subsequence
d	Distance measurement
$\mathbf{D}$	Time series dataset
A, B	Class label
H	Entropy
$\hat{H}$	Weighted average entropy
sp	Split strategy
k	Number of time series objects in the dataset
$\mathbf{C}$	Classifier
$S_{(k)}$	The kth data point in subsequence S

Definition 4. *Subsequence.* Given a time series T of length m, a subsequence S of T is a sampling of length $l \leq m$ of contiguous positions from T, that is, $S = t_p \ldots t_{p+l-1}$, for $1 \leq p \leq m - l + 1$.

Our algorithm needs to extract all of the subsequences of a certain length. This is achieved by using a sliding window of the appropriate size.

Definition 5. *Sliding Window.* Given a time series T of length m, and a user-defined subsequence length of l, all possible subsequences can be extracted by sliding a window of size l across T and considering each subsequence S_p^l of T. Here the superscript l is the length of the subsequence and subscript p indicates the starting position of the sliding window in the time series. The set of all subsequences of length l extracted from T is defined as S_T^l, $S_T^l = \{S_p^l \text{ of } T, \text{for } 1 \leq p \leq m - l + 1\}$.

As with virtually all time series data mining tasks, we need to provide a similarity measure between the time series $\textsc{Dist}(T, R)$.

Definition 6. *Distance between the time series,* $\textsc{Dist}(T, R)$. $\textsc{Dist}(T, R)$ is a distance function that takes two time series T and R of equal length as inputs and returns a nonnegative value d, which is said to be the distance between T and R. We require that the function $\textsc{Dist}$ be symmetrical; that is, $\textsc{Dist}(R, T) = \textsc{Dist}(T, R)$.

The $\textsc{Dist}$ function can also be used to measure the distance between two subsequences of the same length, since the subsequences are of the same format as the time series. However, we will also need to measure the similarity between a short subsequence and a (potentially much) longer time series. We therefore define the distance between two time series T and S, with $|S| < |T|$ as:

Definition 7. *Distance from the time series to the subsequence,* $\textsc{SubsequenceDist}(T, S)$. $\textsc{SubsequenceDist}(T, S)$ is a distance function that takes time series T and subsequence S as inputs and returns a nonnegative value d, which is the distance from T to S. $\textsc{SubsequenceDist}(T, S) = \min(\textsc{Dist}(S, S')),$ for every $S' \in S_T^{|S|}$.

Intuitively, this distance is simply the distance between S and its best matching location somewhere in T, as shown in Figure 4.5.

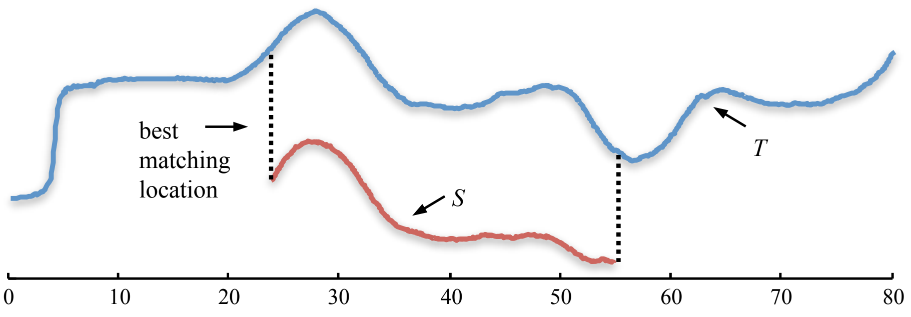


Figure 4.5: Illustration of the best matching location in time series T for subsequence S

As we shall explain in Section 4.3, our algorithm needs some metric to evaluate how well it can divide the entire combined dataset into two original classes. Here, we use concepts very similar to the *information gain* used in the traditional decision tree [7]. The reader may recall the original definition of entropy which we review here:

Definition 8. *Entropy.* A time series dataset $\mathbf{D}$ consists of data from two classes, labeled A and B (for example, A = "stinging nettles" and B = "false nettles"). Given that the proportion of time series objects in class A is $p(A)$ and the proportion of time series objects in class B is $p(B)$, the entropy of $\mathbf{D}$ is:

$$H(\mathbf{D}) = -p(A)log(p(A)) - p(B)log(p(B)). \tag{4.1}$$

Each splitting strategy divides the whole dataset $\mathbf{D}$ into two subsets, $\mathbf{D_1}$ and $\mathbf{D_2}$. Therefore, the information remaining in the entire dataset after splitting is defined by the weighted average entropy of each subset. If the fraction of objects in $\mathbf{D_1}$ is $f(\mathbf{D_1})$ and the fraction of objects in $\mathbf{D_2}$ is $f(\mathbf{D_2})$, the total entropy of $\mathbf{D}$ after splitting is $\hat{H}(\mathbf{D}) = f(\mathbf{D_1})H(\mathbf{D_1}) + f(\mathbf{D_2})H(\mathbf{D_2})$. This allows us to define the information gain for any splitting strategy:

Definition 9. *Information Gain.* Given a certain split strategy sp which divides $\mathbf{D}$ into two subsets $\mathbf{D_1}$ and $\mathbf{D_2}$, the entropy before and after splitting is $H(\mathbf{D})$ and $\hat{H}(\mathbf{D})$. So the information gain for this splitting rule is:

$$Gain(sp) = H(\mathbf{D}) - \hat{H}(\mathbf{D}), \tag{4.2}$$

$$Gain(sp) = H(\mathbf{D}) - (f(\mathbf{D_1})H(\mathbf{D_1}) + f(\mathbf{D_2})H(\mathbf{D_2})). \tag{4.3}$$

As we noted in the introduction, we use the distance to a *shapelet* as the splitting rule. The shapelet is a subsequence of a time series such that most of the time series objects in one class of the dataset are close to the shapelet under SUBSEQUENCEDIST, while most of the time series objects from the other class are far away from it.

To find the best shapelet, we may have to test many shapelet candidates. In the brute force algorithm discussed in Section 4.3, given a candidate shapelet, we calculate the distance between the candidate and every time series object in the dataset. We sort the objects according to the distances and find an optimal split point between two neighboring distances.

Definition 10. *Optimal Split Point (OSP).* A time series dataset $\mathbf{D}$ consists of two classes, A and B. For a shapelet candidate S, we choose some distance threshold d_{th} and split $\mathbf{D}$ into $\mathbf{D_1}$ and $\mathbf{D_2}$, such that for every time series object $T_{1,i}$ in $\mathbf{D_1}$, SUBSEQUENCEDIST$(T_{1,i}, S) < d_{th}$

and for every time series object $T_{2,i}$ in $\mathbf{D_2}$, $\textsc{SubsequenceDist}(T_{2,i}, S) \geq dth$. An *Optimal Split Point* is a distance threshold that

$$Gain(S, d_{OSP(\mathbf{D},S)}) \geq Gain(S, d'_{th}), \tag{4.4}$$

for any other distance threshold d'_{th}.

Using shapelets in a classifier requires two factors: a shapelet and the corresponding optimal split point. As a concrete example, in Figure 4.4 the shapelet is shown in red in the shapelet dictionary along with its optimal split point of 5.1.

We are finally in the position to formally define the shapelet.

Definition 11. *Shapelet.* Given a time series dataset $\mathbf{D}$ which consists of two classes, A and B, $shapelet(\mathbf{D})$ is a subsequence that, with its corresponding optimal split point,

$$Gain(shapelet(\mathbf{D}), d_{OSP(\mathbf{D},shapelet(\mathbf{D}))}) \geq Gain(S, d_{OSP(\mathbf{D},S)}), \tag{4.5}$$

for any other subsequence S.

Since the shapelet is simply *any* time series of some length less than or equal to the length of the shortest time series in our dataset, there is an infinite amount of possible shapes it could have. For simplicity, we assume the shapelet to be a subsequence of a time series object in the dataset. It is reasonable to make this assumption since the time series objects in one class presumably contain some similar subsequences, and these subsequences are good candidates for the shapelet.

Nevertheless, there are still a very large number of possible shapelet candidates. Suppose the dataset $\mathbf{D}$ contains k time series objects. We specify the minimum and maximum length of the shapelet candidates that can be generated from this dataset as *MINLEN* and *MAXLEN*, respectively. Obviously *MAXLEN* $\leq \min(m_i)$, where m_i is the length of the time series T_i from the dataset, $1 \leq i \leq k$. Considering a certain fixed length l, the number of shapelet candidates generated from the dataset is:

$$\sum_{T_i \in \mathbf{D}} (m_i - l + 1) \tag{4.6}$$

So the *total* number of candidates of all possible lengths is:

$$\sum_{MINLEN}^{MAXLEN} \sum_{T_i \in \mathbf{D}} (m_i - l + 1) \tag{4.7}$$

If the shapelet can be any length smaller than that of the shortest time series object in the dataset, the number of shapelet candidates is linear in k, and quadratic in $\bar{m}$, the average length of time series objects. For example, the well-known Trace dataset [61] has 200 instances of length 275. If we set *MINLEN* $= 3$ and *MAXLEN* $= 275$, there will be 7,480,200 shapelet candidates. For each of these candidates, we need to find its nearest neighbor within the k time series objects. Using the brute force search it will take approximately three days to accomplish this. However, as we will show in Section 4.3, we can achieve an identical result in a tiny fraction of this time with a novel pruning strategy.

4.3 Finding the Shapelet

We first show the brute force algorithm for finding shapelets, followed by two simple but highly effective speedup methods.

4.3.1 Brute-Force Algorithm

The most straightforward way for finding the shapelet is using the brute force method. The algorithm is described in Algorithm 8.

Algorithm 8 BRUTEFORCEJOIN(A, B)

```
1: $candidates \leftarrow$ GENERATECANDIDATES($\mathbf{D}$, MAXLEN, MINLEN, STEPSIZE)
2: $bsf\_gain \leftarrow 0$
3: for all $S \in candidates$ do
4:     $gain \leftarrow$ CHECKCANDIDATE($D, S$)
5:     if $gain > bsf\_gain$ then
6:         $bsf\_gain \leftarrow gain$
7:         $bsf\_shapelet \leftarrow S$
8:     end if
9: end for
10: return $bsf\_shapelet$
```

Given a combined dataset $\mathbf{D}$, in which each time series object is labeled either class A or class B, along with the user-defined maximum and minimum lengths of the shapelet, line 1 generates all of the subsequences of all possible lengths and stores them in the unordered list candidates. After initializing the best information gain bsf_gain to be zero (line 2), the algorithm checks how well each candidate in candidates can separate objects in class A and class B (lines 3 to 7). For each shapelet candidate, the algorithm calls the function CHECKCANDIDATE() to obtain the information gain achieved if that candidate is used to separate the data (line 4). As illustrated in Figure 4.6, we can visualize this as placing class-annotated points on the real number line, representing the distance of each time series to the candidate. Intuitively, we hope to find that this mapping produces two well-separated "pure" groups. In this regard the example in Figure 4.6 is very good, but clearly not perfect.

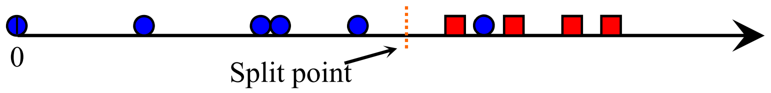


Figure 4.6: The CHECKCANDIDATE() function at the heart of the brute force search algorithm can be regarded as testing to see how mapping all of the time series objects on the number line, based on their SUBSEQUENCEDIST(T, S), separates the two classes

If the information gain is higher than bsf_gain, the algorithm updates bsf_gain and the corresponding best shapelet candidate $bsf\_shapelet$ (lines 5 to 8). Finally, the algorithm returns the candidate with the highest information gain in line 10.

The two subroutines GENERATECANDIDATES() and CHECKCANDIDATE() called in the algorithm are outlined in Algorithm 9 and Algorithm 10, respectively. In Algorithm 9, the algorithm GENERATECANDIDATES() begins by initializing the shapelet candidate pool to be an empty set and the shapelet length l to be *MAXLEN* (lines 1 and 2).

Thereafter, for each possible length l, the algorithm slides a window of size l across all of the time series objects in the dataset $\mathbf{D}$, extracts all of the possible candidates and adds them to the $pool$ (line 5). The algorithm finally returns the $pool$ as the entire set of shapelet

Algorithm 9 GENERATECANDIDATES($\mathbf{D}$, *MAXLEN, MINLEN, STEPSIZE*)

```
1: $pool \leftarrow \emptyset$
2: $l \leftarrow$ MAXLEN
3: while $l \geq$ MINLEN do
4:     for all $T \in \mathbf{D}$ do
5:         $pool \leftarrow pool \cup S_T^l$
6:     end for
7:     $l \leftarrow l-$ STEPSIZE
8: end while
9: return $pool$
```

candidates that we are going to check (line 9). In Algorithm 10 we show how the algorithm evaluates the utility of each candidate by using the information gain.

Algorithm 10 CHECKCANDIDATE($\mathbf{D}, S$)

```
1: $objects\_histogram \leftarrow \emptyset$
2: for all $T \in \mathbf{D}$ do
3:     $d \leftarrow$SUBSEQUENCEDIST($T, S$)
4:     insert $T$ into $objects\_histogram$ by the key $d$
5: end for
6: return CALCULATEINFORMATIONGAIN($objects\_histogram$)
```

First, the algorithm inserts into the histogram $objects\_histogram$ all of the time series objects according to the distance from the time series object to the candidate in lines 1 to 4. After that, the algorithm returns the utility of that candidate by calling CALCULATEINFORMATIONGAIN() (line 6).

The CALCULATEINFORMATIONGAIN() subroutine, as shown in Algorithm 11, takes an object histogram as the input, finds an optimal split point $split\_dist$ (line 1), and divides the time series objects into two subsets by comparing the distance to the candidate with $split\_dist$ (lines 4 to 7). Finally, it calculates the information gain (cf. definitions 6, 7) of the partition and returns the value (line 10).

Algorithm 11 CALCULATEINFORMATIONGAIN(obj_hist)

```
1:  $split\_dist \leftarrow$ OPTIMALSPLITPOINT($obj\_hist$)
2:  $\mathbf{D_1} \leftarrow \emptyset, \mathbf{D_2} \leftarrow \emptyset$
3:  for all $d \in obj\_hist$ do
4:      if $d.dist < split\_dist$ then
5:          $\mathbf{D_1} \leftarrow \mathbf{D_1} \cup d.objects$
6:      else
7:          $\mathbf{D_2} \leftarrow \mathbf{D_2} \cup d.objects$
8:      end if
9:  end for
10: return $H(\mathbf{D}) - \hat{H}(\mathbf{D})$
```

After building the distance histogram for all of the time series objects to a certain candidate, the algorithm will find a split point that divides the time series objects into two subsets (denoted by the dashed line in Figure 4.6). As noted in definition 10, an optimal split point is a distance threshold. Comparing the distance from each time series object in the dataset to the shapelet with the threshold, we can divide the dataset into two subsets which achieves the highest information gain among all of the possible partitions. Any point on the positive real number line could be a split point, so there are infinite possibilities from which to choose. To make the search space smaller, we check only the mean values of each pair of adjacent points in the histogram as a possible split point. This reduction still finds all of the possible information gain values since the information gain cannot change in the region *between* two adjacent points. Furthermore, in this way, we maximize the margin between two subsets.

The naïve brute force algorithm clearly finds the optimal shapelet. It appears that it is extremely space inefficient, requiring the storage of all of the shapelet candidates. However, we can mitigate this with some internal bookkeeping that generates and then discards the candidates one at a time. Nevertheless, the algorithm suffers from high time complexity.

Recall that the number of the time series objects in the dataset is k and the average length of each time series is $\bar{m}$. As we discussed in Section 4.2, the size of the candidate set is $\mathcal{O}(\bar{m}^2k)$. Checking the utility of one candidate requires its comparison with $\mathcal{O}(\bar{m}k)$ subsequences and each comparison (Euclidean distance) takes $\mathcal{O}(\bar{m})$ on average. Hence, the overall time complexity of the algorithm is $\mathcal{O}(\bar{m}^4k^2)$, which makes its usage for real-world problems intractable.

4.3.2 Subsequence Distance Early Abandon

In the brute force method, the distance from the time series T to the subsequence S is obtained by calculating the Euclidean distance of every subsequence of length $|S|$ in T and S and choosing the minimum. This takes $\mathcal{O}(|T|)$ distance calculations between subsequences. However, all we need to know is the *minimum* distance rather than all of the distances. Therefore, instead of calculating the exact distance between every subsequence and the candidate, we can stop distance calculations once the partial distance exceeds the minimum distance known so far. This trick is known as *early abandon*, which is very simple yet has been shown to be extremely effective for similar types of problems [44].

While the premise is simple, for clarity we illustrate the idea in Figure 4.7 and provide the pseudo code in Algorithm 12.

In line 1, we initialize the minimum distance min_dist from the time series T to the subsequence S to be infinity. Thereafter, for each subsequence S_i from T of length $|S|$, we accumulate the distance sum_dist between S_i and S one data point at a time (line 6). Once

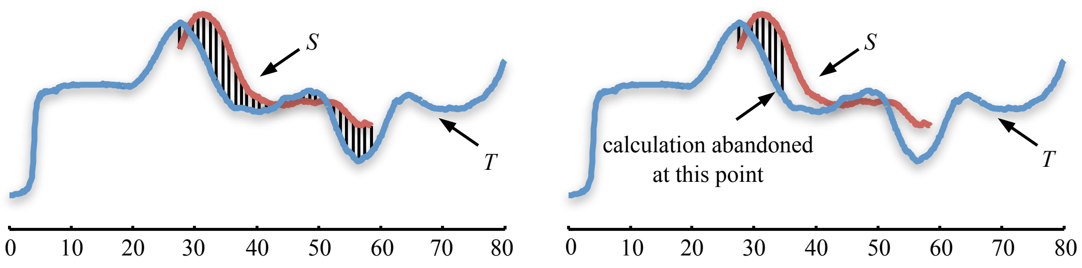


Figure 4.7: *Left*) Illustration of complete Euclidean distance. *Right*) Illustration of Euclidean distance early abandon

Algorithm 12 SUBSEQUENCEDISTANCEEARLYABANDON(T, S)

```
1: $min\_dist \leftarrow \infty$
2: for all $S_i \in S_T^{|S|}$ do
3:     $stop \leftarrow$ false
4:     $sum\_dist \leftarrow 0$
5:     for $k \leftarrow 1$ to $|S|$ do
6:         $sum\_dist \leftarrow sum\_dist + (S_{i(k)} - S_{(k)})^2$
7:         if $sum\_dist \geq min\_dist$ then
8:             $stop \leftarrow$ true
9:             Break
10:        end if
11:    end for
12:    if Not $stop$ then
13:        $min\_dist \leftarrow sum\_dist$
14:    end if
15: end for
16: return $min\_dist$
```

sum_dist is larger than or equal to the minimum distance known so far, we abandon the distance calculation between S_i and S (lines 7 to 9). If the distance calculation between S_i and S finishes, we know that the distance is smaller than the minimum distance known so far. Thus, we update the minimum distance min_dist in line 13. The algorithm returns the true distance from the time series T to the subsequence S in line 16. Although the early abandon search is still $\mathcal{O}(|T|)$, as we will demonstrate later, this simple trick reduces the average running time required by a large factor.

4.3.3 Admissible Entropy Pruning

Our definition of the shapelet requires some measure of how well the distances to a time series subsequence can split the data into two "purer" subsets. The readers will recall that we used the information gain (or entropy) as that measure. However, there are other commonly used measures for distribution evaluation, such as the Wilcoxon signed-rank test [80]. We adopted the entropy evaluation for two reasons. First, it is easily generalized to the multi-class problem. Second, as we will now show, we can use a novel idea called *early entropy pruning* to avoid a large fraction of distance calculations when finding the shapelet.

Obtaining the distance between a candidate and its nearest matching subsequence of each of the objects in the dataset is the most expensive calculation in the brute force algorithm, whereas the information gain calculation takes an inconsequential amount of time. Based on this observation, instead of waiting until we have all of the distances from each of the time series objects to the candidate, we can calculate an *upper bound* of the information gain based on the currently observed distances. If at any point during the search the upper bound cannot beat the best-so-far information gain, we stop the distance calculations and prune that particular candidate from consideration; we can be secure in the knowledge that it cannot be a better candidate than the current best so far.

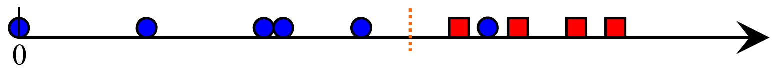


Figure 4.8: Distance arrangement of the time series objects in one-dimensional representation of best-so-far information gain. The positions of the objects represent their distances to the candidate

In order to help the readers understand the idea of pruning with an upper bound of the information gain, we consider a simple example. Suppose as shown in Figure 4.8, ten time series objects are arranged in a one-dimensional representation by measuring their distances to the best-so-far candidate. This happens to be a good case, with five of the six objects from class A (represented by circles) closer to the candidate than any of the four objects from class B (represented by squares). In addition, of the five objects to the right of the split point, only one object from class A is mixed up with the class B. The optimal split point is represented by a vertical dashed line, and the best-so-far information gain is:

[-(6/10)log(6/10)-(4/10)log(4/10)] - [(5/10)[-(5/5)log(5/5)-(0/5)log(0/5)]+(5/10)[-(4/5)log(4/5)-(1/5)log(1/5)]]=0.4228

We now consider another candidate. The distances of the first five time series objects to the candidate have been calculated, and their corresponding positions in a one-dimensional representation are shown in Figure 4.9.

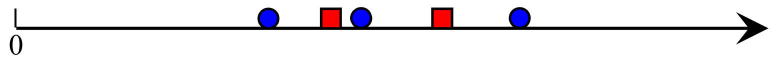


Figure 4.9: The arrangement of first five distances from the time series objects to the candidate

We can ask the following question: of the 30,240 distinct ways the remaining five distances could be added to this line, could any of them result in an information gain that is better than the best-so-far? In fact, we can answer this question in constant time. The idea is to imagine the most optimistic scenarios and test them. It is clear that there are only two optimistic possibilities: either all of the remaining class A objects map to the far right and all of the class B objects map to the far left, or vice versa. In particular, given the placement of the first five objects, one of these two scenarios *must* maximize the probability of best separating the objects from two classes and thus maximize the information gain. Figure 4.10 shows the former scenario applied to the example shown in Figure 4.9.

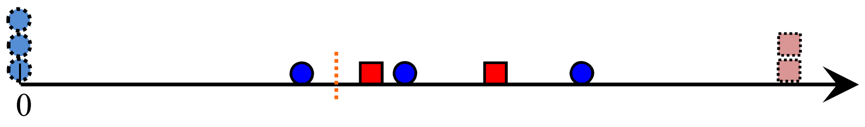


Figure 4.10: One optimistic prediction of distance distribution based on distances that have already been calculated in Figure 4.9. The dashed objects are in the optimistically assumed placements

The information gain of the better of the two optimistic predictions is:

[-(6/10)log(6/10)-(4/10)log(4/10)] - [(4/10)[-(4/4)log(4/4)-(0/4)log(0/4)]+(6/10)[-(4/6)log(4/6)-(2/6)log(2/6)]]=0.2911,

which is lower than the best-so-far information gain. Therefore, at this point, we can stop the distance calculation for the remaining objects and prune this candidate from consideration forever. In this case, we saved 50% of the distance calculations. But in real-life situations, early entropy pruning is generally much more efficient than we have shown in this brief example. In Section 4.5, we will empirically evaluate the time we save.

This intuitive idea is formalized in the algorithm outlined in Algorithm 13. The algorithm takes as the inputs: the best-so-far information gain, the calculated distances from objects to the candidate organized in the histogram (i.e the number line for Figures 4.8, 4.9 and 4.10), and the remaining time series objects in class A and class B. It returns True if we can prune the candidate as the answer. The algorithm begins by finding the two ends of the histogram (discussed in Section 4.2). For simplicity, we set the distance values at the two ends at 0 and maximum distance +1 (in lines 1 and 2). To build the optimistic histogram of the whole dataset based on the existing one (lines 3 and 8), we assign the remaining objects of one class to one end and those of the other class to the other end (lines 4 and 9). If in either case, the information gain of the optimistic histogram is higher than the best-so-far information gain (lines 5 and 10), it is still possible that the actual information gain of the candidate can beat the best so far. Thus, we cannot prune the candidate and we should continue with the test (lines 6 and 11). Otherwise, if the upper bound of the actual information gain is lower than the best so far, we are saved from all of the remaining distance calculations with this candidate (line 13).

Algorithm 13 ENTROPYEARLYPRUNE($bsf\_gain, dist\_hist, c_A, c_B$)

```
1: $minend \leftarrow 0$
2: $maxend \leftarrow$ largest distance value in $dist\_hist + 1$
3: $pred\_dist\_hist \leftarrow dist\_hist$
4: Add to the $pred\_dist\_hist$, $c_A$ at $minend$ and $c_B$ at $maxend$
5: if CALCULATEINFORMATIONGAIN($pred\_dist\_hist$) $> bsf\_gain$ then
6:     return false
7: end if
8: $pred\_dist\_hist \leftarrow dist\_hist$
9: Add to the $pred\_dist\_hist$, $c_A$ at $maxend$ and $c_B$ at $minend$
10: if CALCULATEINFORMATIONGAIN($pred\_dist\_hist$) $> bsf\_gain$ then
11:     return false
12: end if
13: return true
```

The utility of this pruning method depends on the data. If there is any class-correlated structure in the data, we will typically find a good candidate that gives a high information gain early in our search, and thereafter the vast majority of candidates will be pruned quickly.

There is one simple trick we can do to get the maximum pruning benefit. Suppose we tested all of the objects from class A first, then all of the objects from class B. In this case, the upper bound of the information gain must always be maximum until at least after the point at which we have seen the first object from class B. We therefore use a round-robin algorithm to pick the next object to be tested. That is to say, the ordering of objects we use is $a_1, b_1, a_2, b_3, \ldots, a_n, b_n$. This ordering lets the algorithm know very early in the search if a candidate cannot beat the best so far.

4.3.4 Techniques for Breaking Ties

It is often the case that different candidates will have the same best information gain. This is particularly true for small datasets. We propose several options to break this tie depending on applications. We can break such ties by favoring the longest candidate, the shortest candidate, or the one that achieves the largest margin between the two classes.

Longest Candidate

The longest candidate always contains the entire distinguishing feature that is in one class and absent from the other class. However, it is possible that the longest candidate might also contain some irrelevant information or noise near the distinguishing feature subsequence; this is likely to reduce the accuracy in some applications.

Shortest Candidate

In contrast, favoring the shortest candidate can avoid noise in the shapelet. The shortest candidate is useful when there are multiple, discontinuous features in one class. To enumerate each of these shapelets, each time after the algorithm finds a shapelet we replace the feature subsequences in all of the time series objects with random walk subsequences of the same length and rerun the algorithm. By running the algorithm multiple times like this, the method will return short, discontinuous shapelets.

Maximum Separation

Using the maximum separation to break the tie follows the same basic idea of SVMs. The algorithm finds the shapelet that maximizes the distance between the two classes. We calculate the distance between two classes by: first, the mean distances from the time series objects in each individual class to the shapelet, then return the difference between the mean distances as the distance between two classes. Based on comprehensive experiments, the best accuracy is most often achieved when breaking the tie using the shapelet that has the maximum margin, which is intuitive, since this method maximizes the difference between classes.

4.4 Shapelets for Classification

While we believe that shapelets can have implications for many time series data mining problems, including visualization, anomaly detection, and rule discovery, for concreteness we will focus just on classification in this work. Classifying by a shapelet and its corresponding split point makes a binary decision as to whether a time series object belongs to a certain class or not. Obviously, this method is not enough to deal with a multi-class situation. Even with two-class problems, a linear classifier is sometimes inadequate. In order to make the shapelet classifier universal, we frame it as a decision tree [7].

At each step of the decision tree induction, we determine the shapelet and its corresponding split point over the training subset considered in that step. (A similar idea is illustrated in [29].)

After the learning procedure finishes, we can assess the performance of the shapelet classifier by calculating the accuracy of the testing dataset. The way we predict the class label of each testing time series object is very similar to the way this is done with a traditional decision tree. For completeness the algorithm is described in Algorithm 14.

Algorithm 14 CALCULATEACCURACY$(classifier\ \mathbf{C}, dataset\ \mathbf{D_t})$

```
1: for all $T \in \mathbf{D}$ do
2:     $predict\_class\_label \leftarrow$ PREDICT$(C, T)$
3:     if $predict\_class\_label = true\_label$ then
4:         $correct \leftarrow correct + 1$
5:     end if
6: end for
7: return $corrent/\mathbf{D_t}$
```

The technique to predict the class label of each testing object is described in Algorithm 15. For each internal node of the decision tree we have the information of a single shapelet classifier, the left subtree, and the right subtree. The leaf nodes contain information about a predicted class label. Starting from the root of a shapelet classifier, we calculate the distance from the testing object T to the shapelet in that node. If the distance is smaller than the split point, we recursively use the left subtree (lines 6 and 7) or otherwise use the right subtree (lines 8 and 9). This procedure continues until we reach a leaf node and return the predicted class label (lines 1 and 2).

Algorithm 15 PREDICT($classifier\ \mathbf{C}, timeseries\ T$)

```
1:  if C is a leaf node then
2:      return label of C
3:  else
4:      $S \leftarrow$ shapelet on the root node of C
5:      $split\_point \leftarrow$ split point on the root node of C
6:      if SUBSEQUENCEDISTANCEEARLYABANDON($T, S$) $< split\_point$ then
7:          PREDICT(left subtree of C, $T$)
8:      else
9:          PREDICT(right subtree of C, $T$)
10:     end if
11: end if
```

4.5 Experimental Evaluation

4.5.1 Experiment Setup

Experiment Methodology

We emphasize the importance of statistical validity in the design of the experiments. Our case studies include several real-world datasets from various domains such as shape, spectrography, motion capture, as well as synthetic datasets. For all of these datasets except the Mallat and Coffee datasets, we compare the shapelet classifier with the (rotation invariant where appropriate) one-nearest neighbor classifier. For the Mallat dataset, the shapelet classifier is compared with the decision tree classifier presented in a previous work [38]. For the Coffee dataset, the shapelet classifier is compared with linear discriminant analysis [52] utilized in a previous work [41]. Note that recent extensive empirical evaluations have shown that the one-nearest neighbor classifier is (by a significant margin) the best general time series classifier [82].

We use separate subsets for training and testing. The performance comparisons are averaged over ten randomly split runs. For all of the case studies where there is no fixed training/testing split from other literatures which we can compare our result to, the reported shapelets are from a randomly selected training/testing split.

In the shapelet classifier, there are four parameters: *MAXLEN*, *MINLEN*, *STEPSIZE*, and *SEPERATION METHOD*. However, there is essentially zero effort in tuning the parameters. For *MAXLEN*, we always set the longest possible length to the length of the shortest time series in the dataset. For *MINLEN*, we hardcoded the shortest possible length to three since three is the minimum meaningful length. The only special case in our experiments is in the lightning dataset, which is the largest dataset considered. Here, based on a one-time visual inspection of the data, *MINLEN* is set at 1/20 of the entire length of time series and *MAXLEN* at 1/10 of the entire length of time series. The reason for this exception is to reduce the size of the search space, thus making the experiment tenable. The only parameter which really needs to be decided is *SEPERATION METHOD*, which we have discussed in Section 4.3.4. This parameter can be decided using domain knowledge of the application. In the case that the user only cares about the classification accuracy, we can try all three and pick the one with highest training accuracy. In our experiments, we tried each of the three separation methods and picked the one with highest accuracy on the testing data. The slight difference in accuracy using different separation methods are reported in [92]. For tractability, we have the parameter of *STEPSIZE*, which defines the possible lengths of shapelet candidates. We increase the *STEPSIZE* parameter in large datasets: 50 for the lightning dataset, 10 for the wheat dataset, and one for all of the other datasets. In our free code for shapelet discovery, we offer an interface that allows the user to set these parameters.

Performance Metrics

The performance metrics are the running time, the accuracy, and (somewhat subjectively) the interpretability of the results. There are two aspects of the running time: classification running time and model learning time. We compare the classification runtime with the one-nearest neighbor classifier's and the model learning time with the original brute force algorithm's (cf. Algorithm 8). The accuracy is compared against the (rotation invariant) one-nearest neighbor classifier if the dataset has not been studied by other researchers in the past. Otherwise, we compare the classification accuracy with previous works.

Reproducibility

We have designed and conducted all experiments such that they are easily reproducible. With this in mind, we have built a webpage [92] which contains all of the datasets and code used in this work, spreadsheets which contain the raw numbers displayed in all of the figures, larger annotated figures showing the decision trees, etc.

4.5.2 Performance Comparison

We test the scalability of our shapelet finding algorithm on the Synthetic Lightning EMP Classification [36], which, with a 2,000/18,000 training/testing split, is the largest class-labeled time series dataset we are aware of. It also has the highest dimensionality, with each time series being 2,000 data points long. Using four different search algorithms, we started by finding the shapelet in a subset of just ten time series objects. We then iteratively double the size of the data subset until the time for brute force made the experiments untenable. Figure 4.11 and 4.12 shows the results. Each computational time and classification accuracy is averaged over ten runs, with each run a subset is randomly selected from the training dataset.

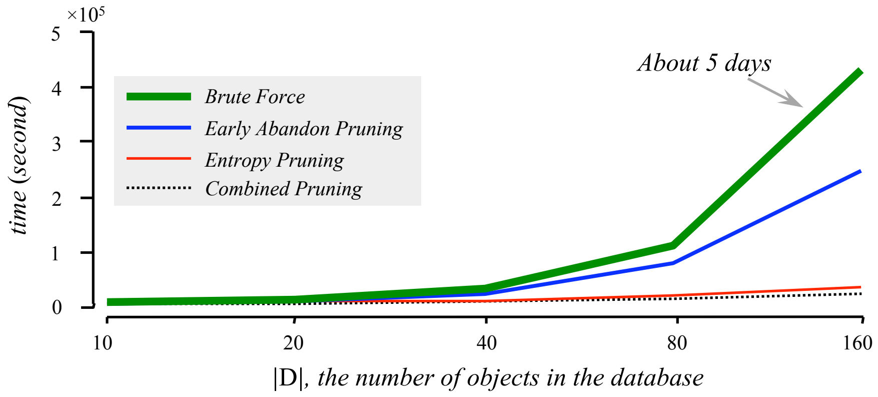


Figure 4.11: The time required to find the best shapelet for increasing large dataset sizes

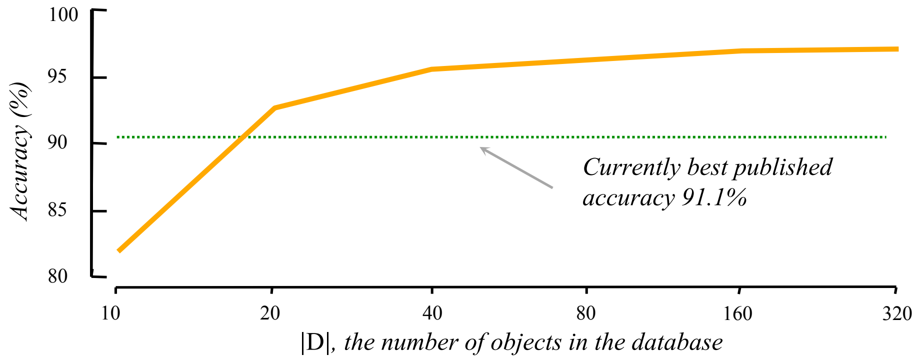


Figure 4.12: The hold-out accuracy for increasing large dataset sizes

The results show that the brute force search quickly becomes untenable, requiring about five days for just 160 objects. Early abandoning helps reduce this by a factor of two, and entropy-based pruning helps reduce this by over one order of magnitude. Combining both ideas produces a two orders of magnitude speedup when the training set is the size of 160.

For different sizes of training datasets, we study how the various lengths of time series would affect the computational time. To keep the original properties and distributions of the data, data points in each of time series are resampled to the lengths of 250, 500, and 1000. The computational time of different sampled lengths of the same datasets is reduced by early abandon pruning. As illustrated in Figure 4.13, the practical complexity is approximately quadratic on the length of time series, while the theoretical worst-case complexity is $\mathcal{O}(m^4)$ on the length of time series.

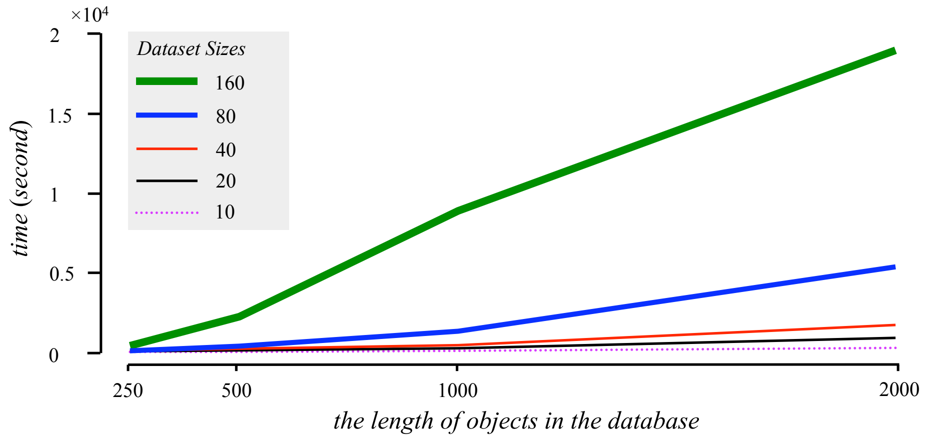


Figure 4.13: The time required to find the best shapelet for increasing lengths of time series objects in the dataset, with various dataset sizes

For each size of the data subset we considered, we also built a decision tree (which can be seen at [92]) and tested the accuracy on the 18,000 holdout data. When only 10 or 20 objects (out of the original 2,000) are examined, the decision tree is slightly worse than the best known result on this dataset [36] (using the one-nearest neighbor Euclidean distance), but after examining just 2% of the training data, it is significantly more accurate. The comparison to the results in dataset [36] is fair, since we use exactly the same training/testing split as they do. Note that their best result, which we so easily beat, is the best of eight diverse and highly optimized approaches they consider.

4.5.3 Projectile Points (Arrowheads)

Projectile point (arrowhead) classification is an important topic in anthropology (see [92] where we have an extensive review of the literature). Projectile points can be divided into different classes based on the location in which they are found, the group that created them, the date they were in use, etc. In Figure 4.14, we show some samples of the projectile points used in our experiments.

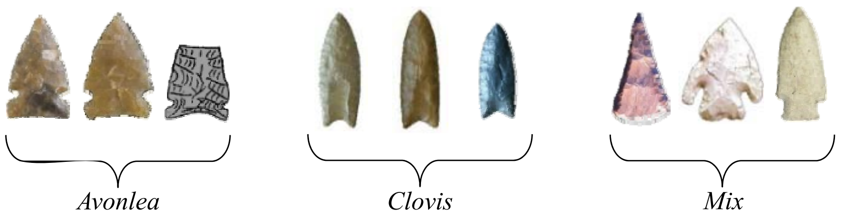


Figure 4.14: Examples of the three classes of projectile points in our dataset. The testing dataset includes some broken points and some drawings taken from anthropologists' field notes

As shown in Figure 4.15 and confirmed by physical anthropologists Dr. Sang-Hee Lee and Taryn Rampley of UCR, the Clovis projectile points can be distinguished from the others by an un-notched hafting area near the bottom connected by a deep concave bottom end. After distinguishing the Clovis projectile points, the Avonlea points are differentiated from the mixed class by a small notched hafting area connected by a shallow concave bottom end.

The shapelet classifier achieves an accuracy of 80.0%, whereas the accuracy of r*otation invariant* one-nearest neighbor classifier is 68.0%. Beyond the advantage of greater accuracy, the shapelet classifier produces the classification result 3×10^3 times faster than the *rotation*

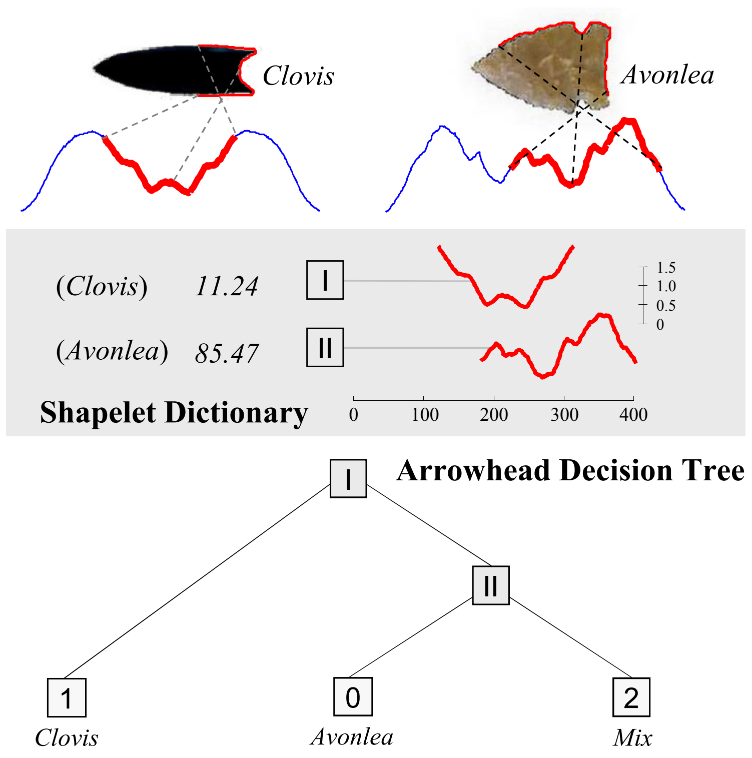


Figure 4.15: *Top*) The dictionary of shapelets together with the thresholds d_{th}. The x-axis shows the position of the shapelets in the original time series. *Bottom*) The decision tree for the 3-class projectile points problem

invariant one-nearest neighbor classifier and is more robust in dealing with the pervasive broken projectile points in the collections.

4.5.4 Mining Historical Documents

In this section we consider the utility of shapelets for an ongoing project in mining and annotating historical documents. Coats of arms or heraldic shields were originally symbols used to identify individuals or groups on the battlefield. Since the beginning of the Middle Ages, thousands of annotated catalogues of these shields have been created and in recent years hundreds of them have been digitized [3, 75]. Naturally, most efforts to automatically extract and annotate these volumes concentrate on the colors and patterns of the shields; however, there is also useful information contained in the shape. Consider for example Figure 4.16, which shows examples of different shapes commonly associated with various countries and heraldic traditions.

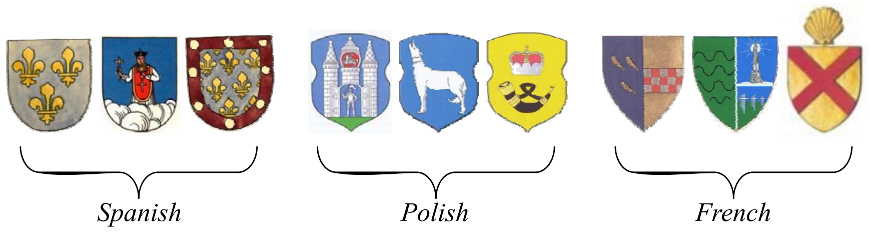


Figure 4.16: Examples of the three classes in our dataset. The shields were hand-drawn one to six centuries ago

Note that in most of these documents, the shields were drawn freehand and thus have natural variability in shape in addition to containing affine transformation artifacts introduced during the digital scanning.

We convert the shapes of the shields to a time series using the angle-based method. Because some shields may be augmented with ornamentation (i.e. far right in Figure 4.16) or torn (i.e. Figure 4.18) and thus may have radically different perimeter lengths, we do not normalize the time series lengths.

We randomly select ten objects from each class as the training dataset, and leave the remaining 129 objects for testing. The final classifier is shown in Figure 4.17.

Note that we can glean some information about this dataset by "brushing" the shapelet back onto the shields as in Figure 4.17.*Top*. For example, both the Spanish and the French

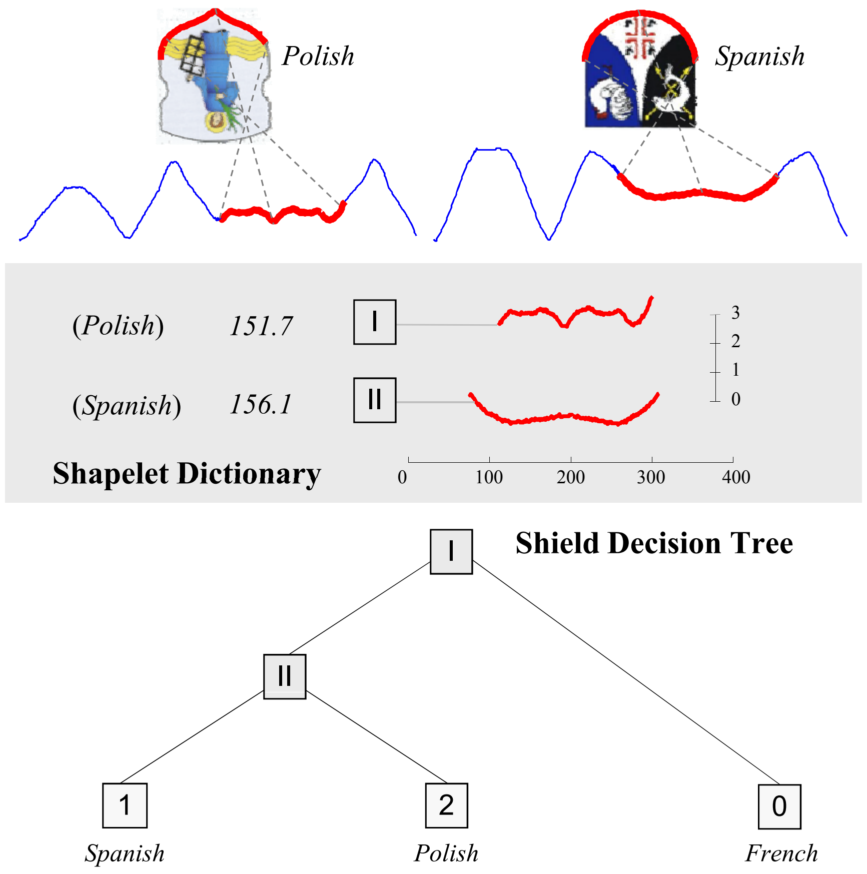


Figure 4.17: *Top*) The dictionary of shapelets, together with the thresholds d_{th}. The x-axis shows the position of the shapelets in the original time series. *Bottom*) A decision tree for heraldic shields

shields have right angle edges at the top of the shield, so the shapelet algorithm does *not* choose that common feature to discriminate between the classes. Instead, the unique semi-circular end of the Spanish crest is used in node II to discriminate it from the French examples.

For our shapelet classifier, we achieve 89.9% accuracy using a maximum mean separation method to break ties while for the rotation invariant one-nearest neighbor Euclidean distance classifier the accuracy is only 82.9%. Beyond the differences in accuracy, there are two additional advantages of shapelets. First, the time to classify is approximately 3×10^4 times faster than for the rotation invariant one-nearest neighbor Euclidean distance, although we could close that difference somewhat if we indexed the training data [16]. Second, as shown in Figure 4.18, many images from historical manuscripts are torn or degraded. Note that the decision tree shown in Figure 4.17 can still correctly classify the shield of Charles II, even though a large fraction of it is missing.

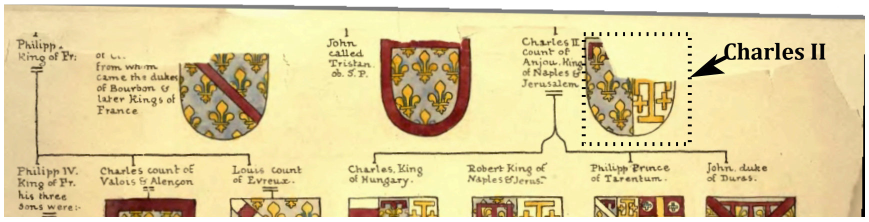


Figure 4.18: The top section of a page of the 1840 text, *A guide to the study of heraldry* [53]. Note that some shields are torn

4.5.5 Understanding the Gun/NoGun Problem

The *Gun/NoGun* motion capture time series dataset is perhaps the most studied time series classification problem in the literature [22, 82]. The dataset consists of 100 instances from each class. In the *Gun* class, the actors have their hands by their sides, draw a gun from a hip-mounted holster, point it at a target for approximately one second, and then return the gun to the holster and their hands to their sides. In contrast, in the *NoGun* class, actors do the similar hands-down, point, hold, and return motion without the gun in their hands and therefore are pointing to a target using the index finger. Our classification problem is to distinguish these two very similar types of motions. The problem is made somewhat more complicated by the fact that there are two actors, who differ in height (by 12 inches) and "style".

We take the standard training/testing split for this dataset and use it to learn the decision tree shown in Figure 4.19.

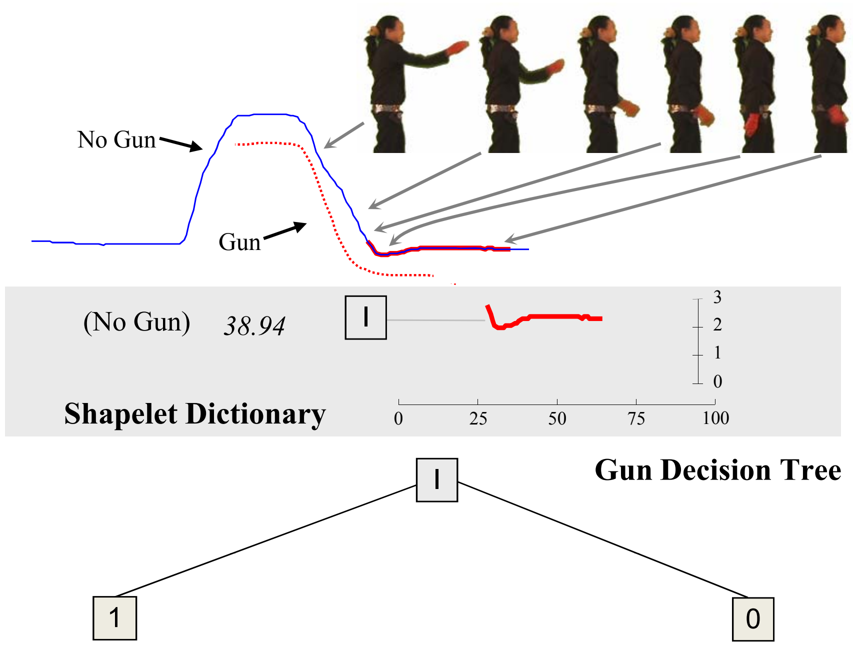


Figure 4.19: *Top*) The dictionary of shapelets, with the thresholds d_{th}. The x-axis shows the position of the shapelet in the original time series. *Bottom*) The decision tree for the Gun/NoGun problem

The holdout accuracy for the decision tree is 93.3%, beating the one-nearest neighbor Euclidean distance classifier, whose accuracy is 91.3%, and unconstrained or constrained DTW [22, 82], with accuracies of 90.7% and 91.3%, respectively. More significantly, the time to classify using the decision tree is about four times faster than the one-nearest neighbor Euclidean distance classifier. This is significant, since surveillance is a domain where classification speed can matter.

Moreover, by "brushing" the shapelet back onto the original video, we are able to gain some understanding of the differences between the two classes. In Figure 4.19, we can see that the *NoGun* class has a "dip" where the actor puts her hand down by her side, and inertia carries her hand a little too far and she is forced to correct for it (a phenomenon known as "overshoot"). In contrast, when the actor has the gun, she returns her hand to her side more carefully, feeling for the gun holster, and no dip is seen.

4.5.6 Wheat Spectrography

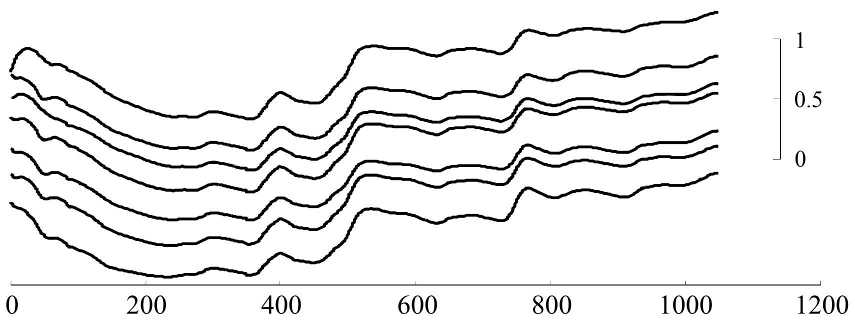


Figure 4.20: One sample from each of the seven classes in the wheat problem. The objects are separated in the y-axis for visual clarity, as they all have approximately the same mean

This dataset consists of 775 spectrographs of wheat samples grown in Canada between 1998 and 2005. The data is made up of several different types of wheat, including *Soft White Spring, Canada Western Red Spring, Canada Western Red Winter*, etc. However, the class label given for this problem is the year in which the wheat was grown. This makes the classification problem very difficult, as some of the similarities/dissimilarities between objects can be attributed to the year grown, but some can be attributed to the wheat type, which we do not know. In Figure 4.20 we plot one example from each class; as the reader can see, the differences between classes are *very* subtle.

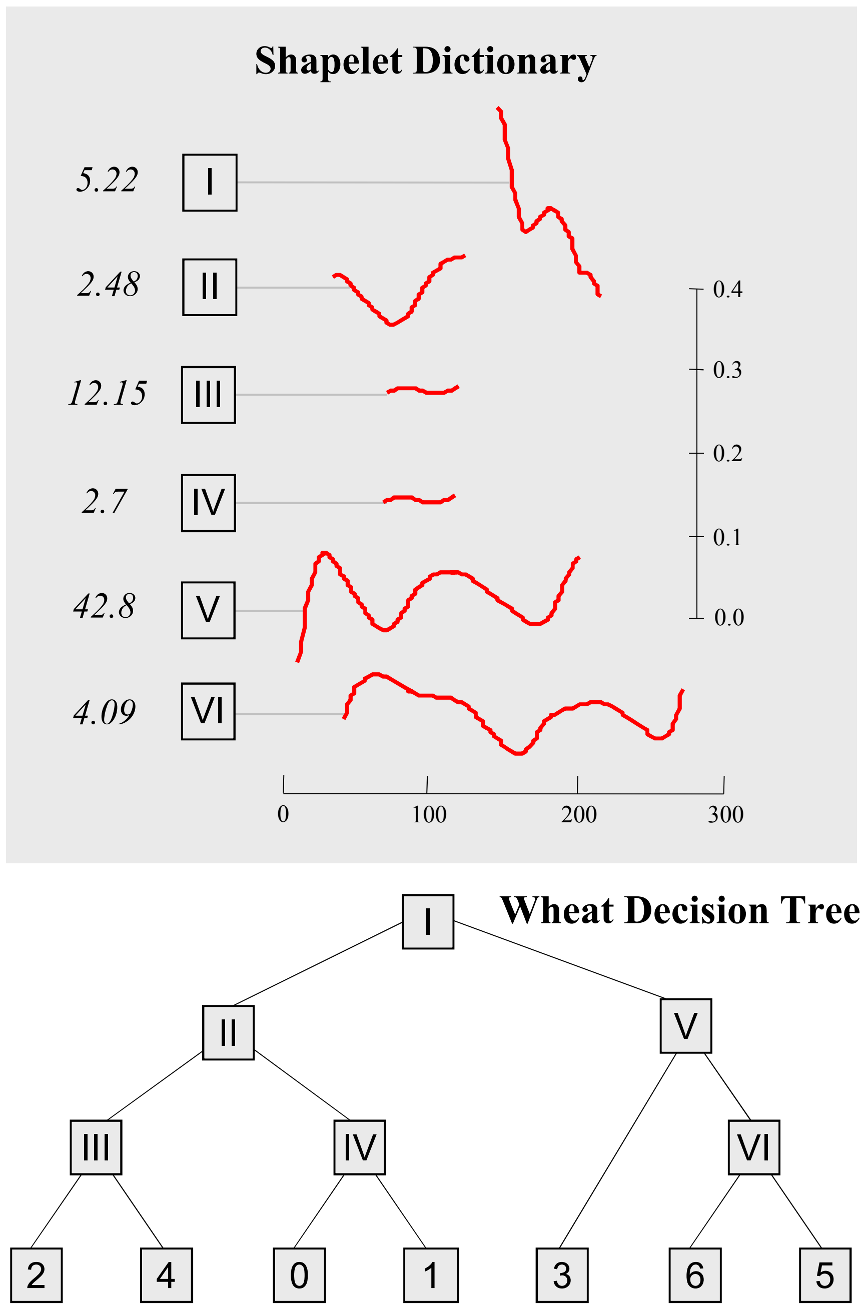


Figure 4.21: *Top*) The dictionary of shapelets, with the thresholds d_{th}. The x-axis shows the position of the shapelet in the original time series. *Bottom*) The decision tree for the wheat spectrography problem

We create a 49/726 training/testing split, ensuring that the training set has seven objects from each class, and then test the classification accuracy of the one-nearest neighbor Euclidean distance classifier, which we find to be 44.1% (Dynamic Time Warping does not outperform Euclidean distance here). We then create a decision tree for the data, using the algorithm introduced in Section 4.4. The output is shown in Figure 4.21.

The accuracy of the decision tree is 72.6%, which is significantly better than the 44.1% achieved by the nearest neighbor method.

4.5.7 Coffee Spectrography

The two main species of coffee cultivated worldwide are *Arabica* (90%) and *Canephora variant Robusta* (9%) [41]. They are different in ingredients, flavor, cultivated environment, and commercial value. Arabica has a finer flavor, and thus is valued higher than Robusta; Robusta is less susceptible to disease and is cultivated as a substitute for Arabica. Robusta also contains about 40∼50% more caffeine than Arabica.

In this experiment, 56 pure samples of freeze-dried coffee samples are analyzed under DRIFT [81] analysis. The data is obtained from the work [41]. We split the data into 28 samples (14 Arabica and 14 Robusta) for training and 28 samples (15 Arabica and 13 Robusta) for testing. The resulting shapelet decision tree is shown in Figure 4.22.

As stated in [41], bands between the region 1550∼1775 cm^{-1} represent the ingredient of caffeine. Since the spectrography data we experiment on is down-sampled, the corresponding region of bands 1550∼1775 cm^{-1} is 187.7∼247.3. The shapelet found by our algorithm is

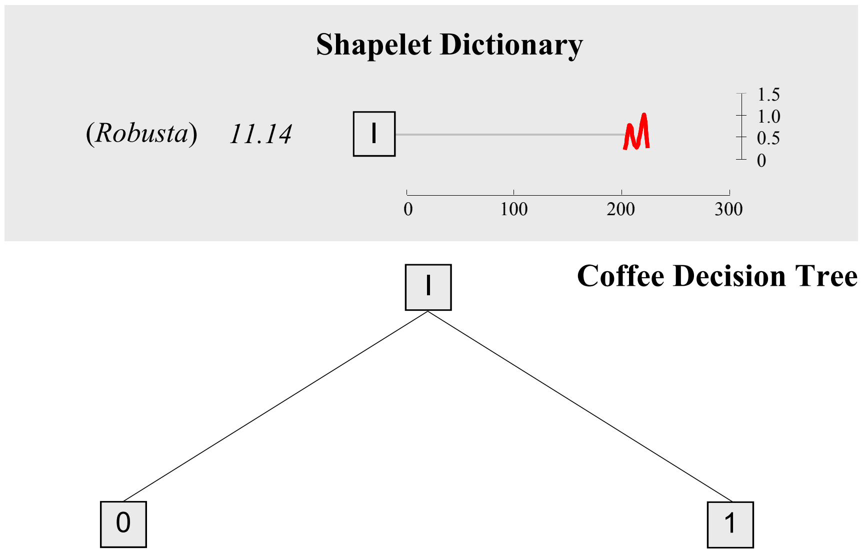


Figure 4.22: *Top*) The dictionary of shapelets, with the thresholds d_{th}. The x-axis shows the position of the shapelet in the original time series. *Bottom*) The decision tree for the coffee problem

between 203∼224, which reveals the major ingredient difference between the Arabica and the Robusta.

In the original paper, using the PCA method, the cross-validation accuracy is 98.8%. In our experiment, we are using much less training data, but achieving perfect accuracy.

4.5.8 Mallat

The Mallat dataset [38] consists of eight classes of curves, one piecewise normal pattern, and its seven fault cases. One example in each class is presented in Figure 4.23. For each of the eight classes, three hundred various replicated curves were generated. These curves were generated from shifting the original curve t time-units ($t = 5, 10, 15, 20, 25, 30$) and adding an Gaussian random noise $N(0, \sigma^2)$ to the entire time series, where $\sigma = 0.1$.

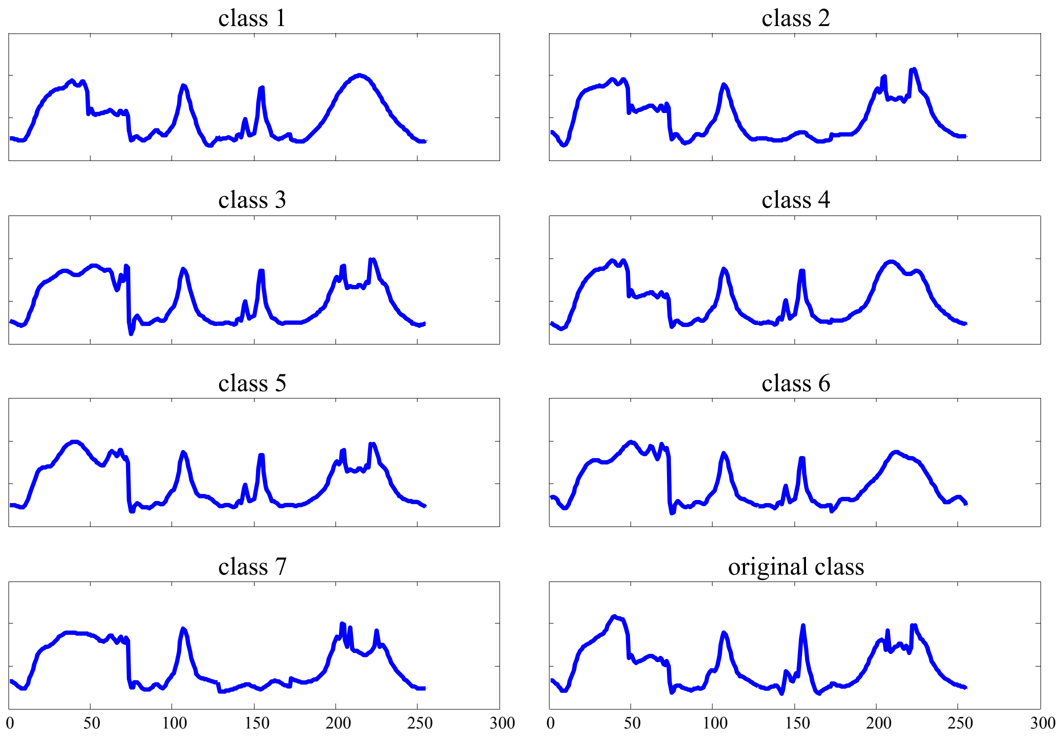


Figure 4.23: One sample from each of the eight classes in the Mallat problem

The original creators of this dataset use a CART decision tree which splits the data based on the features derived from wavelet coefficients. They were able to achieve an error rate of 2.25% by the 1/3 training, 2/3 testing split.

In our experiment, we only randomly use 2/15 as training data and leave all the remaining as the testing data and achieve an error rate of 1.5%. In other words, we can use less than half of the data the original authors used and still decrease the error rate by a large margin.

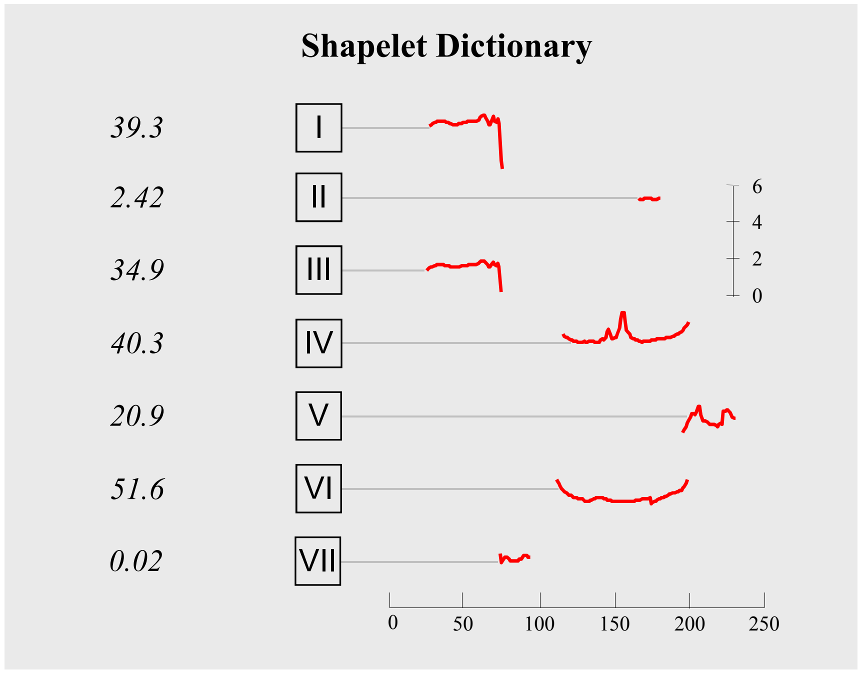
The shapelet decision tree is shown in Figure 4.24. Most of the shapelets in our decision can be easily interpreted. The first shapelet covers the first rectangular dip. All the data objects on the left have either no dip or a shallow one, while objects on the right of the tree have a deeper dip. The third shapelet on the decision tree is similar to the first one; it distinguishes the time series objects that are without dip from those with a shallow dip. The fourth shapelet shows that the most critical difference between the objects in class 5 and 6 is whether the second and the third peaks are present. A similar interpretation also exists in class 1 and 7, and the corresponding shapelet is shown as the sixth node. The fifth shapelet presents the difference in the last peak. The objects on the left have a deep dip and those on the right have no dip or a shallow one.

4.5.9 Gait Analysis

Up to this point we have only considered one-dimensional time series for simplicity. However, as the reader will appreciate, only very minor tweaks are required to allow shapelet discovery in k-dimensional time series. In this section we consider a two-dimensional time series problem in gait recognition.

We do gait analysis using CMU Graphics Lab Motion Capture Database [16]. We labeled the motions in the database containing the keyword “walk” in their motion descriptions into two categories; first is the normal walk, with only “walk” in the motion descriptions, and the other is the abnormal walk, with the motion descriptions containing: “hobble walk,” “walk wounded leg,” “walk on toes bent forward,” “hurt leg walk,” “drag bad leg walk”, or “hurt stomach walk.” In the abnormal walks, the actors are pretending to have difficulty walking normally.

There are several difficulties in gait classification. First, the data items are multivariate time series. The distance measurement for multivariate time series is unclear because indi-



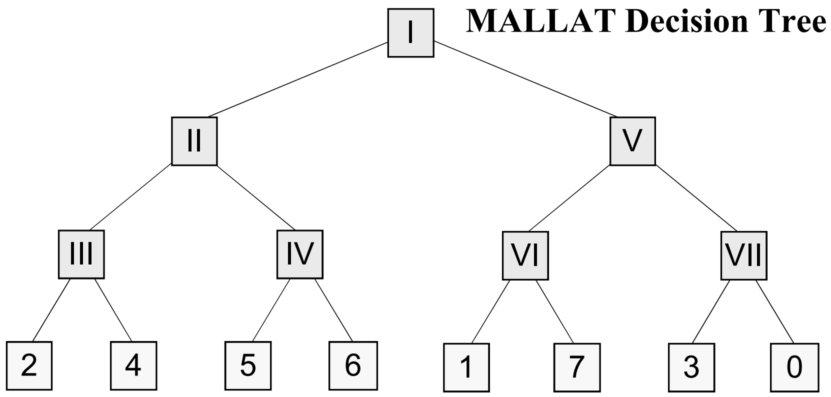


Figure 4.24: *Top*) The dictionary of shapelets, with the thresholds d_{th}. The x-axis shows the position of the shapelet in the original time series. *Bottom*) The decision tree for the Mallat problem

vidual dimensions are correlated. Second, the time recorded for each walking motion varies a lot. Since the walking motions are collected from different subjects and used for different purposes in the original database, some short walks last only three seconds and the other long walks may last as long as 52 seconds. Third, the intra-class variance is large. The walking time series in the normal walk class are collected at least from four actors, everyone walking at their own pace and with their own walking style. Moreover, there are other motions that intermingle with walking such as stops and turns.

We simplify the multivariate motion data by considering only the z-axis value of the markers on the left and right toes. Sliding the window over a two-variable time series item, we get pairs of subsequences within the same sliding window. We use these pairs of subsequences as the shapelet candidates. As an illustration, an item of a walking motion is shown in Figure 4.25. The two lines presented the z-axis values of the left and right toes of a walking motion. The pair of subsequences from the two time series extracted by the same sliding window is a shapelet candidate.

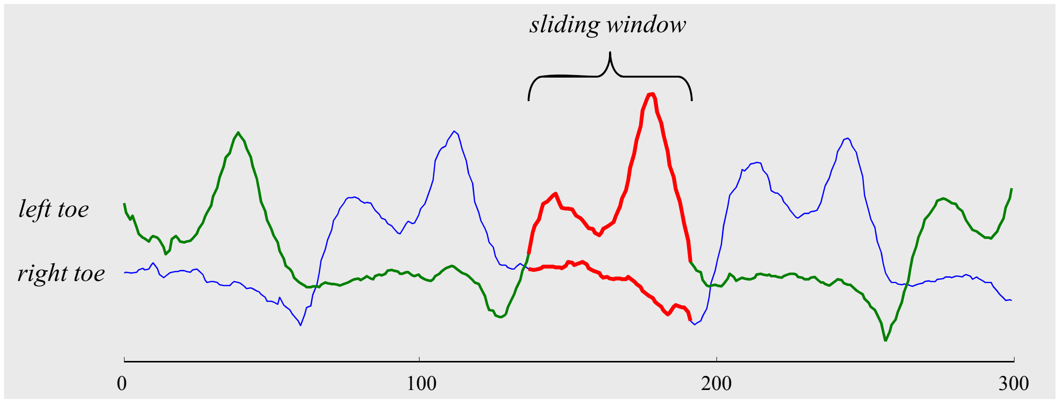


Figure 4.25: The illustration of a data item of a walking motion. Each data item consists of two time series in the same time interval. Using a sliding window across the data item, we get the pairs of subsequences as the shapelet candidates

The distance between the data item and the shapelet candidate is defined as

$$\textsc{MultiSubsequenceDist}(T, S) = \min(\sum_{v=1}^{2} \textsc{Dist}(S_v, S'_v)), \tag{4.8}$$

where $S' \in \mathbf{S}_{\mathbf{T}}^{|\mathbf{S}|}$, and S_v is time series of the vth variant of S.

We randomly pick nine walking motions in each class as the training set and leave the others as the testing set. We test with three different segmentations of the time series. In the first segmentation, we use the entire data item unless the time series is too long. If the time series is too long, we cut it into shorter sequences of 500 data points. In the second segmentation, we make each item start at the peak of the time series of the left toe and make the item as long as three cycles. For those data items that are less than three cycles, we make the item as long as they last. In the third segmentation, we make each item start at the wave trough of the time series of the left toe, and make each item exactly two gait cycles long, abandoning those less than two cycles. The accuracy of different segmentation methods is shown in Figure 4.26. We compare the accuracy between shapelet classification and *rotation invariant* nearest neighbor. The result shows that if we use merely the raw data, the nearest neighbor achieves an accuracy of 53.5% while the shapelet achieves an accuracy of 86%. Only when very careful segmentation is preprocessed can the nearest neighbor classification perform as well as the shapelet classification in accuracy, both at 90% 91%. However, the shapelet still outperforms the nearest neighbor by having the classification done 45 times faster.

Figure 4.27 shows the shapelet that represents exactly one gait cycle. With shapelet classification, it deals with different time intervals of walking motion effectively, and therefore reduces the amount of human labor needed for segmentation.

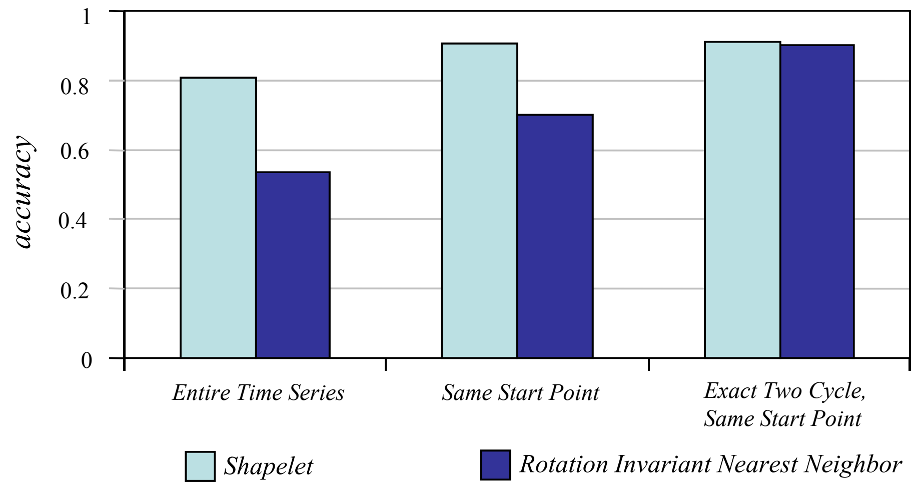


Figure 4.26: The accuracy comparison between the shapelet classifier and the rotation invariant nearest neighbor under three segmentation schemes. The shapelet classification method outperforms the nearest neighbor in all three segmentation schemes. However, careful segmentation reduces the accuracy difference between the two methods

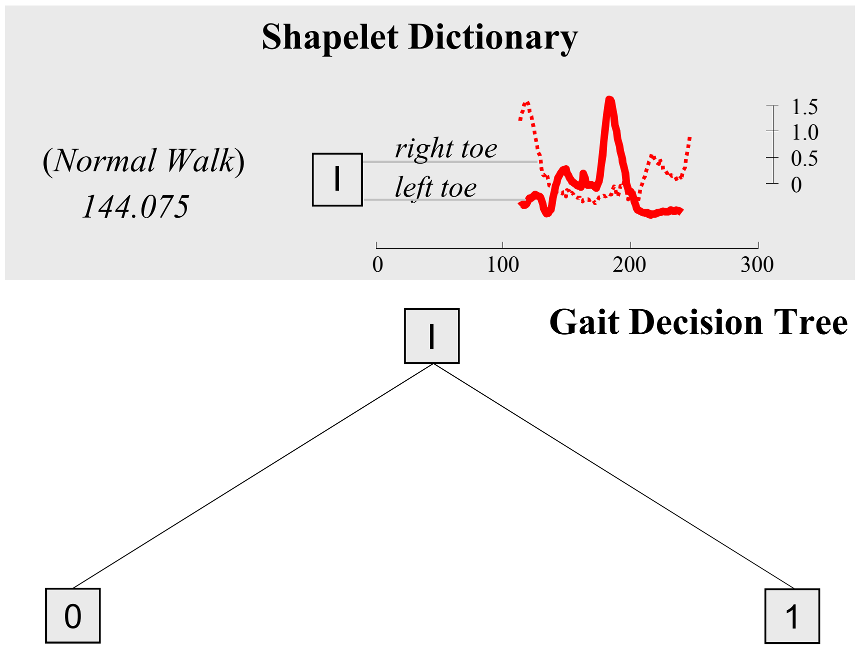


Figure 4.27: *Top*) The dictionary of shapelets, with the thresholds d_{th}. The x-axis shows the position of the shapelet in the original time series. *Bottom*) The decision tree for the 2-class gait analysis problem

4.5.10 The Effect of Noise on Training and Testing a Shapelet Classifier

In previous case studies, we have considered domains that have noise. For example, as we explicitly show in Section 4.5.3, many of the projectile points are broken, and in Section 4.5.4 many of the heraldic shields are torn. In an image processing context, this would be considered “occlusion noise”. In Section 4.5.2 the Lightning EMP Classification dataset is probably one of the nosiest time series classification problems, shown in Figure 4.28.*Left*. In Section 4.5.8, the Mallat data in [38] had Gaussian noise added to each of the training data to generate further samples. In Figure 4.28.*Right*, 17 random samples from the original class (cf. Figure 4.23) are shown. The Wheat Spectrography problem, which is known to have class label noise (that is to say, some training class labels are wrong), is presented in Section 4.5.6.

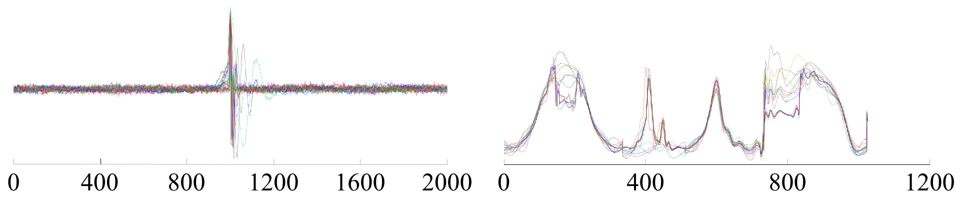


Figure 4.28: Data samples from Lightning EMP dataset (*Left*) and Mallat dataset (*Right*)

In this section however, we explicitly study how noise affects the accuracy of the shapelet classifier. There are two areas where we may encounter noise; one is in the training data and the other is in the testing data. These two types of noise data are evaluated separately in our experiments. That is, when we evaluate how noise influences the training dataset, we only add noise to the training data and keep the testing data unchanged and vice versa. Over several experiments, we add noise to various percentages of a dataset. For each time series object selected, we add a scaled Gaussian random noise $N(0, 1) \times |\max(d) - \min(d)|$ to a single random data point in the time series. The value d is the range of the time series. We evaluate the changes of accuracy on the Gun/noGun dataset in our case studies, since it is perhaps the most studied problem in the literature [22, 82].

We add noise ranging from zero to one-hundred percent of the data, using increments of ten percent. Both tests are averaged over 100 runs, where both the time series to be corrupted and the location of the noisy data point are chosen randomly.

The results are reported in Figure 4.29, where the solid line is the average accuracy and the dashed lines bounding it from above and below are its one standard deviation intervals.

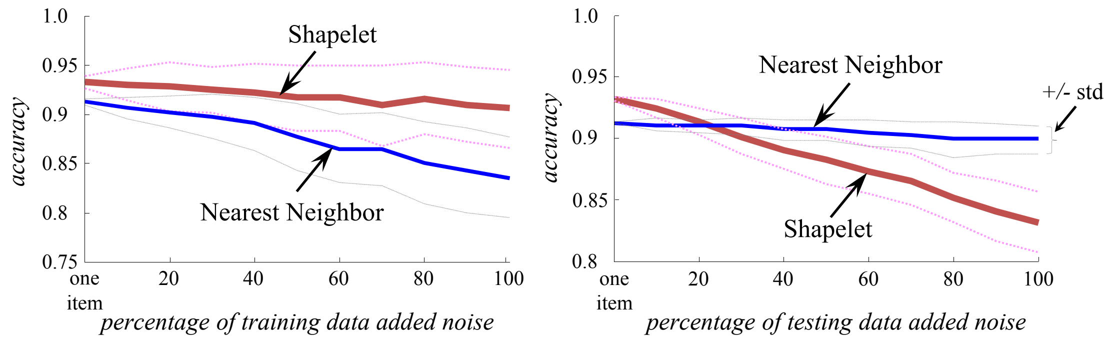


Figure 4.29: The change of classification accuracy on Gun/noGun dataset as we add noise to the data. *Left*) Noise is added only to the training dataset. *Right*) Noise is added only to the testing dataset. The lighter lines indicate plus/minus one standard deviation

We can see that when the training data is corrupted, the shapelet classifier continues to outperform the nearest neighbor classifier. Moreover, the margin of difference increases. However, if the noise occurs in the testing data, the shapelet classifier degrades more quickly than the nearest neighbor classifier. That is because noise could be added randomly to any data point of the time series data. If the noise happens to be added to the original best matching location of the shapelet region, it will greatly affect the accuracy of the shapelet classifier, while for the nearest neighbor classifier, the effect of the noise is mitigated by averaging over more data points (the entire time series). Let us consider an extreme case in the shapelet classifier where each data item of the testing dataset has noise. From the Gun/noGun case study we know that the length of the shapelet is 45 and the total length of the time series 150, so there is a 45/150 probability that noise will appear on the best matching location region. Supposing that half of the induced noise will mislead the sample into the opposite class, the accuracy is

$$1\text{-}(45/150)/2\text{=}0.85,$$

which is consistent with the results of our experiment.

In general, these results are a mixed bag for us. The positive news is that we can construct shapelets even in the presence of significant noise. However, if the data to be classified is very noisy, we may do worse than other methods. One possible way to mitigate this problem is to find multiple shapelets that are indicative of a class and build a disjunctive classifier. For example, an item belongs to Class A if it has a subsequence that is within 5.3 of pattern S1 **OR** within 2.3 of pattern S2. We leave such considerations for future work.

4.6 Conclusions and Future Work

We have introduced a new primitive for time series and shape mining, the *time series shapelet*. We have shown with extensive experiments that we can find the shapelets efficiently, and that this can provide accurate, interpretable, and much faster classification decisions in a wide variety of domains. Ongoing and future work includes extensions to the multivariate case and detailed case studies in the domains of anthropology and motion capture (MOCAP) analyses. In addition, although we used decision trees because of their ubiquity, it is possible that shapelets could be used in conjunction with Associative Classification [63] and other rule-based techniques. Our work does have several limitations that we will also attempt to address in future work. The most obvious weakness is the relatively slow training time. One possible way to mitigate this limitation is by sampling the training set and only using a small subset to build the shapelets. The results in Figure 4.12 suggest that this may be a tenable solution.

Chapter 5

Conclusions

We have seen that data is gathered and organized by a wide variety of methods at an ever-increasing speed thanks to the proliferation of distributed systems, the advances in storage techniques, and the development of data representations. Time series representations of data are prevalent and practical in various domains because of their simplicity and expressiveness. However, high-dimensional and large-scale time series make data more expressive yet potentially cause existing data mining algorithms to become intractable for real-time situations. Low-cost local distributed systems, such as the insect-trap systems discussed in Section 3.5.1, allow the collection of data at greater sampling rates and higher fidelity. Nevertheless, solutions of data mining tasks on these distributed systems must consider the hosts' low-power and low-memory properties more carefully. Common approaches to these resource-limited problems are approximation methods such as dimension reduction and sampling. In this work, we have demonstrated more flexible solutions to these situations where we should trade off the quality of the answer against the computational cost.

We have discussed the anytime framework for the approximate similarity join algorithm in Chapter 2, where an approximate join is presented during the process and the join result improves when more computational time is given. Several case studies have been conducted in a wide variety of domains and have demonstrated that approximate answers may be useful enough for join problems. Because the computational time required for exact searches degrades performance, instead of arguing about "how approximate" is sufficient for a certain problem, our method provides the best-so-far answer under any given amount of computational time.

We have exhibited the anyspace framework for the indexing algorithm in Chapter 3. To mitigate the drawback of the quadratic space complexity of an efficient indexing algorithm, we place in memory *only* the most significant part of the index to optimize the indexing efficiency of the limited memory. Furthermore, taking advantage of the anyspace framework, we provide an auto-cannibalistic algorithm which dynamically deallocates the space to the index on the fly, maximizing the overall computational performance.

We can extend the anytime/anyspace framework to multiple data mining tasks to produce a large library of anytime/anyspace algorithms. The anytime/anyspace framework is a general way to utilize limiting computational resources as much as possible in order to achieve optimal performance.

Lastly, we have argued that in some cases, *less* data achieves better performance in classification. We built a classification algorithm based on local features that are maximally representative of a class. The classifier saves both space and computational time and can achieve better accuracy. Furthermore, the algorithm provides interpretable results, and thus may help domain experts better understand the intrinsic patterns in the original data. Possible future work includes the following:

- The current algorithm discussed in Chapter 4 extracts the most distinguishing features in an exact but inefficient way, on the assumption that the data set to be explored is relatively small, and that there is a very limited amount of such features. However, this is not always the case. For example, if a classification problem contains more than fifty classes or the time series in each class are very high dimensional and contain more than one of such features, a batch algorithm to extract all of the features is more desirable than extracting these features one by one.

- Distinguishing local features are not merely used in the classification, but also can be adapted to other data mining tasks such as clustering and early prediction. For instance, previous work on feature-based time series clustering usually relies on application-dependant features which have limited relevance to specific applications [48]. Since our feature extraction method is generic and works well in unexplored domains, it is flexible enough to be applied to multiple applications.

One final contribution of my PhD study is that we make several image datasets, including those containing arrowheads, shields and butterflies, publicly available.

Note, we would also like to thank all the donors of data sets, In particular: Dr. Agenor Mafra-Neto for his entomological assistance, Dr. Pavlos Protopapas for his help with the star light curve data set, the staff of ISCA Technologies for their assistance with the insect project, Dr. Manuela Veloso and Douglas Vail for donating the robotic data, Ralf Hartemink for the help with the heraldic shields, and Dr. Sang-Hee Lee and Taryn Rampley for their help with the projectile point data set.

Bibliography

[1] Maristella Agosti, Nicola Ferro, and Nicola Orio. Annotations as a tool for disclosing hidden relationships between illuminated manuscripts. In *AI*IA '07: Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence on AI*IA 2007*, pages 662–673, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. Efficient similarity search in sequence databases. In *FODO '93: Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, pages 69–84, London, UK, 1993. Springer-Verlag.

[3] Anon. Founders' and benefectors' book of tewkesbury abbey, in latin england, 1525.

[4] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. *AAAI-94 workshop on knowledge discovery in databases*, pages 229–248, 1994.

[5] Christian Böhm, Bernhard Braunmüller, Markus Breunig, and Hans-Peter Kriegel. High performance clustering based on the similarity join. In *CIKM '00: Proceedings of the ninth international conference on Information and knowledge management*, pages 298–305, New York, NY, USA, 2000. ACM.

[6] Christian Böhm and Florian Krebs. High performance data mining using the nearest neighbor join. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining*, page 43, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Leo Breiman, Jerome Friedman, R. A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Chapman and Hall, January 1984.

[8] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973.

[9] Paolo Capitani and Paolo Ciaccia. Efficiently and accurately comparing real-valued data streams. In *In 13th Italian National Conference on Advanced Data Base Systems (SEBD*, 2005.

[10] SJ Castle, SE Naranjo, JL Bi, FJ Byrne, and Toscano NC. Toscano. phenology and demography of homalodisca coagulata (hemiptera: Cicadellidae) in southern california

citrus and implications for management. *Bulletin of Entomological Research (2005)*, 95(6):621–634, 2005.

[11] Kevin Chen-Chuan Chang, Bin He, and Zhen Zhang. Toward large scale integration: Building a metaquerier over databases on the web, 2004.

[12] Anton Chechetka and Katia Sycara. An any-space algorithm for distributed constraint optimization. In *Proceedings of AAAI Spring Symposium on Distributed Plan and Schedule Management*, 2006.

[13] Yun Chen and Jignesh M. Patel. Efficient evaluation of all-nearest-neighbor queries. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering*, pages 1056–1065, 2007.

[14] Bill Chiu, Eamonn J. Keogh, and Stefano Lonardi. Probabilistic discovery of time series motifs. In *KDD'03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 493–498, New York, NY, USA, 2003. ACM.

[15] Kenneth L. Clarkson. Nearest-neighbor searching and metric space dimensions. In Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk, editors, *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pages 15–59. MIT Press, 2006.

[16] Cmu graphics lab motion capture database. `http://mocap.cs.cmu.edu/`.

[17] Diane J. Cook and Lawrence B. Holder. *Mining Graph Data*. Wiley-Interscience, 2006.

[18] Adnan Darwiche. Any-space probabilistic inference. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 133–142, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[19] Gautam Das, King-ip Lin, Heikki Mannila, Gopal Renganathan, and Padhraic Smyth. Rule discovery from time series. In *Proceedings of the 4th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD98)*, pages 16–22. AAAI Press, 1998.

[20] Hui Ding, Goce Trajcevski, and Peter Scheuermann. Efficient similarity join of large sets of moving object trajectories. In *Proceedings of 15th International Symposium on Temporal Representation and Reasoning*, pages 79–87, 2008.

[21] Hui Ding, Goce Trajcevski, and Peter Scheuermann. Efficient similarity join of large sets of moving object trajectories. In *Proceedings of 15th International Symposium on Temporal Representation and Reasoning*, pages 79–87, 2008.

[22] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. Querying and mining of time series data: Experimental comparison of representations and distance measures. *Proceedings VLDB Endow.*, 1(2):1542–1552, 2008.

[23] D.O. Durosaro and R.A. Shehu. *A Workshop Manual on: Data Collection, Collection and analysis in schools*, 2007.

[24] Herbert A. Edelstein. *Introduction to Data Mining and Knowledge Discovery*. Two Crows Corporation, 2 edition, 1998.

[25] Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th International Conference on Machine Learning (ICML03)*, 2003.

[26] Christos Faloutsos and King-Ip Lin. Fastmap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 163–174, New York, NY, USA, 1995. ACM.

[27] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. *SIGMOD Rec.*, 23(2):419–429, 1994.

[28] Marchionini Gary. Exploratory search: from finding to understanding. *Communications of the ACM*, 49(4):41–46, 2006.

[29] Pierre Geurts. Pattern extraction for time series classification. In *PKDD '01: Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 115–127, London, UK, 2001. Springer-Verlag.

[30] D. Ghosh and A.P. Shivaprasad. Fast codeword search algorithm for real-time codebook generation inadaptive vq. In *Vision, Image and Signal Processing, IEE Proceedings*, volume 144, pages 278–284, 1997.

[31] Jens Gramm, Jiong Guo, and Rolf Niedermeier. On exact and approximation algorithms for distinguishing substring selection. 2751:963–971, 2003.

[32] Joshua Grass and Shlomo Zilberstein. Anytime algorithm development tools. *SIGART Bull.*, 7(2):20–27, 1996.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, 2006.

[34] Malte Herwig. Google's total library: Putting the world's books on the web. `http://www.spiegel.de/international/business/0,1518,473529,00.html`, 2007.

[35] Michael E. Hodgson. Reducing the computational requirements of the minimum-distance classifier. *Remote Sensing of Environment*, 25(1):117–128, 1988.

[36] Christopher Jeffery. Synthetic lightning emp data, 2005.

[37] Heather D. Jennings, Plamen Ch. Ivanov, Allan de M. Martins, P. C. da Silva, and G. M. Viswanathan. Variance fluctuations in nonstationary time series: a comparative study of music genres. *Physica A: Statistical and Theoretical Physics*, 336(3-4):585–594, 2004.

[38] Myong K. Jeong, Jye-Chyi Lu, Xiaoming Huo, Brani Vidakovic, and Di Chen. Wavelet-based data reduction techniques for process fault detection. *Technometrics*, 48(1), 2004.

[39] Adam N. Joinson. Looking at, looking up or keeping up with people?: Motives and use of facebook. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1027–1036, New York, NY, USA, 2008. ACM.

[40] Mohammed Waleed Kadous. Learning comprehensible descriptions of multivariate time series. In *ICML '99: Proceedings of the Sixteenth International Conference on Machine Learning*, pages 454–463, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[41] E. K. Kemsley, S. Ruault, and R. H. Wilson. Discrimination between coffea arabica and coffea canephora variant robusta beans using infrared spectroscopy. *Food Chemistry*, 54(3):321–326, 1995.

[42] Eamonn J. Keogh and Shruti Kasetty. On the need for time series data mining benchmarks: a survey and empirical demonstration. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 102–111, New York, NY, USA, 2002. ACM.

[43] Eamonn J. Keogh and Chotirat Ann Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge Information System*, 7(3):358–386, 2005.

[44] Eamonn J. Keogh, Li Wei, Xiaopeng Xi, Sang-Hee Lee, and Michail Vlachos. Lb_keogh supports exact indexing of shapes under rotation invariance with arbitrary representations and distance measures. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 882–893. VLDB Endowment, 2006.

[45] Sang-Wook Kim, Sanghyun Park, and Wesley W. Chu. An index-based approach for similarity search supporting time warping in large sequence databases. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, page 607, Washington, DC, USA, 2001. IEEE Computer Society.

[46] Sven Koenig and Yury Smirnov. Graph learning with a nearest neighbor approach. In *COLT '96: Proceedings of the ninth annual conference on Computational learning theory*, pages 19–28, New York, NY, USA, 1996. ACM.

[47] Scott T. Leutenegger and Mario A. López. The effect of buffering on the performance of r-trees. *IEEE Trans. on Knowl. and Data Eng.*, 12(1):33–44, 2000.

[48] T. Warren Liao. Clustering of time series data–a survey. *Pattern Recognition*, 38(11):1857 – 1874, 2005.

[49] Jessica Lin, Eamonn J. Keogh, Li Wei, and Stefano Lonardi. Experiencing sax: a novel symbolic representation of time series. *Data Min. Knowl. Discov.*, 15(2):107–144, 2007.

[50] Ying Liu, Dengsheng Zhang, Guojun Lu, and Wei-Ying Ma. A survey of content-based image retrieval with high-level semantics. *Pattern Recognition*, 40(1):262–282, 2007.

[51] Kerriann H. Malatesta, Sara J. Beck, Gamze Menali, and Elizabeth O. Waagen. The aavso data validation project. *The Journal of the American Association of Variable Star Observers*, 34(2):238–250, 2005.

[52] Aleix M. Martínez and Avinash C. Kak. Pca versus lda. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(2):228–233, 2001.

[53] James Augustus Montagu. *A Guide to the Study of Heraldry*. London : W. Pickering, 1984.

[54] T. Mouffet. *The Theater of Insects*, volume 2. Da Capo Press, New York, NY, USA, 1958.

[55] M.T. Orchard. A fast nearest-neighbor search algorithm. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 2297 –2300 vol.4, 14-17 1991.

[56] Rodrigo Paredes and Nora Reyes. Solving similarity joins and range queries in metric spaces with the list of twin clusters. *Journal of Discrete Algorithms*, 7(1):18–35, 2009.

[57] Mihael H. Polymeropoulos, Christian Lavedan, Elisabeth Leroy, Susan E. Ide, Anindya Dehejia, Amalia Dutra, Brian Pike, Holly Root, Jeffrey Rubenstein, Rebecca Boyer, Edward S. Stenroos, Settara Chandrasekharappa, Aglaia Athanassiadou, Theodore Papapetropoulos, William G. Johnson, Alice M. Lazzarini, Roger C. Duvoisin, Giuseppe Di Iorio, Lawrence I. Golbe, and Robert L. Nussbaum. Mutation in the alpha-Synuclein Gene Identified in Families with Parkinson's Disease. *Science*, 276(5321):2045–2047, 1997.

[58] Pavlos Protopapas, J. M. Giammarco, L. Faccioli, M. F. Struble, Rahul Dave, and Charles Alcock. Finding outlier light-curves in catalogs of periodic variable stars. *Monthly Notices of the Royal Astronomical Society*, 369:677–696, 2006.

[59] T. Subba Rao, M.B. Priestly, and O. Lessi. *Applications of Time Series Analysis in Astronomy and Meteorology*. Chapman & Hall, 1 edition, 1997.

[60] Chotirat A. Ratanamahatana and Eamonn J. Keogh. Three myths about dynamic time warping. In *SDM05: Proceeding of SIAM International Conference on Data Mining*, pages 506–510, 2005.

[61] Juan J. Rodríguez and Carlos J. Alonso. Interval and dynamic time warping-based decision trees. In *SAC'04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 548–552, New York, NY, USA, 2004. ACM.

[62] Yasushi Sakurai, Masatoshi Yoshikawa, and Christos Faloutsos. Ftw: Fast similarity search under the time warping distance. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 326–337, New York, NY, USA, 2005. ACM.

[63] Steven L. Salzberg. On comparing classifiers: Pitfalls to avoid and a recommended approach. *Data Mining and Knowledge Discovery*, 1(3):317–328, 1997.

[64] Albertus Seba. *Locupletissimi Rerum Naturalium Thesauri Accurata Descriptio Naaukeurige Beschryving van Het Schatryke Kabinet der Voornaamste Seldzaamheden der Natuur(Accurate Description of the Very Rich Thesaurus of the Principal and Rarest Natural Objects).* 1734.

[65] Marvin Shapiro. The choice of reference points in best-match file searching. *Commun. ACM*, 20(5):339–343, 1977.

[66] Yanxin Shi, Tom Mitchell, and Ziv Bar-Joseph. Inferring Pairwise Regulatory Relationships from Multiple Time Series Datasets. *Bioinformatics*, 23(6):755–763, 2007.

[67] Mitsuomi Shimada and Kuniaki Uehara. Discovering of correlation from multi-stream of human motion. In *DS '00: Proceedings of the Third International Conference on Discovery Science*, pages 290–294, London, UK, 2000. Springer-Verlag.

[68] Paul Smart. *The Illustrated Encyclopedia of the Butterfly World.* Tiger Books International PLC, 1991.

[69] Padhraic Smyth and David Wolpert. Anytime exploratory data analysis for massive data sets. In *KDD97: Proceedings of the 3rd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 54–60, New York, NY, USA, 1997. ACM.

[70] Byung Cheol Song and Jong Beom Ra. A fast search algorithm for vector quantization using l2-norm pyramid of codewords. *Image Processing, IEEE Transactions on*, 11(1):10–15, jan 2002.

[71] Alexander S. Szalay, Jim Gray, and Jan@ vandenBerg. Petabyte scale data mining: Dream or reality? *CoRR*, cs.DB/0208013, 2002.

[72] Ruey S. Tsay. *Analysis of Financial Time Series.* Wiley-Interscience, 2001.

[73] Ken Ueno, Xiaopeng Xi, Eamonn J. Keogh, and Dah-Jye Lee. Anytime classification using the nearest neighbor algorithm with applications to stream mining. In *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, pages 623–632, Washington, DC, USA, 2006. IEEE Computer Society.

[74] Michail Vlachos, Jessica Lin, Eamonn J. Keogh, and Dimitrios Gunopulos. A wavelet-based anytime algorithm for k-means clustering of time series. In *Workshop on Clustering High Dimensionality Data and Its Applications, SDM'03: Proceeding 3rd SIAM International Conference on Data Mining*, 2003.

[75] Koschorreck W. and Werner W. Facsimile edition with commentary: Kommentar zum faksimile des codex manesse: Die grosse heidelberger liederhandschrift, 1981.

[76] Jason Tsong-Li Wang, Xiong Wang, King-Ip Lin, Dennis Shasha, Bruce A. Shapiro, and Kaizhong Zhang. Evaluating a class of distance-mapping algorithms for data mining and clustering. In *KDD '99: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 307–311, New York, NY, USA, 1999. ACM.

[77] Xiaoyue Wang, Lexiang Ye, Eamonn Keogh, and Christian Shelton. Annotating historical archives of images. In *JCDL '08: Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*, pages 341–350, New York, NY, USA, 2008. ACM.

[78] Li Wei and Eamonn J. Keogh. Semi-supervised time series classification. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 748–753, New York, NY, USA, 2006. ACM.

[79] Li Wei, Eamonn J. Keogh, and Xiaopeng Xi. Saxually explicit images: Finding unusual shapes. In *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, pages 711–720, Washington, DC, USA, 2006. IEEE Computer Society.

[80] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[81] R. H. Wilson and H. S. Tapp. Mid-infrared spectroscopy for food analysis: Recent new applications and relevant developments in sample presentation methods. *TrAC Trends in Analytical Chemistry*, 18(2):85–93, 1999.

[82] Xiaopeng Xi, Eamonn J. Keogh, Christian Shelton, Li Wei, and Chotirat Ann Ratanamahatana. Fast time series classification using numerosity reduction. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 1033–1040, New York, NY, USA, 2006. ACM.

[83] Xiaopeng Xi, Eamonn J. Keogh, Li Wei, and Agenor Mafra-Neto. Finding motifs in database of shapes. In *SDM07: Proceeding of SIAM International Conference on Data Mining*, pages 249–260, 2007.

[84] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jing Hu. Gorder: an efficient method for knn join processing. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 756–767. VLDB Endowment, 2004.

[85] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. Top-k set similarity joins. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 916–927, march 2009.

[86] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 131–140, New York, NY, USA, 2008. ACM.

[87] Yuu Yamada, Einoshin Suzuki, Hideto Yokoi, and Katsuhiko Takabayashi. Decision-tree induction from time-series data based on a standard example split test. In *ICML'03: Proceeding of the 20th International Conference on Machine Learning*, pages 840–847, 2003.

[88] Ying Yang, Geoff Webb, Kevin Korb, and Kai Ming Ting. Classifying under computational resource constraints: Anytime classification using probabilistic estimators. *Machine Learning*, 69(1):35–53, 2007.

[89] Dragomir Yankov, Eamonn J. Keogh, and Umaa Rebbapragada. Disk aware discord discovery: Finding unusual time series in terabyte sized datasets. *Knowl. Inf. Syst.*, 17(2):241–262, 2008.

[90] Lexiang Ye. The asymmetric approximate anytime join. `http://www.cs.ucr.edu/~lexiangy/AAAJ/dataset.html`, October 2008.

[91] Lexiang Ye. Auto-cannibalistic and anyspace indexing algorithms with applications to sensor data mining. `http://www.cs.ucr.edu/~lexiangy/anyspace/dataset.html`, October 2009.

[92] Lexiang Ye. The time series shapelet. `http://www.cs.ucr.edu/~lexiangy/shapelet.html`, February 2009.

[93] Lexiang Ye and Eamonn J. Keogh. Time series shapelets: a new primitive for data mining. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 947–956, New York, NY, USA, 2009. ACM.

[94] Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 201–208, Washington, DC, USA, 1998. IEEE Computer Society.

[95] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[96] K. Zatloukal, M. H. Johnson, and R. Ladner. Nearest neighbor search for data compression. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 59, Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, pages 69–86. American Mathematical Society, 2002.

[97] Jun Zhang, Nikos Mamoulis, Dimitris Papadias, and Yufei Tao. All-nearest-neighbors queries in spatial databases. In *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, page 297, Washington, DC, USA, 2004. IEEE Computer Society.

[98] Ying Zhang, Bernard J. Jansen, and Amanda Spink. Time series analysis of a web search engine transaction log. *Inf. Process. Manage.*, 45(2):230–245, 2009.

[99] Shlomo Zilberstein and Stuart Russell. Approximate reasoning using anytime algorithms. In *Imprecise and Approximate Computation*, pages 43–62, 1995.