UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Time Series Retrieval: Indexing and Mining Large Datasets

Permalink
https://escholarship.org/uc/item/7gk6t3b8

Author
Shieh, Jin-Wien

Publication Date
2010

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Time Series Retrieval: Indexing and Mining Large Datasets

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

Doctor of Philosophy

in

Computer Science

by

Jin-Wien Shieh

June 2010

Dissertation Committee:
Dr. Eamonn Keogh, Chairperson
Dr. Stefano Lonardi
Dr. Vassilis Tsotras

Copyright by
Jin-Wien Shieh
2010

The Dissertation of Jin-Wien Shieh is approved:

____

____

____

Committee Chairperson

University of California, Riverside

Acknowledgements

This work was funded by NSF Award 0803410, 0808770, and Career Award 0237918. We would also like to thank the many donors for providing the datasets: Anna Watson and Michael Mayo (Moth); Gregory Walker and Candice Stafford (Leafhopper); Antonio Torralba, Rob Fergus, and William Freeman (Tiny Images).

I would like to take this time to thank the many people who have supported me during this Ph.D process. First and foremost is my advisor Dr. Eamonn Keogh. It has been a pleasure working with him these past couple of years. Eamonn's guidance has been invaluable and his drive and passion for his work continues to impress.

I would also like to thank my committee members: Dr. Stefano Lonardi and Dr. Vassilis Tsotras for their time, support, and suggestions.

My friends and labmates. Together we have shared the highs and lows, the laughs and frustrations. Always good for throwing ideas around or to provide a much needed distraction. Our experiences are something I am extremely thankful of and will forever cherish.

Finally, I would like to thank my parents and my sister. They have always been there for me.

ABSTRACT OF THE DISSERTATION

Time Series Retrieval: Indexing and Mining Large Datasets

by

Jin-Wien Shieh

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

As advances in science and technology have continually increased the existence of, and capability for users to monitor, record, and examine data, data mining has become a common and necessary toolset in order to gain additional insight on this influx of data. In this dissertation, we study methods which are used for overcoming the characteristic challenges of scale in order to perform similarity search on large time series datasets.

We introduce a novel multi-resolution symbolic representation for time series called indexable Symbolic Aggregate approXimation (*i*SAX). The *i*SAX representation allows for the indexing of time series in order to facilitate similarity search. We further demonstrate its utility by performing experimental evaluation on a wide range of diverse datasets and show how exact and approximate search can be used in conjunction to expedite higher level data mining operators to solve real world problems. The size of the datasets we consider are larger

than any other in the current literature and notably, our results confirm the notion that even simple measures perform exceedingly well when the training set becomes very large.

Another aspect of our research considers using similarity search to perform classification under limited computation time and variable response rates. In such contexts, anytime algorithms, amenable to variable response times by exchanging quality of response as a function of time, have been found to be especially useful. We present a generalized framework which utilizes a scoring function that estimates the intermediate result quality of an object being classified. Our contribution extends existing anytime algorithms to concurrent queries by dynamically scheduling computational resources for each object. We show that the lack of such inter-object consideration would otherwise result in poor allocation of computation time and lead to reduced performance.

Contents

List of Tables **xi**

List of Figures **xii**

1 Introduction **1**

1.1 Data Mining Time Series 3

1.2 Indexing and Mining Large Datasets 9

1.3 Similarity Search in Time Constrained Environments 10

1.4 Contributions 12

2 *i*SAX: Indexing and Mining Terabyte Sized Time Series **14**

2.1 Introduction 14

2.2 Background and Related Work 16

2.2.1 Time Series Distance Measures 16

2.2.2 Time Series Representations 17

2.2.3 Review of Classic SAX 20

2.3 The *i*SAX Representation 26
2.3.1 Comparing Different Cardinality *i*SAX Words 27
2.4 *i*SAX Indexing 29
2.4.1 The Intuition Behind *i*SAX Indexing 29
2.4.2 *i*SAX Index Construction 31
2.4.3 Approximate Search 34
2.4.4 Exact Search 36
2.5 Experimental Evaluation 41
2.5.1 Tightness of Lower Bounds 42
2.5.2 Sensitivity to Parameters 44
2.5.3 Indexing Massive Time Series Datasets 49
2.5.4 Approximate Search Evaluation 52
2.5.5 Time Series Set Difference 55
2.5.6 Batch Nearest Neighbor Search 60
2.5.7 Mapping the Rhesus Monkey Chromosomes 65
2.6 Concluding Remarks 69

3 Towards Indexing and Mining One Billion Time Series 70
3.1 Introduction 70
3.2 Bulk Loading Massive Time Series Datasets 71
3.3 Updating Index Node Splitting Policy 72

3.4 Experimental Evaluation 74
3.4.1 Index Scalability 74
3.4.2 A Case Study in Entomology 75
3.4.3 Mining Massive DNA Sequences 78
3.4.4 Mining Massive Image Collections 81
3.5 Concluding Remarks 83

4 Anytime Nearest Neighbor Classification 84

4.1 Introduction 84
4.2 Background and Related Work 87
4.3 Anytime Nearest Neighbor Classification 90
4.4 Concurrent Object Evaluation 93
4.4.1 Scheduling Policies 96
4.4.2 Batch Evaluation 98
4.4.3 Streaming Evaluation 101
4.5 Scoring Function 102
4.6 Experimental Results 103
4.6.1 Classification of Streaming Data 104
4.6.2 Effects of Constrained Memory on Classification Accuracy 106
4.6.3 Streams with Non-Uniform Arrival 108
4.6.4 A Case Study in Commercial Entomology 111

4.7 Concluding Remarks 113

5 Conclusion **114**

Bibliography **117**

List of Tables

2.1 A Hierarchy of Time Series Representations 18

2.2 SAX Breakpoints 22

2.3 It is possible to obtain a reduced (by half) cardinality SAX word simply by ignoring trailing bits 23

2.4 A SAX $dist$ lookup table for α = 4 24

4.1 List of Notation 91

4.2 Datasets used for experimental evaluation 104

List of Figures

1.1 Data represented by time series across various domains. *top*) Comparison of financial stock prices for two automotive companies (as shown by Yahoo! Finance). *bottom left*) Google Trends depicting search popularity over time for a specified query. *bottom right*) An image of a horned lizard skull mapped to time series by measuring the distance from the image outline to the center of the shape 4

1.2 Example of DTW allowing elastic alignment to match morphological similarities between two horned lizard species, *top*) the flat-tailed horned lizard (*Phrynosoma mcallii*) and *bottom*) the texas horned lizard (*Phrynosoma cornutum*) 7

2.1 The error rate of DTW and ED on increasingly large instantiations of the CBF and Two-Pat problems. For even moderately large datasets, there is no difference in accuracy 17

2.2 *left*) A time series T, of length 16. *right*) A PAA approximation of T, with 4 segments . . . 20
2.3 A time series T converted into SAX words of cardinality 4 $\{11,11,01,00\}$ (*left*), and cardinality 2 $\{1,1,0,0\}$ (*right*) . . . 21
2.4 The tightness of lower bounds for increasing SAX cardinalities, compared to a PAA/DWT benchmark . . . 25
2.5 An illustration of an *i*SAX index . . . 32
2.6 MINDIST between PAA and *i*SAX representations. The lower bounding distance is computed from the hatched lines . . . 38
2.7 Improving query performance by constructing time series wedges among a set of time series with a common *i*SAX word . . . 40
2.8 The tightness of lower bounds for various time series representations on the Koski ECG dataset. Similar graphs for thirty additional datasets can be found at [49] . . . 43
2.9 The experiment in the previous figure redone with the *i*SAX word length equal to the dimensionality of the real valued applications (just DCT is shown to allow a "zoom in") . . . 45
2.10 Approximate search rankings for increasing threshold values . . . 46
2.11 Index files created across varying threshold values . . . 47
2.12 Approximate search rankings for increasing word lengths . . . 47
2.13 Index files created across varying word lengths . . . 48

2.14 The percentage of cutoffs for various rankings, for increasingly large databases with approximate search 49
2.15 Estimated wall clock time for exact search averaged over 100 queries 50
2.16 Average disk I/O for exact search averaged over 100 queries 51
2.17 Approximate search result on insect dataset 54
2.18 Sorted distance ratios of 100 random walk queries 55
2.19 Plot showing the random walk query, approximate result, and true nearest neighbor. Together they correspond to the lower median of distance ratios computed 56
2.20 The Time Series Set Difference discovered between ECGs recorded during a waking cycle and the previous 7.2 hours 57
2.21 Corresponding sections of human and chimpanzee DNA 62
2.22 The distribution of the Euclidean distances from subsequences in Rhesus Monkey chromosome 19 to their approximate nearest neighbor in Human chromosome 2. The distribution is normalized such that the area under the curve is one 67
2.23 The distribution of the Euclidean distances from subsequences in Rhesus Monkey chromosomes 19 and 12, to their approximate nearest neighbor in Human chromosome 2 67

2.24 A dot plot showing the alignment of Human chromosome 2 with both chromosome 12 and 13 of the Rhesus Monkey. Each dot represents a location where a subsequence in the monkey (row) is less than 1,250 from a subsequence in a human (column) 68

3.1 A schematic diagram showing an EPG apparatus used to record insect behavior 77

3.2 An EPG insect behavior derived from a subset of Fig. 2 from [51]. An idealized version of the observed behavior created by us is shown with a bold blue line 77

3.3 Query time series and its approximate nearest neighbor 78

3.4 An example of DNA converted into time series 79

3.5 The cells represent potential mappings between the Macaque and Human Genomes. The darker the cell, the more often the nearest neighbor of a time series taken from a particular human chromosome had a nearest neighbor from a particular Macaque chromosome (the smallest chromosomes including the sex chromosomes are omitted) 80

3.6 *top left*) A detail of *The Son of Man* by René Magritte, which we used as a query to our index. *top center*) The best match returned. *top right*) The similarity of the two images in RGB histogram space. *bottom left*) A detail of *The Scream* by Edvard Munch, which we used as a query. *bottom center*) The best match returned. *bottom right*) The similarity seen in RGB space . . 82

4.1 Anytime algorithms are interruptible after initialization. This plot shows the increase in result quality with additional computation time 88

4.2 Net result quality across concurrent queries for various scheduling policies. *left*) Result quality over time for two queries Q_1 and Q_2 (note that Q_2 arrives shortly after Q_1). *right*) Net result quality (Q_1 and Q_2) for various scheduling policies at evaluation time (hatched line) 96

4.3 Classification accuracy of score scheduled anytime classifier on constant data streams with varying rates of arrival 107

4.4 Classification accuracy on data streams with exponentially distributed interarrival times 109

4.5 Classification accuracy on additional datasets with with exponentially distributed interarrival times 110

4.6 *left*) The adult Light Brown Apple Moth is harmless to agriculture, however it's larval form *right*) causes extensive damage to several commercially important crops 111

4.7 Insect classification with memory buffer constrained to four objects 112

List of Algorithms

1 *i*SAX Index Insertion (ts) 35

2 Exact Search (ts) 39

3 An outline to find the TSSD (A, B) 59

4 An algorithm for converting DNA to time series 61

5 Batch Nearest Neighbor(A, B) 64

6 New Node Splitting Policy( ) 73

7 Anytime Nearest Neighbor Classifier(q, D) 93

8 Initializing Object Classification (q, D) 100

9 Updating Object Classification (q, D) 100

10 Batch Score Scheduled Classifier $(Q, D, ScoreFcn)$ 100

11 Streaming Score Scheduled Classifier $(D, ScoreFcn, M)$ 102

Chapter 1

Introduction

Applications of data mining have changed our understanding of, and interaction with the world. For example, data mining has allowed retail businesses such as Wal-Mart to key in on customer behavior with market basket analysis and forecasting to improve sales [38]. In biological sciences, mining from gene expression data has been used to enhance understanding of cellular function [87]. Even in an everyday sense, we expand our productivity and knowledge with the utility of search engines providing information at our fingertips. These and other examples are a testament to the breath and wide-spread applicability and utility which is attributed to data mining.

While an exact definition of knowledge discovery and data mining (KDD) can be broad and nebulous, Frawley et al. provides a close standard: KDD is “the nontrivial extraction of implicit, previously unknown, and potentially useful information from data”[30]. Regardless of whether users are interested in classification [36], regression [22], clustering [14], motif

discovery [65], etc., all data mining tasks are characterized by large volumes of data, and a subsequent need to process that data efficiently and accurately [29]. Efficiency provides the means for making the processing of large data sizes tractable, by minimizing overall running time or algorithmic complexity (time and I/O); and accuracy measures algorithmic performance in adherence to an objective or model.

As advances in science and technology have continually increased the existence of and capability for users to monitor, record, and examine data, data mining has become a common and necessary toolset in order to gain additional insight on this influx of data [67]. In this dissertation, we study the methods which are used for overcoming the aforementioned characteristic challenges of scale and also demonstrate the utility of data mining to solve real world problems. For example, entomologists have acquired large collections of behavioral data from pest insects which threaten to cause millions of dollars in crop damage per year [4]. They require a tool which is capable of searching these massive datasets for characteristic patterns of feeding. In Chapter 2 and Chapter 3 we examine these problems in the context of large and mostly static data, typical of data warehouse collections of historical data. Solutions for environments with limited computation time and flexibility for variable response rates, such as those exhibited in data streams, are presented and studied in Chapter 4.

1.1 Data Mining Time Series

We preface discussion regarding the technical specifics of our contributions by first elucidating key assumptions and providing some background. To begin, we describe the type of data which is of interest to us.

Across different problems and their respective domains there exist many different representations of data. The selection and understanding of the characteristics pertaining to a chosen or encountered representation is a necessary condition towards achieving a data mining solution. Notably, a practitioner must ask: *Is the representation expressive enough? Is it easy to visualize and intuitive enough for users to interpret? Does it lend itself to computationally efficient methodology?* These are key questions which affect all aspects of the data mining process, from problem description to interpretation of results.

Our work in this dissertation primarily utilizes data represented in the form of time series. Time series are descriptive, easy to visually interpret, and are ubiquitous. Any context with values which are measured over time constitutes a time series, and as such, it is the native representation for stocks, medicine, and monitoring data. Even for data types as diverse as images, shapes [90], XML, and DNA [77] there are functions which provide a mapping to a descriptive pseudo time series representation. Figure 1.1 provides some examples of time series across a range of contexts.

Let us first begin by formalizing the definition of time series adopted in the remainder of this work:

Figure 1.1: Data represented by time series across various domains. *top*) Comparison of financial stock prices for two automotive companies (as shown by Yahoo! Finance). *bottom left*) Google Trends depicting search popularity over time for a specified query. *bottom right*) An image of a horned lizard skull mapped to time series by measuring the distance from the image outline to the center of the shape

Definition 1.1. A time series $T = t_1, t_2, \cdots, t_n$ is an ordered set of n real-valued observations. Each observation is measured at equal intervals in time.

It is important to note that it is often the case that real world observations are not measured at equal time intervals and can often exhibit noisy, dropped, or empty values. Such characteristics are often undesired as most algorithms have implicit assumptions regarding standardized inputs and will perform poorly on such data, if at all. To mitigate such effects, a data cleaning step is typically performed prior to data analysis (c.f. [29] for a more in-depth outline of the general knowledge discovery process). Data cleaning is an important pre-processing step to standardize data by removing duplicates, interpolating missing values, etc. The assumptions taken at this step can impact algorithm output as well as the interpretability and usability of results.

For time series, a common task is to perform statistical analysis or construct models (e.g. autoregressive, integrated, moving average, etc.) which characterize the series of interest [22]. Once constructed, models allow a user to identify trends, forecast future points, and perform applications such as burst or anomaly detection [21].

Another salient utility for time series is that of similarity search. For example, given a time series of interest, if a user can identify similar patterns in a dataset, this can allow them to glean important historical insight, perform query by content, or use similarity search as a subroutine in other KDD applications [89] [25]. Given this wide range of usage, significant research has been spent on expediting similarity search over large time series datasets [2][3][20][63][58][91][28].

To perform similarity search, we must first define distance measures to quantify the similarity between two time series. A discriminative distance measure should provide low values for similar time series and large values conversely. A widely used set of distance measures are the family of L_p-norms. Given time series T and S of length n, the L_p-norm is defined as:

$$L_p(T, S) = \left(\sum_{i=1}^{n} |t_i - s_i|^p \right)^{1/p}$$

Of these, $p = \{1, 2, \infty\}$ are commonly used. In our work, we focus on $p = 2$ (also known as Euclidean distance) which has been shown to be a simple, yet competitive and efficient distance measure for time series [48][26]. Euclidean distance naturally lends itself to optimizations such as early abandoning [48] and its *metric* properties can be exploited by higher level algorithms to prune search space (i.e. via triangle inequality) [66][65][75].

Due to the real world nature of data, time series often contain natural perturbations in the form of shifts in time, scale, or noise. For example, consider time series derived from tracking an individual's hand motion. Even if the same person is repeatedly performing exactly the same motion, comparison of time series between two repetitions is guaranteed to exhibit some slight shifts and changes. While each repetition is semantically identical, Euclidean distance may compute a overall value which is quite large. This is because Euclidean distance

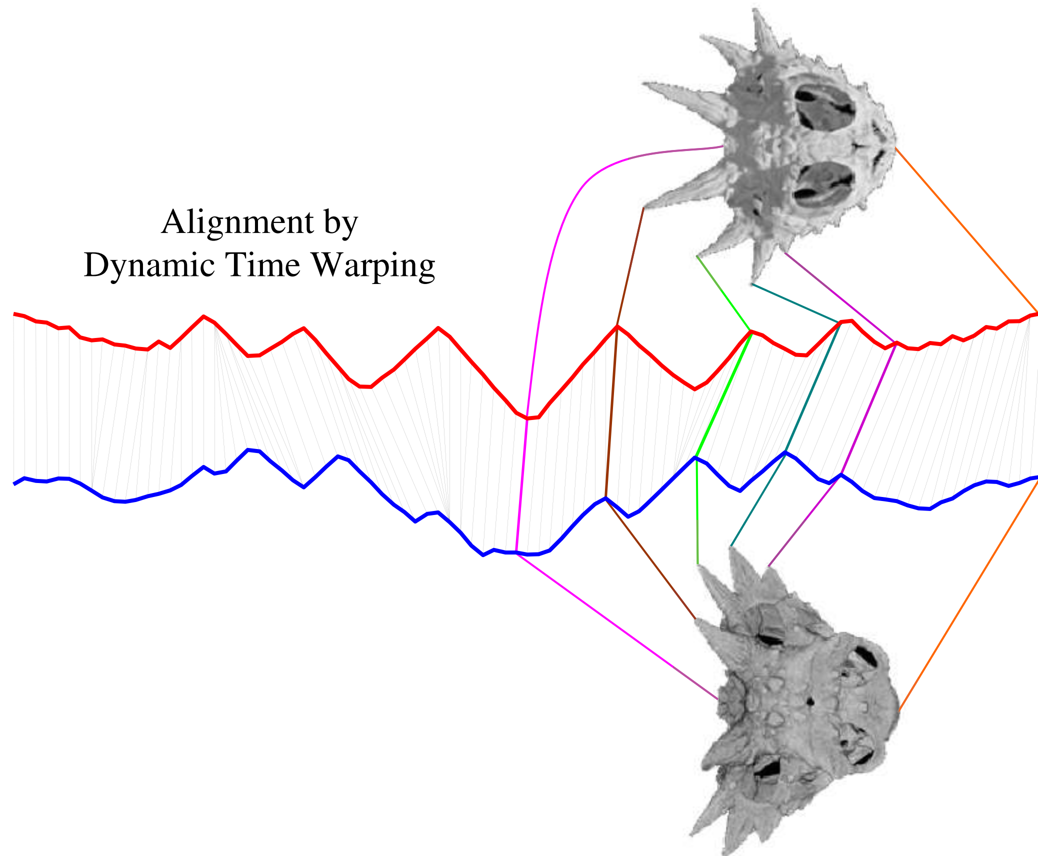

Figure 1.2: Example of DTW allowing elastic alignment to match morphological similarities between two horned lizard species, *top*) the flat-tailed horned lizard (*Phrynosoma mcallii*) and *bottom*) the texas horned lizard (*Phrynosoma cornutum*)

matches point to point with fixed alignment between two time series, making it brittle to perturbations and incapable of matching shifted points.

Normalizing a time series can mitigate this effect, particularly shifts in value and scale, and is sometimes a necessary condition for time series similarity search (especially when users are primarily interested in the "shape" of a pattern [48]). A common technique is to Z-normalize a time series to zero mean and standard deviation of one:

Let μ_T be the mean and σ_T the standard deviation of the original series, T. The Z-normalization of T is $\hat{T} = \hat{t_1}, \hat{t_2}, \cdots, \hat{t_n}$ where $\hat{t_i} = (t_i - \mu_T)/\sigma_T$.

Another means for computing similarity in the presence of such perturbation is to utilize the class of distance measures which are *elastic* and allows for warping between the alignments of two time series. Recall that Euclidean distance has a fixed point to point alignment which can be the cause of poor performance. One elastic measure which has found success in the time series domain is dynamic time warping (DTW) [12][91][44][69]. DTW preserves temporal ordering within each time series but outputs a flexible alignment which minimizes the distance (see Figure 1.2 for an example of a warped alignment). The primary disadvantages of DTW is that metric properties do not hold and that computing DTW via a dynamic programming approach results in quadratic time and space complexity. The latter point may be reduced with minimal loss in quality through various techniques which limit the allowed warping window [72][44]. The recurrence relation for computing DTW is shown below:

$$DTW(T_n, S_m) = (|t_n - s_m|^2)^{1/2} + min \begin{cases} DTW(T_{n-1}, S_m) \\ DTW(T_n, S_{m-1}) \\ DTW(T_{n-1}, S_{m-1}) \end{cases}$$

Now that we have presented our time series definition, its utility, and the basics behind similarity search, we are ready to examine techniques which allow retrieval from large time series datasets as well as general data mining applications.

1.2 Indexing and Mining Large Datasets

Given that time series are ubiquitous, a natural consequence is an abundance of large datasets. For example, astronomical datasets frequently contain millions of star light curves [60]. The difficulty in performing similarity search in such contexts is that the size of data considered drastically hinders the performance of many algorithms. For example, a methodology which assumes data can be memory resident may freely utilize random access, consequently resulting in unacceptable I/O costs. Another cause for diminished performance is the high dimensionality which typically characterizes time series (e.g. sequences with hundreds to thousands of points). It is widely understood that most data structures such as R-trees [34] do not scale well past a few dimensions due to the “curse of dimensionality”[2]. The solution is to utilize the framework presented in [2] by mapping the time series to a reduced space using a time series representation, computing candidates more easily in this reduced space, and then verifying the result in the original space. An overview of some representations for time series is presented in Chapter 2.

In Chapter 2 and Chapter 3 we present a novel multi-resolution symbolic representation called indexable Symbolic Aggregate approXimation (*i*SAX). The *i*SAX representation, being symbolic has the advantage over other real valued representations by being able to utilize data structures and algorithms which are defined for discrete data types, such as suffix trees, Markov models, etc. Our approach is based upon an extension of the well known SAX representation [54][55] to allow indexing of time series without leaf level overlaps unlike R-trees

and other spatial access methods. We demonstrate the ease and simplicity of our approach by creating an index which achieves fast and scalable results even when simply using the standard Windows file system to manage disk access.

We show the scalability and efficacy of our representation by conducting experiments on datasets which are orders of magnitude larger than anything considered in previous literature. Our approach allows for exact nearest neighbor search of large time series datasets as well as fast approximate nearest neighbor results. We further demonstrate how both types of searches can be exploited in conjunction to facilitate data mining algorithms which can easily and exactly mine datasets with of millions of time series.

Chapter 2 also introduces a new data mining operator, *Time Series Set Difference*, which is shown to be an useful tool for contrasting two sets of time series. We demonstrate its utility by identifying suspected sleep disorder across sets of ECG subsequences.

1.3 Similarity Search in Time Constrained Environments

The techniques presented in Chapter 2 and Chapter 3 provide solutions for expediting similarity search in large, mostly static datasets. The results of similarity search can then be used in data mining applications such as nearest neighbor classification. In Chapter 4 we continue to explore data mining techniques for classification, though not in the context of static datasets. Instead, we consider the scenarios where available computation time may be highly variable, such as in data streams. Whereas algorithms in Chapter 2 and Chapter 3 can assume

sufficient computation time to complete execution and work towards reducing that end to end time, performing similar tasks in a data stream is often more difficult than the batch situation because an algorithm must operate in a time sensitive and computationally constrained environment. That is, a data stream may require objects to be classified at a rate that can range from milliseconds to minutes [83].Classic algorithms typically lack the mechanism for providing an intermediate result prior to completion, and contract-based algorithms require the available time duration prior to execution [92]. In these contexts, the anytime algorithms discussed in Chapter 4 have been found to be exceptionally useful, and have recently been the subject of extensive research efforts [27][40][50][56][52][83][88].

Anytime algorithms are a class of algorithms which are amenable to variable response times, by exchanging the quality of response as a function of time [32][92]. In the context of classification, quality is measured by the probability of correct classification and an anytime algorithm, after a short period of initialization, can always be interrupted to return some intermediate result. This flexibility in response time allows anytime algorithms to be used with great success in real world environments with variable constraints [27][41][74]

For anytime classification, one well established technique is the anytime nearest neighbor classification algorithm [83]. This algorithm retains the strong points of the nearest neighbor algorithm, its simplicity and generality , while greatly mitigating the problem associated with the linear time complexity at classification time, a function of its lazy behavior. Previous techniques for improving anytime classification have generally been concerned with optimizing the probability of correctly classifying individual objects. In Chapter 4, we show

that substantial improvement in overall classification accuracy performance can be achieved if the optimization is performed relative not to each individual object, but rather to a (possibly quite small) set of objects.

Our technique presented is a generalized framework which utilizes a scoring function that estimates the intermediate result quality of an object being processed. Here, the quality is an estimate that we have the correct class label for the object. Objects with a high initial quality are unlikely to significantly improve their quality, even with additional computation time. In contrast, objects with poor initial quality have much greater room for improvement, and are deserving of more resources. Using this intuition, our framework intelligently and dynamically schedules computational resources for each object. We demonstrate that the lack of such inter-object consideration would otherwise result in poor allocation of computation time and lead to reduced performance.

1.4 Contributions

The contributions of this dissertation are presented in Chapter 2, Chapter 3, and Chapter 4. More specifically, they are as follows:

- We present *i*SAX, a new reduced representation for time series. *i*SAX has the key properties of being multi-resolution, bit aware, quantized, and supporting variable granularity. We show that an index with *i*SAX is easily implemented and similarity search

can be conducted with exceptional performance even when simply using the native operating system file system to manage storage.

- We demonstrate the capability to index and perform time series experiments on datasets which are orders of magnitude larger than previously encountered in the time series literature.

- We introduce the *Time Series Set Difference* data mining operator and show that it returns useful results on datasets.

- We introduce an anytime framework for improving the overall accuracy of Anytime Nearest Neighbor Classification in data streams which exhibit queries with concurrent lifetimes. Our approach considers the set of concurrent queries and utilizes a scoring function to schedule resources and manage evictions according to an estimate on result quality. We show an improvement in overall classification accuracy by allocating computational resources to queries most likely to improve in classification accuracy with additional computation time.

Chapter 2

*i*SAX: Indexing and Mining Terabyte Sized Time Series

2.1 Introduction

The increasing level of interest in indexing and mining time series data has produced many algorithms and representations. However, with few exceptions, the size of datasets considered, indexed, and mined seems to have stalled at the megabyte level. At the same time, improvements in our ability to capture and store data have lead to the proliferation of terabyte-plus time series datasets. In this work, we show how a novel multi-resolution symbolic representation called *indexable Symbolic Aggregate approXimation* (*i*SAX) can be used to index datasets which are several orders of magnitude larger than anything else considered in current literature [77][78].

The *i*SAX approach allows for both fast exact search and ultra fast approximate search. Beyond mere similarity search, we show how to exploit the combination of both types of search as sub-routines in data mining algorithms, permitting the exact mining of truly massive datasets, with tens of millions of time series, occupying up to a terabyte of disk space.

Our approach is based on a modification of the SAX representation to allow extensible hashing [54][55]. That is, the number of bits used for evaluation of our representation can be dynamically changed, corresponding to a desired resolution. An increased number of bits can then be used to differentiate between non-identical entries. In essence, we show how we can modify SAX to be a multi-resolution representation, similar in spirit to wavelets [20]. It is this multi-resolution property that allows us to index time series with zero overlap at leaf nodes as in TS-tree [7], unlike R-trees [34], and other spatial access methods.

As we shall show, our indexing technique is fast and scalable due to intrinsic properties of the *i*SAX representation. Because of this, we do not require the use of specialized databases or file managers. Our results, conducted on massive datasets, are all achieved using a simple tree structure which uses the standard Windows XP NTFS file system for disk access. While it might have been possible to achieve faster times with a sophisticated DBMS, we feel that the simplicity of this approach is a great strength, and will allow easy adoption, replication, and extension of our work.

A further advantage of our representation is that, being symbolic, it allows the use of data structures and algorithms that are not well defined for real-valued data; including suffix trees, hashing, Markov models, etc [55]. Furthermore, given that *i*SAX is a superset of classic

SAX, the several dozen research groups that use SAX will be able to adopt *i*SAX to improve scalability [45]. The rest of this chapter is organized as follows. In Section 2.2 we review related work and background material. Section 2.3 introduces the *i*SAX representation, and Section 2.4 shows how it can be used for approximate and exact indexing. In Section 2.5 we perform a comprehensive set of experiments on both indexing and data mining problems. Finally, in Section 2.6 we offer conclusions and suggest directions for future work.

2.2 Background and Related Work

2.2.1 Time Series Distance Measures

It is increasingly understood that Dynamic Time Warping (DTW) is better than Euclidean Distance (ED) for most data mining tasks in most domains. It is therefore natural to ask why we are planning to consider Euclidean distance in this work. The well documented superiority of DTW over ED is due to the fact that in small datasets it might be necessary to warp a little to match the nearest neighbor. However, in larger datasets one is more likely to find a close match without the need to warp. As DTW warps less and less, it degenerates to simple ED. This was first noted in [69] and later confirmed in [86] and elsewhere. For completeness, we will show a demonstration of this effect. We measured the leave-one-out nearest neighbor classification accuracy of both DTW and ED on increasingly large datasets containing the CBF and Two-Pat problems, two classic time series benchmarks. Both datasets

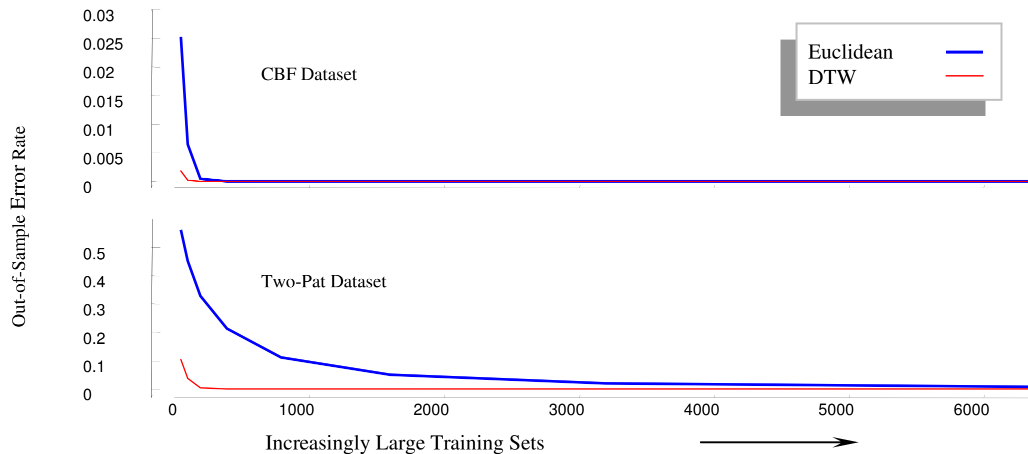

Figure 2.1: The error rate of DTW and ED on increasingly large instantiations of the CBF and Two-Pat problems. For even moderately large datasets, there is no difference in accuracy

allow features to warp up to 1/8 the length of the sequence, so they may be regarded as highly warped datasets. Figure 2.1 shows the result.

As we can see, for small datasets, DTW is significantly more accurate than ED. However, as the datasets get larger, the difference diminishes, and by the time there are mere thousands of objects, there is no measurable difference. In spite of this, and for completeness, we explain in an online Appendix [49] that we can index under DTW with *i*SAX with only trivial modifications.

2.2.2 Time Series Representations

There is a plethora of time series representations proposed to support similarity search and data mining. Table 2.1 shows the major techniques arranged in a hierarchy.

Those representations annotated with an asterisk have the very desirable property of allowing lower bounding. That is to say, we can define a distance measurement on the reduced

Table 2.1: A Hierarchy of Time Series Representations

- Model Based
  - Markov Models
  - Statistical Models
  - Time Series Bitmaps [53]
- Data Adaptive
  - Piecewise Polynomials
    - Interpolation*[64]
    - Regression[76]
  - Adaptive Piecewise Constant Approximation* [47]
  - Singular Value Decomposition*
  - Symbolic
    - Natural Language [68]
    - Strings [39]
      - Non-Lower Bounding [39][6][62]
      - SAX* [55], *i*SAX* [77] [78]
  - Trees
- Non-Data Adaptive
  - Wavelets* [20]
  - Random Mappings [13]
  - Spectral
    - DFT* [28]
    - DCT*
    - Chebyshev Polynomials* [17]
  - Piecewise Aggregate Approximation* [46]
  - IPLA* [24]
- Data Dictated
  - Clipped Data* [10]

abstraction that is guaranteed to be less than or equal to the true distance measured on the raw data. It is this lower bounding property that allows us to use a representation to index the data with a guarantee of no false dismissals [28]. The list of such representations includes (in approximate order of introduction) the discrete Fourier transform (DFT) [28], the discrete Cosine transform (DCT), the discrete Wavelet transform (DWT), Piecewise Aggregate Approximation (PAA) [46], Adaptive Piecewise Constant Approximation (APCA), Chebyshev Polynomials (CHEB) [17] and Indexable Piecewise Linear Approximation (IPLA) [24]. We will provide the first empirical comparison of all these techniques in Section 2.5.

The only lower bounding omissions from our experiments are the eigenvalue analysis techniques such as SVD and PCA. While such techniques give optimal linear dimensionality reduction, we believe they are untenable for massive datasets. For example, while [80] notes that they can transform 70,000 time series in under 10 minutes, this assumes the data can fit in main memory. However, to transform all the out-of-core (disk resident) datasets we consider in this work, SVD would require several months.

There have been several dozen research efforts that propose to facilitate time series search by first symbolizing the raw data [6][39][62]. However, in every case, the authors introduced a distance measure defined on the newly derived symbols. This allows false dismissals with respect to the original data. In contrast, the proposed work uses the symbolic words to internally organize and index the data, but retrieves objects with respect to the Euclidean distance on the original raw data. This point is important enough to restate. Although our proposed representation is an approximation to the original data, and whose creation requires us to

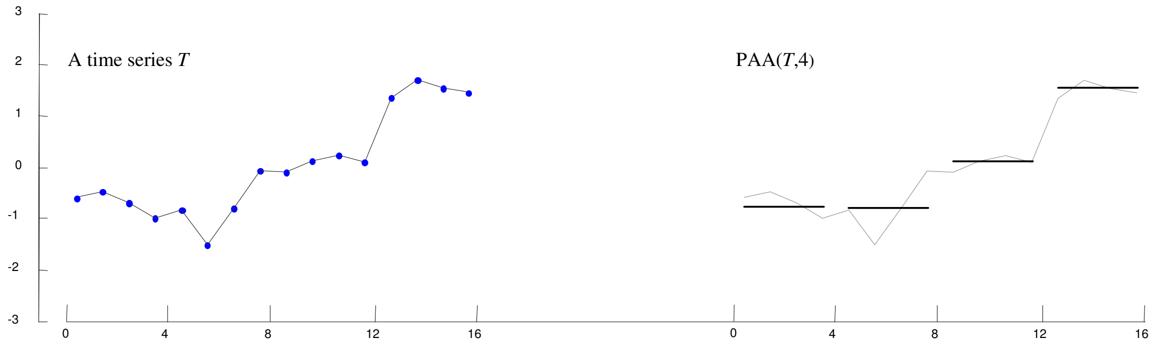

Figure 2.2: *left*) A time series T, of length 16. *right*) A PAA approximation of T, with 4 segments

make a handful of parameters choices, under *any* parameter set the exact search algorithm introduced in Algorithm 2 is guaranteed to find the true exact nearest neighbor.

2.2.3 Review of Classic SAX

The SAX representation was introduced in 2003, since then it has been used by more than 50 groups worldwide to solve a large variety of time series data mining problems [55][45]. For concreteness, we begin with a review of it [55]. In Figure 2.2.*left* we illustrate a short time series T, which we will use as a running example throughout this chapter.

Figure 2.2.*right* shows our sample time series converted into a representation called PAA [46]. PAA represents a time series T of length n in a w-dimensional space by a vector of real numbers, $\overline{T} = \overline{t_1}, \cdots, \overline{t_w}$. The ith element of $\overline{T}$ is calculated by the equation:

$$\overline{t_i} = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} T_j$$

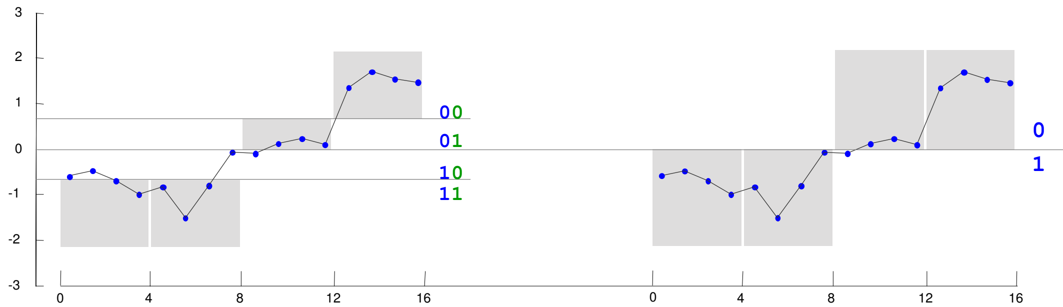

Figure 2.3: A time series T converted into SAX words of cardinality 4 {11,11,01,00} (*left*), and cardinality 2 {1,1,0,0} (*right*)

In the case that n is not divisible by w; the summation can be modified to adopt fractional values. This is illustrated in [55].

PAA is a desirable intermediate representation as it allows for computationally fast dimensionality reduction, provides a distance measure which is lower bounding, and has been shown to be competitive with other dimensionality reduction techniques. In this case, the PAA representation reduces the dimensionality of the time series, from 16 to 4. The SAX representation takes the PAA representation as an input and discretizes it into a small alphabet of symbols with a cardinality of size α. The discretization is achieved by imagining a series of breakpoints running parallel to the x-axis and labeling each region between the breakpoints with a discrete label. Any PAA value that falls within that region can then be mapped to the appropriate discrete value.

While the SAX representation supports arbitrary breakpoints, we can ensure almost equiprobable symbols within a SAX word if we use a sorted list of numbers, $Breakpoints = B_1, \cdots, B_{\alpha-1}$ such that the area under a N(0,1) Gaussian curve from B_i to $B_{i+1} = 1/\alpha$ (B_0 and B_α are

Table 2.2: SAX Breakpoints

β \ α	**2**	**3**	**4**	**5**	**6**	**7**	**8**
β_1	0.00	-0.43	-0.67	-0.84	-0.97	-1.07	-1.15
β_2		0.43	0.00	-0.25	-0.43	-0.57	-0.67
β_3			0.67	0.25	0.00	-0.18	-0.32
β_4				0.84	0.43	0.18	0.00
β_5					0.97	0.57	0.32
β_6						1.07	0.67
β_7							1.15

defined as $-\infty$ and $+\infty$, respectively). Table 2.2 shows a table for such breakpoints for cardinalities from 2 to 8.

A SAX word is simply a vector of discrete symbols. We use a boldface letter to differentiate between a raw time series and its SAX version, and we denote the cardinality of the SAX word with a superscript:

$$SAX(T, w, \alpha) = \mathbf{T}^{\alpha} = \{t_1, t_2, \cdots, t_{w-1}, t_w\}$$

In previous work, we represented each SAX symbol as a letter or integer. Here however, we will use binary numbers for reasons that will become apparent later. For example, in Figure 2.3 we have converted a time series T of length 16 to SAX words. Both examples have a word length of 4, but one has a cardinality of 4 and the other has a cardinality of 2. We therefore have $SAX(T, 4, 4) = \mathbf{T^4} = \{11,11,01,00\}$ and $SAX(T, 4, 2) = \mathbf{T^2} = \{1,1,0,0\}$.

The astute reader will have noted that once we have $\mathbf{T^4}$ we can derive $\mathbf{T^2}$ simply by ignoring the trailing bits in each of the four symbols in the SAX word. As one can readily

Table 2.3: It is possible to obtain a reduced (by half) cardinality SAX word simply by ignoring trailing bits

SAX(T,4,16)	=	$\mathbf{T^{16}}$	=	{ 1100	,1101	,0110	,0001	}
SAX(T,4,8)	=	$\mathbf{T^{8}}$	=	{ 110	,110	,011	,000	}
SAX(T,4,4)	=	$\mathbf{T^{4}}$	=	{ 11	,11	,01	,00	}
SAX(T,4,2)	=	$\mathbf{T^{2}}$	=	{ 1	,1	,0	,0	}

imagine, this is a recursive property. For example, if we convert T to SAX with a cardinality of 8, we have $SAX(T, 4, 8)$ = $\mathbf{T^8}$ = $\{110,110,011,000\}$. From this, we can convert to any lower resolution that differs by a power of two, simply by ignoring the correct number of bits. Table 2.3 makes this clearer.

As we shall see later, this ability to change cardinalities on the fly is a useful and exploitable property. Given two time series T and S, recall that their Euclidean distance is:

$$D(T,S) \equiv \sqrt{\sum_{i=1}^{n} (T_i - S_i)^2}$$

If we have a SAX representation of these two time series, we can define a lower bounding approximation to the Euclidean distance as:

$$MINDIST(\mathbf{T^2}, \mathbf{S^2}) = \sqrt{\frac{n}{w}} \sqrt{\sum_{i=1}^{w} (dist(t_i, s_i))^2}$$

This function requires calculating the distance between two SAX symbols and can be achieved with a lookup table, as in Table 2.4.

Table 2.4: A SAX $dist$ lookup table for α = 4

	00	01	10	11
00	0	0	0.67	1.34
01	0	0	0	0.67
10	0.67	0	0	0
11	1.34	0.67	0	0

The distance between two symbols can be read off by examining the corresponding row and column. For example, $dist$(00,01) = 0 and $dist$(00,10) = 0.67.

For clarity, we will give a concrete example of how to compute this lower bound. Recall our running example time series T which appears in Figure 2.2. If we create a time series S that is simply T's mirror image, then the Euclidean distance between them is $D(T, S)$ = 46.06.

As we have already seen, $SAX(T, 4, 4)$ = $\mathbf{T^4}$ = {11,11,01,00}, and therefore $SAX(S, 4, 4)$ = $\mathbf{S^4}$ = {00,01,11,11}. The invocation of the $MINDIST$ function will make calls to the lookup table shown in Table 2.4 to find:

$$dist(t_1, s_1) = dist(11{,}00) = 1.34$$

$$dist(t_2, s_2) = dist(11{,}01) = 0.67$$

$$dist(t_3, s_3) = dist(01{,}11) = 0.67$$

$$dist(t_4, s_4) = dist(00{,}11) = 1.34$$

Which, when plugged into the $MINDIST$ function, gives:

$$MINDIST(\mathbf{T^2}, \mathbf{S^2}) = \sqrt{\frac{16}{4}}\sqrt{1.34^2 + 0.67^2 + 0.67^2 + 1.34^2}$$

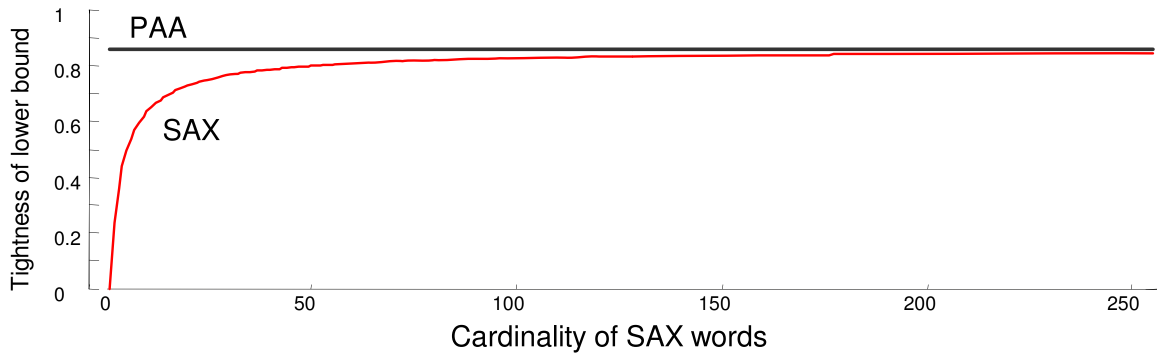

Figure 2.4: The tightness of lower bounds for increasing SAX cardinalities, compared to a PAA/DWT benchmark

$\cdots$ to produce a lower bound value of 4.237. In this case, the lower bound is quite loose; however, having either more SAX symbols or a higher cardinality will produce a tighter lower bound. It is instinctive to ask how tight this lower bounding function can be, relative to natural competitors like PAA or DWT. This depends on the data itself and the cardinality of the SAX words, but coefficient for coefficient, it is surprisingly competitive with the other approaches. To see this, we can measure the *tightness of the lower bounds*, which is defined as the lower bounding distance over the true distance [46]. Figure 2.4 shows this for random walk time series of length 256, with eight PAA or DWT coefficients and SAX words also of length eight. We varied the cardinality of SAX from 2 to 256, whereas PAA/DWT used a constant 4 bytes per coefficient. The results have been averaged over 10,000 random walk time series comparisons.

The results show that for small cardinalities the SAX lower bound is quite weak, but for larger cardinalities it rapidly approaches that of PAA/DWT. At the cardinality of 256, which take 8 bits, the lower bound of SAX is 98.5% that of PAA/DWT, but the latter requires 32

bits. This tells us that if we compare representations, coefficient for coefficient, there is little to choose between them; but if we do bit-for-bit comparisons (cf. Section 2.5.1), SAX allows for much tighter lower bounds. This is one of the properties of SAX that can be exploited to allow ultra scalable indexing.

2.3 The *i*SAX Representation

Because it is tedious to write out binary strings, previous uses of SAX had integers or alphanumeric characters representing SAX symbols [55]. For example:

$$SAX(T,4,8) = \mathbf{T^8} = \{110, 110, 011, 000\} = \{6, 6, 3, 0\}$$

However, this can make the SAX word ambiguous. If we see *just* the SAX word {6,6,3,0} we cannot be sure what the cardinality is (although we know it is at least 7). Since all previous uses of SAX always used a single "hard-coded" cardinality, this has not been an issue. However, the fundamental contribution of this work is to show that SAX allows the comparison of words with different cardinalities, and even different cardinalities *within* a single word. We therefore must resolve this ambiguity. We do this by writing the cardinality as a superscript. For example, in the example above:

$$iSAX(T,4,8) = \mathbf{T^8} = \{6^8, 6^8, 3^8, 0^8\}$$

Because the individual symbols are ordinal, exponentiation is not defined for them, so there is no confusion in using superscripts in this context. Note that we are now using *i*SAX instead of SAX for reasons that will become apparent in a moment. We are now ready to introduce a novel idea that will allow us to greatly expand the utility of *i*SAX.

2.3.1 Comparing Different Cardinality *i*SAX Words

It is possible to compare two *i*SAX words of different cardinalities. Suppose we have two time series, T and S, which have been converted into *i*SAX words:

$$iSAX(T, 4, 8) = \mathbf{T^8} = \{110, 110, 011, 000\} = \{6^8, 6^8, 3^8, 0^8\}$$

$$iSAX(S, 4, 2) = \mathbf{S^2} = \{0\quad, 0\quad, 1\quad, 1\quad\} = \{0^2, 0^2, 1^2, 1^2\}$$

We can find the lower bound between T and S, even though the *i*SAX words that represent them are of different cardinalities. The trick is to *promote* the lower cardinality representation into the cardinality of the larger before giving it to the $MINDIST$ function.

We can think of the tentatively promoted $\mathbf{S^2}$ word as $\mathbf{S^8}$ = $\{0{*}{*}_1, 0{*}{*}_2, 1{*}{*}_3, 1{*}{*}_4\}$, then the question is simply what are correct values of the missing $**_i$ bits? Note that both cardinalities can be expressed as the power of some integer. This guarantees an overlap in the breakpoints used during SAX computation. More concretely, if we have an *i*SAX cardinality of X, and an *i*SAX cardinality of $2X$, then the breakpoints of the former are a proper subset of the latter. This is shown in Figure 2.3.

Using this insight, we can obtain the missing bit values in $\mathbf{S^8}$ by examining each position and computing the bit values at the higher cardinality which are enclosed by the known bits at the current (lower) cardinality and returning the one which is closest in SAX space to the corresponding value in $\mathbf{T^8}$.

This method obtains the $\mathbf{S^8}$ representation usable for $MINDIST$ calculations:

$$\mathbf{S^8} = \{011, 011, 100, 100\}$$

It is important to note that this is not necessarily the same *i*SAX word we would have gotten if we had converted the original time series S. We cannot undo a lossy compression. However, using this *i*SAX word *does* give us an admissible lower bound.

Finally, note that in addition to comparing *i*SAX words of different cardinalities, the promotion trick described above can be used to compare *i*SAX words where *each* word has mixed cardinalities. For example, we can allow *i*SAX words such as $\{111, 11, 101, 0\}$ = $\{7^8, 3^4, 5^8, 0^2\}$. If such words exist, we can simply align the two words in question, scan across each pair of corresponding symbols, and promote the symbol with lower cardinality to the same cardinality as the larger cardinality symbol. In the next section, we explain why it is useful to allow *i*SAX words with different cardinalities.

2.4 *i*SAX Indexing

2.4.1 The Intuition Behind *i*SAX Indexing

As it stands, it may appear that the classic SAX representation offers the potential to be indexed. We could choose a fixed cardinality of, say, 8 and a word length of 4, and thus have 8^4 separate labels for files on disk. For instance, our running example T maps to $\{6^8, 6^8, 3^8, 0^8\}$ under this scheme, and would be inserted into a file that has this information encoded in its name, such as 6.8_6.8_3.8_0.8.txt. The query answering strategy would be very simple. We could convert the query into a SAX word with the same parameters, and then retrieve the file with that label from disk. The time series in that file are likely to be very good approximate matches to the query. In order to find the exact match, we could measure the distance to the best approximate match, then retrieve all files from disk whose label has a $MINDIST$ value less than the value of the best-so-far match. Such a methodology clearly guarantees no false dismissals.

This scheme has a fatal flaw, however. Suppose we have a million time series to index. With 4,096 possible labels, the *average* file would have 244 time series in it, a reasonable number. However, this is the average. For all but the most contrived datasets we find a huge skew in the distribution, with more than half the files being empty, and the largest file containing perhaps 20% of the entire dataset. Either situation is undesirable for indexing, in the former case, if our query maps to an empty file, we would have to do some ad-hoc trick (perhaps trying "misspellings" of the query label) in order to get the first approximate

answer back. In the latter case, if 20% of the data must be retrieved from disk, then we can be at most five times faster than sequential scan. Ideally, we would like to have a user defined threshold th, which is the maximum number of time series in a file, and a mapping technique that ensures each file has at least one and at most th time series in it. As we shall now see, *i*SAX allows us to guarantee exactly this.

*i*SAX offers a multi-resolution, bit aware, quantized, reduced representation with *variable* granularity. It is this variable granularity that allows us to solve the problem above. Imagine that we are in the process of building the index and have chosen th = 100. At some point there may be exactly 100 time series mapped to the *i*SAX word $\{2^4, 3^4, 3^4, 2^4\}$. If, as we continue to build the index, we find another time series maps here, we have an overflow, so we split the file. The idea is to choose one *i*SAX symbol, examine an additional bit, and use its value to create two new files. In this case:

Original File: $\{2^4, 3^4, 3^4, 2^4\}$ splits into $\cdots$

Child file 1: $\{4^8, 3^4, 3^4, 2^4\}$

Child file 2: $\{5^8, 3^4, 3^4, 2^4\}$

Note that in this example we split on the first symbol, promoting the cardinality from 4 to 8. For some time series in the file, the extra bit in their first *i*SAX symbol was a **0**, and for others it was a **1**. In the former case, they are remapped to Child 1, and in the latter, remapped to Child 2. The child files can be named with some protocol that indicates their variable cardinality, for example **5.8**_3.4_3.4_2.4.txt and **4.8**_3.4_3.4_2.4.txt.

The astute reader will have noticed that the intuition here is very similar to the classic idea of extensible hashing. This in essence is the intuition behind building an *i*SAX index, although we have not explained *how* we decide which symbol is chosen for promotion and some additional details. In the next sections, we formalize this intuition and provide details on algorithms for approximately and exactly searching an *i*SAX index.

2.4.2 *i*SAX Index Construction

As noted above, a set of time series represented by an *i*SAX word can be split into two mutually exclusive subsets by increasing the cardinality along one or more dimensions. The number of dimensions d and word length, $w, 1 \leq d \leq w$, provide an upper bound on the fan-out rate. If each increase in cardinality per dimension follows the assumption of iterative doubling, then the alignment of breakpoints contains overlaps in such a way that hierarchical containment is preserved between the common *i*SAX word and the set of *i*SAX words at the finer granularity. Specifically, in iterative doubling, the cardinality to be used after the ith increase in granularity is in accordance with the following sequence, given base cardinality b: $b * 2^i$. The maximum fan-out rate under such an assumption is 2^d.

The use of *i*SAX allows for the creation of index structures that are hierarchical, containing non-overlapping regions [7] (unlike R-trees, etc.[34]), and a controlled fan-out rate. For concreteness, we depict in Figure 2.5 a simple tree-based index structure which illustrates the efficacy and scalability of indexing using iSAX.

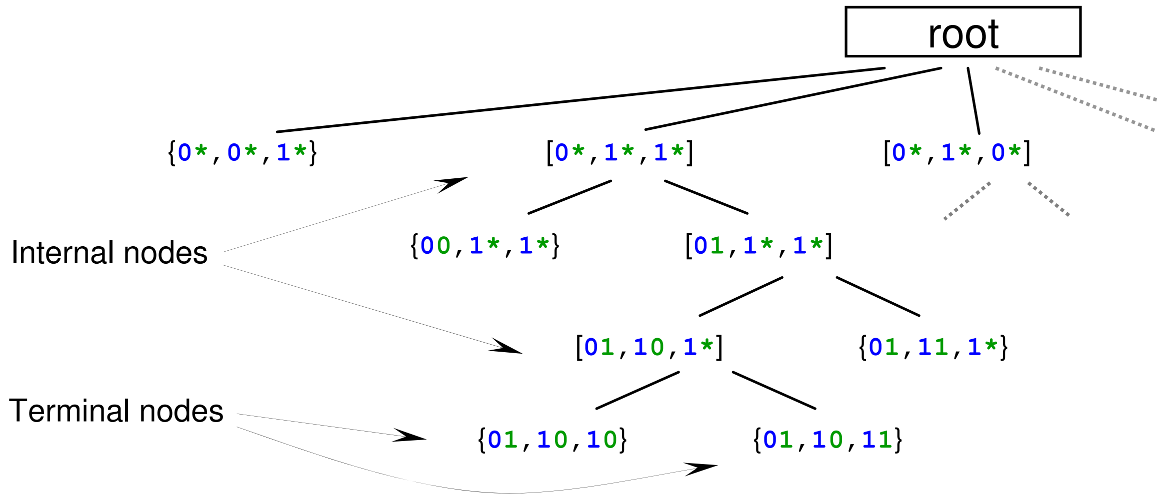

Figure 2.5: An illustration of an *i*SAX index

The index is constructed given base cardinality b, word length w, and threshold th (b is optional; it can be defaulted to 2 or be set for evaluation to begin at higher cardinality). The index structure hierarchically subdivides the SAX space, resulting in differentiation between time series entries until the number of entries in each subspace falls below th. Such a construct is implemented using a tree, where each node represents a subset of the SAX space such that this space is a superset of the SAX space formed by the union of its descendents. A node's representative SAX space is congruent with an *i*SAX word and evaluation between nodes or time series is done through comparison of *i*SAX words. The three classes of nodes found in a tree and their respective functionality are described below:

Terminal Node: A terminal node is a leaf node which contains a pointer to an index file on disk with raw time series entries. All time series in the corresponding index file are characterized by the terminal node's representative *i*SAX word. A terminal node represents the

coarsest granularity necessary in SAX space to enclose the set of contained time series entries. In the event that an insertion causes the number of time series to exceed th, the SAX space (and node) is split to provide additional differentiation.

Internal Node: An internal node designates a split in SAX space and is created when the number of time series contained by a terminal node exceeds th. The internal node splits the SAX space by promotion of cardinal values along one or more dimensions as per the iterative doubling policy. A hash from *i*SAX words (representing subdivisions of the SAX space) to nodes is maintained to distinguish differentiation between entries. Time series from the terminal node which triggered the split are inserted into the newly created internal node and hashed to their respective locations. If the hash does not contain a matching *i*SAX entry, a new terminal node is created prior to insertion, and the hash is updated accordingly. For simplicity, we employ binary splits along a single dimension, using round robin to determine the split dimension.

Root Node: The root node is representative of the complete SAX space and is similar in functionality to an internal node. The root node evaluates time series at base cardinality, that is, the granularity of each dimension in the reduced representation is b. Encountered *i*SAX words correspond to some terminal or internal node and are used to direct index functions accordingly. Un-encountered *i*SAX words during inserts result in the creation of a terminal

node and a corresponding update to the hash table.

Pseudo-code of the insert function used for index construction is shown in Algorithm 1. Given a time series to insert, we first obtain the *i*SAX word representation using the respective *i*SAX parameters at the current node (line 1). If the hash table does not yet contain an entry for the *i*SAX word, a terminal node is created to represent the relevant SAX space, and the time series is inserted accordingly (lines 21-23). Otherwise, there is an entry in the hash table, and the corresponding node is fetched. If this node is an internal node, we call its insert function recursively (line 18). If the node is a terminal node, occupancy is evaluated to determine if an additional insert warrants a split (line 6). If so, a new internal node is created, and all entries enclosed by the overfilled terminal node are inserted (lines 9-15). Otherwise, there is sufficient space and the entry is simply added to the terminal node (line 7).

The deletion function is obvious and omitted for brevity.

2.4.3 Approximate Search

For many data mining applications, an approximate search may be all that is required. An *i*SAX index is able to support very fast approximate searches; in particular, they only require a single disk access. The method of approximation is derived from the intuition that two similar time series are often represented by the same *i*SAX word. Given this assumption, the approximate result is obtained by attempting to find a terminal node in the index with the same *i*SAX representation as the query. This is done by traversing the index in accordance

Algorithm 1 *i*SAX Index Insertion (ts)

```
1:  iSAX_word ← iSAX(ts, this.parameters)
2:
3:  if Hash.ContainsKey(iSAX_word) then
4:      node ← Hash.ReturnNode(iSAX_word)
5:      if node is terminal then
6:          if SplitNode( ) == false then
7:              node.Insert(ts)
8:          else
9:              newnode ← new internal
10:             newnode.Insert(ts)
11:             for each ts ∈ node do
12:                 newnode.Insert(ts)
13:             end for
14:             Hash.Remove(iSAX_word, node)
15:             Hash.Add(iSAX_word, newnode)
16:         end if
17:     else if node is internal then
18:         node.Insert(ts)
19:     end if
20: else
21:     newnode ← new terminal
22:     newnode.Insert(ts)
23:     Hash.Add(iSAX_word, newnode)
24: end if
```

with split policies and matching iSAX representations at each internal node. Because the index is hierarchical and without overlap, if such a terminal node exists, it is promptly identified. Upon reaching this terminal node, the index file pointed to by the node is fetched and returned. This file will contain at least 1 and at most th time series in it. A main memory sequential scan over these time series gives the approximate search result.

In the (very) rare case that a matching terminal node does not exist, such a traversal will fail at an internal node. We mitigate the effects of non-matches by proceeding down the tree, selecting nodes whose last split dimension has a matching iSAX value with the query time series. If no such node exists at a given junction, we simply select the first, and continue the descent.

2.4.4 Exact Search

Obtaining the exact nearest neighbor to a query is both computationally and I/O intensive. To improve search speed, we use a combination of approximate search and lower bounding distance functions to reduce the search space. The algorithm for obtaining the nearest neighbor is presented as pseudo-code in Algorithm 2.

The algorithm begins by obtaining an approximate best-so-far (BSF) answer, using approximate search as described in Section 2.4.3 (lines 1-2). The intuition is that by quickly obtaining an entry which is a close approximation and with small distance to the nearest neighbor, large sections of the search space can be pruned. Once a baseline BSF is obtained,

a priority queue is created to examine nodes whose distance is potentially less than the BSF. This priority queue is first initialized with the root node (line 5).

Because the query time series is available to us, we are free to use its PAA representation to obtain a tighter bound than the $MINDIST$ between two *i*SAX words. More concretely, the distance used for priority queue ordering of nodes is computed using $MINDIST_PAA_iSAX$, between the PAA representation of the query time series and the *i*SAX representation of the SAX space occupied by a node.

Given the PAA representation, T_{PAA} of a time series T and the *i*SAX representation, S_{iSAX} of a time series S, such that $|T_{PAA}| = |S_{iSAX}| = w$, $|T| = |S| = n$, and recalling that the jth cardinal value of S_{iSAX} derives from a PAA value, v between two breakpoints β_L, β_U, $\beta_L < v \leq \beta_U, 1 \leq j \leq w$ we define the lower bounding distance as:

$$MINDIST_PAA_iSAX(T_{PAA}, S_{iSAX}) = \sqrt{\frac{n}{w}}\sqrt{\sum_{i=1}^{w}\begin{cases}(\beta_{Li} - T_{PAAi})^2 & if\ \beta_{Li} > T_{PAAi}\\(\beta_{Ui} - T_{PAAi})^2 & if\ \beta_{Ui} < T_{PAAi}\\0 & otherwise\end{cases}}$$

Recall that we use distance functions that lower bound the true Euclidean distance. That is, if the BSF distance is less than or equal to the minimum distance from the query to a node, we can discard the node and all descendants from the search space without examining their contents or introducing any false dismissals.

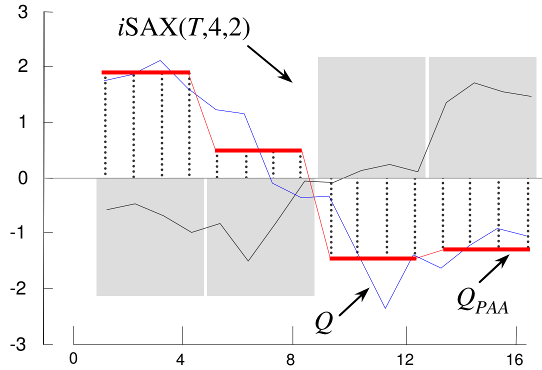

Figure 2.6: MINDIST between PAA and *i*SAX representations. The lower bounding distance is computed from the hatched lines

The algorithm then repeatedly extracts the node with the smallest distance value from the priority queue, terminating when either the priority queue becomes empty or an early termination condition is met. Early termination occurs when the lower bound distance we compute equals or exceeds the distance of the BSF. This implies that the remaining entries in the queue cannot qualify as the nearest neighbor and can be discarded.

If the early termination condition is not met (line 9), the node is further evaluated. In the case that the node is a terminal node, we fetch the index file from disk and compute the distance from the query to each entry in the index file, recording the minimum distance (line 13). If this distance is less than our BSF, we update the BSF (lines 15-16). In the case that the node is an internal node or the root node, its immediate descendents are inserted into the priority queue (lines 19-22). The algorithm then repeats by extracting the next minimum node from the priority queue.

Before leaving this section, we note that we have only discussed 1-NN queries. Extensions to k-NN and range queries are trivial and obvious, and are omitted for brevity.

Algorithm 2 Exact Search (ts)

```
1:  BSF.IndexFile ← ApproximateSearch(ts)
2:  BSF.dist ← IndexFileDist(ts, BSF.IndexFile)
3:
4:  PriorityQueue pq
5:  pq.Add(root)
6:
7:  while !pq.IsEmpty do
8:      min ← pq.ExtractMin()
9:      if min.dist ≥ BSF.dist then
10:         break
11:     end if
12:     if min is terminal then
13:         tmp ← IndexFileDist(ts, min.IndexFile)
14:         if BSF.dist > tmp then
15:             BSF.dist ← tmp
16:             BSF.IndexFile ← min.IndexFile
17:         end if
18:     else if min is internal or root then
19:         for each node ∈ min.children do
20:             node.dist ← MINDIST_PAA_iSAX(ts,node.iSAX)
21:             pq.Add(node)
22:         end for
23:     end if
24: end while
25:
26: return BSF.IndexFile
```

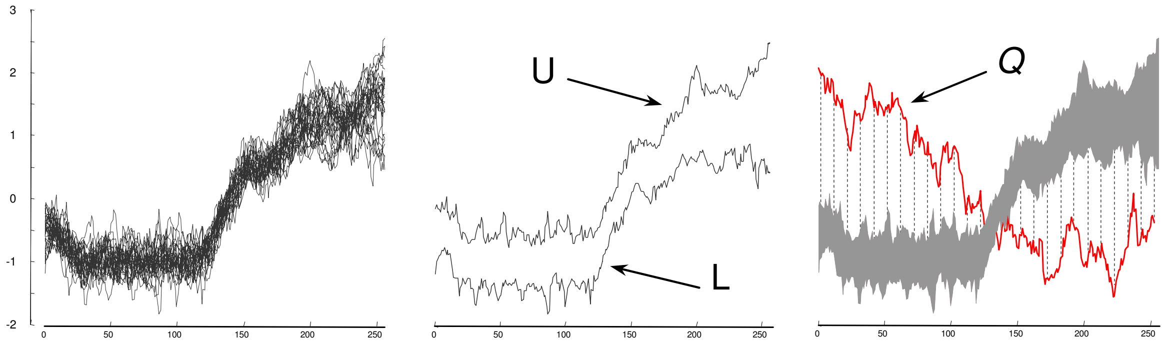

Figure 2.7: *left*) A set of time series which all map to the *i*SAX word $\{2^8, 1^8, 1^8, 1^8, 1^8, 1^8, 1^8, 1^8, 3^8, 5^8, 2^4, 3^4, 3^4, 3^4, 3^4, 3^4\}$. *center*) The maximum/minimum values for the set can be used to define upper/lower wedges. *right*) The square root of the sum of squared lengths of the hatch lines is a lower bound to the Euclidean distance between Q and any within the set

Extension with Time Series Wedges

Extensions to the index are readily facilitated as meta-information can be held in nodes. This allows the index to supplant or be used in concert with external techniques. For experimental purposes, and to expedite exact search, we modified index terminal nodes by storing meta-data which are used to obtain a lower bounding distance to the set of contained time series at each terminal node. This distance is a potentially tighter bound than that of $MINDIST_PAA_iSAX$.

Specifically, terminal nodes in the index now maintain a record of the minimum and maximum value per dimension from the set of contained time series as an upper and lower wedge, a technique described in [84] and illustrated in Figure 2.7. Given that a terminal node in a non-trivial tree is essentially a grouping of similar time series, we expect the wedges to be tight; making them an advantageous addition for search space pruning. When exact search encounters a terminal node and early termination conditions are not met, we compute

a second lower bounding distance using LB_Keogh [84] from the recorded wedges. As the upper and lower wedge is saved as meta-data in each terminal node, the LB_Keogh computation does not require additional disk accesses. If this distance is greater or equal to the BSF, we can safely discard the terminal node from consideration without fetching its index file from disk. Given that repeated disk accesses can become prohibitively expensive; the addition of wedges has significant utility.

2.5 Experimental Evaluation

We begin by discussing our experimental philosophy. We have designed all experiments such that they are not only reproducible, but *easily* reproducible. To this end, we have built a webpage which contains all datasets used in this work, together with spreadsheets which contain the raw numbers displayed in all the figures [49]. In addition, the webpage contains many additional experiments; however, we note that this dissertation is completely self-contained.

We have used random walk datasets for much of our experimental work because it is known to model stock market data very well, and for the simple pragmatic reason that it is easy to create arbitrarily large datasets, which can be exactly recreated by others who only need to know the seed value. We note, however, that in this work we also test on heartbeat and insect data, which are very different from random walks, and in the website built to support this work we show results on 30 diverse datasets.

Experiments are conducted on an AMD Athlon 64 X2 5600+ with 3GB of memory, Windows XP SP2 with /3GB switch enabled, and using version 2.0 of the .NET Framework. All experiments used a 400GB Seagate Barracuda 7200.10 hard disk drive with the exception of the 100M random walk experiment, which required additional space, there we used a 750GB Hitachi Deskstar 7K10000.

2.5.1 Tightness of Lower Bounds

It is important to note that the rivals to *i*SAX are other time series representations, not indexing structures such as R-Trees, VP-Trees, etc [26]. We therefore begin with a simple experiment to compare the tightness of lower bounds of *i*SAX with the other lower bounding time series representations, including DFT, DWT, DCT, PAA, CHEB, APCA and IPLA. We measure TLB, the tightness of lower bounds [46]. This is calculated as:

$$TLB = \frac{LowerBoundDist(T, S)}{TrueEuclideanDist(T, S)}$$

Because DWT and PAA have exactly the same TLB [46] we show one graphic for both. We randomly sample T and S (with replacement) 1,000 times for each combination of parameters. We vary the time series length [480, 960, 1440, 1920] and the number of bytes per time series available to the dimensionality reduction approach [16, 24, 32, 40]. We assume that each real valued representation requires 4 bytes per coefficient, thus they use [4, 6, 8,

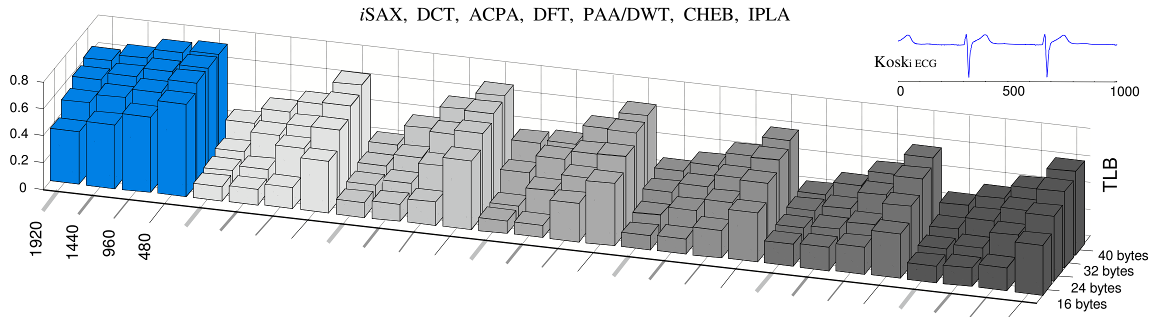

Figure 2.8: The tightness of lower bounds for various time series representations on the Koski ECG dataset. Similar graphs for thirty additional datasets can be found at [49]

10] coefficients. For *i*SAX, we hard code the cardinality to 256, resulting in [16, 24, 32, 40] symbols per word.

Recall that, for TLB, larger values are better. If the value of TLB is zero, then any indexing technique is condemned to retrieving every object from the disk. If the value of TLB is one, then there is no search, we could simply retrieve one object from disk and guarantee that we had the true nearest neighbor. Figure 2.8 shows the result of one such experiment with an ECG dataset.

Note that the speedup obtained is generally non-linear in TLB, that is to say if one representation has a lower bound that is twice as large as another, we can usually expect a *much* greater than two-fold decrease in disk accesses. In a sense, it may be obvious before doing this experiment that *i*SAX will have a smaller reconstruction error, thus a tighter lower bound, and greater indexing efficiency than the real valued competitors. This is because *i*SAX is taking advantage of every bit given to it. In contrast, for the real valued approaches it is clear that the less significant bits contribute *much* less information than the significant bits. If the raw time series is represented with 4 bytes per data point, then each real valued

coefficient must also have 4 bytes (recall that orthonormal transforms are merely rotations in space). This begs the question, why not quantize or truncate the real valued coefficients to save space? In fact, this is a very common idea in compression of time series data. For example, in the medical domain it is frequently done for both the wavelet [23] and cosine [11] representations. However, recall that we are not interested in compression per se. Our interest is in dimensionality reduction that allows indexing with no false dismissals. If, for the other approaches, we save space by truncating the less significant bits, then at least under the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) default policy for rounding (RoundtoNearest) it is possible the distance between two objects can *increase*, thus violating the no false dismissals guarantee. We have no doubt that an indexable bit-adjustable version of the real valued representations could be made to work, however, none exists to date.

Even if we naïvely coded each *i*SAX word with the same precision as the real valued approaches (thus wasting 75% of the main memory space), *i*SAX is still competitive with the other approaches; this is shown in Figure 2.9. Before leaving this section, we note that we have repeated these experiments with thirty additional datasets from very diverse domains with essentially the same results [49].

2.5.2 Sensitivity to Parameters

For completeness, we conduct experiments which evaluate the sensitivity of *i*SAX indexing to parameter values. Recall that an *i*SAX index is constructed given the following: base cardinality b, word length w, and a threshold th (the maximum number of entries in a leaf

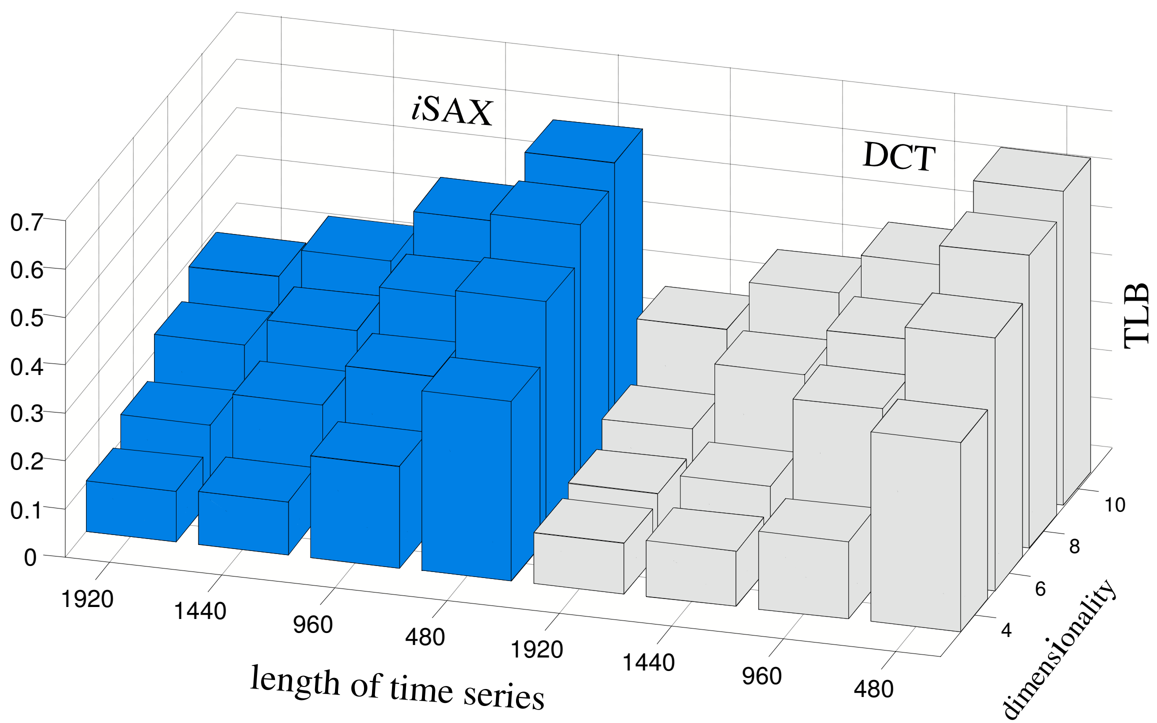

Figure 2.9: The experiment in the previous figure redone with the *i*SAX word length equal to the dimensionality of the real valued applications (just DCT is shown to allow a “zoom in”)

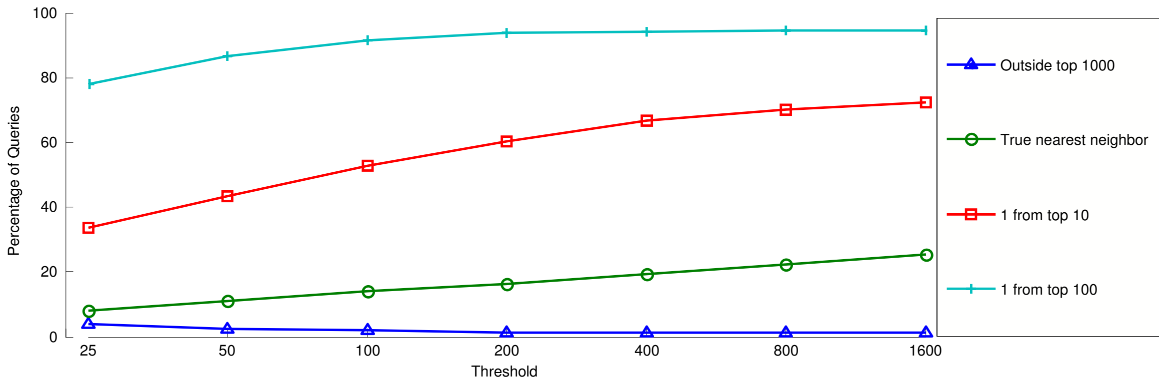

Figure 2.10: Approximate search rankings for increasing threshold values

node). Our analysis focuses on the parameters w and th. We exclude the evaluation of b, which is used during computation of new cardinal values, as this is a procedure which inherently conforms to the size and skew of the indexed data. For b, any low value will suffice. The following analysis identifies the characteristics of an *i*SAX index containing 1 million random walk time series of length 256 from the averaged results of 1000 approximate queries, with respect to a range of th and w values. The quality of index performance can be gauged by consideration of both the number of index files created as well as the rank of approximate search results. For approximate search rankings, we measure the percentage of queries which returns an entry which is the true nearest neighbor, an entry which ranks within the top 10 nearest neighbors, an entry which ranks within the top 100 nearest neighbors, and an entry which ranks outside the top 1000 nearest neighbors. Increases in the first three measures or a decrease in the final, indicate a favorable trend with respect to the quality of index results. The experimental analysis below validates our choice of parameter values used in later sections.

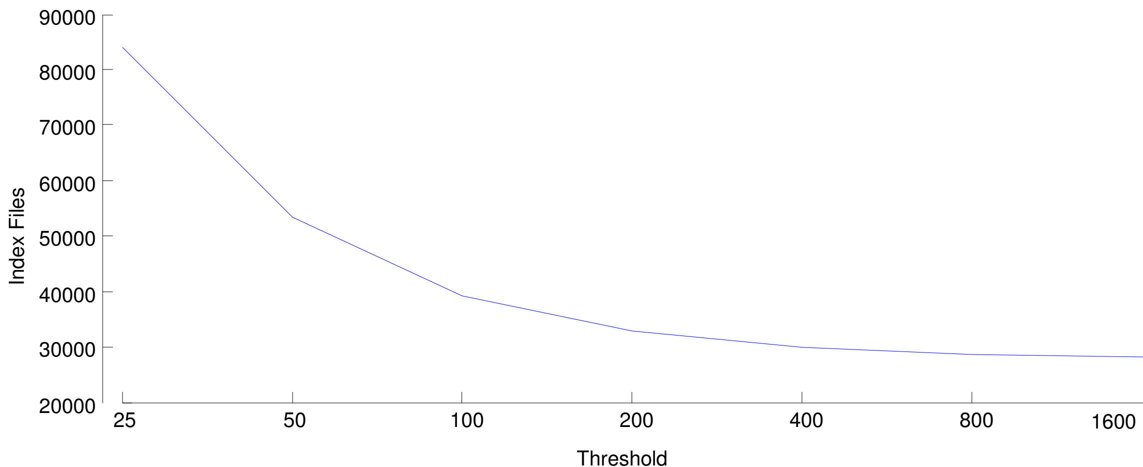

Figure 2.11: Index files created across varying threshold values

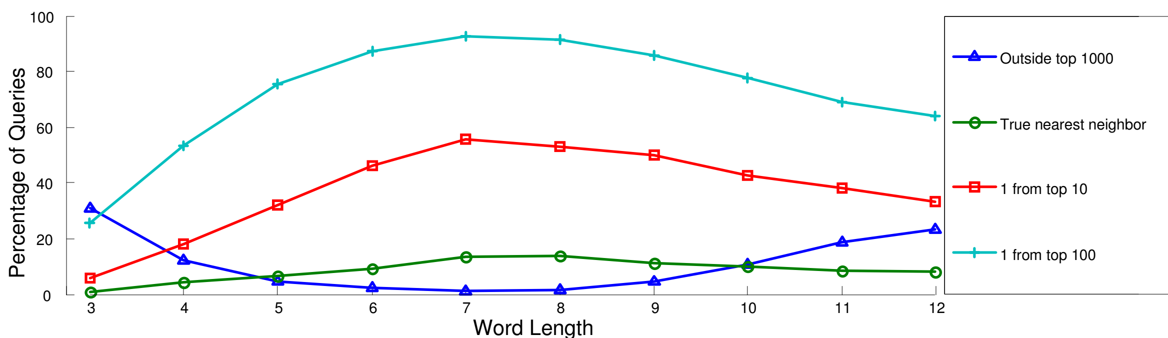

Figure 2.12: Approximate search rankings for increasing word lengths

In Figure 2.10 and Figure 2.11 we vary the th value between [25, 50, 100, 200, 400, 800, 1600] while keeping w and b stationary at 8 and 4, respectively. As illustrated by the gradually sloped curves in Figure 2.10, index performance is not sharply affected by th values. Therefore, the determination of an adequate th value rests on the tradeoff between possible entries retrieved (th), and the number of index files created. Our choice of $th = 100$ in Section 2.5.3 is affirmed as a suitable choice as this is characterized by both a low number of entries examined and by having the number of index files created approach the bottom of the curve in Figure 2.11.

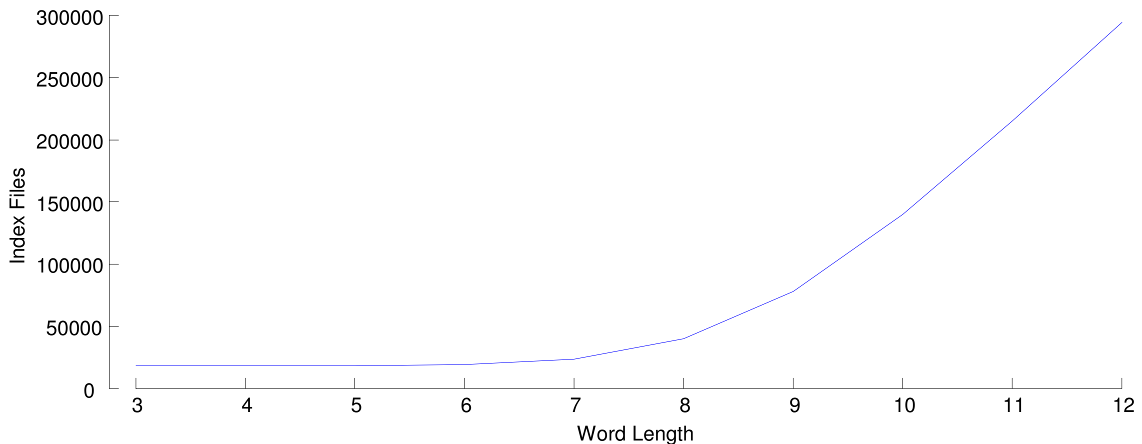

Figure 2.13: Index files created across varying word lengths

In Figure 2.12 and Figure 2.13, we vary the value of w between [3-12] while keeping th and b stationary at 100 and 4, respectively. The results indicate that index performance is not highly dependent on the selection of very precise w values. In Figure 2.12, there exists a range of values, [6-9], where approximate search rankings maintain a high level of performance. We observe some degradation in performance with increasingly longer word lengths, though this is expected as smaller segments result in increased sensitivity to noise. We also examined the number of index files created and showed that this number increases with w (though for low values of w, there may be a minimum number of index files necessary to support the dataset, given th). This increase in index files is an expected trend, as an increase in w corresponds to an increase in the set of possible *i*SAX words (which are used for index filenames). Our analysis affirms our choice of $w = 8$ in Section 2.5.3 as a suitable value, falling in the range of w which returns quality results while also at the lower end of the spectrum with regards to index files created.

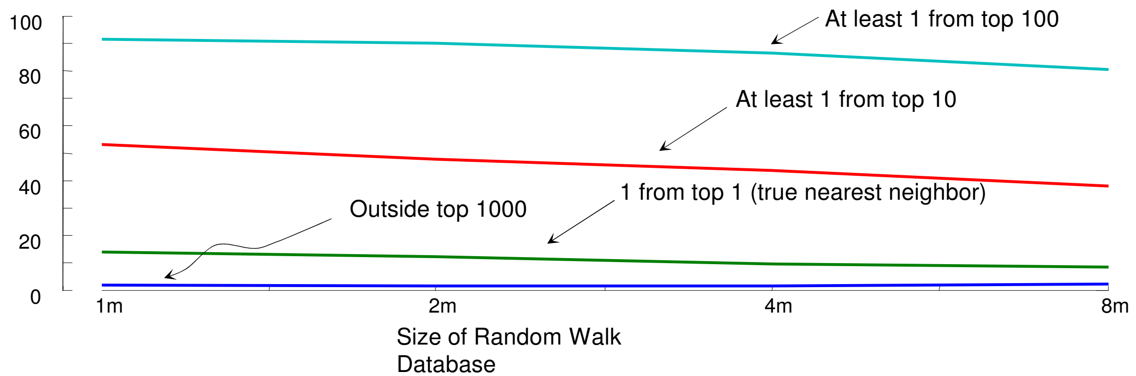

Figure 2.14: The percentage of cutoffs for various rankings, for increasingly large databases with approximate search

Our analysis of index characteristics across a range of parameter values have shown that parameters should be selected in consideration of both search performance as well as the number of index files constructed. The selections of these key parameters, while critical, have been shown to be generally flexible and competitive across a range of values and without the need for exact tuning.

2.5.3 Indexing Massive Time Series Datasets

We tested the accuracy of approximate search for increasingly large random walk databases of sequence length 256, containing [one, two, four, eight] million time series. We used $b = 4, w = 8$, and $th = 100$. This created [39,255; 57,365; 92,209; 162,340] files on disk. We generated 1,000 queries, did an approximate search, and then compared the results with the true ranking which we later obtained with a sequential scan. Figure 2.14 shows the results.

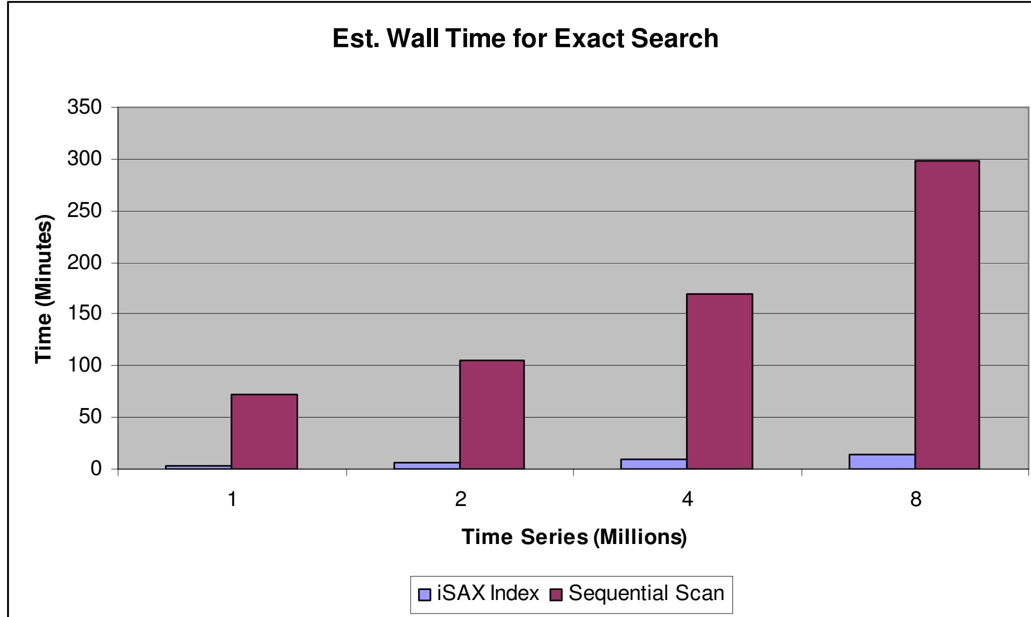

Figure 2.15: Estimated wall clock time for exact search averaged over 100 queries

The figure tells us that when searching one million time series, 91.5% of the time approximate search returns an answer that would rank in the top 100 of the true nearest neighbor list. Furthermore, that percentage only slightly decreases as we scale to eight million time series. Likewise, again, for one million objects, more than half the time the approximate searches return an object that would rank in the top 10, and 14% of the time it returns the *true* nearest neighbor. Recall that these searches require exactly one disk access and at most 100 Euclidean distance calculations, so the average time for a query was less than a second.

We also conducted exact search experiments on 10% of the queries. Figure 2.15 shows the estimated wall clock time and Figure 2.16 shows the average disk I/O for exactly finding the nearest neighbor using the *i*SAX index. Sequential scan is used as a baseline for comparison.

To push the limits of indexing, we considered indexing 100,000,000 random walk time series of length 256. To the best of our knowledge, this is as least two orders of magnitude

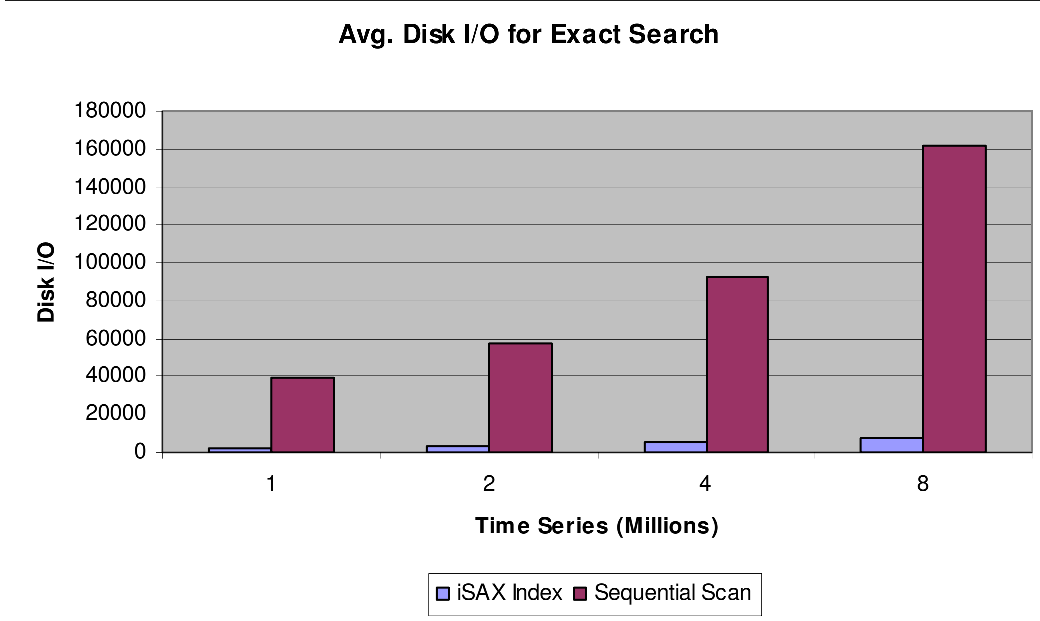

Figure 2.16: Average disk I/O for exact search averaged over 100 queries

larger that any other dataset considered in the literature [7][17][28][62]. Since the publication of *Don Quixote de la Mancha* in the 17th century, the idiom, "*a needle in a haystack*" has been used to signify a near impossible search. If each time series in this experiment was represented by a piece of hay the size of a drinking straw, they would form a cube shaped haystack with 262 meter sides.

Because of the larger size of data, we increased th to 2,000, and used w of 16. This created 151,902 files occupying a half terabyte of disk space. The average occupancy of index files is approximately 658.

We issued ten new random walk approximate search queries. Each query was answered in an average of 1.15 seconds. To find out how good each answer was, we did a linear scan of the data to find the true rankings of the answers. Three of the queries did actually discover their true nearest neighbor, the average rank was 8, and the worst query "only" managed to

retrieve its 25th nearest neighbor. In retrospect, these results are extraordinarily impressive. Faced with one hundred million objects on disk, we can retrieve only 0.0013895% of the data and find an object that is ranked the top 0.0001%. As we shall see in Sections 2.5.5/2.5.6, the extraordinary precision and speed of approximate search combined with fast exact search allows us to consider mining datasets with millions of objects.

We also conducted exact searches on this dataset; each search took an average of 90 minutes to complete, in contrast to a linear scan taking 1,800 minutes.

2.5.4 Approximate Search Evaluation

Approximate search, being orders of magnitude faster than exact search, is inherently attractive for many problems. Because the returned results are approximate in nature, it is necessary for us to ascertain the general quality and effectiveness of said results. We have seen in Section 2.5.3 some measure of this and we reaffirm its utility here with additional visual and quantitative evaluations.

In the arid to semi-arid regions of North America, the Beet leafhopper (*Circulifer tenellus*) is the only known vector (carrier) of curly top virus, which causes major economic losses in a number of crops including sugarbeet, tomato, and beans [43]. In order to mitigate these financial losses, entomologists at the University of California, Riverside are attempting to model and understand the behavior of this insect. It is known that the insects feed by sucking sap from living plants; much like how mosquitoes suck blood from mammals and birds. In order to understand the insect's behaviors, entomologists' glue a thin wire to the insect's

back, complete the circuit through a host plant, and then measure fluctuations in voltage level to create an Electrical Penetration Graph (EPG), a time series, which can then be mined for clues to insect behavior. The problem facing the entomologists is that these experiments have left them with massive data collections which are difficult to search.

We indexed the entire insect data archive of 4,232,591 subsequences of length 150 using $b = 4, w = 8, th = 100$. We asked the entomologist Dr. Greg Walker to draw a query time series. He was interested in knowing if the database contained any examples of a pattern called "Waveform **A**", which he noted is characterized by "*an almost vertical increase in the voltage level from baseline. Immediately after this spike, there is a gradual decline in voltage level which occurs as a smooth downward curve*". This pattern is produced during the initial penetration of the plant tissue through the epidermis.

The idealized query time series and corresponding approximate search result is shown in Figure 2.17. As shown by the figure, a simple approximate search is capable of retrieving a matching shape and corresponding location to researchers for further analysis. Although this experiment searched a database of over four million time series, the result was returned in less than a second, allow rapid interaction and hypothesis testing for the scientist.

Section 2.5.3 identified characteristics of approximate search in the form of nearest neighbor rankings. In this section, we quantify the effectiveness of approximate search results via comparison with actual nearest neighbors. We indexed 9,999,745 random walk time series subsequences of length 256 with parameters $b = 4, w = 8, th = 150$. 100 random walk queries were generated and the approximate search result of each was obtained. To quantify

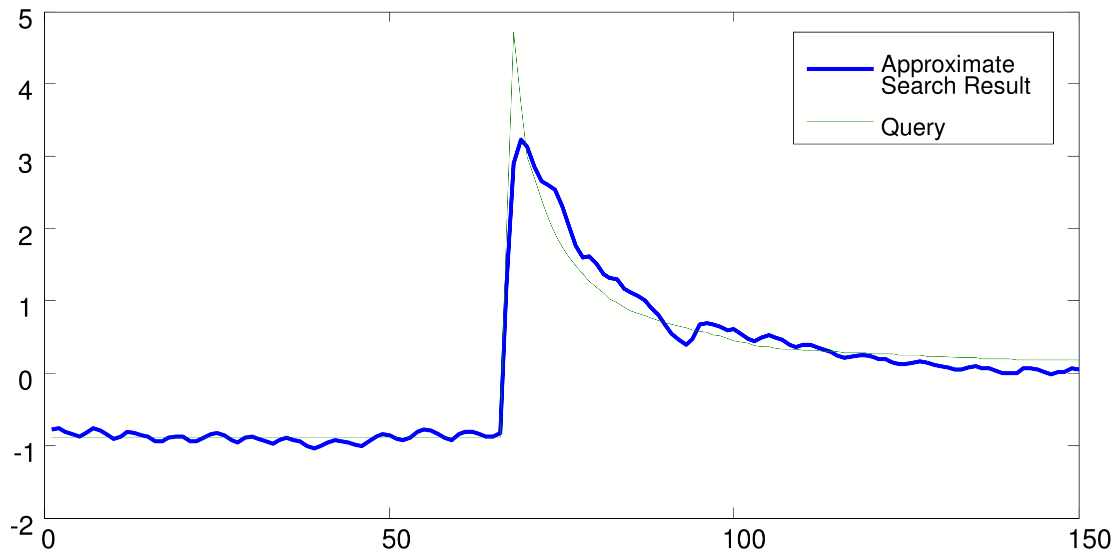

Figure 2.17: Approximate search result on insect dataset

the quality of approximate search results we formulate a distance ratio which compares the true nearest neighbor distance and the approximate search distance. Let time series Q, A, T be the query, the approximate result, and the true nearest neighbor, respectively. Calculate:

$$DistanceRatio = \frac{EuclideanDist(Q,T)}{EuclideanDist(Q,A)}$$

This distance ratio is an indicator of how similar the approximate result is, relative to that of the true nearest neighbor. Figure 2.18 shows the distance ratio for each of the 100 queries, sorted in ascending order. All ratios are above 0.69, which indicates no approximate result deviates significantly from the actual nearest neighbor. For additional illustration on the quality of approximate results, the Q, A, T set of time series corresponding to the lower median of distance ratios (0.907) is shown in Figure 2.19. In fact, it is extremely hard to make

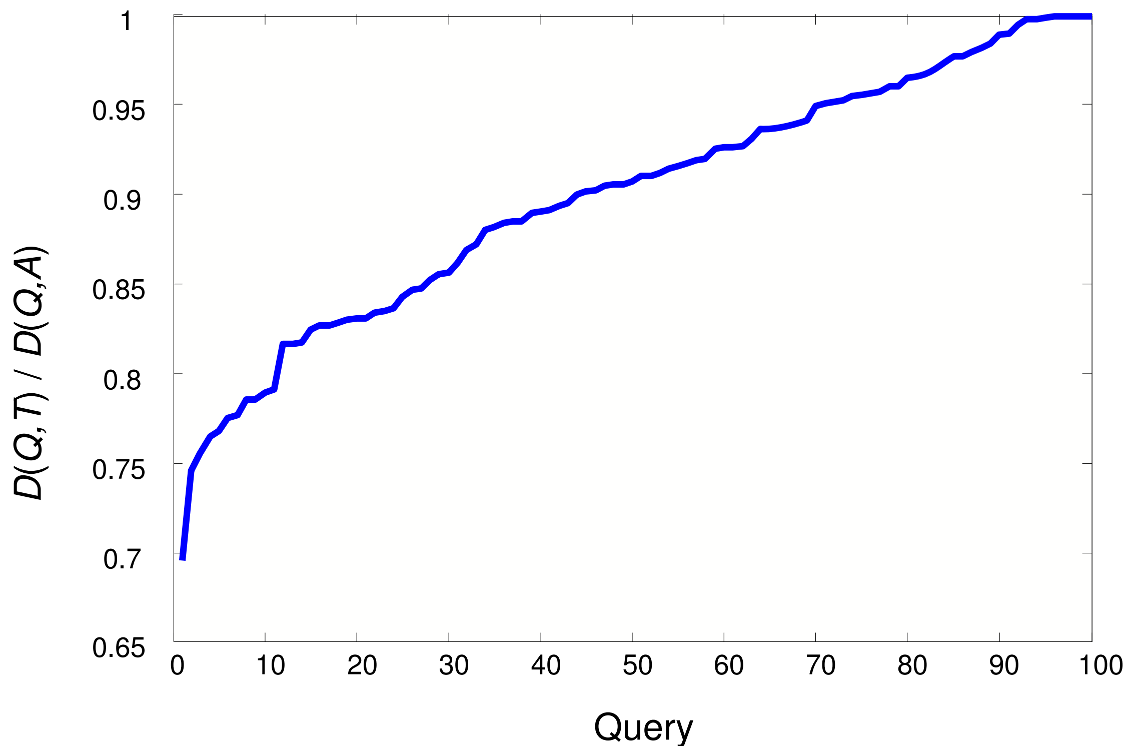

Figure 2.18: Sorted distance ratios of 100 random walk queries

any visual determination as to which plot is the approximate result and which corresponds to the actual nearest neighbor (without the aid of a legend).

2.5.5 Time Series Set Difference

In this section, we give an example of a data mining algorithm that can be built on top of our existing indexing algorithms. The algorithm is interesting in that it uses both approximate search and exact search to compute the ultimate (exact) answer to a problem.

Suppose we are interested in contrasting two collections of time series data. For example, we may be interested in contrasting telemetry from the last Shuttle launch with telemetry from all previous launches, or we may wish to contrast the ten minutes of electrocardiograms

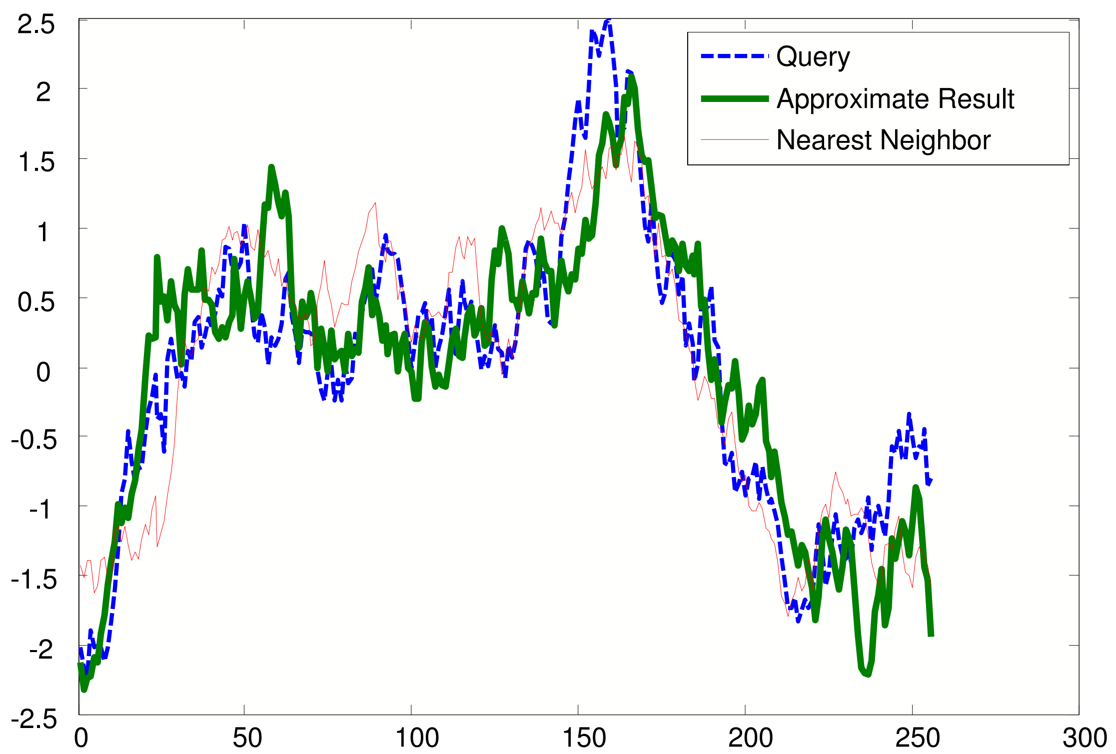

Figure 2.19: Plot showing the random walk query, approximate result, and true nearest neighbor. Together they correspond to the lower median of distance ratios computed

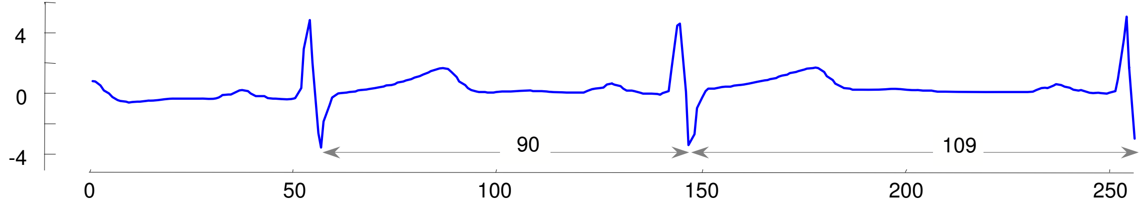

Figure 2.20: The Time Series Set Difference discovered between ECGs recorded during a waking cycle and the previous 7.2 hours

just before a patient wakes up with the preceding seven hours of sleep. To do this, we define the Time Series Set Difference ($TSSD$) operator:

Definition 2.1. Time Series Set Difference(A, B). Given two collections of time series A and B, the time series set difference is the time series in A whose distance from its nearest neighbor in B is maximal.

Note that we are not claiming that this is the best way to contrast two sets of time series; it is merely a sensible definition we can use as a starting point.

We tested this definition on an electrocardiogram dataset. The data is an overnight polysomnogram with simultaneous three-channel Holter ECG from a 45 year old male subject with suspected sleep-disordered breathing. We used the first 7.2 hours of the data as the reference set B, and the next 8 minutes 39 seconds as the "novel" set A. The set A corresponds to the period in which the subject woke up. After indexing the data with an *i*SAX word length of 9 and a maximum threshold value of 100, we had 1,000,000 time series subsequences in 31,196 files on disk, occupying approximately 4.91GB of secondary storage. Figure 2.20 show the TSSD discovered.

We showed the result to UCLA cardiologist Helga Van Herle. She noted that the p-waves in each of the full heartbeats look the same, but there is a 21.1% increase in the length of the second one. This indicated to her that this is almost certainly an example of sinus arrhythmia, where the R-R intervals are changing with the patients breathing pattern. This is likely due to slowing of the heart rate with expiration and increase of the heart rate with inspiration, given that it is well known that respiration patterns change in conjunction with changes in sleep stages [73].

An obvious naïve algorithm to find the TSSD is to do 20,000 exact searches, one for each object in A. This requires ("only") 325,604,200 Euclidean distance calculations, but it requires approximately 5,676,400 disk accesses, for 1.04 days of wall clock time. This is clearly untenable.

We propose a simple algorithm to find the TSSD that exploits the fact that we can do both ultra fast approximate search and fast exact search. We assume that set B is indexed and that set A is in main memory. The algorithm is sketched out in Algorithm 3.

The algorithm begins by obtaining the approximate nearest neighbor in B for each time series in A (lines 4-9). A priority queue is created to order each time series in A according to the distance to its approximate nearest neighbor. Given that approximate search results are generally close to the exact answer, examining priority queue entries in order of descending distance is likely to be an effective heuristic in finding the entry which has the maximum nearest neighbor distance. The algorithm makes a minor addition to the exact search algorithm described previously. An additional parameter, nextDist, is required and denotes the

distance value of the next entry at the top of the priority queue. If at any point during the exact search, the best-so-far falls below nextDist we suspend the search and return null. We can determine the state of exact search by checking the IndexFile for value (line 20). If the search was suspended, we reinsert the entry with its partially suspended state and updated distance back into the priority queue (line 21). Otherwise, if the IndexFile contains a search result, then we have obtained an exact answer whose nearest neighbor is larger than any other entry remaining in the priority queue, the TSSD (line 23).

Algorithm 3 An outline to find the TSSD (A, B)

```
 1: // sort priority queue by entry dist
 2: PriorityQueue pq
 3:
 4: for each ts ∈ A do
 5:         IndexFile ← B.ApproximateSearch(ts)
 6:         entry.dist ← IndexFileDist(ts,IndexFile)
 7:         entry.ts ← ts;
 8:         pq.Add(entry)
 9: end for
10:
11: while !pq.IsEmpty do
12:         entry ← pq.ExtractMax( )
13:         nextDist← pq.FindMax( ).dist
14:
15:         // exact search is suspended if the best-so-far
16:         // becomes greater than nextDist
17:         IndexFile ← B.ExactSearch(entry,nextDist)
18:
19:         // IndexFile returns null when search is suspended
20:         if IndexFile == null then
21:                 pq.Add(entry)
22:         else
23:                 return  Indexfile
24:         end if
25: end while
```

To find the discordant heartbeats shown in Figure 2.20, our algorithm did 43,779 disk accesses (20,000 in the first approximate stage, and the remainder during the refinement search phase), and performed 2,365,553 Euclidean distance calculations. The number of disk accesses for a sequential scan algorithm is somewhat better; it requires only 31,196 disk reads, about 71% of what our algorithm required. However, sequential scan requires 20,000,000,000 Euclidean distance calculations, which is 8,454 times greater than our approach and would require an estimated 6.25 days to complete. In contrast, our algorithm takes only 34 minutes.

Our algorithm is much faster because it exploits the fact that that most candidates in set A can be quickly eliminated by very fast approximate searches. In fact, of the 20,000 objects in set A for this experiment, only two of them (obviously including the eventual answer) had their true nearest neighbor calculated. Of the remainder, 17,772 were eliminated based only on the single disk access made in phase one of the algorithm, and 2,226 required more than one disk access, but less than a compete nearest neighbor search.

2.5.6 Batch Nearest Neighbor Search

We consider another problem which can be exactly solved with a combination of approximate and exact search. The problem is that of batch nearest neighbor search. We begin with a concrete example of the problem before showing our *i*SAX-based solution. Here the context of DNA is used to provide a real world dataset with results which can be easily verified.

It has long been known that all the great apes except humans have 24 chromosomes. Humans, having 23, are quite literally the odd man out. This is widely accepted to be a result of an end-to-end fusion of two ancestral chromosomes. Suppose we do not know which of the ancestral chromosomes were involved in the fusion, we could attempt to efficiently discover this with *i*SAX.

We begin by converting DNA into time series. There are several ways to do this; here we use the simple approach shown in Algorithm 4.

Algorithm 4 An algorithm for converting DNA to time series

```
1:  T_1 ← 0
2:
3:  for i ← 1 to length(DNAstring) do
4:      if DNAstring_i == A then
5:          T_{i+1} ← T_i + 2
6:      end if
7:      if DNAstring_i == G then
8:          T_{i+1} ← T_i + 1
9:      end if
10:     if DNAstring_i == C then
11:         T_{i+1} ← T_i − 1
12:     end if
13:     if DNAstring_i == T then
14:         T_{i+1} ← T_i − 2
15:     end if
16: end for
```

We converted Contig NT_005334.15 of the human chromosome 2 to time series in this manner, and then indexed all subsequences of length 1024 using a sliding window. There are a total of 11,246,491 base pairs (approximately 2,100 pages of DNA text written in this paper's format) and a total of 5,622,734 time series subsequences written to disk.

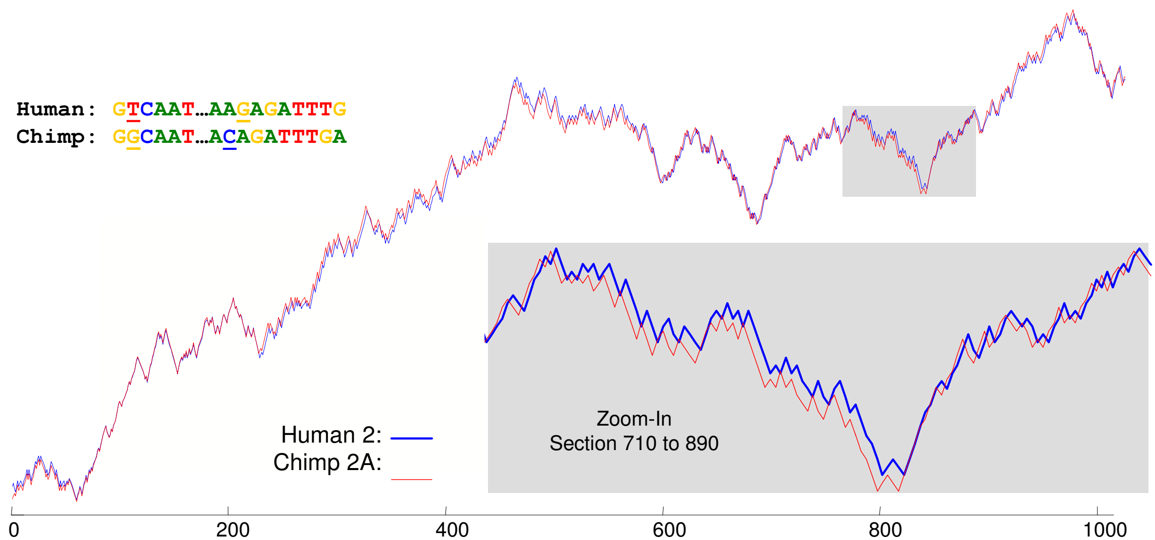

Figure 2.21: Corresponding sections of human and chimpanzee DNA

We converted 43 randomly chosen subsequences of length 1024 of chimpanzee's (Pan troglodytes) DNA in the same manner. We made sure that the 43 samples included at least one sample from each of the chimps 24 chromosomes.

We performed a search to find the chimp subsequence that had the nearest nearest-neighbor in the human reference set. Figure 2.21 shows the two subsequences plotted together. Note that while the original DNA strings are very similar, they are not identical.

Once again, this is a problem where a combination of exact and approximate search can be useful. To speed up the search we use *batch nearest neighbor search*. We define this as the search for the object O in a (relatively small) set A, which has the smallest nearest neighbor distance to an object in a larger set B. Note that to solve this problem, we really only need one exact distance, for object O, to be known. For the remaining objects in A, it suffices to know that a lower bound on their nearest neighbors is greater than the distance from O to its nearest neighbor. With this in mind, we can define an algorithm which is generally much

faster than performing exact search for each of the objects in A. Algorithm 5 outlines the methodology.

The algorithm begins by obtaining the approximate search result for each time series in A (lines 4-14). Exact search is then performed on the query time series with the minimum approximate distance as an initial batch nearest neighbor candidate (line 17). We then perform exact search on the remaining time series in A using the current candidate as the initial distance to use during exact search (this requires a simple modification of lines 1-2 in the exact search algorithm detailed previously). By using the current batch nearest neighbor candidate as the initial distance value, we begin exact search immediately with a reduced distance value, increasing the likelihood of search space pruning from the very onset. If exact search returns a value, then a nearest neighbor less than the current best-so-far was found, and we update the best-so-far accordingly (lines 24-25). Once all time series are examined, the current best-so-far is returned as the batch nearest neighbor.

We can see this algorithm as an anytime algorithm [86][92]. Recall that an anytime algorithm has the advantage that the quality of results increases monotonically with time and that execution is interruptible (after initial setup). Now considering the effect of diminishing returns and possible temporal constraints, it may be desirable to return an answer prior to the complete execution of the algorithm. An algorithm under the anytime framework facilitates this. For example, after the first phase, our algorithm has an approximate answer that we can examine. As the algorithm continues working in the background to confirm or adjust that

Algorithm 5 Batch Nearest Neighbor(A, B)

```
1:  PriorityQueue pq
2:
3:  min.dist ← +∞
4:  for each ts ∈ A do
5:      IndexFile ← B.ApproximateSearch(ts)
6:      dist ← IndexFileDist(ts,IndexFile)
7:      if min.dist > dist then
8:          min.dist ← dist
9:          min.ts ← ts
10:     end if
11:     entry.dist ← dist
12:     entry.ts ← ts
13:     pq.Add(entry)
14: end for
15:
16: pq.Remove(min.ts)
17: IndexFile ← B.ExactSearch(min.ts)
18: min.dist ← IndexFileDist(IndexFile)
19: min.IndexFile ← IndexFile
20:
21: while !pq.IsEmpty do
22:     IndexFile ← B.ExactSearch(pq.ExtractMin( ), min.dist)
23:     if IndexFile != null then
24:         min.dist ← IndexFileDist(IndexFile)
25:         min.IndexFile ← IndexFile
26:     end if
27: end while
28:
29: return min.IndexFile
```

answer, we can evaluate the current answer and make a determination of whether to terminate or allow the algorithm to persist.

In this particular experiment, the first phase of the algorithm returns an answer (which we later confirm to be the exact solution) in just 12.8 seconds, finding that the randomly chosen substring of chimp chromosome 2A, beginning at 7,582 of Contig NW_001231255 is a stunningly close match to the substring beginning at 999,645 of the Contig NT_005334.15 of human chromosome 2. The full algorithm terminates in 21.8 minutes. In contrast, a naïve sequential scan takes 13.54 hours.

2.5.7 Mapping the Rhesus Monkey Chromosomes

In our final experiment we demonstrate the utility of ultra-fast approximate search by conducting a large scale indexing experiment on a real world dataset. In particular we will attempt to discover, and then align the rhesus macaque (*Macaca mulatta*) chromosomes that are homologous to the human chromosome 2 we encountered in the last section. Note that while the chimpanzee and human diverged only 6 million years ago, the human and macaque diverged 25 million years ago. We should therefore not expect that the matches will be as strikingly similar as the matches shown in Figure 2.21. Instead, our approach will be to abandon any notion of *exact* searches, and conduct many *approximate* searches. We will then use the distributions of distances discovered for our approximate solutions to guide our hunt for homology, and to produce the alignment. As we noted before, we are not claiming *i*SAX has

a particular utility for such biological problems. It is merely a very large dataset for which we can obtain ground truth by other methods [42][70].

To begin, we would like to determine some baseline which identifies the level of similarity we should expect between two chromosomes which are *not* related. This could be done analytically, or with experiments on synthetic DNA. As we happen to known from external sources that the macaque chromosome 19 is unrelated to our target human chromosome 2, we will use that.

We converted human chromosome 2 to time series in the manner described in Section 5.6 and down sampled the time series by 4. Non-zero subsequences of length 1024 were extracted using a sliding window and indexed. From a total of 242,951,149 base pairs, 59,444,792 time series subsequences were indexed.

We then converted the macaque chromosome 19 to DNA time series using the same process and used each subsequence of 1024 as a query. Because DNA subsequences may have become been inverted some time in the last 25 million years (ie ..TTGCAT.. becomes ..TCAGTT..) we search for each time series, *and* its mirror image. In Figure 2.22 we show the distribution of the Euclidean distance values from subsequences in macaque chromosome 19 and their approximate nearest neighbor in Human chromosome 2.

This distribution tells us that the average distance between nearest neighbors is approximately 1,774 with a standard deviation of 301. There are very few distances less than 1,000 or greater than 4,000. If we repeat the experiment with randomly generated data or other non related DNA sequences we find nearly identical distributions is every case. We therefore can

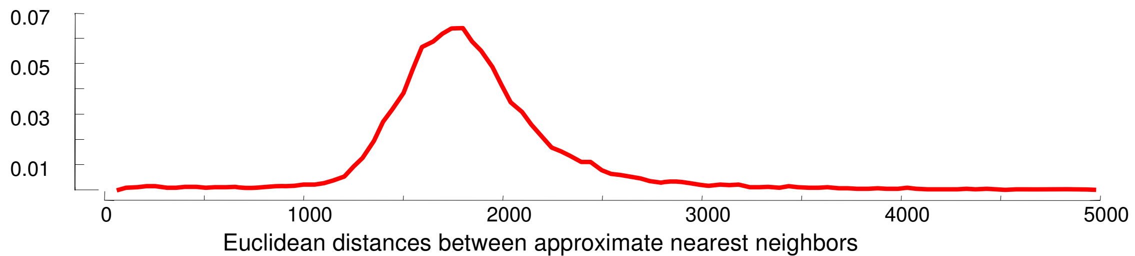

Figure 2.22: The distribution of the Euclidean distances from subsequences in Rhesus Monkey chromosome 19 to their approximate nearest neighbor in Human chromosome 2. The distribution is normalized such that the area under the curve is one

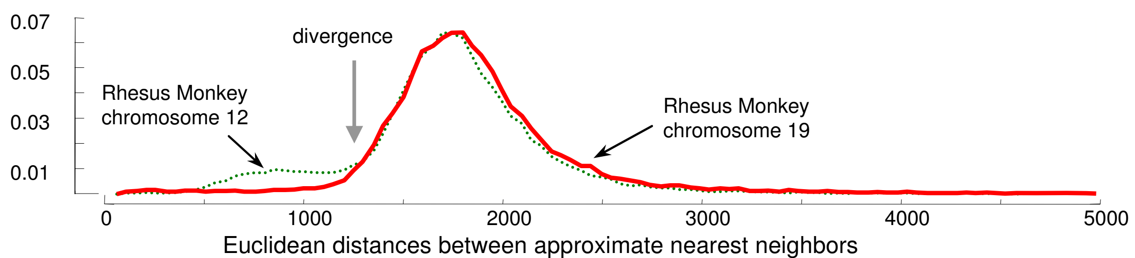

Figure 2.23: The distribution of the Euclidean distances from subsequences in Rhesus Monkey chromosomes 19 and 12, to their approximate nearest neighbor in Human chromosome 2

use this distribution as a baseline for a systematic search through the remaining 19 macaque chromosomes. While we could use a statistical test such as the Kullback-Leibler divergence [31], we simply visually inspected the distributions. Two of the monkey chromosomes, 12 and 13, produce significantly different distributions. In Figure 2.23 we show a comparison of the distributions for macaque chromosome 19 and 12.

Because both chromosome 12 and 13 from the macaque have a suspicious divergence from the expected distribution, we can create a dot plot to see which sequences in the monkey, map closely to which sequences in the human. We need to set some threshold, because we are not interested in knowing where the nearest neighbor to a subsequence is, if that nearest neighbor happens to be relatively far away. We observe from Figure 2.23 that the two

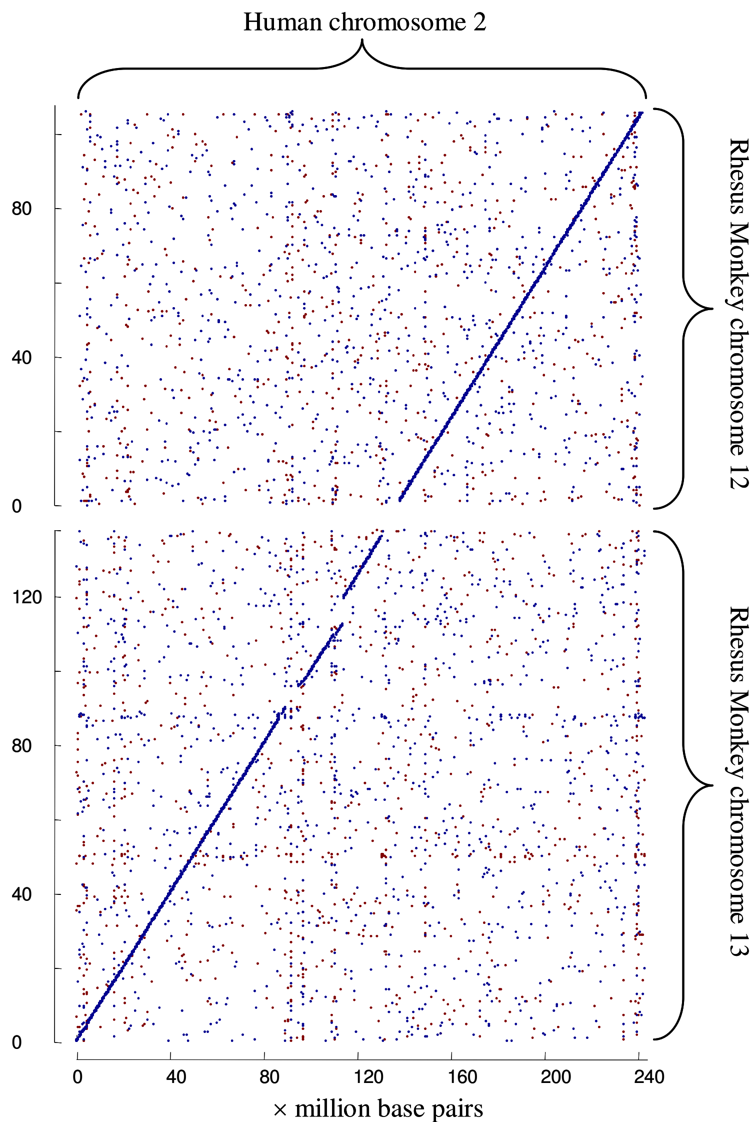

Figure 2.24: A dot plot showing the alignment of Human chromosome 2 with both chromosome 12 and 13 of the Rhesus Monkey. Each dot represents a location where a subsequence in the monkey (row) is less than 1,250 from a subsequence in a human (column)

distributions start to diverge (reading right to left) at about 1,250, so we use that value as the cutoff distance for dot plot construction.

This figure suggests that essentially all of human chromosome 2 can be explained by a fusion of Rhesus Monkey chromosome 12 and 13 (or vice versa). Of course, it has been

suspected that that human chromosome 2 is a recent species-specific fusion of two ancestral primate chromosomes for several decades [42]. More recent studies [70] have confirmed that the mapping above is correct, and the section that the rhesus macaque 12 and 13 maps to are called 2q and 2p, respectively.

In total, we performed 119,400 approximate queries taking slightly little over 4 hours, whereas a naive method of scanning subsequences of one chromosome across the other would result in nearly 119,400 * 59,444,792 distance computations, requiring well over a year, a duration which is clearly unacceptable for such applications.

2.6 Concluding Remarks

In this chapter, we have introduced *i*SAX, a representation that supports indexing of massive datasets, and have shown it can index up to one hundred million time series. We have also provided examples of algorithms that use a combination of approximate and exact search to ultimately produce exact results on massive datasets. Other time series data mining algorithms such as motif discovery, density estimation, anomaly discovery, joins, and clustering can similarly take advantage of combining both types of search, especially when the data is disk resident.

Chapter 3

Towards Indexing and Mining One Billion Time Series

3.1 Introduction

In this chapter, we investigate extensions to the techniques presented in Chapter 2 [77][78] in order to improve index construction time, a property identified as a bottleneck which limits the size of datasets that can be indexed. Note that the material presented here is joint work with the authors in [18]. We outline the theoretical contributions of [18] in Section 3.2 and Section 3.3, detail the improvements in scalability with experimental evaluation, and demonstrate the diverse applicability of our approach by conducting data mining exercises on entomology, DNA, and web-scale image collections.

3.2 Bulk Loading Massive Time Series Datasets

We recognized early on that the lack of a provision for facilitating the fast indexing of large existing datasets may be a limitation. This concern was later confirmed through our discussions with those in industry interested in utilizing *i*SAX. In this section, we address this limitation by outlining a new methodology for bulk loading time series. Notably, this algorithm improves the scalability of index construction by reducing the total number of disk page accesses as well as the number of random disk page accesses.

Prior implementations [77] of time series indexing (utilizing the *i*SAX representation), had constructed indexes via iterative insertion. While this is a useful mechanism to periodically update an index, it is inefficient for initial construction over a large dataset. The bottleneck mentioned previously, results from the observation that in iterative insertion, each time series is inserted in sequence and is potentially flushed to disk prior to the insertion of the next series. Given that no assumption is made regarding a spatial similarity ordering of time series (which may guarantee a clustering of entries and favorable disk locality), one can see that repeated iterative insertion will likely result in high random I/O access.

The approach introduced in [18] is to reduce the number of disk I/O operations during index construction by materializing distinct sub-trees of the index, one at a time, instead of building the index entirely at once. This change in construction methodology minimizes the number of split operations and can achieve streamlining of disk accesses so that the fraction of disk accesses which are sequential is significantly increased.

The algorithm used in [18] is a bulk loading technique which utilizes two buffers which reside in main memory: *First Buffer Layer* (FBL) and *Leaf Buffer Layer* (LBL). The FBL clusters time series which will end up at the same sub-tree and the LBL buffers time series which correspond to the same terminal node and flushes them. Essentially, the algorithm is a repetition of two phases of execution. During the initial phase, time series are read and inserted into the FBL. At the next phase, the time series are pushed down to the LBL (each sub-tree sequentially) and the internal structure of the index materialized. When a sub-tree in FBL has been pushed down to LBL, those corresponding LBL entries can be flushed to disk. This algorithm then repeats until completion of index construction. The authors refer to an index constructed in this manner as an i^2SAX index. Readers can refer to [18] for a more in-depth technical description.

3.3 Updating Index Node Splitting Policy

Another improvement implemented in [18] is an update to the round robin splitting policy originally employed in [77]. Recall that the splitting policy is used when a terminal node exceeds the user specified threshold and defines the dimension for which the cardinality for evaluation will increase. Though round robin is a simple yet effective policy for many datasets, occasionally we will find splits where round robin results in a poor skew of time series. This causes poor node occupancy and will increase the overall structure and size of

the index. To mitigate this, we formulated a new node splitting policy where a more balanced distribution of time series is obtained.

In order to obtain the optimal split for a given node, we must examine each dimension and all possible cardinalities for each time series in the node. This is a cost which is prohibitively expensive. A low cost heuristic is to examine for each dimension, the distribution of symbols at the highest cardinality. Then, select the dimension which will result in a more probable even distribution as the dimension to split. The cost for this new split policy is marginal as all necessary statistics can be kept at insertion. See Algorithm 6 for the new node splitting policy from [18].

Algorithm 6 New Node Splitting Policy( )

```
 1: mean[ ] ← this.GetDimensionMeans( )
 2: std [ ] ← this.GetDimensionStds( )
 3: dimToSplit ← Initalize( )
 4:
 5: for each d in dimensions do
 6:         b ← GetIncreasedCardinalityBreakPoint(d)
 7:         if b within mean[d]±3 std[d] then
 8:                 if  mean[d] closer to b than dimToSplit then
 9:                         dimToSplit ← d
10:                 end if
11:         end if
12: end for
13: IncreaseCardinality(dimToSplit)
```

The idea behind this algorithm is to use a combination of the mean and standard deviation to estimate the distribution of breakpoints (at the highest cardinality for each dimension). Then identify dimensions with entries most likely to be split at the next increase in cardinality, favoring dimensions with means closer to the breakpoint (for a more even spread).

3.4 Experimental Evaluation

Experimental evaluation was conducted on an Intel Xeon E5504 with 24GB of main memory, 2TB Seagate Barracuda LP (5900 RPM) hard disks, and running Windows Vista Business SP2. For an experimental case study, we also used an AMD Athlon 64 X2 5600+ with 3GB of memory, 400 GB Seagate Barracuda 7200.10 hard disk, and running Windows XP SP2 (with /3GB switch). Experiments with the latter setup are explicitly noted. All code is in C# and targets the .NET 3.5 Framework.

Our experiments are divided into the following sections: First we examine index construction properties such as disk accesses, wall time, and index size. Next, we consider three data mining problems from a diverse set of domains.

3.4.1 Index Scalability

To ascertain the scalability of the i^2SAX index, we evaluate the effect of the node splitting and bulk loading algorithms on index construction. Evaluation was conducted on random walk datasets from 1 to 100 million time series of length 256. Results were reported over an average of 10 runs and results were consistent across varying th levels. We compare the construction of the i^2SAX index with the original iSAX index and note the following [18]:

- Index size (in terms of number of nodes) on average was reduced by 34%.
- Index construction time (wall clock) on average was reduced by 30%.
- Terminal node occupancy on average was increased by 54%.

- Minimized the number of random I/O accesses: 99.5% of disk accesses were sequential.
- As an exercise in scalability, we were able to index **one billion** time series in just slightly over two weeks.

3.4.2 A Case Study in Entomology

As bulk loading allows us to easily scale to datasets much larger than those considered in Chapter 2, we will revisit our previously discussed case study regarding pest insects.

Many insects such as aphids, thrips and leafhoppers feed on plants by puncturing their membranes and sucking up the contents. This behavior can spread disease from plant to plant causing discoloration, deformities, and reduced marketability of the crop. It is difficult to overstate the damage these insects can do. For example, just one of the many hundreds of species of Cicadellidae (Commonly known as Sharpshooters or Leafhoppers), *Homalodisca coagulate* first appeared in California around 1993, and has since done several billions of dollars of damage and now threatens California's $34 billion dollar grape industry [4]. In order to understand and ultimately *control* these harmful behaviors, entomologists glue a thin wire to the insect's back, and then measure fluctuations in voltage level to create an Electrical Penetration Graph (EPG). Figure 3.1 shows the basic setup.

This simple apparatus has allowed entomologists to make significant progress on the problem. As USDA scientist Dr. Elaine Backus recently noted, "*Much of what is known today about hemipteran feeding biology .. has been learned via use of EPG technology*" [9].

However, in spite of the current successes, there is a bottleneck in progress due to the huge volumes of data produced. For example, a single experiment can last up to 24 hours. At 100 Hz that will produce a time series with approximately eight-million data points. Entomologists frequently need to search massive archives for known patterns to confirm/refute hypotheses. For example, a recent paper asks if the chemical thiamethoxam causes a reduction in xylem feeding behavior by a Bird Cherry-Oat Aphid (*Rhopalosiphum padi*). The obvious way to test such a hypothesis is to collect EPG data of both a treatment group and a control group and search for occurrences of the (well known) xylem feeding pattern.

Recently, the Entomology Department at the University of California, Riverside asked us to create an efficient tool for mining massive EPG collections. We have used the techniques introduced in this work as a beta version of such a tool, which will eventually be made freely available to the entomological community. Let us consider a typical scenario in which the tool may be used. In Figure 3.2 we see a copy of Fig. 2 from [51]. This time series shows a behavior observed in a Western Flower Thrip (*Frankliniella occidentalis*), an insect which is a vector for more than 20 plant diseases. The Beet Leafhopper (*Circulifer tenellus*) is not particularly closely related to thrips, but it also feeds on plants by puncturing their membranes and sucking sap. Does the Beet Leafhopper exhibit similar behavior?

To answer this question we indexed 20,005,622 subsequences of length 176 from the Beet Leafhopper EPG data, which had been collected in 60 individual experiments conducted from 2007 to 2009. We used a th size of 2000 and w of 8 to construct an index on our AMD machine. Even with fewer resources, it took only 6.25 hours to build the index, which occupied

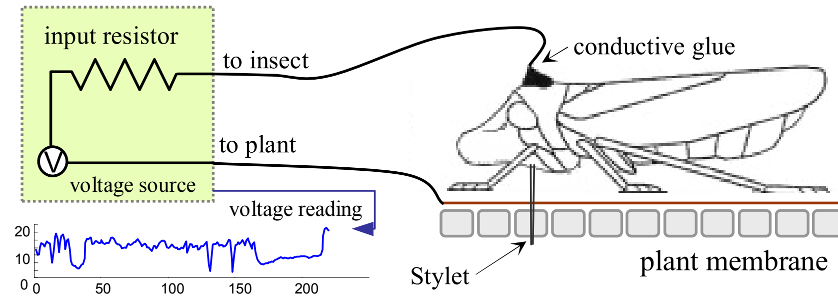

Figure 3.1: A schematic diagram showing an EPG apparatus used to record insect behavior

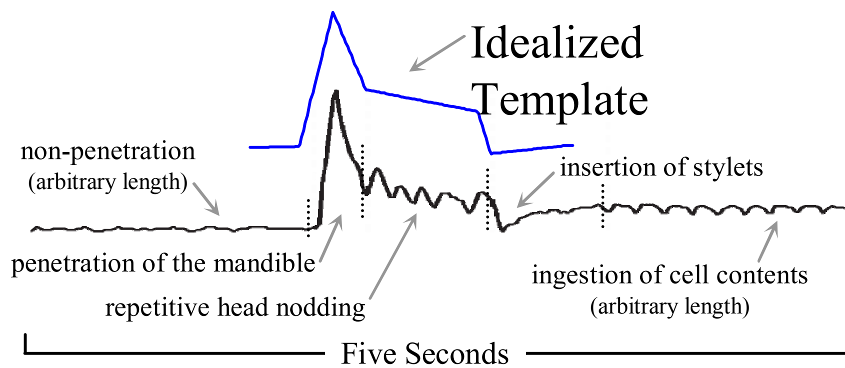

Figure 3.2: An EPG insect behavior derived from a subset of Fig. 2 from [51]. An idealized version of the observed behavior created by us is shown with a bold blue line

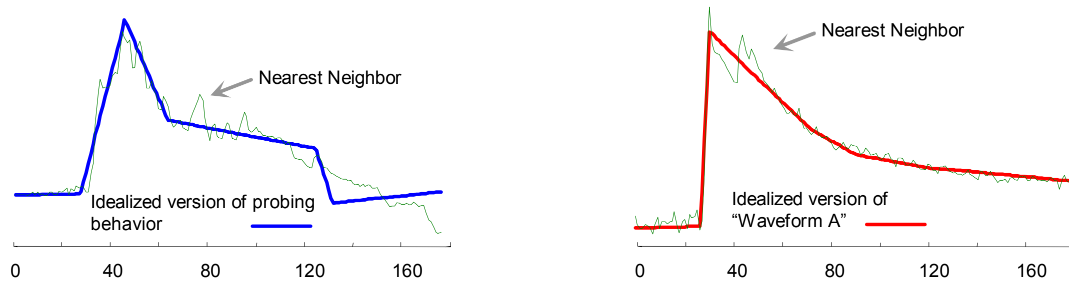

Figure 3.3: Query time series and its approximate nearest neighbor

a total of 26.6 gigabyte on disk space. As shown in Figure 3.2, we used the simple idealized version as a query to our database. Figure 3.3(*left*) shows the result of an approximate search, which takes less than 0.5 seconds to answer.

This result suggests that although the insect species is different (recall we queried a Thrip behavior on Beet Leafhopper database) the behaviors are similar, differing only in the insertion of stylet behavior. As a sanity check we also queried the database with an idealized version of a Beet Leafhopper behavior, the so-called "Waveform A", in this case, Figure 3.3(*right*) shows that the match is much closer.

3.4.3 Mining Massive DNA Sequences

The DNA of the Rhesus Macaque (*Macaca mulatta*) genome consists of nearly 3 billion base pairs (approximately 550,000 pages of text if written out in the format of this paper), beginning with TAACCCTAACCCTAA... We converted this sequence into a time series using the simple algorithm shown in Algorithm 4.

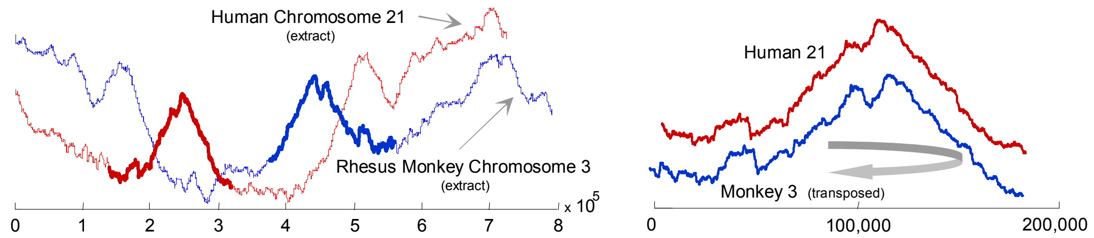

Figure 3.4: An example of DNA converted into time series

Figure 3.4 shows an example of the time series created from the DNA of monkey chromosome 3, together with the human chromosome 21. Note that they are not globally similar, but a *subsection* of each is locally similar if we flip the direction of one sequence. This figure suggests what is already known. Since the most recent common ancestor of the macaque and humans lived *only* about 25 million years ago, we can expect their DNA to be relatively similar. However, since humans have twenty-three chromosomes and the monkey has only twenty-one, the mapping of chromosomes cannot be one-to-one; some chromosomes must be mapped in a jigsaw fashion. But what is the mapping?

To answer this question, we indexed the entire time series corresponding to the macaque DNA (non-sex related). We used a subsequence length of 16,000, down-sampled by a factor of 25 to mitigate "noise". We then used a sliding window with a step size of 5 to extract a total of 21,612,319 subsequences. To index, we used a th size of 1000 and w of 10. In total, it took 9 hours to build the index.

We obtained queries from the human genome in the same manner and queried with both the original and transposed versions. For each human chromosome, we issued an average of 674 approximate searches (recall that chromosomes have differing lengths) and recorded the

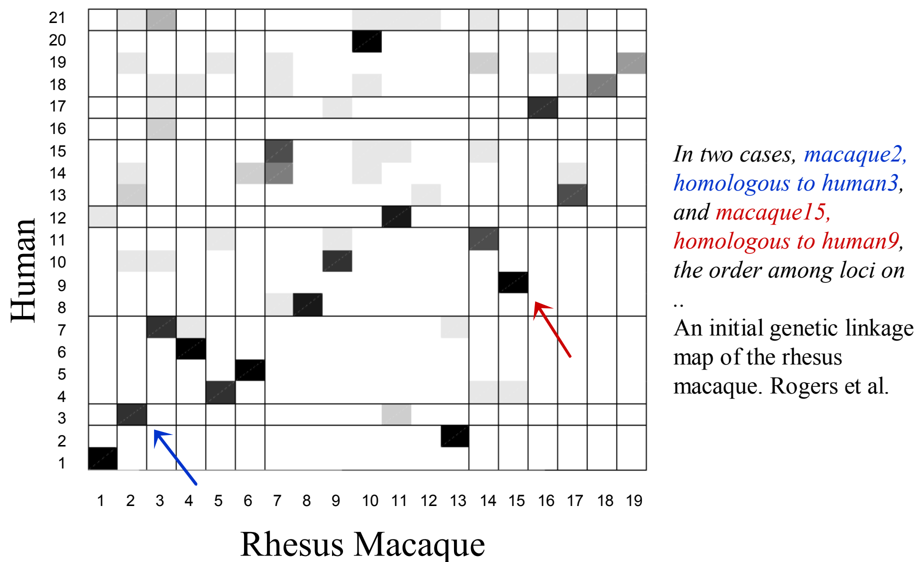

Figure 3.5: The cells represent potential mappings between the Macaque and Human Genomes. The darker the cell, the more often the nearest neighbor of a time series taken from a particular human chromosome had a nearest neighbor from a particular Macaque chromosome (the smallest chromosomes including the sex chromosomes are omitted)

ten nearest neighbors. In Figure 3.5 we summarize where the top ten neighbors are found, by creating a grid and coloring the cell with an appropriate shade of gray. For example, a pure white cell at location $\{i, j\}$ means that no query from human chromosome^{i} mapped to monkey chromosome^{j} and a pure black cell at location $\{i, j\}$ means that all ten queries from human chromosome^{i} mapped to monkey chromosome^{j}. This figure has some unambiguously dark cells, telling us for example that Human 2 is homologous ("equivalent") to Macaque 3. In addition, in some cases the cells in the figure suggest that two human chromosomes may match to a single Macaque chromosome. For example, in the column corresponding to Macaque 7, the two darkest cells are rows 14 and 15. The first paper to publish a genetic linkage map of the two primates tells us "*macaque7 is homologous to human14 and*

human15" [70]. More generally, this correspondence matrix is at least 95% in agreement with the current agreement on homology between these two primates [70]. This experiment demonstrates that we can easily index tens of millions of subsequences in less than a day, answer 13,480 queries in 2.5 hours, and produce objectively correct results.

3.4.4 Mining Massive Image Collections

While there are hundreds of possible distance measures proposed for images, a recent paper has shown that simple Euclidean distance between color histograms is very effective if the training dataset is very large [81]. More generally, there is an increasing understanding that having lots of data without a model can often beat smaller datasets, even if they are accompanied by a sophisticated model [35][5]. Indeed, Peter Norvig, Google's research director, recently noted that "*All models are wrong, and increasingly you can succeed without them*". The ideas introduced in this work offer us a chance to test this theory.

We indexed the color histograms of the famous MIT collection of 80 million low-resolution images [81]. As shown in Figure 3.6, these color histograms can be considered pseudo "time series". At indexing time we omitted very simple images (e.g. those that are comprised of only one or two colors, etc.). In total, our index contains the color histograms of 69,161,598 images.

We made color histograms of length 256, and used a th size of 2000 and w of 8. It took 12.3 hours to build the index, which is inconsequential compared to the nine months of twenty-four hours a day crawling it took to collect it [81]. The data occupies a total of

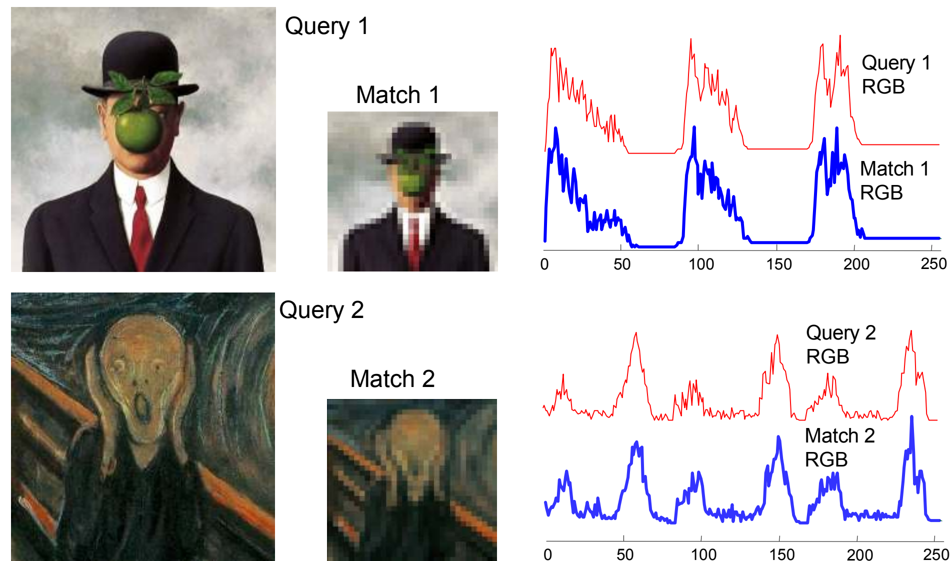

Figure 3.6: *top left*) A detail of *The Son of Man* by René Magritte, which we used as a query to our index. *top center*) The best match returned. *top right*) The similarity of the two images in RGB histogram space. *bottom left*) A detail of *The Scream* by Edvard Munch, which we used as a query. *bottom center*) The best match returned. *bottom right*) The similarity seen in RGB space

133 gigabytes of disk space. The latter figure only includes the space for the time series, the images themselves required an extra of 227 gigabytes.

Does this random sampling of the webs images contain examples of iconic art images? To test this, we found examples of two famous images using Google image search and converted the image to color histograms of length 256. We then used these to search our collection with an approximate search. Each search took less than a second, and the results can be seen in Figure 3.6. Note that we are not claiming that Euclidean distance between color histograms is the best measure for image similarity. This experiment simply demonstrates the scalability and generality of our ideas, as a side effect of demonstrating the *unreasonable effectiveness of* (massive amounts of) *data* [35].

3.5 Concluding Remarks

In this chapter we have introduced joint work with the authors in [18]. We describe i^2SAX, an index structure specifically designed for ultra-large collections of time series, and propose new mechanisms and algorithms for efficient bulk loading and node splitting. We experimentally validate the proposed algorithms, expanding upon the scale of experiments first conducted in Chapter 2; including the first published experiments to consider datasets of sizes up to one billion objects. We have also demonstrated data mining applications on a wide range of diverse datasets, from web-scale images to insect data, further showing the general applicability of our work.

Chapter 4

Anytime Nearest Neighbor Classification

4.1 Introduction

The techniques presented in Chapter 2 and Chapter 3 can be used to expedite similarity search in large static datasets. The results of similarity search can then be used in data mining applications such as nearest neighbor classification. In this chapter, we continue to examine data mining techniques for classification, though not in the context of large static datasets. We now consider the scenario where available computation time may be highly variable, such as exhibited in data streams.

Classification of data arriving from a data stream is often more difficult than the batch situation because the algorithm must operate in a time sensitive and computationally constrained environment. Traditional algorithms are often unable to provide satisfactory performance while supporting the highly variable arrival rates that typify such applications. For

example, a single data stream may produce items to be classified at a rate that can range from milliseconds to minutes [83]. Traditional classification algorithms typically lack the mechanism for providing an intermediate result prior to completion, and contract-based algorithms require the time duration prior to execution [92]. In such contexts, *anytime algorithms* have been found to be exceptionally useful, and have recently been the subject of extensive research efforts [27][40][50][56][52][83][88].

Anytime algorithms are algorithms which are amenable to variable response times, by exchanging the quality of response as a function of time [32][92]. In the case of classification, *quality* is measured by the probability of correct classification. More concretely, an anytime algorithm, after a short period of initialization, can always be interrupted to return some intermediate result. This flexibility in response time allows anytime algorithms to be used with great success in real-world environments with variable constraints [27][41][74].

For anytime classification, one well established technique is the anytime nearest neighbor classification algorithm [83]. This algorithm retains the strong points of the nearest neighbor algorithm, its simplicity and generality [1], while greatly mitigating the problem associated with the linear time complexity at classification time, a function of its "lazy" behavior.

Previous techniques for improving anytime classification have generally been concerned with optimizing the probability of correctly classifying *individual objects*. In this chapter, we show that substantial improvement in overall classification accuracy performance can be

[1] i.e. the ability to use any distance measure, the ability weight features, etc.

achieved if the optimization is performed relative not to each *individual object*, but rather to a (possibly quite small) *set* of objects [79].

Our technique is a generalized framework which utilizes a scoring function that estimates the intermediate result quality of an object being processed. Here, the *quality* is an estimate that we have the correct class label for the object. Objects with a high initial quality are unlikely to significantly improve their quality, even with additional computation time. In contrast, objects with poor initial quality have much greater room for improvement, and are deserving of more resources. Using this intuition, our framework intelligently and dynamically schedules computational resources for each object. We show that the lack of such inter-object consideration would otherwise result in poor allocation of computation time and lead to reduced performance. As expressed by the well-known idiom, we would be *polishing the wrong apple* if we allocated resources to an instance whose class label is unlikely to change, even with more resources.

Our methodology is invariant to object arrival behavior, and perhaps unintuitively, it is notable in that even with a uniform object arrival rate we are capable of attaining a marked improvement in classification performance. This is in contrast to the usual motivation for anytime algorithms, which are typically presented to mitigate the effects of variable object arrival behavior [40][50][83][88].

The remaining sections of this chapter are organized as follows: In Section 4.2, we provide background on anytime algorithms and review related work on classification techniques. We then present an overview of the anytime nearest neighbor classifier in Section 4.3. Sec-

tion 4.4 motivates and introduces techniques for improving classification accuracy by using a scoring function to measure intermediate result quality and performing computational resource allocation. Section 4.5 provides additional details regarding the selection and formulation of a scoring function. In Section 4.6 we verify the utility of our framework with experimental evaluation conducted on a wide range of diverse datasets. Lastly, Section 4.7 offers some discussion and provides a conclusion to our work.

4.2 Background and Related Work

Algorithms are considered *anytime* if they exhibit specific characteristics [32][92], notably:

- After a short period of initialization, the algorithm becomes *interruptible*. That is, an intermediate result can be returned at any time up to completion.
- The quality of this result is measurable and improves with additional computation time.
- The change in quality is typically characterized by diminishing returns, with the largest gains found in the initial stages of computation.
- An interrupted execution of the algorithm can also be resumed for additional refinement without significant overhead.

Figure 4.1 illustrates the prototypical tradeoff between result quality and computation time in an anytime algorithm. Such flexibility is advantageous when available computation time is not known a priori (e.g. in data streams).

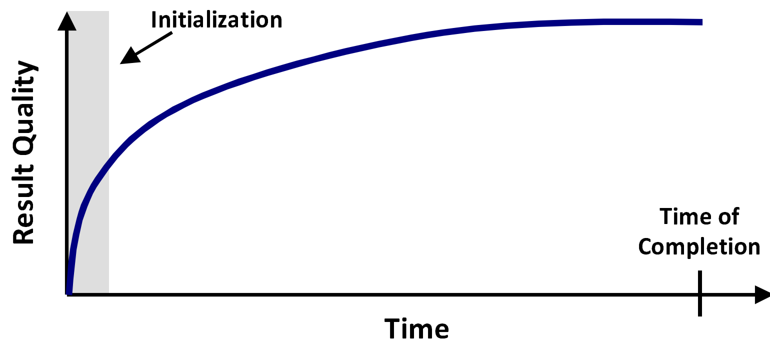

Figure 4.1: Anytime algorithms are interruptible after initialization. This plot shows the increase in result quality with additional computation time

Due to their utility in real-world settings, anytime algorithms have been extensively studied [37] and have found application in a number of diverse domains. Applications range from path planning in real time strategy games [15] to the clustering of time series [56]. Significant work has also been done to adapt or view [71] well established machine learning algorithms under the anytime framework. Examples include Bayesian networks [41], decision trees [33], nearest neighbors [83], and inductive logic programming [57].

For anytime nearest neighbor classification of objects, performance has been shown to be improved by reordering the training set. One technique for generating such an ordering is by repeatedly moving the worst exemplar to the end of the list so that the most characteristic exemplars are examined first [83] [85].

When classifying objects from a data stream it may not be necessary or advantageous to compute classifications serially, where the available computation time for each object is the interarrival time between itself and the next successive object. A more general methodology is to separate the direct relationship between arrival rate and object computation time. In-

stead, the available computation time is dictated by some external variable. This approach is less constrained and allows for the concurrent processing of more than one object. For example, in real-world monitoring scenarios, a typical query may be: "*Monitor object-stream X and event-stream Y* (let λ_X, λ_Y be the arrival rates for X and Y, respectively, where $\lambda_X >> \lambda_Y$)*, classify objects arriving in X, and return their classification upon the next event in Y.*"

Note that if the object processing strategy is serial, then simply optimizing the classification of each object individually is clearly the optimal policy. However, if objects are processed concurrently, then it is possible that we can do much better than simply optimizing each object classification in isolation [40][52]. This follows from the observation that in virtually any set of objects, the change in result quality with additional computation time will likely vary greatly between each object. To obtain the greatest increase in performance, we simply need a way to estimate and then process the object(s) that can benefit the most from additional computation. While this appears to be a "chicken-and-egg" paradox (since during classification, we obviously don't know if we have the true class label), as we shall see, at least in the case of the nearest neighbor algorithm, that we can cheaply obtain an approximation of how likely we have the correct label, a *confidence* measure.

In this work, we apply this intuition to present a general framework which can be used to increase the overall performance in data stream classification. Our method uses a "confidence" scoring function to estimate the quality of intermediate results and assigns computation time to objects with the lowest classification confidence. We demonstrate in our

experimental evaluation that once we have a reasonably accurate scoring function, we can improve overall classification performance. Note that classification remains under anytime conditions and that the end times for objects are never known a priori.

We have chosen to explore our observations with the nearest neighbor algorithm. The nearest neighbor algorithm has proven to be one of the most frequently deployed classifiers, with accuracy which is competitive with other techniques [83]. The nearest neighbor algorithm has the advantage of being non-parametric, capable of handling a large number of classes, and easily adaptable to datasets which are dynamically changed or updated without retraining or overfitting. Its primary disadvantage is the linear time evaluation of training exemplars, a property which can be mitigated by indexing the data (when applicable), or anytime algorithms such as the one presented in [83]. While the work in [83] allows the algorithm to be used in streams where one object may arrive before the current object has completed processing, it does not leverage the idea of varying computational resources according to result quality when classifying a set of concurrent objects.

4.3 Anytime Nearest Neighbor Classification

The nearest neighbor algorithm, well known for its utility and range of applicability, is easily extended to fit under the anytime framework. This section presents a review of the anytime nearest neighbor classification (ANNC) algorithm [83]. In Section 4.4, we will discuss how to generalize ANNC for concurrent classification.

Table 4.1: List of Notation

Name		Description
q		An object to classify. Contains the following fields:
	$q.pos$	Current training set position
	$q.class$	Current classification label
	$q.dist$	Current nearest neighbor distance
	$q.stopped$	Flag denoting user stoppage
Q		A set of objects to classify
Q_i		$Q_i \in Q$. Equivalent to q above
Q'		A set of objects to classify, which are concurrent and in memory
D		The set of training objects
D_i		$D_i \in D$
	$D_i.class$	The class label of this object
M		Number of objects which can be buffered in memory
$NumClasses(D)$		Returns the number of unique class labels represented within training set D
$ScoreFcn(q)$		A function that returns a score which estimates the intermediate result quality of object q
$Distance(,)$		A context appropriate distance measure

For clarity, the notation used in the following sections is first presented in Table 4.1.

Given an object to classify, q, and a set of training instances, D, the ANNC algorithm finds the entry D_i in D which minimizes $Distance(q, D_j)$. That is:

$$\forall D_j \in D, Distance(q, D_i) \leq Distance(q, D_j)$$

The returned classification is the corresponding class label for D_i, $D_i.class$.

Note that we have not explicitly defined the *Distance* measure used in ANNC. The ANNC can use any distance measure (L_p-Norm, Hamming distance, graph edit distance, Dynamic Time Warping, etc.) that is appropriate to a specific context (time series, strings, graphs,

categorical data, etc.). If only a similarity measure is available, i.e. the cosine *similarity* measure, we can simply define the distance as the reciprocal of the similarity measure.

A sketch of the ANNC algorithm is shown in Algorithm 7 .

Lines 1-10 initialize q to an initial result. First, let $NumClasses(D)$ be the number of unique class labels in D, where the first $NumClasses(D)$ instances of D contain exactly one exemplar from each class label. Then, the initialization period does the following: q is iterated over the first exemplar from each class and an initial classification is obtained and saved as the intermediate result. While the algorithm cannot be interrupted during this period, it must only iterate $NumClasses(D)$ times, where $NumClasses(D) \ll |D|$ and thus the time duration spent in initialization is marginal.

Following initialization, the object being classified can be stopped, then resumed at any time leading up to the completion of training set evaluation. Lines 11-18 iterate through the remaining training instances in D or until stopped and update the nearest neighbor for q accordingly. $q.class$ is then returned as the classification result for q.

The generic algorithm just presented falls under the anytime framework and achieves increased quality of results (classification confidence) as a function of additional time.

This, in essence, is the current methodology for ANNC, introduced in [83], except that work also suggests optimizing individual object classification by identifying heuristics for ordering the training set entries so that the most characteristic exemplars are examined early on. This simple idea is useful enough to have found real-world applications; for example, it is used in a surveillance system created by Toshiba [82].

Algorithm 7 Anytime Nearest Neighbor Classifier(q, D)

```
1: q.dist ← +∞
2: q.class ← null
3: for i ← 1 to NumClasses(D) do
4:     Dist ← Distance(q, D_i)
5:     if Dist < q.dist then
6:         q.dist ← Dist
7:         q.class ← D_i.class
8:     end if
9: end for
10: q.pos ← NumClasses(D)
11: while !q.stopped and q.pos < |D| do
12:     q.pos ← q.pos + 1
13:     Dist ← Distance(q, D_{q.pos})
14:     if Dist < q.dist then
15:         q.dist ← Dist
16:         q.class ← D_{q.pos}.class
17:     end if
18: end while
19: return q.class
```

In this work, we adopt the complementary methodology of optimizing performance across a set of objects. In the following sections, the motivation and advantages behind such a method are presented and techniques for improving overall classification performance are introduced.

4.4 Concurrent Object Evaluation

For the purpose of explanation, it was convenient to illustrate anytime classification with regards to a single object. However, streaming data classification is more accurately exemplified by a *sequence* of objects. Given this consideration, our work examines the set of concurrently processing objects and optimizes the scheduling of computational resources to-

wards maximizing overall classification performance. Such evaluation is significant in that simply optimizing individual object classification can result in poor *overall* computational resource allocation and performance. For expository purposes, consider a simple example:

Suppose we have a set of objects to concurrently classify. After some period of initial processing, in all but the most pathological cases, we would expect that this set of objects will contain intermediate results which span some range in quality. As a simple example, suppose we have a database of ten million objects, consisting of two classes of automobiles, Japanese and American, and we have a pool of instances to classify: 1995 Toyota Corolla, 2000 Ford Escort, 1998 Honda Civic,..., 1957 Hudson Hornet [2] . Because the Toyota Corolla is the bestselling car in the world, once we have examined even the first 100 objects in the training database we are practically guaranteed to have seen several examples of Corollas. There is, therefore, little utility in comparing it to the rest of the database. In contrast, the Hornet is so rare, and so unlike other American cars, that it is very unlikely to have been encountered in the first 100 items visited. Its current nearest neighbor is as likely to be Japanese as American.

Scheduling policies which do not take into account the diversity in the set of currently processing objects will almost certainly result in suboptimal resource allocation by scheduling computation time to objects which already have excellent intermediate results (i.e. Toyota Corollas). This is time that can be better spent on objects which may benefit most from additional computation. However, we do not know the true class labels of the objects, and

[2] A rare American-made automobile.

therefore how can we know the likelihood that the *tentative* label is correct? Our task resembles a well observed problem found in nature, and our solution is also very similar to the solution that has evolved in nature.

Many birds and small mammals have large broods, and the parents must allocate food resources among them [59]. From the point of view of the parent who is trying to maximize her (less often, "his") reproductive success, the optimal thing to do is to feed each offspring equally. However, birds cannot differentiate between their offspring, so they cannot use any algorithm that requires labeling their young. If their algorithm is to feed the most aggressive young (a literally *greedy* algorithm), then the next time they return with food, that aggressive offspring, fortified by recent feeding, will be able to force itself to the front again and beg for more food [61]. This will continue indefinitely until the weaker siblings starve [3].

The solution to this problem is that the young signal their hunger level to the parent by the frequency of chirping. The parent's optimal algorithm is reduced to feeding the young that signals the greatest need. This solution is nearly universal among birds, and may have evolved independently in different species [61].

We can see that our problem is nearly identical. We have resources (CPU time) that we must distribute among objects that have possibly varying levels of need (i.e. varying levels of confidence in their classification). As we shall show in the following section, our solution is nearly identical; we simply need to have each object "*signal*" its need, based on an estimate of how confident it feels about its current label. Of course, we can never know exactly the

[3]We know that this would happen, because it *does* happen in cases of brood-parasites such as the cuckoos.

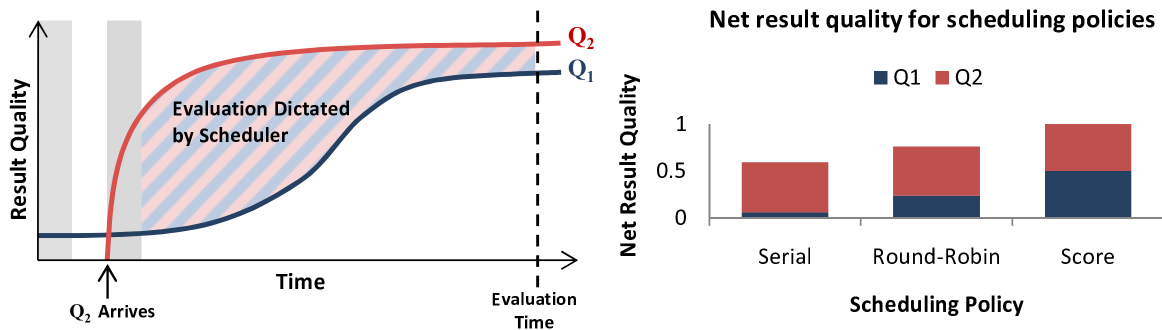

Figure 4.2: Net result quality across concurrent queries for various scheduling policies. *left*) Result quality over time for two queries Q_1 and Q_2 (note that Q_2 arrives shortly after Q_1). *right*) Net result quality (Q_1 and Q_2) for various scheduling policies at evaluation time (hatched line)

probability of an object having the correct classification; however, as we shall see in Section 4.6, a simple heuristic is sufficient to produce improvements in overall accuracy.

4.4.1 Scheduling Policies

If a data stream is sparse, then objects are processed in isolation and without interference from other objects (in effect, complete serial processing). In such scenarios, the best performance can be achieved by utilizing any of the previous approaches which optimize for singular objects. However, in most data streams we can expect to encounter at some point a set of objects which have overlapping lifetimes. Thus, given a data stream which contains a set of Q objects distributed over the duration of the stream, at any given time within the lifetime of the data stream, there exists some subset, Q' of Q, consisting of $|Q'|$ number of objects being classified, $|Q| \geq |Q'| \geq 0$. If the number of objects is neither zero nor singular ($|Q'| > 1$), and because only one object may be processed at a time, what is the scheduling approach that attains the best overall performance over Q'?

Recall that the end time of each object is never known a priori. Consequently, it is not possible to identify a global optimum, over Q, beforehand. However, we *can* optimize locally, over Q', to improve performance. Let us first introduce an illustrative example, consisting of two objects Q_1 and Q_2, $Q' = \{Q_1, Q_2\}$. Assume Q_2 arrives shortly after Q_1 and that the quality of result as a function of computation time for each of the objects is known (as shown in Figure 4.2 *left*).

One scheduling technique is the sequential or serial scheduling of objects. That is, we classify each object, Q_i, using the ANNC algorithm described in Section 4.3 and return a response when complete or upon the next arriving object. Under serial scheduling, Q_1 is stopped prematurely as a result of Q_2's arrival. From the result quality over time plot in Figure 4.2 *left* we can see that Q_2 obtains a high quality result immediately following initialization, whereas Q_1 requires additional computation time to reach a higher level in quality. Due to the sequential nature of this scheduling algorithm, a clear suboptimal result is obtained (see Serial Scheduling Policy in Figure 4.2 *right*. The bulk of computation time is devoted to Q_2, resulting in a marginal improvement in net quality. Notice that a much larger increase in overall quality could have been achieved by continuing to schedule Q_1 even after the arrival of Q_2.

Another scheduling policy is to divide computation time equally among concurrently processing objects in a round robin fashion. The result shown in Figure 4.2 *right* is an improvement over serial scheduling; however, the intuition established in the previous solution

is as follows: since Q_2 is able to obtain a high quality result quickly and Q_1's quality remains poor, computation time is still best spent on classifying Q_1.

Let us now see how we can achieve even better results than round robin scheduling. Our intuition dictates that we should refrain from wasting computational resources on an object when there is the existence of another object whose quality is lower. Observe that such a scheme can be achieved if we have a scoring function which returns an indicator of intermediate result quality per object. Then the scheduling policy is simply to schedule the object with the lowest score. For example, if we use the plots in Figure 4.2 *left* as our scoring mechanism and schedule the computation accordingly, the net result quality is shown in Figure 4.2 *right* and is clearly the best of the three presented techniques.

While it is impossible to have prior knowledge which gives the exact change in quality over time, the score scheduled technique remains highly effective if scoring functions which are a close estimator of result quality exists. We leave the discussion of scoring functions to a later section; first, the score scheduled algorithm is presented.

4.4.2 Batch Evaluation

To introduce the use of a scoring function, we first consider the slightly simplified task of classifying a set of objects in batch fashion. In the next section, extensions necessary for a streaming environment will be presented. The batch score scheduled algorithm is shown in Algorithm 10. Note that the segments of code devoted to the initialization and updating

of objects have been summarized in Algorithm 8 ($Initalize$) and Algorithm 9 ($Update$) respectively.

The batch algorithm begins by initializing each object in the input set (Lines 1-3). A priority queue containing the entire set of objects is then created (Line 4). The priority queue ordering is dictated by the scoring function, $ScoreFcn$ which provides an estimate of the quality of an object's intermediate result. At each iteration, the object with the lowest score (lowest classification confidence) is scheduled for computation and updated (Lines 5-11).

Computational time for initialization is $O(CNumclasses(D))$ per query, with C as the cost per $Distance(\, , \,)$ invocation. The priority queue is initialized in $O(S|Q|)$ time and the cost for each iteration of the score scheduled computation is at most $O(log|Q'|)$ to $RemoveMin(\,)$ and $O(C + S + log|Q'|)$ for the re-insertion/update, given that Q' is the number of currently processing queries, S is the cost per $ScoreFcn(\,)$ invocation, when utilizing a standard heap-based priority queue. We assume that objects can be asynchronously stopped and can be lazily removed from the priority queue or in the case with limited memory, purged at a cost of $O(log|Q'|)$.

Note that the number of scheduling iterations can be significantly reduced by simply tracking the confidence score of the next minimum item in the queue, T, and performing computations on the current object until stopped or its confidence score exceeds T. Another technique is to increase the resources allocated per iteration.

Algorithm 8 Initializing Object Classification (q, D)

1: $q.dist \leftarrow +\infty$
2: $q.class \leftarrow null$
3: **for** $i \leftarrow 1$ to $NumClasses(D)$ **do**
4: Dist $\leftarrow Distance(q, D_i)$
5: **if** Dist $< q.dist$ **then**
6: $q.dist \leftarrow$ Dist
7: $q.class \leftarrow D_i.class$
8: **end if**
9: **end for**
10: $q.pos \leftarrow NumClasses(D)$

Algorithm 9 Updating Object Classification (q, D)

1: **if** $!q.stopped$ **then**
2: $q.pos \leftarrow q.pos + 1$
3: Dist $\leftarrow Distance(q, D_{q.pos})$
4: **if** Dist $< q.dist$ **then**
5: $q.dist \leftarrow$ Dist
6: $q.class \leftarrow D_{q.pos}.class$
7: **end if**
8: **end if**

Algorithm 10 Batch Score Scheduled Classifier $(Q, D, ScoreFcn)$

1: **for** $i \leftarrow 1$ to $|Q|$ **do**
2: $Initalize(Q_i, D)$
3: **end for**
4: $PriorityQueueQueue(Q, ScoreFcn)$
5: **while** $Queue.Size > 0$ **do**
6: $q \leftarrow Queue.RemoveMin(\ )$
7: $Update(q, D)$
8: **if** $!q.stopped$ and $q.pos < |D|$ **then**
9: $Queue.Add(q)$
10: **end if**
11: **end while**

4.4.3 Streaming Evaluation

Extensions to the batch algorithm are necessary to make the score scheduled algorithm suitable for a streaming context. More concretely, we must incorporate the consideration of the online arrival of objects and a finite memory for buffering of concurrent objects. Our complete framework is presented in Algorithm 11.

In Line 1, we initialize an empty priority queue and set the ordering of objects to be dictated by the scoring function, $ScoreFcn$. Line 2 examines a user updateable flag to determine if this online algorithm should continue. The previous batch algorithm was able to simply check queue size for termination; however, an empty queue is not similarly indicative in an online, streaming environment (consider the case where bursts of queries are separated by long interarrival times, resulting in empty queues). The algorithm then checks to see if a new object has arrived and fetches it accordingly (Lines 3-4). Incoming objects are initialized immediately (Line 5) and inserted into the queue (Line 9). In the case that the current queue size plus the new object will exceed the maximum buffer size, M, the object with the largest score, is stopped and evicted (Lines 6-8). This is the in- memory object with the highest classification confidence. Lines 11-15 schedule the minimum scored object for processing and are identical to the batch algorithm. The computational time is the same as the batch algorithm, with the addition that the priority queue is empty at initialization and must also support the $RemoveMax(\,)$ operation, which has $O(log|Q'|)$ time complexity.

Algorithm 11 Streaming Score Scheduled Classifier $(D, ScoreFcn, M)$

1: $PriorityQueueQueue(ScoreFcn)$
2: **while** $ContinueClassification$ **do**
3:     **if** $NewObject$ **then**
4:         $q \leftarrow GetNewObject(\ )$
5:         $Initalize(q)$
6:         **if** $Queue.Size > M - 1$ **then**
7:             $Queue.RemoveMax(\ )$
8:         **end if**
9:         $Queue.Add(q)$
10:     **else**
11:         $q \leftarrow Queue.RemoveMin(\ )$
12:         $Update(q, D)$
13:         **if** $!q.stopped$ and $q.pos < |D|$ **then**
14:             $Queue.Add(q)$
15:         **end if**
16:     **end if**
17: **end while**

4.5 Scoring Function

Recall that the scoring function estimates intermediate result quality. In classification, a high score implies that the current classification label is unlikely to change, even if given sufficient time to run until completion. Conversely, a low score is characteristic of an object whose classification is likely to change.

Given that the nearest neighbor classifier seeks to find the entry in the training set with the minimum distance, one simple scoring method is to use the current best-so-far nearest neighbor distance as the estimate for classification confidence. The distance value is inverted

so that higher distances correspond to lower scores:

$$Score_{BsfDistance}(q) = -min(Distance(q, D_i))1 \leq i \leq q.pos$$
$$= -q.dist$$

Note that we are not claiming that this scoring method is the optimal one; our claim is merely that it is empirically successful for many datasets. The framework is agnostic to the scoring method, and a user with context specific knowledge can formulate and tune a custom scoring function accordingly.

For comparative purposes, we will use the round robin scheduling policy as a competitive baseline. As shown by the example illustrated in Figure 4.2, round robin can offer a significant improvement over serial scheduling. Furthermore, and in contrast to the $Score_{BsfDistance}$ method, round robin has the advantage of being starvation free. This property can mitigate the adverse effects of outliers and prevent otherwise a potential monopolization of computational resources.

4.6 Experimental Results

In this section, we examine the utility of our score scheduled anytime nearest neighbor classifier by conducting experimental evaluation of a wide range of diverse classification datasets [8]. A list of the datasets used in our experiments and their attributes are shown in Table 7.

Table 4.2: Datasets used for experimental evaluation

Name	Classes	Attributes	Instances
Two Pattern	4	128	5,000
AIBO Robot	2	100	12,100
Gun	2	150	200
Face	16	131	2,231
Leaf	6	150	442
CBF	3	128	1,000
Moth	35	Image (∼500x800)	772
JF	2	2	20,000
Letter	26	16	20,000
Pen Digits	10	16	10,992
Ionosphere	2	32	351

For all datasets, we obtained testing and training splits using 10-fold cross validation. The training exemplar order for each fold is randomly permuted and all features are used. Note that the training set invariant, where one exemplar from each class is encountered first, is preserved.

For the distance function we use the Euclidean distance.

While Euclidean distance may not be the optimal distance measure for every dataset, it has been shown to be competitive across many domains [26]. As our primary objective is simply to show the *improvement* as a result of using score scheduling to allocate computational resources, the selection of Euclidean distance as the distance measure is appropriate.

4.6.1 Classification of Streaming Data

To evaluate the utility of score scheduled classification, we simulate the classification of data streams with varying rates of arrival. Our experimental data stream exhibits the characteristic of constant or uniform arrival between successive objects, and the exact interarrival time

between each object is modeled as the number of training set exemplars ($|D|$) which can be evaluated:

$$InterArrivalTime(r) = \lfloor |D|r \rfloor \, 0.1 \leq r \leq 1$$

The arrival rate is modeled as a function of $|D|$ on an account of its generality across all datasets. This is in contrast to concrete numerical values (e.g. the data stream operates at $100Hz$) which may not always be applicable or meaningful (due to the wide variability in dataset characteristics: number of classes, feature space, exemplars available).

Objects from the testing set, Q, enter the data stream in accordance with the interarrival time until exhausted. For $r = 1$, the interarrival time between successive objects is exactly the time needed to evaluate the entire training set and thus is equivalent to complete serial classification. Our experiment concludes when mean interarrival time has elapsed following the arrival of the last object from the test set.

Classification accuracy is computed from the predicted test labels and the true test labels. Note that this experimental setup obtains classification accuracies which are dependent on the order of arrival from the testing set. To remove such bias, we average the classification accuracy for each testing/training split over 10 random permutations of the testing set. For this experiment, we assume that the cardinality of objects being concurrently classified, $|Q'|$, can be accommodated by the memory buffer ($M > |Q'|$).

As a general rule, we expect accuracy to decrease as the arrival rate increases. This behavior can be attributed to the reduced available computation time on average per object for

faster streams. However, we have occasionally observed higher accuracy with increased arrival rate, beyond an expected variability. This phenomenon is often caused by non-separable classes or the presence outliers/noisy data and resulting in a scenario where intermediate results have the correct classification but the final nearest neighbor is of a different class.

The classification accuracies from our score scheduled approach on a variety of datasets are shown in Figure 4.3. From the results, we see that we are typically able to obtain an increase in accuracy over the round robin baseline. This confirms our intuition that $Score_{BsfDistance}$ is a good indicator of result quality by allocating additional computational resources to objects which need it more.

Overall, round robin is a fairly competitive baseline. This can be expected, as prior work [83] has shown that many datasets have query objects which follow the prototypical result quality over time behavior depicted in Figure 4.1. That is, even evaluating just a small portion of the training set can obtain a high quality result quickly, with additional evaluation characterized by diminishing returns.

4.6.2 Effects of Constrained Memory on Classification Accuracy

Performance degradation can occur when objects are stopped prematurely and evicted from the buffer as a result of memory constraints. As the score for each object is an estimate of how confident we are about its class label, it is a principled way of determining the eviction policy when encountering memory constraints. That is, we simply evict the object with the highest confidence score. For the round robin baseline, a randomly chosen object is evicted.

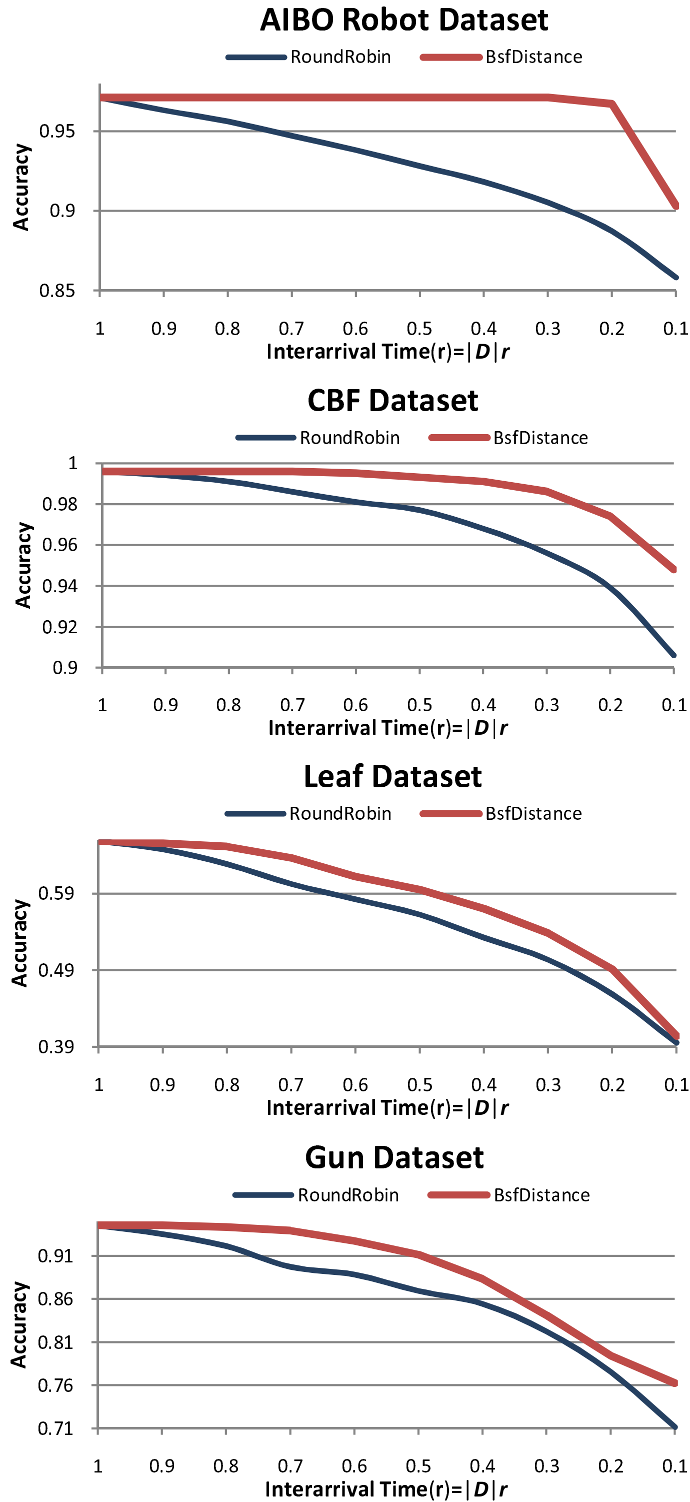

Figure 4.3: Classification accuracy of score scheduled anytime classifier on constant data streams with varying rates of arrival

We conducted experiments which varied the available memory buffer size, M, from $|Q|$ to $0.05 * |Q|$. From the four datasets we evaluated for constant streams, the change in accuracy was negligible. For each dataset the net change in accuracy per arrival rate was 1 percent or less. This indicates that our methodology succeeds in evicting objects which are most likely to have their true class label.

4.6.3 Streams with Non-Uniform Arrival

Data streams are often modeled with non-uniform interarrival times. In this experiment, we show the accuracy of score scheduled classification on such streams. We simulate a data stream with an arrival process which is modeled to be Poisson distributed with mean interarrival times matching the constant streams presented in Section 4.6.1. The arrival rates are:

$$\lambda = \left\{ \frac{1}{\lfloor |D| r \rfloor}, 0.1 \leq r \leq 1 \right\}$$

Figure 4.4 and Figure 4.5 shows the classification accuracy for exponentially distributed interarrival times, computed as a function of r. As shown on the Two Pattern dataset, we are able to obtain a definite increase in accuracy over the round robin baseline. For the Face dataset, we see that our scheduling technique is able to improve performance until $r < 0.3$, upon which there is insufficient computation time to accurately discern a meaningful and

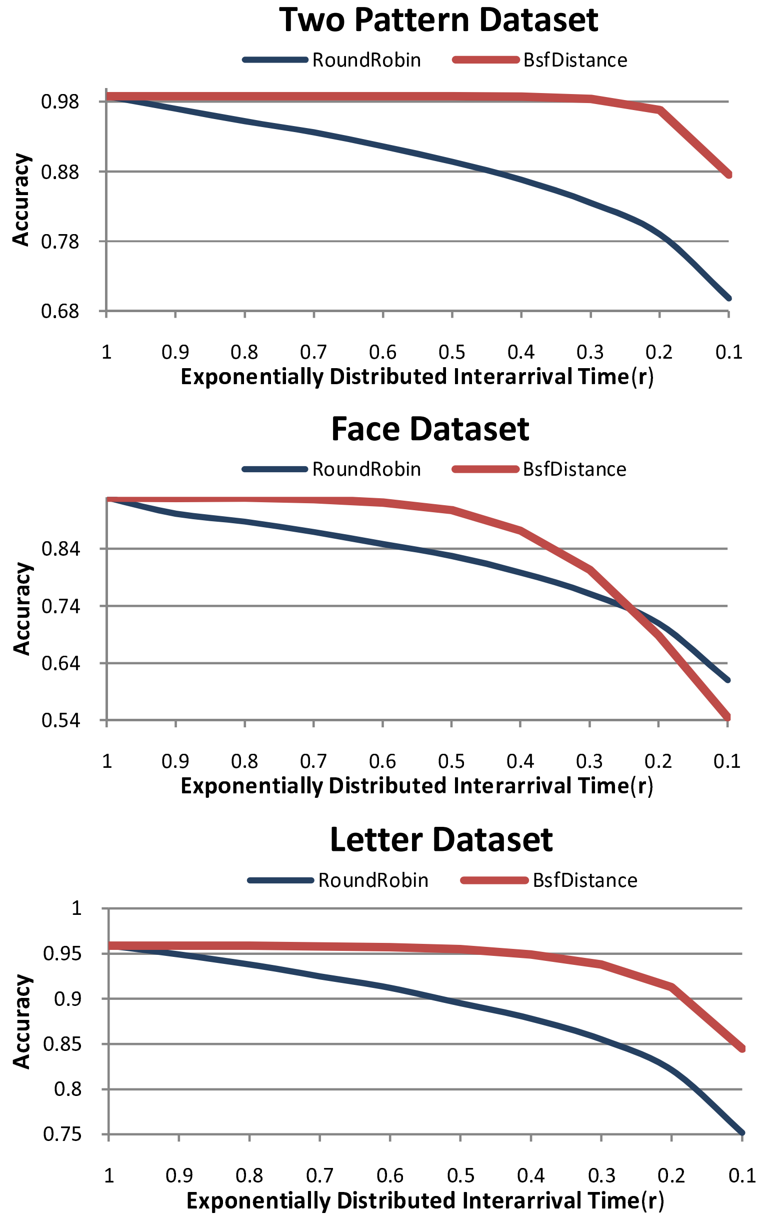

Figure 4.4: Classification accuracy on data streams with exponentially distributed interarrival times

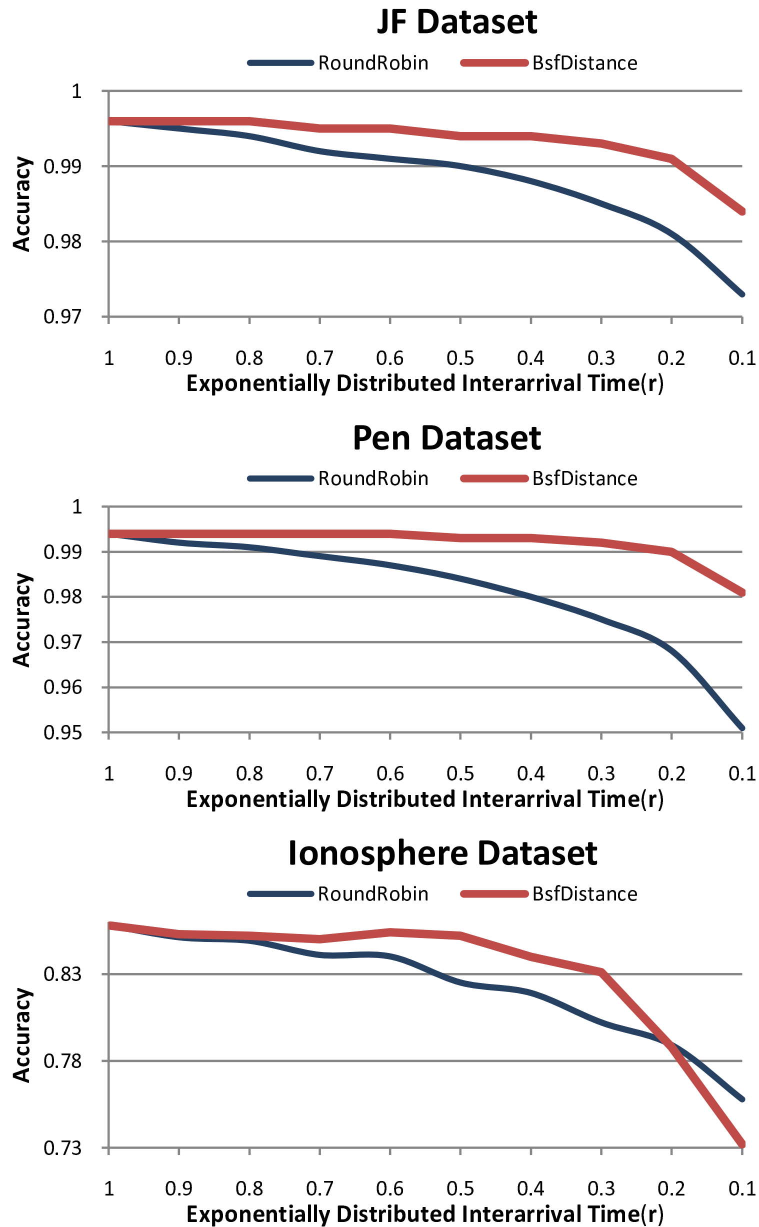

Figure 4.5: Classification accuracy on additional datasets with with exponentially distributed interarrival times

Figure 4.6: *left*) The adult Light Brown Apple Moth is harmless to agriculture, however it's larval form *right*) causes extensive damage to several commercially important crops

differentiating confidence value. Round robin outperforms in this scenario because it is fair for all entries in memory. Similar results can be seen across the remaining datasets.

4.6.4 A Case Study in Commercial Entomology

Several species of moths are harmful to agriculture. For example, *Epiphyas postvittana*, the Light Brown Apple Moth (LBAM) have larvae that feed on leaves and buds of plants, reducing photosynthetic rate, which in turn leads to general weakness and disfigurement.

In grapes and citrus, LBAM larvae can feed directly on the fruit, and the resulting damage renders fruit unmarketable. The LBAM is native to Australia, but appeared in California in 2007. Since that time, the California Department of Food and Agriculture has spent $70 million on attempts to eradicate it from California. If not eradicated, it is estimated it could cause $140 million in damage each year [1].

Of course, attempts at eradication must be very careful; many moths are important pollinators of plants. For example, the Yucca moth (*Tegeticula maculata*) is the only animal that is

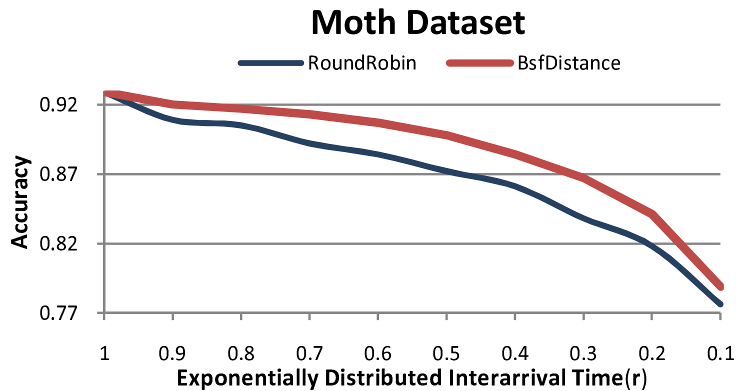

Figure 4.7: Insect classification with memory buffer constrained to four objects

the right size and shape to pollinate yucca flowers. If it is accidentally eradicated along with the LBAM, yucca flowers would be threatened, which could further affect additional fauna. Note that the LBAM is just one of the hundreds of insects which are known to be harmful to agriculture, livestock, or humans.

With this in mind, several companies, including ISCA Technologies of Riverside CA, are developing AVIDs, Automated Visual Identification Devices, which can recognize individual species or genera. Most of these systems currently just count the target insect, however systems are being developed that selectively trap only the target insect, and release all others. In order to be effective AVIDs must be mass produced, and therefore have limited computational resources.

Recent work has shown that it is possible to accurately classify moths using a compression-based distance measure [19]. The distance measure is effective, but not being a metric it does not allow an efficient indexing mechanism to make classification more tractable. Below we

describe our initial experiments to port the compression-based distance measure to resource limited hardware using our anytime framework.

As a preprocessing step, we first cluster the original moth data into three dominant clusters to obtain more exemplars per grouping. We then simulate non-uniform insect arrival [16] using the methodology described in Section 4.6.3. Due to resource constraints inherent in our target environment, we set the available memory size to 5 percent of the testing set, resulting in a memory buffer of only four objects. Our classification accuracy compared to round robin for different arrival rates is shown in Figure 4.7. We see that the score scheduled approach consistently outperforms round robin.

4.7 Concluding Remarks

In this chapter we introduced a framework for improving the overall accuracy of anytime nearest neighbor classification for a set of concurrently processing objects. We have shown, over a wide range of diverse datasets and problems that our method can achieve performance increases by scheduling computation time to objects which need it most. For future work, we look forward to examining more in-depth, the utility of this framework and considering the interplay between variability in object duration, amount of concurrency, and different scoring methods.

Chapter 5

Conclusion

As advances in science and technology have continually increased the existence of, and capability for users to monitor, record, and examine data, data mining has become a common and necessary toolset in order to gain additional insight on this influx of data. In this dissertation, we studied methods which are used for overcoming the characteristic challenges of scale in order to perform similarity search on large time series datasets.

In Chapter 2, we introduced a novel multi-resolution symbolic representation for time series called indexable Symbolic Aggregate approXimation (*i*SAX). The *i*SAX representation allows for the indexing of time series in order to facilitate similarity search. We further demonstrated its utility by performing experimental evaluation on a wide range of diverse datasets and showed how exact and approximate search can be used in conjunction to expedite higher level data mining operators to solve real world problems. The size of the datasets we considered are larger than any other in the current literature and notably, our results

confirmed the notion that even simple measures (such as the Euclidean distance) perform exceedingly well when the training set becomes very large.

In Chapter 3, we improved indexing performance by providing a new bulk loading algorithm. The index splitting policy was also modified to obtain higher node occupancy. These improvements expedite index construction time and resulted in a more compact index structure, allowing us to demonstrate the capability to index up to one billion time series.

Another aspect of our research considered using similarity search to perform classification under limited computation time and variable response rates. In such contexts, anytime algorithms, amenable to variable response times by exchanging quality of response as a function of time, have been found to be especially useful. In Chapter 4, we presented a generalized framework which utilizes a scoring function that estimates the intermediate result quality of an object being classified. Our contribution extends existing anytime algorithms to concurrent queries by dynamically scheduling computational resources for each object (in accordance with its score). We showed that the lack of such inter-object consideration would otherwise result in poor allocation of computation time and lead to reduced performance.

We believe that as the size of datasets continue to increase, the challenges and solutions we discussed in this dissertation will remain relevant for the future. From here, there are a number of research directions which is of interest:

- Utilizing index based similarity search as a subroutine will allow us to develop higher level data mining algorithms such as motif discovery, discord discovery, etc. These

applications are extremely interesting in that they build upon basic similarity search to provide solutions for much more complicated problems.

- Most approaches for time series similarity search require the specification of an "window of interest" upon which to build upon or evaluate. However, it is not uncommon to have patterns of interest be of widely varying lengths. In such situations, a common approach is to perform indexing at the smallest length of interest and evaluate a candidate set at the longer lengths, a potentially expensive post-processing step. Mitigating the performance drawbacks regarding this hard dependency on window length would be invaluable.

- It is not uncommon to see a disconnect between real world data and that which is commonly used in academia. The presence of noise and lack of standardization in real world datasets often make it difficult to apply existing approaches meaningfully. While it is common to brush off such consideration as a data cleaning step, we feel that data mining approaches would have much higher adoption and utility if our techniques are more robust to such data characteristics.

Bibliography

[1] CISR: Light Brown Apple Moth. http://cisr.ucr.edu/light_brown_apple_moth.html. Retrieved Jan 15, 2010.

[2] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Foundations of Data Organization and Algorithms*, volume 730 of *Lecture Notes in Computer Science*, pages 69–84. Springer Berlin / Heidelberg, 1993.

[3] R. Agrawal, K.-I. Lin, H. S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *VLDB*, pages 490–501, 1995.

[4] P. C. Andersen, B. V. Brodbeck, and R. F. M. III. Assimilation efficiency of free and protein amino acids by *homalodisca vitripennis* (hemiptera: Cicadellidae: Cicadellinae) feeding on citrus sinensis and vitis vinifera. *Florida Entomologist*, 92(1):116–122, 2009.

[5] C. Anderson. The end of theory: The data deluge makes the scientific method obsolete. *Wired*, 16(7), June 2008. http://www.wired.com/science/discoveries/magazine/16-07/pb_theory.

[6] H. André-Jönsson and D. Z. Badal. Using signature files for querying time-series data. In *PKDD '97: Proceedings of the First European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 211–220, London, UK, 1997. Springer-Verlag.

[7] I. Assent, R. Krieger, F. Afschari, and T. Seidl. The ts-tree: efficient time series search and retrieval. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 252–263, New York, NY, USA, 2008. ACM.

[8] A. Asuncion and D. Newman. UCI machine learning repository, 2007. http://www.ics.uci.edu/~mlearn/MLRepository.html.

[9] E. A. Backus and W. H. Bennett. The ac-dc correlation monitor: New epg design with flexible input resistors to detect both r and emf components for any piercing-sucking hemipteran. *Journal of Insect Physiology*, 55(10):869 – 884, 2009.

[10] A. Bagnall, C. A. Ratanamahatana, E. Keogh, S. Lonardi, and G. Janacek. A bit level representation for time series data mining with shape based similarity. *Data Min. Knowl. Discov.*, 13(1):11–40, 2006.

[11] L. V. Batista, E. U. K. Melcher, and L. C. Carvalho. Compression of ecg signals by optimized quantization of discrete cosine transform coefficients. *Medical Engineering & Physics*, 23(2):127 – 134, 2001.

[12] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *KDD Workshop*, pages 359–370, 1994.

[13] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250, New York, NY, USA, 2001. ACM.

[14] P. S. Bradley and U. M. Fayyad. Refining initial points for k-means clustering. In *ICML '98: Proceedings of the Fifteenth International Conference on Machine Learning*, pages 91–99, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[15] R. Butt and S. J. Johansson. Where do we go now?: anytime algorithms for path planning. In *FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 248–255, New York, NY, USA, 2009. ACM.

[16] J. A. Byers. Temporal clumping of bark beetle arrival at pheromone traps: Modeling anemotaxis in chaotic plumes. *Journal of Chemical Ecology*, 22(11):2133–2155, November 1996.

[17] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 599–610, New York, NY, USA, 2004. ACM.

[18] A. Camerra, J. Shieh, T. Palpanas, and E. Keogh. Indexing one billion time series, 2010. (In Submission).

[19] B. J. L. Campana and E. J. Keogh. A compression based distance measure for texture. In *SDM*, pages 850–861, 2010.

[20] K.-P. Chan and A. W.-C. Fu. Efficient time series matching by wavelets. In *15th International Conference on Data Engineering (ICDE '99)*, pages 126–135, Washington - Brussels - Tokyo, Mar. 1999. IEEE.

[21] C. Chatfield. *Time-Series Forecasting*. Chapman and Hall / CRC Press, 1st edition, 2000.

[22] C. Chatfield. *The analysis of time series: an introduction*. CRC Press, Florida, US, 6th edition, 2004.

[23] J. Chen and S. Itoh. A wavelet transform-based ECG compression method guaranteeing desired signal quality. *IEEE Transactions on Biomedical Engineering*, 45(12):1414–1419, Dec. 1998.

[24] Q. Chen, L. C. 0002, X. Lian, Y. Liu, and J. X. Yu. Indexable pla for efficient similarity search. In *VLDB*, pages 435–446, 2007.

[25] B. Chiu, E. Keogh, and S. 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.

[26] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proc. VLDB Endow.*, 1(2):1542–1552, 2008.

[27] S. Esmeir and S. Markovitch. Interruptible anytime algorithms for iterative improvement of decision trees. In *UBDM '05: Proceedings of the 1st international workshop on Utility-based data mining*, pages 78–85, New York, NY, USA, 2005. ACM.

[28] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of ACM SIGMOD*, pages 419–429, Minneapolis, MN, 1994.

[29] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The kdd process for extracting useful knowledge from volumes of data. *Commun. ACM*, 39(11):27–34, 1996.

[30] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus. Knowledge discovery in databases: An overview. *AI Magazine*, 13(3):57–70, 1992.

[31] B. Fuglede and F. Topsoe. Jensen-shannon divergence and hilbert space embedding. In *Information Theory, 2004. ISIT 2004. Proceedings. International Symposium on*, page 31, june-2 july 2004.

[32] J. Grass and S. Zilberstein. Anytime algorithm development tools. Technical report, University of Massachusetts, Amherst, MA, USA, 1995.

[33] O. Grumberg, S. Livne, and S. Markovitch. Learning to order bdd variables in verification. *J. Artif. Int. Res.*, 18(1):83–116, 2003.

[34] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM.

[35] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[36] J. Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[37] E. A. Hansen and S. Zilberstein. Monitoring anytime algorithms. *SIGART Bull.*, 7(2):28–33, 1996.

[38] C. L. Hays. What wal-mart knows about customers' habits. The New York Times, November 2004. http://www.nytimes.com/2004/11/14/business/yourmoney/14wal.html.

[39] Y.-W. Huang and P. S. Yu. Adaptive query processing for time-series data. In *KDD '99: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 282–286, New York, NY, USA, 1999. ACM.

[40] B. Hui, Y. Yang, and G. I. Webb. Anytime classification for a pool of instances. *Mach. Learn.*, 77(1):61–102, 2009.

[41] G. Hulten and P. Domingos. Mining complex models from arbitrarily large databases in constant time. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 525–531, New York, NY, USA, 2002. ACM.

[42] J. W. IJdo, A. Baldini, D. C. Ward, S. T. Reeders, and R. A. Wells. Origin of human chromosome 2: an ancestral telomere-telomere fusion. *Proc Natl Acad Sci U S A*, 88(20):9051–9055, Oct 1991.

[43] S. Kaffka, B. Wintermantel, M. Burk, and G. Peterson. Protecting high-yielding sugarbeet varieties from loss to curly top. http://sugarbeet.ucdavis.edu/Notes/Nov00a.htm, 2000.

[44] E. Keogh. Exact indexing of dynamic time warping. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 406–417. VLDB Endowment, 2002.

[45] E. Keogh. SAX home page. http://www.cs.ucr.edu/~eamonn/SAX.htm, 2008.

[46] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*, 3(3):263–286, August 2001.

[47] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 151–162, New York, NY, USA, 2001. ACM.

[48] E. Keogh and S. 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.

[49] E. Keogh and J. Shieh. *i*SAX home page. http://www.cs.ucr.edu/~eamonn/iSAX/iSAX.html, 2008.

[50] E. Keogh, X. Xi, L. Wei, and C. A. Ratanamahatana. The ucr time series classification/clustering homepage. htp://www.cs.ucr.edu/~eamonn/time_series_data, 2006.

[51] F. Kindt, N. N. Joosten, D. Peters, and W. F. Tjallingii. Characterisation of the feeding behaviour of western flower thrips in terms of electrical penetration graph (epg) waveforms. *Journal of Insect Physiology*, 49(3):183 – 191, 2003.

[52] P. Kranen and T. Seidl. Harnessing the strengths of anytime algorithms for constant data streams. *Data Min. Knowl. Discov.*, 19(2):245–260, 2009.

[53] N. Kumar, N. Lolla, E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Time-series bitmaps: a practical visualization tool for working with large time series databases. In *SIAM 2005 Data Mining Conference*, pages 531–535. SIAM, 2005.

[54] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *DMKD '03: Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 2–11, New York, NY, USA, 2003. ACM.

[55] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Min. Knowl. Discov.*, 15(2):107–144, 2007.

[56] J. Lin, M. Vlachos, E. Keogh, and D. Gunopulos. Iterative incremental clustering of time series. In *Advances in Database Technology - EDBT 2004*, volume 2992 of *Lecture Notes in Computer Science*, pages 521–522. Springer Berlin / Heidelberg, 2004.

[57] T. Lindgren. Anytime inductive logic programming. In *Proceedings of the 15th International Conference on Computers and Their Applications*, pages 439–442, 2000.

[58] W.-K. Loh, S.-W. Kim, and K.-Y. Whang. Index interpolation: an approach to subsequence matching supporting normalization transform in time-series databases. In *CIKM '00: Proceedings of the ninth international conference on Information and knowledge management*, pages 480–487, New York, NY, USA, 2000. ACM.

[59] M. R. Macnair and G. A. Parker. Models of parent-offspring conflict. iii. intra-brood conflict. *Animal Behaviour*, 27(Part 4):1202 – 1209, 1979.

[60] K. H. Malatesta, S. J. Beck, G. Menali, and E. O. Waagen. The AAVSO Data Validation Project. *Journal of the American Association of Variable Star Observers (JAAVSO)*, 34:238–250, Dec. 2006.

[61] M. B. Manser and G. Avey. The effect of pup vocalisations on food allocation in a cooperative mammal, the meerkat (suricata suricatta). *Behavioral Ecology and Sociobiology*, 48(6):429–437, November 2000.

[62] V. Megalooikonomou, Q. Wang, G. Li, and C. Faloutsos. A multiresolution symbolic representation of time series. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE05)*, pages 668–679, 2005.

[63] Y.-S. Moon, K.-Y. Whang, and W.-S. Han. General match: a subsequence matching method in time-series databases based on generalized windows. In *SIGMOD Conference*, pages 382–393, 2002.

[64] Y. Morinaka, M. Yoshikawa, T. Amagasa, and S. Uemura. The l-index: An indexing structure for efficient subsequence matching in time sequence databases. In *Pacific-Asian Conf. on Knowledge Discovery and Data Mining*, 2001.

[65] A. Mueen, E. Keogh, and N. Bigdely-Shamlo. Finding time series motifs in disk-resident data. *Data Mining, IEEE International Conference on*, 0:367–376, 2009.

[66] A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, and M. B. Westover. Exact discovery of time series motifs. In *SDM*, pages 473–484, 2009.

[67] M. K. Ng, Z. Huang, and M. Hegland. Data-mining massive time series astronomical data sets - a case study. In *PAKDD '98: Proceedings of the Second Pacific-Asia Conference on Research and Development in Knowledge Discovery and Data Mining*, pages 401–402, London, UK, 1998. Springer-Verlag.

[68] F. Portet, E. Reiter, J. Hunter, and S. Sripada. Automatic generation of textual summaries from neonatal intensive care data. In *Proccedings of the 11th Conference on Artificial Intelligence in Medicine (AIME 07). LNCS*, pages 227–236, 2007.

[69] C. A. Ratanamahatana and E. J. Keogh. Three myths about dynamic time warping data mining. In *SDM*, 2005.

[70] J. Rogers, R. Garcia, W. Shelledy, J. Kaplan, A. Arya, Z. Johnson, M. Bergstrom, L. Novakowski, P. Nair, A. Vinson, D. Newman, G. Heckman, and J. Cameron. An initial genetic linkage map of the rhesus macaque (macaca mulatta) genome using human microsatellite loci. *Genomics*, 87(1):30 – 38, 2006.

[71] N. Roy and A. McCallum. Toward optimal active learning through sampling estimation of error reduction. In *ICML '01: Proceedings of the Eighteenth International Conference on Machine Learning*, pages 441–448, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[72] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 26(1):43 – 49, feb 1978.

[73] S. Scholle and T. Schfer. Atlas of states of sleep and wakefulness in infants and children. *Somnologie - Schlafforschung und Schlafmedizin*, 3(4):163–241, October 1999.

[74] T. Seidl, I. Assent, P. Kranen, R. Krieger, and J. Herrmann. Indexing density models for incremental learning and anytime classification on data streams. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 311–322, New York, NY, USA, 2009. ACM.

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

[76] H. Shatkay and S. B. Zdonik. Approximate queries and representations for large data sequences. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 536–545, Washington, DC, USA, 1996. IEEE Computer Society.

[77] J. Shieh and E. Keogh. iSAX: indexing and mining terabyte sized time series. In *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 623–631, New York, NY, USA, 2008. ACM.

[78] J. Shieh and E. Keogh. iSAX: disk-aware mining and indexing of massive time series datasets. *Data Min. Knowl. Discov.*, 19(1):24–57, 2009.

[79] J. Shieh and E. Keogh. Polishing the right apple: Anytime classification also benefits data streams with constant arrival times, 2010. (In Submission).

[80] M. Steinbach, P.-N. Tan, V. Kumar, S. Klooster, and C. Potter. Discovery of climate indices using clustering. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 446–455, New York, NY, USA, 2003. ACM.

[81] A. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30:1958–1970, 2008.

[82] I. Toyoshima. Surveillance system, surveillance method and computer readable medium, 2008.

[83] K. Ueno, X. Xi, E. Keogh, and D.-J. 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.

[84] L. Wei, E. Keogh, H. V. Herle, and A. Mafra-neto. Atomic wedgie: Efficient query filtering for streaming time series. In *Proceedings of the 5 th IEEE International Conference on Data Mining (ICDM 2005*, pages 490–497, 2005.

[85] D. R. Wilson and T. R. Martinez. Reduction techniques for instance-based learning algorithms. *Machine Learning*, 38(3):257–286, March 2000.

[86] X. Xi, E. J. Keogh, C. R. Shelton, L. Wei, and C. A. Ratanamahatana. Fast time series classification using numerosity reduction. In *ICML*, pages 1033–1040, 2006.

[87] X. Xu. *Data Mining Techniques in Gene Expression Data Analysis*. PhD thesis, National University of Singapore, 2006.

[88] Y. Yang, G. Webb, K. Korb, and K. M. Ting. Classifying under computational resource constraints: anytime classification using probabilistic estimators. *Machine Learning*, 69(1):35–53, October 2007.

[89] D. Yankov, E. Keogh, S. Lonardi, and A. W. Fu. Dot plots for time series analysis. In *ICTAI '05: Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, pages 159–168, Washington, DC, USA, 2005. IEEE Computer Society.

[90] D. Yankov, E. J. Keogh, L. Wei, X. Xi, and W. L. Hodges. Fast best-match shape searching in rotation invariant metric spaces. In *SDM*, 2007.

[91] B.-K. Yi, H. V. Jagadish, and C. 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.

[92] S. Zilberstein and S. Russell. Approximate reasoning using anytime algorithms. In *Imprecise and Approximate Computation*, volume 318 of *The International Series in Engineering and Computer Science*, pages 43–62. Springer US, 1995.