Learning States for Detecting Anomalies in Time Series

by

Stan Weidner Salvador

A master's thesis submitted to the College of Engineering at
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
May, 2004

Technical Report
CS-2004-05

Learning States for Detecting Anomalies in Time Series

a master's thesis by

Stan Weidner Salvador

Approved as to style and content

Abstract

Learning States for Detecting Anomalies in Time Series

by

Stan Weidner Salvador

Thesis Advisor: Philip K. Chan, Ph.D.

The normal operation of a device can be characterized in different operational states. To identify these states, we introduce a segmentation algorithm called Gecko that can determine a reasonable number of segments using our proposed L method. We then use the RIPPER classification algorithm to describe these states in logical rules. Finally, transitional logic between the states is added to create a finite state automaton. Multiple time series may be used for training, by merging several time series into a single representative time series using dynamic time warping.

Our empirical results, on data obtained from the NASA shuttle program, indicate that the Gecko segmentation algorithm is comparable to a human expert in identifying states, and our L method performs better than the existing permutation tests method when determining the number of segments to return in segmentation algorithms. Empirical results have also shown that our overall system can track normal behavior and detect anomalies. Additionally, if multiple time series are used for training, the model will generalize to cover unseen normal variations and time series that are "between" the time series used for training.

Table of Contents

List of Figures vii

List of Tables ix

Acknowledgments x

Chapter 1: Introduction 1

1.1 Problem Statement 1

1.2 Approach 2

1.3 Key Contributions 2

1.4 Organization 3

Chapter 2: Related Work 5

2.1 Clustering 5

2.1.1 Partitioning Clustering Algorithms 5

2.1.2 Hierarchical Clustering Algorithms 6

2.1.3 Density-based Clustering Algorithms 7

2.1.4 Grid-based Clustering Algorithms 8

2.1.5 Applying Clustering Algorithms to Time Series Data 8

2.2 Segmentation 9

2.3 Determine Number of Clusters or Segments 10

2.4 Anomaly Detection in Time Series 13

Chapter 3: Identifying the States in a Time Series 15

3.1 Gecko Algorithm 17

3.1.1 Gecko Overview 17

3.1.2 Phase 1: Create Sub-Clusters 19

3.1.3 Phase 2: Repeatedly Merge Clusters 20

3.1.4 Phase 3: Determine Number of Clusters 23

3.2 Empirical Evaluation 23

3.2.1 Procedures and Criteria 24

3.2.2 Results and Analysis 25
3.3 Summary 27
Chapter 4: Determining the Number of Clusters/Segments in Clustering/Segmentation Algorithms 29
4.1 The L Method 30
4.1.1 Evaluation Graphs 30
4.1.2 Finding the Knee via the L Method 33
4.1.3 Iterative Refinement 36
4.1.4 Refinements for Segmentation Algorithms 38
4.2 Empirical Evaluation 39
4.2.1 Identifying the Number of Clusters 39
4.2.1.1 Procedures and Criteria 40
4.2.1.2 Results and Analysis 42
4.2.2 Identifying the Number of Segments 46
4.2.2.1 Procedures and Criteria 46
4.2.2.2 Results and Analysis 49
4.3 Summary 54
Chapter 5: Time Series Anomaly Detection Using States 55
5.1 Anomaly Detection System 55
5.1.1 Overview 57
5.1.2 Characterizing States by Generating Rules 58
5.1.3 State Transition Logic 58
5.2 Empirical Evaluation 61
5.2.1 Procedures and Criteria 61
5.2.2 Self-Tracking Results 61
5.2.3 Normal Operation Results 62
5.2.4 Detecting Anomalies Results 63
5.3 Summary 64
Chapter 6: Building a Normal Model by Training on Multiple Time Series 65
6.1 Dynamic time Warping 67

6.1.1 Problem Formulation 68
6.1.2 Dynamic Time Warping Algorithm 69
6.1.3 Derivative Time Warping 73
6.2 Using DTW to Train on Multiple Time Series 76
6.3 Empirical Evaluation 79
6.3.1 Procedures and Criteria 79
6.3.2 Results and Analysis 82
6.4 Summary 85
Chapter 7: Concluding Remarks 87
7.1 Summary of Contributions 87
7.2 Limitations and Future Work 89
References 95
Appendix: Sample Run of the Anomaly Detection System 100
Input Files for Training 101
Merging Time Series 102
Identifying States 103
Rule Generation 105
State-Transition Logic 108
Anomaly Detection 112
Usage (command-line interface) 113

List of Figures

Figure 3.1. Overview of the Gecko Algorithm. 18
Figure 3.2. Graphs for ln(*slope*+1) and arctan(*slope*). 22
Figure 3.3. A data set after being clustered by Gecko (16 clusters). 24
Figure 4.1. A sample evaluation graph. 32
Figure 4.2. Finding the number of clusters using the L method. 34
Figure 4.3. All four possible pairs of best-fit lines for a small evaluation graph. 34
Figure 4.4. Full and partial evaluation graphs created by CURE. Only the first 100 points are shown on the right side. 36
Figure 4.5. Pseudocode to Iterative Refine the knee with the L Method. 37
Figure 4.6. Data sets 1-6 for evaluating the L method in clustering algorithms (data set #7 not shown). 41
Figure 4.7. Actual number of clusters and the correct number predicted by the L method (axes: *x*= # of clusters, *y*=evaluation metric – lines: solid lines=correct # of clusters, dashed lines=# of clusters determined by L method). 45
Figure 4.8. Data sets 1, 3, 4, 5, 6, and 7 for evaluating the L method in segmentation algorithms. 47
Figure 4.9. The reasonable range for the number of segments and the number returned by the L method. (axes: *x*=# of segments, *y*=evaluation metric – short dashed line=# of segments determined by the L method, long solid lines=marks the boundaries of the reasonable range for the # of segments. 52
Figure 5.1. Main steps in time series anomaly detection. 57
Figure 5.2. Expanded state-transition logic with transition thresholds. 60
Figure 6.1. Two time series “warped” together using dynamic time warping. 67
Figure 6.2. A cost matrix with the min. dist. warp path traced through it. 70
Figure 6.3. The order that the cost matrix is filled. 71
Figure 6.4. Four shaded areas showing how cells that are calculated. 72

Figure 6.5. A warping between two time series using original values. 74

Figure 6.6. A warping between two time series using slope values. 75

Figure 6.7. Main steps in time series anomaly detection with multiple input time series used during training. 77

Figure 6.8. A valve being turned on and off at different voltages. 81

List of Tables

Table 3.1. Number of segments found by Gecko and a human expert. 25

Table 3.2. Quality of segments produced by Gecko and BUS................................ 26

Table 4.1. Results of using the L method and the Gap statistic with various clustering algorithms. 43

Table 4.2. Results of using the L method with three hierarchical segmentation algorithms. 50

Table 5.1. Self-tracking of a time series. 62

Table 6.1. Results of training on a single time series, testing on all. 83

Table 6.2. Results of training on multiple time series, testing on all. 84

Acknowledgments

I would like to thank my thesis advisor Dr. Philip Chan for his help in writing this thesis, and giving me the opportunity to participate in his research. I also would like to thank the other members of my thesis committee, Dr. Marius-Calin Silaghi and Dr. Georgios Anagnostopoulos.

This work is partially supported by NASA (NAS10-02044). Thanks to Bobby Ferrell and Steven Santuro at NASA for providing data sets, helpful comments, and clustering evaluations. Others who have contributed to this research include Brian Buckley, Steve Creighton, and Walter Schiefele at Interface and Control Systems (ICS) who helped integrate our algorithms into their SCL expert system. John Brodie and Matt Mahoney also offered many helpful suggestions that helped to guide this research.

Chapter 1

Introduction

Expert (knowledge-based) systems are often used to help humans monitor and control critical systems in real-time. For example, NASA uses expert systems to monitor various devices on the space shuttle. However, populating an expert system's knowledge database by hand is a time-consuming and expensive process. In this paper we investigate machine learning techniques for generating knowledge that can monitor the operation of devices or systems. Specifically, we study methods for generating models that can detect anomalies in time series data.

The normal operation of a device can usually be characterized in different operational states. Segmentation or clustering techniques can help identify the various states. However, most methods directly or indirectly require a parameter to specify the number of segments/clusters in the time series data. The output of these algorithms is also not in a logical rule format, which is commonly used in expert systems for its ease of comprehension and modification. Furthermore, the relationships between these states need to be determined to allow tracking from one state to another and to detect anomalies.

1.1 Problem Statement

Given time series data depicting a system's normal operation, we desire to learn a model that can detect anomalies and can be *easily read and modified by human users*. We investigate a few issues in this paper. First, we want a segmentation algorithm that can dynamically determine a reasonable number of segments, and hence the number of states for our purposes. These states, collected from a device, should be comparable to those identified by human experts. Second, we would like to characterize these states in logical rules so that they can be read and modified with relative ease by humans. Third, given the knowledge of the different states, we wish to describe the relationship among them for tracking normal behavior and detecting anomalies.

1.2 Approach

To identify states, we introduce Gecko, which is able to segment time series data and determine a reasonable number of segments (states). Gecko consists of a top-down partitioning phase to find initial sub-clusters and a bottom-up phase which merges them back together. The appropriate number of segments is determined by what we call the L method. To characterize the states as logical rules, we use the RIPPER classification rule learning algorithm (Cohen 1995). Since different states often overlap in the one-dimensional input space, additional attributes are derived to help characterize the states. To track normal behavior and detect anomalies, we construct a finite state automaton (FSA) with the identified states.

1.3 Key Contributions

Our key contributions are:

- We demonstrate a method that performs time series anomaly detection via generated states and logical rules that can easily be understood and modified by humans.
- We introduce an algorithm named Gecko for segmenting time series data into states.

- We propose the L method that dynamically determines a reasonable number of clusters. The L method is general enough to be used with any hierarchical clustering or segmentation algorithm.
- We demonstrate how derivative time warping can be used to merge multiple time series together into an "average" time series in order to extend our anomaly detection so it may train on multiple time series.
- Empirical evaluations, using 14 spatial and time series data sets and 6 different clustering and segmentation algorithms, indicate that our L Method performs favorably to existing methods that determine the number of clusters or segments to return. Our L method is shown to work well for a wide range of algorithms, clusters with elaborate shape, and for clusters/segments that are overlapping and not well-separated.
- Our empirical evaluations, with data from NASA, indicate that Gecko performs comparably with a NASA expert and the overall system can track normal behavior and detect anomalies.

1.4 Organization

Chapter 2 gives an overview of related work on topics related to this research. The topics covered are: anomaly detection, segmentation, clustering, determining the number of clusters in a dataset.

Chapter 3 discusses our Gecko segmentation algorithm that is able to identify the states in a time series. The number of states that are returned is determined automatically by our L Method.

A complete description of our L method is contained in Chapter 4. The L method determines the number of clusters or segments to return from any hierarchical clustering or segmentation algorithm by locating the knee in the curve of an evaluation graph.

In Chapter 5, our overall anomaly detection system is described. The system explained in this chapter takes a single time series, identifies its operational states, characterizes each state, and creates state transition logic between them to implement anomaly detection.

Chapter 6 describes how to use dynamic time warping to extend the simple anomaly detection system in Chapter 5 to incorporate multiple time series for building a single normal model. The dynamic time warping algorithm is also explained in detail.

In Chapter 7, we summarize our work and our key contributions, as well as the limitations of our work. We also state the direction of future work for our research.

Chapter 2

Related Work

Before the details of our anomaly detection system are explained, we give an overview of related work in time series anomaly detection, and also other algorithms that are related to components in our anomaly detection system. Topics covered in this chapter are: clustering, segmentation, determining the number of clusters or segments, and anomaly detection in time series.

2.1 Clustering

Clustering algorithms take spatial data (2 or more dimensions) as input and return a set of clusters such that all points in a cluster are similar to each other and dissimilar to points in other clusters. There are four main categories of clustering algorithms: partitioning, hierarchical, density-based, and grid-based.

2.1.1 Partitioning Clustering Algorithms

Partitioning algorithms are the most classical group of clustering algorithms. The *K*-means algorithm (Hartigan 1975) is the most commonly used clustering algorithm due to its simplicity. *K*-means initially creates *k* random cluster centers and assigns every point to its closest cluster center. The cluster centers are then re-calculated, and every data point is re-assigned to its closest cluster center. The iterative refinement of the *k* clusters stops when no the cluster centers do not change in an iteration. Despite the popularity of *K*-means, it has significant disadvantages:

- The quality of the clusters produced is heavily dependent on the initial cluster centers that are chosen, and the set of clusters returned may vary significantly between runs of the algorithm on the same data set.
- The selection of the value *k* (number of clusters to return) needs to be specified by the user.
- The algorithm is not efficient for large data sets.
- Outliers can decrease the quality of the clusters that are returned.
- Only spherical clusters can be found.

Many partitioning algorithms have also been developed to overcome some of the disadvantages of *K*-means, chiefly the lack of scalability of *K*-means. PAM (Ng & Hah 1994) is one such algorithm that attempts to speed up the *K*-means algorithm by sampling the data.

2.1.2 Hierarchical Clustering Algorithms

Hierarchical algorithms can be agglomerative and/or divisive. The agglomerative (bottom-up) approach initially starts with many clusters and repeatedly merges the two most similar clusters together, while the divisive (top-down) approach initially places all of the data into a single cluster and repeatedly splits a cluster into two. The merging or splitting of clusters continues until the stopping criterion (usually the desired number of clusters *k*) is reached.

Hierarchical clustering algorithms are popular for scientific fields where the data being clustered contains sub-groups within the larger clusters because hierarchical algorithms can create a tree of clusterings called a dendrogram. This dendrogram can be used to view sets of clusters at varying granularities and to discover relationships within the data that would be missed at only a single clustering level (single set of clusters). ROCK (Guha, Rastogi & Shim 1999), CURE (Guha, Rastogi & Shim 1999) and Chameleon (Karypis, Han & Kumar

1999) are hierarchical algorithms that differ mostly in their similarity functions, which favor spherical, elliptical, and non-spherical clusters (respectively). Advantages of hierarchical clustering algorithms include:

- The ability to produce a dendrogram containing sets of clusters for many values of *k*.
- The similarity or distance measure between points or clusters is extremely flexible, which is responsible for the large number of existing hierarchical clustering algorithms.

The major disadvantage of hierarchical clustering algorithms is that once a merge or split has been performed, it cannot be undone. If the merge/split was a poor choice it may lead to future splits/merges that are also of poor quality.

2.1.3 Density-based Clustering Algorithms

Density-based algorithms, e.g., DBSCAN (Ester et al. 1996) and DENCLUE (Hinneburg & Keim 1998), are able to efficiently produce clusters of arbitrary shape and are also able to handle noise. If the density of a region is above a specified threshold, it is assigned to a cluster; otherwise it is considered to be noise. Connected regions of points that have a density above the threshold are considered to be in a single cluster. Density-based algorithms are efficient and are sensitive to the presence of outliers. However, the main disadvantage of density-based clustering algorithms is that several unintuitive parameters (cannot simply specify the number of clusters desired) need to be set for good results. The neighborhood width and the density threshold are two parameters that usually must be set. Another disadvantage is that density-based methods only work well when the density of the data is more or less uniform within the clusters, which is often not the case.

2.1.4 Grid-based Clustering Algorithms

Grid-based algorithms, such as WaveCluster (Seikholeslami, Chatterjee & Zhang 1998), reduce the clustering space into a grid of cells which enables efficient clustering of very large datasets. Many grid-based methods can automatically remove outliers and have a time complexity of O(N) if there is a small number of dimensions and the data is concentrated. Grid-based methods are better suited for clustering large amounts of very concentrated data, rather than sparse data.

2.1.5 Applying Clustering Algorithms to Time Series Data

All of the clustering algorithms discussed so far were designed to cluster spatial data with at least a two dimensional distribution. However, we are interested in finding clusters in time series data with a one dimensional distribution. Even multi-dimensional time series have a one-dimensional data distribution because a time series is a function. We also wish to find states in the time series that are non-overlapping with other states in the time dimension.

Partitioning methods iteratively refine a set of clusters by repeatedly assigning points to the closest cluster center and recalculating the center of the cluster. If this method is constrained so clusters are non-overlapping in the time dimension, local minima are much more likely to occur than in standard usage with two or more dimensions. The local minima are more frequent because the range of freedom when adjusting cluster centers has been decreased from two or more dimensions, to sliding it left or right in the time axis where it can get stuck between two other clusters (overlapping of clusters is not permitted to occur along the time axis).

Density based clustering methods cannot be used to find states in time series data because they return a set of clusters that are isolated regions of the data where the density is above some threshold. Significant portions of data between the clusters that have a density below the density threshold are considered to be noise. For time series data, it is preferable that all of the data points are placed into some cluster, and any noise in the time series should be dealt with by smoothing or filtering the time series rather than simply ignoring it. In order to prevent data from being treated as noise and thrown out, the density threshold must be set to a very low value which will cause only a single cluster to be returned, which is uninformative.

Grid based methods can cause a large increase in execution time by placing the data into a grid and counting the number of points in each grid cell and clustering the cells weighted by the number of points in them, rather than clustering individual data points. However, since time series data is a continuous function and has a single dimensional distribution, the data is much too sparse to benefit from the use of a grid.

Hierarchical clustering algorithms on the other hand can be modified to cluster time series data. They may use any similarity or distance function that is desired, and all that is needed is a restriction on the merging and splitting steps that forces clusters to remain non-overlapping in the time dimension after each step. Additionally, since a wide range clusterings are returned as a hierarchical dendrogram tree, evaluation may be performed to determine what level of the tree produced the best set of clusters. Our Gecko clustering algorithm that is explained in Chapter 3 is a modified hierarchical clustering algorithm.

2.2 Segmentation

Segmentation algorithms take time series data as input and produce a Piecewise Linear Approximation (PLA). A PLA is a set of consecutive line segments that fit the original data points as closely as possible. There are three common approaches to segmentation (Keogh et al. 2001).

1. ***Sliding Window***: A segment is grown until the error of the segment is above a specified threshold, then a new segment is started.

2. ***Top-down***: The entire time series is recursively split until the desired number of segments is reached, or an error threshold is reached.
3. ***Bottom-up***: Begin with *N*/2 segments. Repeatedly merge the two adjacent segments that will increase the approximation error of the PLA the by the smallest amount if they are joined. Keep merging segments until either the desired number of segments is reached, or the error of the PLA reaches the threshold value.

The sliding window approach creates poorest linear approximations but runs the quickest. Top-down segmentation creates the best PLA but runs much slower than the other two methods. Bottom-up segmentation creates PLAs that are nearly as good as those of the top-down method, but runs much quicker than top-down segmentation.

Segmentation algorithms are somewhat related to clustering algorithms in that each segment can be thought of as a cluster. However, since segmentation algorithms attempt to minimize the vertical error of the line segments, they have a bias towards creating more segments in highly sloped regions than lower sloped regions. The vertical error of a segment in a highly sloped region is usually much larger than segments with lower slope, and segmentation algorithms will attempt to minimize that error by creating more segments in those highly sloped areas. This bias favoring more segments in areas of large magnitude slopes causes existing segmentation algorithms to be better suited for producing a fine grain partitioning, rather than a small set of segments that represent natural clusters.

2.3 Determine Number of Clusters or Segments

Five common approaches to estimating the dimension of a model (such as the number of clusters or segments) are: cross-validation, penalized likelihood estimation, permutation tests, resampling, and finding the knee of an error curve.

Cross-validation techniques create models that attempt to fit the data as accurately as possible. Monte Carlo cross-validation (Smyth 1996) has been successfully used to prevent over-fitting (too many clusters/segments). Penalized likelihood estimation also attempts to find a model that fits the data as accurately as possible, but also attempts to minimize the complexity of the model. Specific methods to penalize models based on their complexity are: MML (Baxter & Oliver 1996), MDL (Hansen & Yu 2001), BIC (Fraley & Raftery 1998), AIC, and SIC (Sugiyama & Ogawa 2001). Permutation tests (Vasko & Toivonen 2002) attempt to prevent segmentation algorithms from creating a PLA that over-fits the data by comparing the relative change in approximation error to the relative change of a 'random' time series. If the relative change in error begins to be similar between the time series and a random time series as more segments are added, it means that extra segments are fitting noise and not any underlying structure in the time series. Resampling (Roth et al. 2002) and Consensus Clustering (Monti et al. 2003) attempt to find the correct number of clusters by repeatedly clustering samples of the data set, and determining at what number of clusters the clusterings of the various samples are the most "stable."

The majority of these methods to automatically determine the best number of clusters/segments may not work very well in practice. Model-based methods, such as cross-validation and penalized likelihood estimation, are computationally expensive and often require the clustering/segmentation algorithm to be run several times. Their usefulness is limited to only the smallest data sets. Permutation tests and resampling are extremely inefficient, since they require the entire clustering algorithm to be re-run hundreds or even thousands of times. The majority of existing methods to find the knee of an error curve require the clustering algorithm to be re-run for every potential value of *k*. Even worse, many of the evaluation functions that are used to evaluate a set of clusters run in O(N^2) time. This means that it may take longer just to evaluate a set of clusters than it does to generate them. Most methods that find the knee of a curve also only work well when the clusters are well separated.

Some existing clustering algorithms have built-in mechanisms for automatically determining the number of clusters. The TURN* (Foss & Zaïane 2002) algorithm locates the knee of a curve by location the point where the 2nd derivative increases above some user specified threshold. A variant (Chiu et al. 2001) of the BIRCH (Zhang, Ramakrishnan & Livnv 1996) algorithm uses a mixture of the Bayesian Information Criterion (BIC) and the ratio-change between inter-cluster distance and the number of clusters.

Locating the "knee" of an error curve, in order to determine an appropriate number of clusters or segments, is well known, but it is not a particularly well-studied method. There are methods that statistically evaluate each point in the error curve, and use the point that either minimizes or maximizes some function as the number of clusters/segments to return. Such methods include the Gap statistic (Tibshirani, Walther & Hastie 2003) and prediction strength (Tibshirani et al. 2001). These methods generally (with the exception of hierarchical algorithms) require the entire clustering or segmentation algorithm to be run for each potential value of *k*.

The knee of a curve is loosely defined as the point of maximum curvature. The knee in a "# of clusters vs. classification error" graph can be used to determine the number of clusters to return. Various methods to find the knee of a curve are:

1. The largest magnitude difference between two points.
2. The largest ratio difference between two points (Chiu et al. 2001).
3. The first data point with a second derivative above some threshold value (Ester et al. 1996) (Foss & Zaïane 2002).
4. The data point with the largest second derivative (Harris, Hess & Venegas 2000).

5. The point on the curve that is furthest from a line fitted to the entire curve.
6. Our L-method, which finds the boundary between the pair of straight lines that most closely fit the curve.

This list is ordered from the methods that make a decision about the knee locally, to the methods that locate the knee globally by considering more points of the curve. The first two methods use only single pairs of adjacent points to determine where the knee is. The third and fourth methods uses more than one pair of points, but still only considers local trends in the graph. The last two methods consider all data points at the same time. Local methods may work well for smooth, monotonically increasing/decreasing curves. However, they are very sensitive to outliers and local trends, which may not be globally significant. The fifth method takes every point into account, but only works well for continuous functions, and not curves where the knee is a sharp jump. Our L method considers all points to keep local trends or outliers from preventing the true knee to be located, and is able to find knees that exist as sharp jumps in the curve.

2.4 Anomaly Detection in Time Series

Anomaly detection is the task of learning what is “normal” and determining when an event occurs that differs significantly from expected normal behavior. The approach that anomaly detection takes is the opposite of signature detection. Signature detection is explicitly given information on what is “bad,” and simply attempts to detect it when it happens. False alarms are rare when using signature detection because the algorithm has been programmed to know exactly what to look for to detect the known “bad” conditions. However, signature detection is unable to detect new attacks. Although anomaly detection systems produce more false alarms than signature detection systems, they have the significant advantage that they are able to detect new, previously unknown “bad” behavior. Virus scanners use signature detection to detect viruses. Virus scanners are very good at detecting known viruses with very few false alarms, but they cannot detect new viruses.

Nearly all of the work in time series anomaly detection relies on models that are not easily readable and therefore cannot be modified by a human for tuning purposes. Examples include creating a set of normal sequences through the negative-selection of random sequences (Dasgupta & Forrest 1996), frequency of normal sequences (Keogh, Lonardi, & Chiu 2002), adaptive resonance theory (Caudell & Newman 1993), and neural networks (Kozama et al. 1994). However, Langley et al. (Langley, Bay & Saito 2003) propose a method that uses process models to model a time series and predict future data. These process models are concise and are easily read and modified by humans, but their generation requires parameters to be set by a human that must have knowledge of the underlying processes that produce the time series.

Chapter 3

Identifying the States in a Time Series

The normal operation of a device can usually be characterized in different operational states. An operational state is a period in a time series in which the monitored device is in consistent state. If the operational states in a time series can be reliably discovered, that information may be used for simple discovery of interesting features in a time series, or can aid in anomaly detection by ensuring that the states occur in the expected sequence. A simple example of a device's operational states is the temperature of a light bulb that is turned on and off. A time series containing the temperature of this light bulb contains approximately five operational states: (1) the light bulb is off and is at room temperature; (2) the light bulb is switched on and quickly rises in temperature until a maximum temperature is reached; (3) the maximum temperature is reached and the temperature is constant; (4) the light bulb is turned off and the temperature slowly decreases; (5) the temperature has cooled and is once again at room temperature.

A state in a time series is a period of time where the time series is following a relatively steady trend, and portions of the time series immediately before and after the state do not follow the same trend. Notice that this is nearly identical to a commonly used definition of a cluster: "objects are clustered or grouped based on the principle of maximizing the inter-class similarity and minimizing the intra-class similarity" (Han & Kambler 2000). Thus, the problem of identifying distinct states in a time series is essentially a clustering problem since we wish to find states that are internally homogeneous, and contain data that are dissimilar to the data contained in other (adjacent in our case) states.

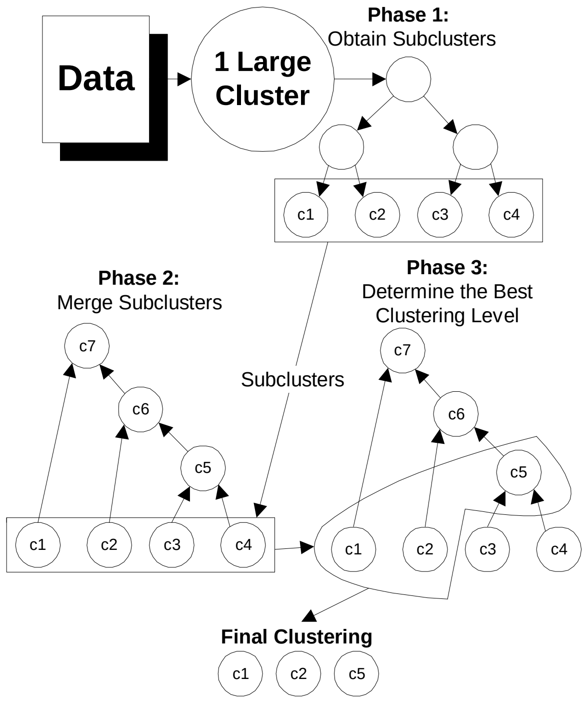
Segmentation techniques can help identify these various states. Segmentation algorithms create a piecewise linear approximation (PLA) of a time series. Since each segment in the PLA spans a period of time and does not overlap with other segments, the period of each segment may be considered to be an operational state of the time series. However, existing segmentation algorithms are better at creating fine-grain approximations of time series for compression, than they are at identifying a small number of distinct states or clusters in the data. Current segmentation algorithms attempt to create a set of line segments that minimize the overall error of the PLA with respect to a time series. However, because the error of a point from a line segment is measured as the vertical distance between them, areas of a time series with large slopes will have an artificially high approximation error. This will cause a bias that favors more segments where the slope has a large magnitude, and fewer segments where the slope is low. The result is that segmentation algorithms often create a poor set of segments when the number of segments becomes small. This is not a problem when segmentation algorithms are used to create fine-approximations of a time series, but if a small number of segments need to be found in the time series, existing segmentation algorithms will often fail to provide an acceptable set of segments. In addition, existing segmentation algorithms directly or indirectly require a parameter to specify the number of segments in the time series. It is often impractical to expect a human with sufficient domain knowledge to be available to select the number of segments to return.

We desire a segmentation algorithm that can dynamically determine a reasonable number of segments, and hence the number of states for our purposes. These states, collected from a device, should be comparable to those identified by human experts. To identify states, we introduce Gecko, which is able to segment time series data and determine a reasonable number of segments. Gecko consists of a top-down partitioning phase that creates initial sub-clusters and a bottom-up phase that merges them back together. The appropriate number of segments is determined by what we call the L method. The next section gives and overview of the Gecko algorithm and is followed by an empirical evaluation against an existing segmentation algorithm in section 3.2; section 3.3 summarizes our findings on the Gecko algorithm.

3.1 Gecko Algorithm

3.1.1 Gecko Overview

While segmentation algorithms typically create only a fine linear approximation of time series data, Gecko divides a time series into clusters. This number of clusters is determined by the algorithm and requires no user input. Note that segmentation is just a special case of clustering where clusters must not overlap along the time dimension. Gecko uses a 2-pass method (similar to Chameleon) that is a combination of both divisive and agglomerative hierarchical clustering. The first is a top-down pass that partitions the data into a large number of sub-clusters. This is followed by a bottom-up pass that merges the sub-clusters back together. The first top-down pass determines all of the potential boundary areas between clusters, which then enables the second bottom-up pass to focus only on the relative similarity of clusters. Hierarchical clustering algorithms are very similar to top-down/bottom-up segmentation. The difference is that hierarchical clustering is more general and any number of methods can be used to determine similarity, while segmentation is typically limited to the error of a segment's best-fit line.



The Gecko Algorithm (overview)

```
Input: D  // time series data
       s  // the minimum cluster size
Output:  c* clusters

Phase 1:
1. build a k-nearest neighbor graph of D (k=2*s)
2. recursively bisect the graph until no bisections can be made without
   creating a cluster smaller than s

Phase 2:
3. recursively merge the sub-clusters together until only one cluster
   remains - a dendrogram is created

Phase 3:
4. find c*, an appropriate number of clusters to return, by using the L method
5. extract c* clusters from the dendrogram and return them
```

Figure 3.1. Overview of the Gecko Algorithm.

The Gecko algorithm consists of three phases:

1. The first phase creates many small sub-clusters by initially putting all of the data points into a single cluster, and repeatedly splitting the largest cluster until all of the clusters can no longer be divided without becoming smaller than a specified parameter *s*.

2. The second phase takes all of the sub-clusters and repeatedly merges the two most similar clusters until all of the data is once again in the same cluster.

3. Using information recorded during merging, phase 3 is able to quickly determine the 'best' number of clusters that should be extracted from the hierarchical clustering.

The following three sections detail each of the phases in the Gecko algorithm.

3.1.2 Phase 1: Create Sub-Clusters

In the first phase, many small sub-clusters are created by a method that is very similar to the one used in Chameleon (Karypris, Hun & Kumar 1999), with the exception that Gecko forces cluster boundaries to be non-overlapping in the time dimension. The sub-clusters are created by initially placing all of the data points in a cluster, and repeatedly splitting the largest cluster until all of the clusters are too small to be split again without violating the minimum possible cluster size *s*.

To determine how to split the largest cluster, a *k*-nearest neighbor graph is built in which each node in the graph is a time series data point (measurements taken at a time-interval), and each edge is the similarity between two data points. Only the slopes of the original values (original sensor readings) are used to determine similarity, and not the original values themselves. Using only the slope will tend to produce sub-clusters that have constant slope, which produces sub-clusters that are as close to straight lines as possible. The *k*-nearest neighbor graph is constructed by creating an edge from every vertex to each of its *k* nearest (most similar) neighbors. The parameter *k* is not an input parameter. It is derived from *s* (smallest possible cluster size), and is defined to be 2**s*. Due to the importance of time, the *k* nearest points of a data point can be assumed to be the *k*/2 points on each size of the point according to the time axis. By using this graph, the similarity between groups of points (clusters) can be determined by computing the edge-cut (sum of the edges) between the two groups. Similarity between two points is defined to be ln(1.0/*distance*+1), where *distance* is the Euclidean distance (or any other distance method) between the two points. Justification for using this distance function will be explained in the next section. If the graph is split where the edge-cut is the smallest, then the two newly separated clusters will be dissimilar to each other and have high internal similarity.

Since all boundaries between clusters are cut cleanly by the time axis with no overlap, the typically NP-hard problem of graph bisection is simplified, and the optimal min-cut partitioning of a cluster can be quickly determined in fewer than *minClusterSize*-1 edge-cut checks (where *minClusterSize* is the number of data points contained in the cluster). There is no need for heuristics, because all possible edge-cut possibilities can be quickly computed with efficient data structures (Fiduccia & Mattheyses 1982).

3.1.3 Phase 2: Repeatedly Merge Clusters

In the second phase, the most similar pair of adjacent (in time) clusters is repeatedly merged until only one cluster remains. To determine which adjacent pair of clusters are the most similar, representative points are generated for each cluster and the two adjacent clusters with the closest representative points are merged. A single representative point is able to represent every point in a cluster because each cluster is internally homogeneous.

The representative point of a cluster contains a value for the slope of every original attribute in the data other than time. Clustering by the slope values causes the time series to be divided into flat regions. If a human is asked to pick out several distinct phases (or states) of a time series graph, he is likely to divide the graph into flat regions. This eyeball method of clustering is also essentially clustering by slope. Segmentation also relies exclusively on slope: if a minimum-error line (segment) is well fitted to a set of points it means that the segment has a consistent slope.

If raw slope values are used in the representative points, then the “distance” between clusters with slope values 100 and 101 would be the same as the distance between clusters with slope values 0 and 1. Differences in slopes that are near zero need to be emphasized because the same absolute change in slope can triple a small value, and be an insignificant increase for a large value. Relative differences between slopes cannot be measured by the percentage increase because in the preceding example, the percentage increase from 0 to 1 is undefined. Gecko uses representative values of slopes to determine the “distance” between two slopes by using the equation:

$$\text{Representative Slope} = \begin{cases} \ln(slope+1) & if\ slope \geq 0 \\ -\ln(-slope+1) & if\ slope < 0 \end{cases}$$

This equation emphasizes slopes near zero and decreases the effect of changes in slope when the slope values are large. Whenever a slope value is squared, its representative slope value (approximately) doubles. In the preceding example of comparing 2 pairs of clusters with slopes {100, 101} and {0, 1} the representative values of their slopes are {4.615, 4.625} and {0, 0.693}. This accurately reflects the relative difference between raw slopes and not the absolute difference.

Calculating the representative slope with a natural logarithm is similar to using the difference between angles to determine which lines are most similar. Consider a line with an angle of zero, if one end of the line is increased, the angle of the line will increase. If another line the same length is already at an eighty-five degree angle, and the higher end of the line is increased the same amount as before, the change of angle is much smaller than before. The idea is that a change in slope near zero is more significant than when the slope has a large magnitude. However, the problem with using the angle in degrees as the representative slope is that once the angle gets near ninety degrees (or π/2 radians), an increase in slope will no longer have an effect on the angle. This prevents lines with large slopes from being accurately compared to each other because they will all have nearly identical large angles. The natural logarithm, ln(*slope*+1), is very similar to converting the slope to degrees (in radians) using arctan(*slope*). In Figure 3.2, the similar relationship between ln(*slope*)+1 and arctan(*slope*) can be easily seen.

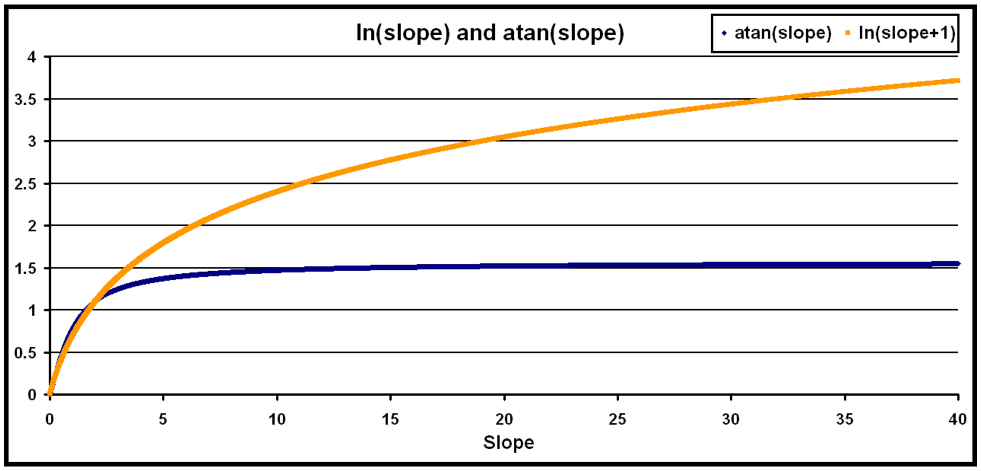


Figure 3.2. Graphs for ln(*slope*+1) and arctan(*slope*).

Both curves are nearly identical until the slope increases past approximately 2.25. After that, arctan(*slope*) soon reaches its maximum value of π/2. However, ln(*slope*+1) continues to increase and has no maximum value. The value of ln(*slope*+1) always doubles (approximately) when the slope is squared.

A potential problem with greedily merging the pair of adjacent clusters with the most similar representative slope is that local minima can prevent the best merging choice to be made. If two large, flat clusters with a slope of zero are separated by a very small cluster with a moderately high slope due to noise, the two large flat clusters will not be merged together to create one large flat cluster because merging with the very small cluster between them seems like a bad choice. Sometimes a “moderately bad” merge needs to be performed in order to set up a “very good” merge. To overcome the local minima, the algorithm should evaluate not only merging a pair of clusters, but also evaluate merging the pair clusters and then merging it with an adjacent cluster. Whichever sequence of merges has the lowest average difference in representative slope values among merged clusters is picked to be the next merge. Generally, only looking two merges ahead is enough for good results.

All sets of clusters encountered throughout the merging process can be efficiently stored in a dendrogram tree. A dendrogram is created by placing all of the sub-clusters at the leaf nodes of the tree. As each pair of clusters is merged, a new node is created to represent the new cluster and points to the two clusters that were merged to create it. The original data points only need to be stored in the leaf nodes, and the only overhead is to store two new pointers after each pair of clusters is merged. The data points contained in a cluster can be determined by finding all of the leaf clusters that are reachable by following its pointers. Recording the order that clusters were merged together enables any number of clusters between 1 and the number of initial sub-clusters to be quickly returned from the tree. The number that will be returned is determined by the final phase of the Gecko algorithm.

3.1.4 Phase 3: Determine Number of Clusters

The last phase of the Gecko algorithm is to determine the number of clusters to return. Once the correct number of clusters is determined, the set of clusters is directly extracted from a dendrogram and returned. Our L method is used to determine the number of clusters. The L method identifies the number of clusters where additional merges between the two most similar clusters begin to greatly decrease the clustering quality. The L method is general enough to be used not only for Gecko, but also for all other segmentation and hierarchical clustering algorithms. The L method is described in detail in Chapter 4, and is evaluated on both segmentation and clustering algorithms.

3.2 Empirical Evaluation

The goal of this evaluation is to demonstrate the ability of the Gecko algorithm to identify states (or clusters) in real time series data. Gecko will be compared to an existing segmentation algorithm to determine its relative performances in finding clusters in time series data. The data used to evaluate Gecko is 10 time series data sets obtained from NASA. The data sets are time series of valves on the space shuttle.

Each data set contains between 1,000 and 20,000 equally spaced measurements of current. These 10 data sets contain time series of valves operating under varying conditions.

3.2.1 Procedures and Criteria

The quality of the clusters produced by Gecko and an existing algorithm will be evaluating by having a domain expert blindly evaluate the output of each algorithm. An example of a set of clusters that is returned by Gecko is shown in Figure 3.3

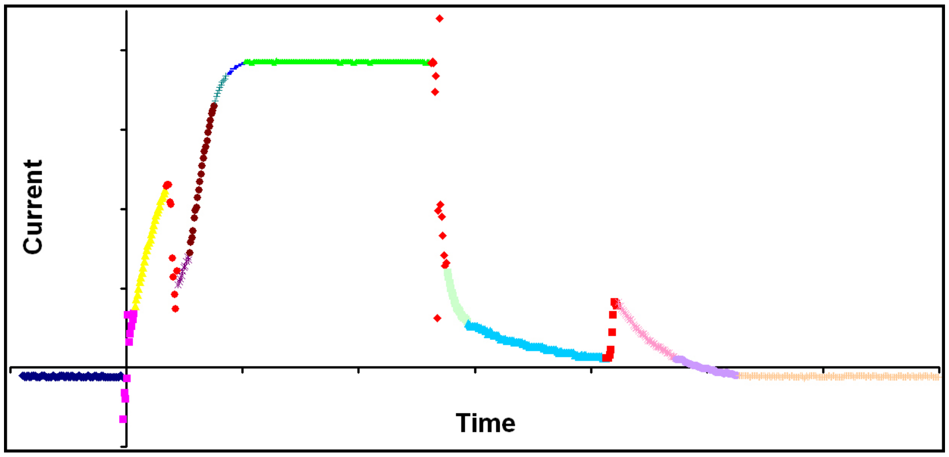


Figure 3.3. A data set after being clustered by Gecko (16 clusters).

A high quality set of clusters has each cluster corresponding to an important state in the time series. The experimental procedure is as follows: Gecko and an existing algorithm, bottom-up segmentation (BUS), cluster the 10 data sets. Without knowing which output is from which algorithm, a NASA valve expert will then rate the quality of each set of clusters from 1 to 10. BUS requires user input to determine the number of segments to return, so the number of segments returned by

BUS is set to be the same number that Gecko returns. Thus, this test is more of a test between the relative quality of the clusters produced by the two algorithms when they create the same number of clusters. BUS returns a set of line segments, but in this evaluation they are considered to be clusters where each segment is a cluster containing all of the data points within the time range of that segment. Finally, the valve expert is asked to go over each of the Gecko data sets that he rated in the second step, and explain his evaluation. Gecko was run with the default parameter for each data set: minimum cluster size *s*=10.

3.2.2 Results and Analysis

The first part of Gecko's evaluation was to compare the number of clusters it produced to the number produced by an expert human. A summary of the results is shown in Table 3.1.

Table 3.1. Number of segments found by Gecko and a human expert.

	Gecko	**NASA Human Expert**	
Data Set	*# of clusters*	*# of clusters*	*Reasonable Range*
1	16	11	9-20
2	16	10	9-20
3	14	10	9-20
4	12	10	9-20
5	13	7	(6-15)
6	10	5	(5-10)
7	7	6	(6-11)
8	16	10	(9-19)
9	16	12	(10-20)
10	15	11	(9-16)

Gecko was able to identify a number of clusters that was within the range specified by the expert to be a 'reasonable range' (for datasets 5-10 the expert did not provide a range and we extrapolated from his hand-clustering and his ranges for data sets 1-4). The human expert consistently created clusterings with fewer clusters than the Gecko algorithm. However, the clusterings are actually quite similar. Gecko identifies the same major clusters as the valve expert, but also produces several ‘transition’ clusters between them. A more detailed evaluation of the L Method’s ability to determine the number of clusters for more diverse data sets can be found in Chapter 4.

Table 3.2. Quality of segments produced by Gecko and BUS.

Data Set	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	**Avg**
Gecko	10	10	9	10	10	10	8	9	9	10	**9.5**
BUS	2	3	3	3	3	3	8	5	7	6	**4.3**

The next task performed by the NASA engineer was to rate the clusterings produced by Gecko and BUS. Table 3.2 contains the clustering quality scores for Gecko and BUS. Gecko’s average score was 9.5, while the bottom-up segmentation algorithm’s average score was only 4.3. Notice that Gecko often receives a perfect clustering score (which signifies a clustering as good as the human expert’s clustering) even though it returns fewer clusters than the human expert. For example, Gecko produced nearly twice as many clusters as the human expert for data set 5, and Gecko still got a perfect rating. This suggests that there is often a range of “very good” numbers of clusters to return, rather than a single correct number.

The final part of Gecko’s evaluation was a discussion with the NASA engineer about why he gave each score. According to the engineer, BUS divides regions of high slope into too many clusters. BUS merges clusters together by keeping the root-mean squared error of the best fit lines to a minimum. This method measures error vertically. As a consequence, lines that are nearly vertical may seem visually to be a nearly perfect fit, but the vertical distances from the points to the line can be huge. Since BUS tends to consider segments with a slope of large magnitude to have a large amount of error, it favors having more segments at this highly sloped region to reduce the overall approximation error. This is the main cause of BUS’s poor performance when rated by the domain expert.

Another advantage that Gecko had over BUS in this evaluation is that Gecko was able to identify every major transition where slopes between states changed drastically. BUS often missed these obvious cluster transitions because it initially partitions the data by creating *N*/2 clusters by initially putting every two points into a cluster. This means that wherever there is a very sharp cluster boundary, there is a 50% chance that BUS's initial segments will straddle the boundary. These small errors often cause more errors during the merging process and the overall clustering quality suffers. In contrast, the initial partitioning produced by Gecko in its first phase is careful make sure that all important cluster boundaries occur only on the edges of clusters.

Our implementation of Gecko is able to cluster a 1,000 point data sets in 7 seconds. A 20,000 point data set takes approximately 7.5 minutes to cluster. However, sampling can be performed to increase the execution time without very little effect on the quality of the output unless the user wishes to discover very small clusters that would be smoothed over by over-sampling. About 90% of the execution time is due to phase 1 of the Gecko algorithm where the initial sub-clusters are created by repeatedly bisecting a *k*-nearest neighbor graph. Building a *k*-nearest neighbour graph and recursively bisecting it is much more computationally expensive than the merging method used in the second phase.

3.3 Summary

The proposed Gecko clustering algorithm is designed to cluster time series data, and uses our proposed L method to determine a reasonable number of clusters efficiently.

Our empirical evaluations have shown that Gecko returns a set of clusters (or states) comparable to that of a human expert. Additionally, the L method used by the Gecko algorithm returns a number of clusters that is similar to the number that is generated by a human expert. When the human expert was asked to rate

Gecko’s clusterings from 1-10, Gecko’s clusterings were given perfect ratings on 6 of 10 data sets and had an average score of 9.5. A perfect rating of 10 signifies that Gecko’s clustering is equally as good as the human expert’s clustering. For comparison, the bottom-up segmentation algorithm was also tested, and was only given an average rating of 4.3.

Gecko is an improvement over existing segmentation algorithms in two ways. First, Gecko uses a relative distance function that creates a more visually appealing set of clusters than existing methods when smaller numbers of clusters are produced. Second, Gecko’s initial sub-clusters that are created in during its top-down pass in phase 1 are an improvement over existing bottom-up segmentation algorithms that initially naively create *N*/2 segments with pairs of points. This improvement in initial partitioning is not specific to the Gecko algorithm and can also be used for other bottom-up segmentation algorithms. .

The L Method, which selects the number of clusters to return in the Gecko algorithm, will be explained in the following chapter.

Chapter 4

Determining the Number of Clusters/Segments in Clustering/Segmentation Algorithms

While clustering and segmentation algorithms are unsupervised learning processes, users are usually required to set some parameters for these algorithms. These parameters vary from one algorithm to another, but most clustering and segmentation algorithms require a parameter that either directly or indirectly specifies the number of clusters/segments. This parameter is typically either *k*, the number of clusters/segments to return, or some other parameter that indirectly controls the number of clusters to return, such as an error threshold. Setting these parameters requires either detailed pre-existing knowledge of the data, or time-consuming trial and error. The latter case still requires that the user has sufficient domain knowledge to know what a good clustering "looks" like. However, if the data set is very large or is multi-dimensional, human verification could become difficult. To automatically find a reasonable number of clusters, many existing methods must be run repeatedly with different parameters, and are impractical for real-world data sets that are often quite large.

We wish to develop an algorithm that can automatically and efficiently determine a reasonable number of clusters/segments to return from any hierarchical clustering/segmentation algorithm. In the previous chapter, the Gecko algorithm made use of the L method to automatically select the number of clusters to return. This chapter will explain the L method in detail and evaluate it on several clustering and segmentation algorithms using multiple data sets.

Section 4.1 gives and overview of the L method. Section 4.2 contains an empirical evaluation of the L method on diverse data sets for three clustering algorithms and three segmentation algorithms. The performance of the L method is also evaluated against two existing methods to determine the number of clusters or segments in a data set. Section 4.3 summarizes our study of the L method.

4.1 The L Method

In order to identify the correct number of clusters to return from a hierarchical clustering/segmentation algorithm, we introduce the L method. Hierarchical algorithms either merge the two most similar clusters together (bottom-up), or split the least internally homogeneous cluster into two (top-down). The definition of a “cluster” is not well-defined, and measuring cluster quality is rather subjective. Thus, there are many clustering algorithms with unique evaluation functions and correspondingly unique notions of what a good cluster “looks” like. The L method makes use of the same evaluation function that is used by a hierarchical algorithm during clustering or segmentation to construct an evaluation graph where the *x*-axis is the number of clusters and the *y*-axis is the value of the evaluation function during the merge or split at *x* clusters. The knee, or the point of maximum curvature of this graph, is used as the number of clusters to return. The knee is determined by finding the area between the two lines that most closely fit the curve. The L method only needs the clustering/segmentation algorithm to be run once, and the overhead of determining the number of clusters is trivial compared to the runtime of the clustering/segmentation algorithm.

4.1.1 Evaluation Graphs

The information required to determine an appropriate number of clusters/segments to return is contained in an evaluation graph that is created by the clustering/segmentation algorithm. The evaluation graph is a two-dimensional plot where the *x*-axis is the number of clusters, and the *y*-axis is a measure of the quality or error of a clustering consisting of *x* clusters. Some approaches use similar graphs that they are often generated by re-running the entire clustering or segmentation algorithm for every value on the *x*-axis, which is quite inefficient. Since hierarchical algorithms either split or merge a pair of clusters at each step, all clusterings containing '*1*' to '*the number of clusters in the fine-grain clustering*' clusters can be produced by running the clustering algorithm only once.

The *y*-axis values in the evaluation graph can be any evaluation metric, such as: distance, similarity, error, or quality. These metrics can be computed globally or greedily. Global measurements compute the evaluation metric based on the entire clustered data set. A common example is the sum of all the pairwise distances between points in each cluster. Most global evaluation metrics are computed in $O(N^2)$ time, where N is the number of points in the data set. Thus, in many cases, it takes longer to evaluate a single set of clusters than it takes to create them. Since the evaluation function must be run for every potential number of clusters, this method is too inefficient. The alternative is to use greedy measurements. The greedy method works in hierarchical algorithms by evaluating only the two clusters that are involved in the current merge or split, rather than the entire data set.

Many "external" evaluation methods attempt to determine a reasonable number of clusters by evaluating the output of an arbitrary clustering algorithm. Each evaluation method has its own notion of cluster quality. Most external methods use pairwise-distance functions that are heavily biased towards spherical clusters. Such methods would be unsuitable for a clustering algorithm that has a different notion of cluster distance/similarity. For example, Chameleon (Karypris, Hun & Kumar 1999) uses a complex similarity function that can produce interesting non-spherical clusters, and even clusters within clusters. Therefore, the

L method is integrated into the clustering algorithm and the metric used in the evaluation graph is the same metric used in the clustering algorithm.

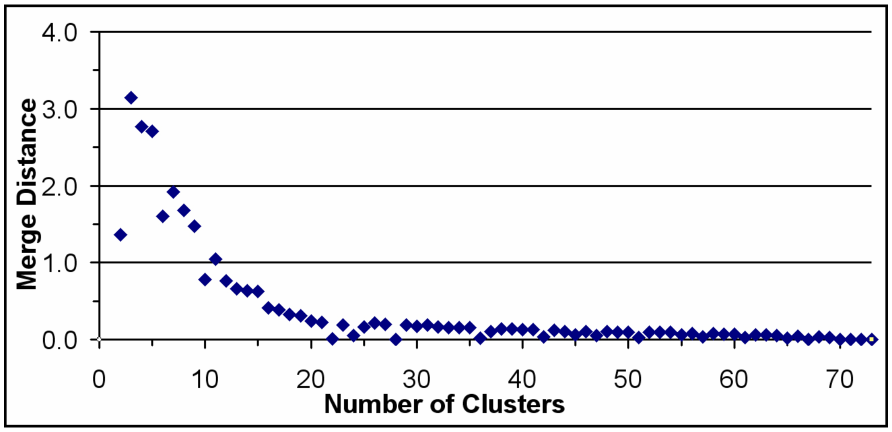


Figure 4.1. A sample evaluation graph.

An example of an evaluation graph produced by the Gecko segmentation algorithm (discussed in the previous chapter) is shown in Figure 4.1. The *y*-axis values are the distances between the two clusters that are most similar at *x* clusters. This is a greedy approach, since only the two closest clusters being merged are used to generate the value on the *y*-axis. The curve in Figure 4.1 has three distinctive areas: a rather flat region to the right, a sharply-sloping region to the left, and a curved transition area in the middle.

In Figure 4.1, starting from the right, where the merging process begins at the initial fine grain clustering (for a bottom-up hierarchical algorithm), there are many very similar clusters to be merged and the trend continues to the left in a rather straight line for some time. In this region, many clusters are similar to each other and should be merged. Another distinctive area of the graph is on the far left side where the merge distances grow very rapidly (moving right to left). This rapid increase in distance indicates that very dissimilar clusters are being merged together, and that the quality of the clustering is becoming poor because clusters are no longer internally homogeneous. If the best available remaining merges start becoming increasingly poor, it means that too many merges have already been performed. A reasonable number of clusters is therefore in the curved area, or the “knee” of the graph. This knee region is between the low distance merges that form a nearly straight line on the right side of the graph, and the quickly increasing region on the left side. Clusterings in this knee region contain a balance of clusters that are both highly homogeneous, and also dissimilar to each other. Determining the number of clusters where this knee region exists will therefore give a reasonable number of clusters to return.

Locating the exact location of the knee, and along with it the number of clusters, would seem problematic if the knee is a smooth curve. In such an instance, the knee could be anywhere on this smooth curve, and thus the number of clusters to be returned seems imprecise. Such an evaluation graph would be produced by a data set with clusters that are overlapping and not very well separated. Time series data is usually a continuous function that does not contain data that is well-separated. In such instances, there is no single ‘correct’ answer and all of the values along the knee region are likely to be reasonable estimates of the number of clusters. Thus, an ambiguous knee indicates that there probably is no single ‘correct’ answer, but rather a range of acceptable answers.

4.1.2 Finding the Knee via the L Method

In order to determine the location of the transition area or knee of the evaluation graph, we take advantage of a property that exists in these evaluation graphs. The regions to both the right and the left of the knee (see Figure 4.2) are often approximately linear. If a line is fitted to the right side and another line is fitted to the left side, then the area between the two lines will be in the same region as the knee. The value of the *x*-axis at the knee can then be used as the number of clusters to return. Figure 4.2 depicts an example.

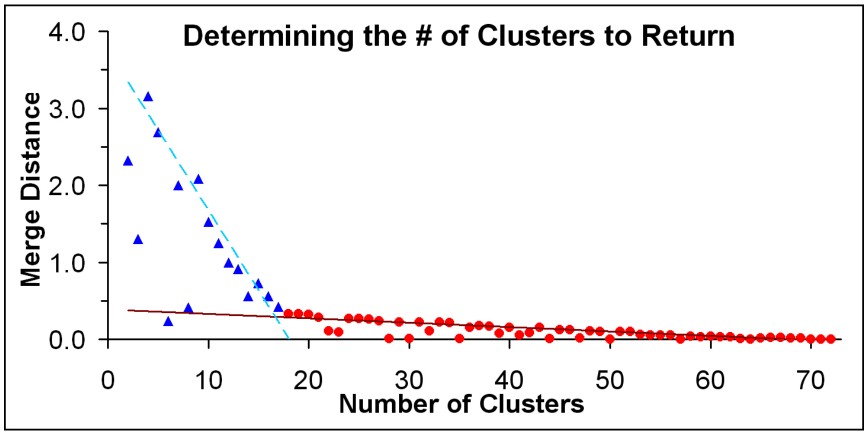


Figure 4.2. Finding the number of clusters using the L method.

To create these two lines that intersect at the knee, we will find the pair of lines that most closely fit the curve. Figure 4.3 shows all possible pairs of best-fit lines for a graph that contains seven data points (eight clusters were repeatedly merged into a single cluster). Each line must contain at least two points, and must start at either end of the data. Both lines together cover all of the data points, so if one line is small, the other is large to cover the rest of the remaining data points. The lines cover sequential sets of points, so the total number of line pairs is *numOfInitialClusters*-4. Of the four possible line pairs in Figure 4.3, the pair that fits their respective data points with the minimum amount of error is the pair on the bottom left. Our approach to finding these two lines is essentially an optimal segmentation algorithm for finding two segments (*k*=2).

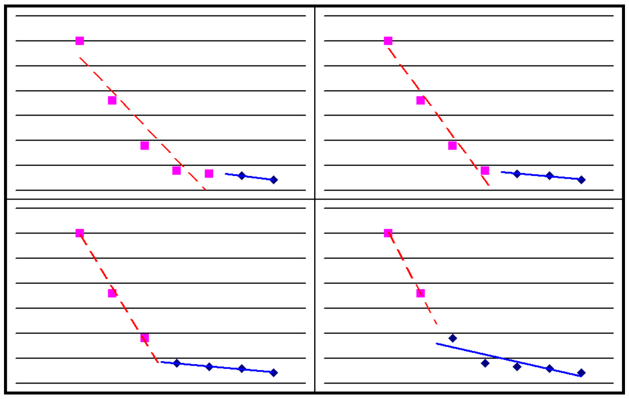


Figure 4.3. All four possible pairs of best-fit lines for a small evaluation graph.

Consider a '# of clusters vs. evaluation metric' graph with values on the *x* axis up to *x*=*b*. The *x*-axis varies from 2 to *b*, hence there are *b*-1 data points in the graph. Let L_c and R_c be the left and right sequences of data points partitioned at *x*=*c*; that is, L_c has points with *x*=2...*c*, and R_c has points with *x*=*c*+1…*b*, where *c*=3…*b*-2. Equation 1 defines the total root mean squared error $RMSE_c$, when the partition of L_c and R_c is at *x*=*c*:

$$RMSE_c = \frac{c-1}{b-1} \times RMSE(L_c) + \frac{b-c}{b-1} \times RMSE(R_c) \qquad [1]$$

where $RMSE(L_c)$ is the root mean squared error of the best-fit line for the sequence of points in L_c (and similarly for R_c). The weights are proportional to the lengths of L_c (*c*-1) and R_c (*b*-*c*). We seek the value of *c*, *c**, such that $RMSE_c$ is minimized, that is:

$$c^* = \arg\min_c \quad RMSE_c \qquad [2]$$

The location of the knee at *x*=*c** is used as the number of clusters to return.

In our evaluation, the L method determined the number of clusters in only 0.00004% to 0.9% of the execution time required by the clustering algorithm. The time it takes for the L method to execute directly corresponds to the number of points in the evaluation graph. Since the number of points in the evaluation graph is controlled by the number of clusters at the finest grain clustering, the L method runs much faster for clustering algorithms that do not have an overly-fine initial clustering.

The L method is very general and contains no parameters or constants. The number of points along the *x*-axis of the evaluation graph is not a parameter. It is a result of the clustering algorithm used to generate those points. The maximum *x* value in the evaluation graph is either the number of clusters at the initial fine grain clustering in a bottom-up algorithm, or the number of clusters in the final clustering in a top-down algorithm.

4.1.3 Iterative Refinement

Some bottom-up algorithms create an initial fine-grain clustering by initially treating every data point as a cluster. This can cause an evaluation graph to be as large as the original data set. If such an evaluation graph has thousands of merge values, the ones representing merges at extremely fine-grain clusterings (large values of *x*) are irrelevant. Such a large number of irrelevant data points in the evaluation graph can prevent an "L" shaped curve, or more specifically a flat region to the right of the knee.

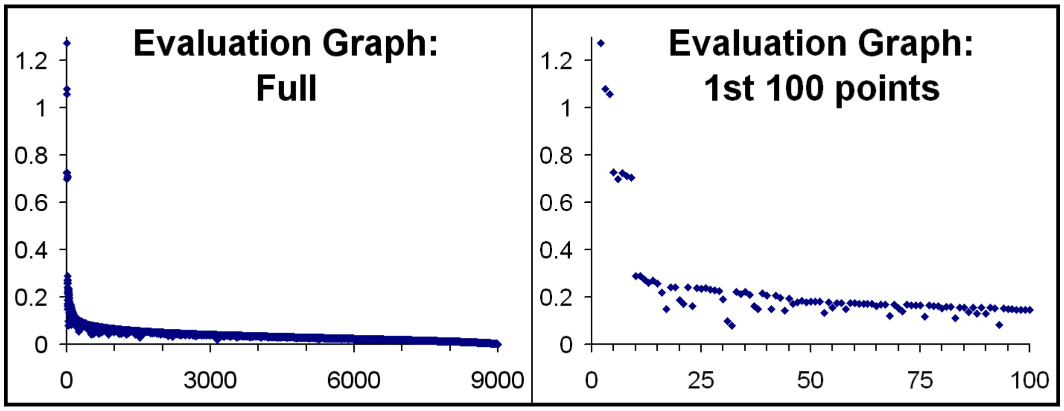


Figure 4.4. Full and partial evaluation graphs created by CURE. Only the first 100 points are shown on the right side.

Figure 4.4 shows a 9,000 point evaluation graph on the left, and the first 100 data points of the same graph on the right. The graph on the right is a more natural "L" shaped curve, and the L method is able to correctly identify that there are 9 clusters in the data set. However, in the full evaluation graph, there are so many data points to the right side of the "correct" knee, that the very few points on the left of that knee become statistically irrelevant. The L method performs best when the sizes of the two lines on each side of the knee are reasonably balanced. When there are far too many points on the right side of the actual knee, the knee that is located by the L method will most likely be larger than the actual knee. In the full evaluation graph, containing 9,000 data points, the knee is incorrectly detected at *x*=359, rather than *x*=9. However, when many of the irrelevant points are removed from the evaluation graph, such as all points greater than *x*=100 (see the right side of Figure 4.4), the correct knee is located at *x*=9. The following algorithm shown in Figure 4.5 iteratively refines the knee by adjusting the focus region and reapplying the L method (note that the clustering algorithm is *not* reapplied).

Iterative Refinement of the Knee

Input: *evalGraph* (a full evaluation graph)
Output: the *x*-axis value location of the knee (also the suggested number of clusters to return)

```
 1|  int cutoff =
 2|      lastKnee =
 3|      currentKnee = EvalGraph.size()
 4|
 5|  REPEAT
 6|  {
 7|    lastKnee = currentKnee
 8|    currentKnee = LMethod(evalGraph,cutoff)
 9|    cutoff = currentKnee*2
10|  } UNTIL currentKnee ≥ lastKnee
11|
12|  RETURN currentKnee
```

Figure 4.5. Pseudocode to Iterative Refine the knee with the L Method.

This algorithm initially runs the L method on the entire evaluation graph. The value of the knee becomes the middle of the next focus region and the L method becomes more accurate because the lines on each side of the true knee are becoming more balanced. Since the refinement stops when the knee does not move to the left after an iteration, the focus region decreases in size after every iteration (except the final iteration). The true knee is located when the L method returns the same value as the previous iteration (line #10, or if the current pass returns a knee that has a roughly balanced number of points on each side of the knee (also line #10). The 9,000 point evaluation graph in Figure 4.4 takes four iterations to correctly determine that there are 9 clusters in the data set (9,000 $\rightarrow$ 359 $\rightarrow$ 15 $\rightarrow$ 9 $\rightarrow$ 9). The cutoff value is not permitted to drop below ~20 in the “LMethod(),” because a reasonable number of points are needed for the two fitted-lines to fit actual trends, rather than detecting spurious trends indicated by a small number of points in the evaluation graph. The minimum cutoff size of 20 performed well on all tests that have been run to date and it will most likely never need to be changed. The minimum cutoff size can therefore most likely be treated as a constant rather than a parameter (keeping the L method ‘parameterless’).

Iteratively refining the knee does not significantly increase the execution time of the L method. Iterative refinement converges on the knee in very few iterations (usually less than three), and the first iteration is run with an evaluation graph that is much larger than those in later iterations. The L method is an O(N^2) algorithm with respect to the size of the evaluation graph. This means that the vast majority of the execution time is during the first iteration, when the evaluation graph is much larger. Evaluation graphs with fewer than 1,000 points can be evaluated in less than a few seconds; however, a 9,000 point evaluation graph takes several minutes. In practice, it is usually permissible to ignore points in an evaluation past some large number when it is unlikely (or undesirable) for such a large number of clusters to exist in the data set.

4.1.4 Refinements for Segmentation Algorithms

Evaluation graphs for segmentation algorithms can often be very jumpy when segmenting noisy data. The exact nature of the curve may be easy to determine visually, but there can be a great number of points that do not fit the curve. These stray points on the evaluation graph generally do not occur consecutively. These stray points can prevent the L method from accurately locating the knee. However, because they do not usually occur consecutively, the curve can be smoothed by only using the highest valued point of every consecutive pair when computing the best-fit lines of the curve.

Another potential problem is that sometimes the evaluation graph will reach a maximum (moving from right to left) and then start to decrease. This can be seen in Figure 4.2, where the distance between the closest segments reaches a maximum at x=4. This can prevent an “L” shaped curve from existing in the evaluation graph. The data points to the left of the maximum value (the ‘worst’ merge) can be ignored. This occurs in some algorithms that have distance functions that become undefined when the remaining clusters are extremely dissimilar to each other.

4.2 Empirical Evaluation

The goal of this evaluation is to demonstrate the ability of the L method to identify a reasonable number of clusters to return in hierarchical clustering and hierarchical segmentation algorithms. Each algorithm will be run on a number of data sets and the number of clusters that the L method identifies is compared to the ‘correct’ answer. Existing methods to determine the number of segments or clusters in a data set will also be evaluated on the same data sets, and their performance will be compared to that of our L method. Section 4.2.1 evaluates the L Method for Clustering algorithms, and section 4.2.1 evaluates the L Method on segmentation algorithms.

4.2.1 Identifying the Number of Clusters

In this section, the L Method and an existing method will be evaluated with hierarchical clustering algorithms on synthetic two dimensional data sets. The data sets are synthetic and have a “correct” number of clusters to compare to the number of clusters identified by the L method.

4.2.1.1 Procedures and Criteria

The seven diverse data sets used to evaluate the L method for clustering algorithms vary in size, number of clusters, separation of clusters, density, and amount of outliers. There are some data sets that contain only spherical clusters, and some which contain very non-spherical clusters, including clusters within clusters. The seven spatial data sets that were used are (see Figure 4.6):

1. A data set with four well separated spherical clusters (4,000 pts).
2. Nine square clusters connected at the corners (9,000 pts).
3. Ten spherical clusters. Five overlapping clusters similar to data set #7 (not shown), as well as five additional well separated clusters and a uniform distribution of outliers (5,200 pts).
4. Ten well separated clusters of varying size and density (5,000 pts).
5. A 9 cluster data set used in the Chameleon paper, but with the outliers removed. Non-spherical clusters with clusters completely contained within other clusters (~9,100 pts).
6. An 8 cluster data set used in the Chameleon paper, but with the outliers removed. Non-spherical clusters with clusters partially enveloping other clusters (~7,600 pts).
7. Five spherical clusters of equal size and density. The clusters are all close to each other and slightly overlapping (5,000 pts, not in Figure 4.6).

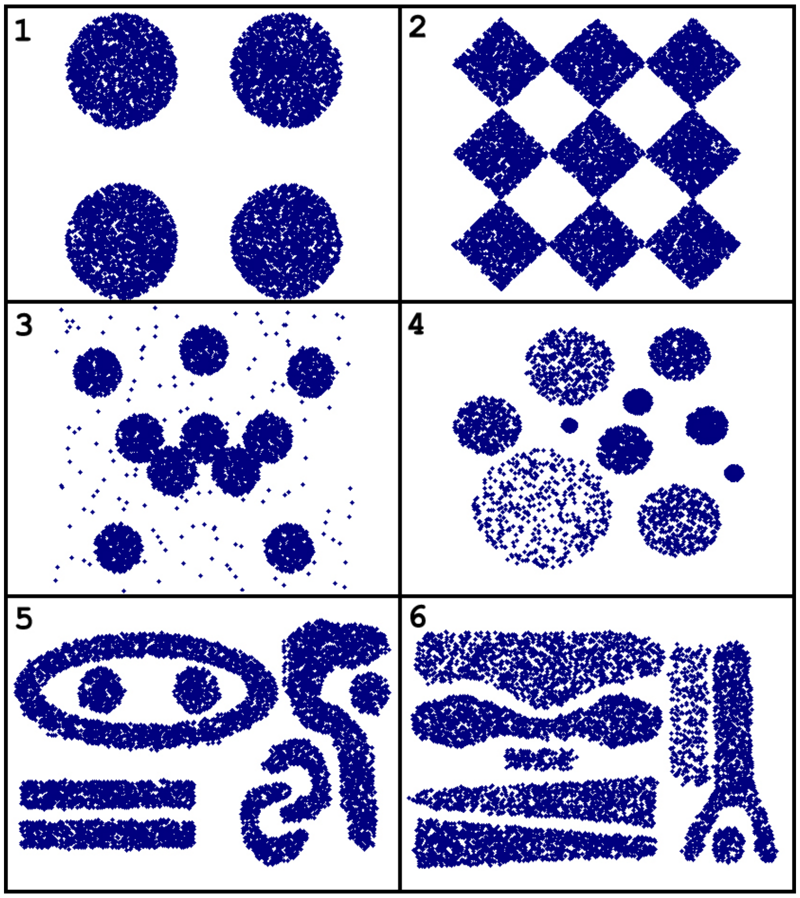


Figure 4.6. Data sets 1-6 for evaluating the L method in clustering algorithms (data set #7 not shown).

The clustering algorithms used to test the L method were Chameleon and CURE. Chameleon was implemented locally and was run with the parameters: *k*=10 (*k* nearest neighbors, not *k* clusters), *minSize*=3%, and α=2. CURE was implemented as specified in the CURE paper (Guha, Rastogi & Shim 1998), with the shrinking factor set to 1/3 and the number of representative points for each cluster set to 10.

CURE, Chameleon, and a standard implementation of *K*-means was used to evaluate the Gap Statistic's relative performance against that of the L method. The Gap Statistic was calculated using the Gap/unf variant (Tibshirani, Walther & Hastie 2003). The Gap statistic must be run for each user-specified potential value for the number of clusters. The potential number of clusters evaluated by the Gap statistic were *k*={2…20}. Both the L method and the Gap statistic were tested on CURE and Chameleon for a direct comparison. However, the Gap statistic was also evaluated on the *K*-means algorithm, because *K*-means was used by Tibshirani, Walther, and Hastie (2003) to evaluate the Gap statistic. It is important to note that for every different clustering algorithm, the L method's evaluation graph contains values (on the *y*-axis) created by the particular clustering algorithm's evaluation function, while the Gap statistic uses pairwise distances to evaluate clusters regardless of the clustering algorithm that produced them. Thus, the L method's performance is more likely to be consistent over different clustering algorithms, while the Gap statistic will only work well for clustering algorithms that measure cluster quality similar to its own fixed method.

The experimental procedure for evaluating the performance of the L method for hierarchical clustering algorithms consists of running the CURE and Chameleon clustering algorithms, which have been modified to automatically determine the number of clusters to return through use of the L method, on seven diverse data sets (shown in Figure 4.6). The number of clusters automatically returned will be compared to the correct number of clusters. The data sets are synthetic, so the correct number of clusters is known. These results will also be compared to the number of clusters suggested by the existing Gap statistic for CURE, Chameleon, and also the *K*-means clustering algorithm.

4.2.1.2 Results and Analysis

The correct number of clusters was determined by the L method 6 out of 7 times for Chameleon and 4 out of 5 times for the CURE algorithm. The results are contained in Table 4.1. The actual number of clusters suggested for CURE on data set #3 was 9. However, in the presence of outliers, CURE creates a number of very small clusters that contain only outliers. After removing these small clusters, only six clusters remained. Data sets #5 and #6 contain complex clusters and could only be properly clustered by Chameleon; "N/A" is placed in the cells of Table 4.1 where the number of clusters suggested was not evaluated because the clustering algorithm was unable to produce the correct set of clusters.

Table 4.1. Results of using the L method and the Gap statistic with various clustering algorithms.

Data Set		**Number of Clusters Predicted by L Method**		**Num of Clusters Predicted by Gap Statistic**		
data set	*correct number of clusters*	*Cham-eleon*	*CURE*	*Cham-eleon*	*CURE*	*K-means*
1	4	4	4	4	4	4
2	9	9	9	2	2	2
3	10	11	6 (9)	2	2	2
4	10	10	10	2	2	2
5	9	9	N/A	2	N/A	N/A
6	8	8	N/A	2	N/A	N/A
7	5	5	5	2	2	2
Exact Matches		6 of 7	4 of 5	1 of 7	1 of 5	1 of 5

The Gap statistic was only able to determine the correct number of clusters for one of the seven data sets, regardless of the clustering algorithm used. The Gap statistic performs similarly to many existing methods (Tibshirani, Walther & Hastie 2003), and only works well for well-separated, circular clusters (only data set #1 satisfies these constraints). The Gap statistic tended to suggest far too few clusters because the cluster separation was not great enough for it to consider the clusters to be distinct.

The correct number of clusters was not determined for either algorithm on data set #3, which contained many outliers and a mixture of both well separated and overlapping clusters. In the evaluation graph for CURE, there is a large smooth knee that spans approximately 200 data points. Most of the merges in this region are between outliers, but there are also merges of the five overlapping clusters mixed in. There is no sharp knee until after all of the five overlapping clusters have already been merged together. The clusters returned by CURE were not ‘correct’, but they weren’t too bad either. The six clusters returned by the L method were the five well separated clusters, and the group of overlapping clusters in the center (see data set #3 in Figure 4.6). Even though the L method recommends four fewer clusters than the ‘correct’ answer, the six recommended clusters have a more uniform separation than the ‘correct’ answer. In this case, the best number of clusters is open to interpretation. The answer given for Chameleon on data set #3 was off by one because the knee of the curve was not sharp enough for the L method to identify the exact number of clusters. This is most likely due to a weakness in our Chameleon implementation, which does not contain a graph bisection algorithm that is as powerful as the one described in the Chameleon (Karypris, Hun & Kumar 1999) paper.

The L method determines the number of clusters to return by examining the evaluation graphs produced by each clustering algorithm. Examples of evaluation graphs are shown in Figure 4.7, where the *x*-axis is the number of clusters, and the *y*-axis is the value of the clustering algorithm’s evaluation function at *x* clusters. Notice that the *y*-axis values in CURE evaluation graphs generally increase from right to left, while the Chameleon evaluation graphs generally decrease from right to left. This is because CURE’s evaluation metric measures distance and Chameleon’s evaluation metric measures similarity.

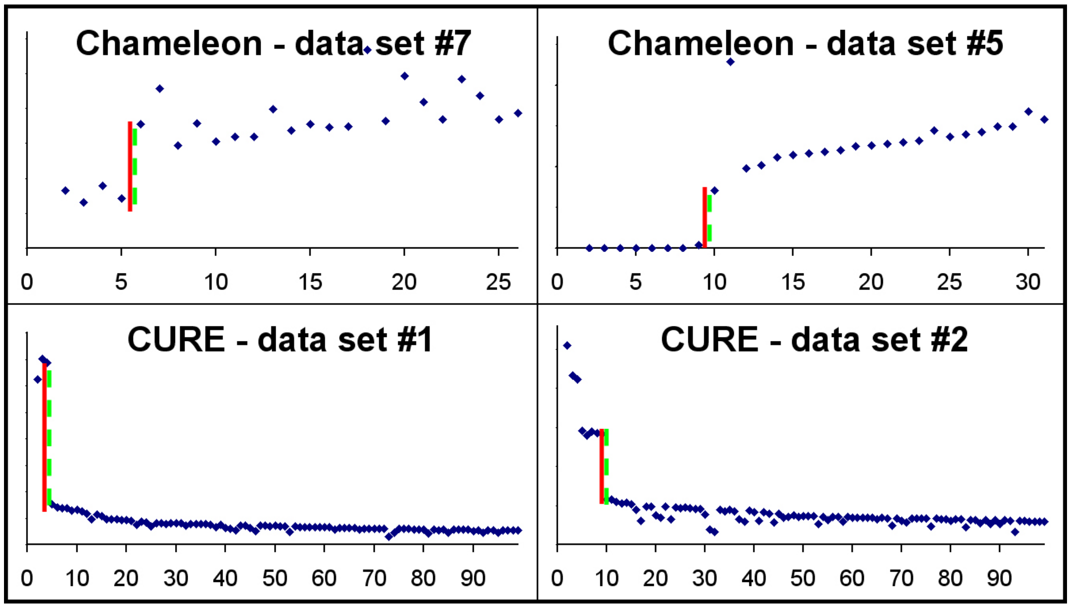


Figure 4.7. Actual number of clusters and the correct number predicted by the L method (axes: *x*= # of clusters, *y*=evaluation metric – lines: solid lines=correct # of clusters, dashed lines=# of clusters determined by L method).

In Figure 4.7, the solid line indicates where the correct number of clusters is, while the dashed line indicates the number of clusters suggested by the L method. The lines are directly next to each other in each case which indicates that the correct number of clusters was determined. The best number of clusters is usually just before a large jump in the evaluation graph. To the left of the jump dissimilar clusters have been merged together creating inhomogeneous clusters; and to the right of the jump there are too many clusters that are similar to each other. Since a good cluster is loosely defined to be one that is both internally homogeneous and dissimilar to other clusters, the location of the jump should be a good measure of the best number of clusters.

The L method runs more quickly for clustering algorithms that do not have an overly-fine initial clustering, because these algorithms have smaller evaluation graphs. Chameleon initially produces fine grain clusterings that contain fewer than 100 clusters and the L method needs less than 0.01 seconds to determine the number of clusters. CURE produces the finest initial clustering possible, which creates evaluation graphs with up to 8,999 points in our evaluation. With CURE, the L method's run-time is between 40 seconds and 4 minutes. This execution time can be drastically reduced, to a much less than one second, by only evaluating the first *maxk* points in the evaluation graph, where *maxk* is some large number guaranteed to be more than the actual number of clusters. In our evaluation, the L method determined the number of clusters to return in less than 0.9% of the total execution time for CURE and less than 0.003% of the total execution time for Chameleon. Runtime for the Gap statistic is significantly slower. The Gap statistic must be run for each potential number of clusters, and since it calculates pairwise distances within a cluster, its run-time (to evaluate just a single potential number of clusters) approaches $O(N^2)$. In our evaluation, the Gap statistic took up to 28 minutes to evaluate the clusterings in the range $k=\{2 \ldots 20\}$, and took several times longer to evaluate each clustering than the clustering algorithm needed to produce it.

4.2.2 Identifying the Number of Segments

In this section the L Method and an existing method will be evaluated on hierarchical segmentation algorithms and one-dimensional time series data. The L Method will be used by three segmentation algorithms on seven data sets. Three data sets are synthetic and have a single "correct" answer, while the other four are real data sets and have a range of reasonable numbers of segments to compare to the number of segments identified by the L method

4.2.2.1 Procedures and Criteria

The experimental procedure for evaluating the L method in segmentation algorithms consists of running two different segmentation algorithms on seven different data sets and determining if a 'reasonable' number of segments is suggested by the L method. This number of segments suggested will then be compared to the ‘correct’ number of segments, and also the number suggested by the existing permutation tests method (Vasko & Toivonen 2002).

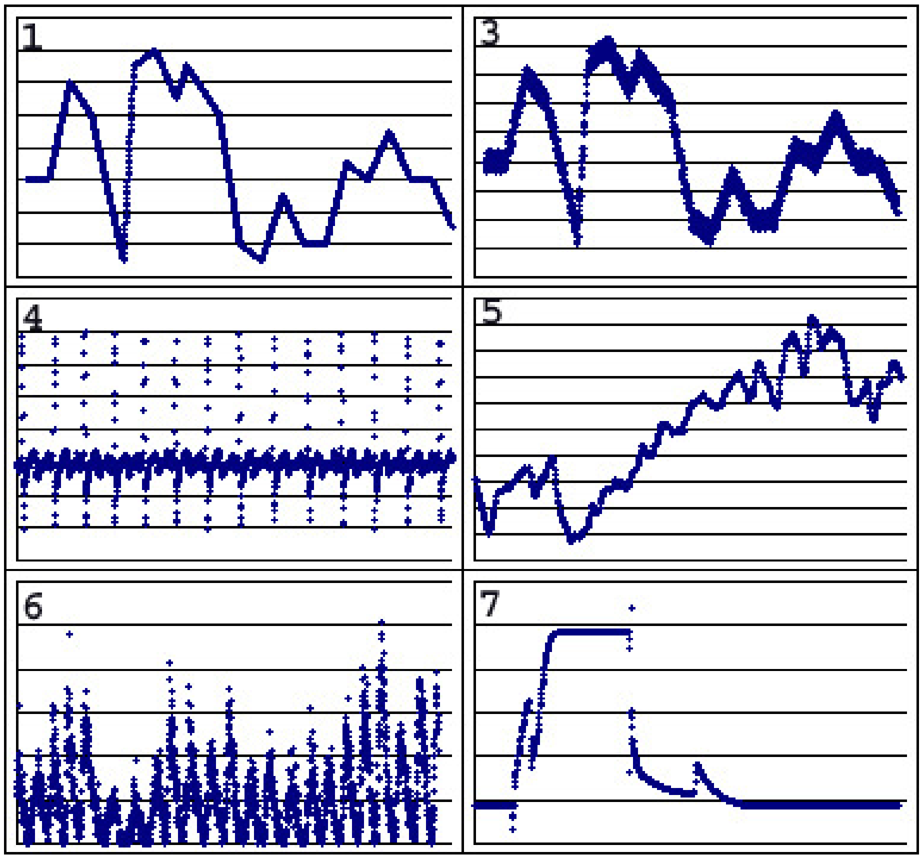


Figure 4.8. Data sets 1, 3, 4, 5, 6, and 7 for evaluating the L method in segmentation algorithms.

The time series data sets used to evaluate the L method for hierarchical segmentation algorithms are a combination of both real and synthetic data. The seven time series data sets used for this evaluation (shown in Figure 4.8) are:

1. A synthetic data set consisting of 20 straight line segments (2,000 pts).
2. The same as #1, but with a moderate amount of random noise added (2,000 pts, not in Figure 4.8).

3. The same as #1, but with a substantial amount of random noise added (2,000 pts).

4. An ECG of a pregnant woman from the Time Series Data Mining Archive (Keogh & Folias 2004). It contains a recurring pattern (a heart beat) that is repeated 13 times (2,500 pts).

5. Measurements from a sensor in an industrial dryer (from the Time Series Data Mining Archive). The time series appears similar to random walk data (876 pts).

6. A data set depicting sunspot activity over time (from the Time Series Data Mining Archive). This time series contains 22 roughly evenly spaced sunspot cycles, however the intensity of each cycle can vary significantly (2,900 pts).

7. A time series of a space shuttle valve energizing and de-energizing (1,000 pts).

A 'correct' number of segments for a particular data set and segmentation algorithm is obtained by running the algorithm with various values of *k* (controls the number of segments returned), and determining what particular value or range of values of *k* produces a 'reasonable' PLA (piecewise linear approximation). The PLAs that are considered 'reasonable' are those at a value of *k*, where no adjacent segments are very similar to each other and all segments are internally homogeneous (segments have small error). The synthetic data sets have a single correct value for *k*. The real sets have no single correct answer, but rather a range of reasonable values. The reasonable and best numbers of segments for the real data sets may vary for each algorithm. A single 'best' number of segments cannot be used for all of the segmentation algorithms because one number that produces the best set of segments for one algorithm may produce a poor set of segments for another on the same data set.

The segmentation algorithms used in this evaluation were Gecko (discussed in Chapter 3) and bottom-up segmentation (BUS). BUS (bottom-up segmentation) is a hierarchical algorithm that initially creates many small segments and repeatedly joins adjacent segments together. More specifically, BUS evaluates every pair of adjacent segments and merges the pair that causes the smallest increase in error when they are merged together. BUS was tested with the L method using two different values on the *y*-axis of the evaluation graph. The two variants are named BUS-greedy and BUS-global. BUS-greedy's *y*-axis in the evaluation graph is the increase in error of the two most similar segments when they are merged, and BUS-global's *y*-axis is the error of the entire linear approximation when there are *x* segments (absolute error). The existing 'permutation tests' method was also evaluated using BUS.

Both Gecko and BUS made use of an initial top-down pass to create the initial fine-grain segments. The minimum size of each initial segment generated in the top down pass was 10. For the permutation test algorithm, *p* was set to 0.05, and 1,000 permutations were created. The parameter *p* controls the percentage of permutated time series that must be increasing in quality faster than the original time series to stop creating more segments.

4.2.2.2 Results and Analysis

A summary of the results of the L method's and permutation tests' ability to automatically determine the number of segments to return from segmentation algorithms is contained in Table 4.2. For both Gecko and BUS, the 'reasonable' range of correct answers is listed. These ranges may vary between the two algorithms because BUS and Gecko do not merge segments in exactly the same sequence. However, BUS-greedy, BUS-global, and permutation tests all produce identical PLAs for *k* segments, and therefore have identical 'reasonable' answers. The first three data sets are synthetic and have a single correct answer, but the other data sets have a range of ‘reasonable’ answers. Data set #5 is similar to random walk data, and any number of segments seemed reasonable because there was no underlying structure in the time series.

Table 4.2. Results of using the L method with three hierarchical segmentation algorithms.

	Gecko		**Bottom-up Segmentation**			
	Gecko w/ L method			**BUS-greedy** w/ L method	**BUS-global** w/ L method	**BUS** w/ permutation Tests
Data Set	*Reasonable # of segments*	*Number of segments found*	*Reasonable # of segments*	*Number of segments found*	*Number of segments found*	*Number of segments found*
1	20	20	20	20	20	25
2	20	20	20	20	20	34
3	20	N/A	20	20	19	25
4	42-123	92	42-123	46	106	2
5	?	32	?	14	39	15
6	44-57	45	45-53	48	39	6
7	9-20	17	14-21	9	13	65
Reasonable -Range Matches	▨	5 of 5	▨	5 of 6	3 of 6	0 of 6

The L method worked very well for both BUS-greedy and Gecko. It correctly identified a number of segments for BUS-greedy that was within the reasonable range in 5 out of the 6 applicable data sets. Gecko, which also uses a greedy evaluation metric (but uses slope rather than segment error), had the L method suggest a number of segments within the reasonable range for all 5 applicable data sets. Gecko was unable to correctly segment data set #3 (indicated by “N/A” in Table 4.2) because it contained too much noise. In all but one test case (10 of 11), the L method was able to correctly determine that the three synthetic data sets contained exactly twenty segments. BUS-global did not perform quite as well. The L method was only able to return a reasonable number of segments for BUS-global in half of its test cases, but all of its incorrect answers were close to being correct.

Permutation tests did not perform well and never determined a reasonable number of segments. The reason that permutation tests did poorly varied depending on the data set. Data set #1 is synthetic and contains no noise, which allows a PLA to approximate it with virtually zero error. However, measuring a relative increase in error when the error is near zero causes unexpected results because relative increases are either very large or undefined when the error is at or near zero. For data set #4 and #6, the relative change in approximation error is rather constant regardless of the number of segments. On data set #4, the PLA between 2 and 3 segments has nearly zero relative change in error, which causes permutation tests to incorrectly assume that the data has been over-fitted and stop producing segments prematurely. An example of far too many segments being returned occurs on data set #7, where the relative error of the time series never falls below the relative error of the permutations until far too many segments are produced.

Some of the evaluation graphs used by the L method for Gecko, BUS-greedy, and BUS-global are shown in Figure 4.9. The lower left portion of Figure 4.9 contains the L method's evaluation graph for Gecko on data set #1, the noise-free synthetic data set. The *x*-axis is the number of segments, and the *y*-axis is Gecko's evaluation metric at *x* segments (distance between two closest adjacent segments when there are *x* segments). The evaluation graph is created right to left as segments are meged together. In this case, the correct number of segments is easily determined by the L method because there is a very large jump at *x*=20. In the lower right corner of Figure 4.9, the range of correct answers lies between the two long lines. The range is larger than for data set #1 because the segments have less ‘separation’ and there is no sharp knee. Instead there is a range of good answers. However, the L method suggests a number of segmetns that just misses the reasonable range.

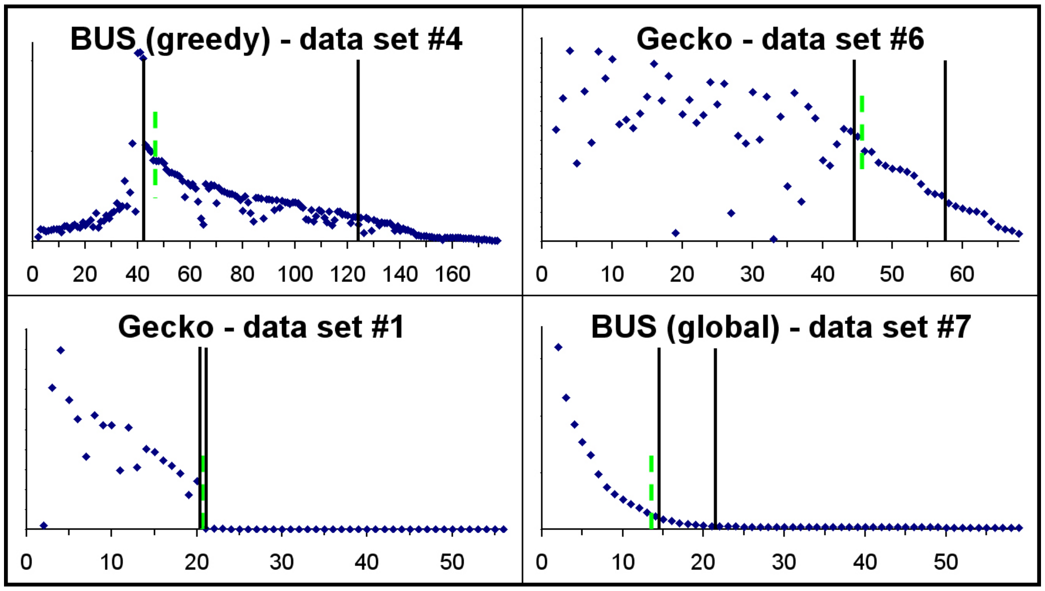


Figure 4.9. The reasonable range for the number of segments and the number returned by the L method. (axes: *x*=# of segments, *y*=evaluation metric – short dashed line=# of segments determined by the L method, long solid lines=marks the boundaries of the reasonable range for the # of segments.

In the evaluation graph in the top left of Figure 4.9 (data set #4 BUS-greedy), the L method returned a number of segments that was towards the low end of the reasonable range. Remember, that for segmentation algorithms, all data ponits to the left of the data point with the maximum value are ignored (discussed in the last section of 3.3). The best number of segments is 42. At 42 segments each heart beat contains approximately 3 segments. If there are fewer than 42 segments, they are no longer homogeneous. However, PLAs with significantly more segments (up to 123) are still reasonable because each new segment still significantly reduces the error. However, if there are more than approximately 123 segments, adjacent segments start to become too similar to each other.

The evaluation graph shown in the upper-right portion of Figure 4.9 also has ‘better’ PLAs when the number of segments is near the low end of the reasonable range (fewer segmetns). This is common because the best set of segments is often the minimal set of segments that adequately represents the data. Even though there is apparently no significant knee in this evaluation graph, a good number of segments can still be found by the L method. This is because the knee found by the L method does not necessarily have to be the point of maxium curvature. It may also be the location between the two regions that have relatively steady trends. Thus, the L method is able to determine the location where there is a significant change in the evaluation graph and it becomes erratic (x<44). In this case it indicates that too many segments have been merged together and the distance function is no longer as well-defined.

The poorer performance of BUS-global (compared to Gecko and BUS-greedy) is due to a lack of prominence in the knee of the curve compared to greedy methods (see lower-right graph in Figure 4.9). Greedy evaluation metrics increase more sharply at the knee, while global metrics have larger more ambiguous knees in their evaluation graph. A potential problem is if more than one knee exists in the evaluation graph. This is typically not a problem if one knee is significantly more prominent than the others. If there are two equally prominent knees, the L method is likely to return a number of segments that falls somewhere between those two knees. This is acceptable if all of the values between the two knees are reasonable. If not, a poor number of segments will most likely be returned by the L method.

The L method took approximately 0.01 seconds to determine the number of segments in every test cases and the segmentation algorithms took anywhere from 9 to 30 seconds to execute. The L method never required more than 0.1% of the total execution time to determine the number of segments. In stark contrast, permutation tests required up to 5 hours because each permutation of the original time series had to be segmented.

4.3 Summary

We have detailed our L method, which has been shown to work reasonably well in determining the number of clusters or segments for hierarchical clustering/segmentation algorithms. Hierarchical algorithms that have greedy evaluation metrics perform especially well. In our evaluation, the L method was able to determine a reasonable number of segments in 10 out of 11 instances for greedy hierarchical segmentation algorithms, and a correct number of clusters in 10 of 12 instances for hierarchical clustering algorithms. Algorithms with global evaluation metrics did not work as well with the L method because the knees in the evaluation graphs are not as prominent and easy to detect. The Gap statistic and permuation tests were also evaluated and the L method achieved much better results in our evaluation. The L method is also much more efficient than both the Gap statistic and permutation tests, typically requiring only a fraction of a second to determine the number of clusters rather than minutes or even many hours in the case of permutation tests.

Iterative refinement of the knee is a very important part of the L method. Without it, the L method would only be effective in determining the number of clusters/segments if the evaluation graph did not contain a large number of points. The iterative refinement algorithm explained in this chapter enables the L method to always run under optimal conditions: balanced lines on each side of the knee no matter how large the evaluation graph is or where the knee is located.

Like most existing methods, the L method is unable to determine if the entire data set is an even distribution and consists of only a single cluster (the null hypothesis). However, the L method also has the limitation that it cannot determine if only two clusters should be returned. Future work will explore possible modifications to the L method that will enable it to determine when only one or two clusters should be returned.

Chapter 5

Time Series Anomaly Detection Using States

The previous two chapters discussed a method to find the operational states in a time series. This chapter will introduce a method that can use the identified operational states in a time series to create a model of the normal time series. This model is in a format that can be easily read and understood by a human user. Once this model is created, it can be used to determine if other time series deviate significantly from it. Any deviation from the normal model is considered to be an anomaly.

5.1 Anomaly Detection System

Expert (knowledge-based) systems are often used to help humans monitor and control critical systems in real-time. For example, NASA uses expert systems to monitor various devices on the space shuttle. However, populating an expert system's knowledge base by hand is a time-consuming process. Instead, machine learning techniques can be used to generate the knowledge necessary to monitor the operation of devices or systems. This section introduces a method for generating models that can detect anomalies in time series data. Nearly all of the existing work in time series anomaly detection relies on models that are not easily readable and hence cannot be modified by a human for tuning purposes. Examples include a set of normal sequences (Dasgupta & Forrest 1996) and adaptive resonance theory (Caudell & Newman 1993). However, Langley et al. (2003) propose a method that uses process models to model a time series and predict future data. These process models are concise and are easily read and modified by humans, but their generation requires parameters to be set by a human that must have knowledge of the underlying processes that produce the time series. It is important for this model to be in an easily readable format that will allow human users to verify the model and modify it if necessary. This allows the user to understand *why* an anomaly was reported, and can help the user to quickly and appropriately respond to it. A transparent anomaly detection system that has a normal model that can be understood by a human user will be "trusted" much more by the user than an anomaly detection system that is a black box. A black box anomaly detection system simply spits out either "normal" or "anomaly" and can offer little or no explanation about why the data is anomalous. Black box anomaly detection systems are not likely to ever be fully trusted for mission critical systems. If a costly response (such as shutting down an assembly line) needs to be performed immediately upon detecting an anomaly, a black box system cannot be fully trusted since its operation and normal model is a complete mystery to the user. However, if the normal model is easily readable, the user can check the model to gain confidence that the anomaly detection system will perform as expected.

The normal operation of a device can usually be characterized in different operational states. In Chapter 3, we introduced the Gecko algorithm which is able to identify these states. Once these states are identified, the relationship between these states needs to be determined to allow tracking from one state to another and to detect anomalies. Given a time series depicting a system's normal operation, we desire to learn a model that can detect anomalies and can be *easily read and modified by human users*.

To create an anomaly detection system, we first characterize the states found by Gecko into logical rules so they may be read and modified with relative ease by humans. Then, given the knowledge of the different states, we determine the relationship among them for tracking normal behavior and detecting anomalies.

To characterize the states as logical rules, we use the RIPPER classification rule learning algorithm (Cohen 1995). Since different states often overlap in the one-dimensional input space, additional attributes (slope and second derivative) are derived to help characterize the states. To track normal behavior and detect anomalies, we construct a finite state automaton (FSA) with the identified states.

5.1.1 Overview

The input to our overall anomaly detection system is "normal" time series data (like the graph at the top left corner of Figure 5.1). The output of the overall system is a set of rules that implement state transition logic on an expert system, and are able to determine if other time series deviate significantly from the learned normal model. Any deviation from the learned "normal" model is considered to be an anomaly. The overall architecture of the anomaly detection system, depicted in Figure 5.1, consists of three components: clustering, rule generation (characterization), and state transition logic.

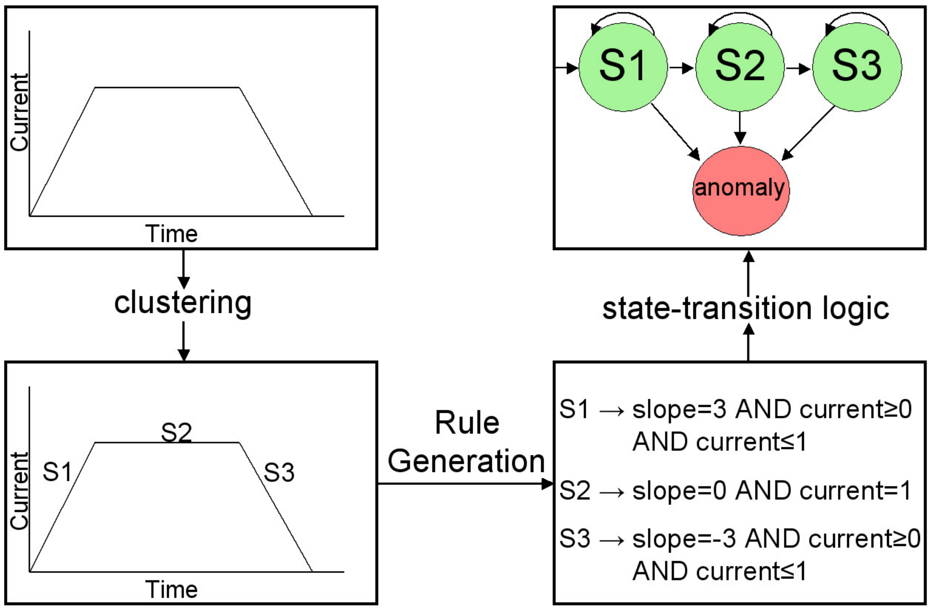

Current
Time
clustering
Current
S1
S2
S3
Time
Rule Generation
S1 → slope=3 AND current≥0 AND current≤1
S2 → slope=0 AND current=1
S3 → slope=-3 AND current≥0 AND current≤1
state-transition logic
S1
S2
S3
anomaly

Figure 5.1. Main steps in time series anomaly detection.

The clustering phase is performed by our Gecko algorithm, which is designed to identify distinct states (or clusters) in a time series. Next, rules are created for each state by the RIPPER algorithm (Cohen 1995). Finally, rules are added for the transitions between states to create a finite state automaton. Gecko was discussed in Chapter 3, while rule generation and state-transition logic will be explained in the next two subsections.

5.1.2 Characterizing States by Generating Rules

We have adapted RIPPER (Cohen 1995) to generate human readable rules that characterize the states identified by the Gecko algorithm. The RIPPER algorithm is based on the Incremental Reduce Error Pruning (IREP) (Furnkranz & Wildmer 1994) over-fit-and-prune strategy. The IREP algorithm is a 2-class approach, where the data set must first be divided into two subsets. The first subset contains examples of the class whose characteristics are desired (the positive example set) and the other subset contains all other data samples (the negative example set). Our implementation of RIPPER acts as an outer loop for the IREP rule construction.

The input to RIPPER is the data produced by Gecko which contains time series data classified into c^* states. RIPPER will execute the IREP algorithm c^* times, once for each state. At each execution of IREP, a different state is considered to be the positive example set and the rest of the states form the negative example set. This creates a set of rules for each state. To describe the relationship among these states, state transition logic is identified as discussed in the following section.

5.1.3 State Transition Logic

The upper right-hand quadrant of Figure 5.1 depicts a simplified state transition diagram for a time series containing just three states. The state transition logic is described by three rules for each state corresponding to each of the three possible state transition conditions on each input data point:

- IF input matches current state's characteristics THEN remain in current state.
- IF input matches the next state's characteristics THEN transition to the next state.
- IF input matches neither the current state's nor the next state's characteristics THEN transition to an anomaly state.

The essential element of each of these three rules is the antecedent condition, which characterizes the data points belonging to each state. The antecedent condition for each state is obtained from the RIPPER rule generation process. The state transition logic simply needs to glue together the proper antecedents to formulate the above three transition rules for each of the c* states identified by Gecko.

Unfortunately, our state transition logic needs to be somewhat more complex. In the domain of devices and systems we are attempting to monitor, sensors may sometimes report short-term, transient, anomalous values – false alarms. In order for our approach to be more robust in handling these transients, we have added extra counting/threshold logic to the transition from a normal state to the error state. Before an anomaly state is entered, the number of consecutively observed anomalous values must exceed a specified threshold. Thus, an anomalous condition is not annunciated unless the observed values have been improper for some length of time. Similar logic is provided for the transition from a normal state to its normal successor to prevent premature state transitions. The expanded state-transition logic is shown by the state-transition diagram in Figure 5.2, where states prefixed with a "S" indicate operational states of the device, and states prefixed with a "T" are transition states that need a consecutive number of points to proceed to the next state..

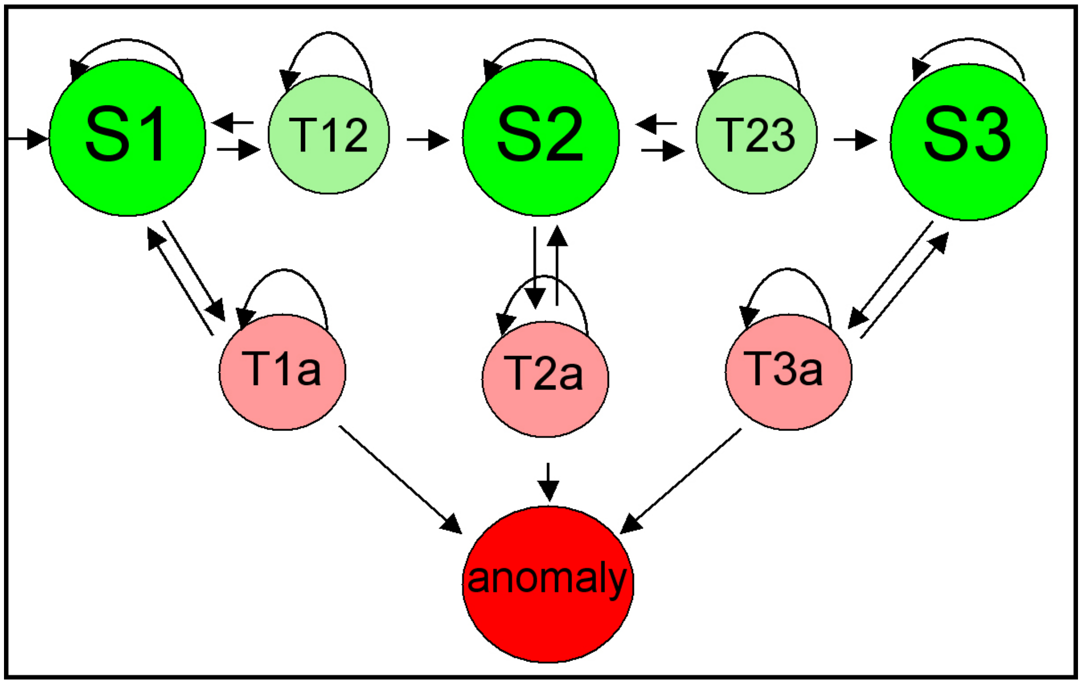


Figure 5.2. Expanded state-transition logic with transition thresholds.

When data is seen that indicates the next state should be transitioned to, the transition states cycle back to themselves a specified number of times before actually transitioning to the next state. If the state machine is currently in a transition state and the current input point is not in the next state, the state machine backtracks to the previous state.

This simple sequential model will get "stuck" in a state if it misses a state transition due to an anomaly. The first anomaly is correctly identified, but the no future data can be tracked because the state machine is stuck in an old state. A solution we have found that performs well is to use a non-deterministic state machine model rather than a deterministic model. When an anomaly is detected, we create several state machines, each starting in a different state. All of the state machines run in parallel until they converge to a single state. This method allows the system to recover from a short sequence of anomalous data and to determine the current state of the input data. If a state machine contains many states and running individual state machines for each state is impractical, states can be searched starting with ones near where the anomaly was detected and increasing the number of states to search if the state machines continue to get "stuck". In our tests, the correct state is determined very quickly.

5.2 Empirical Evaluation

This evaluation will test the anomaly detection system described in this chapter on both normal an anomalous time series. When training on a normal time series, the same time series and additional normal time series should not cause anomalies during testing. However, when training on abnormal time series, the anomaly detection system should find anomalies in the time series.

5.2.1 Procedures and Criteria

In order to test whether the anomaly detection system works correctly we performed three kinds of tests: (1) Self-tracking: Use 90% of the data points to create rules, and then use 100% of the data fed into the expert system to see if the state transitions occur correctly, without detecting any anomalies. (2) Normal operation: Use all of a normal valve's data to create a normal model, and then monitor another valve that is also operating normally. This case should also not trigger any anomalies. (3) Detecting anomalies: Use all of a properly functioning valve's data to learn a normal model, and then use time series of valves that are damaged slightly and run them through the anomaly detection system. The damaged valves should trigger anomalies.

5.2.2 Self-Tracking Results

The baseline test of the anomaly detection system is to train the model with 90% of the data, and seeing if 100% of the data can be tracked without triggering an anomaly. The results of this test are shown in Table 5.1. An error point in Table 5.1 is any point that is unexpected in the state transition logic. This means that the point is neither in the current state or the following state. Time series data often contains noise and minor variations. For this reason, anomalies must not be triggered by only a single data point that does not agree with the model contained in the FSA. By using a threshold counter, an anomaly will only be reported after a certain number of consecutive error points. The last column in Table 5.1 shows what the minimum consecutive error threshold (*errorThreshold*) must be set to for the anomaly detection system to not report an anomaly. A value of 1 in the bottom row means that the anomaly detection system will correctly not report an anomaly as long as *errorThreshold*≥1.

Table 5.1. Self-tracking of a time series.

Data Set	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	**Avg**
Error Pts (%)	1.1	0.8	0.7	0.5	0.0	0.4	0.3	0.2	0.4	1.1	**0.6**
Min. Error Threshold	2	2	1	1	0	1	1	1	1	21	**4.0**

In this experiment, both the "consecutive transition" (*transitionThresh*) and the "consecutive error" (*errorThreshold*) thresholds were set to zero. This causes every possible state transition to be made and every error point triggers an anomaly. This enabled easy computation of the number of error points. Data set number 10 performs poorly in this test because the FSA transitions prematurely near the end of the time series and starts reporting many anomalies, the results for this data set can be improved by increasing *transitionThresh* to prevent it from transitioning too early on a single spurious data point.

5.2.3 Normal Operation Results

This test is to show that the anomaly detection system's generated normal model is general enough to recognize that an untrained normal time series contains no anomalies. In this test, the anomaly detection system trained on data set 1, and then tested on data set 2. Both of these data sets are of normally operating valves that contain minor (but visible) differences. The "consecutive transition" threshold (*transitionThresh*) parameter was set to 2, and *errorThreshold* was set to 10 (minimum possible cluster size *s*=1*0*). This means that more than two consecutive points believed to be in the next state are needed to perform a state transition and more than ten consecutive points believed to be errors are needed to declare that the time series contains anomalies.

The system was able to successfully transition through the states, without detecting any anomalies. Of 979 data points, 61 (2.6%) were error points; they were not believed to belong to the current state, nor to be transition points belonging to the following state. However, since a consecutive number of errors greater than *errorThreshold* was never encountered, an anomaly was never triggered.

5.2.4 Detecting Anomalies Results

This final test is to show that our system is capable of detecting when a time series differs significantly from the learned model. In this test, two data sets containing time series of valves operating normally (data sets 1 and 2) were used to develop the normal models. Each normal model was then run against the remaining anomalous data sets (data sets 3…10).

For each of the 16 tests, the anomaly detection system correctly determined that the time series contained anomalies. Additionally, the system was able to inform the user of the state number where the time series differs from the model. Thus, the system does not only give a yes/no answer to whether a time series contains anomalies, but it is also able to explain to the user where the anomaly occurred. Also, because the rules generated by RIPPER are in a human-readable format, the user can look at the rule for the state where the error occurred and understand exactly why the system reported the anomaly.

5.3 Summary

This chapter detailed an approach to time series anomaly detection by discovering and characterizing the states in a time series, and performing transition logic between these states to construct a finite state automaton that can be used to track normal behavior and detect anomalies. The rules generated for each state by the RIPPER algorithm are in disjunctive normal form and can be *easily understood and modified by humans*. (Moreover, the generated rules can be in a format used by the SCL expert system shell at ICS, which is a collaborator on this NASA project.)

The overall anomaly detection system was able to detect anomalies in every time series that was from a 'damaged' valve, and was also able to monitor a 2nd normal valve without detecting any anomalies. However the anomaly detection system, as it has been described so far, has a severe limitation. The method described in this chapter can only take a single time series as input to build a normal model. This is problematic because the normal range of values that are allowed for each state can only be accurately determined if a range of values are seen during training. Training on only a single time series can create a normal model that is too restrictive because it is not possible for it to know the allowed variation of the system during each state. Chapter 6 will describe how to extend this anomaly detection method to allow training on multiple time series.

Chapter 6

Building a Normal Model by Training on Multiple Time Series

The anomaly detection system described in Chapter 5 takes a single normal time series as input, and builds a normal model that can be used to determine if additional data is anomalous. Any time series that deviates significantly from the normal model is considered to be anomalous. However, the term “deviates significantly” is not easily defined when only a single time series is used for training. A “normal” time series may vary due to different operating conditions of the device being monitored. Variable operating conditions such as ambient temperature, varying voltage, recent use, and the effects of aging on the device can all cause deviations in the times series produced by normally operating devices. This allowed variation usually cannot be expressed by a simple parameter that permits a fixed amount of deviation from the learned normal model because the amount of permitted or normal deviation in the time series may vary between each of the device’s operational states. In order to accurately determine the allowed variation in the time series of monitored devices, the anomaly detection system must be trained on multiple time series. Training on multiple time series allows generalization to occur. If the temperature reading during a particular state is always 20.0 degrees in one normal time series, and 25.0 degrees in another, then it is likely that temperatures between 20 and 25 are also normal.

The initial anomaly detection system described in Chapter 5 was not designed to train on more than a single time series, therefore the permitted deviation of a device could not be observed during training. We wish to extend the anomaly detection system to allow a single normal model to be created by training on multiple time series. In order to create a single model (state machine between states of the time series) from multiple input time series, the rules generated for each state must be generalized to cover all data points of that state across all of the input time series. To do so, the portions of each time series that correspond to the n^{th} state must be determined. However, it is difficult to determine which portions of each time series belong to the n^{th} state. Clustering with Gecko can identify a reasonable set of states for a single time series. However, Gecko cannot be run on each time series to find corresponding states between the time series because Gecko is likely to find slightly different states for each time series, and can also return a different number of states for each time series.

This chapter introduces two improvements to the anomaly detection system described in Chapter 5 that will allow training on more than one time series:

1. Every normal time series is "merged" into a single representative time series before it is clustered by Gecko.

2. The points used to create the rules are now the points from every time series that correspond to that rule's state (states are identified by clustering the merged time series).

Dynamic time warping will be used both to create the merged time series, and also to determine the corresponding regions between all of the time series.

Section 6.1 describes dynamic time warping; Section 6.2 explains how dynamic time warping can be used to extend the anomaly detection system described in Chapter 5 to allow multiple time series to be used during training; Section 6.3 contains an empirical evaluation of the anomaly detection system after it has been extended to allow training on multiple time series; and Section 6.4 summarizes the work in this chapter.

6.1 Dynamic time Warping

Dynamic time warping (DTW) is a technique that finds the optimal match between two time series if one time series may be “warped” non-linearly by stretching or shrinking it along its time axis (Kruskall & Liberman 1983). Dynamic time warping is most commonly used in speech recognition to determine if two waveforms represent the same spoken phrase. The duration of each spoken sound and the interval between sounds are permitted to vary, but the overall speech waveforms must be similar. Dynamic time warping is often used to find the distance along the optimal warp path to determine the similarity between the two speech waveforms. Dynamic time warping is commonly used in data mining as a distance measure between time series. An example of how one time series is “warped” to another is shown in Figure 6.1.

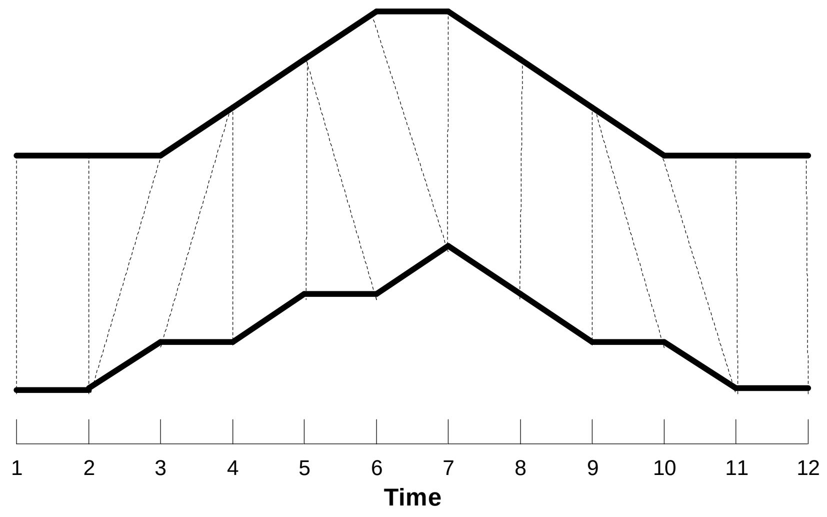


Figure 6.1. Two time series “warped” together using dynamic time warping.

In Figure 6.1, each line connects the point of one time series to its correspondingly similar point in the other time series. The lines actually have similar values on the *y*-axis but have been separated so the vertical lines between them can be viewed more easily. If both of the time series in Figure 6.1 were identical, all of the lines would be straight vertical lines, no warping would be necessary to ‘line up’ the two time series. The warp path distance is a measure of the difference between the two time series after they have been warped together, which is measured by the sum of the distances between each pair of points connected by the vertical lines. Thus, two time series that are identical except for localized stretching and contracting of the time axis will have warp path distances of zero.

Dynamic time warping is typically used in data mining only to determine the similarity between two time series (Keogh & Pazanni 2000). Calculating the distance after warping has the advantage that the two time series do not need to line up absolutely perfectly and be in phase with each other to produce an accurate distance measurement. However, for our purposes, we are more interested in using the calculated warp path to find areas that are similar between two time series. Thus, we are more interested in the warp path rather than the warp path distance.

6.1.1 Problem Formulation

The dynamic time warping problem is stated as follows: Given two time series *X*, and *Y*, of lengths *maxX* and *maxY*:

$$X = x_1, x_2, \ldots, x_i, \ldots, x_{maxX}$$
$$Y = y_1, y_2, \ldots, y_j, \ldots, y_{maxY}$$

construct a warp path *W*:

$$W = w_1, w_2, \ldots, w_K \qquad \max(maxX, maxY) \leq K < maxX + maxY$$

where *K* is the length of the warp path and the kth element of the warp path is:

$$w_k = (i, j)$$

where *i* is an index from time series *X*, and *j* is an index from time series *Y*. The warp path must start at the beginning of each time series at $w_1 = (1, 1)$ and finish at the end of both time series at $w_K = (maxX, maxY)$. This ensures that every index of both time series is used in the warp path. There is also a constraint on the warp path that forces *i* and *j* to be monotonically increasing in the warp path, which is why the lines representing the warp path in Figure 6.1 do not overlap. Every index of each time series must be used. Stated more formally:

$$w_k = (i, j), w_{k+1} = (i', j') \qquad i \le i' \le i+1,\ \ j \le j' \le j+1 \tag{6.1}$$

The warp path found must be the optimal (minimum distance) warp path, where the distance of a warp path *W* is:

$$Dist(W) = \sum_{k=1}^{k=K} Dist(w_{ki}, w_{kj}) \tag{6.2}$$

where *Dist*(*W*) is the distance (typically Euclidean distance) of warp path *W*, and *Dist*(w_{ki}, w_{kj}) is the distance between the two data point indexes (one from *X* and one from *Y*) in the *k*th element of the warp path.

6.1.2 Dynamic Time Warping Algorithm

A dynamic programming approach is used to find this minimum cost warp path. Instead of attempting to solve the entire problem all at once, solutions to sub-problems (portions of the time series) are found, and used to repeatedly find solutions to a slightly larger problem until the solution is found for the entire time series. A two-dimensional *maxX* by *maxY* cost matrix *D*, is constructed where the value at *D*(*i*, *j*) is the minimum distance warp path that can be constructed from the two time series $X' = x_1, \ldots, x_i$ and $Y' = y_1, \ldots, y_j$. The value at *D*(*maxX*, *maxY*) will contain the minimum distance warp path between time series *X* and *Y*. Both axes of *D* represent time. The *x*-axis is the time of time series *X*, and the *y*-axis is the time of time series *Y*. Figure 6.2 *D* shows an example of a cost matrix and a minimum distance warp path traced through it from *D*(1, 1) to *D*(*maxX, maxY*).

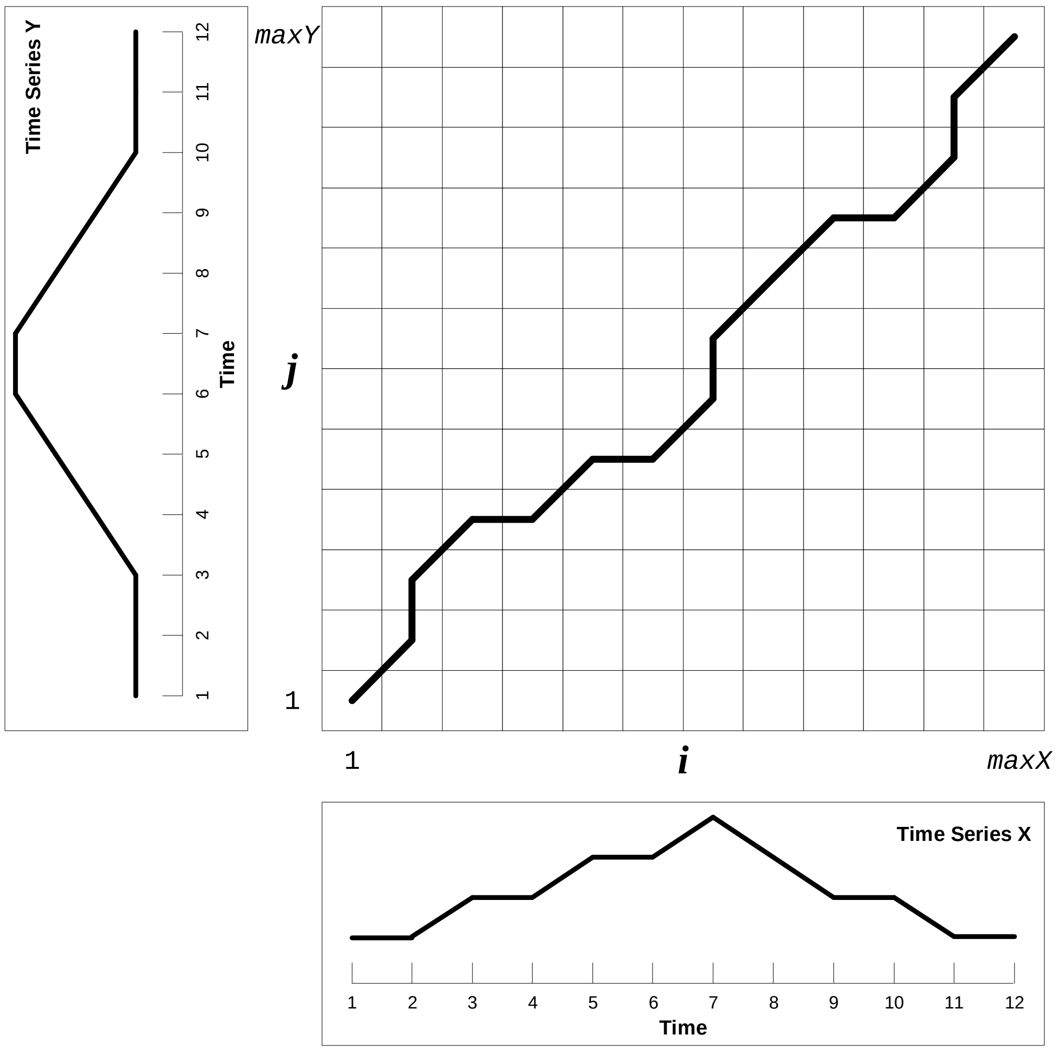


Figure 6.2. A cost matrix with the min. distance warp path traced through it.

The cost matrix and warp path in Figure 6.2 are for the same two time series shown in Figure 6.1. The warp path is W = {(1,1), (2,2), (2,3), (3,4), (4,4), (5,5), (6,5), (7,6,), (,7,7), (8,8), (9,9), (10,9), (11,10), (11,11), (12,12)}. If the warp path line passes through a cell $D(i, j)$ in the matrix, it means that the i^{th} point in time series X is warped to the j^{th} point in time series Y. Notice that where there are vertical sections of the warp path, a single point in time series X is warped to multiple points in time series Y, and the opposite is also true where the warp path is a horizontal line. Since a single point may map to multiple points in the other time series, the time series do not need to be of equal length. If X and Y were identical time series, the warp path through the matrix would be a straight diagonal line.

To find the minimum distance warp path, every cell of the cost matrix must be filled. The rationale behind using a dynamic programming approach to this problem is that since the value at $D(i, j)$ is the minimum warp distance of two time series of lengths i and j, if the minimum warp distances are already known for all slightly smaller portions of that time series that are a single data point away from lengths i and j, then the value at $D(i, j)$ is the minimum distance of all possible warp paths for time series that are one data point smaller than i and j, plus the distance between the two points x_i and y_j. Since the warp past must either be incremented by one or stay the same along the i and j axes, the distances of the optimal warp paths one data point smaller than lengths i and j are contained in the matrix at $D(i\text{-}1, j)$, $D(i, j\text{-}1)$, and $D(i\text{-}1, j\text{-}1)$. So the value of a cell in the cost matrix is:

$$D(i, j) = \min[D(i-1, j), D(i, j-1), D(i-1, j-1)] + Dist(i, j) \qquad [6.3]$$

The warp path to $D(i, j)$ must pass through one of those three grid cells, and since the minimum possible warp path distance is already known for them, all that is needed is to simply add the distance of the current two points to the smallest one. Since Equation 6.3 determines the value of a cell in the cost matrix by using the values in other cells, the order that they are evaluated in is very important. The cost matrix is filled one column at a time from the bottom up, from left to right as depicted in Figure 6.3.

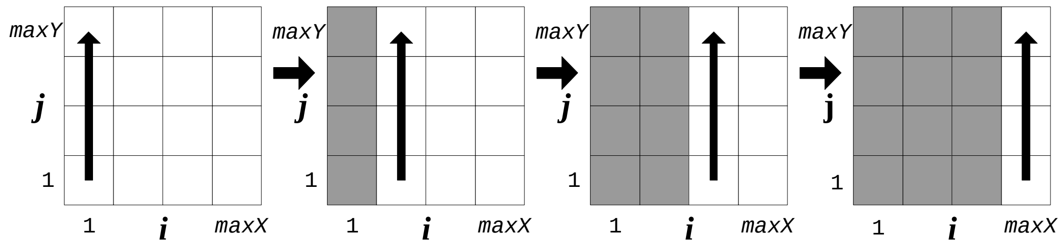


Figure 6.3. The order that the cost matrix is filled.

Equation 6.3 actually only applies to cells that are not in the first row or column because *D*(*i*-1, *j*) is undefined for the first column, and *D*(*i*, *j*-1) is undefined for the first row. There are actually four different cases when calculating the values of the cells, depending on its location:

1) $i = 1, j = 1$ $\quad D(i, j) = Dist(i, j)$

2) $i = 1, j = 2 \ldots maxY$ $\quad D(i, j) = D(i, j-1) + Dist(i, j)$

3) $i = 2 \ldots maxX, j = 1$ $\quad D(i, j) = D(i-1, j) + Dist(i, j)$

4) $i = 2 \ldots maxX, j = 2 \ldots maxY$ $\quad D(i, j) = \min[D(i, j-1), D(i-1, j), D(i-1, j-1)] + Dist(i, j)$

The first case is for the cell at the bottom left corner of the matrix that is filled in first. Case 2 is for the rest of the first column, and case 3 is for the rest of the bottom row. The remaining cells are calculated by the fourth case (Equation 6.3). The cells that are calculated by each of these four cases are shaded in Figure 6.4.

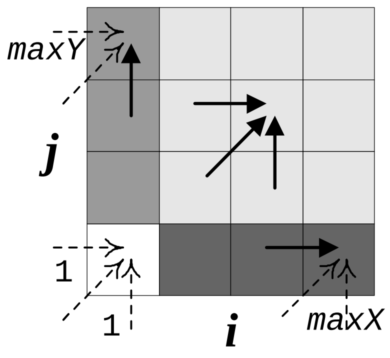


Figure 6.4. Four shaded areas showing how cells that are calculated.

The group of three arrows in Figure 6.4 indicate the locations of *D*(*i*-1, *j*), *D*(*i*, *j*-1), and *D*(*i*-1, *j*-1). The smallest of these three values is added to the distance between the two points x_i and y_j. A dashed line indicates that the location does not exist when calculating the value for the target grid cell.

After the entire matrix is filled, a warp path must be found from *D*(1, 1) to *D*(*maxX*, *maxY*). The warp path is actually calculated in reverse order starting at *D*(*maxX*, *maxY*). A greedy search is performed that evaluates cells to the left, down, and diagonally to the bottom-left (reverse direction of the arrows in Figure 6.3). Whichever of these three adjacent cells has the smallest value is added to the beginning of the warp path found so far, and the search proceeds from that cell. The search continues until $D(1, 1)$ is reached. If there is a tie between the possible cells to add to the warp path, any of the tied cells can be used and a correct answer will be calculated. The warp path is guaranteed to be a minimum distance warp path, but there could possibly be many. In the event of ties, it is usually desirable to move diagonally if possible. This helps to avoid singularities where large sections of two time series are nearly identical. A singularity occurs when a single point in one time series maps to a large number of points in the other. For example, consider to identical time series of 100 points that are straight lines. Any warp path will be the minimum warp path with a distance of zero. However, it is desirable to have the warp path $W_1 = \{(1,1), (2,2),\ldots,(100,100\}\}$ rather than the warp path $W_2 = \{(1,1),(2,1),\ldots,(100,1), (100,2), (100,100)\}$. The warp path W_1 always moves diagonally when there is a tie and has a warp path length of 100. The warp path W_2 moves left when there is a tie and has a warp path length of 199. Avoiding singularities creates a warping that is more natural. Additionally, for some applications it may also be desirable for a warp path to be as close to a linear warp as possible, by breaking ties so the warp path moves in the direction of the right diagonal line (the line where $i=j$).

6.1.3 Derivative Time Warping

Typically, dynamic time warping is performed using the original values when distances between points are determined. This often gives a desirable distance measurement between time series when the amplitude difference between two time series is important. However, using original values in the dynamic time warping algorithm often creates warp paths that are unintuitive for certain domains. Amplitude variations between the time series can easily cause singularities and produce warp paths that may be undesirable for a given domain. In our case, mappings between points based on the warp path do not correspond well to operational states. Figure 6.5 shows a warping between two time series using original values in the dynamic time warping algorithm.

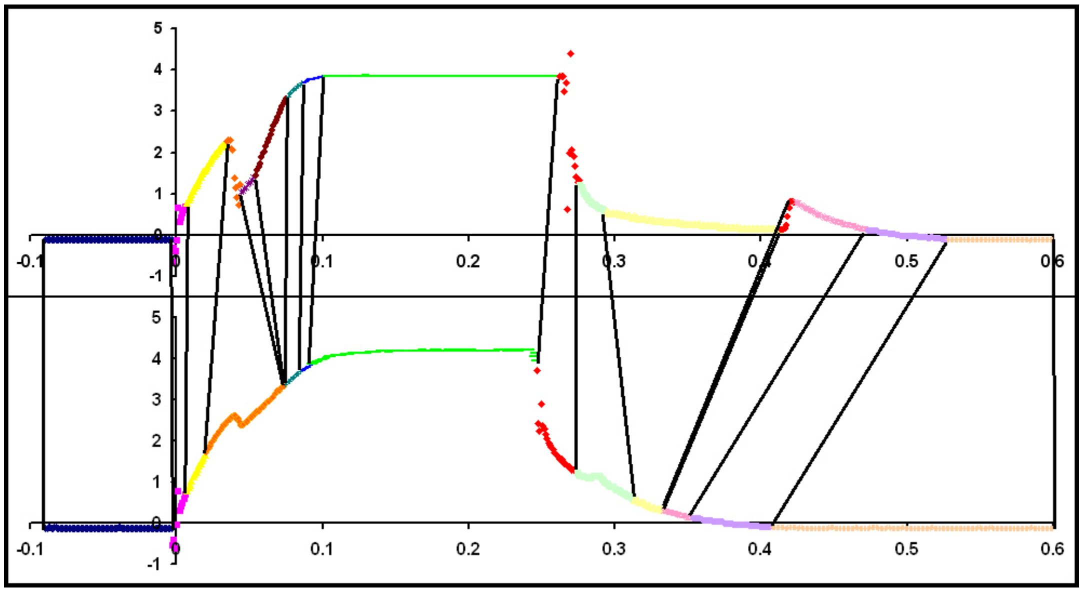


Figure 6.5. A warping between two time series using original values.

Warp path lines are drawn at the boundaries of the states (found by Gecko) simply to aid in the visualization of the overall warp path. Depicting hundreds or thousands of warp path lines would be illegible, and just a few lines at the boundaries of the states are sufficient to gain an overall picture of the warp path. There are many problems with the warp path in Figure 6.5, and only a few will be pointed out. The small number of points in the fourth state, which starts to fall rapidly at *y*=2.2 in the top time series, are mapped to a large number of points that share a common value on the *y*-axis, but otherwise are very dissimilar. In the top time series the slope is very negative, and in the bottom time series the slope is positive. Another problem is that the warp path contains several singularities are *x*=0.08 in the bottom time series. The last problem that will be mentioned is that the flat state in the top time series is warped to a slightly "U" shaped state in the bottom time series. This occurs because the plateau in the bottom time series is slightly higher than at the top. If the height difference in the plateaus were to increase this "U" would become very pronounced. Overall the states in the top time series are mapped to the bottom time series in an unintuitive manner.

An alternative is to consider the "shape" of the time series during warping rather than the original values in the *y*-axis. To do so, the dynamic time warping algorithm should use slope values of the two time series rather than the original values. This method is called derivative dynamic time warping (Keogh & Pazzani 1999). Figure 6.6 shows the warp path between the same two time series, but the warp path minimizes the warp path cost of the differences in slopes rather than original *y*-axis values.

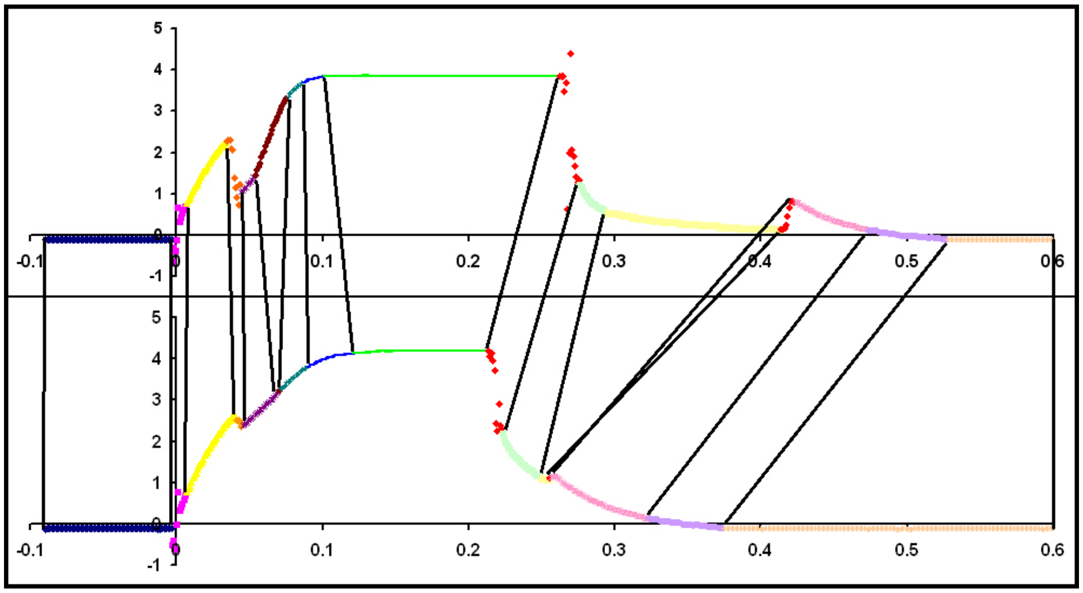


Figure 6.6. A warping between two time series using slope values.

The result is a mapping of states between time series that is very intuitive. The small states of the top time series at *x*=0.4 and *x*=0.42 are correctly warped to the same (but much smaller) states in the bottom time series. Derivative dynamic time warping is an effective method to find regions across different time series that are similar, and will be utilized in the next section to merge multiple time series into a single representative time series.

6.2 Using DTW to Train on Multiple Time Series

To extend the anomaly detection system discussed in Chapter 5 to allow multiple time series to be used during training, we will make use of dynamic time warping's ability to find related sections of different time series. The steps discussed in Chapter 5 (Figure 5.1) are still used: clustering, rule-generation, and state-transition logic. However, two additional steps are introduced:

- A new step to merge multiple time series into a single representative time series
- A step to determine the state for each point in every time series based on the set of states identified for the merged time series

Merging is performed before immediately before clustering, and the state's are determined for every time series immediately after clustering. The new overall approach to anomaly detection is shown in Figure 6.7.

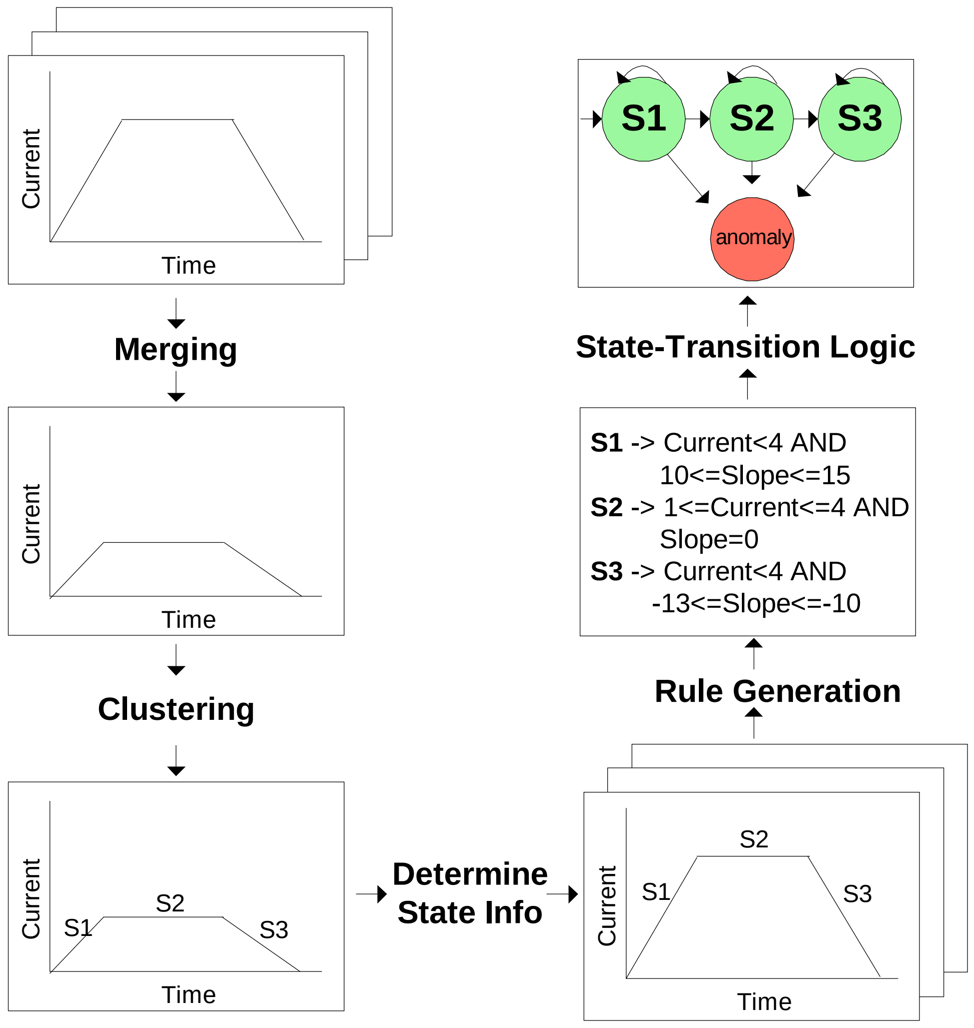


Figure 6.7. Main steps in time series anomaly detection with multiple input time series used during training.

The new Merging step creates a representative time series that is given as input to the Gecko algorithm for clustering. After Gecko finds the states of the merged time series, the next step uses the state information in the merged time series to determine the state of every point in all of the input time series. This is possible because a warp path from the merged time series to every other time series is created during the first merging step. The rule generation program does not need to be modified, but the input to it now contains all of the points from every time series used for training. This allows the rules to be more general and allow variations for each state depending on the amount variation of observed variation for each state across all time series used for training. Once the rules are created state transition logic is created as explained in Chapter 5.

Multiple time series are merged into a single time series using dynamic time warping. One of the time series must be picked to use as a template. For speech recognition, the template is often the time series of average length (Abdulla, Chow & Sin 2003). The best merge will result from using the time series that is the most "average" as the template. The most average time series can be determined by performing dynamic time warping on each combination of time series. The time series that has the smallest warp path distance sum over all of the other time series is the most average because it is the most similar to all of the other time series. The length of the merged time series will be equal to the length of the time series used as the template. The template must first be warped to every other time series. For the k^{h} point in the template, average all of the points that it warps to in other time series (weighted so each time series has equal influence), and use that average as the k^{h} point in the merged time series. Conceptually, this approach "lines up" all of the time series and calculates the average of the points vertically.

Determining the states of every time series, after determining the states of the merged time series is, also makes use of the warp paths created by dynamic time warping. However, in this case there are two levels of indirection. The template and merged time series have a liner warp between them so the n^{th} point in the template warps to the n^{th} point in the merged time series. So the states of the template time series can be directly determined by the merged time series. Once the states are known for the points in the template, all that is needed is to go through the template time series, and assign that merged point's state membership to all of the points that it warps to in other time series.

6.3 Empirical Evaluation

This evaluation will test how well the extended anomaly detection system performs when trained on multiple time series. Training on multiple time series enables the normal model to be created that is more general and contains the amount of allowed variation for each state. If two time series are used to build a normal model, testing on an additional time series that are "between" the two training time series should also not produce any anomalies, even if it differs significantly from all of the training data. As an example, suppose that a normal model was constructed by training on data from a car's engine running at 20 mph and 40 mph. It is desirable that the normal model would cover all normal engine activity between 20 and 40 miles an hour, rather than only covering narrow ranges of activity near 20 and 40 mph.

6.3.1 Procedures and Criteria

To evaluate the ability of our anomaly detection system to generalize beyond the training data to cover unseen normal variations and determine if time series "between" the training data are covered by the generated normal model, data is needed for which "between" is defined. The data used for this evaluation is taken from a valve in the space shuttle under controlled tests where the same is operation is performed each time, but under different controlled conditions. The data was generated by repeatedly turning the valve on and off and increasing the voltage applied to the valve in each run. Ten time series were collected in this manner at the following voltages: 14, 16, 18, 20, 22, 24, 26, 28, 30, and 32. Each time series contains measurements of current over time. The effect of increasing the voltage on the valve has different effects on each operational state. A few of the effects of increasing the voltage on different states of the time series are (in order from most pronounced to marginal):

- The plateau of the steady-state on position increases significantly as the voltage is increased.
- The rate of increase in current after the valve is turned on increases as the voltage increases.
- If the voltage is too low, the valve may fail to open (and later close). The valve opening and closing is indicated in the time series by a “bounce” of current.
- The “bounce” in the time series when the valve opens decreases as the voltage increases, but the bounce when a valve closes in unaffected. The bounce also occurs at a slightly higher level of current as the voltage increases.
- The section of the time series after turning off the valve changes very little between different voltages.
- Steady-state off is unaffected by voltage, and is a flat section with a value of zero for the current in every test case.

Figure 6.8 illustrates the effect of changing the voltage on the valve. The time series on the left was collected at 16 Volts, and the time series on the right was collected at 30 Volts. Both time series are from the same valve, only the voltages differs.

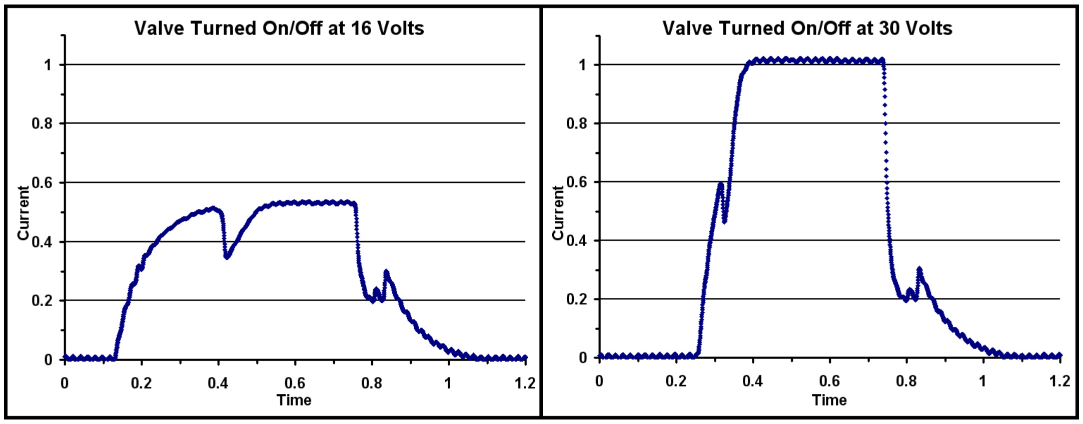


Figure 6.8. A valve being turned on and off at different voltages.

The effects of change the voltage on the time series produced by the valve can be easily seen in Figure 6.8. The plateau is higher and the rising slope is much steeper at higher voltages, but the section of the time series where the valve is turned off (at approximately Time=0.75) is nearly identical regardless of the voltage.

These data sets are collected under exactly the same conditions except for a single parameter, the voltage. The assumption made in this evaluation is that if multiple time series are provided as training data, the normal model that is generated should consider all time series that were trained on to be normal, and also consider all time series at intermediate voltages to be normal. Conversely, all time series at lower or higher voltages than seen during training should cause anomalies to be triggered.

The experimental procedure is simple, train on one or more of the ten time series, and then test on all ten. The time series that should not be considered anomalous are the time series collected at voltages within the range (inclusive) of the training time series voltages. For example, if time series at 20 and 28 Volts were used for training, then the time series with voltages 20, 22, 24, 26, and 28 should not be considered anomalous, but all other time series collected at voltages less than 20 or greater than 24 should cause anomalies. Tests will be performed training on from one to three time series. Training on only a single time series should create a restrictive normal model that will detect anomalies for all other time series.

This evaluation differs from the one performed in Chapter 5 in two major ways. The first difference is that this evaluation trains on multiple time series. The second difference is that in this evaluation, there are no "normal" and "abnormal" valves. Instead we are measuring the ability of the normal model to generalize to unseen training data that is "between" the time series used for training. The notion of normal and abnormal depends more on the domain than actual data. A time series that is abnormal in one domain may be perfectly normal in another. So, in this evaluation, we are not concerned with whether a time series is actually considered normal or abnormal. Instead, we are treating different ranges of time series (in the range of voltages seen during training) as "normal" and everything else abnormal.

The parameters used to create the normal model were: *minClusterSize*=20 (Gecko), *transitionThreshold*=3 (FSA), and *errorThreshold*=10 (FSA). Thus, the clusters returned by Gecko had to contain at least 20 points. The two parameters for the state-transition logic specify more than three consecutive transition points must be seen before the next state is transitioned to, and more than 10 consecutive error points must be seen to raise an anomaly.

6.3.2 Results and Analysis

The first tests performed were to train on only a single time series. When only a single time series is trained, the model is very specific because the amount of normal variation allowed can't be determined when training on only a single time series. It is expected that if a time series is used for training, the model produced will consider all time series other than the one used for training as anomalous. Table 6.1 contains the results of training on only a single time series.

Table 6.1. Results of training on a single time series, testing on all.

		TEST									
		14V	*16V*	*18V*	*20V*	*22V*	*24V*	*26V*	*28V*	*30V*	*32V*
TRAIN	*14V*	P	F	F	F	F	F	F	F	F	F
	16V	F	P	F	F	F	F	F	F	F	F
	18V	F	F	P	F	F	F	F	F	F	F
	20V	F	F	F	P	F	F	F	F	F	F
	22V	F	F	F	F	P	F	F	F	F	F
	24V	F	F	F	F	F	P	P/F	F	F	F
	26V	F	F	F	F	F	F	P	F	P/F	P/F
	28V	F	F	F	F	F	F	F	P	F	F
	30V	F	F	F	F	F	F	F	F	P	F
	32V	F	F	F	F	F	F	F	F	F	P

In Table 6.1, a “P” indicates that when training on the time series in that row, and testing on the time series at that column, the state transition logic reads through the entire test file without any anomalies. An “F” indicates that at least one anomaly is found, and a “P/F” indicates that anomalies are found with an *errorThreshold* of 10, but passes if the threshold is increased to 20. As expected, in Table 6.1 time series were only able to be tracked without anomalies when they were trained on.

The remaining tests are performed by training on either two or three time series. It is expected that the time series used for training will not cause anomalies during testing, and all time series generated at voltages between the training voltages should also not cause anomalies. Table 6.2 displays the tests when training on multiple time series.

Table 6.2. Results of training on multiple time series, testing on all.

		TEST									
		14V	*16V*	*18V*	*20V*	*22V*	*24V*	*26V*	*28V*	*30V*	*32V*
TRAIN	*16V,20V*	F	P	F	P	F	F	F	F	F	F
	20V,24V	F	F	F	P	P	P	F	F	F	F
	24V,28V	F	F	F	F	F	P	P	P	P/F	F
	28V,32V	F	F	F	F	F	F	F	P	P	P
	14V,20V	F	F	F	P	F	F	F	F	F	F
	20V,26V	F	F	F	P	P/F	P	P	F	F	F
	26V,32V	F	F	F	F	F	F	P	P/F	P/F	P
	16V,20V24V	F	F	F	P	P/F	P	F	F	F	F
	24V,28V,32V	F	F	F	F	F	P	P	P	P/F	P
	14V,20V,26V	F	F	F	P	P	F	P	F	F	F
	18V,24V,30V	F	F	P	P/F	P	P	P	F	P	F

The cells are shaded in Table 6.2 to help make it easier to understand. A bright green cell with a "P" indicates that the test case at that cell was expected to pass without an anomaly and it did. A red cell with an "F" indicates that the test case was expected to generate no anomalies, but at least one anomaly was found. The olive "P/F" indicates that the test case correctly has no anomalies if the *errorThreshold* is doubled to 20.

In Table 6.2, the "range" of expected time series that will not cause anomalies can be easily determined. In the first eight rows, two time series were trained with a single time series having an intermediate voltage. The model correctly generalized to include the unseen time series with the in-between voltage 3 out of 4 times, and still correctly produced anomalies when tested on the other 7 time series.

The second group of tests in Table 6.2 also trained on two time series, but had two time series "in-between" them instead of just one. This group of tests gave correct results for 2 out of 3 of the test cases.

The final group of tests in Table 6.2 trained on three time series. This group of tests had more mistakes than the previous tests. The results were good except when training and testing on valves with the lowest voltage. The valve completely fails to open or close at low voltages, and has no pair of “bumps” in the time series. Since the states of the “bounce” that are identified in the merged time series are mapped back to the original time series that do not have those corresponding regions, the missing states end up getting a very small number of points (often one) assigned to those missing states. All other states are mapped correctly, and rules are generated to characterize the states. But if testing occurs on that valve with the missing state, it is impossible to transition into the state that is very small because *transitionThresh* number of consecutive points must be seen to perform a transition. The effect is that the small state is passed before enough points are seen to transition into it and the state transition logic gets lost and throws an anomaly.

It is important to realize that when a cell is contains an “F”, it does not imply that the anomaly detection system simply spits out either “normal” or “abnormal” when testing on a time series. The state transition logic identifies which state the anomalies occurred in. So, for each “F” in Table 6.1 and Table 6.2, it may indicate that a single anomaly was found in a single state, or than dozens of anomalies were found in every state. In other words, not all “F”s are created equal, some may be only one anomaly, or one state away from passing without returning an anomaly. Nearly all of the test cases that found anomalies unexpectedly (red cells) contained only a very small number of anomalies in one or two states.

6.4 Summary

This chapter has discussed how do extend the anomaly detection system described in Chapter 5 to allow training on multiple time series. Two additional steps are required to train on multiple time series. The first is to merge multiple time series into a single time series before clustering. The second is to use the states identified for the merged time series to identify the states in all of the original time series that were used to create the merged time series. Both of these additional steps are performed using dynamic time warping. Dynamic time warping is able to "line up" multiple time series so they can be merged together, and it can find corresponding sections of different time series to use the merged state information to determine the states in the other time series.

Training on multiple time series is necessary to determine the amount of variation that is permitted during the normal operation of a device. Our empirical results indicate that training on multiple data sets does generalize the model to correctly cover time series that are "between" the time series trained on. Additionally, the model does not generalize to cover time series that are anomalous. Generalization only occurs for time series "between" the observed training data.

Chapter 7

Concluding Remarks

We have detailed our approach to time series anomaly detection that discovers and characterizes the states of a time series, and performs transition logic between these states to construct a finite state automaton. This finite state automation can be run on an expert system and used to track normal behavior and detect anomalies, in monitored devices. The proposed Gecko segmentation algorithm is designed to cluster time series data (finds a small number of segments mapping to unique states rather than a fine approximation of many segments), and uses our proposed L method to determine a reasonable number of segments efficiently. The rules generated for each state by the RIPPER algorithm can be *easily understood and modified by humans*. (Moreover, the generated rules can be in a format used by the SCL expert system shell at ICS, which is our collaborator on this NASA project.)

7.1 Summary of Contributions

The following is a summary of our contributions:

- We demonstrate a method that performs time series anomaly detection via generated states and logical rules that can easily be understood and modified by humans. These logical rules can be easily converted into a format that allows the anomaly detection to be performed by an expert system. The knowledge encoded into an expert system must be typically be entered manually by a human which is a costly and time-consuming process. Also, since the rules are human-readable, the reason *why* an anomaly is reported can be understood by a human user so he or she may quickly take appropriate action.

- We introduce an algorithm named Gecko that segments a time series into states.
- We propose the L method that dynamically determines a reasonable number of clusters. The L method is general enough to be used with any hierarchical clustering or segmentation algorithm.
- We demonstrate how derivative time warping can be used to locate corresponding sequences across multiple time series, and how to merge multiple time series together into an "average" time series in order to extend our anomaly detection so it may train on multiple time series. Additionally we propose that the template to use during merging be the time series that is the most similar to every other time series based on the sum of the warp paths to every other time series. We also showed how a classification of the merged time series can be expanded to all of the input time series that were used to make the merged time series.
- Our empirical evaluations, using data from NASA, indicate that Gecko performs comparably with a NASA expert in identifying the operational states of a device. When the human domain expert was asked to rate Gecko's output with a score from 1-10, Gecko was given perfect ratings on 6 of 10 data sets and had an average score of 9.5. A perfect rating of 10 signifies that the set of segments, or clusters, produced by Gecko is equally as good as that of the human expert. For comparison, the bottom-up segmentation algorithm was also tested, and was only given an average rating of 4.3.

- Empirical evaluations, using 14 spatial and time series data sets and 6 different clustering/segmentation algorithms, indicate that our L Method performs favorably to existing methods that determine the number of clusters or segments to return. Our L method is shown to work well for a wide range of algorithms, clusters with elaborate shape, and for clusters/segments that are overlapping and not well-separated. In our evaluation, the L method was able to determine a reasonable number of segments in 10 out of 11 instances for hierarchical segmentation algorithms with greedy evaluation metrics, and a correct number of clusters in 10 of 12 instances for hierarchical clustering algorithms. The L method performed much better than the two existing methods that were also tested in our evaluation.
- Empirical evaluation also shows that the overall system can track normal behavior and detect anomalies. The overall anomaly detection system was able to detect anomalies in every time series that was from a 'damaged' valve, and was also able to monitor a 2nd normal valve without detecting any anomalies. Additionally, when training on multiple time series the anomaly detection system is able to accurately determine the amount of allowed variation in each state. Experiments have shown that when trained on time series of a valve operating at two different voltages, the normal model generated is able to generalize to cover all time series operating at voltages between those two voltages, and will still detect anomalies for valves operating and higher or lower voltages than observed during training.

7.2 Limitations and Future Work

Our anomaly detection trains on one or more normal time series and is then able to determine whether future time series are anomalous. However, our anomaly detection system has some limitations, some of which will be the focus of future work.

Data Used for Training:

- A model of a normal time series consists of a sequential ordering of operational states, and obviously requires training data that also contains states that occur in the same order. Our model can handle differing amounts of allowed variation among the states, but it will not work correctly if the states do not need to occur in the same sequence during normal operation.
- The anomaly detection discussed in this paper has a start state and end state, and therefore is not able to monitor long or cyclical processes. However, if the training data contains various individual cycles, and the normal model created generalizes the permitted cycles, a simple modification to the state machine will allow normal transitions from the last state to the first state. Future work will be done to test our system on cyclic data such as the time series of a heartbeat.
- In a operational state identified by Gecko, all of the different dimensions (sensor values) of the time series are assumed to be related to each other and will have similar values in the same operational state in another normal time series. However, if some of the dimensions are unrelated to each other, restrictions in the normal model that require them to have a similar relationship in all future instances of that state are unreasonable. If the values of dimensions or sensors are unrelated to each other, they should not be used together when building the model. They should be separated and used to create more than one model where each model performs anomaly detection on a different subsystem.

- If multiple time series are used for training, each time series has to be somewhat similar. A large amount of variation is permitted, but they cannot differ too much or they will not be able to be correctly merged into a single time series before clustering. Each time series should be of a device performing a similar operation under varying conditions, rather than a device performing operations that are entirely different that produce time series that seem unrelated to each other.

Gecko (finding states):

- The Gecko algorithm explained in this paper cannot be scaled up to very large time series because the first phase that finds the initial sub-clusters is not scalable. The graph bisection operations become slow as the size of the time series increases. Future work will look into creating a scalable top-down segmentation algorithm that will be able to create a large number of small clusters that to not span important cluster boundaries.

- During clustering with Gecko, if there the data is noisy, the parameter *minClusterSize* needs to be increased to prevent the algorithm from returning "clusters" that are actually only noise and not true clusters. Having the algorithm determine a good value for this parameter automatically will always be possible the smallest size of a state that should be returned may often depend more on the domain than the data. However, future work should explore ways to set this parameter automatically that will at least be an improvement over a single static value or requiring the user to set it every time.

L Method (determining the number of clusters/segments):

- The major limitation of the L method is that it will never suggest that less than three clusters be returned. Future work will look into way to perform extra analysis if the number of clusters returned is only three to determine if two or one clusters would actually be a better choice.

Dynamic Time Warping:

- The standard dynamic warping (DTW) algorithm runs quickly if the time series contains fewer than three thousand points. However, it has an O(N^2) time complexity, and even worse, an O(N^2) space complexity to fill the cost matrix. This means that performing DTW on a time series that contains 20,000 measurements requires that 400 *million* ($20{,}000^2$) floating point numbers be stored in the cost matrix. There are approximations to the DTW algorithm that only calculate the values of the cost matrix near the linear warp line. However, with time series that contain steady state conditions (flat regions) at the beginning or end, the warp path may have to stray very far from the linear warp line. Straying too far from the linear warp line would cause the complexity of DTW to approach O(N^2) as the number of cells that need to be evaluated increases. Other methods simply perform classic DTW on a reduced time series which may produce a pretty accurate warp path distance measurement, but the actual path may be poor in some regions, which would bad for our algorithm and the relationships between time series that we infer using warp paths. Future work will focus on creating a new dynamic time warping algorithm that warps a reduced time series, and then iteratively refines the warp path locally after the rough path is calculated. If successful, the algorithm would scale linearly with the size of the time series, would be near-optimal, and would have no poor warpings between points.

Rule Generation:

- Our work so far has only generated a set of Rules using the RIPPER algorithm. This one set of rules is used for both state transitions and anomaly detection (anomaly is thrown when state transition logic fails).

However, whether to transition between states and whether a data point is anomalous are two different problems. It may be better to generate a more general set of rules for state transitions, and a more restrictive set of rules for anomaly detection. Currently the single set of rules is more similar to the more specific anomaly detection rules. It may be beneficial to have a second set of rules that are only specific enough to tell if a point is in the current or next state, and simultaneously use the more restrictive set of rules for anomaly detection. In this manner, the state machine will not need to get "lost" for an anomaly to be reported.

- Future work will evaluate alternatives to the RIPPER algorithm to generate rules. A possibility is simply to record the minimum and maximum values for each dimension in a state to draw a "box" around the area of permitted values.

State-Transition Logic:

- Our current state transition logic has a consecutive transition threshold that prevents premature state transitions on spurious points. However, requiring a consecutive number of transition points to actually perform a transition has a side effect that requires a state contain a minimum number of points. If one or more states are skipped and it is normal for them to be skipped based on it being observed in training, the current state transition logic will incorrectly throw an anomaly at the skipped state. Future work will extend state transition logic to identify states in the merged time series, but also determine which states in the merged time series do not occur in the original time series used for training. If a particular state is sometimes skipped during training (normal behavior), the state transition logic needs to accommodate this by allowing the previous state to either transition to the next sometimes-skipped state OR two states ahead.

- Sometimes a small state may be mapped to an even smaller state in a time series used during training, and it will create a state that is so small that it is difficult to transition into before it is passed (similar to the previous bullet). Our current anomaly detection system runs a non-deterministic state machine only when an anomaly is found, and runs until it converges to a single state. However, to avoid the problem of having states that are too small to easily transition into, a non-deterministic state machine can always be run. In the always running non-deterministic state machine, the first transition point causes the state machine to both transition AND stay in the same state in different threads. If a thread gets stuck, it is terminated, and an anomaly is only thrown if all threads are unable to continue tracking through the time series.

References

Abdulla, W., D. Chow, and G. Sin, Cross-words reference template for DTW-based speech recognition systems, in *Proc. IEEE TENCON*, Bangalore, India, 2003.

Baxter, R. A. and J. J. Oliver, The Kindest Cut: Minimum Message Length Segmentation, In *Algorithmic Learning Theory, 7th Intl. Workshop*, pp.83-90, Sydney, Australia, 1996.

Caudell, T. and D. Newman, An Adaptive Resonance Architecture to Define Normality and Detect Novelties in Time Series and Databases, In *Proc. IEEE World Congress on Neural Networks*, pp. IV166-176, Portland, OR, 1993

Chiu, T., D. Fang, J. Chen, Y. Wang and C. Jeris, A Robust and Scalable Clustering Algorithm for Mixed Type Attributes in Large Database Environment, In *Proc. Of the 7th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pp. 263-268, San Francisco, CA, 2001.

Cohen, W., Fast Effective Rule Induction, In *Proc. of the 12th Intl. Conf. on Machine Learning*, pp. 115-123, Tahoe City, CA, 1995.

Dasgupta, D. and S. Forrest, Novelty Detection in Time Series Data using Ideas from Immunology, In *Proc. Fifth Intl. Conf. on Intelligent Systems*, pp. 82-87, Reno, NV, 1996.

Ester, M., H. Kriegel, J. Sander, and X. Xu, A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, In *Proc. 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pp. 226-231, Portland OR, 1996.

Fiduccia, C. and R. Mattheyses, A linear-time heuristics for improving network partitions, In *Proc. of the 19th Design Automation Conference*, pp. 175-181, Las Vegas, NV, 1982.

Foss, A. and A. Zaïane, A Parameterless Method for Efficiently Discovering Clusters of Arbitrary Shape in Large Datasets, In *IEEE Intl. Conf. on Data Mining*, pp. 179-186, Maebashi City, Japan, 2002.

Fraley, C. and E. Raftery, How many clusters? Which clustering method? Answers via model-based Cluster Analysis, In *Computer Journal*, vol. 41, pp. 578-588, 1998.

Furnkranz, J. and G. Wildmer, Incremental Reduced Error Pruning, In *Proc. of the 11th Intl. Conf. on Machine Learning*, pp. 70-77, New Brunswick, NJ, 1994.

Guha, S., R. Rastogi and K. Shim, CURE: An Efficient Clustering Algorithm for Large Databases, In *Proc. Of ACM SIGMOD Intl. Conf. on Management of Data*, pp. 73-84, New York, NY, 1998.

Guha, S., R. Rastogi, and K. Shim, ROCK: A Robust Clustering Algorithm for Categorical Attributes, In *The 15th Intl. Conf. on Data Engineering*, pp. 512-523, Sydney, Australia, 1999.

Han, J. and M. Kambler, *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publishers, San Mateo, CA, 2000.

Hansen, M. and B. Yu, Model Selection and the Principle of Minimum Description Length, In *JASA*, vol. 96, pp.746-774, 2001.

Harris, S., D. Hess and J. Venegas, An Objective Analysis of the Pressure-Volume Curve in the Acute Respiratory Distress Syndrome, In *American Journal of Respiratory and Critical Care Medicine*, vol. 161, no. 2, pp. 432-439, 2000.

Hartigan, J., *Clustering Algorithms*, John Wiley & Songs, New York, NY, 1975.

Hinneburg, A and D. Keim, An Efficient Approach to Clustering in Large Multimedia Databases with Noise, In *Proc 4th Intl. Conf. on Knowledge Discovery and Data Mining*, New York City, NY, 1998, pp. 58-65.

Karypis, G., E. Han, and V. Kumar, Chameleon: A hierarchical clustering algorithm using dynamic modeling, *IEEE Computer*, 32(8), pp. 68-75, 1999.

Keogh, E., S. Chu, D. Hart, and M. Pazanni, An Online Algorithm for Segmenting Time Series, In *Proc. IEEE Intl. Conf. on Data Mining*, San Jose, CA, pp. 289-296, 2001.

Keogh, E. and T. Folias. The UCR Time Series Data Mining Archive [http://www.cs.ucr.edu/~eamonn/TSDMA/ index.html], Riverside, CA, University of California – Computer Science and Engineering Department, 2003

Keogh, E., S. Lonardi, and B. Chiu, Finding Surprising Patterns in Time Series Database in Linear Time and Space, In *Proc. ACM Knowledge Discovery and Data Mining*, pp. 550-556, Edmonton, Canada, 2002.

Keogh, E. and M. Pazzani, Derivative Dynamic Time Warping, In *Proc. of the 3rd European Conf. on Principles and Practice of Knowledge Discovery in Database*, pp. 1-11, Prague, Czech Republic, 1999.

Keogh, E. and M. Pazzani, Scaling up Dynamic Time Warping for Datamining Applications, In *Proc. of the 6th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pp. 285-289, Boston, MA, 2000.

Kozama, R., M. Kitamura, M. Sakuma, and Y. Yokoyama, Anomaly Detection by neural network models and statistical time series analysis, In *Proceedings of IEEE Int. Conf. on Neural Networks*, pp. 613-618, Orlando, FL, 1994.

Kruscall, J. and M. Liberman. The symmetric time warping algorithm: From continuous to discrete, In *Time Warps, String Edits, and Macromolocules: The theory and Practice of String Comparison*, Addison-Wesley, 1983.

Langley, P., G. Bay, and K. Saito, Robust Induction of Process Models from Time-Series Data, In *Proc. of the 20th Intl. Conf. on Machine Learning*, pp. 432-439, Washington, DC, 2003.

Monti, S., T. Pablo, J. Mesirov and T. Golub, Consensus Clustering: A Resampling-Based Method for Class Discovery and Visualization of Gene Expression Microarray Data, In *Machine Learning*, vol. 52, pp. 91-118, 2003.

Ng, R. and J. Hah, Efficient and Effective Clustering Methods for Spatial Data Mining, In *The 20th Intl. Conf. On Very Large Data Bases*, Santiago, Chile, pp. 12-15, 1994.

Roth, V., T. Lange, M. Braun and J. Buhmann, A Resampling Approach to Cluster Validation, In Proc. *in Computational Statistics: 15th Symposium (COMPSTAT2002),* Berlin, Germany, pp. 123-128, 2002.

Seikholeslami G., S. Chatterjee, and A. Zhang, WaveCluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases, In *Proc. of the 24th VLDB*, pp. 428-439, New York City, New York, 1998.

Smyth, P., Clustering Using Monte-Carlo Cross-Validation, In *Proc. 2nd KDD*, Portland, OR, pp.126-133, 1996.

Sugiyama, M. and H. Ogawa, Subspace Information Criterion for Model Selection, In *Neural Computation*, vol. 13, no.8, pp. 1863-1889, 2001.

Tibshirani, R., G. Walther, D. Botstein, and P. Brown, Cluster Validation by Prediction Strength, Technical Report, 2001-21, Dept. of Biostatistics, Stanford Univ, 2001.

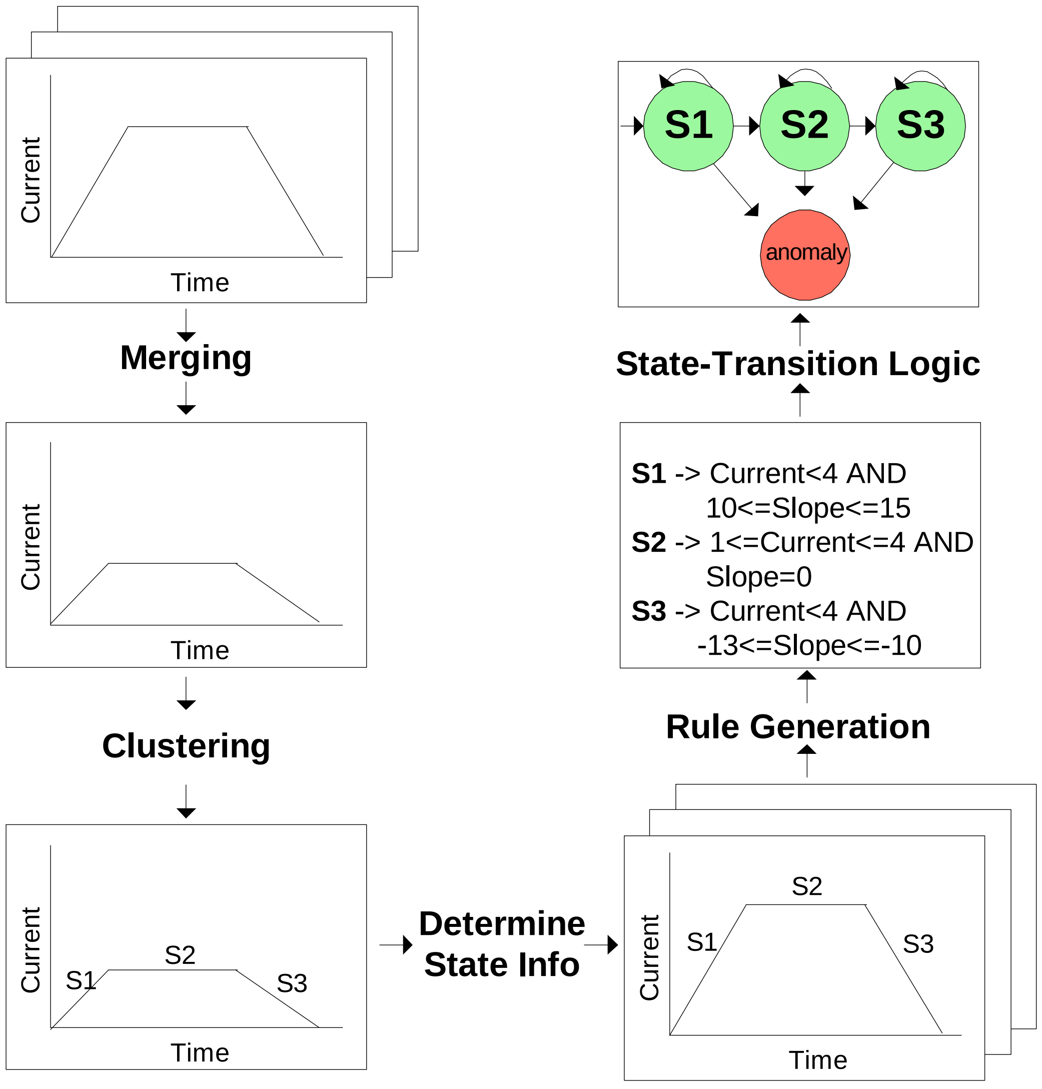
Tibshirani, R., G. Walther, and T. Hastie, Estimating the number of clusters in a dataset via the Gap statistic, In *JRSSB*, 2003.

Vasko, K. and T. Toivonen, Estimating the number of segments in time series data using permutation tests, In *Proc. IEEE Intl. Conf. on Data Mining*, pp. 466-473, Maebashi City, Japan, 2002.

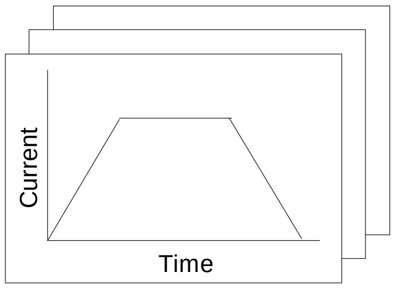
Zhang, T., R. Ramakrishnan, and M. Livny, BIRCH: An Efficient Data Clustering Method for Very Large Databases, In *ACM SIGMOD Intl. Conf. on Management of Data and Symposiium on Principles of Database Systems*, pp. 103-114, Montreal, Canada, 1996

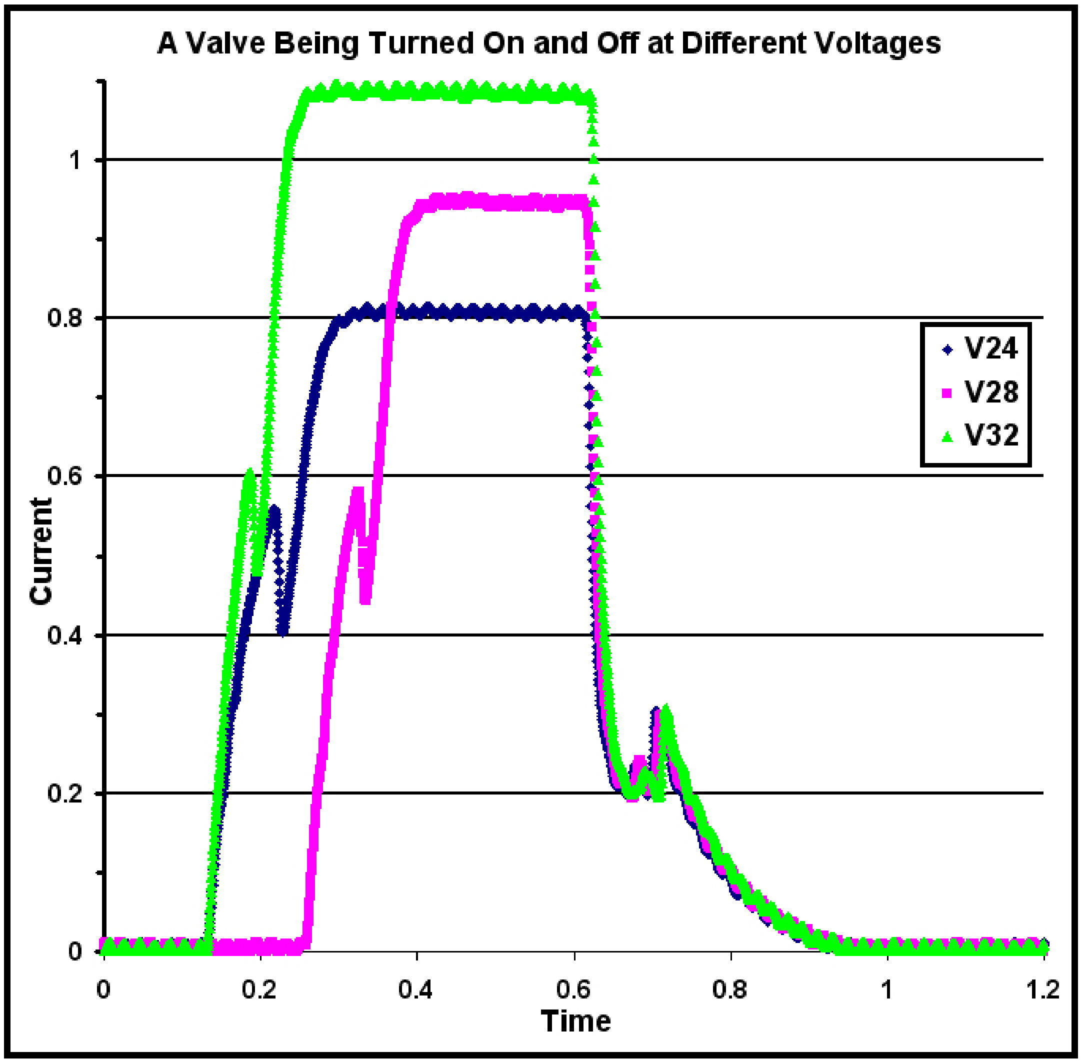
Appendix

Sample Run of the Anomaly Detection System

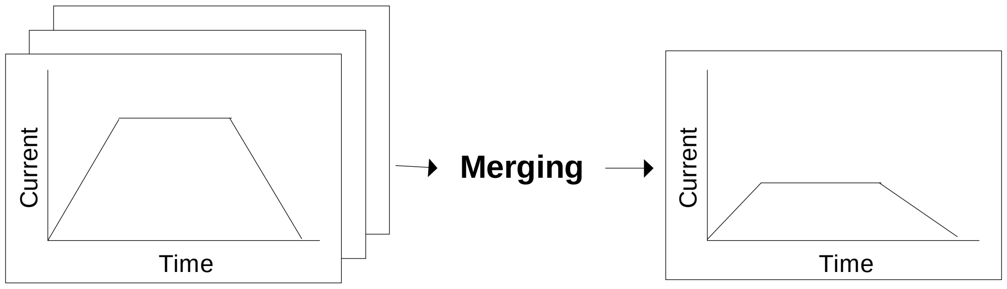


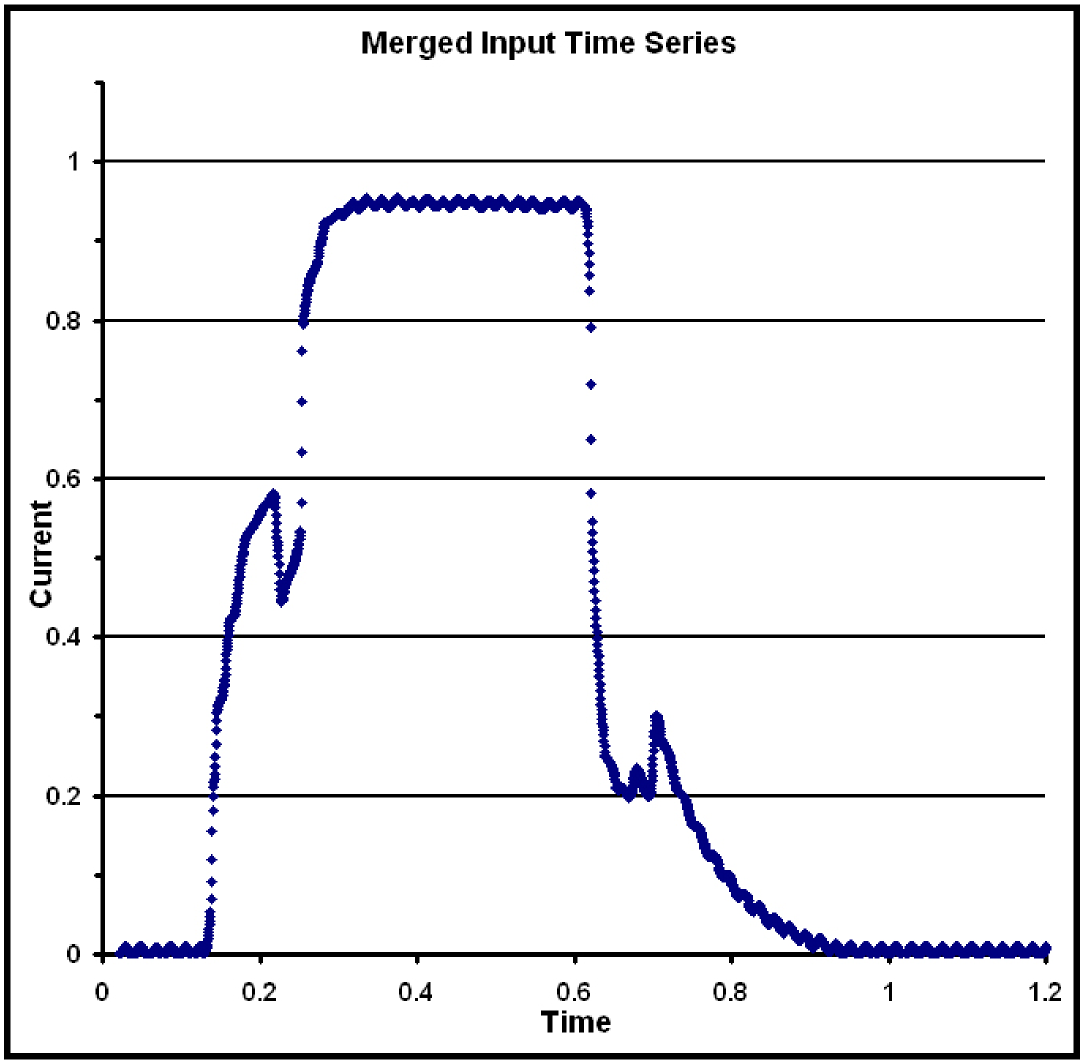
Input Files for Training



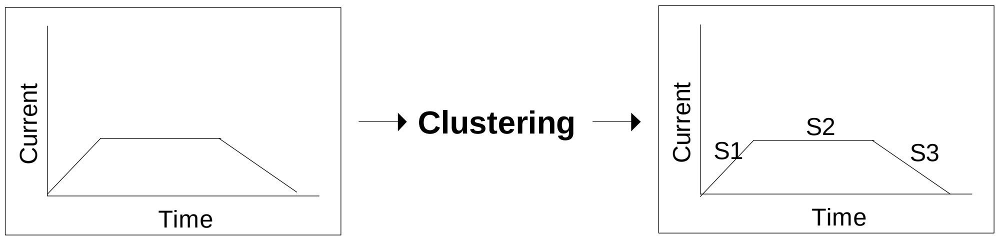


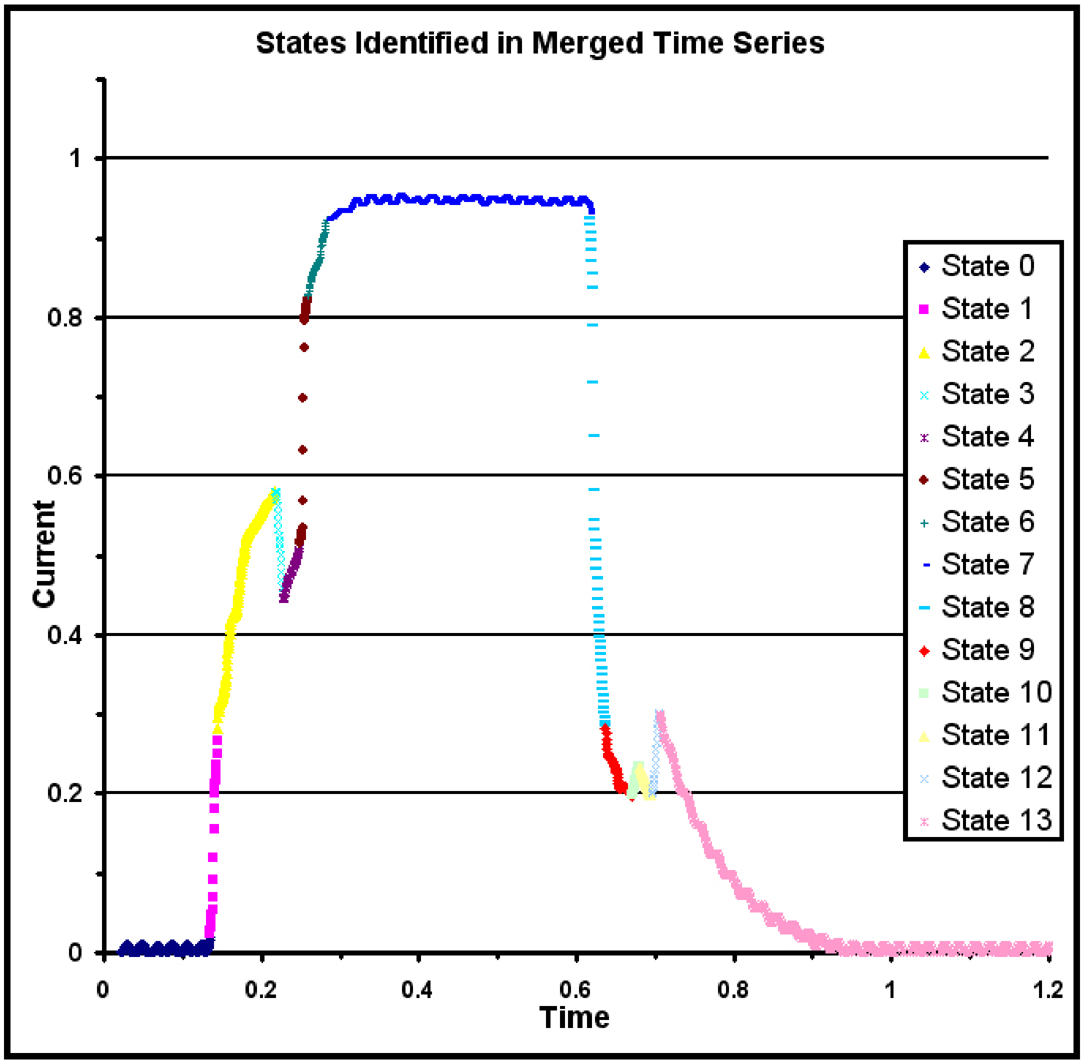
Merging Time Series



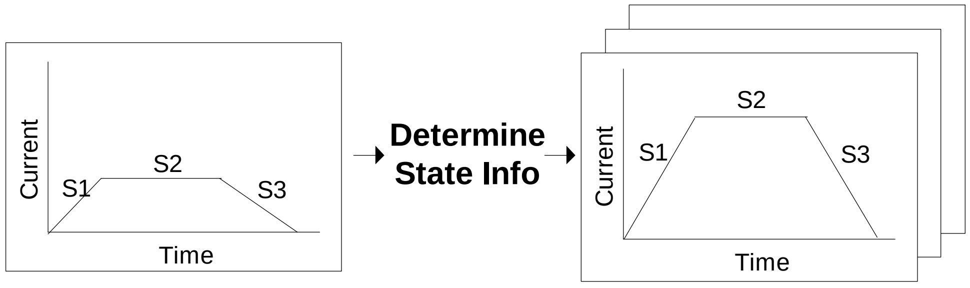


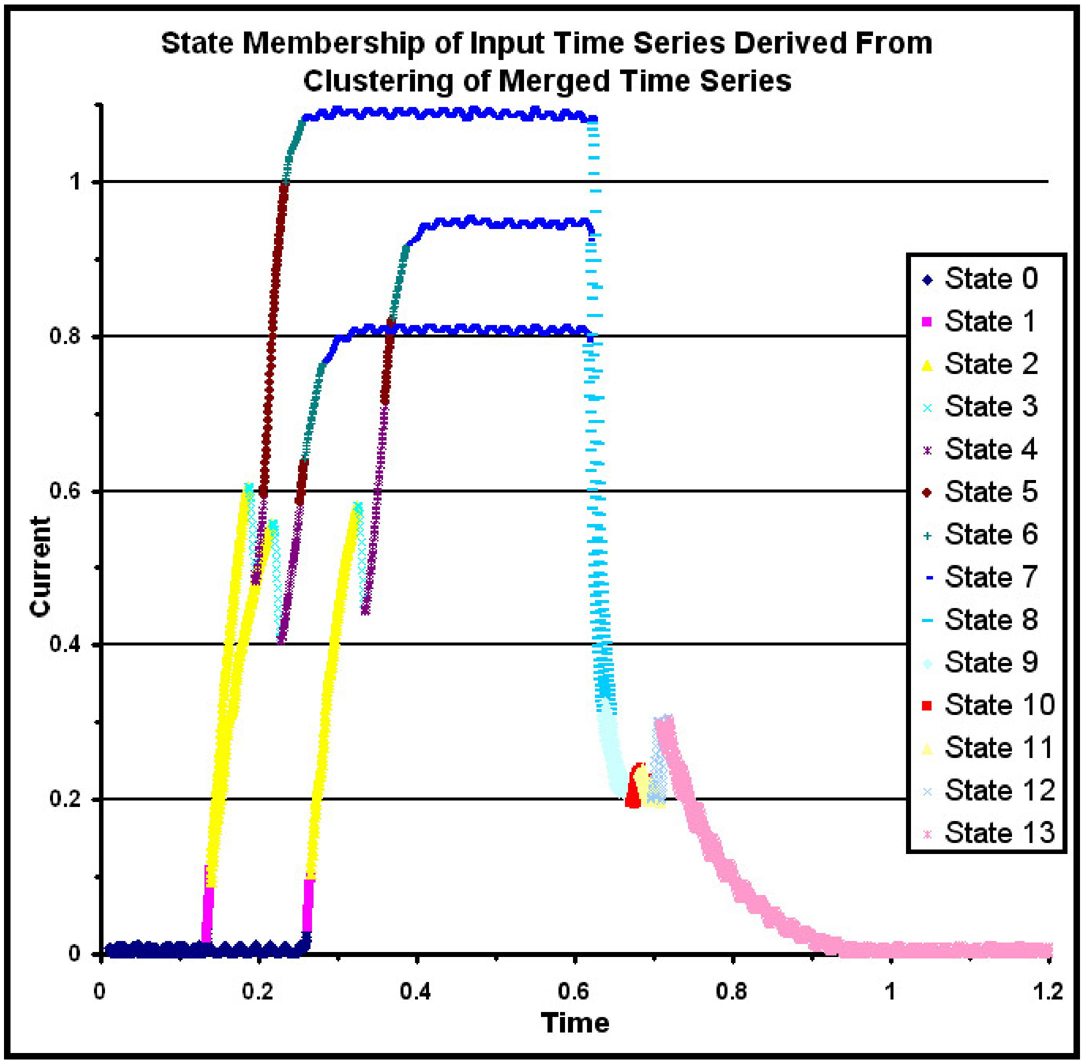
Identifying States



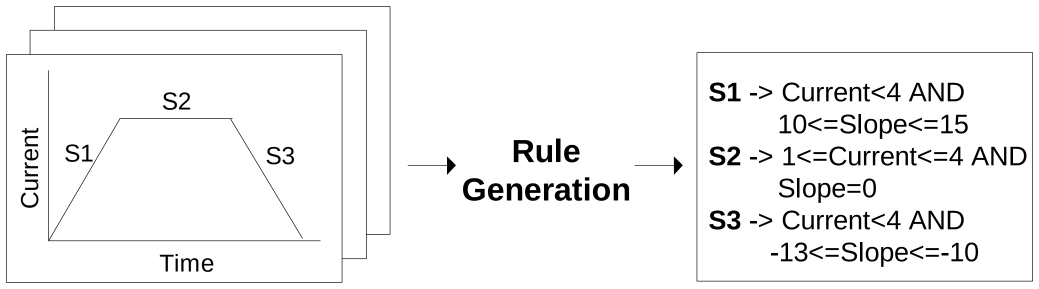


Determine State Information





Rule Generation


Current
S1
S2
S3
Time
Rule Generation
S1 -> Current<4 AND 10<=Slope<=15
S2 -> 1<=Current<=4 AND Slope=0
S3 -> Current<4 AND -13<=Slope<=-10

```
RULE FOR STATE 0:
IF (((Current <= 0.01167725)
   AND (d1_Current >= -2.00887) AND (d1_Current <= 7.41389)
   AND (d2_Current >= -428.885))
OR ((Current >= 0.01104155) AND (Current <= 0.0214917000000000002)
   AND (d1_Current >= -0.5117375)
   AND (d1_Current <= 8.469335000000001)
   AND (d2_Current >= -437.142))
OR ((Current >= 0.01071215) AND (Current <= 0.010840550000000001))
OR ((Current >= 0.010775650000000001) AND (Current <= 0.01090405)))
THEN 0


RULE FOR STATE 1:
IF (((Current <= 0.459586)
   AND (d1_Current >= 8.70961))
OR ((Current <= 0.465255)
   AND (d1_Current >= 9.527095))) THEN 1


RULE FOR STATE 2:
IF (((Current >= 0.1536815) AND (Current <= 0.5997954999999999)
   AND (d1_Current >= 2.35799)
   AND (d1_Current <= 12.481200000000001)
   AND (d2_Current >= -2122.735) AND (d2_Current <= 588.989))
OR ((Current >= 0.552237) AND (Current <= 0.6042005)
   AND (d1_Current >= 0.371603)
   AND (d2_Current <= -1205.78))
OR ((d2_Current >= -1851.675)
   AND (d2_Current <= -1840.4299999999998))) THEN 2
```

RULE FOR STATE 3:

```
IF (((Current >= 0.4248265) AND (Current <= 0.6049884999999999)
   AND (d1_Current >= -18.57515) AND (d1_Current <= -3.96586)
   AND (d2_Current <= 2278.74))
OR ((Current <= 0.6049344999999999)
   AND (d1_Current <= -0.7938265)
   AND (d2_Current <= -2355.495))
OR ((Current >= 0.4165025) AND (Current <= 0.6075254999999999)
   AND (d1_Current >= -12.3338) AND (d1_Current <= -2.1199)
   AND (d2_Current <= 2568.2250000000004))
OR ((d2_Current >= -2088.535)
   AND (d2_Current <= -2051.0699999999997))) THEN 3
```

RULE FOR STATE 4:

```
IF (((Current >= 0.4072015) AND (Current <= 0.5463979999999999)
   AND (d1_Current >= 2.9014949999999997)
   AND (d1_Current <= 8.54674) AND (d2_Current >= -13.97575))
OR ((d1_Current >= -10.99385)
   AND (d2_Current >= 2568.2250000000004))
OR ((d1_Current >= -11.2563)
   AND (d2_Current >= 2253.8))
OR ((d1_Current >= 7.234265) AND (d1_Current <= 7.260475))) THEN 4
```

RULE FOR STATE 5:

```
IF (((Current >= 0.47907900000000003)
   AND (d1_Current >= 8.26221))) THEN 5
```

RULE FOR STATE 6:

```
IF (((Current >= 0.647008) AND (Current <= 1.07862)
   AND (d1_Current >= 1.9650699999999999) AND (d1_Current <=
8.073385))
OR ((d2_Current >= -699.8389999999999)
   AND (d2_Current <= -697.466))
OR ((Current >= 1.079925)
   AND (d1_Current >= 2.242155))) THEN 6
```

RULE FOR STATE 7:

```
IF (((Current >= 0.7691555)
   AND (d1_Current >= -16.107) AND (d1_Current <= 2.4305))
OR ((Current >= 0.7684335) AND (Current <= 0.7691705))) THEN 7
```

RULE FOR STATE 8:

```
IF (((Current <= 1.07578)
   AND (d1_Current <= -7.889145))) THEN 8
```

RULE FOR STATE 9:

```
IF (((Current >= 0.194583) AND (Current <= 0.3004755)
   AND (d1_Current >= -7.752935) AND (d1_Current <= 0.532424)
   AND (d2_Current >= -298.1875) AND (d2_Current <= 1033.72))
OR ((d2_Current >= 202.7235)
   AND (d2_Current <= 202.98649999999998))
OR ((d2_Current >= 937.5535) AND (d2_Current <= 944.7975))) THEN 9
```

RULE FOR STATE 10:

```
IF (((Current >= 0.19351849999999998)
   AND (Current <= 0.24175249999999998)
   AND (d1_Current >= 0.6799845) AND (d1_Current <= 5.983625)
   AND (d2_Current <= 1007.0))
OR ((Current >= 0.2270345) AND (Current <= 0.241927)
   AND (d1_Current >= 0.39421649999999997)
   AND (d2_Current <= -429.7875))
OR ((d2_Current >= 224.6855) AND (d2_Current <= 225.2665))) THEN 10
```

RULE FOR STATE 11:

```
IF (((Current >= 0.1982835) AND (Current <= 0.241559)
   AND (d1_Current >= -3.95517) AND (d1_Current <= 2.346055))
OR ((d2_Current >= 1910.425) AND (d2_Current <= 1918.715))
OR ((d2_Current >= 1780.71)
   AND (d2_Current <= 1794.8400000000001))) THEN 11
```

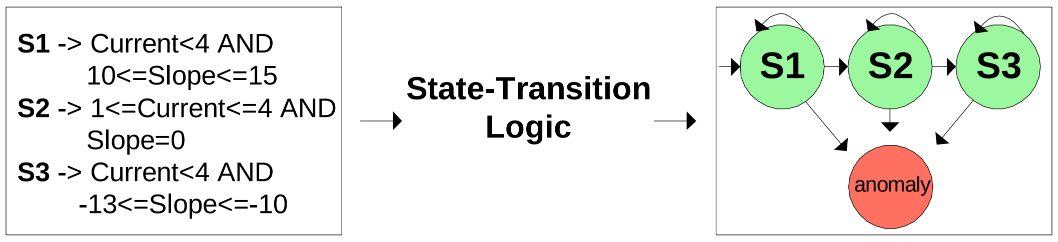
RULE FOR STATE 12:

```
IF (((Current >= 0.196977) AND (Current <= 0.306535)
   AND (d1_Current >= 3.04138)
   AND (d1_Current <= 13.565850000000001))) THEN 12
```

RULE FOR STATE 13:

```
IF (((Current >= -0.004448185) AND (Current <= 0.193331)
   AND (d1_Current <= 1.56491)
   AND (d2_Current <= 454.515))
OR ((Current >= -0.00628144) AND (Current <= 0.3060665)
   AND (d1_Current >= -5.318434999999999)
   AND (d1_Current <= 1.970375)
   AND (d2_Current <= 470.1575))
OR ((Current <= 0.307395)
   AND (d1_Current <= 3.1861050000000004)
   AND (d2_Current <= -9.40672))) THEN 13
```

State-Transition Logic

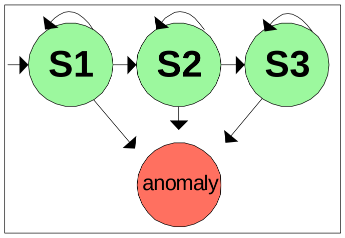

S1 -> Current<4 AND 10<=Slope<=15
S2 -> 1<=Current<=4 AND Slope=0
S3 -> Current<4 AND -13<=Slope<=-10
State-Transition Logic
S1
S2
S3
anomaly

```
script transition_to_merged_0

   if merged_enable_trace > 0 then
      message "script transition_to_merged_0"
   end if

   merged_active_state = 0
   merged_consecutive_error_count = 0
   merged_cumulative_transition_count = 0
   merged_consecutive_transition_count = 0
   merged_num_valid_points = 0

   activate monitor_transition_to_merged_1
   activate monitor_merged_0_curr
   activate monitor_merged_0_next
   activate monitor_merged_0_error

   if merged_enable_trace > 0 then
      execute merged_data_dump
   end if

end transition_to_merged_0


Rule monitor_merged_0_curr

   subsystem owner_of_merged
   category merged_valve_monitor
   priority 20
   activation yes
   continuous yes

   -- test for merged cluster 0
   if merged_active_state = 0 and
      (((merged_Current >= -0.00564919) and
        (merged_Current <= 0.012188049999999999)
      and (merged_d1_Current >= -2.0479950000000002))) then

      if merged_enable_trace > 0 then
         message "monitor_merged_0_curr"
      end if

      -- this point is in active cluster, zero consecutive counts
      merged_consecutive_error_count = 0
      merged_consecutive_transition_count = 0

      -- bump number of valid points found
      increment merged_num_valid_points

   end if

end monitor_merged_0_curr

Rule monitor_merged_0_next

   subsystem owner_of_merged
   category merged_valve_monitor
   priority 20
   activation yes
   continuous yes

   -- test for merged cluster 1
   if merged_active_state = 0 and
      (((merged_Current >= 0.01253285) and
        (merged_Current <= 0.40316300000000005)
      and (merged_d1_Current >= 1.84109) and
          (merged_d1_Current <= 11.075700000000001)
      and (merged_d2_Current >= -892.1320000000001) and
          (merged_d2_Current <= 853.9445000000001))
   or ((merged_Current >= 0.02014075) and
       (merged_Current <= 0.28492700000000004)
      and (merged_d1_Current >= -0.2783255) and
          (merged_d1_Current <= 7.344745)
      and (merged_d2_Current <= 900.0745))
   or ((merged_Current >= 0.2523655) and
       (merged_Current <= 0.2545855)
      and (merged_d1_Current >= -0.4122285) and
          (merged_d1_Current <= -0.3237645))
   or ((merged_d1_Current >= 1.6590099999999999) and
       (merged_d1_Current <= 1.67289))) then

      if merged_enable_trace > 0 then
         message "monitor_merged_0_next"
      end if

      -- this point is in next cluster, try to transition
      increment merged_cumulative_transition_count
      increment merged_consecutive_transition_count
      merged_consecutive_error_count = 0

   end if

end monitor_merged_0_next

Rule monitor_merged_0_error

   subsystem owner_of_merged
   category merged_valve_monitor
   priority 20
   activation yes
   continuous yes

   -- test for merged cluster 0 error condition
   -- that is point NOT in 0 AND NOT in 1
   if merged_active_state = 0 and
      (NOT (((merged_Current >= -0.00564919) and
             (merged_Current <= 0.012188049999999999) and
             (merged_d1_Current >= -2.0479950000000002)))) and
      (NOT (((merged_Current >= 0.01253285) and
             (merged_Current <= 0.40316300000000005) and
             (merged_d1_Current >= 1.84109) and
             (merged_d1_Current <= 11.075700000000001) and
             (merged_d2_Current >= -892.1320000000001) and
             (merged_d2_Current <= 853.9445000000001))
           or ((merged_Current >= 0.02014075) and
              (merged_Current <= 0.28492700000000004) and
              (merged_d1_Current >= -0.2783255) and
              (merged_d1_Current <= 7.344745) and
              (merged_d2_Current <= 900.0745))
           or ((merged_Current >= 0.2523655) and
              (merged_Current <= 0.2545855) and
              (merged_d1_Current >= -0.4122285) and
              (merged_d1_Current <= -0.3237645))
           or ((merged_d1_Current >= 1.6590099999999999) and
              (merged_d1_Current <= 1.67289)))) then

      if merged_enable_trace > 0 then
         message "monitor_merged_0_error"
      end if

      -- this point is in error, bump error counts
      increment merged_cumulative_error_count
      increment merged_consecutive_error_count
      merged_consecutive_transition_count = 0

   end if

end monitor_merged_0_error
```

```
Rule monitor_transition_to_merged_1

   subsystem owner_of_merged
   category merged_valve_monitor
   priority 20
   activation yes
   continuous yes

   if(merged_cumulative_transition_count >
merged_cumulative_transition_threshold)

or(merged_consecutive_transition_count>merged_consecutive_transitio
n_threshold)then

      if merged_enable_trace > 0 then
         message "monitor_transition_to_merged_1"
      end if

      execute transition_to_merged_1 in 1 tick
   end if

end monitor_transition_to_merged_1
```

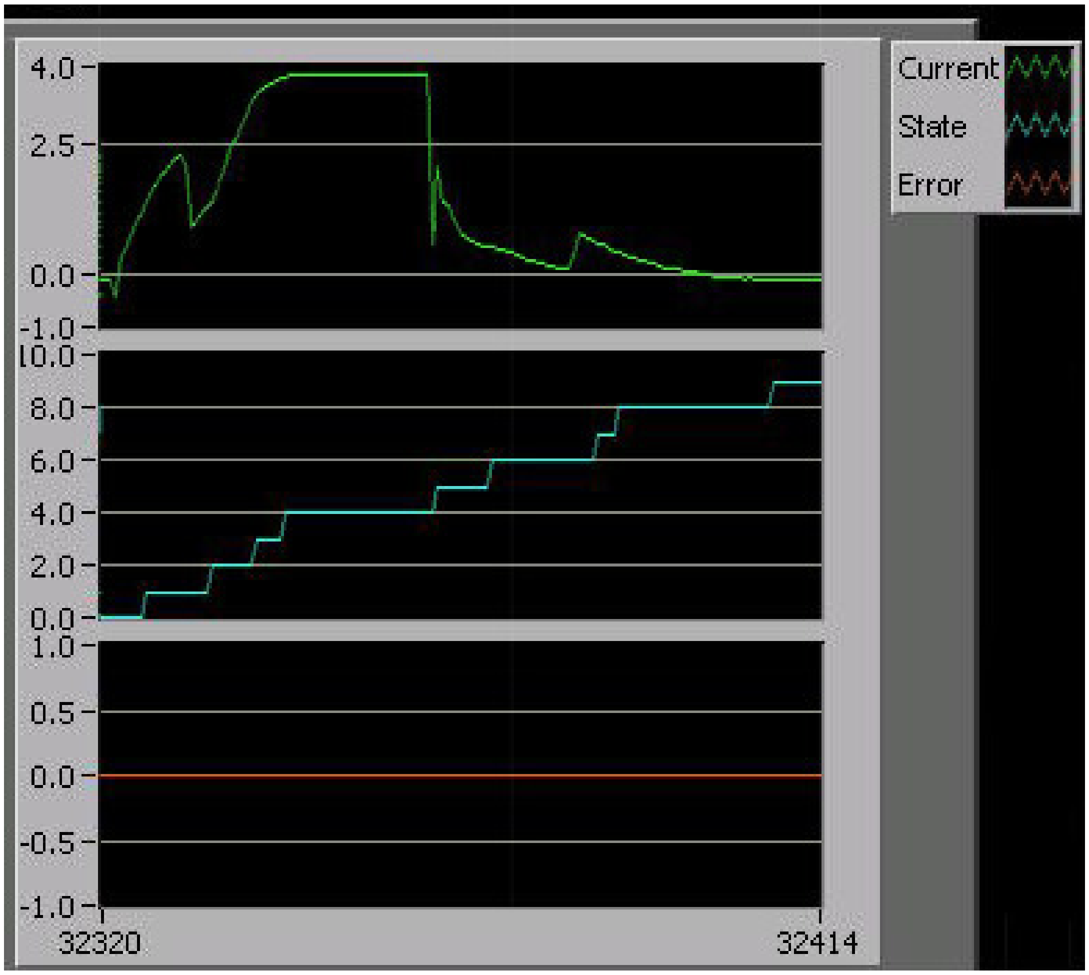
.
.
.

Anomaly Detection



Screen capture courtesy of Interface & Control Systems (ICS)

3 plots: tested time series, state tracking, error level



Usage
(command-line interface)

```
C:\>createRules v28.csv v24.csv v32.csv
- - - - - - - - - - - - - - -
Running Pre-Processor to Smooth, Normalize, and Derive new
Attributes:
Pre-Processing v28.csv....done
Pre-Processing v24.csv....done
Pre-Processing v32.csv....done
- - - - - - - - - - - - - - -
- - - - - - - - - - - - - - -
Running Dynamic Time Warper...
Determining which time series is most normal...done in 13
seconds
The Most 'Average' Input Time Series is:  v24.csv
Creating a merged time series..............done in 0.21
seconds.
- - - - - - - - - - - - - - -
Pre-Processing merged.csv....done
- - - - - - - - - - - - - - -
Running the Clustering Algorithm Gecko
LEAVE FIELDS BLANK AND PRESS RETURN TO AUTOMATICALLY USE
DEFAULT VALUES:

How would you like to set the parameters for Gecko?
   1) Simple [DEFAULT]  (2 parameters to set)
   2) Advanced          (5 parameters to set)
->1

What is the 'Minimum Cluster Size' that you want to be
possible?
DEFAULT = 10
Increasing this value makes the algorithm more tolerant to
noise, but if the setting is too large, smaller clusters will
not be properly found.  Value must be >=5 but is recommended
to be >= 10 for smooth data, and much greater than 10 for
noisy data.
->20

What is the 'Low Slope Sensitivity' setting?
DEFAULT = 0.0
Data sets with a high sample rate or where changes in the
non-time dimensionss
```

```
happen much more slowly than time will tend to need higher
values.  A setting of '0.0' typically works well, but may
need to be changed for the best results.  Lower settings are
more tolerant to noise.  Consult the manual for a full
explaination on setting this parameter.
->0.0

Loading data from file...loading complete in 0.681 seconds.
2411 points sampled down to...2411 points in 0 seconds.
Normalizing data...finished in 0.06 seconds.
Weighting data...finished in 0.02 seconds.
Building k-nearest neighbor graph...done in 12.638 seconds.
Splitting data into initial sub-clusters...90 sub-clusters
created in 24.105 seconds.
Merging remaining sub-clusters...completed in 1.592 seconds.

Creating output files:  14 clusters recommended.
Evaluation file:  'merged_Gecko_excel.csv' created.
- - - - - - - - - - - - - - 
- - - - - - - - - - - - - - 
Running Post-Processor to combine clustering and warp path
information
- - - - - - - - - - - - - - 
- - - - - - - - - - - - - - 
Running Ripper to create rules...done in 8 min.
- - - - - - - - - - - - - - 
- - - - - - - - - - - - - - 
Converting Ripper rules to SCL expert system format.
- - - - - - - - - - - - - - 
Done.
```