Learning States and Rules for Detecting Anomalies in Time Series

Stan Salvador **Philip Chan**

Department of Computer Sciences
Florida Institute of Technology
Melbourne, FL 32901
{ssalvado, pkc}@cs.fit.edu

Abstract

The normal operation of a device can be characterized in different temporal 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. 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.

Keywords: anomaly detection, time series, segmentation, cluster validation, clustering

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 base by hand is a time-consuming 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 be characterized in different temporal 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 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*. 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.

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 [1]. 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.

Our main contributions are: (1) we demonstrate a way to perform time series anomaly detection via generated states and rules that can easily be understood and modified by humans; (2) we introduce an algorithm named Gecko for segmenting time series data into important phases or states; (3) we propose the L method for dynamically finding a reasonable number of clusters–the L method is general enough to be used with either hierarchical clustering or segmentation algorithms [2]; (4) we integrate RIPPER and state transition logic to generate a complete anomaly detection system; (5) 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.

The next section gives an overview of related work. Section 3 provides a detailed explanation of our system, which includes the components: Gecko (clustering), RIPPER (rule generation), and state transition logic. Section 4 contains experimental evaluations of the component algorithms as well as the overall anomaly detection system, and Section 5 summarizes our study

2 Related Work

2.1 Clustering Algorithms

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. Partitioning algorithms, for example *K*-means, and PAM [3], iteratively refine a set of *k* clusters and do not scale well for larger data sets. Density-based algorithms, e.g., DBSCAN [4] and DENCLUE [5], 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. However, sharp spikes in time series data are sometimes important features and could be incorrectly determined to be noise by a density-based clustering algorithm. Hierarchical algorithms can be agglomerative and/or divisive. The agglomerative (bottom-up) approach repeatedly merges two clusters, while the divisive (top-down) approach repeatedly splits a cluster into two. ROCK [6] and Chameleon [7] are hierarchical algorithms that differ mostly in their similarity functions, which favor spherical and non-spherical clusters (respectively). Grid-based algorithms, such as WaveCluster [8], reduce the clustering space into a grid of cells which enables efficient clustering of very large datasets. This is useful for clustering a large amount of very concentrated data, but not for one-dimensional time series data. Existing clustering algorithms are not designed to cluster time series data. Our Gecko algorithm is similar to a hierarchical clustering algorithm that is able to cluster time series data by adding constraints to the merging and splitting of clusters. The main constraint added to our Gecko algorithm is that clusters must be divided cleanly along the time dimension, which makes the Gecko behave like a segmentation algorithm.

2.2 Segmentation Algorithms

Segmentation algorithms usually take time series data as input and produce a Piecewise Linear Representation (PLR) as output. PLR is a set of consecutive line segments that tightly fit the original data points. Segmentation algorithms are somewhat related to clustering algorithms in that each segment can be thought of as a cluster. However, due to their linear representation bias, segmentation algorithms are more effective at producing fine grain partitioning, rather than a smaller set of segments that represent natural clusters.

There are three common approaches to time series segmentation [9]. First, in the Sliding Window approach, a segment is grown until the error of the line is above a specified threshold, then a new segment is started. Second, in the Top-down approach, the entire time series is recursively split until the desired number of segments is reached, or an error threshold is reached. Third, the Bottom-up approach starts off with *n*/2 segments, the 2 most similar adjacent segments are repeatedly joined until either the desired number of segments, or an error threshold is reached. The sliding window approach creates poorest linear approximations but runs the quickest. Top-Down segmentation creates the best PLR but runs much slower than the other two methods. Bottom-up segmentation creates PLRs that are nearly as good as those of the top-down method, but has a much smaller time complexity than top-down segmentation.

2.3 Determining the Number of Segments/Clusters

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 [10] 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 [11], MDL [12], BIC [13], AIC, and SIC [14]. Permutation tests [15] are able to prevent segmentation algorithms from creating a PLR that over-fits the data. Resampling [16] and Consensus Clustering [17] 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.”

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 [18] and prediction strength [18]. 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 [20].
3. The first data point with a second derivative above some threshold value [21].
4. The data point with the largest second derivative [22].
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 locate 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 consider 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 smooth, 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

Nearly all of the 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 [23] and adaptive resonance theory [24]. However, Langley et al. [25] 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.

3 Approach

The input to our overall anomaly detection system is "normal" time series data (like the graph at the top left corner of Figure 1).

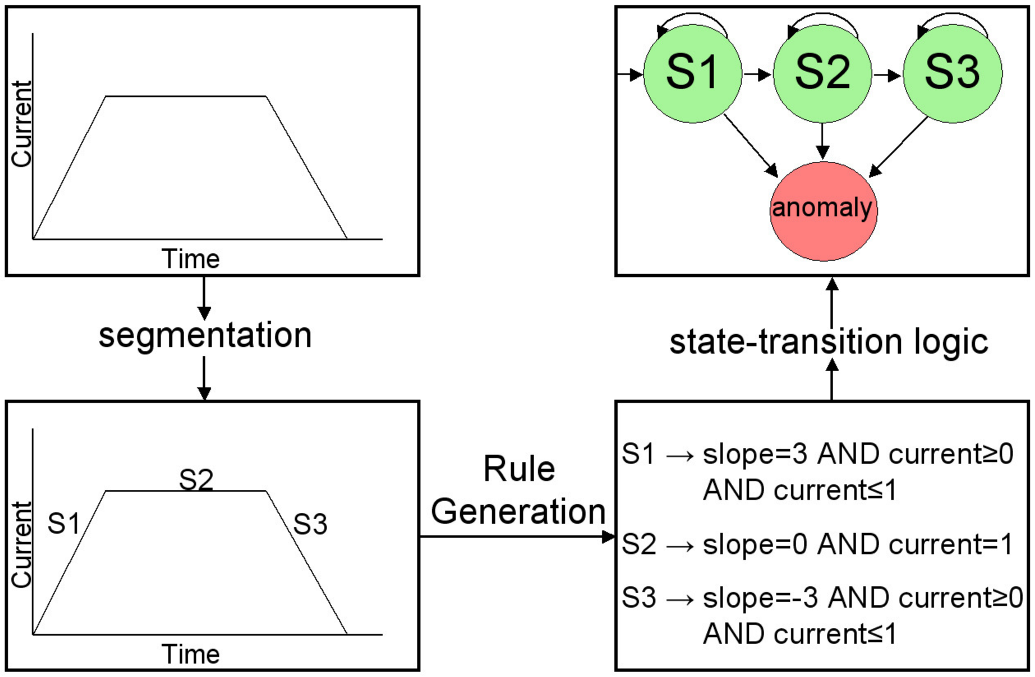
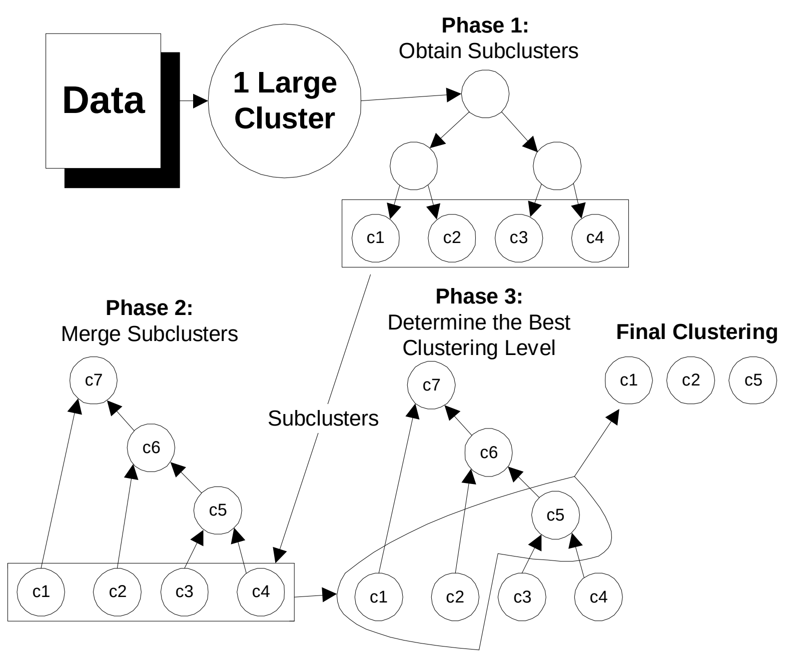

Current
Time
segmentation
S1
S2
S3
Current
Time
Rule Generation
S1 → slope=3 AND current≥0 AND current≤1
S2 → slope=0 AND current=1
S3 → slope=-3 AND current≥0 AND current≤1
state-transition logic
S1
S2
S3
anomaly

Figure 1. Main steps in time series anomaly detection.

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 signatures deviate significantly from the learned signature. Any deviation from the learned "normal" model is considered to be an anomaly. The overall architecture of the anomaly detection system, depicted in Figure 1, consists of three components: segmentation, rule generation (characterization), and state-transition logic. The segmentation phase is performed by our newly-developed segmentation algorithm "Gecko," which is designed to identify distinct states (or clusters) in a time series. Next, rules are created for each state by the RIPPER algorithm [1]. Finally, rules are added for the transitions between states to create a finite state automaton. The three steps in our approach are detailed in the next three subsections.

3.1 Gecko – Identifying States

While segmentation algorithms typically create only a fine linear approximation of time series data, Gecko divides a time series into a smaller number of segments that are analogous to clusters or states in the time series. This number of clusters is determined automatically by the algorithm.



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 2. Overview of the Gecko Algorithm.

The Gecko algorithm consists of three phases, as depicted in Figure 2. The first phase creates many small sub-clusters. The second phase repeatedly merges the two most similar clusters. Phase 3 determines the number of clusters to return.

3.1.1 *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 by Chameleon [7], 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 in the graph are the *k*/2 points on each size of a 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. However any reasonable inverse mapping between distance and similarity can be used. 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 trivialized, and the optimal min-cut partitioning of a cluster can be quickly determined in fewer than *clusterSize*-1 edge-cut checks (where *clusterSize* 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.

3.1.2 *Phase 2: Repeatedly Merge Clusters*

In phase 2, 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 accurately represent every point in a cluster because each cluster is internally homogeneous. The representative point of a cluster contains a slope value for every dimension in the time series data other than time. Clustering by the slope values causes the time series to be divided into flat regions. 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 that summarize a cluster, then the Euclidean distance between a pair of representative points with single slope values 100 and 101 (distance = 101-100=1) would be the same as the distance between a pair of representative points with slope values 0 and 1 (distance = 1-0=1). Differences in slopes that are near zero need to be emphasized, because the same absolute change in slope can triple a small value, but 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 changes of 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.

3.1.3 *Phase 3: Determine the Best Clustering Level*

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, but they are often generated by re-running the entire clustering or segmentation algorithm for every value on the *x*-axis. Since hierarchical algorithms repeatedly split or merge a pair of clusters, many sets of clusters containing '1' to '*the number of clusters in the finest-grain clustering*' clusters can be produced in only a single run of the algorithm.

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 set of clusters. A common example is the average of all the pairwise distances between points in each cluster. Most global evaluation metrics are computed in O(N^2) time. Thus, in many cases, it takes longer to evaluate a single set of clusters than it takes to create them. 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 similarity. Most external methods use 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 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.

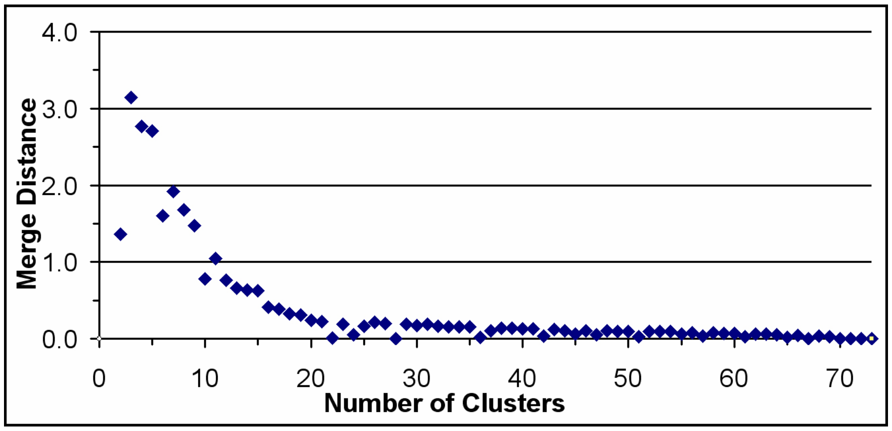


Figure 3. A sample evaluation graph.

An example of an evaluation graph produced by Gecko is shown in Figure 3. The *y*-axis values are the distances between the two clusters that are most similar at *x* clusters. The curve in Figure 3 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 3, starting from the right, where the merging process begins at the initial fine grain clustering, 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 merge distances indicate 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 internally homogeneous, and also dissimilar to each other.

Locating the exact location of the knee, and along with it the number of clusters, seems problematic when 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 is often created for time series data because a time series is a continuous function and a set of well-separated clusters usually does not exist in the time series. 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 is most likely a range of acceptable answers.

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) are often approximately linear. If a line is fitted to the right side and another line is fitted to the left side, then the intersection of 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 depicts an example.

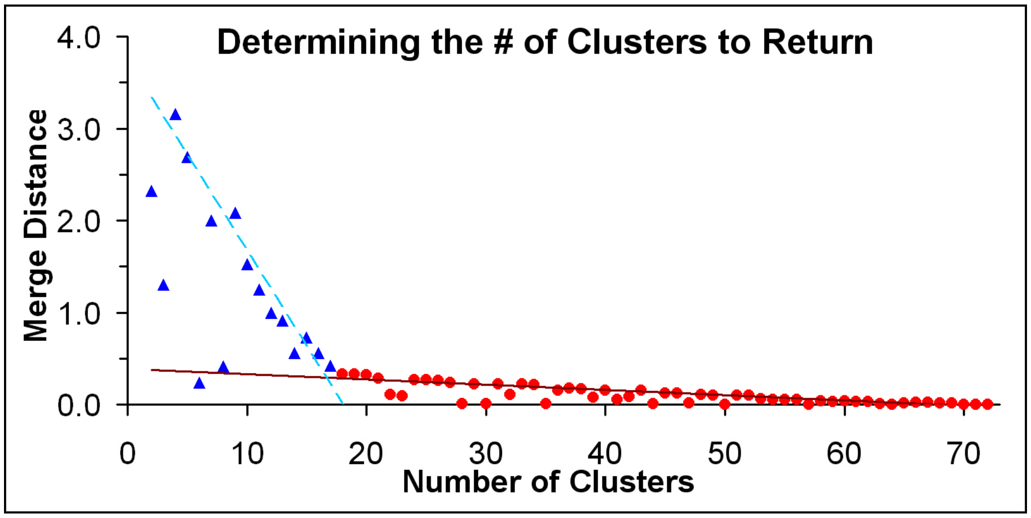


Figure 4. Finding the number of clusters using the L Method.

To create these two lines that intersect at the knee, we will find the pair of lines that most closely fit the curve. Figure 5 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 5, the third pair fits the data points with the smallest amount of error.

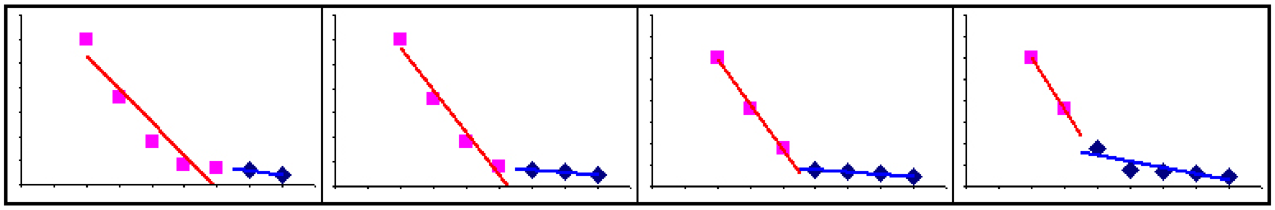


Figure 5. All four possible pairs of best-fit lines for a small evaluation graph.

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

$$c\wedge = \arg\min_c \; RMSE_c \qquad [2]$$

where location of the knee at *x*=*c*^ is used as the number of clusters to return. The L method can be implemented with a linear time complexity [26] and runs in less than 0.01 seconds for evaluation graphs containing fewer than 10,000 points.

The L method is general and has no parameters. 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.

Refinements for Segmentation Algorithms. Evaluation graphs for segmentation algorithms can often be very jumpy and contain a number of points that do not smoothly fit the curve. This is common for non-greedy algorithms that look several merges ahead and may make a seemingly poor merge to be able to make a very good merge at the next step. 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, 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.

3.2 RIPPER – Rule Generation

We have adapted RIPPER [1] to generate human readable rules that characterize the states identified by the Gecko algorithm. The RIPPER algorithm is based on the Incremental Reduced Error Pruning (IREP) [27] 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.

3.3 State Transition Logic

The upper right-hand quadrant of Figure 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 THEN remain in current state.
- IF input matches the next state THEN transition to the next state.
- IF input matches neither the current state nor the next state THEN transition to an anomaly 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 state.

Before an anomaly state is entered, one of two additional criteria must be satisfied: either (1) the number of consecutively observed anomalous values must exceed a specified threshold; or (2) the total number of anomalous values observed has exceeded another 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.

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 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.

4 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, and also to show that our overall system is able to detect anomalies. The data used to evaluate Gecko and the overall anomaly detection system is 10 time series data sets obtained from NASA. The data sets are signatures of a valve from the space shuttle.

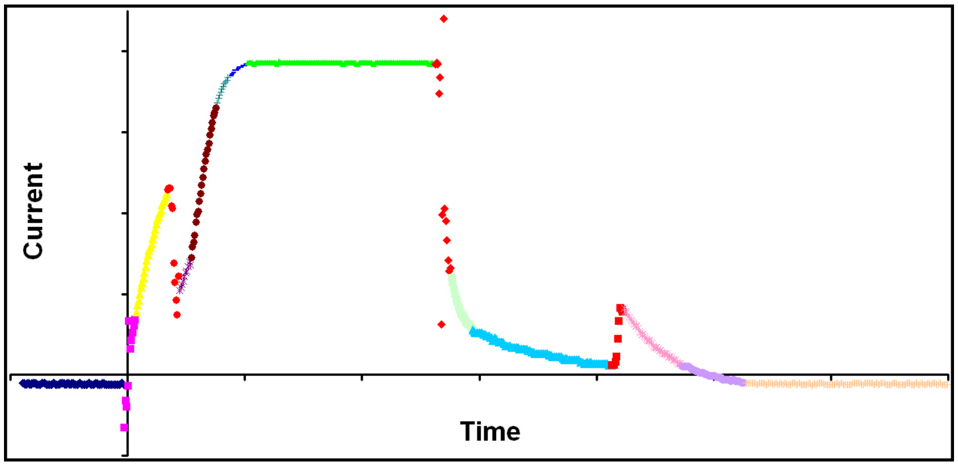


Figure 6. A data set after being clustered by Gecko (16 clusters).

Each data set contains between 1,000 and 20,000 equally spaced measurements of current. These 10 data sets contain signatures of valves that are operating normally, and also signatures of valves that have been damaged. The current method used to test these valves requires a human expert to compare a valve's signature to a known normal signature, and determine if there is any significant variation. We would like to demonstrate that Gecko is able to identify important phases/states in a time series, and that our anomaly detection system is able to determine if a valve is operating normally.

4.1 Determining the Number of Segments with the L Method

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 [15]. The permutation tests algorithm attempts to prevent segmentation algorithms from creating a PLR 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.

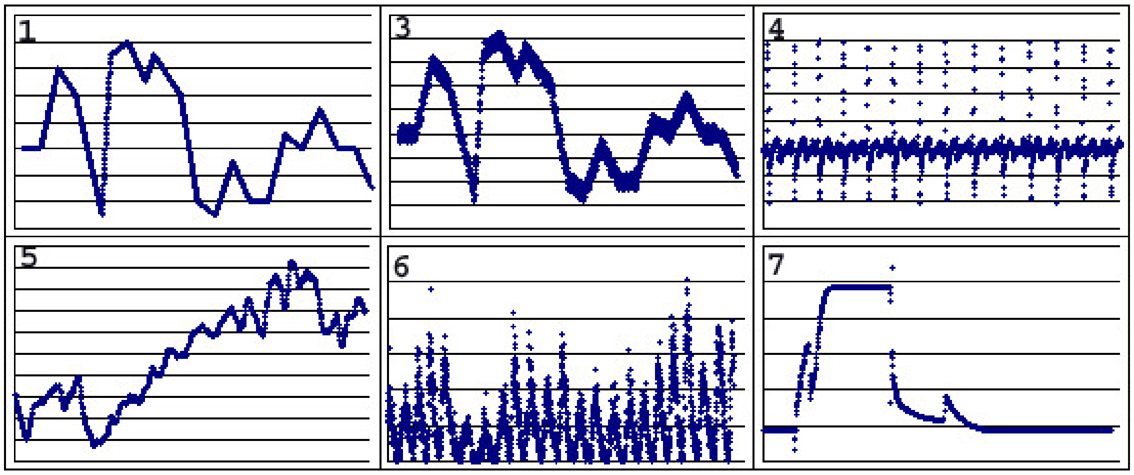


Figure 7. Data sets 1, 3, 4, 5, 6, and 7 for evaluating the L method in segmentation algorithms.

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 7) 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 7).
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 [28]. 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 [28]. 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 [28]. 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).

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. A PLA is considered “reasonable” if no adjacent segments are nearly identical to each other and all segments are internally homogeneous (segments have small error). The “reasonable range” for the number of segments for a data set and a segmentation algorithm is obtained by running the algorithm with various values of *k* (controls the number of segments returned), and determining the range of values that produce a 'reasonable' PLA. A single 'reasonable range' cannot be used for all of the segmentation algorithms because one value for *k* that produces a reasonable 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 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. To determine when to stop creating more segments, the parameter *p* sets the percentage of permutated time series that must have a relative reduction (between *k*-1 and *k* segments) in linear approximation quality larger than the original time series to return *k* segments [15].

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 1. 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 PLRs 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.

	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*	*Num 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

Table 1. Results of using the L method with three hierarchical segmentation algorithms.

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 1) 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, however 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 PLR 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 PLR 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 8. The lower left portion of Figure 8 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 merged 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 8, 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.

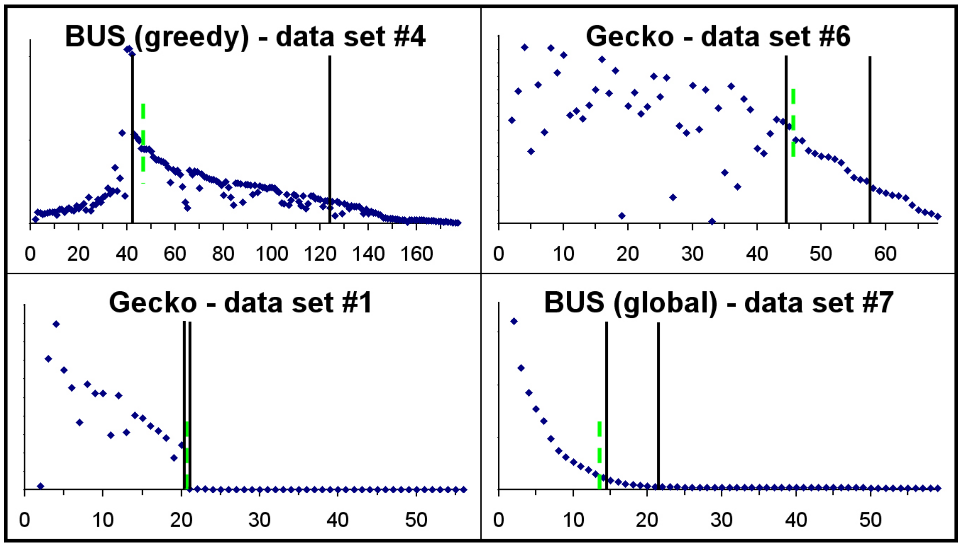


Figure 8. The reasonable range for the number of segments and the number returned by the L method. (axes: *x*=# of segments, *y*=evaluation metric – *short dashed line*=# of segments determined by the L method, *long solid lines*=the boundaries of the reasonable range for the # of segments.

In the evaluation graph at the upper-left of Figure 8(data set #4 BUS-greedy), the L method returned a number of segments that was at 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 3.1). 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 8 also has 'better' PLRs when the number of segments is near the low end of the reasonable range (fewer segments). 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 8). Greedy evaluation metrics increase more sharply at the knee, while global metrics have larger more ambiguous knees in their evaluation graph. A potential problem occurs when 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 less than 0.01 seconds to determine the number of segments in every test case, while the segmentation algorithms took 9-30 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.

4.2 Identifying States with Gecko

Procedures and Criteria. The quality of the segments produced by Gecko and an existing algorithm will be evaluating by having a domain expert blindly evaluate the output of each algorithm. A high quality set of segments has each segment corresponding to an important phase or state in the time series. The experimental procedure is as follows: Gecko and an existing algorithm, bottom-up segmentation (BUS), segment 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 segments from 1 to 10. The number of segments returned by BUS is set to be the same number that Gecko returns. Finally, the valve expert is asked to go over all 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 *clusterSize*=10.

Results and Analysis.

Table 2 contains the scores for Gecko and BUS given by the domain expert. Gecko's average score was 9.5, while the bottom-up segmentation algorithm's average score was only 4.3. Gecko often receives a perfect score (which signifies a set of segments as good as the human expert's) even though it returns more segments than what the human expert previously considered to be the 'ideal' number. For example, Gecko produced nearly twice as many segments as the human expert for data set 5 (13 vs. 7), and Gecko still got a perfect rating. This suggests that there is often a range of "very good" numbers of segments to return, rather than a single correct number.

Table 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 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 segments. BUS merges segments together by keeping the root-mean squared error of the best-fit lines to a minimum. This method measures error vertically, and 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 very large.

4.3 Overall System (FSA)

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 learn its signature, 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 its normal signature, and then take signatures of valves that are damaged slightly and run them through the anomaly detection system. The damaged valves should trigger anomalies.

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 3. An error point in Table 3 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 3 shows what the minimum consecutive error threshold (*CE*) must be set to for the anomaly detection system to not report an anomaly. A value of 1 in this last column means that the anomaly detection system will correctly not report an anomaly as long as $CE \geq 1$.

Table 3. 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" (*CT*) and the "consecutive error" (*CE*) thresholds were set to zero. This causes every possible state transition to be made and every error point to trigger 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 its signature and starts reporting many anomalies, the results for this data set can be improved by increasing *CT* to prevent it from transitioning too early on a single spurious data point.

Normal Operation Results. This test is to show that the anomaly detection system's model of the normal signature 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 (*CT*) parameter was set to 2, and the "consecutive error" threshold (*CE*) was set to 10 (minimum possible cluster size *clusterSize*=10). This means that two consecutive points believed to be in the next state are needed to perform a state transition and 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 *CE* was never encountered, an anomaly was never triggered.

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 signatures 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 signatures contained anomalies. Additionally, the system was able to inform the user of the state number where the signature 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 Concluding Remarks

We have detailed our approach to time series anomaly detection by discovering and characterizing the states of a time series, and performing transition logic between these states to construct a finite state automaton. This finite state automaton can be run on an expert system and used to track normal behavior and detect anomalies. The proposed Gecko segmentation algorithm is designed to cluster time series data (finds a small number of segments mapping to unique phases 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.)

Our empirical evaluations have shown that the L method used by the Gecko algorithm returns a number of segments that is similar to the number that is generated by a human expert. When the human 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. A perfect rating 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. The overall anomaly detection system was able to detect anomalies in every signature that was from a 'damaged' valve, and was also able to monitor a second normal valve without detecting any anomalies.

Future work will evaluate our approach with more datasets from NASA. Work is currently being done to learn a normal model from multiple data sets by using Dynamic Time Warping (DTW). Multiple time series will be warped together into a single time series which will them be clustered by Gecko. After the merged time series is clustered by Gecko, the cluster membership of the points in every normal time series can be determined and fed into the RIPPER algorithm to generate rules. We have also continued to study how the L method performs with other hierarchical clustering algorithms and different data sets [2]. To dynamically set the thresholds used in the state transition logic, we can investigate holding out part of the training data and find thresholds that prevent errors on the unseen portion of the data.

6 Acknowledgements

We thank Bobby Ferrell and Steven Santuro at NASA for providing the data sets, helpful comments, and clustering evaluations. We also thank Brian Buckley, Steve Creighton, and Walter Schiefele at ICS for help integrating our algorithms into their SCL expert system. Matt Mahoney also collaborated with us on this project and provided many helpful suggestions. This research is partially supported by NASA (NAS10-02044).

7 References

[1] W. Cohen, "Fast Effective Rule Induction," In Proc. of the 12 Intl. Conference on Machine Learning, Tahoe City, CA, 1995, pp. 115-123.

[2] S. Salvador and P. Chan, "Determining the Number of Clusters/Segments in Hierarchical Clustering/Segmentation Algorithms," Laboratory for Learning Research, Florida Institute of Technology, Melbourne, FL, Technical Report TR-2003-18, 2003.

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

[4] M. Ester, 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, Portland OR, 1996, pp. 226-231.

[5] A. Hinneburg 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.

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

[7] G. Karypis, E. Han, and V. Kumar, "Chameleon: A hierarchical clustering algorithm using dynamic modeling," IEEE Computer, vol. 32, no 8, pp. 68-75, 1999.

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

[9] E. Keogh, 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, 2001, pp. 289-296.

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

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

[12] M. Hansen and B. Yu, “Model Selection and the Principle of Minimum Description Length,” JASA vol. 96, pp.746-774, 2001.

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

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

[15] K. Vasko and T. Toivonen. “Estimating the number of segments in time series data using permutation tests,” In Proc. IEEE Intl. Conf. on Data Mining, Maebashi City, Japan, 2002, pp. 466-47.

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

[17] S. Monti, P. Tamayo, 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, issue 1-2, pp. 91-118, 2003.

[18] R. Tibshirani, G. Walther, and T. Hastie, “ Estimating the number of clusters in a dataset via the Gap statistic,” Dept. of Biostatistics, Stanford Univ., Stanford, CA, Technical Report 208, 2001.

[19] R. Tibshirani, G. Walther, B. Botstein, and P. Brown, “Cluster validation by prediction strength,” Dept. of Biostatistics, Stanford Univ., Stanford, CA, Technical Report 2001-21, 2001.

[20] T. Chiu, 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, San Francisco, CA, 2001, pp. 263-268.

[21] A. Foss, and A. Zaïane, A Parameterless Method for Efficiently Discovering Clusters of Arbitrary Shape in Large Datasets. In Proc. of the 2002 IEEE Intl. Conf. on Data Mining (ICDM'02), Maebashi City, Japan, 2002, pp. 179-186.

[22] S. Harris, D. Hess, and J. Venegas, "An Objective Analysis of the Pressure-Volume Curve in the Acute Respiratory Distress Syndrome," American Journal of Respiratory and Critical Care Medicine, vol. 161, issue 2, pp. 432-439. 2000.

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

[24] T. Caudell, 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, Portland, OR, pp. IV166-176. 1993.

[25] P. Langley, D. George, S. Bay, and K. Saito, "Robust Induction of Process Models from Time-Series Data," In Proc. of the 20th Intl. Conf. on Machine Learning, Washington, DC, 2003, pp. 32-439.

[26] Weisstein, E. "Least Squares Fitting". From MathWorld-A Wolfram Web Resource. [http://mathworld.wolfram.com/ LeastSquaresFitting.html].

[27] J. Furnkranz, and G. Wildmer, "Incremental reduced error pruning," In Proc. Intl. Conf. on Machine Learning, New Brunswick, NJ, 1994, pp. 70-77.

[28] E. Keogh 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, 2004.