Finding Patterns that Correspond to Episodes

Keywords: knowledge discovery, machine learning, cognitive robotics; Tracking number: 748

Abstract

We present two algorithms for elucidating structures in time series. These are unsupervised algorithms; they discover patterns without any knowledge about the episodic structures in the time series data. Yet, these patterns correspond with episodes, at least in an experiment with data from robot episodes. We offer a preliminary explanation for this result based on the idea that episodes persist. If this explanation is correct, then the algorithms are apt to be more generally applicable.

1 Introduction

Here is the problem we want to solve: We suspect a multivariate time series contains several *episodes*, but we don't know the episode boundaries, the number of episodes, or the structures of the episodes. We suspect that at least some of the episodes are similar, but perhaps no two episodes are identical. Finally, we suspect that episodes have a hierarchical structure in the sense that shorter episodes can be nested inside longer ones. The challenge is to find the episodes and elucidate their structure. We collected a dataset of 48 trials in which a robot approaches and pushes an object until it stalls. Each trial comprises several sub-episodes: a *quick-approach* sequence, a *slow-approach* sequence, a *contact* event, a *push* sequence, followed by a *stall-and-backup* sequence. Two *approach-and-push* sequences may be are similar in the sense that each comprises the same sub-episodes, and they even look roughly the same — the time series have roughly similar morphology — but they are not identical. If someone helpfully marks up a time series with episode and sub-episode boundaries, then it is relatively easy to characterize the structures of episodes and subepisodes (e.g., [2; 5]. The problem is more difficult if episode boundaries are not known. This is the problem we address here.

Note that *episode* is a *semantic* concept, one that makes implicit reference to the processes that generate time series, to what a time series represents. The corresponding *syntactic* concept is *pattern*. One can see several patterns in the sequence $ABCABCCBACBA$, including the triples $t_1 = ABC$ and $t_2 = CBA$; the sequences t_1t_1, t_2t_2 and $t_1t_1t_2t_2$; as well as the palindromic structure of the sequence. One doesn't need to know what the sequence represents — what it means, or denotes in the real world — in order to find these patterns. Suppose that the real world is, in fact, characterized by episodes of the form ABC and CBA. Then, an algorithm that finds these patterns will fortuitously have identified episodes. Unfortunately, there are usually many more patterns in data than episodes in the world, so one easily finds many patterns that do not correspond to episodes. The challenge, then, is to find syntactic patterns that correspond to semantic episodes. An episode boundary is a point or an interval at which a time series stops representing one thing and starts representing another; for example, the point at which a robot stops *approaching* an object and starts *pushing* it. The trick is to find a syntactic basis for dividing sequences such that the resulting subsequences are semantically coherent, that is, episodes.

Many AI problems can be characterized in this way. For example, computer vision researchers distinguish *image features* such as lines from *scene features* such as edges. Finding lines is a formal operation, interpreting them as edges is a semantic one, and, as all vision researchers know, the syntactic, formal, line structure only roughly indicates the semantic or scene structure. One is always looking for formal features that indicate semantic ones.

In general, the more one knows about the semantic structures, the more effectively one can find, design, or otherwise exploit syntactic indicators for them. For example, one might have expectations about the transition probabilities in episode, or a rough sense of how long episodes should be, or knowledge that particular sensor values tend to be more likely in some kinds of episodes than in others. All these can help elucidate patterns that correspond to episodes. Sometimes it suffices to know that subsequences of a time series belong to one class or another. In such cases, one can find syntactic features of the subsequences that predict class membership, and these are apt to have good semantic interpretations (see, for example, [1]).

Alternatively, one could design supervised learning algorithms to associate syntactic patterns with episodes. In this work, we do have a time series marked up (by hand) with episodes, so in principle we could try to learn the correspondences between patterns and episodes. However, many real-

world problems do not provide us with time series nicely marked with their episode structure. For instance, ethologists and psychologists must identify prototypical behaviors by observing continuous behavior; economists must identify periods of inflation and recession by observing economic data; and mobile robots that are learning to plan must figure out which action sequences "go together" in the sense of producing reliable, predictable results. Our methods are for problems like these. They are *knowledge-free* ways to find patterns that correspond to episodes.

2 The Data

The dataset is a time series of 22535 binary vectors of length 9, generated by a mobile robot as it executed 48 replications of a simple *approach-and-push* plan. In each trial, the robot visually located an object, oriented to it, approached it rapidly for a while, slowed down to make contact, attempted to push the object, and, after a variable period of time, stalled and backed up. The sample of trials comprised three blocks. In the first, 18 trials, the object was attached to the wall, so the robot was unable to push it and stalled shortly after trying. In 18 trials, the robot was able to push the object, but the object soon bumped into a wall, and once again the robot stalled. In the remaining 12 trials, the robot pushed the block unimpeded for a few feet. In one trial, the robot got hopelessly wedged in a corner of its playpen.

Data from the robot's sensors were sampled at 10Hz and passed through a simple perceptual system that returned values for nine binary variables: STOP, ROTATE-RIGHT, ROTATE-LEFT, MOVE-FORWARD, NEAR-OBSTACLE, PUSH, TOUCH MOVE-BACKWARD, STALLED. For example, the binary vector [0 1 0 1 1 0 1 0 0] describes a state in which the robot is rotating right while moving forward, near an object, touching it, but not pushing it. Of the 2^9 unique states that could occur, 35 did occur. Some states are very common (e.g., more than 5300 instances of [1 0 0 0 0 0 0 0 0], in which the robot is stopped and nothing else is happening). Fifteen of the 35 unique states account for more than 97% of the time series. Said differently, more than half of the unique states occur very rarely, and five of them occur fewer than five times.

Most of the 2^9 possible states are not semantically valid; for example, the robot cannot simultaneously be moving backward and moving forward. However, the robot's sensors are noisy and its perceptual system makes mistakes, and so some of the 35 observed states contained semantic anomalies; for example, there are 55 instances of states in which the robot is simultaneously stalled and moving backward.

Because the robot collected ten data vectors every second, and its actions and environment did not change that quickly, it is common to see long runs of identical states. The mean, median and standard deviation of run-length are 9.6, 4, and 15.58, respectively; while most runs are short, some are quite long.

We transformed the 22535 binary vectors in the dataset by hashing each vector to a number between 0 and 34 unique to the vector. This kind of transformation, in which a multidimensional state description is replaced by a unique identifier, loses information about similarities and differences between the states. We describe how to compensate this loss in Section 3.2.

3 Two Algorithms

This section describes two algorithms for finding patterns in the dataset we have just described. The first is based on dictionary-based compression algorithms, the second is a kind of "fuzzy" compression in which mismatches are possible.

3.1 Simple Compression

Data compression refers to the process of transforming a sequence in such a way as to make it shorter yet keep most or all of its constituent information [6]. For example, the sequence *A B C A B C C B A C B A* might be compressed as follows: Let $w \leftarrow ABC$ and $x \leftarrow CBA$, then the sequence is $wwxx$. If, further, we have $y \leftarrow xx$ and $z \leftarrow yy$, then the sequence is reduced to yz. This kind of compression, in which subsequences are replaced by symbols (or by locations in the original data sequence) is called *dictionary compression*, and many high-performance algorithms have been developed for it [6]. We are interested in compression algorithms not because we want to reduce the length of time series, but because we want to find repeating patterns in the series. That is, we are interested in the *rewrite rules*, such as $w \leftarrow ABC$, that map subsequences to symbols. The full expansion of a symbol corresponds to a subsequence. For instance, the full expansion of z, above, is first to yy and then, recursively, to $CBACBA$.

We implemented the following simple (and inefficient) algorithm:

Repeat until a stopping criterion is met:

1. Find the most common pair of symbols in the dataset (break ties randomly if two or more pairs of symbols are equally common)
2. Replace each instance of this pair with a new symbol

Given a sequence 1 2 1 2 1 2 1 2 3 4 3 4 3 4, the algorithm first substitutes $a \leftarrow 1, 2$, because this is the most common pair in the dataset, yielding $aaaa343434$. Next, the substitution $b \leftarrow 3, 4$ yields $aaaabbb$. Now a, a pairs are most frequent, so the substitution $c \leftarrow a, a$ gives $ccbbb$, and so on until a stopping criterion is met.

We ran the algorithm on the hashed dataset (22535 numbers, each representing one of 35 states). The results were unsatisfactory. One problem is that runs of a single state are compressed to different symbols. Consider the sequence 1 1 1 2 2 1 1 1 1 1 1 1 1 1. The first substitution is $a \leftarrow 1, 1$, yielding a 1 2 2 a a a a 1. The next is $b \leftarrow a, a$, yielding a 1 2 2 b b 1. At this point, two symbols expand to runs of 1: Symbol a expands to 1,1 and symbol b expands to 1,1,1,1. And because runs may contain an odd number of symbols, a new rule is needed to "pick up" the last symbol in a run. For example, later the algorithm may induce a rule $c \leftarrow b, 1$, the only purpose of which is to extend a run of four 1's to a run of five.

A slightly better version of the algorithm addresses these problems. It reduces runs of a symbol to a single symbol after each iteration of the algorithm. The improved algorithm still has some failings. First, it produces rewrite rules which expand to sequences that occur rarely in the data. Often, shorter rewrite rules, such as $A0 \leftarrow 12, 13$, expand to extremely common sequences, but the algorithm will construct rewrite rules for longer sequences that occur infrequently. This is partly because the algorithm is not inductive in any sense: It finds patterns that are identical to subsequences in the dataset, it does not find patterns that match nonidentical but similar subsequences.

Second, the algorithm has no information about the semantics of the data. It does not know whether some of the 35 states are similar to others, it treats every state as different, and it generates many, many rules that have similar meanings in the sense that the real world states they denote are not very different. For example, states 7 and 6 correspond to states 0 and 3 except that in the former cases an object is in sonar range and in the latter it isn't. In fact, for every state in the dataset in which an object isn't in range, there is an identical state in which one is. Perhaps is it appropriate to treat these sets of states differently, to construct different rules for them, and perhaps not, but the algorithm cannot make the determination for itself.

Third, when rules are expanded recursively to subsequences, one sees that many expansions do not correspond to episodes, nor even to subepisodes. The algorithm does not know where episode boundaries lie, so it cannot help merging subsequences that belong on either side of these boundaries. For example, the symbol A40 expands to the following five states of the world:

```
6:  ((NEAR-OBSTACLE R) (STOP R))
    7:  ((NEAR-OBSTACLE R) (MOVING-
FORWARD R))
    6:  ((NEAR-OBSTACLE R) (STOP R))
    12: ((NEAR-OBSTACLE R) (PUSHING R)
     (MOVING-FORWARD R) (TOUCHING R))
    13: ((NEAR-OBSTACLE R) (STOP R) (TOUCH-
ING R))
```

In the first three states, the robot is approaching an object that is within sonar range; in the latter two steps it makes contact with and begins pushing the object. We might argue that a boundary should be drawn between the third and fourth state, corresponding to a semantic difference in the robot's activities, but of course the algorithm lacks this semantic information and it simply merges all the steps into one subsequence.

We designed a second algorithm to address these problems.

3.2 Tree building

The central idea of this algorithm is to build patterns that need not match exactly to subsequences of data. For example, the data $abcc$ matches the pattern $abbc$ to the degree that the tokens c and b are similar.

Recall that each state in the robot dataset is a binary vector of nine state variables. The similarity of two states may then be defined as the hamming distance between the states.

The algorithm builds a tree, the root of which corresponds to the beginning of a pattern, and in which each path to a leaf node contains the tokens in the pattern. For example, the tree contains the pattern $abbc$ iff a child of the root node is labelled a, a child of this node is labelled b, and it has a child labelled b; and a leaf-node child of this node is labelled c. *Accepting* a subsequence $ij\ldots$ of data means matching it to a pattern in the tree, but the match need not be perfect. For instance, suppose the mismatch between symbols b and c is 3.0. Then the sequence $abcc$ matches the path $abbc$ with a total cost of 3.0 and an average cost of $3/4$. The average mismatch cost (*AMC*) is just the sum of the token-by-token mismatch costs divided by the length of the sequence. The algorithm requires a threshold value k for *AMC*.

Suppose the tree contains only the paths ab, abb, $abbc$; and the time series is $abcc\ldots$. If $k \geq 3/4$, then the pattern $abbc$ will accept the first four tokens in the time series, but what if $k < 3/4$? The pattern ab will accept the first two tokens of the series, ab, with $AMC = 0$; the pattern abb will not accept the first three tokens of the series, abc because $AMC = 3/3$; and the pattern $abbc$ will not accept the first four tokens of the series because $AMC = 3/4$. So the longest pattern that will accept any of the series is ab.

To build a tree of patterns, one repeatedly finds the longest pattern that will accept any of a series, and extends the pattern by a single token, namely, the next token in the series. To continue with the previous example, the pattern ab accepts the tokens ab in the series $abcc$ and is extended by the next token in the series, yielding the pattern abc.

This is how the tree is built:

1. Build a tree T of depth 1 containing one leaf node for each unique state in the dataset. Set a pointer i to the first state in the time series S.
2. Find the longest pattern(s) in the tree that will accept a subsequence s of tokens in S, starting at i, i.e., the longest such pattern with $AMC \leq k$. If two or more patterns have the same length, select the one that has the most tokens position-by-position identical with the tokens in s. If this doesn't narrow the field to one, choose a pattern at random. Let n_s denote the length of the subsequence s (and pattern that accepts it).
3. Extend the pattern by appending to it the token at $i + n_s$ in the time series. Set i to be $i + n_s + 1$, and go to the previous step.

3.3 Selecting patterns that correspond to episodes

Both of the algorithms construct *lots* of patterns, so we might want to filter them. Ideally, patterns should elucidate the unknown structure of episodes in data. We assume a time series comprises a sequence of episodes and we want to select a set of patterns that corresponds to the episodes. Two precise measures of correspondence are given in Section **??**; here we rely on the intuition that a set of patterns corresponds to a set of episodes if the patterns are systematically associated with the episodes in time. Suppose two episodes in the life of a bacterium are "hungry" and "sated," and two observable patterns of behavior are "exploration" and "drifting."

The patterns correspond to the episodes if the bacterium explores when it is hungry and drifts when it is sated, and rarely explores when sated or drifts when hungry. Clearly, given the episode structure of a time series, one can select a set of patterns that corresponds well to the episodes, but this is not the problem we want to solve. We want to select a set of patterns that corresponds well to an *unknown* episode structure, a set that *elucidates* the unknown episode structure.

One option is to introduce more or less strong domain knowledge about episodes into the selection phase; for example, in the experiment described earlier, MOVING-BACKWARD happens at the ends of trials, so we could select patterns in which MOVING-BACKWARD occurs near the end. Lacking this sort of knowledge, though, is there a domain-independent way to select patterns that are likely to correspond to episodes or substructures within episodes? Said differently, how much must we specify about episodes to design a method for selecting corresponding patterns? One assumption goes a long way: *Episodes persist*. If a datum in a time series is from episode e_i then the probability is relatively high that the next datum will also be from episode e_i; more formally,

$$Pr(x_{t+1} \leftarrow e_i | x_t \leftarrow e_i) >> Pr(x_{t+1} \leftarrow e_j | x_t \leftarrow e_i) \quad (1)$$

An immediate implication is that common digrams — consecutive pairs of data that occur frequently in the dataset — are probably from the same episode. (The opposite is not true: Uncommon digrams are not necessarily from different episodes; episodes can contain uncommon state transitions.) Because episodes persist, successive data in a time series are apt to be sampled from the same probability distribution, that is, x_{t+1} is apt to be generated by the same process that generated x_t. This assumption underlies many methods for finding episode boundaries.

This persistence assumption is the basis for selecting patterns generated by the simple compression and tree-building algorithms. Specifically:

- For simple compression, select any rewrite rule that occurs with a frequency greater than some threshold.
- Method 1 for tree building: Select any path through the tree that occurs with a frequency greater than some threshold.
- Method 2 for tree building: Define the conditional entropy of a node in the tree as follows: Each nonleaf node N has k children, C_i, and the digram N, C_i is observed in the data with frequency f_i. Then,

$$Pr(C_i|N) = \frac{f_i}{\sum_{i=1}^{k} f_i} \quad (2)$$

and the conditional entropy is

$$H(C_i|N) = -\sum_{i=1}^{k} Pr(C_i|N) log Pr(C_i|N). \quad (3)$$

Select any path from the root to node N for which the conditional entropy exceeds a threshold.

4 Evaluation

In this section we assess how well the compression and tree-building algorithms, and the selection criteria discussed in the preceding section, find patterns that correspond to episodes and subepisodes in our robot data set. Each of 48 episodes contained some or all of the following sub-episodes:

A:	start a new episode with orientation and finding the target
B1:	forward movement
B2:	forward movement with turning or intruding periods of turning
C1:	B1 + an object is detected by sonars
C2:	B2 + an object is detected by sonars
D:	robot is in contact with object (touching, pushing)
E:	robot stalls, moves backwards or otherwise ends D

By hand, we associated one of these seven sub-episode type labels with each of the 22535 data items in the robot time series, producing an *episode-labelled series* of the same length[1]

4.1 Method

The compression and tree-building algorithms were run on the first 20000 time steps of the hashed robot dataset — a univariate time series in which each binary vector of nine propositions is replaced by a number unique to the vector. Subsets of patterns were selected as follows: First, the conditional entropy threshold was set in such a way as to produce approximately 15, 30, 60 or 90 patterns, then the frequency threshold (method 1 for the tree-building algorithm) was set in such a way as to produce comparable numbers of patterns. It is unclear how best to select rules from the compression algorithm. The algorithm was allowed to make 500 iterations, producing 500 rules of lengths between 2 and 136. Lacking a better method, we simply selected 15, 30, 60 or 90 rules at random from this set for the purpose of comparing compression rules with tree-building rules.

Next, the hashed dataset was *tiled* by each set of patterns. The tiling algorithm is greedy and by no means optimal, but it serves our purposes here. It starts with a set of patterns ordered by length, so longer patterns are laid down as tiles before shorter ones. It greedily places instances of each pattern wherever they will go in the robot dataset, marking the covered intervals of data as occupied. Recall that patterns from the tree-building algorithm are *partial* matches to the original data, so partial matching is allowed during the tiling phase, too: The tiling algorithm greedily places instances of patterns wherever they match a subsequence of data well enough.

Let each pattern have a unique label. Each data item in a tiled series can be replaced by the label of the pattern that covers it or by NIL if no pattern covers it. Call the resulting series the *tile-labelled series*, analogous to the episode-labelled series described in the previous section. Let N be

[1]This cannot be done algorithmically, as some contextual interpretation of subsequences of the series is required. For example, if the sonars temporarily lose touch with an object, only to reaquire it a few seconds later, we label the intervening data C1 or C2, not B1 or B2, even though the data satisfy the criteria for B1 or B2.

the length of these series, and N_{NIL} the number of NILs in the tile-labelled series. Five statistics may now be defined:

Coverage The proportion of the dataset that is tiled by the patterns, i.e., $\frac{N-N_{NIL}}{N}$.

Lambda This statistic is a measure of association between the tile-labelled series and the episode-labelled series. It is the reduction in errors in predicting the episode label of a datum as a consequence of knowing the pattern label of the datum. Space precludes a formal description. Suffice it to say that the λ statistic is zero when pattern labels are not associated with sub-episode labels, and one when pattern labels predict sub-episode labels perfectly.

Purity Align the tile-labelled and episode-labelled series. Each instance of a tile corresponds to several consecutive locations in these series. Suppose an instance of a particular pattern (tile) corresponds to the following locations in the episode-labelled series: (A A A A A A B1 B2). The pattern seems to correspond well to sub-episodes of type A. However, later, the same pattern corresponds to these locations in the episode-labelled series: (B1 B1 B1 B1 B1 B2 C C). Summing up these instances of the pattern, it covers six A's, five B1's, two B2's, and two C's, so it's unclear which sub-episode the pattern corresponds to. A rough measure of this uncertainty is to find the most frequent episode label for all instances of a pattern and divide it by the total number of episode labels associated with a pattern. In the current example, this is $6/15 = .4$. The higher this purity measure, the better.

Rule length Each pattern in a selected set of patterns has a length, we report the mean and median of the lengths.

Used rule length Some patterns are used more frequently than others to tile a series. We report the mean and median length of the patterns that are actually used.

4.2 Comparison of the algorithms

Comparisons of the compression and tree building algorithms under different conditions are shown in Table 4.2. The first block of results in the table is for trials in which the parameters of the entropy-based selection method (Sec 3.3) were fixed to produce roughly 90, 60, 30 or 15 patterns; the parameters of the frequency-based method were fixed to produce comparable numbers of patterns; and comparable numbers of patterns were selected at random from the rules produced by the compression algorithm. This method tended to produce relatively short rules for the frequency-based condition, so a second block of trials was run in which the frequency-based and entropy-based methods returned the most frequent patterns with a minimum length of 10.

The general result is that the entropy-based selection method performs slightly better than either the frequency-based method or the compression method. When each method selects 90 rules, a tiling with these rules cover roughly the same proportion of the original time series (79% to 84% in the first block of trials, 70% and 71% in the second). As the number of rules decreases, the coverage of compression-based rules declines. This is because the tiling algorithm requires a perfect match to use a compression-based rule (unlike rules from the tree-building algorithm, which allow partial matches) and as the number of available rules decreases, less of the series can be perfectly matched. The coverage of entropy-based rules is better than that of frequency-based rules and this disparity increases as the number of rules decreases. This suggests that the former rules are more representative of the data than the latter, an impression borne out by qualitative comparisons of the rules (see below). In all conditions, irrespective of the number of rules, compression and entropy-based selection produce rules with higher predictive accuracy (λ) than the frequency-based method. In the case of compression-based rules, this is because the rule instances used in tiling must match the series perfectly, and so are apt to have higher predictive power. In the case of entropy-based rules, the reason is probably that entropy-based selection produces rules that better match the boundaries of episodes. Certainly, the purity statistic suggests this, although the disparity between frequency-based and entropy-based rules is relatively small in the second block of trials, where all rules had to be at least 10 tokens long. In general, entropy-based rules are longer than frequency-based ones, which means they span more of an episode. It is unclear whether any of thee differences is statistically significant; the results of randomization tests will be provided in the final paper.

A qualitative comparison of the rules returned by each method also favors entropy-based selection. In all conditions, the entropy-based rules were more representative of the episodic structure of the robot data; more to the point, the rules that were actually *used* by the tiling algorithm were more representative. For example, in the 60-rules conditions (in both batches of trials), the 20 most-frequently-used entropy-based rules represented all seven sub-episode types, whereas the comparable set for the frequency-based rules contained nothing to represent sub-episodes of type E, where the robot stalls and backs away from an object. In contrast, the entropy-based method came up with two such rules, one summarized as "pushing,pushing,...,stall" and the other as, "stall,stall,...,stop, stop,...,move-backward-and-rotate-left".

5 Discussion

We have presented two algorithms for finding patterns in time series, and shown that the patterns they find correspond with episodes. The algorithms we describe are what statisticians call discovery procedures; unsupervised methods such as cluster analysis, HMM induction [7], or causal induction [4] that "carve nature at its joints" ([3]). The principal empirical criterion for such algorithms is whether or not they produce structures that analysts can use to elucidate the processes underlying data. We have evaluated our algorithms differently, by testing whether the patterns they produce correspond with *known* episodes in a hand-marked data set. Not surprisingly, performance is not perfect. It is difficult to simultaneously build models of episodes (i.e., patterns) and find episode boundaries, given no knowledge of the domain. When episode boundaries are provided, the data between them can be clustered to produce prototypes of differ-

Method	Rules	Cvrg.	λ	Prty.	Length	Used
Compression	90	.79	.78	.8	19.5,25.5	26.8,56
Tree Freq.	91	.82	.70	.79	7.6,7	7.8,4
Tree Entr.	93	.84	.76	.83	18.9,22	14.1,9
Compression	60	.64	.72	.81	17.1,7	17.0,9
Tree Freq.	64	.80	.70	.79	8,7	7.8,6
Tree Entr.	67	.80	.76	.83	18.5,22	14.7,12
Compression	30	.50	.80	.86	16.6,20	13.6,7
Tree Freq.	31	.57	.63	.74	8.4,8	6,3
Tree Entr.	30	.76	.74	.82	17.5,20.5	12.1,6
Compression	15	.34	.82	.88	15.5,12	16.3,18
Tree Freq.	15	.56	.63	.74	4.5,3	4.8,3
Tree Entr.	15	.57	.79	.84	19.9,22	17.2,20
Tree Freq.	90	.71	.71	.80	13.9,13.5	17,18
Tree Entr.	90	.70	.76	.83	22.8,24.5	23.8,26
Tree Freq.	57	.61	.75	.81	13.1,13	16.6,16
Tree Entr.	58	.68	.78	.84	22.9,25	23.9,26
Tree Freq.	31	.59	.74	.80	13.5,13	15.2,15
Tree Entr.	31	.65	.76	.83	23.1,24	23.4,24
Tree Freq.	16	.34	.58	.75	13.6,13.5	16,16
Tree Entr.	15	.57	.77	.84	23.5,26	22.7,24

Table 1: The coverage and purity of rules generated by simple compression, tree building, and tree building with entropy-based rule selection.

ent episode classes [5; 2; 1]. Our algorithms are not provided this information. The reason they work seems to be a very general principal we call *persistence*: Adjacent data points are more likely to be generated by one process than by different processes. Said differently, boundaries between episodes are much rarer than non-boundaries. This being the case, a method like entropy-based rule selection, which searches for rules that terminate at points of high entropy, perform slightly better that frequency-based rule selection. Of course, this result holds only for the robot data we tested. While persistence is a theoretical property, not an empirical one, it remains to be seen whether algorithms which capitalize on persistence are generally, empirically superior to those which do not.

References

[1] Tim Oates, Zachary Eyler-Walker and Paul Cohen, Toward Natural Langauge Interfaces for Robotic Agents: Grounding Linguistic Meaning in Sensors, Proceedings Fourth International Conference, 227-228, 2000.

[2] Matthew Schmill, Tim Oates and Paul Cohen, Learning Planning Operators in Real-World, Partially Observable Environments, Proceedings Fifth International Conference on Artificial Planning and Scheduling, 246-253, 2000.

[3] David j. Hand. Data mining - reaching beyond statistics. *Research in Official Statistics*, 2:5–17, 1998.

[4] J. Pearl. *Causality: Models,Reasoning and Inference*. Cambridge University Press, 2000.

[5] Marco Ramoni, Paola Sebastiani and Paul Cohen, Multivariate Clustering by Dynamics, *Proceedings of the Seventeenth National Conference on AI*, 633-638, 2000.

[6] David Saloman. *Data Compression*. Springer, 1998.

[7] L. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, 1989.