Online Sequential Prediction via Incremental Parsing: The Active LeZi Algorithm

Karthik Gopalratnam, *University of Texas at Arlington*

Diane J. Cook, *Washington State University*

Intelligent systems that can predict future events can make more reliable decisions. Active LeZi, a sequential prediction algorithm, can reason about the future in stochastic domains without domain-specific knowledge.

As intelligent systems become more widespread, they must be able to predict and then adapt to future events. An especially common problem is sequential prediction, or using an observed sequence of events to predict the next event to occur. In a smart environment, for example, predicting inhabitant activities provides a basis for automating interactions with the environment and improving the inhabitant's comfort.

In this article, we investigate the potential of constructing a prediction algorithm based on data compression techniques. Our Active LeZi prediction algorithm approaches sequential prediction from an information-theoretic standpoint. For any sequence of events that can be modeled as a stochastic process, ALZ uses Markov models to optimally predict the next symbol.

Consider a sequence of events being generated by an arbitrary deterministic source, represented by the stochastic process $X = \{x_i\}$. We can then state the sequential prediction problem as follows. Given a sequence of symbols $\{x_1, x_2, \ldots x_i\}$, what is the next symbol x_{i+1}? Well-investigated text compression methods have established that good compression algorithms are also good predictors.[1] According to information theory, a predictor model with an order that grows at a rate approximating the source's entropy rate is an optimal predictor. Text compression algorithms' incremental parsing feature is desirable for online predictors and is a basis for ALZ.

Markov predictors and LZ78 compression algorithms

Meir Feder, Neri Merhav, and Michael Gutman first considered the problem of constructing a universal predictor for sequential prediction of arbitrary deterministic sequences.[2] They proved the existence of universal predictors that could optimally predict the next item in any deterministic sequence. They also proved that Markov predictors based on the LZ78 family of compression algorithms attain optimal predictability.[2] Feder, Merhav, and Gutman have applied these concepts to branch prediction in computer programs and page prefetching into memory, as well as mobility tracking in PCS networks.

Consider a stochastic sequence $x_1^n = x_1, x_2, \ldots, x_n$. At time t, the predictor will have to predict what the next symbol x_t is going to be on the basis of the observed history (that is, the sequence of input sym-

```
loop
  wait for next symbol v
  if ((w.v) in dictionary):
    w = w.v
  else
    add (w.v) to dictionary
    w = null
    increment frequency for every
        possible prefix of phrase
forever
```

(a)

```
initialize Max_LZ_length = 0
loop
  wait for next symbol v
  if ((w.v) in dictionary):
    w := w.v
  else
    add (w.v) to dictionary
    update Max_LZ_length if necessary
    w := null

  add v to window
  if (length(window) > Max_LZ_length)
    delete window[0]

  Update frequencies of all possible
    contexts within window that includes v
forever
```

(b)

Figure 1. Pseudocode for parsing and processing the input sequence in (a) LZ78 and (b) Active LeZi.

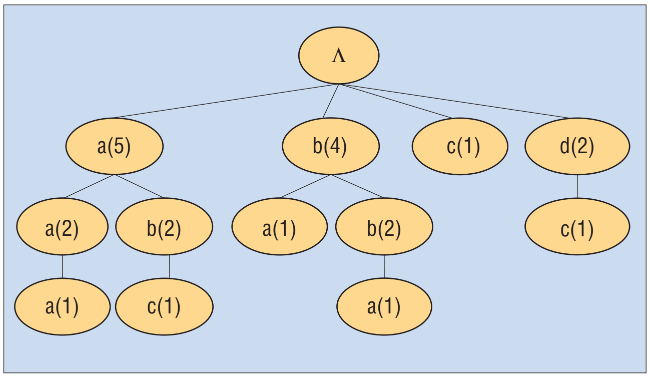


Figure 2. The trie formed by the LZ78 parsing of the sequence *aaababbbbbaabccddcbaaaa*.

bols $x_1^{t-1} = x_1, x_2, \ldots, x_{t-1}$) while minimizing the prediction errors over the course of an entire sequence. In a practical situation, an optimal predictor must belong to the set of all possible finite state machines (FSMs). Feder, Merhav, and Gutman showed that universal FS predictors, independent of the particular sequence being considered, achieve the best possible sequential prediction that any FSM can make (known as *FS predictability*).[2]

An important additional result is that Markov predictors, a subclass of FS predictors, perform as well as any FSM. Markov predictors maintain a set of relative frequency counts for the symbols seen at different contexts in the sequence, thereby extracting the sequence's inherent pattern. Markov predictors then use these counts to generate a posterior probability distribution for predicting the next symbol. Furthermore, a Markov predictor whose order grows with the number of symbols in the input sequence attains optimal predictability faster than a predictor with a fixed Markov order. So, the order of the model must grow at a rate that lets the predictor satisfy two conflicting conditions. It must grow rapidly enough to reach a high order of Markov predictability and slowly enough to gather sufficient information about the relative frequency counts at each order of the model to reflect the model's true nature.

Jacob Ziv and Abraham Lempel's LZ78 data compression algorithm is an incremental parsing algorithm that introduces such a method for gradually changing the Markov order at the appropriate rate.[3] This algorithm has been interpreted as a universal modeling scheme that sequentially calculates empirical probabilities in each context of the data, with the added advantage that the generated probabilities reflect contexts seen from the beginning of the parsed sequence to the current symbol.

LZ78 is a dictionary-based text compression algorithm that incrementally parses an input sequence. This algorithm parses an input string $x_1, x_2, \ldots x_i$ into $c(i)$ substrings $w_1, w_2, \ldots, w_{c(i)}$ such that for all $j > 0$, the prefix of the substring w_j (that is, all but the last character of w_j) is equal to some w_i for $1 < i < j$. Because of this prefix property, parsed substrings (also called *LZ phrases*) and their relative frequency counts can be maintained efficiently in a multiway tree structure called a *trie*. Because LZ78 is a compression algorithm, it has two parts: an encoder and a decoder. For a prediction task, however, we don't need to reconstruct the parsed sequence, so we don't need to consider this as an encoder and decoder system. Instead, we simply must construct a system that breaks up a given sequence (string) of states into phrases (see figure 1a).

Consider the sequence x^n = *aaababbbbbaabccddcbaaaa*. An LZ78 parsing of this string would create a trie (see figure 2) and yield the phrases *a, aa, b, ab, bb, bba, abc, c, d, dc, ba*, and *aaa*. As we described earlier, this algorithm maintains statistics for all contexts within the phrases w_i. For example, the context *a* occurs five times (at the beginning of the phrases *a, aa, ab, abc,* and *aaa*), and the context *bb* occurs two times (in the phrases *bb* and *bba*).

As LZ78 parses the sequence, larger and larger phrases accumulate in the dictionary. As a result, the algorithm gathers the predictability of higher- and higher-order Markov models, eventually attaining the universal model's predictability.

Active LeZi

LZ78's drawback is its slow convergence rate to optimal predictability, which means that the algorithm must process numerous input symbols to reliably perform sequential prediction. LZ78 doesn't exploit all information that can be gathered. For example, the algorithm doesn't know contexts that cross phrase boundaries. In our example string, the fourth symbol (*b*) and fifth and sixth symbols (*ab*) form separate phrases; had they not been split, LZ78 would have found the phrase *bab*, creating a larger context for prediction. Unfortunately, the algorithm processes one phrase at a time and doesn't look back.

Amiya Bhattacharya and Sajal Das partially addressed the slow convergence rate in the LeZi Update algorithm by keeping track of all possible contexts within a given phrase.[4] Similarly, Peter

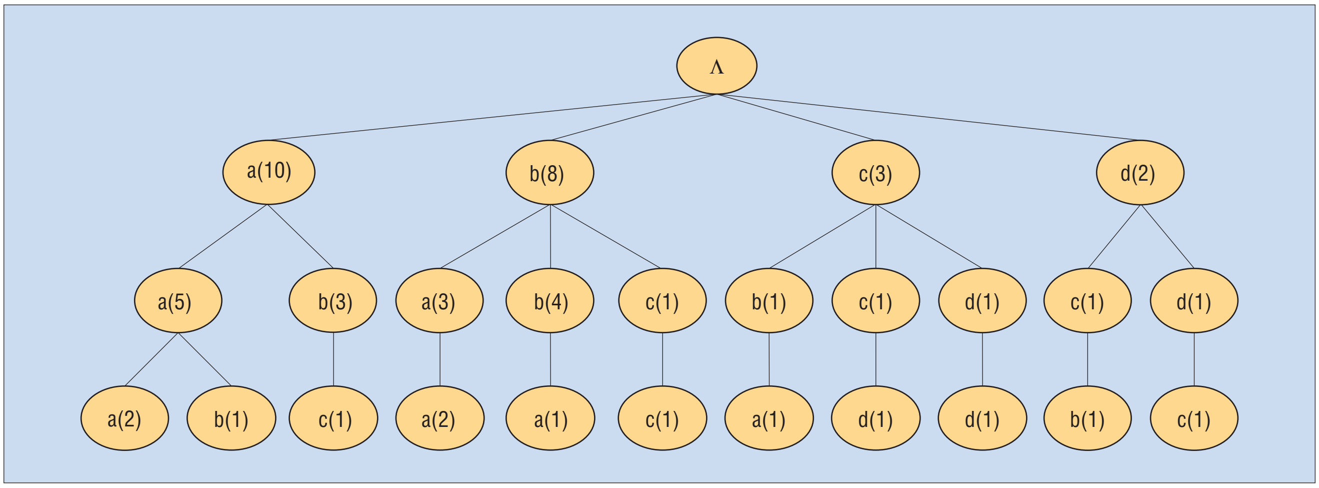


Figure 3. The trie formed by the Active LeZi parsing of the sequence *aaababbbbbaabccddcbaaaa*.

Franaszek, Joy Thomas, and Pantelis Tsoucas addressed context issues by storing multiple dictionaries for differing-size contexts preceding the new phrase.[5] They selected the dictionary for the new phrase that maximizes expected compression. However, neither approach attempts to recapture information lost across phrase boundaries.

Active LeZi is an enhancement of LZ78 and LeZi Update that addresses slow convergence using a sliding window. As the number of states in an input sequence grows, the amount of information being lost across the phrase boundaries increases rapidly. Our solution maintains a variable-length window of previously seen symbols. We make the length of the window at each stage equal to the length of an LZ78 parsing's longest phrase. ALZ can determine this length on the fly as we feed each new input symbol to the predictor without any extra computational overhead. We selected this window size because LZ78 essentially constructs an (approximation to an) order-*k-1* Markov model where *k* is equal to the length of the longest LZ78 phrase seen so far. For example, the trie in figure 2 has a depth of three corresponding to the length of the longest phrase, indicating a Markov order of two.

Within this window, we can now gather statistics on all possible contexts (see figure 1b). By capturing information about contexts that cross LZ78 phrase boundaries, we build a better approximation to the order-*k* Markov model. Therefore, we converge faster to optimal predictability. Figure 3 shows the trie formed by the ALZ parsing of the input sequence *aaababbbbbaabccddcbaaaa*. This is a more complete order-*Max_LZ_length-1* Markov model than the one in figure 2 because it incorporates more information about the contexts that have been seen.

Performance

ALZ is a growing-order Markov model that attains optimal predictability faster than LZ78 because it uses information that's inaccessible to LZ78. Here we look more closely at its performance.

Lemma. The number of nodes in an ALZ trie is $O(n^{3/2})$.

The proof follows from the observation that ALZ adds nodes to a trie only when generating frequency counts. While generating frequency counts of the current window's contexts, ALZ either updates the counts in an existing node or adds a new node to the trie. The worst possible sequence both rapidly increases the maximum phrase length (and consequently the ALZ window size) and yet grows slowly enough that most subphrases in the ALZ window add new nodes to the trie. Given that the maximum LZ phrase length increases by at most one for each new phrase encountered, the worst sequence for ALZ is one in which each new LZ phrase is one symbol longer than the previous LZ phrase. We represent this sequence as $\hat{S} = x_1\, x_1x_2\; x_1x_2x_3 \ldots x_1x_2x_3 \ldots x_k$, where the length of the sequence $|\hat{S}| = n = k(k + 1)/2$.

ALZ models a sequence of this form as an order-*k* Markov model, and this model stays of order-*k* through the next *k* symbols. In the worst case, each subphrase in the ALZ window that includes the subsequent symbol adds a new node to the trie, so at order *k*, the trie gains k^2 nodes before the model transitions to order $k + 1$. Therefore, the number of nodes generated in the trie by the time the model attains order *k* is $O(k^3) = O(n^{3/2})$, because

$$k = O\left(\sqrt{n}\right)$$

In practice, the order grows much more slowly, because the worst sequence is interspersed with intervals of shorter LZ phrases. When the algorithm parses these portions of the sequence, it's merely "filling in" the trie. In other words, for a given fixed alphabet size α, when the Markov model is order *k*, the algorithm is completing the α-way tree of depth *k*, which in the limiting case brings the space requirement of the algorithm to order $O(n)$.

Theorem 1. The time complexity of ALZ is $O(n^{3/2})$.

The proof: The worst sequence $\hat{S}$ in terms of space utilization is also the worst sequence in terms of runtime because it will prompt the most updates (such as the creation of a new node in the trie) for each new symbol processed. Creating or updating a node in the trie requires finding the appropriate child of a given node along the path that phrase traced in the trie. Given a fixed alphabet size, ALZ can access a given node's child in constant time. Therefore, it can find a node in time linear in the depth of the trie. Consider the worst-case sequence $\hat{S}$ where $|\hat{S}| = n$, with order

$$k_n = O\left(\sqrt{n}\right)$$

When the ALZ model is at order $k_i < k_n$, there must be an update for every order less than k_i. Over the next k_i symbols—that is, the number of symbols it takes to transition to the next higher order—the updates take time $O(k_i^2)$. So, attaining order-k_n will take $O(k_n^3)$ time, and the algorithm's runtime is also $O(n_{3/2})$. In practice, the runtime is almost linear in the input size.

Theorem 2. ALZ attains FS predictability.

The proof: Feder, Merhav, and Gutman[2] showed that a Markov predictor with S states converges to FS predictability at the rate of

$$O\left(\sqrt{S/n}\right)$$

Because the ALZ predictor is a Markov predictor with $O(n_{3/2})$ states, we can conclude that ALZ attains FS predictability at the rate of

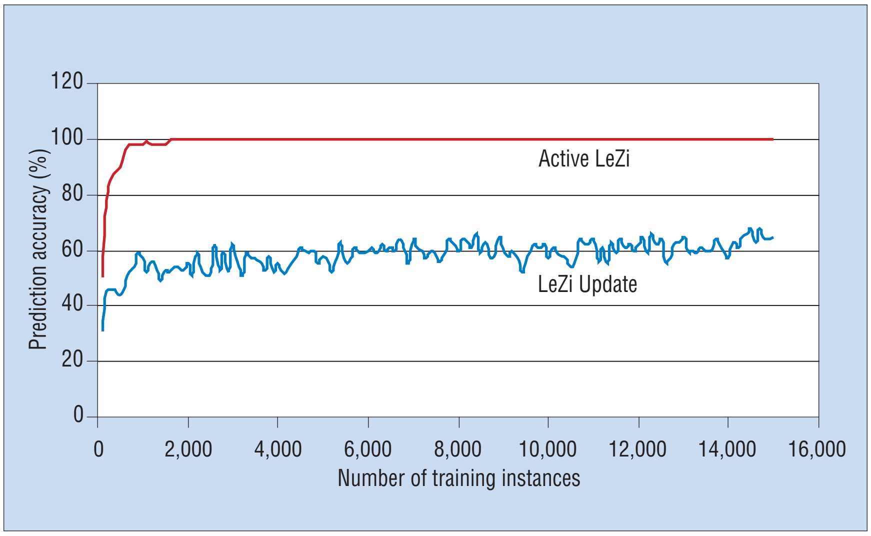


Figure 4. Comparison between Active LeZi and LeZi Update.

$$O\left(\sqrt{n}\right)$$

This is significantly better than the original Lempel-Ziv algorithm's predictor, which attains FS predictability at the rate of

$$O\left(\sqrt{1/\log n}\right)^2$$

This implies that in practical prediction scenarios, ALZ should converge to a pattern much faster than LZ78 or LeZi Update.[4] Figure 4 compares the learning curves for ALZ and LeZi Update on synthetic data. A pattern consisting of 200 random symbols drawn from an alphabet of 100 symbols was repeated 100 times to create the inherent "true model." We then embedded this with noise by randomly replacing 20 percent of the symbols with other symbols. We trained the algorithm on the noisy data by building the corresponding trie, then tested the algorithm against the pure data. ALZ converges much more quickly than LeZi Update to the embedded pattern.

Probability assignment

To predict the next event of the sequence for which ALZ has built a model, we calculate the probability of each state occurring in the sequence and predict the one with the highest probability as the most likely next action.

To achieve better convergence rates to optimal predictability, the predictor must lock on to the minimum possible set of states that the sequence represents. For sequential prediction, this is possible by using a mixture of all possible order models (that is, phrase sizes) to assign a probability estimate to the next symbol. To consider different orders of models, we use the Prediction by Partial Match (PPM) family of predictors, which Bhattacharya and Das used to great effect for an LZ78-based predictive framework.[4] However, their method concentrated on the probability of the next symbol appearing in the LZ phrase as opposed to the next symbol in the sequence.

PPM algorithms consider different-order Markov models to build a probability distribution.[6] In our predictive scenario, ALZ builds an order-k Markov model. We next employ the PPM strategy of *exclusion*[7] to gather information from models of order 1 through k to assign the next symbol its probability value.

ALZ's window represents the set of contexts ALZ uses to compute the probability of the next symbol. Our example uses the last phrase *aaa* (which is also the current ALZ window). In this phrase, the contexts that can be used are all suffixes of the phrase, except the window itself (that is, *aa, a*, and the null context). We can contrast this with LeZi Update,[4] where the context for prediction is only the last complete LZ phrase encountered. As a result, a prediction about the next symbol can only be made given the last completed LZ phrase, which could ignore recent history if the current LZ phrase is particularly long.

Suppose we need to compute the probability that the next symbol is *a*. From figure 3, we see that *a* occurs two out of the five times that the context *aa* appears, the other cases producing two null outcomes and one *b* outcome. Therefore, the probability of encountering *a* at the context *aa* is 2 in 5, and we now escape to the order-1 context (that is, switch to the model with the next smaller order) with probability 2 in 5. This corresponds to the probability that the outcome is null, which forms the context for the next lower length phrase. At the order-1 context, we see *a* five out of the 10 times that we see the *a* context, and of the remaining cases, we see two null outcomes. Therefore, we predict *a* at the order-1 context with probability 5 in 10 and escape to the order-0 model with probability 2 in 10. Using the order 0 model, we see *a* 10 times out of the 23 symbols processed so far, so we predict *a* with probability 10 in 23 at the null context. As a consequence, we compute the blended probability of seeing *a* as the next symbol as

$$\frac{2}{5}+\frac{2}{5}\left\{\frac{5}{10}+\frac{2}{10}\left(\frac{10}{23}\right)\right\}$$

This method of assigning probabilities has several advantages:

- It solves the zero-frequency problem. In the earlier example, if we chose only the longest context to decide on probability, it would have returned a zero probability for the symbol *c*, whereas lower-order models show that this probability is indeed nonzero.

UNIX Command Line Prediction

ALZ was also tested on data from the UNIX command line usage of 168 users, which Saul Greenberg compiled.[1] The tests were performed in the manner in which a sequential predictor would typically be employed—that is, trained on the first *n* examples and tested on the next 100. The average predictive accuracy over all data sets for ALZ was 41.34 percent, marginally better than the 40 percent accuracy of the IPAM algorithm, which uses first-order Markov models to perform prediction.[2]

References

1. S. Greenberg, *Using Unix: Collected Traces of 168 Users*, research report 88/333/45, Computer Science Dept., Univ. of Calgary, 1988.

2. B.D. Davison and H. Hirsh, "Predicting Sequences of User Actions," *Predicting the Future: AI Approaches to Time Series Problems*, AAAI Press, 1998, pp. 5–12.

- This blending strategy assigns greater weight to higher-order models in calculating probability if it finds in that context the symbol being considered, while it suppresses the lower-order models owing to the null context escape probability. This is in keeping with the advisability of making the most informed decision.

This blending strategy considers nodes at every level in the trie from orders 1 through k–1 for an order-k Markov model, which results in a worst-case run time of $O(k^2)$. The prediction time is thus bounded by $O(n)$. This assumes a fixed-size alphabet, which implies that the algorithm can search for the child of any node in the trie in constant time. Later, we discuss an improved implementation that provides

$$O(k) = O(\sqrt{n})$$

prediction time.

Application: Smart home inhabitant prediction

Sequential prediction algorithms can enhance various applications. For example, a smart home provides a ready environment for employing ALZ (for an example of a different application, see the "UNIX Command Line Prediction" sidebar). We define a smart environment as one that acquires and applies knowledge about its inhabitants and their surroundings to adapt to them and meet their comfort and efficiency goals. Such a home could ideally control many aspects of the environment, such as climate, water, lighting, maintenance, and entertainment. The MavHome project at the University of Texas at Arlington characterizes the smart home as an intelligent agent that aims to reduce manual interaction between the inhabitant and the home.[8] To achieve this goal, the home must predict which of its devices the inhabitant will interact with next so it can automate that activity. The home will make this prediction only on the basis of previously seen inhabitant interactions—thus, predictive accuracy directly affects the home's performance.

We can characterize MavHome operations by the following scenario. At 6:45 a.m., MavHome increases the heat to warm the house to Bob's preferred waking temperature. The alarm sounds at 7:00 a.m., after which the bedroom light and kitchen coffee maker turn on. Bob steps into the bathroom and turns on the light. MavHome displays the morning news on the bathroom's video screen and turns on the shower. Once Bob finishes grooming, the bathroom light turns off while the kitchen light and display turn on. When Bob leaves for work, MavHome makes sure that doors and windows are locked and starts the lawn sprinklers, despite knowing about the 30 percent chance of rain that day. When Bob arrives home, the hot tub is waiting for him.

Smart home inhabitants typically interact with various devices as part of their routine activities—interactions that are a sequence of events with some inherent pattern of recurrence. For example, your morning routine is likely the same every day—turn on the kitchen light, turn on the coffee maker, turn on the bathroom light, turn off the bathroom light, turn on the toaster, turn off the coffee maker, and so on.

We characterize each inhabitant–home interaction event e as a triple consisting of the device the user interacts with, the change that occurs in that device, and the time of interaction (for example, e = *<Device#, ON/OFF, TIME>*). We also assume that devices are associated with the binary-value "on" and "off" states.

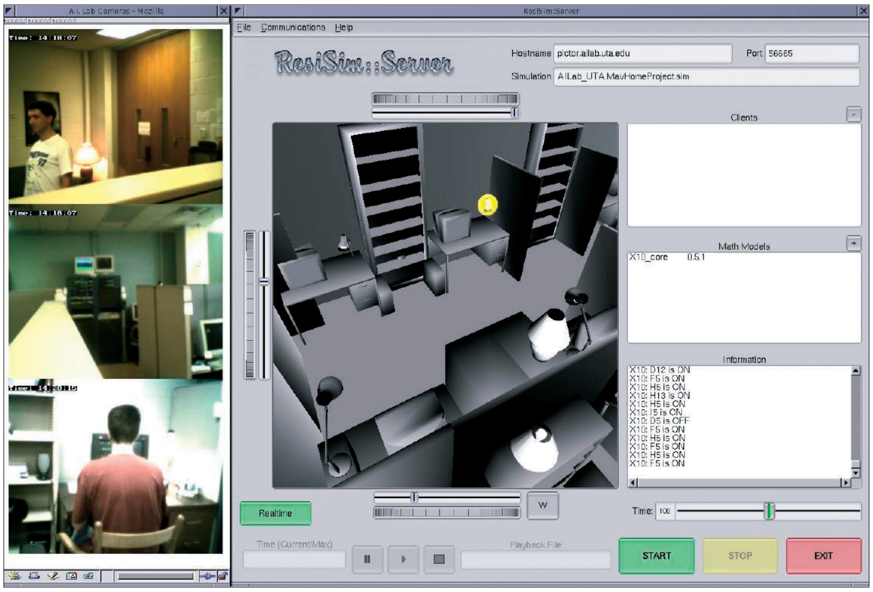
The first set of tests evaluated ALZ's sequential prediction capability, so we didn't consider time information. We therefore identified each unique input symbol, x_t, by the two-tuple consisting of the device ID and that device's state change.

We first tested ALZ on data obtained from a Synthetic Data Generator, which approximates the data that would be obtained in a smart home situation. An SDG can generate data from configurable user-interaction scenarios. SDG parameters let ALZ partially or totally order the scenarios with varying amounts of regularity for the beginning of each scenario and varying distributions of time between each event in the scenarios. Multiple scenarios can also overlap, simulating the interleaving of multiple inhabitants' tasks or activities.

The first set of tests used SDG data sets of 2,000 points. ALZ generated these events from a set of six typical home scenarios reflecting regular daily activity patterns with 15 devices. These patterns consisted of between eight and 12 events (that is, the lengths of the patterns ranged from eight to 12). The data sets were slightly corrupted with randomly inserted events totaling 5 to 15 percent of the data set. When averaged over the six scenarios, the predictive accuracy reached a plateau of 86 percent.

We also tested ALZ on data collected in an actual smart home environment, the MavLab (a testbed for the MavHome). We collected this data by monitoring six participants' device interactions, based on regular daily and weekly activity patterns in April 2003. The 750 data points represent typical data in a home environment, with all the associated randomness and noise. The algorithm peaked at 47 percent accuracy. We also tested ALZ on the real data to see how many of the actual events were among the top five predictions the algorithm made. Altogether, the top five predictions reached 89 percent accuracy. The top-five test implies that, on average, if ALZ chooses a top-five prediction in the smart home for automation, the home would only be right one-fifth of the time. However, the MavHome decision layer can automate more than one event among the top five predictions on the basis of the trade-offs between energy loss due to automating more than is necessary and maximizing inhabitant comfort by automating the correct event. As a baseline comparison, we used approximately 50 devices in these real tests. So, a random guess as to which next device interaction would occur would be correct only 2 percent of the time, and trying to randomly predict from the top five would yield just 20 percent accuracy.

We've successfully used ALZ to control a smart environment such as MavHome.[8] For example, Darin's lamp (in the upper left corner of figure 5) turns on when he enters the room, and Ryan's desk lamp (in the lower left corner) turns on when he sits down to work. These actions occur in response to patterns observed for the inhabitants. We can add motion sensors (shown by the green orbs), temperature sensors, lighting sensors, and other information sources to the ALZ alphabet to improve the generated model's accuracy.



Hostname pictor.ailab.uta.edu
Port 56665
Simulation AILab_UTA.MavHomeProject.sim
X10_core 0.5.1
Time: 100

```
X10: D12 is ON
X10: F5 is ON
X10: H5 is ON
X10: H13 is ON
X10: H5 is ON
X10: I5 is ON
X10: D5 is OFF
X10: F5 is ON
X10: H5 is ON
X10: F5 is ON
X10: H5 is ON
X10: F5 is ON
```

Figure 5. Web camera views of the MavHome environment (left) and ResiSim visualization (right).

Learning relative time between events

For many applications that require prediction, the prediction component is greatly enhanced if it can not only determine the next event in the sequence but also predict how much time will elapse between events. The time between events often depends on the particular sequence or history of events. Given that such a dependency exists in certain sequential processes of events, we can use a sequential predictor such as ALZ to learn a relative time interval between events in the sequence.

We assume that the time interval between events approximates a normal distribution. Therefore, the predictor learns a Gaussian that represents the normal distribution of the time intervals between events, characterized by the mean, μ, and the standard deviation, σ:

$$G(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\left(\frac{(t-\mu)^2}{2\sigma^2}\right)}$$

We store frequency counts of events in the trie. In addition, each node in the trie also incrementally builds a Gaussian that represents the observed normal distribution of the relative time the last event occurred in the corresponding phrase. To efficiently store the timing information, the Gaussians are constructed incrementally without storing each individual event time. The mean of the Gaussian is easily constructed incrementally. The standard deviation can also be built incrementally by recursively defining the value for n data points in terms of the mean and standard deviation for the previous $n-1$ data points as well as the new data point, t_n:

$$\sigma_n = \left(\frac{1}{n}\right)\left\{(n-1)\left[\sigma_{n-1}^2 + \left(\mu_n - \mu_{n-1}\right)^2\right] + \left(t_n - \mu_n\right)^2\right\}$$

To predict the time of an event, information stored at various orders of the trie must be blended to strengthen the resulting prediction. To merge Gaussians from various orders, ALZ randomly generates a set of data points for each Gaussian to create a pool of data points. The number of points that each Gaussian contributes to this pool is proportionate to the weight of the corresponding nodes in the trie. ALZ computes the mean and standard deviation for the merged Gaussian, which represents the learned time interval before the corresponding event in the trie will occur relative to the previous event.

We evaluated ALZ's ability to learn a relative time interval using the synthetic data set. We drew elapsed times from different distributions, depending on the context in which the event took place. Once ALZ converged upon the correct model of sequential events, it could effectively evaluate the learned times' accuracy. As a performance measure, ALZ compared the next event's actual time of occurrence to the predicted time distribution. The resulting algorithm performs well—in 70 percent of the cases, the next event occurred within the mean ± standard deviation of the predicted time.

ALZ and the TDAG algorithm

We compared ALZ with Philip Laird and Ronald Saul's TDAG algorithm, which is based on Markov trees.[9] TDAG constructs a Markov tree of symbols. To keep the tree's size within practical limits, TDAG constrains tree expansion according to user-defined parameters. It uses a set of vine pointers to compute the probability of the next symbol by computing the ratio between the relative frequency counts for the alphabet's symbols and using the pointer to the deepest node in the tree.

Comparison

On our synthetic data, TDAG converges to 100 percent accuracy but requires twice as many training examples than ALZ. On the UNIX data, TDAG's average is 40 percent, marginally lower than ALZ's. These results are based on the TDAG parameters' best setting.

TDAG isn't sensitive to changes in patterns over time. ALZ, on the other hand, weights recent events more heavily and is therefore better tuned to respond to changes in patterns. Also, TDAG's prediction is generated from only the highest-order reliable model and not from combining the information from various order models, which saves turnaround time. For long-term data collection, however, this is a major disadvantage because it makes TDAG insensitive to concept drift.

A more efficient implementation of ALZ

TDAG's prediction is extremely fast because the linear runtime depends on the length of the state queue controlled by user-defined parameters. So, we present a more efficient implementation of ALZ—the ALZ-TDAG algorithm.

TDAG decides whether to extend the Markov tree on the basis of factors such as the number of input symbols seen, the depth of the tree at that node, and the improvement in confidence—all of which are user-defined parameters. In contrast, ALZ-TDAG makes this decision on the basis of whether adding the new node will cause the tree to grow to a height greater than the length of the longest LZ phrase.

ALZ adds nodes to the trie at all positions where a path from the root to the newly added leaf corresponds to a subsequence seen for the first time in the input sequence, and such a path is at most equal to the length of the longest LZ78 phrase. We make use of the observation that the TDAG algorithm is also adding nodes at positions where a path from the root to the newly added leaf corresponds to a subsequence seen in the input sequence. Therefore, the Markov tree ALZ-TDAG built is the same as the trie ALZ built, and the new pruning criterion constrains the tree's growth.

ALZ-TDAG's state queue now only contains pointers to those nodes in the tree that are likely candidates for adding a new node when the next symbol is seen, and it represents the same contexts as the ALZ window. Because the window determines all the contexts that PPM can use for prediction, ALZ-TDAG can use each pointer in the state queue to determine the probability distribution over the alphabet for the next symbol. TDAG maintains the statistics required by the PPM algorithm. Therefore, the time PPM requires to assign probabilities is $O(k)$, where k is the length of the state queue. The value of k here is the ALZ window's length, which we said earlier is

$$O\left(\sqrt{n}\right)$$

This is a significant savings from ALZ's $O(n)$ time. ALZ-TDAG, therefore, can tackle concept drift and also exhibits a fast response time.

ALZ is a fast, effective algorithm for predicting events in sequential data. Its nature makes it useful for smart environments in which you need a model of the inhabitants to predict, and ultimately automate, their activities. Future work will compare this compression-based predictor with alternative sequential predictors and incorporate more robust models of absolute and relative time of events. ■

Acknowledgments

This work was supported in part by US National Science Foundation grant IIS-0121297.

References

1. M. Weinberg and G. Seroussi, *Sequential Prediction and Ranking in Universal Context Modeling and Data Compression,* tech. report HPL-94-111, HP Labs, 1997.
2. M. Feder, N. Merhav, and M. Gutman, "Universal Prediction of Individual Sequences," *IEEE Trans. Information Theory*, vol. 38, no. 4, 1992, pp. 1258–1270.
3. J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable Rate Coding," *IEEE Trans. Information Theory*, vol. 24, no. 5, 1978, pp. 530–536.
4. A. Bhattacharya and S.K. Das, "LeZi-Update: An Information-Theoretic Framework for Personal Mobility Tracking in PCS Networks," *Wireless Networks J.*, vol. 8, nos. 2–3, 2002, pp. 121–135.
5. P. Franaszek, J. Thomas, and P. Tsoucas, "Context Allocation with Application to Data Compression," *IEEE Int'l Symp. Information Theory*, IEEE Press, 1994, p. 12.
6. J.G. Cleary and W.J. Teahan, "Unbounded Length Contexts for PPM," *Computer J.*, vol. 40, nos. 2–3, 1997, pp. 67–75.
7. T.C. Bell, J.G. Cleary, and I.H. Witten, *Text Compression*, Prentice Hall, 1990.
8. S. Das et al., "The Role of Prediction Algorithms in the MavHome Smart Home Architecture," *IEEE Wireless Comm.*, vol. 9, no. 6, 2002, pp. 77–84.
9. P. Laird and R. Saul, "Discrete Sequence Prediction and its Applications," *Machine Learning*, vol. 15, no. 1, 1994, pp. 43–68.

The Authors

Karthik Gopalratnam is a PhD candidate in the University of Washington's Department of Computer Science. His research interests include artificial intelligence and machine learning. He received his BS in computer science engineering from the University of Texas at Arlington. Contact him at the Dept. of Computer Science and Eng., Univ. of Washington, Box 352350, Seattle, WA 98195-2350; karthikg@cs.washington.edu.

Diane J. Cook is a Huie-Rogers Chair Professor in the School of Electrical Engineering and Computer Science at Washington State University. Her research interests include artificial intelligence, machine learning, data mining, robotics, and smart environments. She received her PhD in computer science from the University of Illinois. Contact her at EME 121 Spokane Street, Box 642752, School of Electrical Eng. and Computer Science, Washington State Univ., Pullman, WA 99164-2752; cook@eecs.wsu.edu.