In Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining, August, 1998

Learning to Predict Rare Events in Event Sequences

Gary M. Weiss[†] **and Haym Hirsh**

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
gmweiss@att.com, hirsh@cs.rutgers.edu

Abstract

Learning to predict rare events from sequences of events with categorical features is an important, real-world, problem that existing statistical and machine learning methods are not well suited to solve. This paper describes *timeweaver*, a genetic algorithm based machine learning system that predicts rare events by identifying predictive temporal and sequential patterns. Timeweaver is applied to the task of predicting telecommunication equipment failures from 110,000 alarm messages and is shown to outperform existing learning methods.

Introduction

An *event sequence* is a sequence of timestamped observations, each described by a fixed set of features. In this paper we focus on the problem of predicting *rare* events from sequences of events which contain *categorical* (non-numerical) features. Predicting telecommunication equipment failures from alarm messages is one important problem which has these characteristics. For AT&T, where most traffic is handled by 4ESS switches, the specific task is to predict the failure of 4ESS hardware components from diagnostic alarm messages reported by the 4ESS itself. Predicting fraudulent credit card transactions and the start of transcription in DNA sequences are two additional problems with similar characteristics. For a variety of reasons, these problems cannot be easily solved by existing methods. This paper describes *timeweaver*, a machine learning system specifically designed to solve rare event prediction problems with categorical features by identifying predictive temporal and sequential patterns in the data.

Background

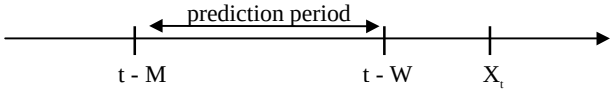
Event prediction problems are very similar to time-series prediction problems. Classical time-series prediction, which has been studied extensively within the field of statistics, involves predicting the next *n* successive observations from a history of past observations (Brockwell & Davis 1996). These statistical techniques are not applicable to the event prediction problems we are interested in because they require numerical features and do not support predicting a specific "event" within a window of time. Relevant work in machine learning has relied on reformulating the prediction problem into a concept learning problem (Dietterich & Michalski 1985). The reformulation process involves *transforming* the event sequence into an unordered set of examples by encoding multiple events as individual examples. The transformation procedure preserves only a limited amount of sequence and temporal information, but enables any concept learning program to be used. This approach has been used to predict catastrophic equipment failures (Weiss, Eddy & Weiss 1998) and to identify network faults (Sasisekharan, Seshadri & Weiss 1996). Non-reformulation based approaches have also been tried. Computational learning theory has focused on learning regular expressions and pattern languages from data, but has produced few practical systems (Jiang & Li 1991; Brazma 1993). Data mining algorithms for identifying *common* patterns in event sequences have been developed, but these patterns are not necessarily useful for prediction. Nonetheless, such algorithms have been used to predict network faults (Manilla, Toivonen & Verkamo 1995).

The Event Prediction Problem

This section defines our formulation of the event prediction problem.

Basic Problem Formulation

An event E_t is a timestamped observation which occurs at time *t* and is described by a set of feature-value pairs. An *event sequence* is a time-ordered sequence of events, $S = Et_1, Et_2, ..., Et_n$, which includes all *n* events in the time interval $t_1 \le t \le t_n$. Events are associated with a domain object *D* which is the source, or generator, of the events. The *target event* is the event to be predicted and is specified by a set of feature-value pairs. Each target event, X_t, occurring at time *t*, has a *prediction period* associated with it, as shown below. The warning time, W, is the "lead time" necessary for a prediction to be useful and the monitoring time, M, determines the maximum amount of time prior to the target event for which a prediction is considered correct.



[Ü] Also AT&T Labs, Middletown NJ 07748

The warning and monitoring times should be set based on the problem domain. In general, the problem will be easier the smaller the value of the warning time and the larger the value of the monitoring time; however, too large a value for the monitoring time will result in meaningless predictions.

The problem is to learn a prediction procedure P that correctly predicts the target events. Thus, P is a function that maps an event sequence to a boolean prediction value. A prediction is made upon observation of each event, so P: Et_1, Et_2,..., $Et_x \rightarrow \{+,-\}$, for each event Et_x The *semantics* of a prediction still need to be specified. A target event is *predicted* if at least one prediction is made within its prediction period, regardless of any subsequent negative predictions. Negative predictions can therefore be ignored, and henceforth "prediction" will mean "positive prediction". A prediction is *correct* if it falls within the prediction period of some target event.

This formulation can be applied to the telecommunication problem. Each 4ESS generated alarm is an event with three features: *device*, which identifies the component within the 4ESS reporting the problem, *severity*, which can take on the value "minor" or "major", and *code*, which specifies the exact problem. Each 4ESS switch is a domain object that generates an event sequence and the target event is any event with *code* set to "FAILURE".

Evaluation Measures

The evaluation measures are summarized in Figure 1. *Recall* is the percentage of target events correctly predicted. S*imple precision* is the percentage of predictions that are correct. Simple precision is misleading since it counts multiple predictions of the same target event multiple times. *Normalized precision* eliminates this problem by replacing the number of correct predictions with the number of target events correctly predicted. This measure still does not account for the fact that incorrect predictions located closely together may not be as harmful as the same number spread out over time. *Reduced precision* remedies this. A prediction is "active" for a period equal to its monitoring time, since the target event should occur somewhere during that period. Reduced precision replaces the number of false predictions with the number of discounted false predictions—the number of complete, non-overlapping, monitoring periods associated with a false prediction. Thus, two false predictions occurring a half monitoring period apart yields 1½ discounted false predictions, due to a ½ monitoring period overlap in their active periods.

$$\text{Recall} = \frac{\text{\# Target Events Predicted}}{\text{Total Target Events}}, \quad \text{Simple Precision} = \frac{TP}{TP + FP}$$

$$\text{Normalized Precision} = \frac{\text{\# Target Events Predicted}}{\text{\# Target Events Predicted} + FP}$$

$$\text{Reduced Precision} = \frac{\text{\# Target Events Predicted}}{\text{\# Target Events Predicted} + \text{Discounted FP}}$$

TP = True Predictions FP = False Predictions

Figure 1: Evaluation Measures for Event Prediction

The Basic Learning Method

Our learning method, which operates directly on the data and does not require the problem to be reformulated, uses the following two steps:

1. *Identify prediction patterns*. The space of prediction patterns is searched to identify a set, C, of candidate prediction patterns. Each pattern $c \in C$ should do well at predicting a subset of the target events.
2. *Generate prediction rules*. An ordered list of prediction patterns is generated from C. Prediction rules are then formed by creating a disjunction of the top *n* prediction patterns, thereby creating solutions with different precision/recall values.

This two step approach allows us to focus our effort on the more difficult task of identifying prediction patterns. Also, by using a general search based method in the first step, we are able to use our own evaluation metrics—something which cannot be done with existing learning programs, which typically use predictive accuracy. For efficiency, our learning method exploits the fact that target events are expected to occur infrequently. It does this by maintaining, for each prediction pattern, a boolean prediction vector of length *n* that indicates which of the *n* target events in the training set are correctly predicted. This information is used in step 1 to ensure that a *diverse* set of patterns is identified and in step 2 to intelligently construct prediction rules from the patterns.

The learning method requires a well defined space of prediction patterns. The language for representing this space is similar to the language for expressing the raw data. A *prediction pattern* is a sequence of events connected by *ordering primitives* that define sequential or temporal constraints between consecutive events. The ordering primitives are defined in the list below, in which A, B, C, and D represent individual events.

- the *wildcard* "*" primitive matches any number of events so the prediction pattern A*D matches ABCD
- the *next* "." primitive matches no events so the prediction pattern A.B.C only matches ABC
- the *unordered* "|" primitive allows events to occur in any order and is commutative so that the prediction pattern A|B|C will match, amongst others, CBA.

The "|" primitive has highest precedence so the pattern "A.B*C|D|E" matches an A, followed immediately by a B, followed sometime later by a C, D and E, in any order. Each feature in an event is permitted to take on the "?" value that matches any feature value. A prediction pattern also has an integer-valued *pattern duration*. A prediction pattern *matches* a sequence of events within an event sequence if 1) the events within the prediction pattern match events within the event sequence, 2) the ordering constraints expressed in the prediction pattern are obeyed, and 3) the events involved in the match occur within the pattern duration. This language enables flexible and noise-tolerant prediction rules to be constructed, such as the rule: *if 3 (or more) A events and 4 (or more) B events occur*

within an hour, then predict the target event. This language was designed to provide a small set of features useful for many real-world prediction tasks. Extensions to this language require making changes only to timeweaver's pattern-matching routines.

A Genetic Algorithm for Identifying Prediction Patterns

We use a genetic algorithm (GA) to identify a diverse set of prediction patterns. Each individual in the GA's population represents part of a complete solution and should perform well at classifying a subset of the target events. Our approach resembles that of classifier systems, which are GAs that evolve a set of classification rules (Goldberg 1989). The main differences are that in our approach rules cannot chain together and that instead of forming a ruleset from the entire population, we use a second step to prune "bad" rules. Our approach is also similar to the approach taken by other GA's which learn disjunctive concepts from examples (Giordana, Saita & Zini 1994).

We use a steady-state GA, where only a few individuals are modified each "iteration", because such a GA is believed to be more computationally efficient than a generational GA when the time to evaluate an individual is large (true in our case due to the assumption of large data sets). The basic steps in our GA are shown below.

1. Initialize population
2. **while** stopping criteria not met
3. select 2 individuals from the population
4. apply mutation operator to both individuals with P_M; else apply crossover operator
5. evaluate the 2 newly formed individuals
6. replace 2 existing individuals with the new ones
7. **done**

The population is initialized by creating prediction patterns containing a single event, with the feature values set 50% of the time to the wildcard value and the remaining time to a randomly selected feature value. The GA continues until either a pre-specified number of iterations are executed or the performance of the population peaks. The mutation operator randomly modifies a prediction pattern, changing the feature values, ordering primitives, and/or the pattern duration. Crossover is accomplished via a variable length crossover operator, as shown in Figure 2. The lengths of the offspring may differ from that of the parents and hence over time prediction patterns of any size can be generated. The pattern duration of each child is set by trying each parent's pattern duration and the average of the two, and then selecting the value which yields the best results.

```
A || B   C   D   E
X   Y || Z
-------------------
A   Z
X   Y   B   C   D   E
```

Figure 2: Variable Length Crossover

The Selection and Replacement Strategy

The GA's selection and replacement strategies must balance two opposing criteria: they must focus the search in the most profitable areas of the search space but also maintain a diverse population, to avoid premature convergence and to ensure that the individuals in the population collectively cover most of the target events. The challenge is to maintain diversity using a minimal amount of global information that can be efficiently computed.

The fitness of a prediction pattern is based on both its precision and recall and is computed using the F-measure, defined in equation 1, where β controls the importance of precision relative to recall. Any fixed value of β yields a fixed bias and, in practice, leads to poor performance of the GA. To avoid this problem, for each iteration of the GA the value of β is randomly selected from the range of 0 to 1, similar to what was done by Murata & Ishibuchi (1995).

$$\text{fitness} = \frac{(\beta^2 + 1)\,\text{precision} \cdot \text{recall}}{\beta^2 \text{precision} + \text{recall}} \qquad \textbf{(1)}$$

To encourage diversity, we use a *niching* strategy called *sharing* that rewards individuals based on how different they are from other individuals in the population (Goldberg 1989). Individuals are selected proportional to their *shared fitness*, which is defined as fitness divided by niche count. The niche count, defined in equation 2, measures the degree of similarity of individual *i* to the *p* individuals comprising the population.

$$\text{niche count}_i = \sum_{j=1}^{n} (1 - \text{distance}(i,j))^3 \qquad \textbf{(2)}$$

The similarity of two individuals is measured using a phenotypic distance metric that measures the distance based on the *performance* of the individuals. In our case, this distance is simply the fraction of bit positions in the two prediction vectors that differ (i.e., the fraction of target events for which they have different predictions). The more similar an individual to the rest of the individuals in the population, the smaller the distances and the greater the niche count value; if an individual is identical to every other individual in the population, then the niche count will be equal to the population size.

The replacement strategy also uses shared fitness. Individuals are chosen for deletion *inversely* proportional to their shared fitness, where the fitness component is computed by averaging together the F-measure of equation 1 with β values of 0, ½, and 1, so the patterns that perform poorly on precision *and* recall are most likely to be deleted.

Creating Prediction Rules

A greedy algorithm, shown below, is used to form a list of prediction rules, S, from the set of candidate patterns, C, returned by the GA. The precision, recall, and prediction vector information computed in the first step for each prediction pattern are used, so that only step 11 requires access to the training set; this step is therefore the most time-intensive step in the algorithm.

```
1.  C = patterns returned from the GA; S = {};
2.  while C ≠ ∅ do
3.     for c ∈ C do
4.        if (increase_recall(S+c, S) ≤ THRESHOLD)
5.        then C = C - c;
6.        else c.eval = PF × (c.precision - S.precision) +
7.                      increase_recall(S+c, S);
8.     done
9.     best = {c ∈ C, ∀x∈ C| c.eval ≥ x.eval}
10.    S = S || best; C = C - best;
11.    recompute S.precision on training set;
12. done
```

This algorithm builds solutions with increasing recall by heuristically selecting the best prediction pattern remaining in C, using the evaluation function on line 6. The evaluation function rewards those candidate patterns that have high precision and predict many target events not predicted by S. The Prediction Factor (PF) controls the importance of precision vs. recall. Prediction patterns that do not increase the recall by at least THRESHOLD are discarded. Both THRESHOLD and PF affect the complexity of the learned concept and can prevent overfitting of the data. The algorithm returns an ordered list of patterns, where the first solution contains the first prediction pattern in the list, the second solution the first two prediction patterns, etc. Thus, a precision/recall curve can be constructed from S and the user can select a solution based on the relative importance of precision and recall for the problem at hand.

The algorithm is quite efficient: if p is the population size of the GA (i.e., p patterns are returned), then the algorithm requires $O(p^2)$ computations of the evaluation function and $O(p)$ evaluations on the training data (step 11). Since the information required to compute the evaluation function is available, this leads to an $O(ps)$ algorithm, given the assumption of large data sets and a small number of target events (where s is the training set size). In practice, fewer than p iterations of the for loop will be necessary, since most prediction patterns will not pass the test on line 4.

Experiments

Timeweaver was applied to the task of predicting 4ESS equipment failures, using a training set of 110,000 alarms reported from 55 4ESS switches. The test set contained 40,000 alarms from 20 *different* 4ESS switches. This data included 1200 alarms which indicate equipment failure. For all experiments, THRESHOLD is set to 1% and PF to 10, and, unless otherwise noted, all results are based on 2000 iterations of the GA. Precision is measured using reduced precision, except in Figure 6 where simple precision is used in order to permit comparison with other approaches. Unless stated otherwise, all experiments use a 20 second warning time and an 8 hour monitoring time.

Figure 3 shows the performance of the learned prediction rules, generated at different points during the execution of the GA. The "Best 2000" curve shows the performance of the prediction rules formed by combining the "best" prediction patterns from the first 2000 iterations. Improvements were not found after iteration 2000.

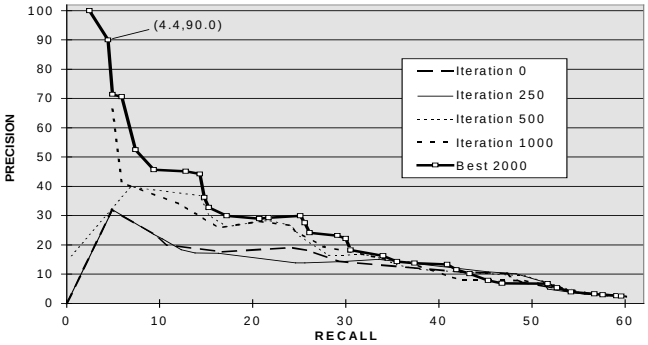


Figure 3: The Performance of the Prediction Rules

These results are notable—a baseline strategy that predicts a failure every warning time (20 seconds) only yields a precision of 3% and a recall of 63%. A recall greater than 63% can never be achieved since 37% of the failures have no events within their prediction period. The pattern 351:<TMSP,?,MJ>*<?,?,MJ>*<?,?,MN> corresponds to the first data point in the "Best 2000" curve in Figure 3. This pattern indicates that within a 351 second time period, a major severity alarm on a TMSP device is followed by a major alarm and then a minor alarm.

The results of varying the warning time, shown in Figure 4, demonstrate that for this domain it is *much* easier to predict failures when only a short warning time is required. These results make sense since we expect the alarms most indicative of a failure to occur shortly before the failure.

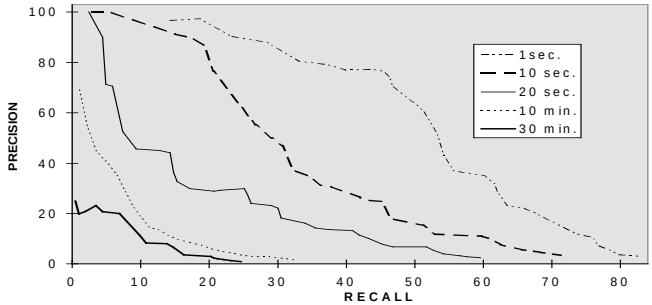


Figure 4: Effect of Warning Time on Learning

Figure 5 shows that increasing the monitoring time from 1 to 8 hours significantly improves timeweaver's ability to predict failures; we believe no such improvement is seen when the monitoring time increases to 1 day because the larger prediction period leads timeweaver to focus its attention on "spurious correlations" in the data.

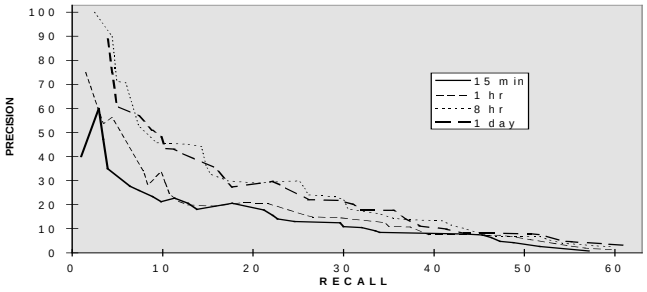


Figure 5: Effect of Monitoring Time on Learning

Comparison with Other Methods

Timeweaver's performance was compared to C4.5rules (Quinlan 1993) and RIPPER (Cohen 1995), two rule induction systems, and FOIL, a system that learns logical definitions from relations (Quinlan 1990). In order to use the "example-based" rule induction systems, the event sequences were transformed into examples by using a sliding window. With a window size of 2, examples are generated with the features: device1, severity1, code1, device2, severity2, and code2. Each example's classification is based on whether the last event included in the example falls within the prediction period of a target event. Because equipment failures are rare, the class distribution of the generated examples is skewed; this prevented C4.5rules and RIPPER from predicting any failures. To compensate, various values of misclassification cost (i.e., the relative cost of false negatives to false positives) were tried and the best results are shown in Figure 6. In the figure, the number after the *w* indicates the window size and the number after the *m* the misclassification cost. FOIL is a more natural choice for solving event prediction problems since the problem is easily translated into a relational learning problem. With FOIL, the sequence information is encoded via the extensionally defined *successor* relation.

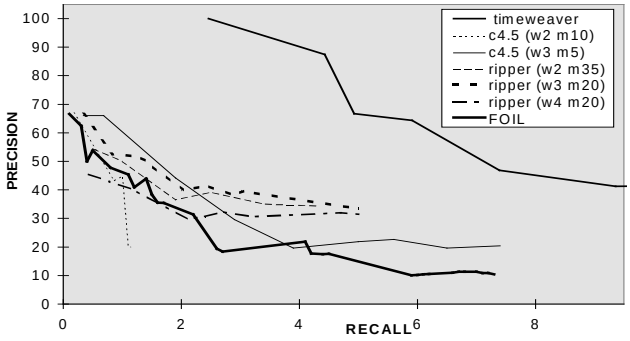


Figure 6: Comparison with Other ML Methods

Figure 6 shows that timeweaver outperforms the other methods; it produces higher precision solutions that span a much wider range of recall values. RIPPER's best performance resulted from a window size of 3; due to computational limits, a window size greater than 3 could not be used with C4.5rules. FOIL produced results inferior to the other methods, but its performance might improve if relations encoding temporal information were added.

Timeweaver can also be compared against ANSWER, the expert system which handles 4ESS alarms (Weiss, Ros & Singhal 1998). ANSWER uses a simple thresholding strategy to generate an alert when more than a specified number of interrupt alarms occur within a specified time period. These alerts can be interpreted as a prediction that the device generating the alarm is going to fail. Various thresholding strategies were tried and those yielding the best results are shown in Figure 7. By comparing these results with those in Figure 3, we see that timeweaver yields results with precision 3-5 times higher for a given recall value. Much of this improvement is undoubtedly due to the fact that timeweaver's concept space is much more expressive than that of a simple thresholding strategy.

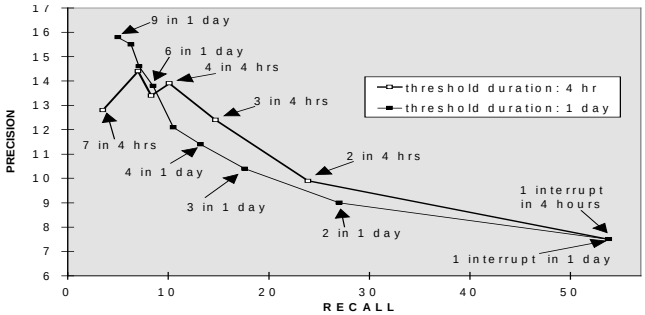


Figure 7: Using Interrupt Thresholding to Predict Failures

Conclusion

This paper investigated the problem of predicting rare events from sequences of events with categorical features and showed that timeweaver, a GA-based machine learning system, is able to outperform existing methods at this prediction task. Additional information is available from http://paul.rutgers.edu/~gweiss/thesis/timeweaver.html.

References

Brazma, A. 1993. Efficient identification of regular expressions from representative examples. In *Proceedings of the Sixth Annual Workshop on Computational Learning Theory*, 236-242.

Brockwell, P. J., and Davis, R. 1996. *Introduction to Time-Series and Forecasting*. Springer-Verlag.

Cohen, W. 1995. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, 115-123.

Dietterich, T., and Michalski, R. 1985. Discovering patterns in sequences of events, *Artificial Intelligence*, 25:187-232.

Giordana, A., Saitta, L., and Zini, F. 1994. Learning disjunctive concepts by means of genetic algorithms. In *Proceedings of the Eleventh International Conference on Machine Learning*, 96-104.

Goldberg, D. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley.

Jiang, T., and Li, M. 1991. On the complexity of learning strings and sequences. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, 367-371.

Manilla, H., Toivonen, H., and Verkamo, A. 1995. Discovering frequent episodes in sequences. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, 210-215, AAAI Press.

Murata, T., and Ishibuchi, H. 1995. MOGA: Multi-objective genetic algorithms. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, 289-294.

Quinlan, J. R., 1990. Learning logical definitions from relations, *Machine Learning*, 5: 239-266.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Sasisekharan, R., Seshadri, V., and Weiss, S. 1996. Data mining and forecasting in large-scale telecommunication networks, *IEEE Expert*, 11(1): 37-43.

Weiss, G. M., Eddy, J., and Weiss, S. 1998. Intelligent technologies for telecommunications. In *Intelligent Engineering Applications*, Chapter 8, CRC Press.

Weiss, G. M., Ros J. P., and Singhal, A. 1998. ANSWER: network monitoring using object-oriented rules. In *Proceedings of the Tenth Conference on Innovative Applications of Artificial Intelligence*, AAAI Press.