A comparison of sequence-learning approaches: implications for intelligent user interfaces

Tessa Lau
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
tlau@cs.washington.edu

February 14, 1999

1 Introduction

As computer applications become increasingly complex, the human-computer interfaces for such applications grow complex as well. There is a strong need for intelligent user interfaces (IUI) that not only assist novice users in navigating complex applications, but allow expert users to accomplish their tasks more efficiently.

In particular, we are interested a subproblem of IUI known as programming by demonstration (PBD) [2]. In PBD, a user constructs a program not by writing code in a programming language, but by demonstrating the effects of the program in the user interface. Using PBD, a user is able to communicate with an application using the user interface she is accustomed to using, and describe her task in terms of user interface actions. We define two challenges that must be solved to implement a PBD system:

1. **Action prediction**: given that a user has performed a sequence of actions already, what action is she likely to perform next?

2. **Task generalization**: given that a user has performed some sequence of actions, what task is she likely to be performing? Can we generalize from the action sequence to an abstract representation of her task?

A complete PBD system solves these two problems in conjunction. Notably, it is able to take the sequence of actions performed by a user, generalize from that sequence to a model of the user's task, and use this model to predict her subsequent actions.

Both of these problems can be viewed as sequence learning problems. Work in the field of sequence learning aims to use inductive techniques on training data that is explicitly ordered. Many interesting domains show this property, besides the domain of user interface actions: speech signals, alarms in telecommunications networks, customer transactions, and DNA sequences. In many cases, the ordering is temporal. In these cases, the time at which an example occurs is significant, as is the temporal relationship between different examples.

This paper presents a survey of four approaches to the problem of sequence learning and compares them on their applicability to the two challenges of action prediction and task generalization. Section 2 describes and compares two data mining approaches to sequence learning. Section 3 presents two Markov process approaches to sequence learning. In section 4 we propose a taxonomy of the problem spaces addressed by each of the four approaches, and speculate on hybrid algorithms that combine the best of each. Section 5 puts this work in context by examining related work. Finally, section 6 concludes.

2 Data mining approaches to sequence learning

One popular approach to sequence learning comes from the data mining community and is targeted at extracting frequently-occurring patterns in large databases of sequence data. This section presents two such algorithms (WINEPI by Mannila et al [10], and AprioriAll by Srikant and Agrawal [1, 14]).

At a high level, sequence learning algorithms first learn a model of their input, and then use this model to predict future elements in the sequence. In both of the data mining algorithms considered in this section, the model of the input sequence consists of a set of frequently-occurring patterns (possibly-discontiguous subsequences of the input). These patterns together form a representation of common sequences, and are used as the basis for the prediction step, as described below.

The two algorithms described in this section share a common framework and can be viewed as parameterizations of this single framework. This section first presents the framework, then describes the algorithms WINEPI and AprioriAll in terms of this framework.

2.1 A unifying framework for data mining patterns

The two data mining algorithms discussed in this section take as input a sequence of events, and output a set of frequent patterns mined from that sequence. We introduce some common terminology before describing the algorithms.

A sequence is a list of events. Each event occurs at a specific time (represented as an integer) and is of a certain type. A pattern is a tuple $(V, \leq, g)$ where V is a set of nodes, $\leq$ is a partial order over the nodes, and g is a mapping that associates each node with an event type. Serial patterns impose a total ordering on all nodes in the pattern, while parallel patterns impose no ordering on the nodes.

A pattern *occurs* in a sequence if there exists a mapping between nodes in the pattern and events in the sequence such that each node is mapped to a distinct event, and the ordering over the nodes is preserved. A pattern is said to be *frequent* over a sequence if it occurs more than the minimum frequency *min_fr* in the sequence. This constant is defined differently for WINEPI and AprioriAll, as will be discussed in the descriptions of the algorithms below. First, however, we describe the commonalities between the two algorithms.

Figure 1 shows a pseudocode shell for the WINEPI and AprioriAll algorithms. Each algorithm instantiates the framework by implementing the four functions `Transform_Input`, `Initialize_Candidates`, `Find_Frequent`, and `Make_Candidates`.

```
    # Initialization
1:  Transform_Input()
    # Iteration variable
2:  pat_len = 1
3:  candidate_set[pat_len] = Initialize_Candidates()
    # Iteration
4:  while candidate_set[pat_len] is not empty:
5:      frequent_set[pat_len] = Find_Frequent(candidate_set[pat_len])
        # Generate candidate set of patterns of length pat_len+1
6:      candidate_set[pat_len+1] = Make_Candidates(frequent_set[pat_len])
        # Next iteration
7:      pat_len = pat_len + 1
    # Output
8:  output frequent_set[pat_len] for all pat_len
```

Figure 1: Pseudocode for the data mining sequence learning algorithms. Functions typeset in *italic* are implemented differently in WINEPI and AprioriAll.

The algorithm begins with `Transform_Input`, an optional initialization step. AprioriAll transforms its input database into a more tractable representation, while WINEPI performs no initialization.

The algorithms then call the `Initialize_Candidates` function to construct the starting set of candidate patterns of unit length. This candidate set forms the basis of an inductive loop that terminates when the set of candidates becomes empty. On pass i through the loop, the algorithm computes the frequent set of length i candidates by making a pass over the database.

The `Make_Candidates` function takes a frequent set of patterns of length i and joins it against itself to form a candidate set of items of length $i+1$ as follows. For each pair of patterns a and b such that a and b have $i-1$ event types in common, create a new pattern c that contains the $i-1$ common event types, plus the two event types in a and b that are not part of the common pattern. For example, two sequences `ABCD` and `ABCE` are joined to form `ABCDE`.

Some patterns in the candidate set may be pruned at this point. If a sequence is frequent, all of its subsequences must also be frequent (since the subsequence can be found in the input sequence at least as many times as the sequence itself). Therefore if a sequence contains a subsequence that is not frequent, the sequence cannot possibly itself be frequent. Subsequences may be discontiguous. If a proposed candidate sequence of length $i+1$ contains a subsequence of length i that is not frequent (*i.e.*, it does not appear in the frequent set for length i) then that sequence cannot possibly be frequent, and it is pruned from the set of candidate sequences of length $i+1$. In our example, the pattern `ABCDE` will be pruned if any of the patterns `BCDE`, `ACDE`, `ABDE`, `ABCE`, or `ABCD` are not frequent.

When the candidate set becomes empty, the iteration terminates and the algorithm returns a list of all the frequent sets found over the course of the iteration. Otherwise, the algorithm continues with the next iteration, using the newly created candidate set.

This algorithm framework finds the set of all frequent patterns in the input sequence. In order to use these patterns for sequence prediction, one creates a set of *association rules*—rules of the form $\alpha \xrightarrow{c} \beta$ such that if the pattern α occurs in the input sequence, β is likely to also occur with confidence c. Assume β is a frequent pattern. Pick a subsequence α of β that is also frequent. The ratio $c = frequency(\beta)/frequency(\alpha)$ is the confidence of the association rule.

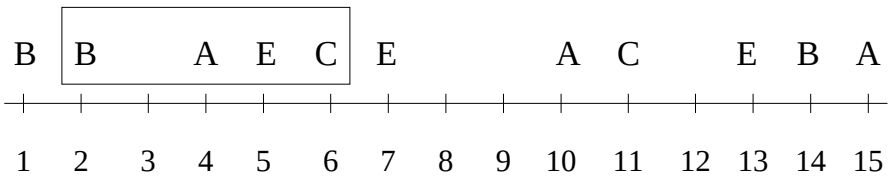

Figure 2: An example WINEPI input sequence. Events in the sequence are shown above the line, and the time of each event is shown below the line. The box shows the window (2, 7) of size 5 that contains events B, A, E, and C.

2.2 WINEPI: finding frequent episodes

WINEPI [10] is an algorithm for finding frequent episodes (patterns) in a telecommunications alarm event log. The input to WINEPI is a sequence of N events starting at time t_s and ending at time t_e, such that all N events occur within this time period ($t_s \leq time(e_i) \leq t_e, 1 \leq i \leq N$); no two events can occur at the same time, nor does an event occur at every time point. Figure 2 shows a sequence starting at time 1 and ending at time 15. WINEPI finds frequent patterns that occur inside fixed-size windows over the input sequence. A window of size *winsize* is a range of times $(t_s, t_s + winsize)$ over the input sequence. For example, the window (2, 7) is a window of size 5 that includes events B, A, E, and C (but not the E that occurs at time 7).

A pattern occurs in a window if there is a one-to-one mapping from nodes in the pattern to events in the window, and the pattern's ordering constraints are met. A pattern is deemed frequent if it occurs in more than a fixed percentage *min_fr* of the total number of windows in the input sequence.

WINEPI only handles two types of patterns: parallel (no ordering between events in the pattern) and serial (all events in the pattern totally ordered). An example parallel pattern in WINEPI is $((x, y), \emptyset, (x \rightarrow A, y \rightarrow C))$. x and y are the nodes in the pattern, the set of ordering constraints over the nodes is empty, node x maps to an event of type A, and node y maps to an event of type C. Considering windows of size 3 in Figure 2, this pattern occurs in windows (4, 7), (9, 12), and (10, 13). For brevity, we refer to this pattern as `AC`. Enhancements to the algorithm to handle patterns that are not strictly parallel or serial are sketched out in [10].

WINEPI follows the general algorithm framework presented in section 2.1. It performs no initial transfor-

mation, so the `Transform_Input` function does nothing. The `Initialize_Candidates` function constructs the first candidate set by returning all candidate patterns of length one (*i.e.*, those patterns containing only a single event).

WINEPI then iterates until the set of candidates is empty. On the i^{th} iteration, the algorithm calls `Find_Frequent` to determine which of the set of candidates of length i are actually frequent. `Find_Frequent` makes a pass over the input sequence and counts the number of fixed-size windows in which the candidate pattern appears.

Once the frequent set is found, it is joined against itself to construct a candidate set of size $i + 1$ using the `Make_Candidates` function. Any of these generated candidates may be immediately pruned if it contains a subsequence of length i that is not frequent (*i.e.*, the subsequence is not a member of `frequent-set[i]`).

When the candidate set is empty, the algorithm terminates and returns all frequent sets discovered thus far. These frequent patterns are turned into association rules for sequence prediction as discussed in section 2.1.

An alternative algorithm (MINEPI) is discussed in [10]. Instead of counting occurrences of patterns in windows, MINEPI finds all minimal occurrences of patterns. A minimal occurrence of a pattern p is a window w such that p occurs in w, and p does not occur in any subwindow of w. MINEPI is more useful for finding association rules such that the patterns on the left and right sides of the association rule can occur within different-sized windows.

2.3 AprioriAll: mining association rules

The input to AprioriAll is a database of customer-sequences. Each customer-sequence represents the items purchased (over time) by a single customer, as a list of transactions ordered by the time of the transaction. A customer may purchase more than one item in a single transaction. For example, a bookstore might be interested in the customer-sequence: (*Foundation, Ringworld*) at time 1, (*Second Foundation*) at time 5, and (*Ringworld Engineers*) at time 10. In this sequence, the customer purchased both *Foundation* and *Ringworld* in a single transaction at time 1, purchased *Second Foundation* by itself at time 5, and later purchased *Ringworld Engineers*.

In AprioriAll terms, a pattern consists of an ordered list of sets of items (itemsets): $\langle s_1 s_2 \ldots s_n \rangle$. Each itemset s_i is a set of items taken from the domain I of possible items. A sequence $\langle a_1 a_2 \ldots a_n \rangle$ is a subsequence of another sequence $\langle b_1 b_2 \ldots b_m \rangle$ if there exist integers $i_1 < i_2 < \cdots < i_n$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$, $\ldots, a_n \subseteq b_{i_n}$. A pattern is said to be supported by a customer-sequence if it is a subsequence of the customer-sequence. Frequent patterns are patterns that are supported by more than a constant percentage *min_fr* of customer-sequences. The length of a pattern is the number of itemsets it contains.

For example, a customer-sequence $\langle (10, 40, 80), (20, 30), (50) \rangle$ has length 3. It contains the subsequence $\langle (10, 40), (50) \rangle$.

Before finding the frequent patterns, AprioriAll transforms its input to make it more amenable to finding frequent patterns as follows (the `Transform_Input` function). First it finds the set of all frequent patterns of length one. This is equivalent to finding those itemsets that are supported by a sufficient number of customer-sequences. Then it maps each such itemset to an integer, and transforms each customer-sequence into sequences of sets of these integers by converting each transaction into the set of itemsets contained in it. Itemsets that are not frequent are simply dropped from the customer-sequence. This transformation minimizes the time needed to test whether an itemset is contained within a sequence, since it becomes simply a test of set membership rather than a subset test.

This transformation step converts itemsets into integers. After this transformation, patterns are simply sequences of integers. In our example, suppose the frequent itemsets are $(10, 40)$, (80), (30), and (50). We map these to the integers 1 through 4. The customer sequence becomes $\langle (1, 2), (3), (4) \rangle$ and the subsequence is $\langle 1, 4 \rangle$.

The `Initialize_Candidates` function returns the set of all frequent itemsets previously computed in the `Transform_Input` step, which corresponds to all frequent sequences of length one.

AprioriAll then uses the standard inductive algorithm to find the set of all frequent patterns. In the i^{th} iteration, the `Find_Frequent` function steps through all customer-sequences in the database, and identifies which candidate patterns have enough support to be frequent. These frequent patterns form the set `frequent-set[i]`.

The `Make_Candidates` function performs a join over the frequent patterns of length i to find the set of

candidates of length $i+1$. Two patterns can be joined if they have a common subsequence of length $i-1$; the joined pattern contains this subsequence and the two itemsets that were not common to the original patterns. To form candidates of length 2, all pairs of candidates are joined. As in WINEPI, a candidate pattern of length $i+1$ may be pruned if it contains a subsequence that is not frequent.

The algorithm then proceeds to the next iteration, and terminates when the candidate set becomes empty. On termination, AprioriAll returns the set of all frequent patterns found over the course of the algorithm's run.

The AprioriAll algorithm discards the transaction time information on each transaction. As a consequence, AprioriAll has two time-related limitations. First, all items in an itemset must be bought in the same transaction, even though the next transaction may occur only a short time later. Second, long delays between transactions are ignored; a customer who first bought *Foundation* and then bought *Second Foundation* a year later would provide just as much support for a pattern as one who bought them within two months of each other.

These limitations are addressed by the GSP algorithm [14], an extension to AprioriAll. In AprioriAll, all of the items in an itemset must be bought in the same transaction. In contrast, GSP removes the single-transaction restriction by treating each itemset as a parallel pattern and allowing items in the itemset to be bought in different transactions as long as the transactions occur within a constant window-size of time. These parallel patterns are combined sequentially to make a serial pattern of parallel patterns.

To address the transaction-delay problem, GSP adds two parameters to the algorithm: min-gap and max-gap. The min-gap constraint restricts the minimum time between the last event in one parallel pattern and the first event in the next parallel pattern. The max-gap constraint restricts the maximum time between the first event in a parallel pattern and the last event in the next parallel pattern. Together, these constraints assure that two sequential parallel patterns neither occur too closely spaced in time nor too far apart.

An orthogonal enhancement in GSP allows the user of the algorithm to define a hierarchy of item types and perform pattern matching based on this hierarchy rather than a simple equality test.

2.4 Discussion of WINEPI and AprioriAll

The major difference between the two algorithms is that whereas WINEPI is given a single input sequence, AprioriAll's input is segmented by the customer making the transaction. Thus, while the support for a pattern in WINEPI is the number of windows over the single input sequence that contain that pattern, the support in AprioriAll is the number of customer-sequences containing the pattern.

A second difference is the types of patterns each algorithm can find. The basic WINEPI algorithm can find either parallel or serial patterns, but not both simultaneously (although an extension is discussed to generalize WINEPI to handling patterns that are not strictly serial or parallel). In contrast, AprioriAll finds only serial patterns.

WINEPI finds only patterns that occur within a fixed-size window over the input sequence. As a consequence, WINEPI is useful for finding patterns with temporal locality—patterns whose constituent events occur near each other in time. From the standpoint of using WINEPI for mining common sequences of user actions, this assumption is sensible, since we believe that related user actions will exhibit temporal locality. In contrast, AprioriAll finds patterns that occur within a single customer-sequence. This algorithm could be useful for intelligent user interfaces if the sequence of user actions has already been segmented in some fashion (for example, by user, or perhaps by the user's task).

From the standpoint of the challenges proposed in our introduction, both of these algorithms would be suitable for action prediction and task generalization. The models they learn of the input sequence are sets of frequent association rules; these association rules represent commonly occuring straight-line programs executed by the user, and are useful in predicting the user's next actions. Since they only cover frequently-occurring sequences of actions, they are tolerant of noisy data (extraneous actions not directly related to the task at hand). Both WINEPI and AprioriAll are able to model interleaved tasks—processes with temporal overlap that generate interleaved events.

The main drawback of the data mining algorithms from the standpoint of IUI is that they require a large corpus of data, which makes them more useful as offline algorithms. For example, they could collect a log of the user's actions during the course of a day, learn on it overnight, and then use the association rules the next day for prediction.

3 Markov process approach

In this section, we present two approaches to sequence learning and prediction: hidden Markov models (HMMs) and TDAG. Both construct a model of the input sequence that may be used to predict the next event in a sequence based on the events immediately preceding it. We first describe each of the algorithms in turn, then compare them on their applicability to intelligent user interfaces.

3.1 Hidden Markov models

Hidden Markov models (HMMs) are a general-purpose method for sequence learning.

A Markov model consists of a set of states and probabilities of transitions between the states. More formally, a Markov model is a tuple (S, K, A, B, π) where S is the set of states, K is the output alphabet, A is the state transition matrix (a_{ij} is the probability of transitioning from state i to state j), B is the output token probability matrix (b_{ijt} is the probability of outputting token o_t during the transition from state i to state j), and π is the initial probability distribution (π_i is the probability of starting in state i).

In a visible Markov model, the state of the model is known at all times. In a hidden Markov model, the state is unknown and must be guessed from the observed output sequence. Markov processes are characterized by two assumptions:

1. **Limited horizon**: the next state depends only on the last N states
2. **Time invariant**: the probability distributions don't change over time

Any process satisfying these assumptions can be modelled using Markov methods. A model for which the next token depends only on the last token is also known as a first-order Markov model. An N^{th} order Markov model allows the next token to depend on the N past tokens. Note that any model with a finite size token horizon can be recoded as a first-ordel model by considering each possible combination of past tokens as a single state in the first-order model.

The advantage to modelling a process using Markov techniques is that there exist efficient solutions to learning the model from observed sequences of data. The three fundamental problems of hidden Markov models are the following:

1. Given an observation sequence O and a model μ, how does one efficiently compute $P(O|\mu)$, the probability of the observation sequence given the model?

2. Given an observation sequence O and a model μ, how does one choose an "optimal" state sequence $X = X_1 \ldots X_t$, *i.e.*, one that best explains the observations?

3. Given an observation sequence O and a space of possible models $\mu = (A, B, \pi)$, how does one adjust the parameters to the model so as to maximize $P(O|\mu)$, the probability of the observation sequence given the model?

HMMs are used to perform sequence prediction as follows. First, a number of models are learned from different training sequences using the solution to Problem 3; one model is learned for each training sequence. To predict the next token of a target sequence, the solutions to Problems 1 and 2 are used. First the solution to Problem 1 is applied to find the most likely model given the target sequence. Then the solution to Problem 2 is applied to find the most likely state sequence using that model. Finally, a prediction is made based on the state the model is in after seeing the target sequence.

We first describe the solutions to these three problems, then discuss applying HMMs to IUI in more detail.

3.1.1 Problem 1: Finding $P(O|\mu)$

The first problem is solved with the Forward-Backward Procedure, a dynamic-programming algorithm for calculating the probablity of observing a sequence given a model.

Forward procedure: Define the forward variable $\alpha_i(t) = P(o_1 o_2 \ldots o_{t-1}, X_t = i|\mu)$ as the probability of ending up in state s_i at time t, given that the observation sequence $O = o_1 \ldots o_{t-1}$ has been seen. The forward variables are computed inductively as follows:

1. Initialization

$$\alpha_i(1) = \pi_i \qquad 1 \leq i \leq N$$

2. Induction

$$\alpha_j(t+1) = \sum_{i=1}^{N} \alpha_i(t) a_{ij} b_{ijt} \qquad 1 \leq t \leq T, 1 \leq j \leq N$$

3. Total

$$P(O|\mu) = \sum_{i=1}^{N} \alpha_i(T+1)$$

The Forward Procedure suffices to solve Problem 1 (finding $P(O|\mu)$). However, we also present the Backwards Procedure here because it is useful in solving the third problem.

Backwards procedure: Define the backwards variable $\beta_i(t) = P(o_t \dots o_T | X_t = i, \mu)$ as the total probability of seeing the rest of the output sequence given that we were in state s_i at time t. It is computed inductively as follows:

1. Initialization

$$\beta_i(T+1) = 1 \qquad 1 \leq i \leq N$$

2. Induction

$$\beta_i(t) = \sum_{j=1}^{N} a_{ij} b_{ijt} \beta_j(t+1) \qquad 1 \leq t \leq T, 1 \leq i \leq N$$

3. Total

$$P(O|\mu) = \sum_{i=1}^{N} \pi_i \beta_i(1)$$

Both the Forwards and Backwards procedures can be used to solve Problem 1.

3.1.2 Problem 2: Finding the optimal state sequence

The Viterbi algorithm finds the most likely sequence of states given an observed sequence. Define $\delta_j(t)$, the probability of the most probable path to state j at time t, as follows:

$$\delta_j(t) = \max_{X_1 \dots X_{t-1}} P(X_1 \dots X_{t-1}, o_1 \dots o_{t-1}, X_t = j | \mu)$$

It is computed as follows, where argmax is a function that returns the variable for which its argument is maximized:

1. Initialization

$$\delta_j(1) = \pi_j \qquad 1 \leq j \leq N$$

2. Induction

$$\delta_j(t+1) = \max_{1 \le i \le N} \delta_i(t) a_{ij} b_{ijt} \qquad 1 \le j \le N$$

Store backtrace:

$$\psi_j(t+1) = \arg \max_{1 \le i \le N} \delta_i(t) a_{ij} b_{ijt} \qquad 1 \le j \le N$$

3. Termination. The backtrace ψ_j is used to read out the most likely state sequence, working backwards in time.

$$\begin{aligned} \hat{X}_{T+1} &= \arg \max_{1 \le i \le N} \delta_i(T+1) \\ \hat{X}_t &= \psi_{\hat{X}_{t+1}}(t+1) \\ P(\hat{X}) &= \max_{1 \le i \le N} \delta_i(T+1) \end{aligned}$$

3.1.3 Problem 3: Finding the best model

This is the most difficult of the problems, and has no optimal solution. However, approximation procedures exist; the Baum-Welch method (also known as the Forward-Backward algorithm) is an iterative hill-climbing algorithm for iteratively updating model parameters so as to maximize $P(O|\mu)$ for a giving training sequence.

Baum-Welch algorithm: Define $p_t(i,j), 1 \le t \le T, 1 \le i,j \le N$, as the probability of traversing an arc in the Markov model at time t given observation sequence O (see [13, 11] for derivations):

$$p_t(i,j) = \frac{\alpha_i(t) a_{ij} b_{ijt} \beta_j(t+1)}{\sum_{m=1}^{N} \alpha_m(t) \beta_m(t)}$$

In addition, define $\gamma_i(t) = \sum_{j=1}^{N} p_t(i,j)$.

Then, given an existing model $\mu = (A, B, \pi)$ we can estimate a new model $\hat{\mu}$ using the following definitions:

$$\begin{aligned} \hat{\pi}_i &= \gamma_i(1) \\ \hat{a}_{ij} &= \frac{\sum_{t=1}^{T} p_t(i,j)}{\sum_{t=1}^{T} \gamma_i(t)} \\ \hat{b}_{ijt} &= \frac{\sum_{t:o_t=k, 1 \le t \le T} p_t(i,j)}{\sum_{t=1}^{T} p_t(i,j)} \end{aligned}$$

Baum proved that $P(O|\hat{\mu}) \ge P(O|\mu)$ and thus that the model $\hat{\mu}$ better fits the observed sequence on each iteration, until it reaches a local maximum.

3.1.4 Applying Markov models to IUI

The main application of Markov models as described in Rabiner [13] is speech recognition. In this application, a different Markov model is learned for each word in the vocabulary. Given a speech sequence of an unknown word, and a set of learned Markov models for each of the words in the vocabulary, the task is then to guess the word by finding the most likely Markov model given this observed sequence. This is exactly the solution to Problem 1 described in the previous section.

Neither Rabiner nor Manning [11] directly address how to apply Markov models to the two PBD problems with which we are concerned. However, the task of action prediction is an application of the solutions to Problems 1 and 2 as described in the previous section. First one finds the Markov model most likely given the observed input sequence, then one finds the optimal state sequence through that model given the input sequence. Given that the state sequence ends in a state s_i at time $t-1$, the probability distribution for the next token is then:

$$P(o_t|X_{t-1},\mu) = \sum_{j=1}^{N} a_{ij} b_{ijt}$$

The second problem in IUI is that of task generalization: generalizing from a sequence to an abstract model of the sequence such that the sequence can be recreated from the model. We note that a Markov model is indeed such a model. It captures the statistical regularities in an input sequence and represents the sequence as a probabilistic finite state machine. However, the drawback of using Markov models to represent user tasks is that the models are nondeterministic. Thus a given Markov model represents a (possibly infinite) set of programs (*i.e.*, sequences of tokens), not all of which may be meaningful.

An alternate method of applying Markov models to task generalization is to parallel the work in speech recognition and learn a distinct Markov model for each task. An unknown input sequence is then tested against these models using the solution to Problem 1, and the most likely model is selected as the most probable task. The drawback to this method is the need for a large library of tasks and action sequences representing these tasks. This solution bears a strong resemblance to the problem of plan recognition, discussed below in section 5.

The above discussion of Markov models assumes a fixed structure (*i.e.*, number of states and connectedness of those states). In some domains, the number of states corresponds naturally to a unit in the domain.

For example, in parts of speech tagging, it is natural to have one state in the model for each part of speech. An open problem in the application of Markov models to IUI is to discover a natural mapping between model states and objects in the domain of user actions and tasks.

3.2 TDAG: Discrete sequence prediction

The TDAG algorithm builds a Markov tree representation of the input sequence. Each node in this tree corresponds to one token in the input sequence; the children of a node represent tokens immediately following that node's token in the input sequence. Pseudocode for the TDAG algorithm is shown in Figure 3. Figure 4 shows the tree resulting from adding the string `ab` to an empty tree.

```
     # State is initialized to contain only the root node
1:   state = list of [RootNode]

     # Input a new token x
2:   function input(x):
3:        let new_state = list of [RootNode]
4:        for each node v in state:
5:             let u = make_child(v, x)
6:             enqueue u onto new_state
7:        state = new_state

8:   function make_child(v, x):
9:        if children(v) has no node u with token x:
10             let u = new_node(x) and add u to children(v)
11:            u.in_count = 0
12:            u.out_count = 0
13:       u.in_count++
14:       v.out_count++

15:  function project_from(v):
16:       let projection = empty list
17:       for each child u in children(v):
18:            add [u.token, (u.in_count/v.out_count)] to projection
19:       return projection
```

Figure 3: Pseudocode for the TDAG algorithm for discrete sequence prediction.

TDAG maintains a state list: a list of nodes that can be extended at a given point in time. The state list starts out containing only the root node, representing an empty tree. When TDAG receives a new input token t, it updates the tree structure as follows. Initialize the new-state variable to contain only the root. For each node u on the state list, find the child of u corresponding to token t (call this node v; it is created

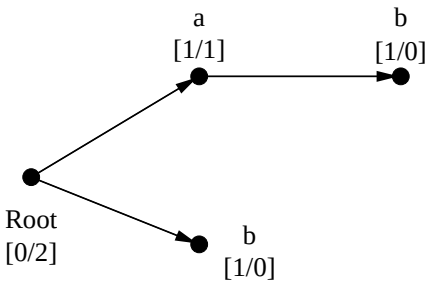

Figure 4: TDAG tree showing the result of adding the string `ab` to an empty tree. Numbers in brackets near a node show the `in-count` and `out-count`, respectively, of that node. The state list after the string has been added is `[Root, ab, b]`.

if it doesn't exist). The out-count of u and the in-count of v are incremented, representing the fact that this token appears on the transition from u to v. v is added to the new-state list. After all nodes on the state list are processed, the new-state list becomes the state list and the algorithm is ready for the next input token.

Each node in the tree can be viewed as representing sequence of tokens (namely, the path from the root to this node), or more generally, a context with which to predict the next token. The state list represents all possible suffixes of the input sequence seen thus far; alternatively, it can be viewed as a list of contexts with which to predict the next token.

A node's out-count corresponds to the number of times this node has been removed from the state list and replaced with its child. The in-count of a node corresponds to the number of times the path to that node has appeared in the input sequence. For a child v of u, the ratio in-count(v)/out-count(u) is the proportion of transitions to v out of all transitions from u to its children, or equivalently, an estimate of the probability of transitioning from u to its child v.

Thus, to predict the next token from a given node u (Laird and Saul call this *projection*), TDAG returns the ratio in-count(v)/out-count(u) for all children v of u. This forms a probability distribution for the next token. The confidence in this projection increases exponentially with the out-count(u). While a projection may be made from any of the nodes on the state list, TDAG uses a heuristic function to determine from which node to project. It selects the last node on the state list with a sufficiently high out-count. Thus TDAG is selecting the node with the longest context (longest path from root to node) that still has a sufficiently high statistical confidence. An alternate implementation might use different criteria to select the node from

which to project.

The basic TDAG algorithm described here scales poorly with the size of the input sequence. In particular, the size of the state list grows linearly with the number of input tokens, and the total number of nodes may be a quadratic function of the number of input tokens. Laird and Saul describe three optimizations to the basic algorithm to make it more practical for large inputs:

1. *Bounding the node probability.* Some nodes in the graph are unlikely to be visited a large number of times, if they represent uncommon strings of tokens. The enhanced TDAG algorithm refuses to extend (add children to) a node whose probability is less than a given threshold.

2. *Bounding the height of the graph.* Define the height of a node as its distance from the root node. By refusing to extend nodes of height H (for some constant H), this enhancement essentially bounds the size of the context with which a prediction can be made.

3. *Bounding the prediction size.* In the general case, there may be an infinite number of possible input tokens. By limiting the size of the projection to at most K tokens, one ensures that the time to compute a projection is bounded. The algorithm drops any token with less than $1/K$ probability, thus ensuring that at most K tokens appear in the projection.

With these enhancements, the TDAG algorithm is practical for large input sequences and may be applied to the challenges in IUI, as discussed below.

3.2.1 Applying TDAG to IUI

TDAG is eminently suitable for addressing the problem of action prediction. The user's actions can be transformed into a sequence of tokens and fed to the TDAG algorithm, which will learn a Markov tree for the sequence. The projection made by TDAG is a probability distribution of possible next actions.

TDAG's Markov tree model may also be viewed as a model of the user's task. In particular, it represents common contiguous subsequences in the user's input. Paths through the tree that include nodes with high `out-count` and `in-count` represent common sequences of actions.

3.3 Comparison of Markov process algorithms

Both HMMs and TDAG are subject to the two Markov properties. They obey the assumption that the model does not change over time, and thus that regularities observed in the past are a good predictor of future behavior. As for the other Markov property (limited horizon), the Markov trees used in TDAG differ from the first-order Markov models described in section 3.1 in that Markov trees allow the context for prediction to vary, rather than assuming a fixed-size horizon. The state list in the TDAG algorithm is a collection of possible contexts, ranging over all the suffixes of the input sequence.

Unlike the hidden Markov models described in section 3.1, there are no algorithms to compute or compare the probability of different TDAG Markov trees given a single input sequence, thus limiting TDAG's usefulness in an architecture where multiple models are learned and chosen from at runtime, such as that proposed in section 3.1.4.

However, TDAG was designed to be usable for online applications where the algorithm receives an input token and must update its model and make a prediction in real time. For this class of application, TDAG is suitable and runs efficiently (in many cases, linear in the bounded height of the tree).

In contrast, HMMs are more expressive yet require extensive computation (reestimation of model parameters) on each new training example. As Laird and Saul note, the general HMM approach can be computationally intractable and thus is not suitable for online applications in which the model must continually be relearned for new tokens in the input sequence. In contrast, the TDAG algorithm was designed specifically for use in an online learning environment, which is a plus for responsive user interfaces.

The benefit of the Markov process approaches is that they are able to predict the next token with some probability distribution. They make the assumption that the next token is based on the tokens immediately preceding this token. The drawback to this assumption is that the algorithms deal poorly with noisy input sequences in which extraneous tokens can interrupt common sequences.

A final drawback of both general Markov models and TDAG is that they have no explicit notion of time, merely token ordering. Thus the algorithms cannot distinguish two tokens happening right after each other from two identical tokens separated by a long pause. In the IUI domain, this type of distinction might be useful, for example in performing task segmentation to decide when one task ends and another begins.

4 Discussion and Implications for IUI

We have classified the four learning algorithms investigated in this paper into two approaches: the data mining approach and the Markov approach. The first part of this section compares their very different assumptions made about the problem domain. The second part of this section suggests hybrid algorithms that may be more applicable to programming by demonstration.

4.1 Taxonomy of sequence learning problem domain

Figure 4.1 summarizes a number of key differences between the assumptions made by each algorithm. The main difference in the two approaches is the type of input each assumes. The data mining approaches assume that the input is a large database of events, each of which occurs at a particular time. The problem is to find subsequences in the database that occur more frequently than the others. Given a prefix, these common sequences can then be used to predict the next token in the sequence if the prefix matches one of the common prefixes.

	Data mining		Markov	
Feature	WINEPI	AprioriAll	HMMs	TDAG
Model of input	Patterns	Patterns	Markov model	Markov tree
Robust for noisy data?	Yes	Yes	No	No
Explicit time variable?	Yes	Yes	No	No
Model single or multiple processes?	Multiple	Multiple	Single	Single
Online or offline?	Offline	Offline	Offline	Online
Structured or atomic events?	Structured	Atomic	Atomic	Atomic
Domain knowledge useful?	Yes	Yes	Yes	No

Figure 5: Comparison of sequence learning algorithms.

The fundamental difference between the algorithms surveyed in this paper is the model each algorithm constructs of the input sequence. The two data mining algorithms model only common patterns in their input. This makes these algorithms tolerant of noise, since extraneous events in their input do not affect the

patterns that are learned. They do not attempt to learn to predict every single event in the input sequence, but instead only mine interesting nuggets of information. In contrast, the two Markov algorithms assume all tokens in their input are important, and that the next token is a probabilistic function of the tokens immediately prior to it. The models they learn capture the probability that a token will be emitted, given the sequence of tokens seen thus far.

This difference (between pattern-oriented and model-oriented approaches) results in a second difference: the pattern-oriented algorithms both have an explicit time variable, whereas the model-based approaches are only able to handle ordered sequences with no explicit time representation. Because of this explicit time variable, the pattern-oriented algorithms are more robust in the face of noisy data—sequences in which the patterns of interest are not all contiguous. This is especially important for IUI because users might generally intersperse non-task-related actions with task-related actions. An algorithm that is able to handle noisy data will be able to ignore the irrelevant actions.

In addition, the pattern-oriented algorithms do not require the input to be segmented by task. Thus they are able to model sequences that represent the output of multiple simultaneous processes. Both AprioriAll and WINEPI will learn useful patterns from a long input sequence that includes actions from several different tasks. In contrast, since TDAG and HMMs learn a model that exactly describes the input, they will do better if the input sequence is segmented by task so that they can learn different models for different tasks.

Most of the algorithms surveyed here are primarily offline algorithms. They require extensive computation to build or update their model; they have no method of incrementally updating the model when a new input token arrives. In an intelligent user interface, offline algorithms would not be fast enough to revise their model of the user in real time. In contrast, TDAG was designed from the start to be an online algorithm; TDAG updates its model quickly and incrementally. HMMs also have an incremental update algorithm, but they require more computation and are computationally intractable in an online situation.

A natural way of representing user actions in IUI is to use structured events. For example, Lau and Weld [7] describe an ontology in which actions are composed of commands and objects; objects may be represented with hierarchical structure attributes. For example, an action may be to select an email message; the message has attributes such as a date, a sender, and a subject. None of the four algorithms surveyed in

this paper deal directly with structured events, but Mannila and Toivonen [9] present an extension to the MINEPI algorithm for learning patterns over structured events.

Knowledge of the domain can be used to bias the search towards more interesting or more likely patterns. Domain knowledge is useful in three of the four algorithms. HMMs could benefit from domain knowledge in creating model structure (choosing the number of states and the connectivity between those states). Ideally the model structure would correspond to structure in the user interface domain, but the problem of mapping a user interface domain to a Markov model structure has yet to be addressed.

In addition, domain knowledge can be used to represent relationships between user actions that have similar functions, and learn patterns based on these generalized actions. Srikant and Agrawal [14] describe the GSP algorithm (an extension to AprioriAll) that matches patterns using a taxonomy of event types, rather than simple equality, to determine whether two events match in a pattern. Mannila and Toivonen [9] describe an extension to MINEPI that supports events with multiple attributes; this work could be extended to use a taxonomy in a fashion similar to GSP.

4.2 Hybrid sequence-learning algorithms for PBD

The taxonomy in the previous section identified features of each of the algorithms that make them suitable for IUI. All of the algorithms solve both the problems of action prediction and task generalization to some degree. The data mining algorithms build models of their input that represent common subsequences in the input sequence, and use these to predict the user's future actions. These models represent simple straight-line programs with no variables, loops, or conditionals. The Markov algorithms build probabilistic representations of a noise-free input sequence, and predict the next action based on the actions immediately prior to it. The Markov models represent multiple straight-line programs with many probabilistic choice points.

While all the algorithms are useful for both problems, they suffer from several limitations. In this section, we consider two hybrid algorithms that would be more useful for PBD.

4.2.1 Using frequent patterns for prediction

None of the work on frequent pattern discovery has discussed exactly how to use the discovered patterns for prediction; the association rules themselves are the interesting artifact of the discovery process. However, to apply these algorithms to IUI, we would like to use association rules to predict the next user action based on common patterns in the past. We imagine a straightforward approach: keep the list of frequent association rules, in some order (*e.g.*, from most specific to least specific using the length of a pattern as a measure of its specificity). Every time a new user action occurs, compare the last N actions of the user's action history against each of the association rules. Return the first association rule whose left hand side matches the history. This will return the most specific rule, but in the simplest case requires a linear search through the set of association rules.

The next section sketches out an alternative algorithm for using association rules for prediction.

4.2.2 Using HMMs for task generalization

HMMs may be used along with a data mining algorithm to assist in the prediction step. The drawback to HMMs is that their input must be segmented by task, in order to use the sequence to train the appropriate model. We propose an algorithm that uses a data mining algorithm to provide the necessary sequence segmentation for input to a HMM algorithm.

Given the set of association rules found by one of the data mining algorithms, cluster[1] them into similar groups of rules based on the task they are suited for. Train a HMM on each task cluster. This is very similar to the application of HMMs to speech recognition as described in Rabiner [13].

To make a prediction for a given target sequence O, calculate the probabilities $P(O|\mu)$ for each of the models μ using the Forward-Backward procedure. Pick the model $\hat{\mu}$ with the highest probability. Using the Viterbi algorithm, find the optimal sequence of states $X_1 \ldots X_T$ for model $\hat{\mu}$ given the observed sequence O. Assuming the model is now in state T at the end of this sequence, return the probability distribution for the next token based on all possible transitions out of state T.

[1]The problem of clustering user action sequences could be addressed in a paper of its own. A simple approach would be to manually cluster based on an expert user's knowledge of the user interface domain. We also imagine using automatic clustering methods based on sequence similarity metrics such as the string edit distance metric described by Wang *et al.* [16].

5 Related Work

Dietterich and Michalski [4] were among the first to consider the problem of learning from sequences. They were primarily concerned with nondeterministic mathematical sequences such as those found in the card game Eleusis, in which players try to guess the next card in a sequence by guessing the rule governing which cards may be played in sequence. An example of such a rule is "play a card one point higher or one point lower than the last card." This rule is nondeterministic in that there is no single correct next card in the sequence, but rather a set of cards (in this case, a disjunction of all the cards either one higher or one lower than the last card). Dietterich and Michalski describe a framework for discovering these kinds of rules using models and sequence transformations.

Plan recognition [6] is a related approach to the construction of intelligent user interfaces. Goodman and Litman [5] add plan recognition to a chemical process design system. Their successes included the ability to predict the user's next action by finding a common action in all possible plans projected from past actions. The major difference between plan recognition and the algorithms examined in this paper is that plan recognition requires a large library of user plans. This requirement necessitates a potentially large knowledge engineering effort. In addition, the plan recognition approach suffers from an inability to recognize novel or original plans that may not be in the plan library; in effect, a plan library constrains the possible tasks a user is able to accomplish with the aid of the system. In contrast, the approaches surveyed in this work do not require any domain-specific knowledge.

The problem of goal recognition as formulated in [8] is similar to the sequence learning algorithms surveyed in this paper. Lesh and Etzioni define a goal recognition problem as finding the set of goals that are consistent with a user's observed sequence of actions. In contrast with the data mining algorithms described in this paper, their system requires as input a set of possible goal schemas; the user is assumed to have as a goal one of the (instantiated) goals in this set.

There is much work on hidden Markov models, although none applying HMMs directly to the problems of IUI addressed in this paper. The basic HMM algorithms described by Rabiner assume a fixed model structure. One of the open problems in applying HMMs to IUI is that there is no natural mapping from entities in the domain to the set of states in the HMM, so there is no natural way to choose a model structure.

To address this problem, Stolcke and Omohundro [15] describe an algorithm for inducing the most probable model from multiple training sequences.

Research on time series is related to the problem of sequence learning. A time series is a sequence of real-valued observations that vary over time. Work in this area includes finding similarities between time series [3], which could be useful in determining the similarity of patterns in the data mining algorithms.

The MSDD algorithm [12] finds patterns in multiple streams of data, such as the outputs of a robot's sensors. MSDD finds patterns of the form "if an instance of pattern x begins in the streams at time t, then an instance of pattern y will begin at time $t + \delta$ with probability p." MSDD is similar to WINEPI in that it finds patterns within a constant sized window of time; however, it finds patterns over multiple input sequences rather than WINEPI's single input sequence. MSDD's multiple input streams are similar to AprioriAll's customer-sequences, except that MSDD's streams are assumed to be causally related whereas AprioriAll's customer-sequences are independent. MSDD finds patterns spanning multiple streams while AprioriAll finds patterns confined to a single stream.

The SEQUITUR system infers sequential structure in a single long sequence. SEQUITUR finds a hierarchical representation of the original sequence that may be used to recreate the sequence exactly. SEQUITUR's operation is driven by two constraints: digram uniqueness (no pair of adjacent symbols appears more than once in the grammar) and rule utility (each nonterminal rule must be used at least twice). SEQUITUR is most related to the TDAG algorithm in that it finds regularities in a sequence based on the adjacency of tokens. Whereas TDAG generalizes from the input sequence to a probabilistic model that is able to generate other sequences similar to the input sequence, SEQUITUR performs no generalization. However, SEQUITUR abstracts contiguous subsequences into single nonterminals that represent multiple tokens in the input sequence, and thus could potentially useful for programming by demonstration.

6 Conclusions

We have proposed two challenges for intelligent user interfaces (and programming by demonstration in particular): action prediction and task generalization. Both of these problems are fundamentally sequence learning problems.

In light of these two challenges, we surveyed four algorithms for sequence learning. The WINEPI and AprioriAll data mining algorithms find frequent patterns in a large database of input sequences. Hidden Markov models and TDAG build a statistical model of the input sequence. We generalized WINEPI and AprioriAll into a common framework and described them both in terms of this framework.

We compared these four algorithms in terms of a taxonomy of the problem space. The main distinction is the type of model each algorithm builds of its input. WINEPI and AprioriAll extract the common patterns in a large sequence of noisy data. HMMs and TDAG build a probabilistic model of an input sequence that is assumed to come from a single noise-free process.

None of the algorithms are perfectly suited to solve the PBD challenges of action prediction or task generalization. We have described two hybrid approaches that combine elements of each of the algorithms to create algorithms that are potentially more suitable for PBD.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Mining Sequential Patterns. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Taipei, Taiwan, March 1995.

[2] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.

[3] Gautam Das, Dimitrios Gunopulos, and Heikki Mannila. Finding Similar Time Series. In *Proceedings of the First European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD 97)*, pages 88–100, Trondheim, Norway, June 1997.

[4] Thomas G. Dietterich and Ryszard S. Michalski. Learning to Predict Sequences. In Michalski et al, editor, *Machine Learning, Volume II*, pages 63–106. Morgan Kaufman, Los Altos, CA, 1986.

[5] Bradley A. Goodman and Diane J. Litman. Plan Recognition for Intelligent Interfaces. In *Proceedings of the Sixth Conference on Artificial Intelligence Applications (CAIA)*, pages 297–303, Santa Barbara, CA, USA, March 1990.

[6] H. Kautz. *A Formal Theory Of Plan Recognition.* PhD thesis, University of Rochester, 1987.

[7] Tessa Lau and Daniel S. Weld. Programming by Demonstration: an Inductive Learning Formulation. In *Proceedings of the 1999 International Conference on Intelligent User Interfaces (IUI 99)*, pages 145–152, Redondo Beach, CA, USA, January 1999.

[8] Neal Lesh and Oren Etzioni. A Sound and Fast Goal Recognizer. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 95)*, 1995.

[9] Heikki Mannila and Hannu Toivonen. Discovering Generalized Episodes Using Minimal Occurences. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, USA, August 1996.

[10] Heikki Mannila, Hannu Toivonen, and Inkeri A. Verkamo. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, November 1997.

[11] Christopher Manning and Hinrich Schütze. Markov models. In *Foundations of Statistical Natural Language Processing*, chapter 9, pages 293–315. MIT Press, Cambridge, Massachusetts, USA, 1999. Draft.

[12] Tim Oates and Paul R. Cohen. Searching for Structure in Multiple Streams of Data. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 346–354, 1996.

[13] Lawrence R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–285, February 1989.

[14] Ramakrishnan Srikant and Rakesh Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT 96)*, pages 3–17, Avignon, France, March 1996.

[15] Andreas Stolcke and Stephen Omohundro. Hidden Markov Model Induction by Bayesian Model Merging. *Advances in Neural Information Processing Systems*, 5, 1993.

[16] Jason Tsong-Li Wang, Gung-Wei Chirn, Thomas G. Marr, Bruce Shapiro, Dennis Shasha, and Kaizhong Zhang. Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 115–125, Minneapolis, MN, USA, 1994.