7

Activity Prediction

In addition to discovering and recognizing activities from sensor data, activity learning includes predicting activities. Activity recognition considers past or current sensor data in order to map the data to an activity label. In contrast, activity prediction (AP) algorithms hypothesize information about activities that will occur in the future. Two specific prediction questions are addressed in this chapter:

1. Given the sequence of events (or activities) that has been observed up to the current point in time, can we predict what event (or activity) will occur next (*sequence prediction*)?
2. For any given activity, can we predict when in the future it will occur again (*activity forecasting*)?

Activity predictions are valuable for providing activity-aware services in the home or using mobile phones, tablets, phablets, and other devices. For example, anticipating a resident's activities facilitates home automation. An activity-aware sensor network can use predictions of upcoming activities to automate the home in preparation for the activities (e.g., warm up the house before the resident returns home). By combining AP with activity recognition, AP can also be used to provide automated prompting. If the time window when the activity normally occurs passes and the activity recognizer did not detect that the activity was performed, a prompt can be issued and repeated until the user interacts with the environment or the activity has been performed by the user and detected by the activity recognizer.

In this chapter, we will introduce a compression-based approach to perform sequence prediction. Moreover, we will consider three alternative methods for predicting the timing of activity occurrences, which include time series-based

Activity Learning: Discovering, Recognizing, and Predicting Human Behavior from Sensor Data, First Edition. Diane J. Cook and Narayanan C. Krishnan.

activity forecasting, probabilistic graph-based activity prediction, and induction of activity timing rules.

7.1 ACTIVITY SEQUENCE PREDICTION

Activity prediction is the cornerstone of activity-aware service algorithms. Part of this problem is sequential prediction, or using an observed sequence of events to predict the next event to occur. Symbolic sequence prediction is based on information theoretic ideas for designing lossless compression strategies. While we illustrate the algorithm using an example sequence of sensor events, the algorithm can be used to predict any symbol from a fixed vocabulary, including predicting navigation trails, locations within a home, or upcoming activities. Sequence prediction has also been applied to challenges outside the realm of activity learning, including biological sequence analysis, speech modeling, text analysis, and music generation. For any sequence of symbols that can be modeled as a stochastic process, the sequence prediction algorithm we describe here will use variable-order Markov models to optimally predict the next symbol.

Consider a sequence of events being generated by an arbitrary process, 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 in the sequence, x_{i+1}? Well-investigated text compression methods have established that good compression algorithms are also good predictors. In particular, a prediction model with an order that grows at a rate approximating the source's entropy rate is an optimal predictor. Text compression algorithms commonly make use of an incremental parsing approach to processing information. This incremental parsing feature is also useful for online predictors and forms the basis of one approach to sequence prediction.

Consider a stochastic sequence $x_1^n = x_1, x_2, \ldots, x_n$. At time t, the predictor will output the next likely symbol x_t based on the observed history, or the sequence of input symbols $x_1^{t-1} = x_1, x_2, .., x_{t-1}$, while minimizing the prediction errors over the entire sequence. Theoretically, an optimal predictor must belong to the set of all possible finite state machines and it has been shown that Markov predictors perform as well as any finite state machine. Markov predictors maintain a set of relative frequency counts for the symbols that are observed at different contexts in the sequence, thereby extracting the sequence's inherent pattern. Markov predictors 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. As a result, 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 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.

The LZ78 data compression algorithm is an incremental text parsing algorithm that introduces such a method for gradually changing the Markov order at the appropriate rate. This algorithm is a modeling scheme that sequentially calculates empirical

```
Algorithm LZ78StringParsing(x)
  // x is sequence of observed symbols x_1, .., x_N
  // Initialization
  w = null
  Dictionary = null
  i = 1
  while i <= N
    v = Append(w, x_i)
    if v ∈ Dictionary
      w = v
    else
      Insert v in Dictionary
      w = null
      Increment the frequency for every prefix in Dictionary of phrase v
    end if
    i = i + 1
  done
return
```

FIGURE 7.1 *LZ78 string parsing algorithm.*

probabilities for each subsequence, or phrase, of the data, with the added advantage that the generated probabilities reflect phrase contexts observed from the beginning of the parsed sequence to the current symbol.

LZ78 creates and maintains a dictionary of substrings to use for compression. The algorithm parses an input string $x_1, x_2, \ldots, x_N$ into $c(N)$ substrings $w_1, w_2, \ldots, w_{c(N)}$ such that for all $j > 0$, the prefix of the substring w_j (all but the last character of w_j) is equal to some w_i for $1 < i < j$. This prefix property ensures that the dictionary of parsed substrings and their relative frequency counts can be maintained efficiently in a multiway tree structure called a trie.

As LZ78 is a compression algorithm, it consists of both an encoder and a decoder. When we are performing prediction, we do not need to reconstruct the parsed sequence so we do not rely on encoding or decoding phrases. Instead, we must simply construct an algorithm that breaks the input sequence (string) of events or states into phrases. The algorithm for parsing and processing an input symbol sequence is shown in Figure 7.1.

Example 7.1 Consider a string, $x^n = aaababbbbbaabccddcbaaaa$, that represents a sequence of locations visited by a resident in a home, where $a =$ Kitchen, $b =$ LivingRoom, $c =$ Bathroom, and $d =$ Bedroom. The algorithm can be applied to this sequence to determine the most likely location that will be visited next. An LZ78 parsing of this string would create a trie as shown in Figure 7.2 and would yield the phrases *a*, *aa*, *b*, *ab*, *bb*, *bba*, *abc*, *c*, *d*, *dc*, *ba*, and *aaa*. The parsing algorithm maintains frequency counts for all prefixes, or 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 (at the beginning of

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

The drawback of LZ78 is its slow convergence rate to optimal predictability. This means that the algorithm must process numerous input symbols to reliably perform sequential prediction. However, LZ78 does not exploit all of the available information. For example, context information that crosses phrase boundaries is lost. Using 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 without looking back. As the number of symbols in an input sequence grows, the amount of information that is lost across phrase boundaries increases rapidly.

This slow convergence rate can be partially addressed by keeping track of all possible contexts within a given phrase. In order to recapture information lost across phrase boundaries, an alternative sliding window approach can be considered. The Active LeZi, or ALZ, approach to sequence prediction maintains a variable-length window of previously seen symbols. Throughout the ALZ algorithm, the length of the window is set to k, the length of the longest phrase that has been parsed so far by LZ78. As a result, this enhanced algorithm allows LZ78 to construct an approximation to an order $k - 1$ Markov model. Our example trie shown in Figure 7.2 has a depth of three, which corresponds to the length of the longest phrase and indicates a Markov order of two.

Within the sliding window ALZ gathers statistics on all possible contexts, as shown in Figure 7.3. By capturing information about contexts that would cross

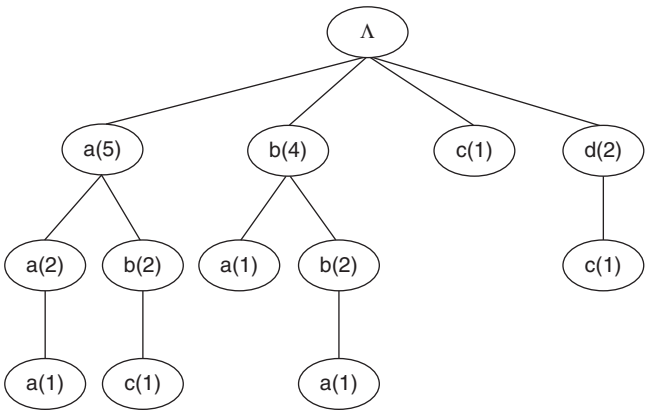

FIGURE 7.2 *The trie formed by the LZ78 parsing of the location sequence aaababbbbbaabccddcbaaaa. The numbers in parentheses indicate the frequency of the corresponding phrase encountered within the sequence as it is parsed.*

```
Algorithm ALZStringParsing(x)
  // x is sequence of observed symbols x_1, .., x_N

  // Initialization
  w = null
  Dictionary = null
  window = null
  MaxLength = 0

  i = 1
  while i <= N
    v = Append(w, x_i)
    if v ∈ Dictionary
      w = v
    else
      Insert v in Dictionary
      if (Length(v) > MaxLength)
        MaxLength = Length(v)
      end if
      w = null
    end if

    Add v to window
    if (Length(window) > MaxLength)
      Delete the first symbol in window
    end if

    // Increment frequency for related contexts
    Increment the frequency for every prefix in Dictionary that includes v
    i = i + 1
  done
return
```

FIGURE 7.3 *ALZ string parsing algorithm.*

phrase boundaries in the original LZ78, we can build a closer approximation to an order- k Markov model and converge more quickly to optimal predictability.

Example 7.2 Figure 7.4 shows the trie that is formed by an ALZ parsing of the input sequence *aaababbbbbaabccddcbaaaa*. This is a more complete model than the one shown in Figure 7.2 and it represents an order-MaxLength-1 Markov model.

ALZ's trie and corresponding frequency counts are managed at the same time as the input sequence is parsed. The more complex case occurs when the input sequence both rapidly increases maximum phrase length (and consequently the ALZ window size) and yet grows slowly enough that the most possible 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, the most complex case occurs when each new LZ phrase is one symbol longer than the previous phrase. In the worst case, each subphrase in the ALZ window that includes the subsequent symbols adds a new node to the trie. This means that at order k, the trie increases by k^2 nodes before the model

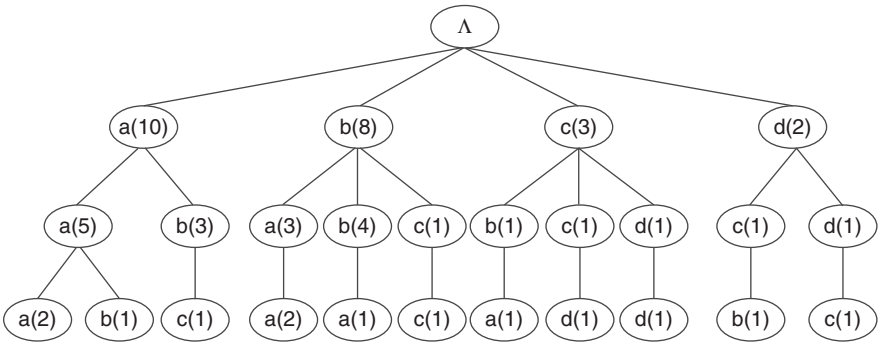

FIGURE 7.4 *The trie formed by the ALZ parsing of the location sequence* aaababbbbb aabccddcbaaaa.

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 $k = O(\sqrt{n})$. In practice, however, the order tends to grow much more slowly because the worst case is interspersed with intervals of shorter phrases. The limiting case brings the space requirement to $O(n)$ and the time complexity to $O(n^{3/2})$.

Ultimately, we want to use the model to generate a probability distribution over the possible symbols that will occur next in the sequence. Given the distribution the sequence prediction algorithm outputs the symbol with the highest probability as the most likely event (or activity, or location, or state) to occur next.

To achieve better rates of convergence 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 (i.e., phrase sizes) to assign a probability estimate to the next symbol. Multiple order models can be incorporated using the Prediction by Partial Match (PPM) family of predictors. PPM algorithms generate and fuse Markov models from multiple orders to build a probability distribution. As described earlier, ALZ builds an order- k Markov model. The PPM strategy then gathers information from Markov models of orders 1 through k to assign each possible next symbol its corresponding probability value.

Example 7.3 ALZ's window represents the set of contexts that can be used 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 (i.e., *a*, *aa*, and the null context).

Consider the case of computing the probability that the next symbol is *a*. We compute the probability separately using each available context order and combine the results to generate the prediction probability for a particular symbol. The trie in Figure 7.4 has a depth of 3 so for any particular prediction task we can consider contexts of order 0, 1, and 2. Following the leftmost path in Figure 7.4, we see that the symbol *a* occurs two out of the five times that the context phrase *aa* appears, the other

cases producing two null (end of phrase) 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 the symbol 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 the symbol 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 the symbol a 10 times out of the 23 symbols processed so far, so we predict a with probability 10 in 23 at the order-0 context. As a consequence, we compute the combined probability of seeing a, representing the Kitchen location in our example, as the next symbol as

$$\frac{2}{5} + \frac{2}{5}\left\{\frac{5}{10} + \frac{2}{10}\left(\frac{10}{23}\right)\right\} = 0.635.$$

We make a few observations about this approach for assigning probabilities. First, we note that it solves the zero-frequency problem. In the earlier example, if we chose only the longest context to calculate probabilities, it would have returned a zero probability for the symbol c, whereas lower-order models reveal that the probability should be non-zero. Second, the blending strategy assigns greater weight to higher-order models when calculating probabilities if it finds the symbol being considered in that context, while it suppresses the lower-order models owing to the null context escape probability. This strategy is consistent with the advisability of making the most informed decision.

The blending strategy considers nodes at every level in the trie from order 1 through $k - 1$ for an order-k Markov model, which results in a worst-case run time of $O(k^2)$. The prediction run time is thus bounded by $O(n)$. This assumes that a fixed-sized alphabet is used to represent what is being predicted (sensor events, ranges of sensor values, locations, states, or activities), which implies that the prediction algorithm can search for the child of any node in the trie in constant time. By efficiently maintaining pointers only to nodes in the tree that are likely to be expanded, the algorithm can be further improved to perform predictions in $O(k) = O(\sqrt{n})$ time. Additional information can be stored in the trie along with frequencies as needed. For example, elapsed time between observed symbols can be stored to predict not only the next event that will occur but also how much time will pass before the event occurs.

7.2 ACTIVITY FORECASTING

We now consider the task of predicting at what time in the future a particular activity will occur. To accomplish this task, we can borrow ideas from time series analysis. A time series is a sequence of random variable values over time, $x_1, x_2, x_3, \ldots, x_t$. Forecasting has been used in time series analysis to predict future values of a target variable given observed past and current values. Given the observed sequence through

time t, the forecaster will output values x_{t+1}, x_{t+2}, x_{t+3}, ... In the case of activity forecasting, the random variable of interest is X_a, the amount of time that will elapse until the next occurrence of activity $a \in A$. Given labeled training data, the values of X_a for past points in time can be calculated.

There are two assumptions that are typically made when performing forecasting. The first is that the data have a natural temporal ordering. The second is that the forecasted variable has numeric values. In the case of AP, both of these assumptions hold. We assume that the input to the prediction problem is a time-ordered sequence of sensor events in which the temporal ordering is enforced. The output of the prediction algorithm is an estimated number of time units until the targeted activity will occur, so the output can be represented using a numeric variable.

As with activity recognition, historic training data must be available in which each sensor event is tagged with the number of time units that elapsed between the sensor event and the next occurrence of the target activity.

Example 7.4 As an example, consider the labeled data shown in Figure 5.4. The values of variable $X_{\text{EnterHome}}$, the amount of time that will elapse until the next occurrence of the Enter Home activity, for each of these sensor events is shown in Figure 7.5. This example shows a forecasted value for each sensor event. Most time series analysis techniques assume that the observations are made at equal-length time periods. The forecasted values can be interpolated to provide values at equal intervals, as needed.

When performing activity prediction, the output of the forecast algorithm is the time that will elapse until the next occurrence of the targeted activity. Standard forecasting techniques can be used for this task. These include calculating a moving average or average value within a sliding window over the data. The moving average can be combined with an autoregressive function that generates a forecasted value based on a weighted function of past observed values, called an autoregressive

2009-07-19	10:18:59.406001	LivingRoom	ON	6060.594
2009-07-19	10:19:00.406001	Bathroom	OFF	6059.594
2009-07-19	10:19:03.015001	OtherRoom	OFF	6056.985
2009-07-19	10:19:03.703001	LivingRoom	OFF	6056.297
2009-07-19	10:19:07.984001	LivingRoom	ON	6052.016
2009-07-19	10:19:11.921001	LivingRoom	OFF	6048.079
2009-07-19	10:19:13.203001	OtherRoom	ON	6046.797
2009-07-19	10:19:14.609001	Kitchen	ON	6045.391
2009-07-19	10:19:17.890001	OtherRoom	OFF	6042.110
2009-07-19	10:19:18.890001	Kitchen	OFF	6041.110
2009-07-19	10:19:24.781001	FrontMotion	ON	6035.219
2009-07-19	10:19:28.796001	FrontMotion	OFF	6031.204
2009-07-19	10:19:31.109001	FrontDoor	CLOSE	6028.891
2009-07-19	12:05:13.296001	FrontDoor	OPEN	0.000

FIGURE 7.5 *Forecasted time elapsed values for Enter Home activity.*

moving average, or ARMA, model. In the case where the underlying process that generates the values changes over time, the function can be applied to value differences over time. The learned function can be integrated to map it back to the original data, resulting in an autoregressive integrated moving average, or ARIMA, approach.

One limitation of these standard techniques is that the function cannot always adequately model complex processes with nonlinear interactions between the features, as often occurs when learning activity models. Forecasting the numeric activity timing variables can thus be framed as a supervised learning problem where the classifier maps a feature vector onto the corresponding time unit value. As a result, any machine learning algorithm, such as regression, which maps input features onto continuous-valued output can be used. Such techniques include backpropagation and recurrent neural networks and support vector machines, described in Chapter 4. However, in addition to the feature values (described in Chapter 3) that represent the state at the current time t_i, additional features must be included. These features are called the "lag" values and represent the value of the output, or class variable, at previous time units $t_{i-L}, \ldots, t_{i-1}$. In the case of activity prediction, the lag values represent the forecasted activity elapsed time for the L past sensor events, where L represents the maximum lag.

A supervised learning algorithm must thus be selected that can handle numeric or discrete input values and that maps the values onto an output numeric value. Because AP must be updated after each sensor event, an algorithm should be selected that can update its model without a large computational cost. The SVM learning algorithm described in Chapter 4 is appropriate for such a problem. Another algorithm that fits these constraints with a lower computational cost is a regression tree. Here we describe how a regression tree is learned and then explain how it can be used as the backbone of an AP algorithm. Since this is a forecasting-based AP algorithm, we will refer to it as FAP.

Regression trees, like decision trees, are created in a top-down method. Moreover, like traditional decision trees, the tree is traversed from the root node to a leaf node. Each internal node queries one feature of the data point and a path is traversed based on the outcome of the query. When a leaf node is reached in a regression tree, information contained in the leaf node is used to generate a value for the data point.

Example 7.5 Figure 7.6 shows a sample regression tree that was automatically constructed from sensor data representing the predicted number of seconds that will elapse until the next occurrence of a Bed Toilet Transition activity.

While the structure of a regression tree is similar to a decision tree, the split criteria at each internal node and the value computation at the leaf nodes are different from a traditional decision tree. Most decision tree algorithms choose an attribute at each split point that maximizes information gain, as described in Chapter 4. In contrast, a regression tree selects an attribute based on maximizing the reduction in error. For a regression tree, the standard deviation in class values that result from splitting the dataset for a particular attribute is an estimate of the error for the corresponding

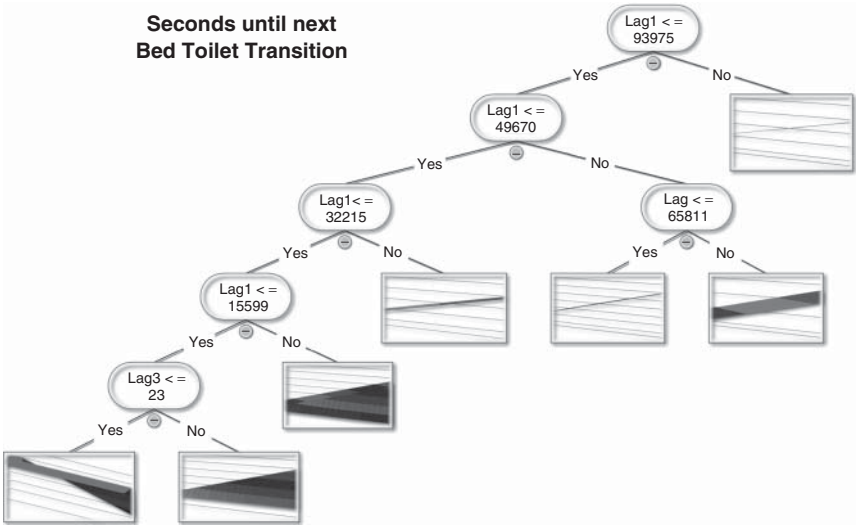

FIGURE 7.6 *A regression tree to output expected number of seconds until the next occurrence of a Bed Toilet Transition activity. Each internal node queries the value of a particular variable and each node contains a multivariate linear regression function. Each of the regression functions generates a prediction value as a function of the lags and other input feature values. The features include lag variables, or values of the predicted variable at previous time units. Lag1 indicates the prediction value at time t – 1 and Lag3 indicates the prediction value at time t – 3.*

subset of data points. Letting T represent the set of training examples, or labeled sensor events, the regression tree selects an attribute that maximizes gain as defined in Equation 7.1.

$$\text{Gain}(T, A) = \sigma(T) - \sum_{v \in \text{Values}(A)} \frac{|T_v|}{|T|} \times \sigma(T_v) \tag{7.1}$$

The attribute that maximizes gain (reduces the expected error) is chosen for querying and splitting at the internal node. Splitting terminates when all of the attributes have appeared along the current path or the error for the remaining subset of data points (the standard deviation of class values for the data points) is below a threshold fraction of the error in the original training dataset. Unlike traditional decision trees, each leaf node represents a linear regression of the class values for the data points that are contained in that leaf node. The multivariate linear model is constructed using linear regression. The model takes the form $w_0 + w_1a_1 + w_2a_2 + \ldots + w_ka_k$, where $a_1, \ a_2, \ldots, a_k$ are attribute values and the weights $w_1, \ w_2, \ldots, w_k$ are calculated using standard regression. Instead of using all of the attributes in the linear model, only those that are queried along the path to the leaf node are included in the linear model.

The model is then updated by removing as many attributes (variables) as possible. Using a greedy search, the attributes are removed if the removal does not affect the accuracy of the model. While most of the features described in Chapter 3 are numeric

and thus easily fit into this model, discrete attributes with v possible values can be incorporated by transforming them into a set of $v - 1$ binary attributes.

Like traditional decision trees, regression trees can be pruned to improve overall predictive performance. In order to perform pruning, linear models must be built for each internal node as well as each leaf. The regressed model takes the form $w_0 + w_1a_1 + w_2a_2 + \ldots + w_ka_k$, where $a_1, a_2, \ldots, a_k$ only represent values for attributes found below the current node. Once the tree is fully constructed, pruning is performed in a bottom-up manner along each path in the tree and the linear models at each node can be used to compare the error that would result from turning the internal node into a leaf with the error resulting from leaving the branch untouched.

When a new data point is processed, the tree is traversed from the root down to the appropriate leaf node. The resulting linear model at the leaf node can be evaluated to yield the prediction value. However, the value can be improved through a smoothing process, which is particularly valuable if the leaf node was constructed from only a few data points. Smoothing combines the leaf's model with the model for each internal node along the path from the leaf node to the root. At each internal node S, the corresponding class value, $C(S)$, is calculated as shown in Equation 7.2.

$$C(S) = \frac{n_i \times C(S_i) + k \times M(S)}{n_i + k} \tag{7.2}$$

where S_i and n_i refer to the node's i^{th} branch that was traversed for this data point and the number of training points that followed that branch, respectively. $M(S)$ represents the value that is calculated by the linear model at node S and k is a smoothing constant.

In order to use a regression tree as an activity forecaster, additional L features must be added to the feature vector. These represent the L lag values, or previous class values. Other features, called time indexes, are also generated. The time index provides an indication of where the data point falls along the timeline from the beginning of the data sequence to the current point in time. This is useful if the data points do not occur at equal time intervals, as was the case for the data shown in Figure 5.4. This timestamp feature can be calculated as shown in Equation 7.3.

$$\text{index}(e_i) = \frac{\text{time}\,(e_i) - \text{time}\,(e_1)}{\text{time}\,(e_n) - \text{time}\,(e_1)} \tag{7.3}$$

where n represents the number of data points in the entire sequence being processed and time (e_i) is a numeric representation of the time at which sensor event i occurred.

7.3 PROBABILISTIC GRAPH-BASED ACTIVITY PREDICTION

As we discussed in Chapter 4, probabilistic graphs such as naive Bayes graphs, hidden Markov models (HMMs), and conditional random fields can be used to calculate a probability distribution over activity labels given observed information such as sensor events. They can also be used for sequential AP by maintaining and updating a

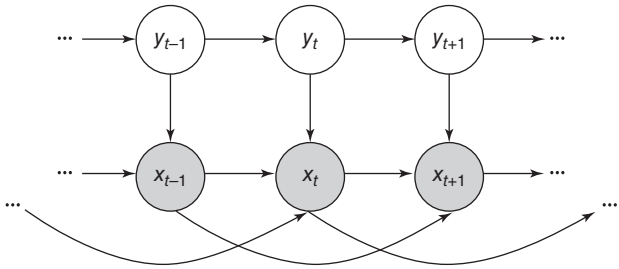

FIGURE 7.7 *An order 2 AR-HMM.*

probability distribution over the labels of activities and their corresponding likelihood of occurring next in the sequence.

As is shown in Figure 4.3, a HMM is a specific type of probabilistic graph that is well suited to incorporating temporal relationships between activities as well as evidence relationships between activities and sensor readings in its calculation of probability distributions. Note that in using an HMM, the joint probability distribution calculation shown in Equation 4.4.1 indicates that the activity transition probabilities $P(y_t|y_{t-1})$ are conditioned only on the previous activity. In addition, the activity evidence distributions $P(x_t|y_t)$ are conditioned only on the current activity. As a result, these traditional HMMs may not adequately capture long-term correlations between sensor events. This could affect the performance of activity prediction, particularly when activities are interwoven or performed in parallel.

For activity prediction, we consider a variation of an HMM that is known as an autoregressive HMM (AR-HMM). An autoregressive model considers the influence of past sensor events on the current sensor events and values, and the order (number of past events considered) is generally fixed. Figure 7.7 shows an example HMM with order 2. If all of the model variables had full observability, the conditional probability table (CPT) values could be estimated from sample data. However, in the AR-HMM some of the variables are not directly observed. These are the hidden variables y_i that represent the activity classes. As a result, we can use an expectation maximization (EM) algorithm to find a locally optimal maximum likelihood estimate of the parameters.

The idea behind EM is that if we know values of the variables for each node in the HMM learning the CPTs would be easy. As a result, in the E step we compute the expected value of each node using an inference algorithm and then treat the expected values as though the corresponding distribution was actually observed. In the M step, we maximize the parameter likelihood from the corresponding expected counts.

Example 7.6 Using our running example, if we consider the value of y_t as Eat and the value of y_{t-1} as Sleep, we can compute

$$P(y_t = \text{Eat}|y_{t-1} = \text{Sleep}) = \frac{EN(y_t = \text{Eat}, y_{t-1} = \text{Sleep})}{EN(y_{t-1} = \text{Sleep})},$$

where $EN(y)$ represents the expected number of times activity y occurs in the dataset given the current guess for the parameter values. Given the expected activity counts, we maximize the parameters and re-compute the expected counts in a repetitive process.

A similar process is applied to estimate the emission probabilities. Continuing our example, we can compute the probability that x_t refers to a particular sensor "M02" being fired at time t as

$$P(x_t = M02|y_t = Eat, x_{t-1} = M07) = \frac{EN(x_t = M02, y_t = Eat, x_{t-1} = M07)}{EN(y_t = Eat, x_{t-1} = M07)}.$$

As seen in Figure 7.7, each observable state in an AR-HMM has extra links from earlier observations. As a result, we need to calculate observation transition probabilities in the same way HMMs calculate transition probabilities for hidden states. The learning process thus requires the update of three sets of probabilities throughout the network: the hidden state transition probabilities $P(y_t|y_{t-1})$, the observation state transition probabilities $P(x_t|x_{t-1})$, and the emission probabilities $P(x_t|y_t, \ x_{t-1})$.

In this discussion, we have used HMMs and AR-HMMs to perform sequential prediction. However, HMMs can be used to model time as well. As a result, they can predict not only the next activity (or location, or sensor event) that will occur but also how many time units into the future it will occur. This is known as HMMs with explicit state duration. In these graphs, the probability of transitioning from a state to itself is modeled by a separate duration distribution. The likelihood of an activity occurring at a given time point in the future is thus dependent upon the current and previous activities, the current and previous observed sensor events, and the probability of the current activity having a duration that is one time unit greater than the current amount of time that has already elapsed for the activity.

7.4 RULE-BASED ACTIVITY TIMING PREDICTION

One approach for predicting the timing of activities is to learn logical rules that describe when an activity is typically initiated as a function of other events that are automatically detected from sensor events and values. Activities that are part of an individual's regular routine are usually initiated based on wall clock time or based on activity context.

Example 7.7 As an example of the first pattern, one activity of interest may be Pick Up Grandchildren From School, which occurs every Tuesday afternoon at 3:00pm. Another activity to track might be Taking Medicine while eating breakfast, which is an example of the second type of pattern.

A rule-based activity predictor, or RAP, learns patterns for each activity, *PA*, as a function of another reference activity, *RA*, with which it is correlated. RAP models

the relative time offset between initiation of *RA* and *PA* as a Gaussian distribution with a corresponding mean and standard deviation. The specific format for an activity pattern is thus:

<activity> [<relative_activity> <mean (s)> <standard_deviation (s)>]+

where the "+" means one or more relative activities.

Possible relative activities for *PA* include all other activities the individual performs, combined with periodic clock-based activities. The clock-based activities include the start of each year, month, day, week, and hour, as well as the start of each specific month of the year (January, February, … , December), each specific day of the week (Sunday, … , Saturday), and calendar-based events of interest (birthdays, holidays). This way, activity timings can be learned both for activities that occur at regular times and activities whose occurrence is relative to another activity. Each activity timing rule is represented by the name of the prompted activity *PA*, the name of the relative activity *RA*, the mean time delay in seconds between *RA* and *PA*, and the time standard deviation in seconds.

Example 7.8 If the Pick Up Grandchildren activity takes place every Tuesday around 2:40pm (+/−5 minutes), the associated prediction rule would be represented as:

Pick_up_grandchildren Activity_Tuesday 52800 300

If the individual picked up their grandchildren every Tuesday and Thursday around 2:40pm, then the rule would be:

Pick_up_grandchildren Activity_Tuesday 52800 300 *Activity_Thursday* 52800 300

On the other hand, if the individual takes medicine about ten minutes (+/−5 minutes) after breakfast begins each morning, then the corresponding rule would be:

Take_Medicine Eat_Breakfast 600 300

For each activity *PA*, RAP can learn a timing rule using a two-step process: consider timing patterns that are based on a single relative activity, and then consider timing patterns that are based on multiple relative activities. All of these possibilities are evaluated and the highest-ranked pattern is chosen for the prediction rule. First, consider the method for evaluating patterns based on a single relative activity. RAP must select a relative activity other than *PA* from among the activities the individual performs, along with the clock-based activities described earlier. An ideal relative activity *RA* is one that always occurs before each instance of the activity *PA* and always at the same (ideally small) time before *PA*. Therefore, the score for a relative activity *RA* should increase proportional to the number of times it co-occurs with *PA*, should decrease proportional to the variance in the time delay between each *RA* and

PA, and should decrease proportional to the absolute time delay between each *RA* and *PA*. Each potential relative activity *RA* is thus evaluated according to three properties: (i) The likelihood that activity *PA* occurs after each activity *RA*, (ii) The confidence in the distribution of the occurrence times of *PA* relative to *RA*, and (iii) The mean delay between *RA* and *PA*.

Property 1 is essentially the probability that *RA* occurs before each instance of *PA*. We estimate this probability from sample data. Given m instances of relative activity *RA* in the sensor data and n instances of activity *PA* occurring between two consecutive *RA*s, we estimate the occurrence likelihood as n/m. This forms the first factor of our overall rule score P, shown in Equation 7.4, for *RA* as the relative activity for *PA*. Property 2 measures the variance in the delay between the two activities. Again, we want minimal variance, so this factor will be in the denominator of Equation 7.4. There are two contributions to the variance in the delays. The first contribution is the actual variance in the distribution of the delays between each co-occurrence of *RA* and *PA*. For all such occurrences of *PA* preceded by *RA*, the rule-based learner models the time delay between the two activities as a Gaussian and computes the corresponding mean μ and standard deviation σ for these delays. The standard error $\sigma/\sqrt{n}$ can be used as an estimate of the confidence (smaller the better) that *PA* follows μ time units after *RA*. This comprises the second factor in P below, which decreases P based on increased distribution error. The second contribution to the delay error involves the $(m - n)$ occurrences of *RA* that are not followed by an occurrence of *PA*. This contribution to the distribution error can be estimated as the standard error based on a variance of one and a sample size of $(m - n)$. This comprises the third factor in P below, which decreases P based on increased distribution error due to the absence of a *PA* after *RA*. Property 3 prefers a smaller mean delay time μ. Therefore, the fourth factor $1/\sqrt{\mu}$ is included in P below, which decreases P as the mean delay increases. Combining all these factors, we arrive at the following predictability measure P shown in Equation 7.4.

$$P = \left(\frac{n}{m}\right)\left(\frac{1}{\sigma/\sqrt{n}}\right)\left(\frac{1}{\sqrt{m-n}}\right)\left(\frac{1}{\sqrt{\mu}}\right) \tag{7.4}$$

This predictability measure estimates the correlation between the two activities and thus represents how well *RA* plays the role of the relative activity for *PA*. If $m = 0$ or $n = 0$, we set $P = 0$. If $\sigma = 0$, we set $\sigma = 1$. If $\mu = 0$, we set $\mu = 1$. If $(m - n) = 0$, we set $(m - n) = 1$. The relative activity with the highest P value, along with its associated mean and standard deviation, are output as the activity's timing rule. If two relative activities have the same P value, we prefer the one with the smaller mean.

The second step in the process is to consider rules where the prompt activity *PA* occurs relative to several other relative activities, not just one. While timing patterns can be learned as a function of multiple relative activities, considering all subsets of activities as potential patterns is not computationally tractable. However, RAP can consider patterns that involve subsets of selected clock or calendar-based events such as the months of the year (January, February, … , December), the days of the week (Sunday, Monday, … , Saturday), and the hours of the day, since many activities

occur relative to specific sets of months, days or hours (e.g., leaving for work at 7am Monday through Friday). Additional relative activities can be included for this, such as month-of-year, day-of-week, and hour-of-day, where their predictability P values are computed as the sum of the above-average P values of each individual month, day, or hour within the set. If one of these multiple relative activity patterns wins out over all the others, then the output pattern consists of all the individual month, day, or hour relative activities whose frequency is in the upper half of the range of normalized frequencies.

Example 7.9 Using our example of leaving for work at 7am Monday through Friday, the day-of-week relative activity would be considered by summing the above-average P values for each individual day of the week. RAP would detect that the frequencies for Sunday and Saturday are low, and these days are thus not included. Therefore, the final activity timing pattern would look as follows (assuming 7am $+/-$ 15 minutes):

Leave_for_Work *Activity_Monday* 25200 900 *Activity_Tuesday* 25200 900
Activity_Wednesday 25200 900 *Activity_Thursday* 25200 900
Activity_Friday 25200 900

A feature of the rule-based approach is that the expected timing of an activity is learned together with the typical timing variance. If the prediction rules are used in the context of an application such as activity prompting or home automation, for example, the application software will need to monitor the current time, the activity context as determined by an activity recognizer, and the activity timing rules that were learned. When an action such as turning on a device or issue a prompt is warranted, it can be sent to the appropriate device, such as a touch-screen computer located in the home or a mobile device.

7.5 MEASURING PERFORMANCE

We summarize some methods for prediction evaluation here.

Leave n-at-the-End-Out Testing Many of the performance measures that were introduced in Chapter 5 can be used to evaluate the performance of sequential prediction algorithms as well. One important distinction, however, is the fact that unlike many standard machine learning datasets, the data points used to train and test sequential prediction algorithms are not independent. As a result, the evaluation method cannot randomly select data points to hold out for testing because the held out data points are part of a sequence and therefore play a role in the definition of other data points as well. Instead, data at the end of a sequence can be held out for evaluation. In this way, the data leading up to the held out set can be used for training and is not interrupted by data that was extracted for the hold out set. If one point is evaluated

at a time then the last data point is held out for testing. However, multiple points at the end can be held out to determine how effectively the algorithm can predict sensor events, user locations, or activities that occur multiple points into the time horizon.

While data can be effectively held out from the end of the sequence to evaluate a prediction algorithm, the evaluation method should be careful in selecting data for training. If the evaluation procedure is repeated in order to allow every data point to be used as a test instance, and the data leading up to the test instance is used for training the model, then the amount of training data will vary for each test point. As a result, a sliding window evaluation should be used in which a fixed number of data points m are used in one iteration of the evaluation process. If horizon n of future data points are predicted, then the first $m - n$ data points are used to train the model and the remaining n points are used for testing. The window can then move one position over in the sequence of available data and the process can be repeated. The method can be repeated for multiple window sizes as needed to show the sensitivity of the performance to alternative choices of window size.

Example 7.10 We can train the model on 10 symbols, test on the last symbol, and repeat the process 13 times in order to eventually process the entire sequence. If we employ the sequential prediction approach described in section 7.1, then the correct symbol will be output 5 out of the 13 iterations, resulting in an accuracy of 0.38. The performance is fairly low for this example because there are a very small number of data points on which to train the algorithm.

Jaccard Index Another method to evaluate the accuracy of sequential predictions is the Jaccard index. The Jaccard index, or Jaccard similarity coefficient, compares a set of labels generated by the model with the corresponding set of ground truth labels, as shown in Equation 7.5.

$$\text{Jaccard}\left(\text{Predicted}_{e\in\text{Events}}, \text{Actual}_{e\in\text{Events}}\right) = \frac{|\text{Predicted} \cap \text{Actual}|}{|\text{Predicted} \cup \text{Actual}|} \tag{7.5}$$

Note that this measure corresponds to the calculation of classification accuracy as shown in Equation 5.11. The Jaccard index for our sequential prediction example is thus 5/13, the same as the accuracy measure. However, the numerator and denominator of the ratio in Equation 7.5 can be replaced by distance measures that calculate how close the corresponding symbols are in the sequence.

Mean Absolute Error (MAE), Mean Squared Error (MSE) If a forecasting technique is used to estimate when an activity will occur, then the typical supervised learning notions of correct, incorrect, true positive, and false positive do not directly apply. Instead, we want to know how close the predicted value is to the actual value, and we want to accumulate these differences over every test point. Mean absolute error directly measures this quantity. As shown in Equation 7.6, the absolute value of the difference between the predicted and actual value is summed and normalized

over each of the data points. A well-known alternative to mean absolute error (MAE) is mean squared error (MSE), computed in Equation 7.7.

$$\text{MAE} = \frac{\sum_{e=1}^{\#\text{events}} |\text{predicted}(e) - \text{actual}(e)|}{\#\text{events}} \tag{7.6}$$

$$\text{MSE} = \frac{\sum_{e=1}^{\#\text{events}} (\text{predicted}(e) - \text{actual}(e))^2}{\#\text{events}} \tag{7.7}$$

Mean Signed Difference (MSD) Both the mean absolute error and the mean squared error disregard the direction of the error (predicting lower or higher than the actual value of the activity timing). The mean signed difference, shown in Equation 7.8, takes the direction into account in the aggregated error calculation.

$$\text{MSD} = \frac{\sum_{e=1}^{\#\text{events}} \text{predicted}(e) - \text{actual}(e)}{\#\text{events}} \tag{7.8}$$

Root Mean Squared Error (RMSE), Normalized Root Mean Squared Error (NRMSE) Yet another common measure of prediction error is the RMSE, shown in Equation 7.9. Like the earlier measures, this formula aggregates the difference between predicted and actual error and squares each difference to remove the sign factor. The square root is computed of the final estimate to offset the scaling factor of squaring the individual differences.

$$\text{RMSE} = \sqrt{\frac{\sum_{e=1}^{\#\text{events}} (\text{predicted}(e) - \text{actual}(e))^2}{\#\text{events}}} \tag{7.9}$$

Normalized Root Mean Squared Error (NRMSE) A difficulty with the measures that have been summarized so far is that the values are sensitive to the unit size of the predicted value. In the case of activity forecasting, a prediction that is one minute longer than the actual time (i.e., predicting a Bed Toilet Transition occurring in 6 minutes rather than 5) yields a smaller MAE, MSE, MSD, or RMSE value than if the prediction were made in seconds (i.e., the prediction was 360 seconds instead of 300, yielding an error of 60). Normalized versions of the error measures thus facilitate more direct comparison of error between different datasets and aids in interpreting the error measures. Two common methods are to normalize the error to the range of the observed data or normalize to the mean of the observed data. These formulas for the normalized root mean squared error are given in Equations 7.10 and 7.11.

$$\text{NRMSE} = \frac{\text{RMSE}}{\text{Max}\,(\text{Actual}) - \text{Min}\,(\text{Actual})} \tag{7.10}$$

$$\text{NRMSE} = \frac{\text{RMSE}}{\text{Mean}\,(\text{Actual})} \tag{7.11}$$

Pearson's Correlation Coefficient (r) Correlation, often measured as a correlation coefficient, indicates both the strength and the direction of a relationship between two variables. A number of different coefficient calculations have been used for different situations. The most popular is the Pearson product-moment correlation coefficient, which is obtained by computing the ratio between the covariance of the two variables to the product of their standard deviations. This is shown in Equation 3.11. When evaluating forecasting algorithms, n represents the number of evaluated data points and the correlation coefficient is used to estimate the correlation between the predicted values output by a learned model and the actual observed values. The correlation coefficient is $+1$ in the case of a perfect linear relationship and -1 in the case of an inverse relationship. A coefficient of 0 indicates that there is no linear relationship between the two sets of values. Calculating the correlation coefficient makes it easier to intuitively evaluate the scale-free performance of a forecasting algorithm because the coefficient range is $r = [-1 \ldots +1]$, regardless of the scale of the variable that is being predicted. In the case of activity prediction, the correlation coefficient is calculated as shown in Equation 7.12.

$$r = \frac{\text{Combined}}{\sqrt{\text{Predicted}} \times \sqrt{\text{Actual}}} \tag{7.12}$$

where the numerator represents the covariance of the predicted and actual values and is calculated as $\text{Combined} = (\Sigma_{e=1}^{n}(predicted(e) - \overline{predicted}) \times (actual(e) - \overline{actual}))/(n-1)$. The denominator represents the product of the standard deviations for the predicted values and the actual values, calculated as $\text{Predicted} = (\Sigma_{e=1}^{n}(predicted(e) - \overline{predicted})^2)/(n-1)$ and $\text{Actual} = (\Sigma_{e=1}^{n}(actual(e) - \overline{actual})^2)/(n-1)$.

Example 7.11 Table 7.1 summarizes the forecasting performance metrics for an example AP task. In this example, 80,000 data points were used to train a model to predict the number of seconds that would elapse before a Bed Toilet Transition activity was performed. In addition to utilizing the bag of sensors, sensor counts, and time of day features that are described in Chapter 3, additional forecasting features that index the predicted value at the previous three time units were included. The regression tree that was learned from this data is shown in Figure 7.6. Figure 7.8 shows the 1-step-ahead predicted values and the actual values for the Bed Toilet Transition times based on this learned model over a period of one week.

After the tree is learned, it is used to generate predicted values for 20 data points that were held out from training. Table 7.1 shows the actual and predicted values for the data points, along with the associated error for each data point. As can be seen, the tree tends to under-predict the time until the next occurrence of the activity, although there are occasionally very small values output by the tree. Table 7.2 summarizes the forecasting performance metrics for this example AP problem.

TABLE 7.1 Actual and Predicted Elapsed Time Until the Next Occurrence of a Bed Toilet Transition Activity. The Associated Error is Shown for Each Data Point

Actual	Predicted	Error
37,779	37,680.582	–98.418
37,770	37,674.573	–95.427
31,165	37,665.561	6,500.561
31,159	31,049.914	–109.086
31,147	31,043.903	–103.097
31,076	31,034.769	–41.231
31,016	30,963.587	–52.413
31,010	30,903.436	–106.564
30,999	30,897.452	–101.550
30,976	30,886.450	–89.550
30,967	30,863.393	–103.607
27,725	30,854.374	3,129.374
27,631	27,603.967	–27.033
27,627	27,509.727	–117.273
27,143	27,507.139	364.139
27,139	27,021.923	–117.077
27,115	27,017.915	–97.085
27,108	26,994.065	–113.935
27,100	26,987.048	–112.952
27,090	26,979.038	–110.962

7.6 ADDITIONAL READING

The LZ78 compression algorithms were introduced by Ziv and Lempel[152] and were adapted to address the problem of sequential prediction by Feder et al.[153]. Bhattacharya and Das[154] investigated methods to improve the convergence rate for the sequential prediction algorithm and Gopalratnam and Cook[155] incorporated a sliding window to improve predictive accuracy. Begleiter et al.[156] compare several alternative variable-order models for sequence prediction.

Quinlan introduced the idea of a regression tree with the M5 algorithm[157]. Decision trees are appealing because of the ability to interpret the learned model and also because training time is typically shorter than for neural network and SVM approaches. In addition, Utgoff[158] proposed methods to allow incremental induction of decision trees. This allows the activity model to be refined as new training data becomes available, without a need to store all of the training data and learn the model from scratch with each new batch of training data. Recurrent neural networks have

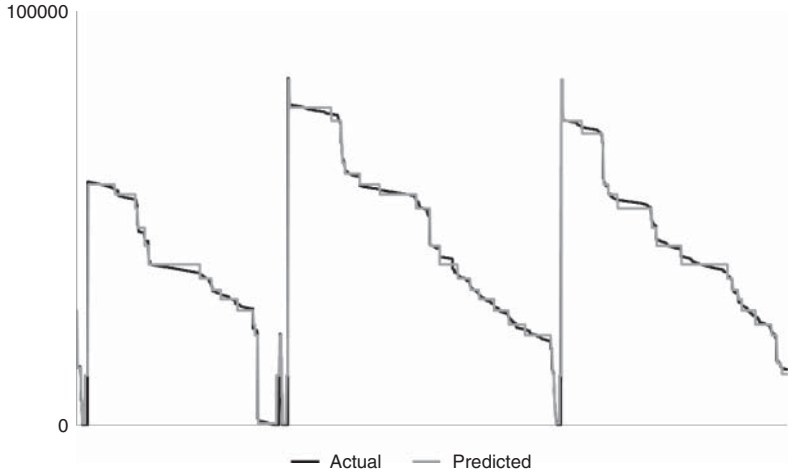

FIGURE 7.8 *Actual and predicted class values over a three day period based on a regression tree model. In this graph, the x-axis represents the time of day (from the beginning of day one to the end of day three) and the y-axis represents the number of seconds (predicted or actual) that will elapse before the next occurrence of the target activity, Bed Toilet Transition. The class variable is the number of seconds that will elapse before the next occurrence of the target activity.*

TABLE 7.2 Forecasting-Based Performance Evaluation for Regression Tree Algorithm Applied to Bed Toilet Transition Activity Prediction

Performance Metric and Value	
MAE	579.5667
MSE	2,617,237.0000
MSD	419.8407
RMSE	1,617.7880
NRMSE-Range	0.1514
NRMSE-Mean	0.0539
r	0.8943

been investigated by Mahmoud et al.[159] to predict occupancy regions in a home. The original ARIMA model was introduced by Box and Jenkins[160].

Krumm and Horvitz[161] applied Bayesian updating to infer the next location for drivers based on grid locations. The notion of explicit state duration HMMs has been explored by Dewar et al.[162], while autoregressive HMMs have been extensively used for applications such as speech processing[163] and visual tracking[164]. The idea of rule-based activity timings was investigated by Cook and Holder[165] and was used to generate automated activity initiation reminders.