Chapter 15

Mining Discrete Sequences

"*I am above the weakness of seeking to establish a sequence of cause and effect.*"—Edgar Allan Poe

15.1 Introduction

Discrete sequence data can be considered the categorical analog of timeseries data. As in the case of timeseries data, it contains a single *contextual* attribute that typically corresponds to time. However, the behavioral attribute is categorical. Some examples of relevant applications are as follows:

1. *System diagnosis:* Many automated systems generate discrete sequences containing information about the *system state*. Examples of system state are UNIX system calls, aircraft system states, mechanical system states, or network intrusion states.

2. *Biological data:* Biological data typically contains sequences of amino acids. Specific patterns in these sequences may define important properties of the data.

3. *User-action sequences:* Various sequences are generated by user actions in different domains.

   (a) Web logs contain long sequences of visits to Websites by different individuals.

   (b) Customer transactions may contain sequences of buying behavior. The individual elements of the sequence may correspond to the identifiers of the different items, or sets of identifiers of different items that are bought.

   (c) User actions on Websites, such as online banking sites, are frequently logged. This case is similar to that of Web logs, except that the logs of banking sites often contain more detailed information for security purposes.

Biological data is a special kind of sequence data, in which the contextual data is not temporal but relates to the *placement* of the different attributes. Methods for temporal sequence data can be leveraged for biological sequence data and vice versa. A discrete sequence is formally defined as follows:

Definition 15.1.1 (Discrete Sequence Data) *A discrete sequence $\overline{Y_1} \dots \overline{Y_n}$ of length n and dimensionality d, contains d discrete feature values at each of n different timestamps $t_1 \dots t_n$. Each of the n components $\overline{Y_i}$ contains d discrete behavioral attributes $(y_i^1 \dots y_i^d)$, collected at the ith timestamp.*

In many practical scenarios, the timestamps $t_1 \dots t_n$ may simply be tick values indexed from 1 through n. This is especially true in cases such as biological data, in which the contextual attribute represents placement. In general, the actual timestamps are rarely used in most sequence mining applications, and the discrete sequence values are assumed to be equally spaced in time. Furthermore, most of the analytical techniques are designed for the case where $d = 1$. Such discrete sequences are also referred to as *strings*. This chapter will, therefore, use these terms interchangeably. Most of the discussion in this chapter focuses on these more common and simpler cases.

In some applications, such as *sequential pattern mining*, each $\overline{Y_i}$ is not a vector but a set of unordered values. This is a variation from Definition 15.1.1. Therefore, the notation Y_i (without an overline) will be used to denote a set rather than a vector. For example, in a supermarket application, the set Y_i may represent a set of items bought by the customer at a given time. There is no temporal ordering among the items in Y_i. In a Web log analysis application, the set Y_i represents the Web pages browsed by a given user in a single session. Thus, discrete sequences can be defined in a wider variety of ways than timeseries data. This is because of the ability to define sets on discrete items. Each position in the sequence is also referred to as an *element* and is composed of individual *items* in the set. Throughout this chapter, the word "element" will refer to one of the sets of items within the sequence, including a 1-itemset.

These variations in definitions arise out of a natural variation in the different kinds of application scenarios. This chapter will study the different problem definitions relevant to discrete sequence mining. The four major problems of pattern mining, clustering, outlier analysis, and classification, are each defined differently for discrete sequence mining than for multidimensional data. These different definitions will be studied in detail. A few models, such as *Hidden Markov Models*, are used widely across many different application domains. These commonly used models will also be studied in this chapter.

This chapter is organized as follows. Section 15.2 introduces the problem of sequential pattern mining. Sequence clustering algorithms are discussed in Sect. 15.3. The problem of sequence outlier detection is discussed in Sect. 15.4. Section 15.5 introduces Hidden Markov Models (HMM) that can be used for either clustering, classification, or outlier detection. The problem of sequence classification is addressed in Sect. 15.6. Section 15.7 gives a summary.

15.2 Sequential Pattern Mining

The problem of sequential pattern mining can be considered the temporal analog of frequent pattern mining. In fact, most algorithms for frequent pattern mining can be directly adapted to sequential pattern mining with a systematic approach, although the latter problem is more complex. As in frequent pattern mining, the original motivating application for sequential pattern mining was market basket analysis, although the problem is now used in a wider variety of temporal application domains, such as computer systems, Web logs, and telecommunication applications.

The sequential pattern mining problem is defined on a set of N sequences. The ith sequence contains n_i *elements* in a specific temporal order. Each element contains a set of *items*. The complex element is, therefore, a set, such as a basket of items bought by a customer. For example, consider the sequence:

$\langle \{Bread, Butter\}, \{Butter, Milk\}, \{Bread, Butter, Cheese\}, \{Eggs\} \rangle$

Here, $\{Bread, Butter\}$ is an element, and $Bread$ is an item inside the element. A subsequence of this sequence is also a temporal ordering of sets, such that each element in the subsequence is a subset of an element in the base sequence *in the same temporal order*. For example, consider the following sequence:

$\langle \{Bread, Butter\}, \{Bread, Butter\}, \{Eggs\} \rangle$

The second sequence is a subsequence of the first because each element in the second sequence can be matched to a corresponding element in the first sequence *by a subset relationship*, so that the matching elements are in the same temporal order. Unlike transactions that are sets, note that sequences (and the mined subsequences) contain *ordered* (and possibly repeated) elements, each of which is itself like a transaction. For example, $\{Bread, Butter\}$ is a repeated element in one of the aforementioned sequences, and it may correspond to two separate visits of a customer to the supermarket at different times. Formally, a subsequence relationship is defined as follows:

Definition 15.2.1 (Subsequence) *Let $\mathcal{Y} = \langle Y_1 \ldots Y_n \rangle$ and $\mathcal{Z} = \langle Z_1 \ldots Z_k \rangle$ be two sequences, such that all the elements Y_i and Z_i in the sequences are sets. Then, the sequence $\mathcal{Z}$ is a subsequence of $\mathcal{Y}$, if k elements $Y_{i_1} \ldots Y_{i_k}$ can be found in $\mathcal{Y}$, such that $i_i < i_2 < \ldots < i_k$, and $Z_r \subseteq Y_{i_r}$ for each $r \in \{1 \ldots k\}$.*

Consider a sequence database $\mathcal{T}$ containing a set of N sequences $\mathcal{Y}_1 \ldots \mathcal{Y}_N$. The *support* of a subsequence $\mathcal{Z}$ with respect to database $\mathcal{T}$ is defined in an analogous way to frequent pattern mining.

Definition 15.2.2 (Support) *The support of a subsequence $\mathcal{Z}$ is defined as the fraction of sequences in the database $\mathcal{T} = \{\mathcal{Y}_1 \ldots \mathcal{Y}_N\}$, that contain $\mathcal{Z}$ as a subsequence.*

The sequential pattern mining problem is that of identifying all subsequences that satisfy the required level of minimum support *minsup*.

Definition 15.2.3 (Sequential Pattern Mining) *Given a sequence database $\mathcal{T} = \{\mathcal{Y}_1, \ldots \mathcal{Y}_N\}$, determine all subsequences whose support with respect to the database $\mathcal{T}$ is at least* minsup.

It is easy to see that this definition is very similar to that of the definition of association pattern mining in Chap. 4. The minimum support value *minsup* can be specified either as an absolute value, or as a relative support value. As in the case of frequent pattern mining, a relative value will be assumed, unless otherwise specified.

An *Apriori*-like algorithm, known as *Generalized Sequential Pattern Mining (GSP)*, was proposed as the first algorithm for sequential pattern mining. This algorithm is very similar to *Apriori*, in terms of how candidates are generated and counted. In fact, many frequent pattern mining algorithms, such as *TreeProjection* and *FP-growth*, have direct analogs in sequential pattern mining. This section describes only the *GSP* algorithm in detail. A later

```
Algorithm GSP(Sequence Database: $\mathcal{T}$, Minimum Support: minsup)
begin
  $k = 1$;
  $\mathcal{F}_k$ = { All Frequent 1-item elements };
  while $\mathcal{F}_k$ is not empty do begin
    Generate $\mathcal{C}_{k+1}$ by joining pairs of sequences in $\mathcal{F}_k$, such that
        removing an item from the first element of one sequence matches the sequence
        obtained by removing an item from the last element of the other;
    Prune sequences from $\mathcal{C}_{k+1}$ that violate downward closure;
    Determine $\mathcal{F}_{k+1}$ by support counting on $(\mathcal{C}_{k+1}, \mathcal{T})$ and retaining
        sequences from $\mathcal{C}_{k+1}$ with support at least minsup;
    $k = k + 1$;
  end;
  return($\cup_{i=1}^{k} \mathcal{F}_i$);
end
```

Figure 15.1: The *GSP* algorithm is related to the *Apriori* algorithm. The reader is encouraged to compare this pseudocode with the *Apriori* algorithm described in Fig. 4.2 of Chap. 4

section provides a broad overview of how enumeration tree algorithms can be generalized to sequential pattern mining.

The *GSP* and *Apriori* algorithms are similar, except that the former needs to be designed for finding frequent sequences rather than sets. First, the notion of the *length* of candidates needs to be defined in sequential pattern mining. This notion needs to be defined more carefully, because the individual elements in the sequence are sets rather than items. The *length* of a candidate or a frequent sequence is equal to the number of items (not elements) in the candidate. In other words, a k-sequence) $\langle Y_1 \ldots Y_r \rangle$ is a sequence with length $\sum_{i=1}^{r} |Y_i| = k$. Thus, $\langle \{Bread, Butter, Cheese\}, \{Cheese, Eggs\} \rangle$ is a 5-candidate, even though it contains only 2 elements. This is because this sequence contains 5 items in total, including a repetition of "*Cheese*" in two distinct elements. A $(k-1)$-subsequence of a k-candidate can be generated by removing an item from any element in the k-sequence. The *Apriori* property continues to hold for sequences because any $(k-1)$-subsequence of a k-sequence will have support at least equal to that of the latter. This sets the stage for a candidate generate-and-test approach, together with downward closure pruning, which is analogous to *Apriori*.

The *GSP* algorithm starts by generating all frequent 1-item sequences by straightforward counting of individual items. This set of frequent 1-sequences is represented by $\mathcal{F}_1$. Subsequent iterations construct $\mathcal{C}_{k+1}$ by joining pairs of sequence patterns in $\mathcal{F}_k$. The join process is different from association pattern mining because of the greater complexity in the definition of sequences. Any pair of frequent k-sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ can be joined, if removing an item from the first element in one frequent k-sequence $\mathcal{S}_1$ is identical to the sequence obtained by removing an item from the last element in the other frequent sequence $\mathcal{S}_2$. For example, the two 5-sequences $\mathcal{S}_1 = \langle \{Bread, Butter, Cheese\}, \{Cheese, Eggs\} \rangle$ and $\mathcal{S}_2 = \langle \{Bread, Butter\}, \{Milk, Cheese, Eggs\} \rangle$ can be joined because removing "*Cheese*" from the first element of $\mathcal{S}_1$ will result in an identical sequence to that obtained by removing "*Milk*" from the last element of $\mathcal{S}_2$. Note that if $\mathcal{S}_2$ were a 5-candidate with 3 elements corresponding to $\mathcal{S}_2 = \langle \{Bread, Butter\}, \{Cheese, Eggs\}, \{Milk\} \rangle$, then a join can also be performed. This is because removing the last item from $\mathcal{S}_2$ creates a sequence with 2 elements and 4 items, which is identical to $\mathcal{S}_1$. However, the nature of the join will be somewhat different in these cases. In general, cases where the last element of $\mathcal{S}_2$ is a 1-itemset need to be treated specially. The following rules can be used to execute the join:

1. If the last element of $\mathcal{S}_2$ is a 1-itemset, then the joined candidate may be obtained by *appending* the last element of $\mathcal{S}_2$ to $\mathcal{S}_1$ as a separate element. For example, consider the following two sequences:

   $\mathcal{S}_1 = \langle\{Bread, Butter, Cheese\}, \{Cheese, Eggs\}\rangle$
   $\mathcal{S}_2 = \langle\{Bread, Butter\}, \{Cheese, Eggs\}, \{Milk\}\rangle$

   The join of the two sequences is $\langle\{Bread, Butter, Cheese\}, \{Cheese, Eggs\}, \{Milk\}\rangle$.

2. If the last element of $\mathcal{S}_2$ is not a 1-itemset, but a superset of the last element of $\mathcal{S}_1$, then the joined candidate may be obtained by *replacing* the last element of $\mathcal{S}_1$ with the last element of $\mathcal{S}_2$. For example, consider the following two sequences:

   $\mathcal{S}_1 = \langle\{Bread, Butter, Cheese\}, \{Cheese, Eggs\}\rangle$
   $\mathcal{S}_2 = \langle\{Bread, Butter\}, \{Milk, Cheese, Eggs\}\rangle$

   The join of the two sequences is $\langle\{Bread, Butter, Cheese\}, \{Milk, Cheese, Eggs\}\rangle$.

These key differences from *Apriori* joins are a result of the temporal complexity and the set-based elements in sequential patterns. Alternative methods exist for performing the joins. For example, an alternative approach is to remove one item from the *last* elements of *both* $\mathcal{S}_1$ and $\mathcal{S}_2$ to check whether the resulting sequences are identical. However, in this case multiple candidates might be generated from the same pair. For example, $\langle a, b, c\rangle$ and $\langle a, b, d\rangle$ can join to any of $\langle a, b, c, d\rangle$, $\langle a, b, d, c\rangle$, and $\langle a, b, cd\rangle$. The first join rule of removing the first item from $\mathcal{S}_1$ and the last item from $\mathcal{S}_2$ has the merit of having a unique join result. For any specific join rule, it is important to ensure exhaustive and nonrepetitive generation of candidates. As we will see later, a similar notion to the frequent-pattern enumeration tree can be introduced in sequential pattern mining to ensure exhaustive and nonrepetitive candidate generation.

The *Apriori* trick is then used to prune sequences that violate downward closure. The idea is to check if each k-subsequence of a candidate in $\mathcal{C}_{k+1}$ is present in $\mathcal{F}_k$. The candidate set is pruned of those candidates that do not satisfy this closure property. The frequent $(k+1)$-candidate sequences $\mathcal{C}_{k+1}$ are then checked against the sequence database $\mathcal{T}$, and the support is counted. The counting of the support is executed according to the notion of a subsequence, as presented in Definition 15.2.1. All frequent candidates in $\mathcal{C}_{k+1}$ are retained in $\mathcal{F}_{k+1}$. The algorithm terminates, when no frequent sequences are generated in $\mathcal{F}_{k+1}$ in a particular iteration. All the frequent sequences generated during the different iterations of the levelwise approach are returned by the algorithm. The pseudocode of the *GSP* algorithm is illustrated in Fig. 15.1.

15.2.1 Frequent Patterns to Frequent Sequences

It is easy to see that the *Apriori* and *GSP* algorithms are structurally similar. This is not a coincidence. The basic structure of the frequent pattern and sequential pattern mining problems are similar. Aside from the differences in the support counting approach, the main difference between *GSP* and *Apriori* is in terms of how candidates are generated. The join

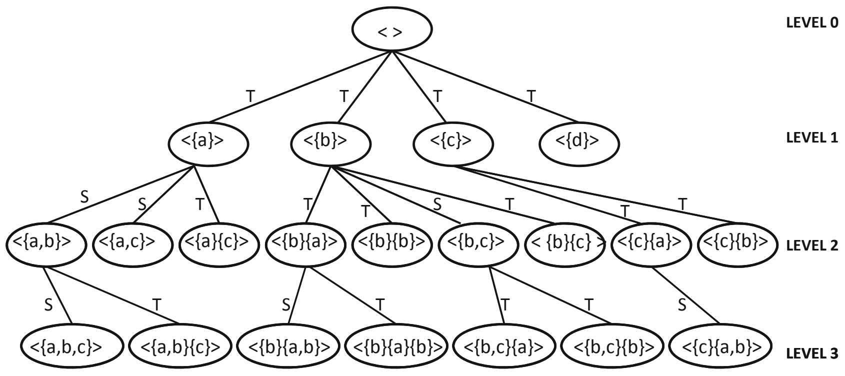


Figure 15.2: The equivalent of an enumeration tree for sequential pattern mining

generation in *GSP* is defined in terms of two separate cases. The two cases correspond to *temporal* extensions and *set-wise* extensions of candidates.

As discussed in Chap. 4, the *Apriori* algorithm can be viewed as an enumeration tree algorithm. It is also possible to define an analogous candidate tree in sequential pattern mining, albeit with a somewhat different structure than the enumeration tree in frequent pattern mining. The key differences in join-based candidate generation between *Apriori* and *GSP* algorithms translate to differences in the structure and growth of the candidate tree in sequential pattern mining. In general, candidate trees for sequential pattern mining are more complex because they need to accommodate both temporal and set-wise growth of sequences. Therefore, the definition of candidate extensions of a tree-node needs to be changed. A node for sequence $\mathcal{S}$ can be extended to a lower-level node in one of two ways:

1. *Set-wise extension:* In this case, an item is added to the last element in the sequence $\mathcal{S}$ to create a candidate pattern. Therefore, the number of elements does not increase. For an item to be added to the last element of $\mathcal{S}$, it must satisfy two properties; (a) the item successfully extends the parent sequence of $\mathcal{S}$ in the candidate tree with either a set-wise or temporal extension to another frequent sequence, and (b) the item must be *lexicographically* later than all items in the last element of $\mathcal{S}$. As in frequent pattern mining, a lexicographic ordering of items needs to be fixed up front.

2. *Temporal extension:* A new element with a single item is added to the end of the current sequence S. As in the previous case of set-wise extensions, any frequent item extension of the parent of $\mathcal{S}$ may be used to extend $\mathcal{S}$ (condition (a)). However, the added item need not be lexicographically later than the items in the last element of sequence $\mathcal{S}$.

These two kinds of extensions can be shown to be equivalent to the two kinds of joins in the *GSP* algorithm. As in frequent pattern mining, the candidate extensions of a node are a subset of the corresponding frequent extensions of its parent node. An example of the frequent portion of the candidate tree for sequential pattern mining is illustrated in Fig. 15.2. Note the greater complexity of the tree, because of set-wise and temporal addition of items at each level of the tree. The set-wise extensions are marked as “S”, and the temporal extensions are marked as “T” on the corresponding tree edges. A particularly illuminating discussion on this topic may be found in [243], on which this example is based.

It is possible to convert any enumeration tree algorithm for frequent pattern mining to a sequential pattern mining algorithm by systematically making appropriate modifications. These changes account for the different structure of the candidate tree in sequential pattern mining compared to that in frequent pattern mining. This candidate tree is implicitly generated by all sequential pattern mining algorithms, such as *GSP* and *PrefixSpan*. Because the enumeration tree[1] is the generic framework describing all frequent pattern mining algorithms, it implies that virtually all frequent pattern mining algorithms can be modified to the sequential pattern mining scenario. For example, the work in [243] generalizes *TreeProjection*, and the *PrefixSpan* algorithm generalizes *FP-growth*. Savasere et al.'s vertical format [446] has also been generalized to sequential pattern mining algorithms. The main difference between these algorithms is the different efficiency of counting with the use of a variety of data structures, projection-based reuse tricks, and different candidate-tree exploration strategies such as breadth-first or depth-first, rather than a fundamental difference in search space size. The size of the candidate tree is fixed, although pruning strategies such as the *Apriori*-style pruning can reduce its size.

The notion of projection-based reuse can also be extended to sequential pattern mining. The projected representation $\mathcal{T}(\mathcal{P})$ of the sequence database $\mathcal{T}$ is associated with a sequential pattern $\mathcal{P}$ in the candidate enumeration tree (as modified for sequential pattern mining). Then, each sequence $\mathcal{Y} \in \mathcal{T}$ in the database is projected at $\mathcal{P}$ according to the following rules:

1. The sequential pattern $\mathcal{P}$ needs to be a subsequence of $\mathcal{Y}$ for the projection of $\mathcal{Y}$ to be included in the projected database $\mathcal{T}(\mathcal{P})$.

2. All items that are either not in the last element of $\mathcal{P}$, or are not successful frequent extensions (either temporal or set-wise) of the parent of $\mathcal{P}$ are not included in the projection of $\mathcal{Y}$ because they are irrelevant for counting frequent extensions of $\mathcal{P}$.

3. The *earliest* temporal occurrence of $\mathcal{P}$ in $\mathcal{Y}$, as a subsequence, is determined. Let the last element P_r in $\mathcal{P}$ be matched to the element Y_k in $\mathcal{Y}$ according to this subsequence matching. Then, the first element of the projected representation of $\mathcal{Y}$ is equal to the set of items in $Y_k - P_r$ that are lexicographically later than all items in P_r. If the resulting element Q is null, then it is not included in the projection of $\mathcal{Y}$. This first element, if non-null, is special because it may only be used for counting set-wise extensions of element P_r in the enumeration tree and is, therefore, denoted as $\_Q$ with an underscore in front of it.

4. The remaining elements in the projected sequence after $\_Q$ correspond to the elements in $\mathcal{Y}$ occurring temporally after the last matched element Y_k in $\mathcal{Y}$. All these elements are included in the projection of $\mathcal{Y}$ after removing the irrelevant items discussed in step 2. These remaining elements can be used for counting either set-wise extensions of the last element P_r in $\mathcal{P}$, or temporal extensions of $\mathcal{P}$. For any of these remaining elements (other than $\_Q$) to be used for counting the set-wise extensions of P_r, the element would already need to contain P_r.

The projected database $\mathcal{T}(\mathcal{P})$ can be used to count the frequent extensions of $\mathcal{P}$ more efficiently and determine the frequent ones. As in the frequent pattern mining, this projection can be performed successively in top-down fashion during the construction an enumeration-tree-like candidate structure. The projected database at a node can be generated by recursively projecting the database at its parent. The basic approach is exactly analogous to

[1]See discussion in Sect. 4.4.4.5 of Chap. 4. A similar argument applies to sequential pattern mining.

projection-based frequent pattern mining algorithms discussed in Chap. 4. The algorithm starts with a candidate tree $\mathcal{ET}$ which is the null node with the entire sequence database $\mathcal{T}$ at that node. This tree is extended repeatedly using the following step until no nodes remain in $\mathcal{ET}$ for further extension.

```
Select a node $(\mathcal{P}, \mathcal{T}(\mathcal{P}))$ in $\mathcal{ET}$ for extension;
Generate the candidate temporal and set-wise extensions of $\mathcal{P}$;
Determine the frequent extensions of $\mathcal{P}$ using support counting on $\mathcal{T}(P)$;
Extend $\mathcal{ET}$ with frequent extensions and their recursively projected databases;
```

The final candidate tree $\mathcal{ET}$ contains all the frequent sequential patterns. Different strategies for selecting the node $\mathcal{P}$ can lead to the generation of the candidate tree in a different order such as breadth-first or depth-first order. This simplified and generalized description is roughly based on the frameworks independently proposed in [243] and *PrefixSpan*, which are closely related. The reader is referred to the bibliographic notes for discussion of the specific algorithms.

15.2.2 Constrained Sequential Pattern Mining

In many cases, additional constraints are imposed on the sequential patterns, such as constraints on gaps between successive elements of the sequence. One solution is to use the unconstrained *GSP* algorithm, and then, as a postprocessing step, remove all subsequences not satisfying the constraint. However, this brute-force approach is a very inefficient solution because the number of constrained patterns may be orders of magnitude smaller than the unconstrained patterns. Therefore, the incorporation of the constraints *directly* into the *GSP* algorithm, during the candidate-generation or support-counting step, is significantly more efficient. Depending on the nature of the constraints, the changes required to the *GSP* algorithm may be minor, or significant. In all cases, the support-counting procedure for $\mathcal{F}_k$ needs to be modified. The constraints are explicitly checked during the support counting. This reduces the number of frequent patterns generated, and makes the process more efficient than the brute-force method. However, the incorporation of such constraints may sometimes result in invalidation of the downward closure property of the mined patterns. In such cases, appropriate changes may need to be made to the *GSP* algorithm. In cases where the downward closure property is not violated, the *GSP* algorithm can be used with very minor modifications for constraint checking during support counting.

An important constraint that does not violate the downward closure property is the *maxspan* constraint. This constraint specifies that the time difference between the first and last elements of a subsequence must be no larger than *maxspan*. Therefore, the *GSP* algorithm can be used directly, with the modification that the constraint is checked during support counting. Thus, the approach works with a much smaller set of subsequences in each step and is generally more efficient than the brute-force method.

Another common constraint in sequential pattern mining is the maximum gap constraint. This is also referred to as the *maxgap* constraint. Note that all $(k-1)$-subsequences of a particular frequent k-sequence may not be valid because of the maximum gap constraint. This is somewhat problematic because the *Apriori* principle cannot be used effectively. For example, the subsequence a_1a_5 is not supported by the transaction database sequence $a_1a_2a_3a_4a_5$ under the *maxgap* value of 1, because the gap value between a_1 and a_5 is 3. However, the subsequence $a_1a_3a_5$ is supported by the same transaction database sequence under this *maxgap* value. It is, therefore, possible for a_1a_5 to have lower support than $a_1a_3a_5$. Thus, *Apriori* pruning cannot be applied. However, the sequence obtained by dropping items from the *first* or *last* elements of a frequent sequence will always be frequent. Therefore, the *specific* join-based approach discussed in this chapter can still be used to exhaustively generate all candidates without losing any frequent patterns. A modified pruning rule is used. While checking $(k-1)$-subsequences of a candidate for pruning, only subsequences containing the same number of *elements* as the candidate are checked. In other words, elements containing 1 item cannot be removed from the candidate for checking its subsequences. Such subsequences are referred to as *contiguous subsequences*. A noteworthy special case is one in which $maxgap = 0$. This case is often used for determining timeseries motifs after a time series has been converted to a discrete sequence using methods discussed in Sect. 14.4 of Chap. 14.

Another constraint of interest is the minimum gap constraint, or *mingap* constraint between successive elements. A minimum gap constraint between successive elements will always satisfy the downward closure property. Therefore, the *GSP* approach can be used with very minor modifications. The only modification is that this constraint is checked during support counting. This generates a smaller set of subsequences $\mathcal{F}_k$. The join and pruning steps remain unchanged. The bibliographic notes contain pointers to many other interesting constraints such as the window-size constraint.

15.3 Sequence Clustering

As in the case of timeseries data, the clustering of sequences is heavily dependent on the definition of similarity. When a similarity function has been defined, many of the traditional multidimensional methods such as k-medoids and graph-based methods can be easily adapted to sequence data. It should be pointed out that these two methods can be used for virtually any data type, and are dependent only on the choice of distance function.

Similarity measures for sequence data have been defined in Chap. 3. The most common similarity functions used for sequence data are as follows:

1. *Match-based measure:* This measure is equal to the number of matching positions between the two sequences. This can be meaningfully computed only when the two sequences are of equal length, and a one-to-one correspondence exists between the positions.

2. *Dynamic time warping (DTW):* In this case, the number of *nonmatches* between the two sequences can be used with dynamic time warping. The dynamic time warping (*DTW*) method is discussed in detail in Sect. 3.4.1.3 of Chap. 3. The idea is to stretch and shrink the time dimension dynamically to account for the varying speeds of data generation for different series.

3. *Longest common subsequence (LCSS):* As the name of this measure suggests, the longest matching subsequence between the two sequences is computed. This is then used to measure the similarity between the two sequences. The *LCSS* method is discussed in detail in Sect. 3.4.2.2 of Chap. 3.

4. *Edit distance:* This is defined as the cost of edit operations required to transform one sequence into another. The edit distance measure is described in Sect. 3.4.2.1 of Chap. 3. A number of alignment methods, such as BLAST, are specifically designed for biological sequences. Pointers to these methods may be found in the bibliographic notes.

5. *Keyword-based similarity:* In this case, a k-gram representation is used, in which each sequence is represented by a bag of segments of length k. These k-grams are extracted from the original data sequences by using a sliding window of length k on the sequences. Each such k-gram represents a new "keyword," and a *tf-idf* representation can be used in terms of these keywords. If desired, the infrequent k-grams can be dropped. Any text-based, vector-space similarity measure, discussed in Chap. 13, may be used. Since the ordering of the segments is no longer used after the transformation, such an approach allows the use of a wider range of data mining algorithms. In fact, any text mining algorithm can be used on this transformation.

6. *Kernel-based similarity:* Kernel-based similarity is particularly useful for SVM classification. Some examples of kernel-based similarity are discussed in detail in section 15.6.4.

The different measures are used in different application-specific scenarios. Many of these scenarios will be discussed in this chapter.

15.3.1 Distance-Based Methods

When a distance or similarity function has been defined, the k-medoids method can be generalized very simply to sequence data. The k-medoids method is agnostic as to the choice of data type and the similarity function because it is designed as a generic hill-climbing approach. In fact, the *CLARANS* algorithm, discussed in Sect. 7.3.1 of Chap. 7, can be easily generalized to work with any data type. The algorithm selects k representative sequences from the data, and assigns each data point to their closest sequence, using the selected distance (similarity) function. The quality of the representative sequences is improved iteratively with the use of a hill-climbing algorithm.

The hierarchical methods, discussed in Sect. 6.4 of Chap. 6, can also be generalized to any data type because they work with pairwise distances between the different data objects. The main challenge of using hierarchical methods is that they require $O(n^2)$ pairwise distance computations between n objects. Distance function computations on sequence data are generally expensive because they require expensive dynamic programming methods. Therefore, the applicability of hierarchical methods is restricted to smaller data sets.

15.3.2 Graph-Based Methods

Graph-based methods are a general approach to clustering that is agnostic to the underlying data type. The transformation of different data types to similarity graphs is described in Sect. 2.2.2.9 of Chap. 2. The broader approach in graph-based methods is as follows:

1. Construct a graph in which each node corresponds to a data object. Each node is connected to its k-nearest neighbors, with a weight equal to the similarity between the corresponding pairs of data objects. In cases where a distance function is used, it is converted to a similarity function as follows:

$$w_{ij} = e^{-d(O_i,O_j)^2/t^2} \tag{15.1}$$

Here, $d(O_i, O_j)$ represents the distance between the objects O_i and O_j and t is a parameter.

2. The edges are assumed to be undirected, and any parallel edges are removed by dropping one of the edges. Because the distance functions are assumed to be symmetric, the parallel edges will have the same weight.

3. Any of the clustering or community detection algorithms discussed in Sect. 19.3 of Chap. 19 may be used for clustering nodes of the newly created graph.

After the nodes have been clustered, these clusters can be mapped back to clusters of data objects by using the correspondence between nodes and data objects.

15.3.3 Subsequence-Based Clustering

The major problem with the aforementioned methods is that they are based on similarity functions that use *global* alignment between the sequences. For longer sequences, global alignment becomes increasingly ineffective because of the noise effects of computing similarity between pairs of long sequences. Many local portions of sequences are noisy and irrelevant to similarity computations even when large portions of two sequences are similar. One possibility is to design local alignment similarity functions or use the keyword-based similarity method discussed earlier.

A more direct approach is to use frequent subsequence-based clustering methods. Some related approaches also use k-grams extracted from the sequence instead of frequent subsequences. However, k-grams are generally more sensitive to noise than frequent subsequences. The idea is that the frequent subsequences represent the key structural characteristics that are common across different sequences. After the frequent subsequences have been determined, the original sequences can be transformed into this new feature space, and a "bag-of-words" representation can be created in terms of these new features. Then, the sequence objects can be clustered in the same way as any text-clustering algorithm. The overall approach can be described as follows:

1. Determine the frequent subsequences $\mathcal{F}$ from the sequence database $\mathcal{D}$ using any frequent sequential pattern mining algorithm. Different applications may vary in the specific constraints imposed on the sequences, such as a minimum or maximum length of the determined sequences.

2. Determine a subset $\mathcal{F}_S$ from the frequent subsequences $\mathcal{F}$ based on an appropriate selection criterion. Typically, a subset of frequent subsequences should be selected, so as to maximize coverage and minimize redundancy. The idea is to use only a modest number of relevant features for clustering. For example, the notion of *Frequent Summarized Subsequences (FSS)* is used to determine condensed groups of sequences [505]. The bibliographic notes contain specific pointers to these methods.

3. Represent each sequence in the database as a "bag of frequent subsequences" from $\mathcal{F}_S$. In other words, the transformed representation of a sequence contains all frequent subsequences from $\mathcal{F}_S$ that it contains.

4. Apply any text-clustering algorithm on this new representation of the database of sequences. Text-clustering algorithms are discussed in Chap. 13. The tf-idf weighting may be applied to the different features, as discussed in Chap. 13.

The aforementioned discussion is a broad overview of frequent subsequence-based clustering, although the individual steps are implemented in different ways by different methods. The key differences among the different algorithms are in terms of methodology for feature

```
**Algorithm** *CLUSEQ*(Sequence Database: $\mathcal{D}$, Similarity Threshold: $t$)
**begin**
  $k = f = 1$;
  Let $\mathcal{C}_1$ be a singleton cluster with randomly chosen sequence;
  **repeat**
    Add $k_a = k \cdot f$ new singleton clusters containing sequences
      that are as different as possible from existing clusters/each other;
    $k = k + k_a$;
    Assign (if possible) each sequence in $\mathcal{D}$ to each cluster in
      $\mathcal{C}_1 \ldots \mathcal{C}_k$ for which the similarity is at least $t$;
    Eliminate the $k_r$ clusters containing less than *minthresh*
        sequences *uniquely* assigned to them;
    $k = k - k_r$;
    $f = \frac{\max\{k_a - k_r, 0\}}{k_a}$;
  **until** no change in clustering result;
  **return** clusters $\mathcal{C}_1 \ldots \mathcal{C}_k$;
**end**
```

Figure 15.3: The simplified *CLUSEQ* Algorithm

construction and the choice of the text-clustering algorithm. The *CONTOUR* method [505] uses a two-level hierarchical clustering, where fine-grained microclusters are generated in the first step. Then, these microclusters are agglomerated into higher-level clusters. The bibliographic notes contain pointers to specific instantiations of this framework.

15.3.4 Probabilistic Clustering

Probabilistic clustering methods are based on the generative principle, that a symbol in a given sequence is generated with a probability defined by statistical correlations with the symbols before it. This is based on the general principle of *Markovian Models*. Therefore, the similarity between a sequence and a cluster is computed using the generative probability of the symbols within that cluster. After a similarity function has been defined between a cluster and a sequence, it can be used to create a distance-based algorithm. The *CLUSEQ* algorithm is based on this principle.

15.3.4.1 Markovian Similarity-Based Algorithm: CLUSEQ

The *CLUstering SEQuences (CLUSEQ)* algorithm is based on the broader principle of Markovian Models. Markovian models are used to define a similarity function between a sequence and cluster. The *CLUSEQ* algorithm can otherwise be considered a similarity-based iterative partitioning algorithm. While traditional partitioning algorithms fix the number of clusters over multiple iterations, this is not the case in *CLUSEQ*. The *CLUSEQ* algorithm starts with only a single cluster. A carefully controlled number of new clusters containing individual sequences are added in each iteration, and older ones are removed when they are deemed to be too similar to existing clusters. The initial growth in the number of clusters is rapid but it slows down over the course of the algorithm. It is even possible for the number of clusters to shrink in later iterations. One advantage of this approach is that the algorithm can automatically determine the natural number of clusters.

Instead of using the number of clusters as an input parameter, the *CLUSEQ* algorithm works with a similarity threshold t. A sequence is assigned to a cluster, if its similarity to the cluster exceeds the threshold t. Sequences may be assigned to any number of clusters (or no cluster) as long as the similarity is greater than t. The *CLUSEQ* algorithm has three main steps corresponding to addition of new clusters, assignment of sequences to clusters, and elimination of clusters. These steps are repeated iteratively until there is no change in the clustering result. A simplified version[2] of the *CLUSEQ* algorithm is described in Fig. 15.3. A detailed description of the individual steps is provided below.

1. *Cluster addition:* The number of clusters added is $k \cdot f$, where k is the number of clusters at the end of the last iteration. The value of f is in the range $(0, 1)$, and is computed as follows. Let k_a be the number of clusters added in the previous iteration, and let k_r be the number of clusters removed because of elimination of overlapping clusters in the previous iteration. Then, the value of f is computed as follows:

$$f = \frac{\max\{k_a - k_r, 0\}}{k_a} \tag{15.2}$$

   The rationale for this is that when the algorithm reaches its "natural" number of clusters, eliminations will dominate. In such cases, f will be small or 0, and few new clusters need to be added. On the other hand, in cases where the current number of clusters is significantly lower than the "natural" number of clusters in the data, the value of f should be close to 1. In earlier iterations, the number of added clusters is much larger than the number of removed clusters, which results in rapid growth.

   The new clusters created are singleton clusters. The sequences that are as different as possible from both the existing clusters and each other are selected. Therefore pairwise similarity needs to be computed between each unclustered sequence and other clusters/unclustered sequences. Because it can be expensive to compute pairwise similarity between the clusters and all unclustered sequences, a sample of unclustered sequences is used to restrict the scope of new seed selection. The approach for computing similarity will be described later.

2. *Sequence assignment to clusters:* Sequences are assigned to clusters for which the similarity to the cluster is larger than a user-specified threshold t. The original *CLUSEQ* algorithm provides a way to adjust the threshold t as well, though the description in this chapter provides only a simplified version of the algorithm, where t is fixed and specified by the user. A given sequence may be assigned to either multiple clusters or may remain unassigned to any cluster. The actual similarity computation is performed using a *Markovian* similarity measure. This measure will be described later.

3. *Cluster elimination:* Many clusters are highly overlapping because of assignment of sequences to multiple clusters. It is desired to restrict this overlap to reduce redundancy in the clustering. If the number of sequences that are *unique* to a particular cluster is less than a predefined threshold, then such a cluster is eliminated.

The only step that remains to be described is the computation of the *Markovian similarity measure* between sequences and clusters. The idea is that if a sequence of symbols $S = s_1 s_2 \ldots s_n$ is similar to a cluster $\mathcal{C}_i$, then it should be "easy" to generate S using the

[2]The original *CLUSEQ* algorithm also adjusts the similarity threshold t iteratively to optimize results.

conditional distribution of the symbols inside the cluster. Then, the probability $P(S|\mathcal{C}_i)$ is defined as follows:

$$P(S|\mathcal{C}_i) = P(s_1|\mathcal{C}_i) \cdot P(s_2|s_1, \mathcal{C}_i) \ldots P(s_n|s_1 \ldots s_{n-1}, \mathcal{C}_i) \tag{15.3}$$

This is the generative probability of the sequence S for cluster $\mathcal{C}_i$. Intuitively, the term $P(s_j|s_1 \ldots s_{j-1}, \mathcal{C}_i)$ represents the fraction of times that s_j follows $s_1 \ldots s_{j-1}$ in cluster $\mathcal{C}_i$. This term can be estimated in a data-driven manner from the sequences in $\mathcal{C}_i$. When a cluster is highly similar to a sequence, this value will be high. A *relative* similarity can be computed by comparing with a sequence generation model in which all symbols are generated randomly in proportion to their presence in the full data set. The probability of such a random generation is given by $\prod_{j=1}^{n} P(s_j)$, where $P(s_j)$ is estimated as the fraction of sequences containing symbol s_j. Then, the similarity of S to cluster $\mathcal{C}_i$ is defined as follows:

$$sim(S, \mathcal{C}_i) = \frac{P(S|\mathcal{C}_i)}{\prod_{j=1}^{n} P(s_j)} \tag{15.4}$$

One issue is that many parts of the sequence S may be noisy and not match the cluster well. Therefore, the similarity is computed as the maximum similarity of any contiguous segment of S to $\mathcal{C}_i$. In other words, if S_{kl} be the contiguous segment of S from positions k to l, then the final similarity $SIM(S, \mathcal{C}_i)$ is computed as follows:

$$SIM(S, \mathcal{C}_i) = \max_{1 \leq k \leq l \leq n} sim(S_{kl}, \mathcal{C}_i) \tag{15.5}$$

The maximum similarity value can be computed by computing $sim(S_{kl}, \mathcal{C}_i)$ over all pairs $[k, l]$. This is the similarity value used for assigning sequences to their relevant clusters.

One problematic issue is that the computation of each of the terms $P(s_j|s_1 \ldots s_{j-1}, \mathcal{C}_i)$ on the right-hand side of Eq. 15.3 may require the examination of all the sequences in the cluster $\mathcal{C}_i$ for probability estimation purposes. Fortunately, these terms can be estimated efficiently using a data structure, referred to as *Probabilistic Suffix Trees*. The *CLUSEQ* algorithm always dynamically maintains the *Probabilistic Suffix Trees (PST)* whenever new clusters are created or sequences are added to clusters. This data structure will be described in detail in Sect. 15.4.1.1.

15.3.4.2 Mixture of Hidden Markov Models

This approach can be considered the string analog of the probabilistic models discussed in Sect. 6.5 of Chap. 6 for clustering multidimensional data. Recall that a generative mixture model is used in that case, where each component of the mixture has a Gaussian distribution. A Gaussian distribution is, however, appropriate only for generating numerical data, and is not appropriate for generating sequences. A good generative model for sequences is referred to as *Hidden Markov Models (HMM)*. The discussion of this section will assume the use of HMM as a black box. The actual details of HMM will be discussed in a later section. As we will see later in Sect. 15.5, the HMM can itself be considered a kind of mixture model, in which states represent dependent components of the mixture. Therefore, this approach can be considered a *two-level* mixture model. The discussion in this section should be combined with the description of HMMs in Sect. 15.5 to provide a complete picture of HMM-based clustering.

The broad principle of a mixture-based generative model is to assume that the data was generated from a mixture of k distributions with the probability distributions $\mathcal{G}_1 \ldots \mathcal{G}_k$, where each $\mathcal{G}_i$ is a Hidden Markov Model. As in Sect. 6.5 of Chap. 6, the approach assumes the use of prior probabilities $\alpha_1 \dots \alpha_k$ for the different components of the mixture. Therefore, the generative process is described as follows:

1. Select one of the k probability distributions with probability α_i where $i \in \{1 \dots k\}$. Let us assume that the rth one is selected.

2. Generate a sequence from $\mathcal{G}_r$, where $\mathcal{G}_r$ is a Hidden Markov Model.

One nice characteristic of mixture models is that the change in the data type and corresponding mixture distribution does not affect the broader framework of the algorithm. The analogous steps can be applied in the case of sequence data, as they are applied in multidimensional data. Let S_j represent the jth sequence and Θ be the entire set of parameters to be estimated for the different HMMs. Then, the E-step and M-step are exactly analogous to the case of the multidimensional mixture model.

1. (E-step) Given the current state of the trained HMM and priors α_i, determine the posterior probability $P(\mathcal{G}_i|S_j, \Theta)$ of each sequence S_j using the HMM generative probabilities $P(S_j|\mathcal{G}_i, \Theta)$ of S_j from the ith HMM, and priors $\alpha_1 \dots \alpha_k$ in conjunction with the Bayes rule. This is the posterior probability that the sequence S_j was generated by the ith HMM.

2. (M-step) Given the current probabilities of assignments of data points to clusters, use the *Baum–Welch algorithm* on each HMM to learn its parameters. The assignment probabilities are used as weights for averaging the estimated parameters. The Baum–Welch algorithm is described in Sect. 15.5.4 of this chapter. The value of each α_i is estimated to be proportional to average assignment probability of all sequences to cluster i. Thus, the M-step results in the estimation of the entire set of parameters Θ.

Note that there is an almost exact correspondence in the steps used here, and to those used for mixture modeling in Sect. 6.5 of Chap. 6. The major drawback of this approach is that it can be rather slow. This is because the process of training each HMM is computationally expensive.

15.4 Outlier Detection in Sequences

Outlier detection in sequence data shares a number of similarities with timeseries data. The main difference between sequence data and timeseries data is that sequence data is discrete, whereas timeseries data is continuous. The discussion in the previous chapter showed that time series outliers can be either *point outliers*, or *shape outliers*. Because sequence data is the discrete analog of timeseries data, an identical principle can be applied to sequence data. Sequence data outliers can be either *position outliers* or *combination outliers*.

1. *Position outliers:* In position-based outliers, the values at specific positions are predicted by a model. This is used to determine the *deviation* from the model and predict specific *positions* as outliers. Typically, Markovian methods are used for predictive outlier detection. This is analogous to deviation-based outliers discovered in timeseries data with the use of regression models. Unlike regression models, Markovian models are better suited to discrete data. Such outliers are referred to as *contextual* outliers because they are outliers in the *context* of their immediate temporal neighborhood.

2. *Combination outliers:* In combination outliers, an entire test sequence is deemed to be unusual because of the combination of symbols in it. This could be the case because this combination may rarely occur in a sequence database, or its distance (or similarity) to most other subsequences of similar size may be very large (or small). More complex models, such as Hidden Markov Models, can also be used to model the frequency of presence in terms of generative probabilities. For a longer test sequence, smaller subsequences are extracted from it for testing, and then the outlier score of the entire sequence is predicted as a combination of these values. This is analogous to the determination of unusual shapes in timeseries data. Such outliers are referred to as *collective outliers* because they are defined by combining the patterns from multiple data items.

The following section will discuss these different types of outliers.

15.4.1 Position Outliers

In the case of continuous timeseries data discussed in the previous chapter, an important class of outliers was designed by determining significant deviations from *expected* values at timestamps. Thus, these methods intimately combine the problems of *forecasting* and *deviation-detection*. A similar principle applies to discrete sequence data, in which the discrete positions at specific timestamps can be predicted with the use of different models. When a position has very low probability of matching its forecasted value, it is considered an outlier. For example, consider an RFID application, in which event sequences are associated with product items in a superstore with the use of semantic extraction from RFID tags. A typical example of a normal event sequence is as follows:

```
PlacedOnShelf, RemovedFromShelf, CheckOut, ExitStore.
```

On the other hand, in a shoplifting scenario, the event sequence may be *unusually* different. An example of an event sequence in the shoplifting scenario is as follows:

```
PlacedOnShelf, RemovedFromShelf, ExitStore.
```

Clearly, the sequence symbol *ExitStore* is anomalous in the second case but not in the first case. This is because it does not depict the *expected* or *forecasted* value for that position in the second case. It is desirable to detect such anomalous *positions* on the basis of expected values. Such anomalous positions may appear *anywhere* in the sequence and not necessarily in the last element, as in the aforementioned example. The basic problem definition for position outlier detection is as follows:

Definition 15.4.1 *Given a set of N training sequences $\mathcal{D} = T_1 \dots T_N$, and a test sequence $V = a_1 \dots a_n$, determine if the position a_i in the test sequence should be considered an anomaly based on its expected value.*

Some formulations do not explicitly distinguish between training and test sequences. This is because a sequence can be used for both model construction and outlier analysis when it is very long.

Typically, the position a_i can be predicted in temporal domains only from the positions before a_i, whereas in other domains, such as biological data, both directions may be relevant. The discussion below will assume the temporal scenario, though generalization to the placement scenario (as in biological data) is straightforward by examining windows on both sides of the position.

Just as regression modeling of continuous streams uses small windows of past history, discrete sequence prediction also uses small windows of the symbols. It is assumed that the prediction of the values at a position depends upon this short history. This is known as the *short memory property* of discrete sequences, and it generally holds true across a wide variety of temporal application domains.

Definition 15.4.2 (Short Memory Property) *For a sequence of symbols $V = a_1 \ldots a_i \ldots$, the value of the probability $P(a_i|a_1 \ldots a_{i-1})$ is well approximated by $P(a_i|a_{i-k} \ldots a_{i-1})$ for some small value of k.*

After the value of $P(a_i|a_{i-k} \ldots a_{i-1})$ is estimated, a position in a test sequence can be flagged as an outlier, if it has very low probability on the basis of the models derived from the training sequences. Alternatively, if a different symbol (than one present in the test sequence) is predicted with very high probability, then that position can be flagged as an outlier.

This section will discuss the use of *Markovian* models for the position outlier detection problem. This model exploits the *short memory* property of sequences to explicitly model the sequences as a set of states in a Markov Chain. These models represent the sequence-generation process with the use of transitions in a Markov Chain defined on the alphabet Σ. This is a special kind of *Finite State Automaton*, in which the states are defined by a short (immediately preceding) history of the sequences generated. Such models correspond to a set of states A that represent the different kinds of memory about the system events. For example, in *first-order* Markov Models, each state represents the last symbol from the alphabet Σ that was generated in the sequence. In kth order Markov Models, each state corresponds to the subsequence of the last k symbols $a_{n-k} \ldots a_{n-1}$ in the sequence. Each transition in this model represents an event a_n, the transition probability of which from the state $a_{n-k} \ldots a_{n-1}$ to the state $a_{n-k+1} \ldots a_n$ is given by the conditional probability $P(a_n|a_{n-k} \ldots a_{n-1})$. A Markov Model can be depicted as a set of nodes representing the states and a set of edges representing the events that cause movement from one state to another. The probability of an edge provides the conditional probability of the corresponding event. Clearly, the order of the model encodes the memory length of the string segment retained for the modeling process. First-order models correspond to the least amount of retained memory.

To understand how Markov Models work, the previous example of tracking items with RFID tags will be revisited. The actions performed an item can be viewed as a sequence drawn on the alphabet $\Sigma = \{P, R, C, E\}$. The semantic meaning of each of these symbols is illustrated in Fig. 15.4. A state of an order-k Markov model corresponds to the previous k (action) symbols of the sequence drawn on the alphabet $\Sigma = \{P, R, C, E\}$. Examples of different states along with transitions are illustrated in Fig. 15.4. Both a first-order and a second-order model have been illustrated in the figure. The edge transition probabilities are also illustrated in the figure. These are typically estimated from the training data. The transitions that correspond to the shoplifting anomaly are marked in both models. In each case, it is noteworthy that the corresponding transition probability for the actual shoplifting event is very low. This is a particularly simple example, in which a memory of one event is sufficient to completely represent the state of an item. This is not the case in general. For example, consider the case of a Web log in which the Markov Models correspond to sequences of Web pages visited by users. In such a case, the probability distribution of the next Web page visited depends not just on the last page visited, but also on the other preceding visits by the user.

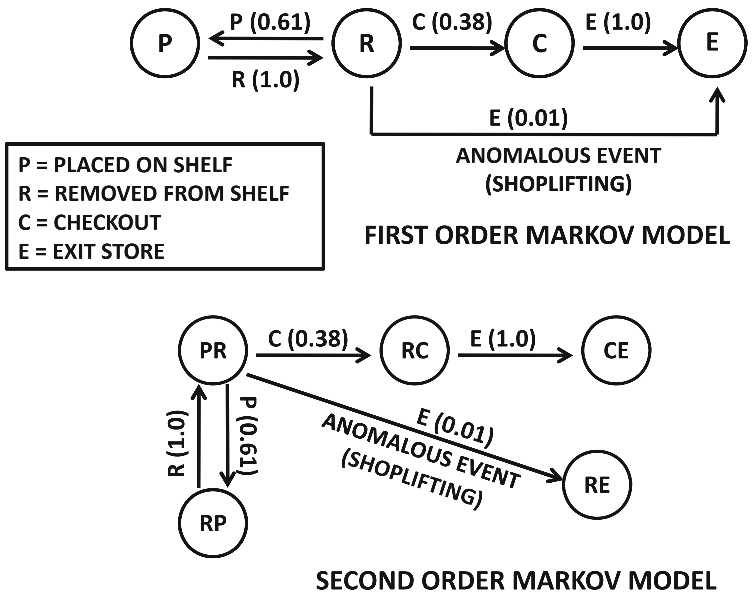


Figure 15.4: Markov model for the RFID-based shoplifting anomaly

An observation from Fig. 15.4 is that the number of states in the second-order model is larger than that in the first-order model. This is not a coincidence. As many as $|\Sigma|^k$ states may exist in an order-k model, though this provides only an upper bound. Many of the subsequences corresponding to these states may either not occur in the training data, or may be invalid in a particular application. For example, a PP state would be invalid in the example of Fig. 15.4 because the same item cannot be sequentially placed twice on the shelf without removing it at least once. Higher-order models represent complex systems more accurately at least at a theoretical level. However, choosing models of a higher order degrades the efficiency and may also result in overfitting.

15.4.1.1 Efficiency Issues: Probabilistic Suffix Trees

It is evident from the discussion in the previous sections that the Markovian and rule-based models are equivalent, with the latter being a simpler and easy-to-understand heuristic approximation of the former. Nevertheless, in both cases, the challenge is that the number of possible antecedents of length k can be as large as $|\Sigma|^k$. This can make the methods slow, when a lookup for a test subsequence $a_{i-k} \dots a_{i-1}$ is required to determine the probability of $P(a_i|a_{i-k} \dots a_{i-1})$. It is expensive to either compute these values on the fly, or even to retrieve their precomputed values, if they are not organized properly. The *Probabilistic Suffix Tree (PST)* provides an efficient approach for retrieval of such precomputed values. The utility of probabilistic suffix trees is not restricted to outlier detection, but is also applicable to clustering and classification. For example, the *CLUSEQ* algorithm, discussed in Sect. 15.3.4.1, uses PST to retrieve these prestored probability values.

Suffix trees are a classical data structure that store all subsequences in a given database. Probabilistic suffix trees represent a generalization of this structure that also stores the conditional probabilities of generation of the next symbol for a given sequence database. For order-k Markov Models, a suffix tree of depth at most k will store all the required conditional

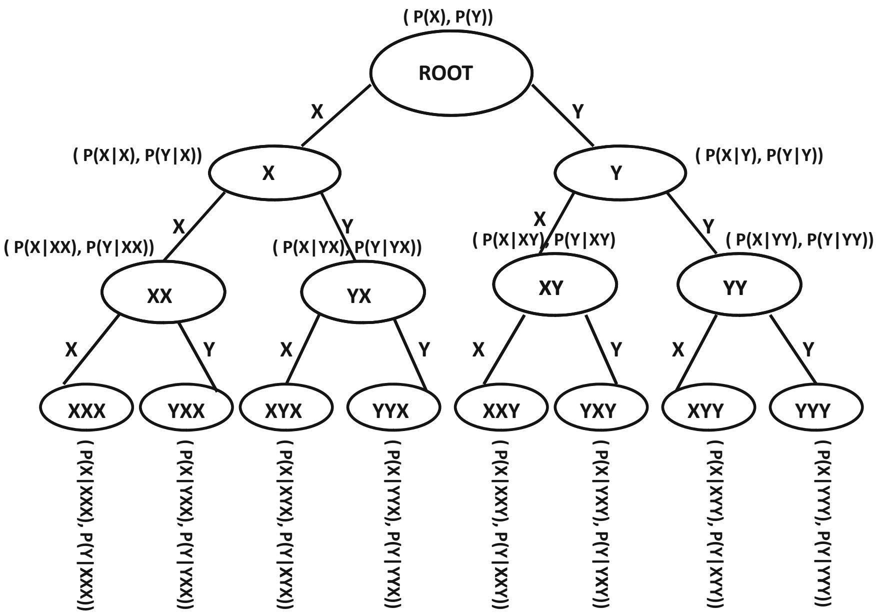


Figure 15.5: Probabilistic suffix tree

probability values for the kth order Markovian models, including the conditionals for all lower-order Markov Models. Therefore, such a structure encodes all the information required for variable-order Markov Models as well. A key challenge is that the number of nodes in such a suffix tree can be as large as $\sum_{i=0}^{k} |\Sigma|^i$, an issue that needs to be addressed with selective pruning.

A probabilistic suffix tree is a hierarchical data structure representing the different suffixes of a sequence. A node in the tree with depth k represents a suffix of length k and is, therefore, labeled with a sequence of length k. The parent of a node $a_{i-k} \dots a_i$ corresponds to the sequence $a_{i-k+1} \dots a_i$. The latter is obtained by removing the *first* symbol from the former. Each edge is labeled with the symbol that needs to be removed to derive the sequence at the parent node. Thus, a path in the tree corresponds to *suffixes* of the same sequence. Each node also maintains a vector Σ of probabilities that correspond to the conditional probability of the generation of any symbol from $\Sigma = \{\sigma_1 \dots \sigma_{|\Sigma|}\}$ after that sequence. Therefore, for a node corresponding to the sequence $a_{i-k} \dots a_i$, and for each $j \in \{1 \dots |\Sigma|\}$, the values of $P(\sigma_j | a_{i-k} \dots a_i)$ are maintained. As discussed earlier, this corresponds to the conditional probability that σ_j appears immediately *after* $a_{i-k} \dots a_i$, once the latter sequence has already been observed. This provides the generative probability crucial to the determination of position outliers. Note that this generative probability is also useful for other algorithms such as the *CLUSEQ* algorithm discussed earlier in this chapter.

An example of a suffix tree with the symbol set $\Sigma = \{X, Y\}$ is illustrated in Fig. 15.5. The two possible symbol-generation probabilities at each node corresponding to either of the symbols X and Y are placed next to the corresponding nodes. It is also evident that a probabilistic suffix tree of depth k encodes *all* the transition probabilities for Markovian models up to order k. Therefore, such an approach can be used for higher-order Markovian models.

The probabilistic suffix true is pruned significantly to improve its compactness. For example, suffixes that correspond to very low counts in the original data can be pruned from consideration. Furthermore, nodes with low generative probabilities of their underlying sequences can be pruned from consideration. The generative probability of a sequence $a_1 \dots a_n$ is approximated as follows:

$$P(a_1 \dots a_n) = P(a_1) \cdot P(a_2|a_1) \dots P(a_n|a_1 \dots a_{n-1}) \tag{15.6}$$

For Markovian models of order $k < n$, the value of $P(a_r|a_1 \dots a_{r-1})$ in the equation above is approximated by $P(a_r|a_{r-k} \dots a_{r-1})$ for any value of k less than r. To create Markovian models of order k or less, it is not necessary to keep portions of the tree with depth greater than k.

Consider the sequence $a_1 \dots a_i \dots a_n$, in which it is desired to test whether position a_i is a position outlier. Then, it is desired to determine $P(a_i|a_1 \dots a_{i-1})$. It is possible that the suffix $a_1 \dots a_{i-1}$ may not be present in the suffix tree because it may have been pruned from consideration. In such cases, the *short memory* property is used to determine the longest suffix $a_j \dots a_{i-1}$ present in the suffix tree, and the corresponding probability is estimated by $P(a_i|a_j \dots a_{i-1})$. Thus, the probabilistic suffix tree provides an efficient way to store and retrieve the relevant probabilities. The length of the longest path that exists in the suffix tree containing a nonzero probability estimate of $P(a_i|a_j \dots a_{i-1})$ also provides an idea of the level of rarity of this particular sequence of events. Positions that contain only short paths preceding them in the suffix tree are more likely to be outliers. Thus, outlier scores may be defined from the suffix tree in multiple ways:

1. If only short path lengths exist in the (pruned) suffix tree corresponding to a position a_i and its preceding history, then that position is more likely be an outlier.

2. For the paths of lengths $1 \dots r$ that do exist in the suffix tree for position a_i, a combination score may be used based on the models of different orders. In some cases, only lower-order scores are combined. In general, the use of lower-order scores is preferable, since they are usually more robustly represented in the training data.

15.4.2 Combination Outliers

In combination outliers, the goal is to determine unusual combinations of symbols in the sequences. Consider a setting, where a set of training sequences is provided, together with a test sequence. It is desirable to determine whether a test sequence is an anomaly, based on the "normal" patterns in the training sequences. In many cases, the test sequences may be quite long. Therefore, the combination of symbols in the full sequence may be unique with respect to the training sequences. This means that it is hard to characterize "normal" sequences on the basis of the full sequence. Therefore, small windows are extracted from the training and test sequences for the purpose of comparison. Typically, all windows (including overlapping ones) are extracted from the sequences, though it is also possible to work with nonoverlapping windows. These are referred to as *comparison units.* The anomaly scores are defined with respect to these comparison units. Thus, unusual *windows* in the sequences are reported. The following discussion will focus exclusively on determining such unusual windows.

Some notations and definitions will be used to distinguish between the training database, test sequence, and the comparison units.

1. The training database is denoted by $\mathcal{D}$, and contains sequences denoted by $T_1 \dots T_N$.

2. The test sequence is denoted by V.

3. The comparison units are denoted by $U_1 \dots U_r$. Typically, each U_i is derived from small, contiguous windows of V. In domain-dependent cases, $U_1 \dots U_r$ may be provided by the user.

The model may be a distance-based, or frequency-based or may be a Hidden Markov Model. Each of these will be discussed in subsequent sections. Because Hidden Markov Models are general constructs that are used for different problems such as clustering, classification, and outlier detection, they will be discussed in a section of their own, immediately following this section.

15.4.2.1 Distance-Based Models

In distance-based models, the absolute distance (or similarity) of the comparison unit is computed to equivalent windows of the training sequence. The distance of the k-th nearest neighbor window in the training sequence is used to determine the anomaly score. In the context of sequence data, many proximity-functions are *similarity* functions rather than *distance* functions. In the former case, higher values indicate greater proximity. Some common methods for computing the similarity between a pair of sequences are as follows:

1. *Simple matching coefficient:* This is the simplest possible function and determines the number of matching positions between two sequences of equal length. This is also equivalent to the Hamming distance between a pair of sequences.

2. *Normalized longest common subsequence:* The longest common subsequence can be considered the sequential analog of the cosine distance between two ordered sets. Let T_1 and T_2 be two sequences, and the length of (unnormalized) longest common subsequence between T_1 and T_2 be denoted by $L(T_1, T_2)$. The unnormalized longest common subsequence can be computed using methods discussed in Sect. 3.4.2 of Chap. 3. Then, the value $NL(T_1, T_2)$ of the normalized longest common subsequence is computed by normalizing $L(T_1, T_2)$ with the underlying sequence lengths in a way similar to the cosine computation between unordered sets:

$$NL(T_1, T_2) = \frac{L(T_1, T_2)}{\sqrt{|T_1|} \cdot \sqrt{|T_2|}} \tag{15.7}$$

   The advantage of this approach is that it can match two sequences of unequal lengths. The drawback is that the computation process is relatively slow.

3. *Edit distance:* The edit distance is one of the most common similarity functions used for sequence matching. This similarity function is discussed in Chap. 3. This function measures the distance between two sequences by the minimum number of edits required to transform one sequence to the other. The computation of the edit distance can be computationally very expensive.

4. *Compression-based dissimilarity:* This measure is based on principles of information theory. Let W be a window of the training data, and $W \oplus U_i$ be the string representing the concatenation of W and U_i. Let $DL(S) < |S|$ be the description length of any string S after applying a standard compression algorithm to it. Then, the compression-based dissimilarity $CD(W, U_i)$ is defined as follows:

$$CD(W, U_i) = \frac{DL(W \oplus U_i)}{DL(W) + DL(U_i)} \tag{15.8}$$

This measure always lies in the range $(0, 1)$, and lower values indicate greater similarity. The intuition behind this approach is that when the two sequences are very similar, the description length of the combined sequence will be much smaller than that of the sum of the description lengths. On the other hand, when the sequences are very different, the description length of the combined string will be almost the same as the sum of the description lengths.

To compute the anomaly score for a comparison unit U_i with respect to the training sequences in $T_1 \dots T_N$, the first step is to extract equivalent windows from $T_1 \dots T_N$ as the size of the comparison unit. The k-th nearest neighbor distance is used as the anomaly score for that window. The unusual windows may be reported, or the scores from different windows may be consolidated into a single anomaly score.

15.4.2.2 Frequency-Based Models

Frequency-based models are typically used with domain-specific comparison units specified by the user. In this case, the relative frequency of the comparison unit needs to be measured in the training sequences and the test sequences, and the level of surprise is correspondingly determined.

When the comparison units are specified by the user, a natural way of determining the anomaly score is to test the frequency of the comparison unit U_j in the training and test patterns. For example, when a sequence contains a hacking attempt, such as a sequence of *Login* and *Password* events, this sequence will have much higher frequency in the test sequence, as compared to the training sequences. The specification of such relevant comparison units by a user provides very useful domain knowledge to an outlier analysis application.

Let $f(T, U_j)$ represent the number of times that the comparison unit U_j occurs in the sequence T. Since the frequency $f(T, U_j)$ depends on the length of T, the normalized frequency $\hat{f}(T, U_j)$ may be obtained by dividing the frequency by the length of the sequence:

$$\hat{f}(T, U_j) = \frac{f(T, U_j)}{|T|}$$

Then, the anomaly score of the training sequence T_i with respect to the test sequence V is defined by subtracting the relative frequency of the training sequence from the test sequence. Therefore, the anomaly score $A(T_i, V, U_j)$ is defined as follows:

$$A(T_i, V, U_j) = \hat{f}(V, U_j) - \hat{f}(T_i, U_j)$$

The absolute value of the average of these scores is computed over all the sequences in the database $\mathcal{D} = T_1 \dots T_N$. This represents the final anomaly score.

A useful output of this approach is the specific subset of comparison units specified by the user that are the most anomalous. This provides intensional knowledge and feedback to the analyst about *why* a particular test sequence should be considered anomalous. A method called *TARZAN* uses suffix tree representations to efficiently determine all the anomalous subsequences in a comparative sense between a test sequence and a training sequence. Readers are referred to the bibliographic notes for pointers to this method.

15.5 Hidden Markov Models

Hidden Markov Models (HMM) are probabilistic models that generate sequences through a sequence of transitions between states in a Markov chain. Hidden Markov Models are used for clustering, classification, and outlier detection. Therefore, the applicability of these models is very broad in sequence analysis. For example, the clustering approach in Sect. 15.3.4.2 uses Hidden Markov Models as a subroutine. This section will use outlier detection as a specific application of HMM to facilitate understanding. In Sect. 15.6.5, it will also be shown how HMM may be used for classification.

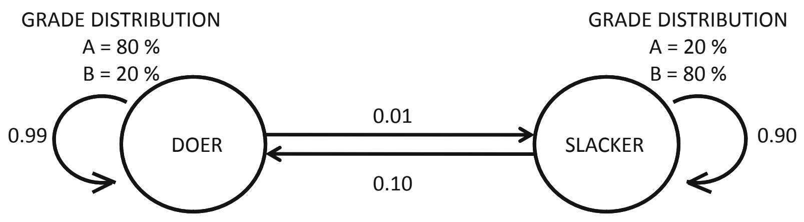
So how are Hidden Markov Models different from the Markovian techniques introduced earlier in this chapter? Each state in the Markovian techniques introduced earlier in this chapter is well defined and is based on the last k positions of the sequence. This state is also directly visible to the user because it is defined by the latest sequence combination of length k. Thus, the generative behavior of the Markovian model is always known *deterministically*, in terms of the correspondence between states and sequence positions for a *particular* input string.

In a Hidden Markov Model, the states of the system are *hidden* and not directly visible to the user. Only a sequence of (typically) discrete observations is visible to the user that is generated by symbol emissions from the states after each transition. The generated sequence of symbols corresponds to the application-specific sequence data. In many cases, the states may be defined (during the modeling process) on the basis of an *understanding* of how the underlying system behaves, though the precise sequence of transitions may not be known to the analyst. This is why such models are referred to as "*hidden.*"

Each state in an HMM is associated with a *set of emission probabilities* over the symbol Σ. In other words, a visit to the state j leads to an emission of one of the symbols $\sigma_i \in \Sigma$ with probability $\theta^j(\sigma_i)$. Correspondingly, a sequence of transitions in an HMM corresponds to an *observed data sequence*. Hidden Markov Models may be considered a kind of mixture model of the type discussed in Chap. 6, in which the different components of the mixture are not independent of one another, but are related through sequential transitions. Thus, each state is analogous to a component in the multidimensional mixture model discussed in Chap. 6. Each symbol generated by this model is analogous to a data point generated by the multidimensional mixture model. Furthermore, unlike multidimensional mixture models, the successive generation of individual data items (sequence symbols) are also not independent of one another. This is a natural consequence of the fact that the successive states emitting the data items are dependent on one another with the use of probabilistic transitions. Unlike multidimensional mixture models, Hidden Markov Models are designed for sequential data that exhibits temporal correlations.

To better explain Hidden Markov Models, an illustrative example will be used for the specific problem of using HMMs for anomaly detection. Consider the scenario where a set of students register for a course and generate a sequence corresponding to the grades received in each of their weekly assignments. This grade is drawn from the symbol set $\Sigma = \{A, B\}$. *The model created by the analyst is that* the class contains students who, at any given time, are either *doers* or *slackers* with different grade-generation probabilities. A student in a *doer* state may sometimes transition to a *slacker* state and vice versa. These represent the two states in the system. Weekly home assignments are handed out to each student and are graded with one of the symbols from Σ. This results in a *sequence* of grades for each student, and it represents the only *observable* output for the analyst. The state of a student represents only a *model* created by the analyst to *explain* the grade sequences and is, therefore, not observable in of itself. It is important to understand that if this model is a poor reflection of the true generative process, then it will impact the quality of the learning process.

Assume that a student in a *doer* state is likely to receive an A grade in a weekly assignment with 80% probability and a B with 20% probability. For *slacker*s, these probability



SOME EXAMPLES OF STUDENT GRADE SEQUENCES

AAABAAAABAAAA	DOER (VERY COMMON)
BBBBABBBABBBB	SLACKER (LESS COMMON)
AAABAAABBABBB	DOER TURNS SLACKER (VERY RARE)
ABABABABABABA	UNPREDICTABLE (EXTREMELY RARE)

Figure 15.6: Generating grade sequences from a Hidden Markov Model

values are reversed. Although these probabilities are explicitly specified here for illustrative purposes, they need to be *learned* or *estimated* from the observed grade sequences for the different students and are not known *a priori*. The precise status (state) of any student in a given week is not known to the analyst at any given time. These grade sequences are, in fact, the only *observable* outputs for the analyst. Therefore, from the perspective of the analyst, this is a *Hidden* Markov Model, which generates the sequences of grades from an *unknown* sequence of states, representing the state transitions of the students. The precise sequence of transitions between the states can be only *estimated* for a particular observed sequence.

The two-state Hidden Markov Model for the aforementioned example is illustrated in Fig. 15.6. This model contains two states, denoted by *doer* and *slacker*, that represent the state of a student in a particular week. It is possible for a student to transition from one state to another each week, though the likelihood of this is rather low. It is assumed that set of initial state probabilities governs the *a priori* distribution of *doer*s and *slacker*s. This distribution represents the a priori understanding about the students when they join the course. Some examples of *typical sequences* generated from this model, along with their rarity level, are illustrated in Fig. 15.6. For example, the sequence `AAABAAAABAAAA` is most likely generated by a student who is consistently in a *doer* state, and the sequence `BBBBABBBABBBB` is most likely generated by a student who is consistently in *slacker* state. The second sequence is typically rarer than the first because the population mostly contains[3] *doers*. The sequence `AAABAAABBABBB` corresponds to a *doer* who eventually transitions into a *slacker*. This case is even rarer because it requires a transition from the *doer* state to a *slacker* state, which has a very low probability. The sequence `ABABABABABABA` is *extremely anomalous* because it does not represent temporally consistent *doer* or *slacker* behavior that is implied by the model. Correspondingly, such a sequence has very low probability of fitting the model.

A larger number of states in the Markov Model can be used to encode more complex scenarios. It is possible to encode domain knowledge with the use of states that describe different generating scenarios. In the example discussed earlier, consider the case that *doer*s sometimes slacks off for short periods and then return to their usual state. Alternatively,

[3]The assumption is that the initial set of state probabilities are approximately consistent with the steady state behavior of the model for the particular set of transition probabilities shown in Fig. 15.6.

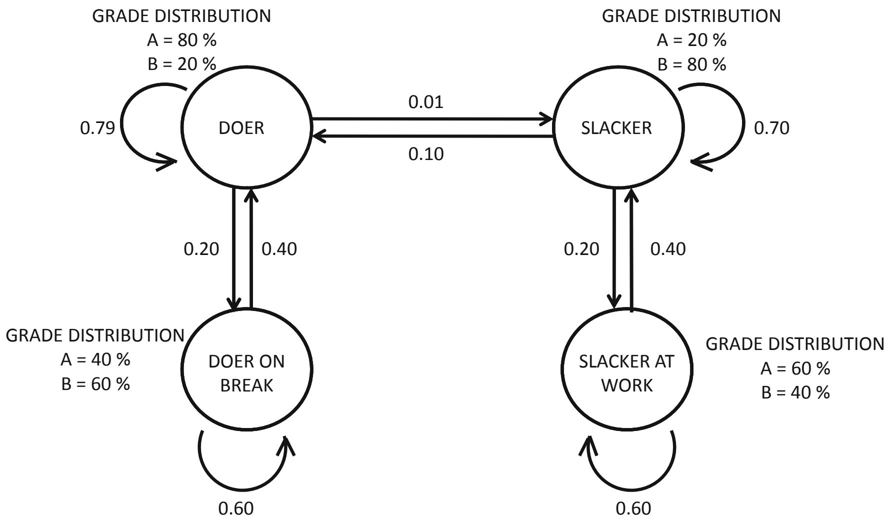


Figure 15.7: Extending the model in Fig. 15.6 with two more states provides greater expressive power for modeling sequences

*slacker*s may sometimes become temporarily inspired to be *doer*s, but may eventually return to what they are best at. Such episodes will result in local portions of the sequence that are distinctive from the remaining sequence. These scenarios can be captured with the four-state Markov Model illustrated in Fig. 15.7. The larger the number of states, the more complex the scenarios that can be captured. Of course, more training data is required to learn the (larger number of) parameters of such a model, or this may result in overfitting. For smaller data sets, the transition probabilities and symbol-generation probabilities are not estimated accurately.

15.5.1 Formal Definition and Techniques for HMMs

In this section, Hidden Markov Models will be formally introduced along with the associated training methods. It is assumed that a Hidden Markov Model contains n states denoted by $\{s_1 \dots s_n\}$. The symbol set from which the observations are generated is denoted by $\Sigma = \{\sigma_1 \dots \sigma_{|\Sigma|}\}$. The symbols are generated from the model by a sequence of transitions from one state to the other. Each visit to a state (including self-transitions) generates a symbol drawn from a categorical[4] probability distribution on Σ. The symbol emission distribution is specific to each state. The probability $P(\sigma_i|s_j)$ that the symbol σ_i is generated from state s_j is denoted by $\theta^j(\sigma_i)$. The probability of a transition from state s_i to s_j is denoted by p_{ij}. The initial state probabilities are denoted by $\pi_1 \dots \pi_n$ for the n different states. The topology of the model can be expressed as a network $G = (M, A)$, in which M is the set of states $\{s_1 \dots s_n\}$. The set A represents the possible transitions between the states. In the most common scenario, where the architecture of the model is constructed with a domain-specific understanding, the set A is not the complete network. In cases where domain-specific knowledge is not available, the set A may correspond to the complete

[4]HMMs can also generate continuous time series, though they are less commonly used in timeseries analysis.

network, including self-transitions. *The goal of training the HMM model is to learn the initial state probabilities, transition probabilities, and the symbol emission probabilities from the training database* $\{T_1 \dots T_N\}$. Three methodologies are commonly leveraged in creating and using a Hidden Markov Model:

- *Training:* Given a set of training sequences $T_1 \dots T_N$, estimate the model parameters, such as the initial probabilities, transition probabilities, and symbol emission probabilities with an Expectation-Maximization algorithm. The Baum–Welch algorithm is used for this purpose.

- *Evaluation:* Given a test sequence V (or comparison unit U_i), determine the probability that it fits the HMM. This is used to determine the anomaly scores. A recursive forward algorithm is used to compute this.

- *Explanation:* Given a test sequence V, determine the most likely sequence of states that generated this test sequence. This is helpful for providing an understanding of why a sequence should be considered an anomaly (in outlier detection) or belong to a specific class (in data classification). The idea is that the states correspond to an intuitive understanding of the underlying system. In the example of Fig. 15.6, it would be useful to know that an observed sequence is an anomaly because of the unusual oscillation of a student between *doer* and *slacker* states. This can provide the *intensional knowledge* for understanding the state of a system. This most likely sequence of states is computed with the Viterbi algorithm.

Since the description of the training procedure relies on technical ideas developed for the evaluation method, we will deviate from the natural order of presentation and present the training algorithms last. The evaluation and explanation techniques will assume that the model parameters, such as the transition probabilities, are already available from the training phase.

15.5.2 Evaluation: Computing the Fit Probability for Observed Sequence

One approach for determining the fit probability of a sequence $V = a_1 \dots a_m$ would be to compute all the n^m possible sequences of states (paths) in the HMM, and compute the probability of each, based on the observed sequence, symbol-generation probabilities, and transition probabilities. The sum of these values can be reported as the fit probability. Obviously, such an approach is not practical because it requires the enumeration of an exponential number of possibilities.

This computation can be greatly reduced by recognizing that the fit probability of the first r symbols (and a fixed value of the rth state) can be recursively computed in terms of the corresponding fit probability of first $(r-1)$ observable symbols (and a fixed $(r-1)$th state). Specifically, let $\alpha_r(V, s_j)$ be the probability that the first r symbols in V are generated by the model, and the last state in the sequence is s_j. Then, the recursive computation is as follows:

$$\alpha_r(V, s_j) = \sum_{i=1}^{n} \alpha_{r-1}(V, s_i) \cdot p_{ij} \cdot \theta^j(a_r)$$

This approach recursively sums up the probabilities for all the n different paths for different penultimate nodes. The aforementioned relationship is iteratively applied for $r = 1 \dots m$. The probability of the first symbol is computed as $\alpha_1(V, s_j) = \pi_j \cdot \theta^j(a_1)$ for initializing the recursion. This approach requires $O(n^2 \cdot m)$ time. Then, the overall probability is computed by summing up the values of $\alpha_m(V, s_j)$ over all possible states s_j. Therefore, the final fit $F(V)$ is computed as follows:

$$F(V) = \sum_{j=1}^{n} \alpha_m(V, s_j)$$

This algorithm is also known as the *Forward Algorithm.* Note that the fit probability has a direct application to many problems, such as classification and anomaly detection, depending upon whether the HMM is constructed in supervised or unsupervised fashion. By constructing separate HMMs for each class, it is possible to test the better-fitting class for a test sequence. The fit probability is useful in problems such as data clustering, classification and outlier detection. In data clustering and classification, the fit probability can be used to model the probability of a sequence belonging to a cluster or class, by creating a group-specific HMM. In outlier detection, it is possible to determine poorly fitting sequences with respect to a global HMM and report them as anomalies.

15.5.3 Explanation: Determining the Most Likely State Sequence for Observed Sequence

One of the goals in many data mining problems is to provide an explanation for why a sequence fits part (e.g. class or cluster) of the data, or does not fit the whole data set (e.g. outlier). Since the sequence of (hidden) generating states often provides an intuitive explanation for the observed sequence, it is sometimes desirable to determine the *most likely sequence of states* for the observed sequence. The Viterbi algorithm provides an efficient way to determine the most likely state sequence.

One approach for determining the most likely state path of the test sequence $V = a_1 \dots a_m$ would be to compute all the n^m possible sequences of states (paths) in the HMM, and compute the probability of each of them, based on the observed sequence, symbol-generation probabilities, and transition probabilities. The maximum of these values can be reported as the most likely path. Note that this is a similar problem to the fit probability except that it is needed to determine the *maximum* fit probability, rather than the *sum* of fit probabilities, over all possible paths. Correspondingly, it is also possible to use a similar recursive approach as the previous case to determine the most likely state sequence.

Any subpath of an optimal state path must also be optimal for generating the corresponding subsequence of symbols. This property, in the context of an optimization problem of sequence selection, normally enables dynamic programming methods. The best possible state path for generating the first r symbols (with the rth state fixed to j) can be recursively computed in terms of the corresponding best paths for the first $(r-1)$ observable symbols and different penultimate states. Specifically, let $\delta_r(V, s_j)$ be the probability of the best state sequence for generating the first r symbols in V and also ending at state s_j. Then, the recursive computation is as follows:

$$\delta_r(V, s_j) = MAX_{i=1}^{n} \delta_{r-1}(V, s_i) \cdot p_{ij} \cdot \theta^j(a_r)$$

This approach recursively computes the maximum of the probabilities of all the n different paths for different penultimate nodes. The approach is iteratively applied for $r = 1 \dots m$. The first probability is determined as $\delta_1(V, s_j) = \pi_j \cdot \theta^j(a_1)$ for initializing the recursion. This approach requires $O(n^2 \cdot m)$ time. Then, the final best path is computed by using the maximum value of $\delta_m(V, s_j)$ over all possible states s_j. This approach is, essentially, a dynamic programming algorithm. In the anomaly example of student grades, an oscillation between *doer* and *slacker* states will be discovered by the Viterbi algorithm as the causality for outlier behavior. In a clustering application, a consistent presence in the *doer* state will explain the cluster of diligent students.

15.5.4 Training: Baum–Welch Algorithm

The problem of learning the parameters of an HMM is a very difficult one, and no known algorithm is guaranteed to determine the global optimum. However, options are available to determine a reasonably effective solution in most scenarios. The Baum–Welch algorithm is one such method. It is also known as the *Forward-backward* algorithm, and it is an application of the EM approach to the generative Hidden Markov Model. First, a description of training with the use of a single sequence $T = a_1 \dots a_m$ will be provided. Then, a straightforward generalization to N sequences $T_1 \dots T_N$ will be discussed.

Let $\alpha_r(T, s_j)$ be the *forward* probability that the first r symbols in a sequence T of length m are generated by the model, and the last symbol in the sequence is s_j. Let $\beta_r(T, s_j)$ be the *backward* probability that the portion of the sequence after *and not including the* r*th position* is generated by the model, *conditional on the fact that* the state for the rth position is s_j. Thus, the forward and backward probability definitions are not symmetric. The forward and backward probabilities can be computed from model probabilities in a way similar to the evaluation procedure discussed above in Sect. 15.5.2. The major difference for the backward probabilities is that the computations start from the end of the sequence in the backward direction. Furthermore, the probability value $\beta_{|T|}(T, s_j)$ is initialized to 1 at the bottom of the recursion to account for the difference in the two definitions. Two additional probabilistic quantities need to be defined to describe the EM algorithm:

- $\psi_r(T, s_i, s_j)$: Probability that the rth position in sequence T corresponds to state s_i, the $(r+1)$th position corresponds to s_j.

- $\gamma_r(T, s_i)$: Probability that the rth position in sequence T corresponds to state s_i.

The EM procedure starts with a random initialization of the model parameters and then iteratively estimates $(\alpha(\cdot), \beta(\cdot), \psi(\cdot), \gamma(\cdot))$ from the model parameters, and vice versa. Specifically, the iteratively executed steps of the EM procedure are as follows:

- (E-step) Estimate $(\alpha(\cdot), \beta(\cdot), \psi(\cdot), \gamma(\cdot))$ from currently estimated values of the model parameters $(\pi(\cdot), \theta(\cdot), p_{\cdot\cdot})$.

- (M-step) Estimate model parameters $(\pi(\cdot), \theta(\cdot), p_{\cdot\cdot})$ from currently estimated values of $(\alpha(\cdot), \beta(\cdot), \psi(\cdot), \gamma(\cdot))$.

It now remains to explain how each of the above estimations is performed. The values of $\alpha(\cdot)$ and $\beta(\cdot)$ can be estimated using the forward and backward procedures, respectively. The forward procedure is already described in the evaluation section, and the backward procedure is analogous to the forward procedure, except that it works backward from the end of the sequence. The value of $\psi_r(T, s_i, s_j)$ is equal to $\alpha_r(T, s_i) \cdot p_{ij} \cdot \theta^j(a_{r+1}) \cdot \beta_{r+1}(T, s_j)$ because the sequence-generation procedure can be divided into three portions corresponding to that up to position r, the generation of the $(r+1)$th symbol, and the portion after the

$(r+1)$th symbol. The estimated values of $\psi_r(T, s_i, s_j)$ are normalized to a probability vector by ensuring that the sum over different pairs $[i, j]$ is 1. The value of $\gamma_r(T, s_i)$ is estimated by summing up the values of $\psi_r(T, s_i, s_j)$ over fixed i and varying j. This completes the description of the E-step.

The re-estimation formulas for the model parameters in the M-Step are relatively straightforward. Let $I(a_r, \sigma_k)$ be a binary indicator function, which takes on the value of 1 when the two symbols are the same, and 0 otherwise. Then the estimations can be performed as follows:

$$\pi(j) = \gamma_1(T, s_j), \;\; p_{ij} = \frac{\sum_{r=1}^{m-1} \psi_r(T, s_i, s_j)}{\sum_{r=1}^{m-1} \gamma_r(T, s_i)}$$

$$\theta^i(\sigma_k) = \frac{\sum_{r=1}^{m} I(a_r, \sigma_k) \cdot \gamma_r(T, s_i)}{\sum_{r=1}^{m} \gamma_r(T, s_i)}$$

The precise derivations of these estimations, on the basis of expectation-maximization principles, may be found in [327]. This completes the description of the M-step.

As in all EM methods, the procedure is applied iteratively to convergence. The approach can be generalized easily to N sequences by applying the steps to each of the sequences, and averaging the corresponding model parameters in each step.

15.5.5 Applications

Hidden Markov Models can be used for a wide variety of sequence mining problems, such as clustering, classification, and anomaly detection. The application of HMM to clustering has already been described in Sect. 15.3.4.2 of this chapter. The application to classification will be discussed in Sect. 15.6.5 of this chapter. Therefore, this section will focus on the problem of anomaly detection.

In theory, it is possible to compute anomaly scores directly for the test sequence V, once the training model has been constructed from the sequence database $\mathcal{D} = T_1 \ldots T_N$. However, as the length of the test sequence increases, the robustness of such a model diminishes because of the increasing noise resulting from the curse of dimensionality. Therefore, the comparison units (either extracted from the test sequence or specified by the domain expert), are used for computing the anomaly scores of windows of the sequence. The anomaly scores of the different windows can then be combined together by using a simple function such as determining the number of anomalous window units in a sequence.

Some methods also use the Viterbi algorithm on the test sequence to mine the most likely state sequence. In some domains, it is easier to determine anomalies in terms of the state sequence rather than the observable sequence. Furthermore, low transition probabilities on portions of the state sequence provide anomalous localities of the observable sequence. The downside is that the most likely state sequence may have a very low probability of matching the observed sequence. Therefore, the estimated anomalies may not reflect the true anomalies in the data when an *estimated* state sequence is used for anomaly detection. The real utility of the Viterbi algorithm is in providing an *explanation* of the anomalous behavior of sequences in terms of the intuitively understandable states, rather than anomaly score quantification.

15.6 Sequence Classification

It is assumed that a set of N sequences, denoted by $S_1 \ldots S_N$, is available for building the training model. Each of these sequences is annotated with a class label drawn from $\{1 \ldots k\}$. This training data is used to construct a model that can predict the label of unknown test sequences. Many modeling techniques, such as nearest neighbor classifiers, rule-based methods, and graph-based methods, are common to timeseries and discrete sequence classification because of the temporal nature of the two data types.

15.6.1 Nearest Neighbor Classifier

The nearest neighbor classifier is used frequently for different data types, including discrete sequence data. The basic description of the nearest neighbor classifier for multidimensional data may be found in Sect. 10.8 of Chap. 10. For discrete sequence data, the main difference is in the similarity function used for nearest neighbor classification. Similarity functions for discrete sequences are discussed in Sects. 3.4.1 and 3.4.2 of Chap. 3. The basic approach is the same as in multidimensional data. For any test instance, its k-nearest neighbors in the training data are determined. The dominant label from these k-nearest neighbors is reported as the relevant one for the test instance. The optimal value of k may be determined by using leave-one-out cross-validation. The effectiveness of the approach is highly sensitive to the choice of the distance function. The main problem is that the presence of noisy portions in the sequences throw off global similarity functions. A common approach is to use keyword-based similarity in which n-grams are extracted from the string to create a vector-space representation. The nearest-neighbor (or any other) classifier can be constructed with this representation.

15.6.2 Graph-Based Methods

This approach is a *semisupervised* algorithm because it combines the knowledge in the training and test instances for classification. Furthermore, the approach is transductive because out-of-sample classification of test instances is generally not possible. Training and testing instances must be specified at the same time. The use of similarity graphs for semisupervised classification was introduced in Sect. 11.6.3 of Chap. 11. Graph-based methods can be viewed as general semisupervised meta-algorithms that can be used for any data type. The basic approach constructs a similarity graph from *both* the training and test instances. A graph $G = (V, A)$ is constructed, in which a node in V corresponds to each of the training and test instances. A subset of nodes in G is labeled. These correspond to instances in the training data, whereas the unlabeled nodes correspond to instances in the test data. Each node in V is connected to its k-nearest neighbors with an undirected edge in A. The similarity is computed using any of the distance functions discussed in Sects. 3.4.1 and 3.4.2 of Chap. 3. In the resulting network, a subset of the nodes are labeled, and the remaining nodes are unlabeled. The specified labels of nodes in N are then used to predict labels for nodes where they are unknown. This problem is referred to as *collective classification*. Numerous methods for collective classification are discussed in Sect. 19.4 of Chap. 19. The derived labels on the nodes are then mapped back to the data objects. As in the case of nearest-neighbor classification, the effectiveness of the approach is sensitive to the choice of distance function used for constructing the graph.

15.6.3 Rule-Based Methods

A major challenge in sequence classification is that many parts of the sequence may be noisy and not very relevant to the class label. In some cases, a short pattern of two symbols may be relevant to classification, whereas in other cases, a longer pattern of many symbols may be discriminative for classification. In some cases, the discriminative patterns may not even occur contiguously. This issue was discussed in the context of timeseries classification in Sect. 14.7.2.1 of Chap. 14. However, discrete sequences can be converted into binary timeseries sequences, with the use of binarization. These binary timeseries can be converted to multidimensional wavelet representations. This is described in detail in Sect. 2.2.2.6, and the description is repeated here for completeness.

The first step is to convert the discrete sequence to a *set* of (binary) time series, where the number of time series in this set is equal to the number of distinct symbols. The second step is to map each of these time series into a multidimensional vector using the wavelet transform. Finally, the features from the different series are combined to create a single multidimensional record. A rule-based classifier is constructed on this multidimensional representation.

To convert a sequence to a binary time series, one can create a binary string, in which each position value denotes whether or not a particular symbol is present at a position. For example, consider the following nucleotide sequence drawn on four symbols:

```
ACACACTGTGACTG
```

This series can be converted into the following set of four binary time series corresponding to the symbols A, C, T, and G, respectively.

```
10101000001000
01010100000100
00000010100010
00000001010001
```

A wavelet transformation can be applied to each of these series to create a multidimensional set of features. The features from the four different series can be appended to create a single numeric multidimensional record. After a multidimensional representation has been obtained, any rule-based classifier can be utilized. Therefore, the overall approach for data classification is as follows:

1. Generate wavelet representation of each of the N sequences to create N numeric multidimensional representations, as discussed above.

2. Discretize wavelet representation to create categorical representations of the timeseries wavelet transformation. Thus, each categorical attribute value represents a range of numeric values of the wavelet coefficients.

3. Generate a set of rules using any rule-based classifier described in Sect. 10.4 of Chap. 10. The patterns on the left-hand represent the patterns of different granularities defined by the combination of wavelet coefficients on the left-hand side.

When the rule set has been generated, it can be used to classify arbitrary test sequences by first transforming the test sequence to the same wavelet-based numeric multidimensional representation. This representation is used with the fired rules to perform the classification.

Such methods are discussed in Sect. 10.4 of Chap. 10. It is not difficult to see that this approach is a discrete version of the rule-based classification of time series, as presented in Sect. 14.7.2.1 of Chap. 14.

15.6.4 Kernel Support Vector Machines

Kernel support vector machines can construct classifiers with the use of kernel similarity between training and test instances. As discussed in Sect. 10.6.4 of Chap. 10, kernel support vector machines do not need the feature values of the records, as long as the kernel-based similarity $K(Y_i, Y_j)$ between any pair of data objects are available. In this case, these data objects are strings. Different kinds of kernels are very popular for string classification.

15.6.4.1 Bag-of-Words Kernel

In the bag-of-words kernel, the string is treated as a bag of alphabets, with a frequency equal to the number of alphabets of each type in the string. This can be viewed as the vector-space representation of a string. Note that a text document is also a string, with an alphabet size equal to the lexicon. Therefore, the transformation $\Phi(\cdot)$ can be viewed as almost equivalent to the vector-space transformation for a text document. If $\overline{V(Y_i)}$ be the vector-space representation of a string, then the kernel similarity is equal to the dot product between the corresponding vector space representations.

$$\begin{aligned}\Phi(Y_i) &= \overline{V(Y_i)}\\ K(Y_i, Y_j) &= \Phi(Y_i) \cdot \Phi(Y_j) = \overline{V(Y_i)} \cdot \overline{V(Y_j)}\end{aligned}$$

The main disadvantage of the kernel is that it loses all the positioning information between the alphabets. This can be an effective approach for cases where the alphabet size is large. An example is text, where the alphabet (lexicon) size is of a few hundred thousand words. However, for smaller alphabet sizes, the information loss can be too significant for the resulting classifier to be useful.

15.6.4.2 Spectrum Kernel

The bag-of-words kernel loses *all* the sequential information in the strings. The spectrum kernel addresses this issue by extracting k-mers from the strings and using them to construct the vector-space representation. The simplest spectrum kernel pulls out all k-mers from the strings and builds a vector space representation from them. For example, consider the string `ATGCGATGG` constructed on the alphabet $\Sigma = \{A, C, T, G\}$. Then, the corresponding spectrum representation for $k = 3$ is as follows:

```
ATG(2), TGC(1), GCG(1), CGA(1), GAT(1), TGG(1)
```

The values in the brackets correspond to the frequencies in the vector-space representation. This corresponds to the feature map $\Phi(\cdot)$ used to define the kernel similarity.

It is possible to enhance the spectrum kernel further by adding a *mismatch neighborhood* to the kernel. Thus, instead of adding only the extracted k-mers to the feature map, we add all the k-mers that are m mismatches away from the k-mer. For example, at a mismatch level of $m = 1$, the following k-mers are added to the feature map for each instance of `ATG`:

```
CTG, GTG, TTG, ACG, AAG, AGG, ATC, ATA, ATT
```

This procedure is repeated for each element in the k-mer, and each of the neighborhood elements are added to the feature map. is procedure is repeated for each element in the k-mer, and each of the neighborhood elements are added to the feature map. The dot product is performed on this expanded feature map $\Phi(\cdot)$. The rationale for adding mismatches is to allow for some noise in the similarity computation. The bag-of-words kernel can be viewed as a special case of the spectrum kernel with $k = 1$ and no mismatches. The spectrum kernel can be computed efficiently with the use of either the trie or the suffix tree data structure. Pointers to such efficient computational methods are provided in the bibliographic notes. One advantage of spectrum kernels is that they can compute the similarity between two strings in an intuitively appealing way, even when the lengths of the two strings are widely varying.

15.6.4.3 Weighted Degree Kernel

The previous two kernel methods directly define a feature map $\Phi(\cdot)$ explicitly that largely ignores the ordering between the different k-mers. The weighted degree kernel directly defines $K(Y_i, Y_j)$, without explicitly defining a feature map $\Phi(\cdot)$. This approach is in the spirit of exploiting the full power of kernel methods. Consider two strings Y_i and Y_j of the same length n. Let $KMER(Y_i, r, k)$ represent the k-mer extracted from Y_i starting from position r. Then, the weighted degree kernel computes the kernel similarity as the number of times the k-mers of a *maximum* specified length, in the two strings *at exactly corresponding positions*, match perfectly. Thus, unlike spectrum kernels, k-mers of varying lengths are used, and the contribution of a particular length s is weighted by coefficient β_s. In other words, weighted degree kernel of order k is defined as follows:

$$K(Y_i, Y_j) = \sum_{s=1}^{k} \beta_s \sum_{r=1}^{n-s+1} I(KMER(Y_i, r, s) = KMER(Y_j, r, s)) \tag{15.9}$$

Here, $I(\cdot)$ is an indicator function that takes on the value of 1 in case of a match, and 0 otherwise. One drawback of the weighted degree kernel over the spectrum kernel, is that it requires the two strings Y_i and Y_j to be of equal length. This can be partially addressed by allowing shifts in the matching process. Pointers to these enhancements may be found in the bibliographic notes.

15.6.5 Probabilistic Methods: Hidden Markov Models

Hidden Markov Models are an important tool that are utilized in a wide variety of tasks in sequence analysis. It has already been shown earlier in this chapter, in Sects. 15.3.4.2 and 15.5, how Hidden Markov Models can be utilized for both clustering and outlier detection. In this section, the use of Hidden Markov Models for sequence classification will be leveraged. In fact, the most common use of HMMs is for the problem of classification. HMMs are very popular in computational biology, where they are used for protein classification.

The basic approach for using HMMs for classification is to create a separate HMM for each of the classes in the data. Therefore, if there are a total of k classes, this will result in k different Hidden Markov Models. The Baum–Welch algorithm, described in Sect. 15.5.4, is used to train the HMMs for each class. For a given test sequence, the fit of each of the k models to the test sequence is determined using the approach described in Sect. 15.5.2. The best matching class is reported as the relevant one. The overall approach for training and testing with HMMs may be described as follows:

1. (Training) Use Baum–Welch algorithm of Sect. 15.5.4 to construct a separate HMM model for each of the k classes.

2. (Testing) For a given test sequence Y, determine the fit probability of the sequence to the k different Hidden Markov Models, using the evaluation procedure discussed in Sect. 15.5.2. Report the class, for which the corresponding HMM has the highest fit probability to the test sequence.

Many variations of this basic approach have been used to achieve different trade-offs between effectiveness and efficiency. The bibliographic notes contain pointers to some of these methods.

15.7 Summary

Discrete sequence mining is closely related to timeseries data mining, just as categorical data mining is closely related to numeric data mining. Therefore, many algorithms are very similar across the two domains. The work on discrete sequence mining originated in the field of computational biology, where DNA strands are encoded as strings.

The problem of sequential pattern mining discovers frequent sequences from a database of sequences. The *GSP* algorithm for frequent sequence mining is closely based on the *Apriori* method. Because of the close relationship between the two problems, most algorithms for frequent pattern mining can be generalized to discrete sequence mining in a relatively straightforward way.

Many of the methods for multidimensional clustering can be generalized to sequence clustering, as long as an effective similarity function can be defined between the sequences. Examples include the k-medoids method, hierarchical methods, and graph-based methods. An interesting line of work converts sequences to bags of k-grams. Text-clustering algorithms are applied to this representation. In addition, a number of specialized methods, such as *CLUSEQ*, have been developed. Probabilistic methods use a mixture of Hidden Markov Models for sequence clustering.

Outlier analysis for sequence data is similar to that for timeseries data. Position outliers are determined using Markovian models for probabilistic prediction. Combination outliers can be determined using distance-based, frequency-based, or Hidden Markov Models. Hidden Markov Models are a very general tool for sequence analysis and are used frequently for a wide variety of data mining tasks. HMMs can be viewed as mixture models, in which each state of the mixture is sequentially dependent on the previous states.

Numerous techniques from multidimensional classification can be adapted to discrete sequence classification. These include nearest neighbor methods, graph-based methods, rule-based methods, Hidden Markov Models, and Kernel Support Vector Machines. Numerous string kernels have been designed for more effective sequence classification.

15.8 Bibliographic Notes

The problem of sequence mining has been studied extensively in computational biology. The classical book by Gusfield [244] provides an excellent introduction of sequence mining algorithms from the perspective of computational biology. This book also contains an excellent survey on most of the other important similarity measures for strings, trees, and graphs. String indexing is discussed in detail in this work. The use of transformation rules for timeseries similarity has been studied in [283, 432]. Such rules can be used to create edit distance-like measures for continuous time series. Methods for the string edit distance are proposed in [438]. It has been shown in [141], how the L_p-norm may be combined with the edit distance. Algorithms for the longest common subsequence problem may be found in [77, 92, 270, 280]. A survey of these algorithms is available in [92]. Numerous other measures for timeseries and sequence similarity may be found in [32]. Timeseries and discrete sequence similarity measures are discussed in detail in Chap. 3 of this book, and in an earlier tutorial by Gunopulos and Das [241]. In the context of biological data, the BLAST system [73] is one of the most popular alignment tools.

The problem of mining sequential patterns was first proposed in [59]. The *GSP* algorithm was also proposed in the same work. The *GSP* algorithm is a straightforward modification of the *Apriori* algorithm. Most frequent pattern mining problems can be extended to sequential pattern mining because of the relationship between the two models. Subsequently, most of the algorithms discussed in Chap. 4 on frequent pattern mining have been generalized to sequential pattern mining. Savasere et al.'s vertical data structures [446] have been generalized to *SPADE* [535], and the *FP-growth* algorithm has been generalized to *PrefixSpan* [421]. The *TreeProjection* algorithm has also been generalized to sequential pattern mining [243]. Both *PrefixSpan* and the *TreeProjection*-based methods are based on combining database projection with exploration of the candidate search space with the use of an enumeration tree. The description of this chapter is a simplified and generalized description of these two related works [243, 421]. Methods for finding constraint-based sequences are discussed in [224, 346]. A recent survey on sequential pattern mining may be found in [392].

The problem of sequence data clustering has been studied extensively. A detailed survey on clustering sequence data, in the context of the biological domain, may be found in [32]. The *CLUSEQ* algorithm is described in detail in [523]. The Probabilistic Suffix Trees, used by *CLUSEQ*, are discussed in the same work. The earliest frequent sequence-based approach for clustering was proposed in [242]. The *CONTOUR* method for frequent sequence mining was proposed in [505]. This method uses a combination of frequent sequence mining and microclustering to create clusters from the sequences. The use of Hidden Markov Models for discrete sequence clustering is discussed in [474].

A significant amount of work has been done on the problem of temporal outlier detection in general, and discrete sequences in particular. A general survey on temporal outlier detection may be found in [237]. The book [5] contains chapters on temporal and discrete sequence outlier detection. A survey on anomaly detection in discrete sequences was presented in [132]. Two well-known techniques that use Markovian techniques for finding position outliers are discussed in [387, 525]. Combination outliers typically use windowing techniques in which comparison units are extracted from the sequence for the purposes of analysis [211, 274]. The information-theoretic measures for compression-based similarity were proposed in [311]. The frequency-based approach for determining the surprise level of comparison units is discussed in [310]. The *TARZAN* algorithm, proposed in this work, uses suffix trees for efficient computation. A general survey on Hidden Markov Models may be found in [327].

The problem of sequence classification is addressed in detail in the surveys [33, 516]. The use of wavelet methods for sequence classification was proposed in [51]. A description of a variety of string kernels for SVM classification is provided in [85]. The use of Hidden Markov Models for string classification is discussed in [327].

15.9 Exercises

1. Consider the sequence $ABCDDCBA$, defined on the alphabet $\Sigma = \{A, B, C, D\}$. Compute the vector-space representation for all k-mers of length 1, and the vector-space representation of all k-mers of length 2. Repeat the process for the sequence $CCCCDDDD$.

2. Implement the *GSP* algorithm for sequential pattern mining.

3. Consider a special case of the sequential pattern mining problem where elements are always singleton items. What difference would it make to (a) *GSP* algorithm, and (b) algorithms based on the candidate tree.

4. Discuss the generalizability of k-medoids and graph-based methods for clustering of arbitrary data types.

5. The chapter introduces a number of string kernels for classification with SVMs. Discuss some other data mining applications you can implement with string kernels.

6. Discuss the similarity and differences between Markovian models for discovering position outliers in sequential data, with autoregressive models for discovering point outliers in timeseries data.

7. Write a computer program to determine all maximal frequent subsequences from a collection using *GSP*. Implement a program to express the sequences in a database in terms of these subsequences, in vector space representation. Implement a k-means algorithm on this representation.

8. Write a computer program to determine position outliers using order-1 Markovian Models.

9. Consider the discrete sequence ACGTACGTACGTACGTATGT. Construct an order-1 Markovian model to determine the position outliers. Which positions are found as outliers?

10. For the discrete sequence of Exercise 9, determine all subsequences of length 2. Use a frequency-based approach to assign combination outlier scores to subsequences. Which subsequences should be considered combination outliers?

11. Write a computer program to learn the state transition and symbol emission probabilities for a Hidden Markov Model. Execute your program using the sequence of Exercise 9.

12. Compute the kernel similarity between the two sequences in Exercise 1 with a bag-of-words kernel and a spectrum kernel in which sequences of length 2 are used.

13. What is the maximum number of possible sequential patterns of length at most k, where the alphabet size is $|\Sigma|$. Compare this with frequent pattern mining. Which is larger?

14. Suppose that the speed of an athlete on a racetrack probabilistically depends upon whether the day is cold, moderate, or hot. Accordingly, the athlete runs a race that is graded either Fast (F), Slow (S), or Average (A). The weather on a particular day probabilistically depends on the weather on the previous day. Suppose that you have a sequence of performances of the athlete on successive days in the form of a string, such as $FSFAAF$. Construct a Hidden Markov Model that explains the athlete's performance, without any knowledge of the weather on those days.