2 Association Rules and Sequential Patterns

Association rules are an important class of regularities in data. Mining of association rules is a fundamental data mining task. It is perhaps the most important model invented and extensively studied by the database and data mining community. Its objective is to find all **co-occurrence** relationships, called **associations**, among data items. Since it was first introduced in 1993 by Agrawal et al. [9], it has attracted a great deal of attention. Many efficient algorithms, extensions and applications have been reported.

The classic application of association rule mining is the **market basket** data analysis, which aims to discover how items purchased by customers in a supermarket (or a store) are associated. An example association rule is

$$\text{Cheese} \rightarrow \text{Beer} \qquad [\text{support} = 10\%, \text{confidence} = 80\%].$$

The rule says that 10% customers buy Cheese and Beer together, and those who buy Cheese also buy Beer 80% of the time. Support and confidence are two measures of rule strength, which we will define later.

This mining model is in fact very general and can be used in many applications. For example, in the context of the Web and text documents, it can be used to find word co-occurrence relationships and Web usage patterns as we will see in later chapters.

Association rule mining, however, does not consider the sequence in which the items are purchased. Sequential pattern mining takes care of that. An example of a sequential pattern is "5% of customers buy bed first, then mattress and then pillows". The items are not purchased at the same time, but one after another. Such patterns are useful in Web usage mining for analyzing **clickstreams** in server logs. They are also useful for finding **language** or **linguistic patterns** from natural language texts.

2.1 Basic Concepts of Association Rules

The problem of mining association rules can be stated as follows: Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of **items**. Let $T = (t_1, t_2, \ldots, t_n)$ be a set of **transactions** (the database), where each transaction t_i is a set of items such that $t_i \subseteq I$. An **association rule** is an implication of the form,

$X \rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \varnothing$.

X (or Y) is a set of items, called an **itemset**.

Example 1: We want to analyze how the items sold in a supermarket are related to one another. I is the set of all items sold in the supermarket. A transaction is simply a set of items purchased in a basket by a customer. For example, a transaction may be:

{Beef, Chicken, Cheese},

which means that a customer purchased three items in a basket, Beef, Chicken, and Cheese. An association rule may be:

Beef, Chicken $\rightarrow$ Cheese,

where {Beef, Chicken} is X and {Cheese} is Y. For simplicity, brackets "{" and "}" are usually omitted in transactions and rules. ■

A transaction $t_i \in T$ is said to **contain** an itemset X if X is a subset of t_i (we also say that the itemset X **covers** t_i). The **support count** of X in T (denoted by $X.count$) is the number of transactions in T that contain X. The strength of a rule is measured by its **support** and **confidence**.

Support: The support of a rule, $X \rightarrow Y$, is the percentage of transactions in T that contains $X \cup Y$, and can be seen as an estimate of the probability, $\Pr(X \cup Y)$. The rule support thus determines how frequent the rule is applicable in the transaction set T. Let n be the number of transactions in T. The support of the rule $X \rightarrow Y$ is computed as follows:

$$support = \frac{(X \cup Y).count}{n}. \quad (1)$$

Support is a useful measure because if it is too low, the rule may just occur due to chance. Furthermore, in a business environment, a rule **covering** too few cases (or transactions) may not be useful because it does not make business sense to act on such a rule (not profitable).

Confidence: The confidence of a rule, $X \rightarrow Y$, is the percentage of transactions in T that contain X also contain Y. It can be seen as an estimate of the conditional probability, $\Pr(Y \mid X)$. It is computed as follows:

$$confidence = \frac{(X \cup Y).count}{X.count}. \quad (2)$$

Confidence thus determines the **predictability** of the rule. If the confidence of a rule is too low, one cannot reliably infer or predict Y from X. A rule with low predictability is of limited use.

Objective: Given a transaction set T, the problem of mining association rules is to discover all association rules in T that have support and confidence greater than or equal to the user-specified **minimum support** (denoted by **minsup**) and **minimum confidence** (denoted by **minconf**).

The keyword here is "all", i.e., association rule mining is complete. Previous methods for rule mining typically generate only a subset of rules based on various heuristics (see Chap. 3).

Example 2: Figure 2.1 shows a set of seven transactions. Each transaction t_i is a set of items purchased in a basket in a store by a customer. The set I is the set of all items sold in the store.

t_1: Beef, Chicken, Milk
t_2: Beef, Cheese
t_3: Cheese, Boots
t_4: Beef, Chicken, Cheese
t_5: Beef, Chicken, Clothes, Cheese, Milk
t_6: Chicken, Clothes, Milk
t_7: Chicken, Milk, Clothes

Fig. 2.1. An example of a transaction set

Given the user-specified minsup = 30% and minconf = 80%, the following association rule (**sup** is the support, and **conf** is the confidence)

Chicken, Clothes $\rightarrow$ Milk [sup = 3/7, conf = 3/3]

is valid as its support is 42.84% (> 30%) and its confidence is 100% (> 80%). The rule below is also valid, whose consequent has two items:

Clothes $\rightarrow$ Milk, Chicken [sup = 3/7, conf = 3/3].

Clearly, more association rules can be discovered, as we will see later. ■

We note that the data representation in the transaction form of Fig. 2.1 is a simplistic view of shopping baskets. For example, the quantity and price of each item are not considered in the model.

We also note that a text document or even a sentence in a single document can be treated as a transaction without considering word sequence and the number of occurrences of each word. Hence, given a set of documents or a set of sentences, we can find word co-occurrence relations.

A large number of association rule mining algorithms have been reported in the literature, which have different mining efficiencies. Their resulting sets of rules are, however, all the same based on the definition of association rules. That is, given a transaction data set T, a minimum support and a minimum confidence, the set of association rules existing in T is

uniquely determined. Any algorithm should find the same set of rules although their computational efficiencies and memory requirements may be different. The best known mining algorithm is the **Apriori** algorithm proposed in [11], which we study next.

2.2 Apriori Algorithm

The Apriori algorithm works in two steps:

1. **Generate all frequent itemsets:** A frequent itemset is an itemset that has transaction support above minsup.
2. **Generate all confident association rules from the frequent itemsets:** A confident association rule is a rule with confidence above minconf.

We call the number of items in an itemset its **size**, and an itemset of size k a k-itemset. Following Example 2 above, {Chicken, Clothes, Milk} is a frequent 3-itemset as its support is 3/7 (minsup = 30%). From the itemset, we can generate the following three association rules (minconf = 80%):

Rule 1:	Chicken, Clothes → Milk	[sup = 3/7, conf = 3/3]
Rule 2:	Clothes, Milk → Chicken	[sup = 3/7, conf = 3/3]
Rule 3:	Clothes → Milk, Chicken	[sup = 3/7, conf = 3/3].

Below, we discuss the two steps in turn.

2.2.1 Frequent Itemset Generation

The Apriori algorithm relies on the ***apriori*** or **downward closure** property to efficiently generate all frequent itemsets.

Downward Closure Property: If an itemset has minimum support, then every non-empty subset of this itemset also has minimum support.

The idea is simple because if a transaction contains a set of items X, then it must contain any non-empty subset of X. This property and the minsup threshold prune a large number of itemsets that cannot be frequent.

To ensure efficient itemset generation, the algorithm assumes that the items in I are sorted in **lexicographic order** (a total order). The order is used throughout the algorithm in each itemset. We use the notation $\{w[1], w[2], \ldots, w[k]\}$ to represent a k-itemset w consisting of items $w[1]$, $w[2]$, …, $w[k]$, where $w[1] < w[2] < \ldots < w[k]$ according to the total order.

The Apriori algorithm for frequent itemset generation, which is given in Fig. 2.2, is based on **level-wise search**. It generates all frequent itemsets by

```
Algorithm Apriori(T)
1   C1 ← init-pass(T);                          // the first pass over T
2   F1 ← {f | f ∈ C1, f.count/n ≥ minsup};      // n is the no. of transactions in T
3   for (k = 2; Fk–1 ≠ ∅; k++) do               // subsequent passes over T
4     Ck ← candidate-gen(Fk–1);
5     for each transaction t ∈ T do             // scan the data once
6       for each candidate c ∈ Ck do
7         if c is contained in t then
8           c.count++;
9       endfor
10    endfor
11    Fk ← {c ∈ Ck | c.count/n ≥ minsup}
12  endfor
13  return F ← ⋃k Fk;
```

Fig. 2.2. The Apriori algorithm for generating frequent itemsets

```
Function candidate-gen(Fk–1)
1   Ck ← ∅;                                     // initialize the set of candidates
2   forall f1, f2 ∈ Fk–1                        // find all pairs of frequent itemsets
3     with f1 = {i1, … , ik–2, ik–1}            // that differ only in the last item
4     and f2 = {i1, … , ik–2, i'k–1}
5     and ik–1 < i'k–1 do                       // according to the lexicographic order
6       c ← {i1, …, ik–1, i'k–1};               // join the two itemsets f1 and f2
7       Ck ← Ck ∪ {c};                          // add the new itemset c to the candidates
8       for each (k–1)-subset s of c do
9         if (s ∉ Fk–1) then
10          delete c from Ck;                   // delete c from the candidates
11      endfor
12  endfor
13  return Ck;                                  // return the generated candidates
```

Fig. 2.3. The candidate-gen function

making multiple passes over the data. In the first pass, it counts the supports of individual items (line 1) and determines whether each of them is frequent (line 2). F_1 is the set of frequent 1-itemsets. In each subsequent pass k, there are three steps:

1. It starts with the seed set of itemsets F_{k-1} found to be frequent in the $(k-1)$-th pass. It uses this seed set to generate **candidate itemsets** C_k (line 4), which are possible frequent itemsets. This is done using the candidate-gen() function.
2. The transaction database is then scanned and the actual support of each candidate itemset c in C_k is counted (lines 5–10). Note that we do not need to load the whole data into memory before processing. Instead, at

any time, only one transaction resides in memory. This is a very important feature of the algorithm. It makes the algorithm scalable to huge data sets, which cannot be loaded into memory.
3. At the end of the pass or scan, it determines which of the candidate itemsets are actually frequent (line 11).

The final output of the algorithm is the set F of all frequent itemsets (line 13). The candidate-gen() function is discussed below.

Candidate-gen function: The candidate generation function is given in Fig. 2.3. It consists of two steps, the **join step** and the **pruning step**.

Join step (lines 2–6 in Fig. 2.3): This step joins two frequent (k–1)-itemsets to produce a possible candidate c (line 6). The two frequent itemsets f_1 and f_2 have exactly the same items except the last one (lines 3–5). c is added to the set of candidates C_k (line 7).

Pruning step (lines 8–11 in Fig. 2.3): A candidate c from the join step may not be a final candidate. This step determines whether all the k–1 subsets (there are k of them) of c are in F_{k-1}. If anyone of them is not in F_{k-1}, c cannot be frequent according to the downward closure property, and is thus deleted from C_k.

The correctness of the candidate-gen() function is easy to show (see [11]). Here, we use an example to illustrate the working of the function.

Example 3: Let the set of frequent itemsets at level 3 be

$$F_3 = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{1, 3, 5\}, \{2, 3, 4\}\}.$$

For simplicity, we use numbers to represent items. The join step (which generates candidates for level 4) will produce two candidate itemsets, {1, 2, 3, 4} and {1, 3, 4, 5}. {1, 2, 3, 4} is generated by joining the first and the second itemsets in F_3 as their first and second items are the same respectively. {1, 3, 4, 5} is generated by joining {1, 3, 4} and {1, 3, 5}.

After the pruning step, we have only:

$$C_4 = \{\{1, 2, 3, 4\}\}$$

because {1, 4, 5} is not in F_3 and thus {1, 3, 4, 5} cannot be frequent.

Example 4: Let us see a complete running example of the Apriori algorithm based on the transactions in Fig. 2.1. We use minsup = 30%.

F_1: {{Beef}:4, {Cheese}:4, {Chicken}:5, {Clothes}:3, {Milk}:4}

Note: the number after each frequent itemset is the support count of the itemset, i.e., the number of transactions containing the itemset. A minimum support count of 3 is sufficient because the support of 3/7 is greater than 30%, where 7 is the total number of transactions.

C_2: {{Beef, Cheese}, {Beef, Chicken}, {Beef, Clothes}, {Beef, Milk}, {Cheese, Chicken}, {Cheese, Clothes}, {Cheese, Milk}, {Chicken, Clothes}, {Chicken, Milk}, {Clothes, Milk}}

F_2: {{Beef, Chicken}:3, {Beef, Cheese}:3, {Chicken, Clothes}:3, {Chicken, Milk}:4, {Clothes, Milk}:3}

C_3: {{Chicken, Clothes, Milk}}

Note: {Beef, Cheese, Chicken} is also produced in line 6 of Fig. 2.3. However, {Cheese, Chicken} is not in F_2, and thus the itemset {Beef, Cheese, Chicken} is not included in C_3.

F_3: {{Chicken, Clothes, Milk}:3}. ■

Finally, some remarks about the Apriori algorithm are in order:

- Theoretically, this is an exponential algorithm. Let the number of items in I be m. The space of all itemsets is $O(2^m)$ because each item may or may not be in an itemset. However, the mining algorithm exploits the sparseness of the data and the high minimum support value to make the mining possible and efficient. The **sparseness** of the data in the context of market basket analysis means that the store sells a lot of items, but each shopper only purchases a few of them.
- The algorithm can scale up to large data sets as it does not load the entire data into the memory. It only scans the data K times, where K is the size of the largest itemset. In practice, K is often small (e.g., < 10). This scale-up property is very important in practice because many real-world data sets are so large that they cannot be loaded into the main memory.
- The algorithm is based on level-wise search. It has the flexibility to stop at any level. This is useful in practice because in many applications, long frequent itemsets or rules are not needed as they are hard to use.
- As mentioned earlier, once a transaction set T, a minsup and a minconf are given, the set of frequent itemsets that can be found in T is uniquely determined. Any algorithm should find the same set of frequent itemsets. This property about association rule mining does not hold for many other data mining tasks, e.g., classification or clustering, for which different algorithms may produce very different results.
- The main problem with association rule mining is that it often produces a huge number of itemsets (and rules), tens of thousands, or more, which makes it hard for the user to analyze them to find those useful ones. This is called the **interestingness** problem. Researchers have proposed several methods to tackle this problem (see Bibliographic Notes).

An efficient implementation of the Apriori algorithm involves sophisticated data structures and programming techniques, which are beyond the

scope of this book. Apart from the Apriori algorithm, there is a large number of other algorithms, e.g., FP-growth [220] and many others.

2.2.2 Association Rule Generation

In many applications, frequent itemsets are already useful and sufficient. Then, we do not need to generate association rules. In applications where rules are desired, we use frequent itemsets to generate all association rules.

Compared with frequent itemset generation, rule generation is relatively simple. To generate rules for every frequent itemset f, we use all non-empty subsets of f. For each such subset α, we output a rule of the form

$$(f - \alpha) \rightarrow \alpha, \text{ if}$$

$$confidence = \frac{f.count}{(f - \alpha).count} \geq minconf, \tag{3}$$

where $f.count$ (or $(f-\alpha).count$) is the support count of f (or $(f - \alpha)$). The support of the rule is $f.count/n$, where n is the number of transactions in the transaction set T. All the support counts needed for confidence computation are available because if f is frequent, then any of its non-empty subsets is also frequent and its support count has been recorded in the mining process. Thus, no data scan is needed in rule generation.

This exhaustive rule generation strategy is, however, inefficient. To design an efficient algorithm, we observe that the support count of f in the above confidence computation does not change as α changes. It follows that for a rule $(f - \alpha) \rightarrow \alpha$ to hold, all rules of the form $(f - \alpha_{sub}) \rightarrow \alpha_{sub}$ must also hold, where α_{sub} is a non-empty subset of α, because the support count of $(f - \alpha_{sub})$ must be less than or equal to the support count of $(f - \alpha)$. For example, given an itemset {A, B, C, D}, if the rule (A, B $\rightarrow$ C, D) holds, then the rules (A, B, C $\rightarrow$ D) and (A, B, D $\rightarrow$ C) must also hold.

Thus, for a given frequent itemset f, if a rule with consequent α holds, then so do rules with consequents that are subsets of α. This is similar to the downward closure property that, if an itemset is frequent, then so are all its subsets. Therefore, from the frequent itemset f, we first generate all rules with one item in the consequent. We then use the consequents of these rules and the function candidate-gen() (Fig. 2.3) to generate all possible consequents with two items that can appear in a rule, and so on. An algorithm using this idea is given in Fig. 2.4. Note that all 1-item consequent rules (rules with one item in the consequent) are first generated in line 2 of the function genRules(). The confidence is computed using (3).

```
Algorithm genRules(F)                    // F is the set of all frequent itemsets
1   for each frequent k-itemset f_k in F, k ≥ 2 do
2       output every 1-item consequent rule of f_k with confidence ≥ minconf and
            support ← f_k.count / n     // n is the total number of transactions in T
3       H_1 ←{consequents of all 1-item consequent rules derived from f_k above};
4       ap-genRules(f_k, H_1);
5   endfor

Procedure ap-genRules(f_k, H_m)          // H_m is the set of m-item consequents
1   if (k > m + 1) AND (H_m ≠ ∅) then
2     H_{m+1} ← candidate-gen(H_m);
3     for each h_{m+1} in H_{m+1} do
4         conf ← f_k.count / (f_k – h_{m+1}).count;
5         if (conf ≥ minconf) then
6             output the rule (f_k – h_{m+1}) → h_{m+1} with confidence = conf and
              support = f_k.count / n;     // n is the total number of transactions in T
7         else
8           delete h_{m+1} from H_{m+1};
9     endfor
10    ap-genRules(f_k, H_{m+1});
11  endif
```

Fig. 2.4. The association rule generation algorithm

Example 5: We again use transactions in Fig. 2.1, minsup = 30% and minconf = 80%. The frequent itemsets are as follows (see Example 4):

F_1: {{Beef}:4, {Cheese}:4, {Chicken}:5, {Clothes}:3, {Milk}:4}
F_2: {{Beef, Cheese}:3, {Beef, Chicken}:3, {Chicken, Clothes}:3, {Chicken, Milk}:4, {Clothes, Milk}:3}
F_3: {{Chicken, Clothes, Milk}:3}.

We use only the itemset in F_3 to generate rules (generating rules from each itemset in F_2 can be done in the same way). The itemset in F_3 generates the following possible 1-item consequent rules:

Rule 1: Chicken, Clothes → Milk [sup = 3/7, conf = 3/3]
Rule 2: Chicken, Milk → Clothes [sup = 3/7, conf = 3/4]
Rule 3: Clothes, Milk → Chicken [sup = 3/7, conf = 3/3].

Due to the minconf requirement, only Rule 1 and Rule 3 are output in line 2 of the algorithm genRules(). Thus, H_1 = {{Chicken}, {Milk}}. The function ap-genRules() is then called. Line 2 of ap-genRules() produces H_2 = {{Chicken, Milk}}. The following rule is then generated:

Rule 4: Clothes → Milk, Chicken [sup = 3/7, conf = 3/3].

Thus, three association rules are generated from the frequent itemset {Chicken, Clothes, Milk} in F_3, namely Rule 1, Rule 3 and Rule 4. ■

2.3 Data Formats for Association Rule Mining

So far, we have used only transaction data for mining association rules. Market basket data sets are naturally of this format. Text documents can be seen as transaction data as well. Each document is a transaction, and each distinctive word is an item. Duplicate words are removed.

However, mining can also be performed on relational tables. We just need to convert a table data set to a transaction data set, which is fairly straightforward if each attribute in the table takes **categorical** values. We simply change each value to an **attribute–value** pair.

Example 6: The table data in Fig. 2.5(A) can be converted to the transaction data in Fig. 2.5(B). Each attribute–value pair is considered an **item**. Using only values is not sufficient in the transaction form because different attributes may have the same values. For example, without including attribute names, value a's for Attribute1 and Attribute2 are not distinguishable. After the conversion, Fig. 2.5(B) can be used in mining. ■

If an attribute takes numerical values, it becomes complex. We need to first discretize its value range into intervals, and treat each interval as a categorical value. For example, an attribute's value range is from 1–100. We may want to divide it into 5 equal-sized intervals, 1–20, 21–40, 41–60, 61–80, and 81–100. Each interval is then treated as a categorical value. Discretization can be done manually based on expert knowledge or automatically. There are several existing algorithms [151, 501].

A point to note is that for a table data set, the join step of the candidate generation function (Fig. 2.3) needs to be slightly modified in order to ensure that it does not join two itemsets to produce a candidate itemset containing two items from the same attribute.

Clearly, we can also convert a transaction data set to a table data set using a binary representation and treating each item in I as an attribute. If a transaction contains an item, its attribute value is 1, and 0 otherwise.

2.4 Mining with Multiple Minimum Supports

The key element that makes association rule mining practical is the minsup threshold. It is used to prune the search space and to limit the number of frequent itemsets and rules generated. However, using only a single min-

Attribute1	Attribute2	Atribute3
a	a	x
b	n	y

(A) Table data

t_1: (Attribute1, a), (Attribute2, a), (Attribute3, x)
t_2: (Attribute1, b), (Attribute2, n), (Attribute3, y)

(B) Transaction data

Fig. 2.5. From a table data set to a transaction data set

sup implicitly assumes that all items in the data are of the same nature and/or have similar frequencies in the database. This is often not the case in real-life applications. In many applications, some items appear very frequently in the data, while some other items rarely appear. If the frequencies of items vary a great deal, we will encounter two problems [344]:

1. If the minsup is set too high, we will not find rules that involve infrequent items or **rare items** in the data.
2. In order to find rules that involve both frequent and rare items, we have to set the minsup very low. However, this may cause combinatorial explosion and make mining impossible because those frequent items will be associated with one another in all possible ways.

Let us use an example to illustrate the above problem with a very low minsup, which will actually introduce another problem.

Example 7: In a supermarket transaction data set, in order to find rules involving those infrequently purchased items such as FoodProcessor and CookingPan (they generate more profits per item), we need to set the minsup very low. Let us use only frequent itemsets in this example as they are generated first and rules are produced from them. They are also the source of all the problems. Now assume we set a very low minsup of 0.005%. We find the following meaningful frequent itemset:

{FoodProcessor, CookingPan} [sup = 0.006%].

However, this low minsup may also cause the following two meaningless itemsets being discovered:

f_1: {Bread, Cheese, Egg, Bagel, Milk, Sugar, Butter} [sup = 0.007%],

f_2: {Bread, Egg, Milk, CookingPan} [sup = 0.006%].

Knowing that 0.007% of the customers buy the seven items in f_1 together is useless because all these items are so frequently purchased in a supermar-

ket. Worst still, they will almost certainly cause combinatorial explosion! For itemsets involving such items to be useful, their supports have to be much higher. Similarly, knowing that 0.006% of the customers buy the four items in f_2 together is also meaningless because Bread, Egg and Milk are purchased on almost every grocery shopping trip. ■

This dilemma is called the **rare item problem**. Using a single minsup for the whole data set is inadequate because it cannot capture the inherent natures and/or frequency differences of items in the database. By the natures of items we mean that some items, by nature, appear more frequently than others. For example, in a supermarket, people buy FoodProcessor and CookingPan much less frequently than Bread and Milk. The situation is the same for online stores. In general, those durable and/or expensive goods are bought less often, but each of them generates more profit. It is thus important to capture rules involving less frequent items. However, we must do so without allowing frequent items to produce too many meaningless rules with very low supports and cause combinatorial explosion [344].

One common solution to this problem is to partition the data into several smaller blocks (subsets), each of which contains only items of similar frequencies. Mining is then done separately for each block using a different minsup. This approach is, however, not satisfactory because itemsets or rules that involve items across different blocks will not be found.

A better solution is to allow the user to specify multiple minimum supports, i.e., to specify a different **minimum item support** (**MIS**) to each item. Thus, different itemsets need to satisfy different minimum supports depending on what items are in the itemsets. This model thus enables us to achieve our objective of finding itemsets involving rare items without causing frequent items to generate too many meaningless itemsets. This method helps solve the problem of f_1. To deal with the problem of f_2, we prevent itemsets that contain both very frequent items and very rare items from being generated. A **constraint** will be introduced to realize this.

An **interesting by-product** of this extended model is that it enables the user to easily instruct the algorithm to generate only itemsets that contain certain items but not itemsets that contain only the other items. This can be done by setting the MIS values to more than 100% (e.g., 101%) for these other items. This capability is very useful in practice because in many applications the user is only interested in certain types of itemsets or rules.

2.4.1 Extended Model

To allow multiple minimum supports, the original model in Sect. 2.1 needs to be extended. In the extended model, the minimum support of a rule is

expressed in terms of **minimum item supports** (**MIS**) of the items that appear in the rule. That is, each item in the data can have a MIS value specified by the user. By providing different MIS values for different items, the user effectively expresses different support requirements for different rules. It seems that specifying a MIS value for each item is a difficult task. This is not so as we will see at the end of Sect. 2.4.2.

Let MIS(i) be the MIS value of item i. The **minimum support** of a rule R is the lowest MIS value among the items in the rule. That is, a rule R,

$$i_1, i_2, \ldots, i_k \rightarrow i_{k+1}, \ldots, i_r,$$

satisfies its minimum support if the rule's actual support in the data is greater than or equal to:

$$min(\text{MIS}(i_1), \text{MIS}(i_2), \ldots, \text{MIS}(i_r)).$$

Minimum item supports thus enable us to achieve the goal of having higher minimum supports for rules that involve only frequent items, and having lower minimum supports for rules that involve less frequent items.

Example 8: Consider the set of items in a data set, {Bread, Shoes, Clothes}. The user-specified MIS values are as follows:

MIS(Bread) = 2% MIS(Clothes) = 0.2% MIS(Shoes) = 0.1%.

The following rule doesn't satisfy its minimum support:

Clothes $\rightarrow$ Bread [sup = 0.15%, conf = 70%].

This is so because *min*(MIS(Bread), MIS(Clothes)) = 0.2%. The following rule satisfies its minimum support:

Clothes $\rightarrow$ Shoes [sup = 0.15%, conf = 70%].

because *min*(MIS(Clothes), MIS(Shoes)) = 0.1%. ■

As we explained earlier, the **downward closure property** holds the key to pruning in the Apriori algorithm. However, in the new model, if we use the Apriori algorithm to find all frequent itemsets, the downward closure property no longer holds.

Example 9: Consider the four items 1, 2, 3 and 4 in a data set. Their minimum item supports are:

MIS(1) = 10% MIS(2) = 20% MIS(3) = 5% MIS(4) = 6%.

If we find that itemset {1, 2} has a support of 9% at level 2, then it does not satisfy either MIS(1) or MIS(2). Using the Apriori algorithm, this itemset is discarded since it is not frequent. Then, the potentially frequent itemsets {1, 2, 3} and {1, 2, 4} will not be generated for level 3. Clearly, itemsets {1,

2, 3} and {1, 2, 4} may be frequent because MIS(3) is only 5% and MIS(4) is 6%. It is thus wrong to discard {1, 2}. However, if we do not discard {1, 2}, the downward closure property is lost. ■

Below, we present an algorithm to solve this problem. The essential idea is to sort the items according to their MIS values in ascending order to avoid the problem.

Note that MIS values prevent low support itemsets involving only frequent items from being generated because their individual MIS values are all high. To prevent very frequent items and very rare items from appearing in the same itemset, we introduce the **support difference constraint**.

Let $sup(i)$ be the actual support of item i in the data. For each itemset s, the support difference constraint is as follows:

$$max_{i \in s}\{sup(i)\} - min_{i \in s}\{sup(i)\} \leq \varphi,$$

where $0 \leq \varphi \leq 1$ is the user-specified **maximum support difference**, and it is the same for all itemsets. The constraint basically limits the difference between the largest and the smallest actual supports of items in itemset s to φ. This constraint can reduce the number of itemsets generated dramatically, and it does not affect the downward closure property.

2.4.2 Mining Algorithm

The new algorithm generalizes the Apriori algorithm for finding frequent itemsets. We call the algorithm, **MS-Apriori**. When there is only one MIS value (for all items), it reduces to the Apriori algorithm.

Like Apriori, MS-Apriori is also based on level-wise search. It generates all frequent itemsets by making multiple passes over the data. However, there is an exception in the second pass as we will see later.

The key operation in the new algorithm is the sorting of the items in I in ascending order of their MIS values. This order is fixed and used in all subsequent operations of the algorithm. The items in each itemset follow this order. For example, in Example 9 of the four items 1, 2, 3 and 4 and their given MIS values, the items are sorted as follows: 3, 4, 1, 2. This order helps solve the problem identified above.

Let F_k denote the set of frequent k-itemsets. Each itemset w is of the following form, $\{w[1], w[2], \ldots, w[k]\}$, which consists of items, $w[1]$, $w[2]$, …, $w[k]$, where $MIS(w[1]) \leq MIS(w[2]) \leq \ldots \leq MIS(w[k])$. The algorithm MS-Apriori is given in Fig. 2.6. Line 1 performs the sorting on I according to the MIS value of each item (stored in MS). Line 2 makes the first pass over the data using the function init-pass(), which takes two arguments, the

```
Algorithm MS-Apriori(T, MS, φ)                    // MS stores all MIS values
1   M ← sort(I, MS);                              // according to MIS(i)'s stored in MS
2   L ← init-pass(M, T);                          // make the first pass over T
3   F1 ← {{l} | l ∈ L, l.count/n ≥ MIS(l)};       // n is the size of T
4   for (k = 2; Fk−1 ≠ ∅; k++) do
5     if k = 2 then
6       Ck ← level2-candidate-gen(L, φ)           // k = 2
7     else Ck ← MScandidate-gen(Fk−1, φ)
8     endif;
9     for each transaction t ∈ T do
10      for each candidate c ∈ Ck do
11        if c is contained in t then             // c is a subset of t
12          c.count++
13        if c – {c[1]} is contained in t then    // c without the first item
14          (c – {c[1]}).count++
15      endfor
16    endfor
17    Fk ← {c ∈ Ck | c.count/n ≥ MIS(c[1])}
18  endfor
19  return F ← ⋃k Fk;
```

Fig. 2.6. The MS-Apriori algorithm

data set T and the sorted items M, to produce the seeds L for generating candidate itemsets of length 2, i.e., C_2. init-pass() has two steps:

1. It first scans the data once to record the support count of each item.
2. It then follows the sorted order to find the first item i in M that meets MIS(i). i is inserted into L. For each subsequent item j in M after i, if $j.count/n \geq$ MIS(i), then j is also inserted into L, where $j.count$ is the support count of j, and n is the total number of transactions in T.

Frequent 1-itemsets (F_1) are obtained from L (line 3). It is easy to show that all frequent 1-itemsets are in F_1.

Example 10: Let us follow Example 9 and the given MIS values for the four items. Assume our data set has 100 transactions (not limited to the four items). The first pass over the data gives us the following support counts: {3}.*count* = 6, {4}.*count* = 3, {1}.*count* = 9 and {2}.*count* = 25. Then,

$$L = \{3, 1, 2\}, \text{ and } F_1 = \{\{3\}, \{2\}\}.$$

Item 4 is not in L because 4.*count*/n < MIS(3) (= 5%), and {1} is not in F_1 because 1.*count* / n < MIS(1) (= 10%). ■

For each subsequent pass (or data scan), say pass k, the algorithm performs three operations.

1. The frequent itemsets in F_{k-1} found in the $(k-1)th$ pass are used to generate the candidates C_k using the MScandidate-gen() function (line 7). However, there is a special case, i.e., when $k = 2$ (line 6), for which the candidate generation function is different, i.e., level2-candidate-gen().
2. It then scans the data and updates various support counts of the candidates in C_k (line 9–16). For each candidate c, we need to update its support count (lines 11–12) and also the support count of c without the first item (lines 13–14), i.e., $c - \{c[1]\}$, which is used in rule generation and will be discussed in Sect. 2.4.3. If rule generation is not required, lines 13 and 14 can be deleted.
3. The frequent itemsets (F_k) for the pass are identified in line 17.

We present candidate generation functions level2-candidate-gen() and MScandidate-gen() below.

Level2-candidate-gen function: It takes an argument L, and returns a superset of the set of all frequent 2-itemsets. The algorithm is given in Fig. 2.7. Note that in line 5, we use $|sup(h) - sup(l)| \leq \varphi$ because $sup(l)$ may not be lower than $sup(h)$, although $MIS(l) \leq MIS(h)$.

Example 11: Let us continue with Example 10. We set $\varphi = 10\%$. Recall the MIS values of the four items are (in Example 9):

$$MIS(1) = 10\% \qquad MIS(2) = 20\%$$
$$MIS(3) = 5\% \qquad MIS(4) = 6\%.$$

The level2-candidate-gen() function in Fig. 2.7 produces

$$C_2 = \{\{3, 1\}\}.$$

{1, 2} is not a candidate because the support count of item 1 is only 9 (or 9%), less than MIS(1) (= 10%). Hence, {1, 2} cannot be frequent. {3, 2} is not a candidate because $sup(3) = 6\%$ and $sup(2) = 25\%$ and their difference is greater than $\varphi = 10\%$ ■

Note that we must use L rather than F_1 because F_1 does not contain those items that may satisfy the MIS of an earlier item (in the sorted order) but not the MIS of itself, e.g., item 1 in the above example. Using L, the problem discussed in Sect. 2.4.1 is solved for C_2.

MScandidate-gen function: The algorithm is given in Fig. 2.8, which is similar to the candidate-gen function in the Apriori algorithm. It also has two steps, the **join step** and the **pruning step**. The join step (lines 2–6) is the same as that in the candidate-gen() function. The pruning step (lines 8–12) is, however, different.

For each (k-1)-subset s of c, if s is not in F_{k-1}, c can be deleted from C_k. However, there is an exception, which is when s does not include $c[1]$

```
Function level2-candidate-gen(L, φ)
1  C2 ← ∅;                                  // initialize the set of candidates
2  for each item l in L in the same order do
3     if l.count/n ≥ MIS(l) then
4        for each item h in L that is after l do
5           if h.count/n ≥ MIS(l) and |sup(h) − sup(l)| ≤ φ then
6              C2 ← C2 ∪ {{l, h}};   // insert the candidate {l, h} into C2
```

Fig. 2.7. The level2-candidate-gen function

```
Function MScandidate-gen(Fk–1, φ)
1   Ck ← ∅;                                 // initialize the set of candidates
2   forall f1, f2 ∈ Fk                      // find all pairs of frequent itemsets
3     with f1 = {i1, … , ik–2, ik–1}        // that differ only in the last item
4     and f2 = {i1, … , ik–2, i'k–1}
5     and ik-1 < i'k–1 and |sup(ik-1) − sup(i'k–1)| ≤ φ do
6        c ← {i1, …, ik–1, i'k–1};          // join the two itemsets f1 and f2
7        Ck ← Ck ∪ {c};                     // insert the candidate itemset c into Ck
8        for each (k–1)-subset s of c do
9           if (c[1] ∈ s) or (MIS(c[2]) = MIS(c[1])) then
10             if (s ∉ Fk–1) then
11                delete c from Ck;         // delete c from the set of candidates
12       endfor
13  endfor
14  return Ck;                              // return the generated candidates
```

Fig. 2.8. The MScandidate-gen function

(there is only one such s). That is, the first item of c, which has the lowest MIS value, is not in s. Even if s is not in F_{k-1}, we cannot delete c because we cannot be sure that s does not satisfy MIS(c[1]), although we know that it does not satisfy MIS(c[2]), unless MIS(c[2]) = MIS(c[1]) (line 9).

Example 12: Let F_3 = {{1, 2, 3}, {1, 2, 5}, {1, 3, 4}, {1, 3, 5}, {1, 4, 5}, {1, 4, 6}, {2, 3, 5}}. Items in each itemset are in the sorted order. The join step produces (we ignore the support difference constraint here)

{1, 2, 3, 5}, {1, 3, 4, 5} and {1, 4, 5, 6}.

The pruning step deletes {1, 4, 5, 6} because {1, 5, 6} is not in F_3. We are then left with C_4 = {{1, 2, 3, 5}, {1, 3, 4, 5}}. {1, 3, 4, 5} is not deleted although {3, 4, 5} is not in F_3 because the minimum support of {3, 4, 5} is MIS(3), which may be higher than MIS(1). Although {3, 4, 5} does not satisfy MIS(3), we cannot be sure that it does not satisfy MIS(1). However, if MIS(3) = MIS(1), then {1, 3, 4, 5} can also be deleted. ■

The problem discussed in Sect. 2.4.1 is solved for C_k ($k > 2$) because, due to the sorting, we do not need to extend a frequent (k–1)-itemset with any item that has a lower MIS value. Let us see a complete example.

Example 13: Given the following seven transactions,

Beef, Bread
Bread, Clothes
Bread, Clothes, Milk
Cheese, Boots
Beef, Bread, Cheese, Shoes
Beef, Bread, Cheese, Milk
Bread, Milk, Clothes

and MIS(Milk) = 50%, MIS(Bread) = 70%, and 25% for all other items. Again, the support difference constraint is not used. The following frequent itemsets are produced:

F_1 = {{Beef}, {Cheese}, {Clothes}, {Bread}}
F_2 = {{Beef, Cheese}, {Beef, Bread}, {Cheese, Bread}
{Clothes, Bread}, {Clothes, Milk}}
F_3 = {{Beef, Cheese, Bread}, {Clothes, Milk, Bread}}. ■

To conclude this sub-section, let us further discuss two important issues:

1. Specify MIS values for items: This is usually done in two ways:
   - Assign a MIS value to each item according to its actual support/frequency in the data set T. For example, if the actual support of item i in T is $sup(i)$, then the MIS value for i may be computed with $\lambda \times sup(i)$, where λ is a parameter ($0 \leq \lambda \leq 1$) and is the same for all items in T.
   - Group items into clusters (or blocks). Items in each cluster have similar frequencies. All items in the same cluster are given the same MIS value. We should note that in the extended model frequent itemsets involving items from different clusters will be found.
2. Generate itemsets that must contain certain items: As mentioned earlier, the extended model enables the user to instruct the algorithm to generate itemsets that must contain certain items, or not to generate any itemsets consisting of only the other items. Let us see an example.

   Example 14: Given the data set in Example 13, if we want to generate frequent itemsets that must contain at least one item in {Boots, Bread, Cheese, Milk, Shoes}, or not to generate itemsets involving only Beef and/or Clothes, we can simply set

   MIS(Beef) = 101%, and MIS(Clothes) = 101%

Then the algorithm will not generate the itemsets, {Beef}, {Clothes} and {Beef, Clothes}. However, it will still generate such frequent itemsets as {Cheese, Beef} and {Cheese, Bread, Beef}. ■

In many applications, this feature comes quite handy because the user is often only interested in certain types of itemsets or rules.

2.4.3 Rule Generation

Association rules are generated using frequent itemsets. In the case of a single minsup, if f is a frequent itemset and f_{sub} is a subset of f, then f_{sub} must also be a frequent itemset. All their support counts are computed and recorded by the Apriori algorithm. Then, the confidence of each possible rule can be easily calculated without seeing the data again.

However, in the case of MS-Apriori, if we only record the support count of each frequent itemset, it is not sufficient. Let us see why.

Example 15: Recall in Example 8, we have

MIS(Bread) = 2% MIS(Clothes) = 0.2% MIS(Shoes) = 0.1%.

If the actual support for the itemset {Clothes, Bread} is 0.15%, and for the itemset {Shoes, Clothes, Bread} is 0.12%, according to MS-Apriori, {Clothes, Bread} is not a frequent itemset since its support is less than MIS(Clothes). However, {Shoes, Clothes, Bread} is a frequent itemset as its actual support is greater than

min(MIS(Shoes), MIS(Clothes), MIS(Bread)) = MIS(Shoes)).

We now have a problem in computing the confidence of the rule,

Clothes, Bread → Shoes

because the itemset {Clothes, Bread} is not a frequent itemset and thus its support count is not recorded. In fact, we may not be able to compute the confidences of the following rules either:

Clothes → Shoes, Bread
Bread → Shoes, Clothes

because {Clothes} and {Bread} may not be frequent. ■

Lemma: The above problem may occur only when the item that has the lowest MIS value in the itemset is in the consequent of the rule (which may have multiple items). We call this problem the **head-item problem**.

Proof by contradiction: Let f be a frequent itemset, and $a \in f$ be the item with the lowest MIS value in f (a is called the **head item**). Thus, f uses

MIS(a) as its minsup. We want to form a rule, $X \rightarrow Y$, where X, $Y \subset f$, $X \cup Y = f$ and $X \cap Y = \emptyset$. Our examples above already show that the head-item problem may occur when $a \in Y$. Now assume that the problem can also occur when $a \in X$. Since $a \in X$ and $X \subset f$, a must have the lowest MIS value in X and X must be a frequent itemset, which is ensured by the MS-Apriori algorithm. Hence, the support count of X is recorded. Since f is a frequent itemset and its support count is also recorded, then we can compute the confidence of $X \rightarrow Y$. This contradicts our assumption. ■

The lemma indicates that we need to record the support count of $f - \{a\}$. This is achieved by lines 13–14 in MS-Apriori (Fig. 2.6). All problems in Example 15 are solved. A similar rule generation function as genRules() in Apriori can be designed to generate rules with multiple minimum supports.

2.5 Mining Class Association Rules

The mining models studied so far do not use any targets. That is, any item can appear as a consequent or condition of a rule. However, in some applications, the user is interested in only rules with some fixed **target items** on the right-hand side. For example, the user has a collection of text documents from some topics (target items), and he/she wants to know what words are correlated with each topic. In [352], a data mining system based entirely on such rules (called **class association rules)** is reported, which is in production use in Motorola for many different applications. In the Web environment, class association rules are also useful because many types of Web data are in the form of transactions, e.g., search queries issued by users, and pages clicked by visitors. There are often target items as well, e.g., advertisements. Web sites want to know how user activities are associated with advertisements that they may like to view. This touches the issue of classification or prediction, which we will study in the next chapter.

2.5.1 Problem Definition

Let T be a transaction data set consisting of n transactions. Each transaction is labeled with a class y. Let I be the set of all items in T, Y be the set of all **class labels** (or target items) and $I \cap Y = \emptyset$. A **class association rule** (**CAR**) is an implication of the form

$$X \rightarrow y, \text{ where } X \subseteq I, \text{ and } y \in Y.$$

The definitions of **support** and **confidence** are the same as those for nor-

mal association rules. In general, a class association rule is different from a normal association rule in two ways:

1. The consequent of a CAR has only a single item, while the consequent of a normal association rule can have any number of items.
2. The consequent y of a CAR can only be from the class label set Y, i.e., $y \in Y$. No item from I can appear as the consequent, and no class label can appear as a rule condition. In contrast, a normal association rule can have any item as a condition or a consequent.

Objective: The problem of mining CARs is to generate the complete set of CARs that satisfies the user-specified minimum support (minsup) and minimum confidence (minconf) constraints.

Example 16: Figure 2.9 shows a data set which has seven text documents. Each document is a transaction and consists of a set of keywords. Each transaction is also labeled with a topic class (education or sport).

I = {Student, Teach, School, City, Game, Baseball, Basketball, Team, Coach, Player, Spectator}
Y = {Education, Sport}.

	Transactions	**Class**
doc 1:	Student, Teach, School	: Education
doc 2:	Student, School	: Education
doc 3:	Teach, School, City, Game	: Education
doc 4:	Baseball, Basketball	: Sport
doc 5:	Basketball, Player, Spectator	: Sport
doc 6:	Baseball, Coach, Game, Team	: Sport
doc 7:	Basketball, Team, City, Game	: Sport

Fig. 2.9. An example of a data set for mining class association rules

Let minsup = 20% and minconf = 60%. The following are two examples of class association rules:

Student, School → Education [sup= 2/7, conf = 2/2]
Game → Sport [sup= 2/7, conf = 2/3]. ■

A question that one may ask is: can we mine the data by simply using the Apriori algorithm and then perform a post-processing of the resulting rules to select only those class association rules? In principle, the answer is yes because CARs are a special type of association rules. However, in practice this is often difficult or even impossible because of combinatorial explosion, i.e., the number of rules generated in this way can be huge.

2.5.2 Mining Algorithm

Unlike normal association rules, CARs can be mined directly in a single step. The key operation is to find all **ruleitems** that have support above minsup. A **ruleitem** is of the form:

$$(condset, y),$$

where **condset** $\subseteq I$ is a set of items, and $y \in Y$ is a class label. The support count of a condset (called **condsupCount**) is the number of transactions in T that contain the condset. The support count of a ruleitem (called **rulesupCount**) is the number of transactions in T that contain the condset and are labeled with class y. Each ruleitem basically represents a rule:

$$condset \rightarrow y,$$

whose **support** is ($rulesupCount$ / n), where n is the total number of transactions in T, and whose **confidence** is ($rulesupCount$ / $condsupCount$).

Ruleitems that satisfy the minsup are called **frequent** ruleitems, while the rest are called infrequent ruleitems. For example, ({Student, School}, Education) is a ruleitem in T of Fig. 2.9. The support count of the condset {Student, School} is 2, and the support count of the ruleitem is also 2. Then the support of the ruleitem is 2/7 (= 28.6%), and the confidence of the ruleitem is 100%. If minsup = 10%, then the ruleitem satisfies the minsup threshold. We say that it is frequent. If minconf = 80%, then the ruleitem satisfies the minconf threshold. We say that the ruleitem is **confident**. We thus have the class association rule:

Student, School → Education [sup= 2/7, conf = 2/2].

The rule generation algorithm, called **CAR-Apriori**, is given in Fig. 2.10, which is based on the Apriori algorithm. Like the Apriori algorithm, CAR-Apriori generates all the frequent ruleitems by making multiple passes over the data. In the first pass, it computes the support count of each 1-ruleitem (containing only one item in its condset) (line 1). The set of all 1-candidate ruleitems considered is:

$$C_1 = \{(\{i\}, y) \mid i \in I, \text{and } y \in Y\},$$

which basically associates each item in I (or in the transaction data set T) with every class label. Line 2 determines whether the candidate 1-ruleitems are frequent. From frequent 1-ruleitems, we generate 1-condition CARs (rules with only one condition) (line 3). In a subsequent pass, say k, it starts with the seed set of (k–1)-ruleitems found to be frequent in the (k–1)-th pass, and uses this seed set to generate new possibly frequent k-ruleitems, called **candidate *k*-ruleitems** (C_k in line 5). The actual support

Algorithm CAR-Apriori(T)
1 $C_1 \leftarrow$ init-pass(T); // the first pass over T
2 $F_1 \leftarrow \{f \mid f \in C_1, f.\ rulesupCount / n \geq minsup\}$;
3 $CAR_1 \leftarrow \{f \mid f \in F_1, f.rulesupCount / f.condsupCount \geq minconf\}$;
4 **for** ($k = 2$; $F_{k-1} \neq \emptyset$; k++) **do**
5 $C_k \leftarrow$ CARcandidate-gen(F_{k-1});
6 **for** each transaction $t \in T$ **do**
7 **for** each candidate $c \in C_k$ **do**
8 **if** $c.condset$ is contained in t **then** // c is a subset of t
9 $c.condsupCount$++;
10 **if** $t.class = c.class$ **then**
11 $c.rulesupCount$++
12 **endfor**
13 **end-for**
14 $F_k \leftarrow \{c \in C_k \mid c.rulesupCount / n \geq minsup\}$;
15 $CAR_k \leftarrow \{f \mid f \in F_k, f.rulesupCount / f.$condsupCount $\geq minconf\}$;
16 **endfor**
17 **return** $CAR \leftarrow \bigcup_k CAR_k$;

Fig. 2.10. The CAR-Apriori algorithm

counts, both *condsupCount* and *rulesupCount*, are updated during the scan of the data (lines 6–13) for each candidate k-ruleitem. At the end of the data scan, it determines which of the candidate k-ruleitems in C_k are actually frequent (line 14). From the frequent k-ruleitems, line 15 generates k-condition CARs (class association rules with k conditions).

One interesting note about ruleitem generation is that if a ruleitem/rule has a confidence of 100%, then extending the ruleitem with more conditions (adding items to its condset) will also result in rules with 100% confidence although their supports may drop with additional items. In some applications, we may consider these subsequent rules **redundant** because additional conditions do not provide any more information. Then, we should not extend such ruleitems in candidate generation for the next level, which can reduce the number of generated rules substantially. If desired, redundancy handling can be added in the CAR-Apriori algorithm easily.

The CARcandidate-gen() function is very similar to the candidate-gen() function in the Apriori algorithm, and it is thus omitted. The only difference is that in CARcandidate-gen() ruleitems with the same class are joined by joining their condsets.

Example 17: Let us work on a complete example using our data in Fig. 2.9. We set minsup = 15%, and minconf = 70%

F_1: {({School}, Education):(3, 3), ({Student}, Education):(2, 2),
({Teach}, Education):(2, 2), ({Baseball}, Sport):(2, 2),

({Basketball}, Sport):(3, 3), ({Game}, Sport):(3, 2),
({Team}, Sport):(2, 2)}

Note: The two numbers within the parentheses after each ruleitem are its *condSupCount* and *ruleSupCount* respectively.

CAR_1: School → Education [sup = 3/7, conf = 3/3]
Student → Education [sup = 2/7, conf = 2/2]
Teach → Education [sup = 2/7, conf = 2/2]
Baseball → Sport [sup = 2/7, conf = 2/2]
Basketball → Sport [sup = 3/7, conf = 3/3]
Game → Sport [sup = 2/7, conf = 2/3]
Team → Sport [sup = 2/7, conf = 2/2]

Note: We do not deal with rule redundancy in this example.

C_2: {({School, Student}, Education), ({School, Teach}, Education),
({Student, Teach}, Education), ({Baseball, Basketball}, Sport),
({Baseball, Game}, Sport), ({Baseball, Team}, Sport),
({Basketball, Game}, Sport), ({Basketball, Team}, Sport),
({Game, Team}, Sport)}

F_2: {({School, Student}, Education):(2, 2),
({School, Teach}, Education):(2, 2), ({Game, Team}, Sport):(2, 2)}

CAR_2: School, Student → Education [sup = 2/7, conf = 2/2]
School, Teach → Education [sup = 2/7, conf = 2/2]
Game, Team → Sport [sup = 2/7, conf = 2/2] ■

We note that for many applications involving target items, the data sets used are relational tables. They need to be converted to transaction forms before mining. We can use the method in Sect. 2.3 for the purpose.

Example 18: In Fig. 2.11(A), the data set has three data attributes and a class attribute with two possible values, positive and negative. It is converted to the transaction data in Fig. 2.11(B). Notice that for each class, we only use its original value. There is no need to attach the attribute "Class"

Attribute1	Attribute2	Atribute3	Class
a	a	x	positive
b	n	y	negative

(A) Table data

t_1: (Attribute1, a), (Attribute2, a), (Attribute3, x) : Positive
t_2: (Attribute1, b), (Attribute2, n), (Attribute3, y) : negative

(B) Transaction data

Fig. 2.11. Converting a table data set (A) to a transaction data set (B)

because there is no ambiguity. As discussed in Sect. 2.3, for each numeric attribute, its value range needs to be discretized into intervals either manually or automatically before conversion and rule mining. There are many discretization algorithms. Interested readers are referred to [151]. ■

2.5.3 Mining with Multiple Minimum Supports

The concept of mining with multiple minimum supports discussed in Sect. 2.4 can be incorporated in class association rule mining in two ways:

1. **Multiple minimum class supports:** The user can specify different minimum supports for different classes. For example, the user has a data set with two classes, Yes and No. Based on the application requirement, he/she may want all rules of class Yes to have the minimum support of 5% and all rules of class No to have the minimum support of 20%.
2. **Multiple minimum item supports:** The user can specify a minimum item support for every item (either a class item/label or a non-class item). This is more general and is similar to normal association rule mining discussed in Sect. 2.4.

For both approaches, similar mining algorithms to that given in Sect. 2.4 can be devised. The *support difference constraint* in Sect. 2.4.1 can be incorporated as well. Like normal association rule mining with multiple minimum supports, by setting minimum class and/or item supports to more than 100% for some items, the user effectively instructs the algorithm not to generate rules involving only these items.

Finally, although we have discussed only multiple minimum supports so far, we can easily use different minimum confidences for different classes as well, which provides an additional flexibility in applications.

2.6 Basic Concepts of Sequential Patterns

Association rule mining does not consider the order of transactions. However, in many applications such orderings are significant. For example, in market basket analysis, it is interesting to know whether people buy some items in sequence, e.g., buying bed first and then buying bed sheets some time later. In Web usage mining, it is useful to find navigational patterns in a Web site from sequences of page visits of users (see Chap. 12). In text mining, considering the ordering of words in a sentence is vital for finding linguistic or language patterns (see Chap. 11). For these applications, association rules will not be appropriate. Sequential patterns are needed. Be-

low, we define the problem of mining sequential patterns and introduce the main concepts involved.

Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items. A **sequence** is an ordered list of itemsets. Recall an **itemset** X is a non-empty set of items $X \subseteq I$. We denote a sequence s by $\langle a_1 a_2 ... a_r \rangle$, where a_i is an itemset, which is also called an **element** of s. We denote an element (or an itemset) of a sequence by $\{x_1, x_2, \ldots, x_k\}$, where $x_j \in I$ is an item. We assume without loss of generality that items in an element of a sequence are in **lexicographic order**. An item can occur only once in an element of a sequence, but can occur multiple times in different elements. The **size** of a sequence is the number of elements (or itemsets) in the sequence. The **length** of a sequence is the number of items in the sequence. A sequence of length k is called a ***k*-sequence**. If an item occurs multiple times in different elements of a sequence, each occurrence contributes to the value of k. A sequence $s_1 = \langle a_1 a_2 ... a_r \rangle$ is a **subsequence** of another sequence $s_2 = \langle b_1 b_2 ... b_v \rangle$, or s_2 is a **supersequence** of s_1, if there exist integers $1 \leq j_1 < j_2 < ... < j_{r-1} < j_r \leq v$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}, ..., a_r \subseteq b_{j_r}$. We also say that s_2 **contains** s_1.

Example 19: Let $I = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The sequence $\langle \{3\}\{4, 5\}\{8\} \rangle$ is contained in (or is a subsequence of) $\langle \{6\}\ \{3, 7\}\{9\}\{4, 5, 8\}\{3, 8\} \rangle$ because $\{3\} \subseteq \{3, 7\}$, $\{4, 5\} \subseteq \{4, 5, 8\}$, and $\{8\} \subseteq \{3, 8\}$. However, $\langle \{3\}\{8\} \rangle$ is not contained in $\langle \{3, 8\} \rangle$ or vice versa. The size of the sequence $\langle \{3\}\{4, 5\}\{8\} \rangle$ is 3, and the length of the sequence is 4. ■

Objective: Given a set S of input **data sequences** (or **sequence database**), the problem of mining sequential patterns is to find all sequences that have a user-specified **minimum support**. Each such sequence is called a **frequent sequence**, or a **sequential pattern**. The **support** for a sequence is the fraction of total data sequences in S that contains this sequence.

Example 20: We use the market basket analysis as an example. Each sequence in this context represents an ordered list of transactions of a particular customer. A transaction is a set of items that the customer purchased at a time (called the transaction time). Then transactions in the sequence are ordered by increasing transaction time. Table 2.1 shows a transaction database which is already sorted according to customer ID (the major key) and transaction time (the minor key). Table 2.2 gives the data sequences (also called **customer sequences**). Table 2.3 gives the output sequential patterns with the minimum support of 25%, i.e., two customers. ■

Table 2.1. A set of transactions sorted by customer ID and transaction time

Customer ID	Transaction Time	Transaction (items bought)
1	July 20, 2005	30
1	July 25, 2005	90
2	July 9, 2005	10, 20
2	July 14, 2005	30
2	July 20, 2005	10, 40, 60, 70
3	July 25, 2005	30, 50, 70, 80
4	July 25, 2005	30
4	July 29, 2005	30, 40, 70, 80
4	August 2, 2005	90
5	July 12, 2005	90

Table 2.2. The sequence database produced from the transactions in Table 2.1.

Customer ID	Data Sequence
1	⟨{30} {90}⟩
2	⟨{10, 20} {30} {10, 40, 60, 70}⟩
3	⟨{30, 50, 70, 80}⟩
4	⟨{30} {30, 40, 70, 80} {90}⟩
5	⟨{90}⟩

Table 2.3. The final output sequential patterns

	Sequential Patterns with Support ≥ 25%
1-sequences	⟨{30}⟩, ⟨{40}⟩, ⟨{70}⟩, ⟨{80}⟩, ⟨{90}⟩
2-sequences	⟨{30} {40}⟩, ⟨{30} {70}⟩, ⟨{30}, {90}⟩, ⟨{30, 70}⟩, ⟨{30, 80}⟩, ⟨{40, 70}⟩, ⟨{70, 80}⟩
3-sequences	⟨{30} {40, 70}⟩, ⟨{30, 70, 80}⟩

2.7 Mining Sequential Patterns Based on GSP

This section describes two algorithms for mining sequential patterns based on the GSP algorithm in [500]: the original GSP, which uses a single minimum support, and MS-GSP, which uses multiple minimum supports.

2.7.1 GSP Algorithm

GSP works in almost the same way as the Apriori algorithm. We still use F_k to store the set of all frequent k-sequences, and C_k to store the set of all

Algorithm GSP(S)

```
1   C_1 ← init-pass(S);                          // the first pass over S
2   F_1 ← {⟨{f}⟩| f ∈ C_1, f.count/n ≥ minsup};  // n is the number of sequences in S
3   for (k = 2; F_{k-1} ≠ ∅; k++) do             // subsequent passes over S
4     C_k ← candidate-gen-SPM(F_{k-1});
5     for each data sequence s ∈ S do            // scan the data once
6       for each candidate c ∈ C_k do
7         if c is contained in s then
8           c.count++;                           // increment the support count
9       endfor
10    endfor
11    F_k ← {c ∈ C_k | c.count/n ≥ minsup}
12  endfor
13  return ⋃_k F_k;
```

Fig. 2.12. The GSP Algorithm for generating sequential patterns

Function candidate-gen-SPM(F_{k-1}) // SPM: Sequential Pattern Mining

1. **Join step.** Candidate sequences are generated by joining F_{k-1} with F_{k-1}. A sequence s_1 joins with s_2 if the subsequence obtained by dropping the first item of s_1 is the same as the subsequence obtained by dropping the last item of s_2. The candidate sequence generated by joining s_1 with s_2 is the sequence s_1 extended with the last item in s_2. There are two cases:
   - the added item forms a separate element if it was a separate element in s_2, and is appended at the end of s_1 in the merged sequence, and
   - the added item is part of the last element of s_1 in the merged sequence otherwise.

   When joining F_1 with F_1, we need to add the item in s_2 both as part of an itemset and as a separate element. That is, joining $\langle\{x\}\rangle$ with $\langle\{y\}\rangle$ gives us both $\langle\{x, y\}\rangle$ and $\langle\{x\}\{y\}\rangle$. Note that x and y in $\{x, y\}$ are ordered.
2. **Prune step.** A candidate sequence is pruned if any one of its (k–1)-subsequences is infrequent (without minimum support).

Fig. 2.13. The candidate-gen-SPM function

candidate k-sequences. The algorithm is given in Fig. 2.12. The main difference is in the candidate generation, candidate-gen-SPM(), which is given in Fig. 2.13. We use an example to illustrate the function.

Example 21: Table 2.4 shows F_3, and C_4 after the join and prune steps. In the join step, the sequence ⟨{1, 2}{4}⟩ joins with ⟨{2}{4, 5}⟩ to produce ⟨{1, 2}{4, 5}⟩, and joins with ⟨{2}{4}{6}⟩ to produce ⟨{1, 2}{4} {6}⟩. The other sequences cannot be joined. For instance, ⟨{1}{4, 5}⟩ does not join with any sequence since there is no sequence of the form ⟨{4, 5}{x}⟩ or ⟨{4, 5, x}⟩. In the prune step, ⟨{1, 2}{4} {6}⟩ is removed since ⟨{1}{4} {6}⟩ is not in F_3. ■

Table 2.4. Candidate generation: an example

Frequent 3-sequences	Candidate 4-sequences	
	after joining	after pruning
⟨{1, 2} {4}⟩	⟨{1, 2} {4, 5}⟩	⟨{1, 2} {4, 5}⟩
⟨{1, 2} {5}⟩	⟨{1, 2} {4} {6}⟩	
⟨{1} {4, 5}⟩		
⟨{1, 4} {6}⟩		
⟨{2} {4, 5}⟩		
⟨{2} {4} {6}⟩		

2.7.2 Mining with Multiple Minimum Supports

As in association rule mining, using a single minimum support in sequential pattern mining is also a limitation for many applications because some items appear very frequently in the data, while some others appear rarely.

Example 22: One of the Web mining tasks is the mining of comparative sentences such as "*the picture quality of camera X is better than that of camera Y*." from product reviews, forum postings and blogs (see Chap. 11). Such a sentence usually contains a comparative indicator word such as better in the example. We want to discover linguistic patterns involving a set of given comparative indicators, e.g., better, more, less, ahead, win, superior, etc. Some of these indicators (e.g., more and better) appear very frequently in natural language sentences, while some others (e.g., win and ahead) appear rarely. In order to find patterns that contain such rare indicators, we have to use a very low minsup. However, this causes patterns involving frequent indicators to generate a huge number of spurious patterns. Moreover, we need a way to tell the algorithm that we only want patterns that contain at least one comparative indicator. Using GSP with a single minsup is no longer appropriate. The multiple minimum supports model solves both problems nicely. ■

We again use the concept of **minimum item supports** (**MIS**). The user is allowed to assign each item a MIS value. By providing different MIS values for different items, the user essentially expresses different support requirements for different sequential patterns. To ease the task of specifying many MIS values by the user, the same strategies as those for mining association rules can also be applied here (see Sect. 2.4.2).

Let MIS(i) be the MIS value of item i. The **minimum support** of a sequential pattern P is the lowest MIS value among the items in the pattern. Let the set of items in P be: $i_1, i_2, \ldots, i_r$. The minimum support for P is:

```
Algorithm MS-GSP(S, MS)                          // MS stores all MIS values
1   M ← sort(I, MS);                             // according to MIS(i)'s stored in MS
2   L ← init-pass(M, S);                         // make the first pass over S
3   F1 ← {⟨{l}⟩ | l ∈ L, l.count/n ≥ MIS(l)};    // n is the size of S
4   for (k = 2; Fk–1 ≠ ∅; k++) do
5     if k = 2 then
6       Ck ← level2-candidate-gen-SPM(L)
7     else Ck ← MScandidate-gen-SPM(Fk–1)
8     endif
9     for each data sequence s ∈ S do
10      for each candidate c ∈ Ck do
11        if c is contained in s then
12          c.count++
13        if c' is contained in s, where c' is c after an occurrence of
              c.minMISItem is removed from c then
14          c.rest.count++          // c.rest: c without c.minMISItem
15      endfor
16    endfor
17    Fk ← {c ∈ Ck | c.count/n ≥ MIS(c.minMISItem)}
18  endfor
19  return F ← ⋃k Fk;
```

Fig. 2.14. The MS-GSP algorithm

$$\text{minsup}(P) = min(\text{MIS}(i_1), \text{MIS}(i_2), \ldots, \text{MIS}(i_r)).$$

The new algorithm, called **MS-GSP**, is given in Fig. 2.14. It generalizes the GSP algorithm in Fig. 2.12. Like GSP, MS-GSP is also based on level-wise search. Line 1 sorts the items in ascending order according to their MIS values stored in *MS*. Line 2 makes the first pass over the sequence data using the function init-pass(), which performs the same function as that in MS-Apriori to produce the seeds set L for generating the set of candidate sequences of length 2, i.e., C_2. Frequent 1-sequences (F_1) are obtained from L (line 3).

For each subsequent pass, the algorithm works similarly to MS-Apriori. The function level2-candidate-gen-SPM() can be designed based on level2-candidate-gen in MS-Apriori and the join step in Fig. 2.13. MScandidate-gen-SPM() is, however, complex, which we will discuss shortly.

In line 13, *c.minMISItem* gives the item that has the lowest MIS value in the candidate sequence c. Unlike that in MS-Apriori, where the first item in each itemset has the lowest MIS value, in sequential pattern mining the item with the lowest MIS value may appear anywhere in a sequence. Similar to those in MS-Apriori, lines 13 and 14 are used to ensure that all sequential rules can be generated after MS-GSP without scanning the original data. Note that in traditional sequential pattern mining, sequential rules are not defined. We will define several types in Sect. 2.9.

Let us now discuss MScandidate-gen-SPM(). In MS-Apriori, the ordering of items is not important and thus we put the item with the lowest MIS value in each itemset as the first item of the itemset, which simplifies the join step. However, for sequential pattern mining, we cannot artificially put the item with the lowest MIS value as the first item in a sequence because the ordering of items is significant. This causes problems for joining.

Example 23: Assume we have a sequence $s_1 = \langle\{1, 2\}\{4\}\rangle$ in F_3, from which we want to generate candidate sequences for the next level. Suppose that item 1 has the lowest MIS value in s_1. We use the candidate generation function in Fig. 2.13. Assume also that the sequence $s_2 = \langle\{2\}\{4, 5\}\rangle$ is not in F_3 because its minimum support is not satisfied. Then we will not generate the candidate $\langle\{1, 2\}\{4, 5\}\rangle$. However, $\langle\{1, 2\}\{4, 5\}\rangle$ can be frequent because items 2, 4, and 5 may have higher MIS values than item 1. ■

To deal with this problem, let us make an observation. The problem only occurs when the first item in the sequence s_1 or the last item in the sequence s_2 is the only item with the lowest MIS value, i.e., no other item in s_1 (or s_2) has the same lowest MIS value. If the item (say x) with the lowest MIS value is not the first item in s_1, then s_2 must contain x, and the candidate generation function in Fig. 2.13 will still be applicable. The same reasoning goes for the last item of s_2. Thus, we only need special treatment for these two cases.

Let us see how to deal with the first case, i.e., the first item is the only item with the lowest MIS value. We use an example to develop the idea. Assume we have the frequent 3-sequence of $s_1 = \langle\{1, 2\}\{4\}\rangle$. Based on the algorithm in Fig. 2.13, s_1 may be extended to generate two possible candidates using $\langle\{2\}\{4\}\{x\}\rangle$ and $\langle\{2\}\{4, x\}\rangle$

$c_1 = \langle\{1, 2\}\{4\}\{x\}\rangle$ and $c_2 = \langle\{1, 2\}\{4, x\}\rangle$,

where x is an item. However, $\langle\{2\}\{4\}\{x\}\rangle$ and $\langle\{2\}\{4, x\}\rangle$ may not be frequent because items 2, 4, and x may have higher MIS values than item 1, but we still need to generate c_1 and c_2 because they can be frequent. A different join strategy is thus needed.

We observe that for c_1 to be frequent, the subsequence $s_2 = \langle\{1\}\{4\}\{x\}\rangle$ must be frequent. Then, we can use s_1 and s_2 to generate c_1. c_2 can be generated in a similar manner with $s_2 = \langle\{1\}\{4, x\}\rangle$. s_2 is basically the subsequence of c_1 (or c_2) without the second item. Here we assume that the MIS value of x is higher than item 1. Otherwise, it falls into the second case.

Let us see the same problem for the case where the last item has the only lowest MIS value. Again, we use an example to illustrate. Assume we have the frequent 3-sequence $s_2 = \langle\{3, 5\}\{1\}\rangle$. It can be extended to produce two possible candidates based on the algorithm in Fig. 2.13,

Function MScandidate-gen-SPM(F_{k-1})

1 **Join Step**. Candidate sequences are generated by joining F_{k-1} with F_{k-1}.

2 **if** the MIS value of the first item in a sequence (denoted by s_1) is less than (<) the MIS value of every other item in s_1 **then** // s_1 and s_2 can be equal
Sequence s_1 joins with s_2 if (1) the subsequences obtained by dropping the second item of s_1 and the last item of s_2 are the same, and (2) the MIS value of the last item of s_2 is greater than that of the first item of s_1. Candidate sequences are generated by extending s_1 with the last item of s_2:
- **if** the last item l in s_2 is a separate element **then**
  $\{l\}$ is appended at the end of s_1 as a separate element to form a candidate sequence c_1.
  if (the length and the size of s_1 are both 2) AND (the last item of s_2 is greater than the last item of s_1) **then** // maintain lexicographic order
  l is added at the end of the last element of s_1 to form another candidate sequence c_2.
- **else if** ((the length of s_1 is 2 and the size of s_1 is 1) AND (the last item of s_2 is greater than the last item of s_1)) OR (the length of s_1 is greater than 2) **then**
  the last item in s_2 is added at the end of the last element of s_1 to form the candidate sequence c_2.

3 **elseif** the MIS value of the last item in a sequence (denoted by s_2) is less than (<) the MIS value of every other item in s_2 **then**
A similar method to the one above can be used in the reverse order.

4 **else** use the **Join Step** in Fig. 2.13

5 **Prune step**: A candidate sequence is pruned if any one of its (k–1)-subsequences is infrequent (without minimum support) except the subsequence that does not contain the item with strictly the lowest MIS value.

Fig. 2.15. The MScandidate-gen-SPM function

$c_1 = \langle\{x\}\{3, 5\}\{1\}\rangle$, and $c_2 = \langle\{x, 3, 5\}\{1\}\rangle$.

For c_1 to be frequent, the subsequence $s_1 = \langle\{x\}\{3\}\{1\}\rangle$ has to be frequent (we assume that the MIS value of x is higher than that of item 1). Thus, we can use s_1 and s_2 to generate c_1. c_2 can be generated with $s_1 = \langle\{x, 3\}\{1\}\rangle$. s_1 is basically the subsequence of c_1 (or c_2) without the second last item.

The MScandidate-gen-SPM() function is given in Fig. 2.15, which is self-explanatory. Some special treatments are needed for 2-sequences because the same s_1 (or s_2) may generate two candidate sequences. We use two examples to show the working of the function.

Example 24: Consider the items 1, 2, 3, 4, 5, and 6 with their MIS values,

MIS(1) = 0.03 MIS(2) = 0.05 MIS(3) = 0.03
MIS(4) = 0.07 MIS(5) = 0.08 MIS(6) = 0.09.

The data set has 100 sequences. The following frequent 3-sequences are in F_3 with their actual support counts attached after ":":

(a). ⟨{1}{4}{5}⟩:4	(b). ⟨{1}{4}{6}⟩:5	(c). ⟨{1}{5}{6}⟩:6
(d). ⟨{1}{5, 6}⟩:5	(e). ⟨{1}{6}{3}⟩:4	(f). ⟨{6}{3}{6}⟩:9
(g). ⟨{5, 6}{3}⟩:5	(h). ⟨{5}{4}{3}⟩:4	(i). ⟨{4}{5}{3}⟩:7.

For sequence (a) (= s_1), item 1 has the lowest MIS value. It cannot join with sequence (b) because condition (1) in Fig. 2.15 is not satisfied. However, (a) can join with (c) to produce the candidate sequence, ⟨{1}{4}{5}{6}⟩. (a) can also join with (d) to produce ⟨{1}{4}{5, 6}⟩. (b) can join with (e) to produce ⟨{1}{4}{6}{3}⟩, which is pruned subsequently because ⟨{1}{4}{3}⟩ is infrequent. (d) and (e) can be joined to give ⟨{1}{5, 6}{3}⟩, but it is pruned because ⟨{1}{5}{3}⟩ does not exist. (e) can join with (f) to produce ⟨{1}{6}{3}{6}⟩ which is done in line 4 because both item 1 and item 3 in (e) have the same MIS value. However, it is pruned because ⟨{1}{3}{6}⟩ is infrequent. We do not join (d) and (g), although they can be joined based on the algorithm in Fig. 2.13, because the first item of (d) has the lowest MIS value and we use a different join method for such sequences.

Now we look at 3-sequences whose last item has strictly the lowest MIS value. (i) (= s_1) can join with (h) (= s_2) to produce ⟨{4}{5}{4}{3}⟩. However, it is pruned because ⟨{4}{4}{3}⟩ is not in F_3. ■

Example 25: Now we consider generating candidates from frequent 2-sequences, which is special as we noted earlier. We use the same items and MIS values in Example 24. The following frequent 2-sequences are in F_2 with their actual support counts attached after ":":

(a). ⟨{1}{5}⟩:6	(b). ⟨{1}{6}⟩:7	(c) ⟨{5}{4}⟩:8
(d). ⟨{1, 5}⟩:6	(e). ⟨{1, 6}⟩:6.	

(a) can join with (b) to produce both ⟨{1}{5}{6}⟩ and ⟨{1}{5, 6}⟩. (b) can join with (d) to produce ⟨{1, 5}{6}⟩. (e) can join with (a) to produce ⟨{1, 6}{5}⟩. Clearly, there are other joins. Again, (a) will not join with (c). ■

Note that the **support difference constraint** in Sect. 2.4.1 can also be included. We omitted it to simplify the algorithm as it is already complex. Also, the user can instruct the algorithm to generate only certain sequential patterns or not to generate others by setting the MIS values suitably.

2.8 Mining Sequential Patterns Based on PrefixSpan

We now introduce another sequential pattern mining algorithm, called PrefixSpan [439], which does not generate candidates. Different from the GSP

algorithm [500], which can be regarded as performing breadth-first search to find all sequential patterns, PrefixSpan performs depth-first search.

2.8.1 PrefixSpan Algorithm

It is easy to introduce the original PrefixSpan algorithm using an example.

Example 26: Consider again mining sequential patterns from Table 2.2 with minsup = 25%. PrefixSpan first sorts all items in each element (or itemset) as shown in the table. Then, by one scan of the sequence database, it finds all frequent items, i.e., 30, 40, 70, 80 and 90. The corresponding length one sequential patterns are ⟨{30}⟩, ⟨{40}⟩, ⟨{70}⟩, ⟨{80}⟩ and ⟨{90}⟩.

We notice that the complete set of sequential patterns can actually be divided into five mutually exclusive subsets: the subset with prefix ⟨{30}⟩, the subset with prefix ⟨{40}⟩, the subset with prefix ⟨{70}⟩, the subset with prefix ⟨{80}⟩, and the subset with prefix ⟨{90}⟩. We only need to find the five subsets one by one.

To find sequential patterns having prefix ⟨{30}⟩, the algorithm extends the prefix by adding items to it one at a time. To add the next item x, there are two possibilities, i.e., x joining the last itemset of the prefix (i.e., ⟨{30, x}⟩) and x forming a separate itemset (i.e., ⟨{30}{x}⟩). PrefixSpan performs the task by first forming the ⟨{30}⟩-projected database and then finding all the cases of the two types in the projected database. The projected database is produced as follows: If a sequence contains item 30, then the suffix following the first 30 is extracted as a sequence in the projected database. Furthermore, since infrequent items cannot appear in a sequential pattern, all infrequent items are removed from the projection. The first sequence in our example, ⟨{30}{90}⟩, is projected to ⟨{90}⟩. The second sequence, ⟨{10, 20}{30}{10, 40, 60, 70}⟩, is projected to ⟨{40, 70}⟩, where the infrequent items 10 and 60 are removed. The third sequence ⟨{30, 50, 70, 80}⟩ is projected to ⟨{_, 70, 80}⟩, where the infrequent item 50 is removed. Note that the underline symbol "_" in this projection denotes that the items (only 30 in this case) in the last itemset of the prefix are in the same itemset as items 50, 70 and 80 in the sequence. The fourth sequence is projected to ⟨{30, 40, 70, 80}{90}⟩. The projection of the last sequence is empty since it does not contain item 30. The final projected database for prefix ⟨{30}⟩ contains the following sequences:

⟨{90}⟩, ⟨{40, 70}⟩, ⟨{_, 70, 80}⟩, and ⟨{30, 40, 70, 80}{90}⟩

By scanning the projected database once, PrefixSpan finds all possible one item extensions to the prefix, i.e., all x's for ⟨{30, x}⟩ and all x's for ⟨{30}{x}⟩. Let us discuss the details.

Find All Frequent Patterns of the Form ⟨{30, x}⟩: Two templates {_, x} and {30, x} are used to match each projected sequence to accumulate the support count for each possible x (here x matches any item). If in the same sequence multiple matches are found with the same x, they are only counted once. Note that in general, the second template should use the last itemset in the prefix rather than only its last item. In our example, they are the same because there is only one item in the last itemset of the prefix.

Find All Frequent Patterns of the Form ⟨{30}{x}⟩: In this case, x's are frequent items in the projected database that are not in the same itemset as the last item of the prefix.

Let us continue with our example. It is easy to check that both items 70 and 80 are in the same itemset as 30. That is, we have two frequent sequences ⟨{30, 70}⟩ and ⟨{30, 80}⟩. The support count of ⟨{30, 70}⟩ is 2 based on the projected database; one from the projected sequence ⟨{_, **70**, 80}⟩ (a {_, x} match) and one from the projected sequence ⟨{**30**, 40, **70**, 80}{90}⟩ (a {30, x} match). In both cases, the x's are the same, i.e., 70. Similarly, the support count of ⟨{30, 80}⟩ is 2 as well and thus frequent.

It is also easy to check that items 40, 70, and 90 are also frequent but not in the same itemset as 30. Thus, ⟨{30}{40}⟩, ⟨{30}{70}⟩, and ⟨{30}{90}⟩ are three sequential patterns. The set of sequential patterns having prefix ⟨{30}⟩ can be further divided into five mutually exclusive subsets: the ones with prefixes ⟨{30, 70}⟩, ⟨{30, 80}⟩, ⟨{30}{40}⟩, ⟨{30}{70}⟩, and ⟨{30}{90}⟩.

We can recursively find the five subsets by forming their corresponding projected databases. For example, to find sequential patterns having prefix ⟨{30}{40}⟩, we can form the ⟨{30}{40}⟩-projected database containing projections ⟨{_, 70}⟩ and ⟨{_, 70, 80}{90}⟩. Template ⟨{_, x}⟩ has two matches and in both cases x is 70. Thus, ⟨{30}{40, 70}⟩ is output as a sequential pattern. Since there is no other frequent item in this projected database, the prefix cannot grow longer. The depth-first search returns from this branch.

After completing the mining of the ⟨{30}⟩-projected database, we find all sequential patterns with prefix ⟨{30}⟩, i.e., ⟨{30}⟩, ⟨{30}{40}⟩, ⟨{30}{40, 70}⟩, ⟨{30}{70}⟩, ⟨{30}{90}⟩, ⟨{30, 70}⟩, ⟨{30, 80}⟩ and ⟨{30, 70, 80}⟩

By forming and mining the ⟨{40}⟩-, ⟨{70}⟩-, ⟨{80}⟩- and ⟨{90}⟩-projected databases, the remaining sequential patterns can be found. ■

The pseudo code of PrefixSpan can be found in [439]. Comparing to the breadth-first search of GSP, the key advantage of PrefixSpan is that it does not generate any candidates. It only counts the frequency of local items. With a low minimum support, a huge number of candidates can be generated by GSP, which can cause memory and computational problems.

2.8.2 Mining with Multiple Minimum Supports

The PrefixSpan algorithm can be adapted to mine with multiple minimum supports. Again, let MIS(i) be the user-specified **minimum item support** of item i. Let φ be the user-specified support difference threshold in the **support difference constraint** (Sect. 2.4.1), i.e., $|sup(i) - sup(j)| \leq \varphi$, where i and j are items in the same sequential pattern, and $sup(x)$ is the actual support of item x in the sequence database S. PrefixSpan can be modified as follows. We call the modified algorithm **MS-PS**.

1. Find every item i whose actual support in the sequence database S is at least MIS(i). i is called a frequent item.
2. Sort all the discovered frequent items in ascending order according to their MIS values. Let $i_1, \ldots, i_u$ be the frequent items in the sorted order.
3. For each item i_k in the above sorted order,
   (i) identify all the data sequences in S that contain i_k and at the same time remove every item j in each sequence that does not satisfy $|sup(j) - sup(i_k)| \leq \varphi$. The resulting set of sequences is denoted by S_k. Note that we are not using i_k as the prefix to project the database S.
   (ii) call the function r-PrefixSpan(i_k, S_k, count(MIS(i_k))) (*r*estricted *PrefixSpan*), which finds all sequential patterns that contain i_k, i.e., no pattern that does not contain i_k should be generated. r-PrefixSpan() uses count(MIS(i_k)) (the minimum support count in terms of the number of sequences) as the only minimum support for mining in S_k. The sequence count is easier to use than the MIS value in percentage, but they are equivalent. Once the complete set of such patterns is found from S_k, All occurrences of i_k are removed from S.

r-PrefixSpan() is almost the same as PrefixSpan with one important difference. During each recursive call, either the prefix or every sequence in the projected database must contain i_k because, as we stated above, this function finds only those frequent sequences that contain i_k. Another minor difference is that the support difference constraint needs to be checked during each projection as $sup(i_k)$ may not be the lowest in the pattern.

Example 27: Consider mining sequential patterns from Table 2.5. Let MIS(20) = 30% (3 sequences in minimum support count), MIS(30) = 20% (2 sequences), MIS(40) = 30% (3 sequences), and the MIS values for the rest of the items be 15% (2 sequences). We ignore the support difference constraint as it is simple. In step 1, we find three frequent items, 20, 30 and 40. After sorting in step 2, we have (30, 20, 40). We then go to step 3.

In the first iteration of step 3, we work on i_1 = 30. Step 3(i) gives us the second, fourth and sixth sequences in Table 2.5, i.e.,

Table 2.5. An example of a sequence database

Sequence ID	Data Sequence
1	⟨{20, 50}⟩
2	⟨{40}{30}{40, 60}⟩
3	⟨{40, 90, 120}⟩
4	⟨{30}{20, 40}{40, 100}⟩
5	⟨{20, 40}{10}⟩
6	⟨{40}{30}{110}⟩
7	⟨{20}{80}{70}⟩

S_1 = {⟨{40}{30}{40, 60}⟩, ⟨{30}{20, 40}{40, 100}⟩, ⟨{40}{30}{110}⟩}.

We then run r-PrefixSpan(30, S_1, 3) in step 3(ii). The frequent items in S_1 are 30, and 40. They both have the support of 3 sequences. The length one frequent sequence is only ⟨{30}⟩. ⟨{40}⟩ is not included because we require that every frequent sequence must contain 30. We next find frequent sequences having prefix ⟨{30}⟩. The database S_1 is projected to give ⟨{40}⟩ and ⟨{40}{40}⟩. 20, 60 and 100 have been removed because their supports in S_1 are less than the required support for item 30 (i.e., 3 sequences). For the same reason, the projection of ⟨{40}{30}{110}⟩ is empty. Thus, we find a length two frequent sequence ⟨{30}{40}⟩. In this case, there is no item in the same itemset as 30 to form a frequent sequence of the form ⟨{30, x}⟩.

Next, we find frequent sequences with prefix ⟨{40}⟩. We again project S_1, which gives us only ⟨{30}{40}⟩ and ⟨{30}⟩. ⟨{40, 100}⟩ is not included because it does not contain 30. This projection gives us another length two frequent sequence ⟨{40}{30}⟩. The first iteration of step 3 ends.

In the second iteration of step 3, we work on i_2 = 20. Step 3(i) gives us the first, fourth, fifth and seventh sequences in Table 2.5 with item 30 removed, S_2 = {⟨{20, 50}⟩, ⟨{20, 40}{40, 100}⟩, ⟨{20, 40}{10}⟩, ⟨{20}{80}{70}⟩}. It is easy to see that only item 20 is frequent, and thus only a length one frequent sequence is generated, ⟨{20}⟩.

In the third iteration of step 3, we work on i_3 = 40. We can verify that again only one frequent sequence, i.e., ⟨{40}⟩, is found.

The final set of sequential patterns generated from the sequence database in Table 2.5 is {⟨{30}⟩, ⟨{20}⟩, ⟨{40}⟩, ⟨{40}{30}⟩, ⟨{30}{40}⟩}. ■

2.9 Generating Rules from Sequential Patterns

In classic sequential pattern mining, no rules are generated. It is, however, possible to define and generate many types of rules. This section intro-

duces only three types, **sequential rules**, **label sequential rules** and **class sequential rules**, which have been used in Web usage mining and Web content mining (see Chaps. 11 and 12).

2.9.1 Sequential Rules

A **sequential rule** (**SR**) is an implication of the form, $X \rightarrow Y$, where Y is a sequence and X is a **proper subsequence** of Y, i.e., X is a subsequence of Y and the length Y is greater than the length of X. The **support** of a sequential rule, $X \rightarrow Y$, in a sequence database S is the fraction of sequences in S that contain Y. The **confidence** of a sequential rule, $X \rightarrow Y$, in S is the proportion of sequences in S that contain X also contain Y.

Given a minimum support and a minimum confidence, according to the downward closure property, all the rules can be generated from frequent sequences without going to the original sequence data. Let us see an example of a sequential rule found from the data sequences in Table 2.6.

Table 2.6. An example of a sequence database for mining sequential rules

	Data Sequence
1	⟨{1}{3}{5}{7, 8, 9}⟩
2	⟨{1}{3}{6}{7, 8}⟩
3	⟨{1, 6}{7}⟩
4	⟨{1}{3}{5, 6}⟩
5	⟨{1}{3}{4}⟩

Example 28: Given the sequence database in Table 2.6, the minimum support of 30% and the minimum confidence of 60%, one of the sequential rules found is the following,

⟨{1}{7}⟩ → ⟨{1}{3}{7, 8}⟩ [sup = 2/5, conf = 2/3]

Data sequences 1, 2 and 3 contain ⟨{1}{7}⟩, and data sequences 1 and 2 contain ⟨{1}{3}{7, 8}⟩. ■

If multiple minimum supports are used, we can employ the results of multiple minimum support pattern mining to generate all the rules.

2.9.2 Label Sequential Rules

Sequential rules may not be restrictive enough in some applications. We introduce a special kind of sequential rules called **label sequential rules**. A label sequential rule (LSR) is of the form, $X \rightarrow Y$, where Y is a sequence

and X is a sequence produced from Y by replacing some of its items with wildcards. A wildcard is denoted by an "*" which matches any item. These replaced items are usually very important and are called **labels**. The labels are a small subset of all the items in the data.

Example 29: Given the sequence database in Table 2.6, the minimum support of 30% and the minimum confidence of 60%, one of the label sequential rules found is the following,

$$\langle\{1\}\{*\}\{7, *\}\rangle \rightarrow \langle\{1\}\{3\}\{7, 8\}\rangle \quad [\text{sup} = 2/5, \text{conf} = 2/2].$$

Notice the confidence change compared to the rule in Example 28. The supports of the two rules are the same. In this case, data sequences 1 and 2 contain $\langle\{1\}\{*\}\{7, *\}\rangle$, and they also contain $\langle\{1\}\{3\}\{7, 8\}\rangle$. Items 3 and 8 are labels. ■

LSRs are useful because in some applications we need to predict the labels in an input sequence, e.g., items 3 and 8 above. The confidence of the rule simply gives us the estimated probability that the two "*"s are 3 and 8 given that an input sequence contains $\langle\{1\}\{*\}\{7, *\}\rangle$. We will see an application of LSRs in Chap. 11, where we want to predict whether a word in a comparative sentence is an entity (e.g., a product name), which is a label.

Note that due to the use of wildcards, frequent sequences alone are not sufficient for computing rule confidences. Scanning the data is needed. Notice also that the same pattern may appear in a data sequence multiple times. Rule confidences thus can be defined in different ways according to application needs. The wildcards may also be restricted to match only certain types of items to make the label prediction meaningful and unambiguous (see some examples in Chap. 11).

2.9.3 Class Sequential Rules

Class sequential rules (CSR) are analogous to class association rules (CAR). Let S be a set of data sequences. Each sequence is also labeled with a class y. Let I be the set of all items in S, and Y be the set of all class labels, $I \cap Y = \varnothing$. Thus, the input data D for mining is represented with $\{(s_1, y_1), (s_2, y_2), \ldots, (s_n, y_n)\}$, where s_i is a sequence in S and $y_i \in Y$ is its class. A **class sequential rule** (**CSR**) is of the form

$$X \rightarrow y, \text{ where } X \text{ is a sequence, and } y \in Y.$$

A data instance (s_i, y_i) is said to **cover** a CSR, $X \rightarrow y$, if X is a subsequence of s_i. A data instance (s_i, y_i) is said to **satisfy** a CSR if X is a subsequence of s_i and $y_i = y$.

Example 30: Table 2.7 gives an example of a sequence database with five data sequences and two classes, c_1 and c_2. Using the minimum support of 30% and the minimum confidence of 60%, one of the discovered CSRs is:

$\langle\{1\}\{3\}\{7, 8\}\rangle \rightarrow c_1$ [sup = 2/5, conf = 2/3].

Data sequences 1 and 2 satisfy the rule, and data sequences 1, 2 and 5 cover the rule. ■

Table 2.7. An example of a sequence database for mining CSRs

	Data Sequence	Class
1	$\langle\{1\}\{3\}\{5\}\{7, 8, 9\}\rangle$	c_1
2	$\langle\{1\}\{3\}\{6\}\{7, 8\}\rangle$	c_1
3	$\langle\{1, 6\}\{9\}\rangle$	c_2
4	$\langle\{3\}\{5, 6\}\rangle$	c_2
5	$\langle\{1\}\{3\}\{4\}\{7, 8\}\rangle$	c_2

As in class association rule mining, we can modify the GSP and PrefixSpan algorithms to produce algorithms for mining all CSRs. Similarly, we can also use multiple minimum class supports and/or multiple minimum item supports as in class association rule mining.

Bibliographic Notes

Association rule mining was introduced in 1993 by Agrawal et al. [9]. Since then, numerous research papers have been published on the topic. This short chapter only introduces some basics, and it, by no means, does justice to the huge body of work in the area. The bibliographic notes here should help you explore further.

Since given a data set, a minimum support and a minimum confidence, the solution (the set of frequent itemsets or the set of rules) is unique, most papers improve the mining efficiency. The most well-known algorithm is the Apriori algorithm proposed by Agrawal and Srikant [11], which has been studied in this chapter. Another important algorithm is the **FP-growth** algorithm proposed by Han et al. [220]. The algorithm compresses the data and stores it in memory using a frequent pattern tree. It then mines all frequent itemsets without candidate generation. Other notable algorithms include those by Agarwal et al. [2], Mannila et al. [361], Park et al. [435], Zaki et al. [589], etc. An efficiency comparison of various algorithms was reported by Zheng et al. [616].

Apart from performance improvements, several variations of the original model were also proposed. Srikant and Agrawal [499], and Han and Fu

[217] proposed two algorithms for mining **generalized association rules** or **multi-level association rules**. Liu et al. [344] extended the original model to take **multiple minimum supports**, which was also studied by Wang et al. [534], Seno and Karypis [482], Xiong et al. [562], etc. Srikant et al. [502] proposed to mine association rules with **item constraints**. The model restricts the rules that should be generated. Ng et al. [408] generalized the idea, which was followed by many subsequent papers on the topic of **constrained rule mining**.

It is well known that association rule mining often generates a huge number of frequent itemsets and rules. Bayardo [42], and Lin and Kedem [334] introduced the problem of mining **maximal frequent itemsets**, which are itemsets with no frequent supersets. Improved algorithms are reported in many papers [e.g., 2, 73]. Since maximal pattern mining only finds longest patterns, the support information of their subsets, which are obviously also frequent, is not found. As a result, association rules cannot be generated. The next significant development was the mining of **closed frequent itemsets** studied by Pasquier et al. [436], Zaki and Hsiao [588], and Wang et al. [529]. Closed itemsets are better than maximal frequent itemsets because closed frequent itemsets provide a lossless concise representation of all frequent itemsets.

Other developments on association rules include **cyclic association rules** proposed by Ozden et al. [420], **periodic patterns** by Yang et al. [571], **negative association rules** by Savasere [476] and Wu et al. [560], **weighted association rules** by Wang et al. [539], **association rules with numerical variables** by Webb [541], **class association rules** by Liu et al. [343], **high-performance rule mining** by Buehrer et al. [72] and many others. Recently, Cong et al. [112, 113] introduced association rule mining from bioinformatics data, which typically have a very large number of attributes (more than ten thousands) but only a very small number of records or transactions (less than 100).

Another major research area on association rules is the **interestingness** of discovered rules. Since an association rule miner often generates a huge number of rules, it is very difficult, if not impossible, for human users to inspect them in order to find those truly interesting rules. Researchers have proposed many techniques to help users identify such rules easily [e.g., 43, 283, 342, 345, 346, 352, 421, 492, 511, 522, 535, 565]. A deployed data mining system that uses some of the techniques is reported in [352].

Regarding sequential pattern mining, the first algorithm was proposed by Agrawal and Srikant [12], which was a direct application of the Apriori algorithm. Improvements were made subsequently by several researchers, e.g., Ayres et al. [29], Pei et al. [439], Srikant and Agrawal [500], Zaki [586], etc. The MS-GSP and MS-PS algorithms for mining sequential pat-

terns with multiple minimum supports and the support difference constraint are introduced in this book. Label and class sequential rules have been used in [255, 256] for mining comparative sentences from text documents.

There are several publicly available implementations of algorithms for mining frequent itemsets, maximal frequent itemsets, closed frequent itemsets, and sequential patterns from various research groups, most notably from those of Jiawei Han, Johnanne Gehrke, and Mohammed Zaki. There were also two workshops dedicated to frequent itemset mining organized by Roberto Bayardo, Bart Goethals, and Mohammed J. Zaki, which reported many efficient implementations. The workshop Web sites are http://fimi.cs.helsinki.fi/fimi03/ and http://fimi.cs.helsinki.fi/fimi04/.