4.1 Introduction

Many of the pattern finding algorithms such as those for decision tree building, classification rule induction, and data clustering that are frequently used in data mining have been developed in the machine learning research community. Frequent pattern and association rule mining is one of the few exceptions to this tradition. Its introduction boosted data mining research and its impact is tremendous. The basic algorithms are simple and easy to implement. In this chapter the most fundamental algorithms of frequent pattern and association rule mining, known as Apriori and AprioriTid [3, 4], and Apriori’s extension to sequential pattern mining, known as AprioriAll [6, 5], are explained based on the original papers with working examples, and performance analysis of Apriori is shown using a freely available implementation [1] for a dataset in UCI repository [8]. Since Apriori is so fundamental and the form of database is limited to market transaction, there have been many works for improving computational efficiency, finding more compact representation, and extending the types of data that can be handled. Some of the important works are also briefly described as advanced topics.

4.2 Algorithm Description

4.2.1 Mining Frequent Patterns and Association Rules

One of the most popular data mining approaches is to find frequent itemsets from a transaction dataset and derive association rules. The problem is formally stated as follows. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of items. Let $\mathcal{D}$ be a set of transactions, where each transaction $t$ is a set of items such that $t \subseteq \mathcal{I}$. Each transaction has a unique identifier, called its $TID$. A transaction $t$ contains $X$, a set of some items in $\mathcal{I}$, if $X \subseteq t$. An association rule is an implication of the form $X \Rightarrow Y$, where $X \subseteq \mathcal{I}, Y \subseteq \mathcal{I},$ and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds in $\mathcal{D}$ with confidence $c$ ($0 \leq c \leq 1$) if the fraction of transactions that also contain $Y$ in those which contain $X$ in $\mathcal{D}$ is $c$. The rule $X \Rightarrow Y$ (and equivalently $X \cup Y$) has support $s$ ($0 \leq s \leq 1$) in $\mathcal{D}$ if the fraction of transactions in $\mathcal{D}$ that contain $X \cup Y$ is $s$. Given a set of transactions $\mathcal{D}$, the problem of mining association rules is to generate all association rules that have support and confidence no less than the user-specified minimum support (called $\minsup$) and minimum confidence (called $\minconf$), respectively.

Finding frequent\(^2\) itemsets (itemsets with support no less than $\minsup$) is not trivial because of the computational complexity due to combinatorial explosion. Once

\(^1\)An alternative support definition is the absolute count of frequency. In this chapter the latter definition is also used where appropriate.

\(^2\)The Apriori paper [3] uses “large” to mean “frequent,” but large is often associated with the number of items in the itemset. Thus, we prefer to use “frequent.”
frequent itemsets are obtained, it is straightforward to generate association rules with confidence no less than \textit{minconf}. Apriori and AprioriTid, proposed by R. Agrawal and R. Srikant, are seminal algorithms that are designed to work for a large transaction dataset [3].

4.2.1.1 Apriori

Apriori is an algorithm to find all sets of items (itemsets) that have support no less than \textit{minsup}. The support for an itemset is the ratio of the number of transactions that contain the itemset to the total number of transactions. Itemsets that satisfy minimum support constraint are called frequent itemsets. Apriori is characterized as a level-wise complete search (breadth first search) algorithm using anti-monotonicity property of itemsets: "If an itemset is not frequent, any of its superset is never frequent," which is also called the downward closure property. The algorithm makes multiple passes over the data. In the first pass, the support of individual items is counted and frequent items are determined. In each subsequent pass, a seed set of itemsets found to be frequent in the previous pass is used for generating new potentially frequent itemsets, called candidate itemsets, and their actual support is counted during the pass over the data.

At the end of the pass, those satisfying minimum support constraint are collected, that is, frequent itemsets are determined, and they become the seed for the next pass. This process is repeated until no new frequent itemsets are found.

By convention, Apriori assumes that items within a transaction or itemset are sorted in lexicographic order. The number of items in an itemset is called its size and an itemset of size \(k\) is called a \(k\)-itemset. Let the set of frequent itemsets of size \(k\) be \(F_k\) and their candidates be \(C_k\). Both \(F_k\) and \(C_k\) maintain a field, support count.

Apriori algorithm is given in Algorithm 4.1. The first pass simply counts item occurrences to determine the frequent 1-itemsets. A subsequent pass consists of two phases. First, the frequent itemsets \(F_{k-1}\) found in the \((k-1)\)-th pass are used to generate the candidate itemsets \(C_k\) using the apriori-gen function. Next, the database is scanned and the support of candidates in \(C_k\) is counted. The subset function is used for this counting.

The apriori-gen function takes as argument \(F_{k-1}\), the set of all frequent \((k-1)\)-itemsets, and returns a superset of the set of all frequent \(k\)-itemsets. First, in the join steps, \(F_{k-1}\) is joined with \(F_{k-1}\).

\begin{verbatim}
insert into \(C_k\)
select \(p.fitemset_1, p.fitemset_2, \ldots, p.fitemset_{k-1}, q.fitemset_{k-1}\)
from \(F_{k-1} p, F_{k-1} q\)
where \(p.fitemset_1 = q.fitemset_1, \ldots, p.fitemset_{k-2} = q.fitemset_{k-2},\)
\(p.fitemset_{k-1} < q.fitemset_{k-1}\)
\end{verbatim}

Here, \(F_k p\) means that the itemset \(p\) is a frequent \(k\)-itemset, and \(p.fitemset_k\) is the \(k\)-th item of the frequent itemset \(p\).

Then, in the prune step, all the itemsets \(c \in C_k\) for which some \((k-1)\)-subset is not in \(F_{k-1}\) are deleted.
Algorithm 4.1 Apriori Algorithm

\begin{algorithm}
\begin{align*}
F_1 &= \{\text{frequent 1-itemsets}\}; \\
\text{for } (k = 2; F_{k-1} \neq \emptyset; k + +) \text{ do begin} \\
C_k &= \text{apriori-gen}(F_{k-1}); //\text{New candidates} \\
\text{foreach transaction } t \in D \text{ do begin} \\
C_t &= \text{subset}(C_k, t); //\text{Candidates contained in } t \\
\text{foreach candidate } c \in C_t \text{ do} \\
&\quad c.\text{count} + +; \\
&\quad F_k = \{c \in C_k | c.\text{count} \geq \minsup\}; \\
&\end{algorithm}
\end{algorithm}

\text{end} \]
\begin{algorithm}
\begin{align*}
\text{Answer} &= \bigcup_k F_k;
\end{align*}
\end{algorithm}

The subset function takes as arguments \(C_k\) and a transaction \(t\), and returns all the candidate itemsets contained in the transaction \(t\). For fast counting, Apriori adopts a hash-tree to store the candidate itemsets \(C_k\). Itemsets are stored in leaves. Every node is initially a leaf node, and the depth of the root node is defined to be 1. When the number of itemsets in a leaf node exceeds a specified threshold, the leaf node is converted to an interior node. An interior node at depth \(d\) points to nodes at depth \(d + 1\). Which branch to follow is decided by applying a hash function to the \(d\)-th item of the itemset. Thus, each leaf node is ensured to contain at most a certain number of itemsets (to be precise, this is true only when creating an interior node takes place at depth \(d\) smaller than \(k\)), and an itemset in the leaf node can be reached by successively hashing each item in the itemset in sequence from the root. Once the hash-tree is constructed, the subset function finds all the candidates contained in a transaction \(t\), starting from the root node. At the root node, every item in \(t\) is hashed, and each branch determined is followed one depth down. If a leaf node is reached, itemsets in the leaf that are in the transaction \(t\) are searched and those found are made reference to the answer set. If an interior node is reached by hashing the item \(i\), items that come after \(i\) in \(t\) are hashed recursively until a leaf node is reached. It is evident that itemsets in the leaves that are never reached are not contained in \(t\).

Clearly, any subset of a frequent itemset satisfies the minimum support constraint. The join operation is equivalent to extending \(F_{k-1}\) with each item in the database and then deleting those itemsets for which the \((k-1)\)-itemset obtained by deleting the \((k-1)\)-th item is not in \(F_{k-1}\). The condition \(p.fitemset_{k-1} < q.fitemset_{k-1}\) ensures that no duplication is made. The prune step where all the itemsets whose \((k-1)\)-subsets are not in \(F_{k-1}\) are deleted from \(C_k\) does not delete any itemset that could be in \(F_k\). Thus, \(C_k \supseteq F_k\), and Apriori algorithm is correct.

The remaining task is to generate the desired association rules from the frequent itemsets. A straightforward algorithm for this task is as follows. To generate rules,
4.2 Algorithm Description

all nonempty subsets of every frequent itemset \( f \) are enumerated and for every such subset \( a \), a rule of the form \( a \Rightarrow (f - a) \) is generated if the ratio of \( \text{support}(f) \) to \( \text{support}(a) \) is at least \( \text{minconf} \). Here, note that the confidence of the rule \( \hat{a} \Rightarrow (f - \hat{a}) \) cannot be larger than the confidence of \( a \Rightarrow (f - a) \) for any \( \hat{a} \subset a \). This in turn means that for a rule \( (f - a) \Rightarrow a \) to hold, all rules of the form \( (f - \hat{a}) \Rightarrow \hat{a} \) must hold. Using this property, the algorithm to generate association rules is given in Algorithm 4.2.

**Algorithm 4.2 Association Rule Generation Algorithm**

\[
\begin{align*}
H_1 &= \emptyset \quad //\text{Initialize} \\
\text{foreach: frequent } k\text{-itemset } f_k, k \geq 2 \text{ do begin} \\
&\quad A = (k - 1)\text{-itemsets } a_{k-1} \text{ such that } a_{k-1} \subset f_k; \\
&\quad \text{foreach } a_{k-1} \in A \text{ do begin} \\
&\quad \quad \text{conf} = \frac{\text{support}(f_k)}{\text{support}(a_{k-1})}; \\
&\quad \quad \text{if } (\text{conf} \geq \text{minconf}) \text{ then begin} \\
&\quad \quad \quad \text{output the rule } a_{k-1} \Rightarrow (f_k - a_{k-1}) \\
&\quad \quad \quad \quad \text{with confidence } = \text{conf} \text{ and support } = \text{support}(f_k); \\
&\quad \quad \quad \text{add } (f_k - a_{k-1}) \text{ to } H_1; \\
&\quad \quad \text{end} \\
&\quad \text{end} \\
&\quad \text{call ap-genrules}(f_k, H_1); \\
&\text{end} \\
\end{align*}
\]

**Procedure ap-genrules**

\[
\begin{align*}
&\text{if } (k > m + 1) \text{ then begin} \\
&\quad H_{m+1} = \text{apriori-gen}(H_m); \\
&\quad \text{foreach } h_{m+1} \in H_{m+1} \text{ do begin} \\
&\quad \quad \text{conf} = \frac{\text{support}(f_k)}{\text{support}(f_k - h_{m+1})}; \\
&\quad \quad \text{if } (\text{conf} \geq \text{minconf}) \text{ then} \\
&\quad \quad \quad \text{output the rule } f_k - h_{m+1} \Rightarrow h_{m+1} \\
&\quad \quad \quad \quad \text{with confidence } = \text{conf} \text{ and support } = \text{support}(f_k); \\
&\quad \quad \text{else} \\
&\quad \quad \quad \text{delete } h_{m+1} \text{ from } H_{m+1}; \\
&\quad \quad \text{end} \\
&\quad \text{call ap-genrules}(f_k, H_{m+1}); \\
&\text{end} \\
\end{align*}
\]

Apriori achieves good performance by reducing the size of candidate sets. However, in situations with very many frequent itemsets or very low minimum support, it still suffers from the cost of generating a huge number of candidate sets and scanning the database repeatedly to check a large set of candidate itemsets.
4.2.1.2 AprioriTid

AprioriTid is a variation of Apriori. It does not reduce the number of candidates but it does not use the database $\mathcal{D}$ for counting support after the first pass. It uses a new dataset $\mathcal{C}_k$. Each member of the set $\mathcal{C}_k$ is of the form $<\text{TID}, \{ID\}>$, where each $ID$ is the identifier of a potentially frequent $k$-itemset present in the transaction with identifier $\text{TID}$ except $k = 1$. For $k = 1$, $\mathcal{C}_1$ corresponds to the database $\mathcal{D}$, although conceptually each item $i$ is replaced by the itemset $\{i\}$. The member of $\mathcal{C}_k$ corresponding to a transaction $t$ is $<t.\text{TID}, \{c \in \mathcal{C}_k | c \text{ contained in } t\}>$.

The intuition for using $\mathcal{C}_k$ is that it will be smaller than the database $\mathcal{D}$ for large values of $k$ because some transactions may not contain any candidate $k$-itemset, in which case $\mathcal{C}_k$ does not have an entry for this transaction, or because very few candidates may be contained in the transaction and each entry may be smaller than the number of items in the corresponding transaction. AprioriTid algorithm is given in Algorithm 4.3. Here, $c[i]$ represents the $i$-th item in $k$-itemset $c$.

**Algorithm 4.3 AprioriTid Algorithm**

```
\begin{align*}
F_1 &= \{\text{frequent 1-itemsets}\}; \\
\mathcal{C}_1 &= \text{database } \mathcal{D}; \\
\text{for } (k = 2; F_{k-1} \neq \emptyset; k++) \text{ do begin} \\
C_k &= \text{apriori-gen}(F_{k-1}); \quad \text{//New candidates} \\
\mathcal{C}_k &= \emptyset; \\
\text{foreach entry } t \in \mathcal{C}_{k-1} \text{ do begin} \\
\text{// determine candidate itemsets in } C_k \text{ contained} \\
\text{// in the transaction with identifier } t.\text{TID} \\
C_t &= \{c \in C_k | (c - c[k]) \in t.\text{set-of-itemsets} \land \\
&(c - c[k - 1]) \in t.\text{set-of-itemsets}\}; \\
\text{foreach candidate } c \in C_t \text{ do} \\
&c.\text{count}++; \\
\text{if } (C_t \neq \emptyset) \text{ then } \mathcal{C}_k+= = \langle t.\text{TID}, C_t \rangle; \\
\text{end} \\
F_k &= \{c \in C_k | c.\text{count} \geq \text{minsup} \}; \\
\text{end} \\
\text{Answer} &= \bigcup_k F_k;
\end{align*}
```

Each $\mathcal{C}_k$ is stored in a sequential structure. A candidate $k$-itemset $c_k$ in $\mathcal{C}_k$ maintains two additional fields; generator and extensions, in addition to the field, support count. The generator field stores the IDs of the two frequent $(k - 1)$-itemsets whose join generated $c_k$. The extension field stores the IDs of all the $(k + 1)$-candidantes that are extensions of $c_k$. When a candidate $c_k$ is generated by joining $f_{k-1}^1$ and $f_{k-1}^2$, their IDs are saved in the generator field of $c_k$ and the ID of $c_k$ is added to the extension field of $f_{k-1}^1$. The $t.\text{set-of-itemsets}$ field of an entry $t$ in $\mathcal{C}_{k-1}$ gives the IDs of all
4.2 Algorithm Description

$(k - 1)$-candidates contained in $t.TID$. For each such candidate $c_{k-1}$ the extension field gives $T_k$, the set of IDs of all the candidate $k$-itemsets that are extensions of $c_{k-1}$. For each $c_k$ in $T_k$, the generator field gives the IDs of the two itemsets that generated $c_k$. If these itemsets are present in the entry for $t.set-of-itemsets$, it is concluded that $c_k$ is present in transaction $t.TID$, and $c_k$ is added to $C_t$.

AprioriTid has an overhead to calculate $C_k$ but an advantage that $C_k$ can be stored in memory when $k$ is large. It is thus expected that Apriori beats AprioriTid in earlier passes (small $k$) and AprioriTid beats Apriori in later passes (large $k$). Since both Apriori and AprioriTid use the same candidate generation procedure and therefore count the same itemsets, it is possible to make a combined use of these two algorithms in sequence. AprioriHybrid uses Apriori in the initial passes and switches to AprioriTid when it expects that the set $C_k$ at the end of the pass will fit in memory.

4.2.2 Mining Sequential Patterns

Agrawal and Srikant extended Apriori algorithm to the problem of sequential pattern mining [6]. In Apriori there is no notion of sequence, and thus, the problem of finding which items appear together can be viewed as finding intratransaction patterns. Here, sequence matters and the problem of finding sequential patterns can be viewed as intertransaction patterns.

Each transaction consists of sequence-id, transaction-time, and a set of items. The same sequence-id has no more than one transaction with the same transaction-time. A sequence is an ordered list of itemsets. Thus, a sequence consists of a list of sets of characters (items), rather than being simply a list of characters. The length of a sequence is the number of itemsets in the sequence. A sequence of length $k$ is called a $k$-sequence. Without loss of generality, the set of items is assumed to be mapped to a set of contiguous integers, and an itemset $i$ is denoted by $(i_1 i_2 \ldots i_m)$ where $i_j$ is an item. A sequence $s$ is denoted by $\langle s_1 s_2 \ldots s_n \rangle$. A sequence $\langle a_1 a_2 \ldots a_n \rangle$ is contained in another sequence $\langle b_1 b_2 \ldots b_m \rangle$ ($n \leq m$) if there exist integers $i_1 < i_2 < \cdots < i_n$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$, $\ldots$, $a_n \subseteq b_{i_n}$. All the transactions with the same sequence-id which are sorted by transaction-time together form a sequence (transaction sequence). A sequence-id supports a sequence $s$ if $s$ is contained in its transaction sequence. The support for a sequence is defined as the fraction of total number of sequence-ids that support this sequence. Likewise, the support for an itemset $i$ is defined as the fraction of sequence-ids that have items in $i$ in any one of their transactions. Note that this definition is different from that used in Apriori. Thus the itemset $i$ and the 1-sequence $\langle i \rangle$ have the same support.

Given a transaction database $D$, the problem of mining sequential patterns is to find the maximal\textsuperscript{3} sequences among all sequences that satisfy a certain user-specified minimum support constraint. Each such maximal sequence represents a sequential pattern. A sequence satisfying the minimum support constraint is called a frequent sequence (not necessarily maximal), and an itemset satisfying the minimum support constraint is called a frequent itemset.

\textsuperscript{3}Later R. Agrawal and R. Srikant removed this constraint in their generalized sequential patterns (GSP) [32].
constraint is called a frequent itemset, or fitemset for short. Any frequent sequence must be a list of fitemsets.

The algorithm consists of five phases: (1) sort phase, (2) fitemset phase, (3) transformation phase, (4) sequence phase, and (5) maximal phase. The first three are preprocessing phases and the last one is a postprocessing phase.

In the sort phase, the database $D$ is sorted with sequence-id as the major key and transaction-time as the minor key. In the fitemset phase, the set of all fitemsets is obtained using Apriori algorithm with the corresponding modification of counting a support, and is mapped to a set of contiguous integers. This makes comparing two fitemsets for equality in a constant time. Note that the set of all frequent 1-sequences are simultaneously found in this phase. In the transformation phase, each transaction is replaced by the set of all fitemsets that are in that transaction. If a transaction does not contain any fitemset, it is not retained in the transformed sequence. If a transaction sequence does not contain any fitemset, this sequence is removed from the transformed database, but it is still used in counting the total number of sequence-ids. After the transformation, a transaction sequence is represented by a list of sets of fitemsets. Each set of fitemsets is represented by $\{f_1, f_2, \ldots, f_n\}$, where $f_i$ is an fitemset. This transformation is designed for efficiently testing which given frequent sequences are contained in a transaction sequence. The transformed database is denoted as $D_T$.

The sequence phase is the main part where the frequent sequences are to be enumerated. Two families of algorithms are proposed: count-all and count-some. They differ in the way the frequent sequences are counted. Count-all algorithm counts all the frequent sequences, including nonmaximal sequences that must be pruned later, whereas count-some algorithm avoids counting sequences which are contained in a longer sequence because the final goal is to obtain only maximal sequences. Agrawal and Srikant developed one count-all algorithm called AprioriAll and two count-some algorithms called AprioriSome and DynamicSome. Here, only AprioriAll is explained due to the space limitation.

In the last maximal phase, maximal sequences are extracted from the set of all frequent sequences. The hash-tree (similar to the one used in the subset function in Apriori) is used to quickly find all subsequences of a given sequence.

### 4.2.2.1 AprioriAll

The algorithm is given in Algorithm 4.4. In each pass the frequent sequences from the previous pass are used to generate the candidate sequences and then their support is measured by making a pass over the database. At the end of the pass, the support of the candidates is used to determine the frequent sequences.

The apriori-gen-2 function takes as argument $F_{k-1}$, the set of all frequent $(k-1)$-sequences. First, join operation is performed as

```sql
insert into C_k
select p.fitemset_1, p.fitemset_2, \ldots, p.fitemset_{k-2}, q.fitemset_{k-1}
from F_{k-1} p, F_{k-1} q
where p.fitemset_1 = q.fitemset_1, \ldots, p.fitemset_{k-2} = q.fitemset_{k-2},
```
Algorithm 4.4 AprioriAll Algorithm

\[ F_1 = \{\text{frequent 1-sequences}\} ; \quad \text{// Result of itemset phase} \]
\[
\text{for } (k = 2; F_{k-1} \neq \emptyset; k++) \text{ do begin}
\]
\[
C_k = \text{apriori-gen-2}(F_{k-1}) ; \quad \text{//New candidate sequences}
\]
\[
\text{foreach transaction sequence } t \in D_T \text{ do begin}
\]
\[
C_t = \text{subseq}(C_k, t) ; \quad \text{//Candidate sequences contained in } t
\]
\[
\text{foreach candidate } c \in C_t \text{ do}
\]
\[
c.\text{count} ++ ;
\]
\[
F_k = \{c \in C_k \mid c.\text{count} \geq \text{mins}\}
\]
\[
\text{end}
\]
\[
\text{end}
\]
\[
\text{Answer} = \text{maximal sequences in } \bigcup_k F_k ;
\]

then, all the sequences \( c \in C_k \) for which some \((k - 1)\)-subsequence is not in \( F_{k-1} \) are deleted. The subseq function is similar to the subset function in Apriori. As in Apriori, the candidate sequences \( C_k \) are stored in a hash-tree to quickly find all candidates contained in a transaction sequence. Note that the transformed transaction sequence is a list of sets of itemsets and all the itemsets in a set have the same transaction-time, and no more than one transaction with the same transaction-time is allowed for the same sequence-id. This constraint has to be imposed in the subseq function.

4.2.3 Discussion

Both Apriori and AprioriTid need \( \text{mins} \) and \( \text{minconf} \) to be specified in advance. The algorithms have to be rerun each time these values are changed, throwing everything away that was obtained in previous runs. If no appropriate values for these thresholds are known in advance and we want to know how the results change with these values without rerunning the algorithms, the best we can do is to generate and count only those itemsets that appear at least once in the database without duplication and store them all in an efficient way. Note that Apriori generates candidates that do not exist in the database.

Apriori and AprioriTid use a hash-tree to store the candidate itemsets. Another data structure that is often used is a trie-structure [35, 9]. Each node in the depth \( k \) of the trie corresponds to a candidate \( k \)-itemset and stores the \( k \)-th item and the support of the itemset. As two frequent \( k \)-itemsets that share the first \((k - 1)\)-itemsets are siblings below their parent node at the depth \( k - 1 \) in the trie, the candidate generation is simply to join the two siblings, and extend the tree to one more depth below the first frequent \( k \)-itemset after pruning. In order to find the candidate \( k \)-itemsets that are contained in a transaction \( t \), each item in the transaction is fed from the root node and the branch is followed according to the succeeding item until a \( k \)-th item is reached. Many practical implementations of Apriori use this trie-structure to store not only candidates but also transactions [10, 9].
If we go a step further, we can get rid of generating candidate itemsets at all. Further, it is not necessary to enumerate all the frequent itemsets. These topics are discussed in Section 4.5.

Apriori and almost all other association rule minings use two-phase strategy: first mine frequent patterns and then generate association rules. This is not the sole way. Webb’s MagnumOpus uses another strategy that immediately generates a large subset of all association rules [38].

There are direct extensions of the original Apriori family. Use of taxonomy and incorporating temporal constraint are two examples. Generalized association rules [30] employ a set of user-specified taxonomies, which makes it possible to extract frequent itemsets that are expressed by higher concepts even when use of the base level concepts produces only infrequent itemsets. The basic algorithm is to add all ancestors of each item in a transaction to the transaction and then run Apriori algorithm. Several optimizations can be added to improve efficiency, one example being that the support for an itemset \( X \) that contains both an item \( x \) and its ancestor \( \hat{x} \) is the same as the support of the itemset \( X - \hat{x} \), and thus need not be counted. Generalized sequential patterns [32] place, in addition to the introduction of taxonomies, time constraints that specify a minimum and/or maximum time period between adjacent elements (itemsets) in a pattern and relax the restrictions that items in an element of a sequential pattern must come from the same transaction by allowing the items to be present in a set of transactions of the same sequence-id whose transaction-times are within a user-specified time window. It also finds all frequent sequential patterns (not limited to maximal sequential patterns). GSP algorithm runs about 20 times faster than AprioriAll, one reason being that GSP counts fewer candidates than AprioriAll.

### 4.3 Discussion on Available Software Implementations

There are many available implementations of Apriori ranging from free software to commercial products. Here, we will present only three well-known implementations which are freely downloadable via Internet.

The first one is an implementation embedded in the most famous open-source machine learning and data mining toolkit, Weka, provided by the University of Waikato [40]. Apriori in Weka can be used through Weka’s common graphical user interface together with many other algorithms that are available in Weka. The implementation includes Weka’s own extensions. For example, \( \text{minsup} \) is iteratively decreased from an upper bound \( U_{\text{minsup}} \) to a lower bound \( L_{\text{minsup}} \) with an interval \( \delta_{\text{minsup}} \). Further, in addition to confidence the metrics lift, leverage, and conviction are available to evaluate association rules. Lift and leverage are discussed in Section 4.5. Conviction [11] is a metric that was proposed to measure the departure from independence of an association rule taking implication into account. When using one of these metrics, its minimal value has to be given as a threshold.
4.4 Two Illustrative Examples

The second implementation is the one by Christian Borgelt [1], which is distributed under the terms of the GNU Lesser (Library) General Public License. This implementation is basically a command line application, and some graphical user interfaces are separately available. It essentially follows the flow of the original Apriori, but has its own extensions, too, to make it faster and to reduce its memory use. It employs a trie called the prefix tree to store both transactions and itemsets for efficient support counting [10]. The prefix tree is slightly different from the trie explained in Subsection 4.2.3. Optionally, the user can choose to use a simple list instead of a prefix tree to store transactions. Furthermore, this implementation can find not only frequent itemsets and association rules, but also closed itemsets, and maximal itemsets. Closed and maximal itemsets are discussed in Section 4.5. In addition, several metrics other than confidence, such as information gain, are also available in this implementation to evaluate and select association rules.

The third implementation is the one by Fence Bodon, which is freely distributed for research purposes [2]. This implementation is also trie-based, similar to Borgelt’s, but adopts a trie with a simpler structure, and computes only frequent itemsets and association rules. It works as a command line application, and accepts four arguments. The first three are mandatory: an input file, including transactions, an output file, and minsup. The fourth is minconf, which is optional. If minconf is given, association rules are mined, as well as frequent itemsets; otherwise, it outputs only frequent itemsets. This implementation is written in C++ to provide object-oriented components which can be easily reused to develop other Apriori-based algorithms.

4.4 Two Illustrative Examples

4.4.1 Working Examples

We will illustrate the detailed behavior of the aforementioned algorithms using a small database shown in Table 4.1, where SID and TT mean the sequence-id and transaction-time, respectively. We use this database in both association rule (frequent itemset) mining and maximal sequential pattern mining. In the former case SID and TT are ignored.

4.4.1.1 Frequent Itemset and Association Rule Mining

Suppose that we want to find frequent itemsets under minsup = 0.2 and association rules with minconf = 0.6.

Apriori (Algorithm 4.1)

Apriori first scans the whole database and derives a set of frequent 1-itemsets appearing in at least three transactions, $F_1 = \{a, c, d, f, g\}$. From this $F_1$, the apriori-gen function derives a set of candidate frequent 2-itemsets $C_2 = \{ac, ad, af, ag, cd, cg, df, dg, fg\}$. $C_2$ consists of all possible pairs of elements of $F_1$ since no pruning is made at this stage.
Next, Apriori computes their support by scanning the database using the subset function, which utilizes a hash-tree. Figure 4.1 briefly illustrates how a hash-tree is constructed and used. Suppose that the elements of $C_2$ are added into the hash-tree in lexicographic order, and the maximum number of itemsets allowed to be in a leaf node is 4. Thus, the number of itemsets in the root (leaf) node exceeds the threshold when the fifth itemset $cd$ is to be added. Then, the node is converted into an interior node.

![Example of hash-tree.](image)

(a) Make a hash-tree

(b) Check which itemsets are included in a transaction

**Figure 4.1** Example of hash-tree.
4.4 Two Illustrative Examples

one, and each itemset branches into the corresponding new leaf node according to the hash value given by the function \( h(x) \), where \( x \) is an item, the first item in each itemset in this case. We assume that \( h(x) \) is given in advance and is common for all nodes. Since the first four itemsets share the same first item \( a \), they fall into the same leaf node, while \( cd \) falls into a different one. When checking which of the candidates are included in a transaction, for example, Transaction 004, each item in the transaction is hashed at the root node. For example, by hashing \( c \) in \( cd \), it reaches the second left leaf node, and two itemsets \( cd \) and \( cf \) are found to be subsets of \( cd \) as shown in the left tree of Figure 4.1(b). Next, by hashing \( d \), \( df \) is found in the third left leaf node (the middle tree), but by hashing \( f \), no subset of \( cd \) is found in the rightmost leaf node (the right tree). As a result, the support counts of these itemsets found, \( cd \), \( cf \), and \( df \), are increased by 1. Note that, after all the transactions have been processed, the frequencies of the candidates \( af \) and \( ag \) are found to be 0. This means that Apriori may generate candidates that do not exist in a given database.

After this support counting, \( F_2 = \{cd, cf, df, dg\} \) is derived. These frequent 2-itemsets in \( F_2 \) are used as the seeds of frequent 3-itemsets. The itemsets \( cd \) and \( cf \) in \( F_2 \) sharing the first item \( c \) are joined and yield a new candidate \( cd \) by apriori-gen because \( df \) is also included in \( F_2 \). The itemsets \( df \) and \( dg \) are also joined as well, but the resulting candidate is pruned because its subset \( fg \) is not included in \( F_2 \). Consequently, \( C_3 \), a set of candidate frequent 3-itemsets, consists of \( cd \) only. Then, Apriori counts its support by scanning the database again, and derives \( F_3 = \{cd\} \).

No candidate frequent 4-itemsets can be generated from this \( F_3 \) because it contains only one itemset. Thus, Apriori terminates.

**AprioriTid** (Algorithm 4.3)

Apriori has to scan the whole database three times to obtain these frequent itemsets, but AprioriTid (Algorithm 4.3) scans it only once for the first pass, and makes and uses new datasets \( C_1 \) and \( C_2 \) to count the support of candidates in \( C_2 \) and \( C_3 \), respectively. Figure 4.2 illustrates how AprioriTid finds frequent itemsets from these datasets. \( \overline{C}_2 \) is generated while counting the support of each candidate in \( C_2 \), whereas \( \overline{C}_1 \) is generated directly from the given database. Suppose \( t = \langle 001, \{c, d\} \rangle \in C_1 \). Then, a candidate \( cd \) in \( C_2 \) is added to \( C_t \) because \( t \)-set-of-itemsets (\( \{\{c\}, \{d\}\} \)) contains both 1-itemsets constituting \( cd \). More precisely, \( cd \) is added to \( C_t \) because it is a union of two 1-itemsets in \( t \), which means Transaction 001 supports \( cd \). No other candidate is added to \( C_t \) as Transaction 001 does not support any other candidate in \( C_2 \). Then, the support count of \( cd \) is increased by 1, and \( \langle 001, \{cd\} \rangle \) is added to \( \overline{C}_2 \). Similarly, \( \langle 003, \{ac\} \rangle \) is added to \( \overline{C}_2 \) because Transaction 003 supports \( ac \in C_2 \), although an entry corresponding to \( \langle 002, \{f\} \rangle \) of \( \overline{C}_1 \) is not because Transaction 002 does not support any 2-itemsets. Eventually, \( \overline{C}_2 \) has 9 entries, as shown in Figure 4.2, whose size is smaller than that of the given database. \( \overline{C}_3 \) is generated in the same manner during the support counting of candidates in \( C_3 \). Since the unique candidate in \( C_3 \) is \( cd \), only the three entries of \( \overline{C}_2 \), including both \( cd \) and \( cf \), whose union is \( cd \), survive in \( \overline{C}_3 \). Note that \( \overline{C}_3 \) is generated, but actually never used because \( C_4 \) becomes empty.
Association rules (Algorithm 4.2)
Next, association rules are generated from the found frequent itemsets according to Algorithm 4.2 for the given minconf = 0.6. Let us consider frequent 2-itemsets, \(cd, cf, df\), and \(dg\), first. It is obvious that only two kinds of rules can be generated from each itemset. Table 4.2 summarizes the resulting rules and their confidence. The association rules 1 and 8 are the outputs by Algorithm 4.2 because they satisfy the minconf constraint. The procedure \(\text{ap-genrules}\) is called for each of these satisfactory rules, but it outputs nothing because it no longer generates other rules from the 2-itemsets.

Then, Algorithm 4.2 tries to generate association rules from the frequent 3-itemset, \(cdf\). First, it generates three association rules with 1-item consequent as shown in the left half of Table 4.3. Algorithm 4.2 returns all of them as they satisfy the minconf constraint. After that, the procedure \(\text{ap-genrules}\) is called, taking \(cdf\) and \(\{c, d, f\}\)

<table>
<thead>
<tr>
<th>TABLE 4.2</th>
<th>Association Rules Generated from Frequent 2-Itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.</td>
<td>Rule</td>
</tr>
<tr>
<td>1</td>
<td>(c \Rightarrow d)</td>
</tr>
<tr>
<td>2</td>
<td>(d \Rightarrow c)</td>
</tr>
<tr>
<td>3</td>
<td>(c \Rightarrow f)</td>
</tr>
<tr>
<td>4</td>
<td>(f \Rightarrow c)</td>
</tr>
</tbody>
</table>
4.4 Two Illustrative Examples

The table shows the association rules generated from frequent 3-itemsets.

<table>
<thead>
<tr>
<th>No.</th>
<th>Rule</th>
<th>Confidence</th>
<th>No.</th>
<th>Rule</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>cd ⇒ f</td>
<td>0.60</td>
<td>12</td>
<td>f ⇒ cd</td>
<td>0.43</td>
</tr>
<tr>
<td>10</td>
<td>cf ⇒ d</td>
<td>0.75</td>
<td>13</td>
<td>d ⇒ cf</td>
<td>0.33</td>
</tr>
<tr>
<td>11</td>
<td>df ⇒ c</td>
<td>0.75</td>
<td>14</td>
<td>c ⇒ df</td>
<td>0.43</td>
</tr>
</tbody>
</table>

As its arguments, a set of 2-itemsets \{cd, cf, df\} is derived by the function \text{apriori-gen} called within \text{ap-genrules}, each of which is used as the consequent of a new association rule. The resulting three rules are shown in the right half of Table 4.3. But, none of them can be the outputs because their confidence is less than the specified \text{minconf} = 0.6. Since 3-item consequents cannot be obtained from \text{cd}f, \text{ap-genrules} terminates, and Algorithm 4.2 terminates too because \text{F}_4 = \emptyset.

### 4.4.1.2 Sequential Pattern Mining

Next, we find frequent maximal sequential patterns from the same transaction database in Table 4.1 by using \text{AprioriAll} (Algorithm 4.4) for \text{minsup} = 0.3. Figure 4.3 illustrates the flow of the first three phases, that is, sort phase, itemset phase, and transformed database.

![Figure 4.3](image-url)
Frequent Sequences and Candidate Sequences

<table>
<thead>
<tr>
<th>F₁</th>
<th>C₂</th>
<th>F₂</th>
<th>C₃</th>
<th>F₃</th>
<th>C₄</th>
<th>F₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>124</td>
<td>124</td>
<td>124</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>23</td>
<td>22</td>
<td>134</td>
<td>134</td>
<td>134</td>
</tr>
<tr>
<td>3</td>
<td>52</td>
<td>52</td>
<td>24</td>
<td>224</td>
<td>224</td>
<td>224</td>
</tr>
<tr>
<td>4</td>
<td>53</td>
<td>53</td>
<td>244</td>
<td>244</td>
<td>244</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>124</td>
<td>124</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>23</td>
<td>22</td>
<td>134</td>
<td>134</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>52</td>
<td>24</td>
<td>224</td>
<td>224</td>
<td>224</td>
</tr>
<tr>
<td></td>
<td>53</td>
<td>53</td>
<td>244</td>
<td>244</td>
<td>244</td>
<td></td>
</tr>
</tbody>
</table>

The transformation phase on this example. In the sort phase, transactions in the database are sorted with sequence-id (SID) as the major key and transaction-time (TT) as the minor key. Then, in the itemset phase, itemsets are derived in the similar manner to Apriori. Note that the support of an itemset is the number of transaction sequences, including the itemset, but not the number of transactions including it. Thus, the resulting set of frequent 1-itemsets in this case is \{c, d, f\}. In the transformation phase, each transaction sequence is transformed into a list of sets of itemsets as shown in the bottom of Figure 4.3 by replacing each transaction in the sequence with a set of itemsets the transaction contains. Note that the second transaction is dropped in the transaction sequence 4 because it consists of only one nonfrequent itemset \{a\}.

AprioriAll generates a set of candidate sequences C₂ from F₁ by calling the function apriori-gen-2. The resulting C₂ is shown in Table 4.4. The function apriori-gen-2 is similar to apriori-gen, but differs in its join operation. The join operation of apriori-gen-2 generates two new \(k\)-sequences from two \((k - 1)\)-sequences whenever they are joinable, while the join operation of apriori-gen generates only one \(k\)-itemset from two \((k - 1)\)-itemsets. For example, when deriving C₂, both two sequences \(\langle12\rangle\) and \(\langle21\rangle\) are generated from \(\langle1\rangle\) and \(\langle2\rangle\). In addition, \(\langle11\rangle\) is also generated by joining the identical sequence \(\langle1\rangle\). This is necessary to generate a sequence in which multiple occurrences of an itemset is allowed.

Counting the support of each candidate sequence is done in the similar way as Apriori using a hash-tree, and F₂, a set of frequent 2-sequences, is derived as shown in Table 4.4. This F₂ is used to generate a set of candidate sequences C₃ as well. Note that from \(\langle11\rangle\) and \(\langle12\rangle\), a 3-sequence \(\langle112\rangle\) is generated by joining them, but not \(\langle121\rangle\) because its subsequence \(\langle21\rangle\) is not included in F₂. This process consisting of the candidate generation and support counting is repeated until no more frequent sequences are derived. In this example, since no candidate of 5-sequences can be generated from F₄, F₅ becomes empty and thus, the iteration terminates. Finally, AprioriAll outputs \(\langle1424\rangle, \langle2424\rangle, \langle3424\rangle, \langle11\rangle, \langle13\rangle, \langle52\rangle\) as the maximal frequent sequences as the other frequent sequences are included in one of them.

4.4.2 Performance Evaluation

In this section, we discuss the performance of Apriori with respect to its runtime, the number of derived association rules and frequent itemsets when \(\minsup,\ minconf\), and the number of transactions are varied. We used the implementation by Christian Borgelt [1] for this assessment because it provides options that allow us to simulate a
4.4 Two Illustrative Examples

Figure 4.4  Runtime for various minsup and minconf values.

naive implementation closest to the original Apriori. Thus, we disabled its functions of sorting items with respect to their support and of filtering unused items from transactions.

As a benchmark dataset, we used the Mushroom dataset downloadable from UCI Machine Learning Repository [8], which contains 8124 cases with 23 nominal attributes including a class attribute. Each case is regarded as a transaction, and each attribute value of each case is converted into an item by joining it with the corresponding attribute name, for example, “cap-shape=x,” where cap-shape is an attribute name and x is an attribute value. In 2480 cases, the attribute value of one attribute is missing. Since we ignored missing values, the transactions corresponding to them have 22 items, while the others have 23 items. Some attribute values have different meanings for different attributes. For example, “n” means “none” for the attribute “odor,” while “brown” for “cap-color.” As a result, the number of valid pairs of attribute name and attribute value, that is, number of distinct items, became 118.

First, we show the runtime of Apriori for various minsup and minconf values in Figure 4.4. All runtimes shown in this section were measured on a PC running Windows XP with 2.8 GHz Pentium IV and 4 GB memory. In these experiments, the maximal number of items per rule is set to 5 for convenience. We also limited the minimal number of items per rule to 2 in order to prevent a rule with no premise from being derived. In addition, a prefix tree was not used to store transactions. From the results, it is obvious that the change of minconf does not affect the runtime so much, but the runtime exponentially increases as minsup becomes smaller. The similar tendency is observed in Figure 4.5, showing the relation between minsup and the number of derived association rules. This is because the number of frequent itemsets exponentially increases as minsup becomes smaller, as shown in Figure 4.6.
These results show that \textit{minsup}, or the antimonotonicity property of itemsets, is very effective to prune nonfrequent itemsets.

Next, we show the relation between the runtime and the number of transactions in Figure 4.7. In this evaluation, we copied the original dataset multiple times (up to
4.4 Two Illustrative Examples

4 times). Note that the fraction of each item remains the same for all datasets, so is the number of resulting association rules (frequent itemsets). Figure 4.7 shows that the runtime linearly increases as the number of transactions becomes larger. Consequently, under a certain distribution of items, \( \text{minsup} \) is much more influential to the runtime than both \( \text{minconf} \) and the number of transactions in Apriori.

Finally, we briefly mention association rules mined through the experiments, especially, for convenience, those which have only one item representing the class attribute in the consequent. The class value is either “edible” (e) or “poisonous” (p). A typical rule found under \( \text{minsup} = 0.3 \) and \( \text{minconf} = 0.9 \) is “\( \text{odor} = \text{n} \) \( \text{gill-size} = \text{b} \) \( \text{ring-number} = \text{o} \) \( \Rightarrow \text{class} = \text{e} \),” which is the simplest one among those whose consequent is “\( \text{class} = \text{e} \),” confidence is 1.0, and support is maximum (0.331). This rule means a mushroom is edible if its order is none, the size of its gill is broad, and the number of its rings is one. The attributes “odor” and “gill-size” appear as the first and the third test nodes, respectively, in the decision tree learned from this dataset by J48, a decision tree learner available in Weka, under its default setting. A similar rule “\( \text{odor} = \text{n} \) \( \text{spore-print-color} = \text{w} \) \( \text{gill-size} = \text{b} \) \( \Rightarrow \text{class} = \text{e} \)” can be derived from the decision tree and its confidence is 1.0, too, but it is true for only 528 cases, while the association rule is true for 2689 cases. On the other hand, no rule whose confidence is 1.0 and consequent is “\( \text{class} = \text{p} \)” was found under this setting because \( \text{minsup} \) was too high. When setting \( \text{minsup} = 0.2 \), 470 such rules were found.

In general we can obtain a small number of association rules in a short runtime for a high \( \text{minsup} \), but many of them could be trivial. To find more interesting rules, we have to use a smaller \( \text{minsup} \), but it leads to an unacceptable runtime and a huge number of association rules, which in turn would make it harder to find interesting association

![Figure 4.7 Runtime for various sizes of the dataset (\( \text{minsup} = 5 \)).](image)
rules. More efficient algorithms and better measures are required to find frequent itemsets and interesting association rules, which are the topics of the next section.

4.5 Advanced Topics

Since the first proposal of frequent pattern and association rule mining algorithm by Agrawal and Srikant, there have been many publications on various kinds of improvements, extensions, and applications, ranging from efficient scalable data mining methodologies, to handling a wide diversity of data types, various extended mining tasks, and a variety of new applications. Some of the important advanced topics are briefly described in this section. There are good tutorials and surveys for frequent pattern mining by Han et al. [16] and Goethals [15] that contain a substantial amount of references.

4.5.1 Improvement in Apriori-Type Frequent Pattern Mining

There have been many attempts to devise more efficient algorithms of frequent itemset mining in the framework of Apriori algorithm in that they generate candidates. These include hash-based technique, partitioning, sampling, and using vertical data format.

- **Hash-based technique** can reduce the size of candidate itemsets. Each itemset is hashed into a corresponding bucket by using an appropriate hash function. Since a bucket can contain different itemsets, if its count is less than a minimum support, these itemsets in the bucket can be removed from the candidate sets. DHP [26] uses this idea.

- **Partitioning** can be used to divide the entire mining problem into $n$ smaller ones [29]. The dataset is divided into $n$ nonoverlapping partitions such that each partition fits into main memory and each partition is mined separately. Since any itemset that is potentially frequent must occur as a frequent itemset in at least one of the partitions, all the frequent itemsets found this way are candidates, which can be checked by accessing the entire dataset only once.

- **Sampling** is simply to mine a random sampled small subset of the entire data. Since there is no guarantee that we can find all the frequent itemsets, normal practice is to use a lower support threshold. Trade-off has to be made between accuracy and efficiency.

- **Vertical data format** associates TID with each itemset, whereas Apriori uses a horizontal data format, that is, frequent itemsets are associated with each transaction. With the vertical data format, mining can be performed by taking the intersection of TIDs. The support count is simply the length of the TID set for the itemset. There is no need to scan the database because TID set carries the complete information required for computing support. This technique requires,
Algorithm 4.5 FP-Growth Algorithm: $F[I](FP$-tree)

1. $F[I] = \emptyset$;
2. foreach $i \in I$ that is in $D$ in frequency increasing order do begin
   $F[I] = F[I] \cup \{ I \cup \{i\} \}$;
   $D' = \emptyset$;
   $H = \emptyset$;
   foreach $j \in I$ in $D$ such that $j < i$ do begin
     // $(j$ is more frequent than $i$)
     Select $j$ for which support $(I \cup \{i, j\}) \geq \text{minsup}$;
     $H = H \cup \{j\}$;
   end
   foreach $(Tid, X) \in D$ with $i \in X$ do
     $D' = D' \cup \{(Tid, \{X \setminus \{i\} \cap H\})\}$;
   Construct conditional FP-tree from $D'$;
   Call $F[I \cup \{i\}]$(conditional FP-tree);
   $F[I] = F[I] \cup F[I \cup \{i\}]$(conditional FP-tree);
end

given a set of candidate itemsets, that their TIDs are available in main memory, which is of course not always the case. However, it is possible to significantly reduce the total size by using a depth-first search. Eclat [43] uses this strategy. In the depth-first approach, it is necessary to store at most the TID list of all $k$-itemsets with the same first $k - 1$ items ($k - 1$ prefix) at depth $d$ with $k \leq d$ in the main memory.

4.5.2 Frequent Pattern Mining Without Candidate Generation

The most outstanding improvement over Apriori would be a method called FP-growth (frequent pattern growth) that succeeded in eliminating candidate generation [17, 18]. It adopts a divide and conquer strategy by (1) compressing the database representing frequent items into a structure called FP-tree (frequent pattern tree) that retains all the essential information and (2) dividing the compressed database into a set of conditional databases, each associated with one frequent itemset and mining each one separately. It scans the database only twice. In the first scan, all the frequent items and their support counts (frequencies) are derived and they are sorted in the order of descending support count in each transaction. In the second scan, items in each transaction are merged into an FP-tree and items (nodes) that appear in common in different transactions are counted. Each node is associated with an item and its count. Nodes with the same label are linked by a pointer called a node-link. Since items are sorted in the descending order of frequency, nodes closer to the root of the FP-tree are shared by more transactions, thus resulting in a very compact representation that stores all the necessary information. Pattern growth algorithm works on FP-tree
by choosing an item in the order of increasing frequency and extracting frequent itemsets that contain the chosen item by recursively calling itself on the conditional FP-tree, that is, FP-tree conditioned to the chosen item. FP-growth is an order of magnitude faster than the original Apriori algorithm. The algorithm of FP-growth is given in Algorithm 4.5. \( F[\emptyset](\text{FP-tree}) \) returns all the frequent itemsets. As noted easily, the divide and conquer strategy mentioned by Han et al. is equivalent to the depth-first search without candidate generation. The \( D_i \) is called \( i \)-projected database and generally much smaller than the FP-tree of the whole database. It is, thus, expected that \( D_i \) fits in the main memory even if the latter does not. The idea of pattern growth can also be applicable to closed itemset mining \cite{27} (see Section 4.5.4) and sequential pattern mining \cite{28} (see Section 4.5.8).

### 4.5.3 Incremental Approach

When the database is not stationary and a new batch of transactions keeps being added, it happens that some items that were frequent become no more frequent (losers) and some other items that were infrequent become frequent (winners). Rerunning Apriori or any other frequent pattern mining algorithm each time the database is updated is not efficient. The FUP algorithm in \cite{12} provides a way to incrementally update the frequent itemsets using Apriori framework. It works efficiently on the updated database since the size of the increment database \( \Delta D \) is generally much smaller than the initial database \( D \).

Let \( F_k, F'_k \) be the frequent \( k \)-itemsets in \( D \) and \( D \cup \Delta D \), respectively, and \( C_k \) be the candidate frequent itemsets in \( D \cup \Delta D \). At \( k \)-th iteration, \( C_k \) can be generated from \( F'_{k-1} \) using apriori-gen function. Any itemset in \( F_k \) that contains any one of the losers of size \( k-1 \) (those which are in \( F_{k-1} \) but not in \( F'_{k-1} \)) as its subset are filtered out from \( F_k \) without checking \( \Delta D \). Frequency of the remaining itemsets in \( F_k \) are counted over \( \Delta D \) and those frequent in \( D \cup \Delta D \) are identified (A), and excluded from \( C_k \) because we know that they are frequent. The remaining itemsets are those not in \( F_k \). Their frequency is counted over \( \Delta D \) and those not frequent in \( \Delta D \) are removed from \( C_k \) because we know that they are infrequent in \( D \). Frequency of the remaining elements in \( C_k \) are counted over \( D \cup \Delta D \) and the frequent ones are retained (B). \( F'_k \) is \( A \cup B \). As can be seen above, FUP has to scan the updated database for each \( k \), but the size of the \( C_k \) is expected to be very small. The experiment shows that it is only about 2 to 5% of that of rerunning Apriori for the updated database, and FUP runs 2 to 16 times faster than Apriori.

### 4.5.4 Condensed Representation: Closed Patterns and Maximal Patterns

An itemset (pattern) \( X \) is a maximal itemset if (1) there exists no itemset \( X' \) such that \( X' \) is a proper superset of \( X \). An itemset (pattern) \( X \) is a closed itemset if (1) there exists no itemset \( X' \) such that \( X' \) is a proper superset of \( X \) and (2) every transaction containing \( X \) also contains \( X' \). They are frequent if their support is no less than the \( \text{minsup} \). A closed itemset satisfies \( I(T(X)) = X \), where \( T(X) = \{ t \in D \mid X \subseteq t \} \) and \( I(S) = \cap_{t \in S}^c \) for \( S \subseteq D \). For any two itemsets \( X \) and \( Y \), if \( X \subset Y \) and their support
4.5 Advanced Topics

is the same, X is not a closed itemset. A closed itemset is a lossless representation, whereas a maximal itemset is not. Thus, once the closed itemsets are found, all the frequent itemsets can be derived from them. A rule $X \Rightarrow Y$ is an association rule on frequent closed itemsets if (1) both $X$ and $X \cup Y$ are frequent closed itemsets, (2) there does not exist a frequent closed itemset $Z$ such that $X \subset Z \subset (X \cup Y)$, and (3) the confidence of the rule is no less than $\text{minconf}$. The complete set of association rules can be generated once frequent closed itemsets are found.

CLOSET partitions the database and decomposes the problem into a set of subproblems, each with the corresponding conditional database, and it is known efficient [27]. First, all the frequent items are derived and sorted in the order of descending support count as $f_{\text{list}} = \langle i_1, i_2, \ldots, i_n \rangle$. The $j$-th subproblem $(1 \leq j \leq n)$ is to find the complete set of frequent closed itemsets containing $i_{n+1-j}$ but no $i_k$ (for $n+1-j < k \leq n$).

LCM is another algorithm, known to be the most efficient, to find the closed patterns (itemsets) [34]. It derives frequent closed itemsets via a closure operation without generating nonclosed itemsets. A closure of an itemset $X$, denoted by $\text{Clo}(X)$, is the unique smallest closed itemset including $X$, that is, $I(T(X))$. Without loss of generality, we assume all items in a transaction database are uniquely indexed by contiguous natural numbers. Then, $X(i) = X \cap \{1, \ldots, i\}$ is called the $i$-prefix of $X$, which is the subset of $X$ having only elements no greater than $i$. The core index of a closed itemset $X$, denoted by $\text{core}_j(X)$, is the minimum index $i$ such that $T(X(i)) = T(X)$. LCM generates, from a frequent closed itemset $X$, another frequent closed itemset $Y$ such that $Y = \text{Clo}(X \cup \{i\})$ and $X(i-1) = Y(i-1)$, where $i$ is an item that satisfies $i \notin X$ and $i > \text{core}_j(X)$. $Y$ is called the prefix-preserving closure extension, or ppc-extension for short, of $X$. LCM recursively applies this closure operation to closed itemsets from an empty itemset to larger ones in a depth-first manner. Completeness and nonredundancy of the enumeration of closed itemsets by LCM are guaranteed by the following property: If $Y$ is a nonempty closed itemset, then there is just one closed itemset $X$ such that $Y$ is a ppc-extension of $X$. Since LCM generates a new frequent closed itemset $Y$ from $T(X)$ and a subset of $I$, its time complexity to enumerate all frequent closed itemsets for $X$ is $O(||T(X)|| \times |I|)$, where $||T(X)||$ is the summation of size of each transaction included in $T(X)$. Let $C$ be a set of all frequent closed itemsets in $D$. Then, the time complexity of LCM is linear in $|C|$ with a factor depending on $||T|| \times |I|$. In fact, to improve the computation time and memory use, LCM incorporates three techniques: occurrence deliver, anytime database reduction, and fast prefix-preserving test. Occurrence deliver constructs
\( T(X \cup \{i\}) \) for all \( i \) by scanning \( T(X) \) only once instead of scanning it for each \( i \).

Anytime database reduction reduces the size of the database by removing unnecessary transactions and items from it each time before an iteration starts with the current closed itemset to reduce both the computation time and memory use. Fast prefix-preserving test significantly reduces the number of items to be accessed to test the equality \( X(i - 1) = Y(i - 1) \) by checking only items \( j \) such that \( j < i, j \not\in X(i - 1) \) and they are included in the transaction of the minimum size in \( T(X \cup \{i\}) \) instead of actually generating a closure when performing a ppc-extension. If an item \( j \) is included in every transaction in \( T(X \cup \{i\}) \), then \( j \) is included in \( Clo(X \cup \{i\}) \), thus \( X(i - 1) \neq Y(i - 1) \).

### 4.5.5 Quantitative Association Rules

When the item has a continuous numeric value, current frequent itemset mining algorithms are not applicable unless the values are discretized and appropriate intervals defined. This is known as quantitative frequent itemset (QFI) mining. The items can be both categorical and numeric. An example is \( \{ \text{Age: [30, 39]}, \text{House-owner: Yes}, \text{Married: Yes} \} \), where an item is represented as \( \langle \text{attribute: its value (range)} \rangle \). QFI mining was initially proposed in the study of mining quantitative association rules [31], but later density-based subspace clustering has commonly been applied because a QFI is viewed as an axis-parallel hyper-rectangular containing a cluster of transactions in a numeric attribute space. SUBCLUE [20] and QFIMiner [36] are two such examples. QFIMiner finds in \( O(N \log N) \) all dense clusters of no less than \( \text{minsup} \) in all subspaces formed by both numeric and categorical attributes, where \( N \) is the number of transactions. An optimal value interval for each numeric item in each frequent itemset is obtained by Apriori-like level-wise algorithm with the antimonotonicity property of dense clusters. QFIMiner is shown to be faster than SUBCLUE and scales very well.

### 4.5.6 Using Other Measure of Importance/Interestingness

The problem of support-confidence framework is that there is no valid means to determine appropriate values for \( \text{minsup} \) and \( \text{minconf} \). Especially setting \( \text{minsup} \) too high will miss important rules and setting it too low will generate too many rules. In fact, it is possible that a rule with infrequent itemsets is of great interest for some applications. Further, this framework fails to capture the notion of correlation. It can happen that a rule \( X \Rightarrow Y \) which satisfies both \( \text{minsup} \) and \( \text{minconf} \) constraints has no correlation between \( X \) and \( Y \), that is, \( \text{support}(X) \times \text{support}(Y) = \text{support}(X \cup Y) \).

Therefore, an alternative approach is to use other measures that account for importance or interestingness of a rule and select rules that have high score for these measures. Support and confidence can still be used as a constraint (setting \( \text{minsup} \) and \( \text{minconf} \) to 0 means not to use them at all). These measures include lift, leverage, redundancy, productivity, and well-known statistical measures such as chi-square, correlation coefficient, information gain, and so on.

Lift and leverage represent the ratio and the difference between the support and the support that would be expected if \( X \) and \( Y \) were independent, respectively. They try
4.5 Advanced Topics

to find rules with strong correlations between \( X \) and \( Y \).

\[
\text{lift}(X \Rightarrow Y) = \frac{\text{confidence}(X \Rightarrow Y)}{\text{confidence}(\emptyset \Rightarrow Y)} = \frac{\text{support}(X \Rightarrow Y)}{\text{support}(X) \times \text{support}(Y)}
\]

\[
\text{leverage}(X \Rightarrow Y) = \text{support}(X \Rightarrow Y) - \text{support}(X) \times \text{support}(Y) = \text{support}(X) \times (\text{confidence}(X \Rightarrow Y) - \text{support}(Y))
\]

Redundant rule constraint discards a rule \( X \Rightarrow Y \) if \( \exists Z \in X : \text{support}(X \Rightarrow Y) = \text{support}(X - [Z] \Rightarrow Y) \). A more powerful constraint is productive constraint. A rule is said to be productive if its improvement is greater than 0, where the rule’s improvement is defined as

\[
\text{improvement}(X \Rightarrow Y) = \text{confidence}(X \Rightarrow Y) - \max_{Z \subset X} (\text{confidence}(Z \Rightarrow Y)).
\]

The improvement of a redundant rule cannot be greater than 0 and hence a constraint that rules must be productive discards all redundant rules. Further, it can discard rules that include items in the antecedent that are independent of the consequent, given the remaining items in the antecedent.

Statistical measures are useful in finding discriminative patterns (itemsets). However, these measures do not satisfy the antimonotonicity property, and finding the best \( k \) patterns or rules is not that easy. If a measure is convex with respect to its arguments, it is possible to estimate its upperbound for supersets of a pattern \( X \) (itemset) for a fixed conclusion \( Y \) (normally, a class value) [23] and use this to prune the search space. Statistical measures mentioned above satisfy this property.

Webb’s KORD algorithm [39] finds \( k \)-optimal rules through the space of pairs \( X \) and \( Y \) (without fixing \( Y \)) and uses leverage as a measure to optimize using various pruning strategies.

4.5.7 Class Association Rules

When a transaction \( t \) is associated with a class \( cl \), it is natural to use association rules for classification purpose. The association rules mined for classification purpose are called class association rules (CARs). CARs have the form \{\langle p_1 : q_1 \rangle, \langle p_2 : q_2 \rangle, \ldots, \langle p_m : q_m \rangle \} \Rightarrow cl \). Here a numeric item has a numeric interval value, whereas a categorical item has a categorical value. Let \( D_{cl} \) be a set of all instances having a class \( cl \) in \( D \). CBA [22], CMAR [21], and CAEP [14] are the representative CAR-based classification systems. Especially, CAEP introduces a notion of emergent patterns and uses the strength of all CARs. Let the support of an itemset \( a \) by \( D_{cl} \) be \( \text{support}_{D_{cl}}(a) = |\{t \in D_{cl} | a \in t\}|/|D_{cl}| \). A set of QFIs, FQFI\( (cl) \), in which every itemset \( a \) satisfies \( \text{support}_{D_{cl}}(a) \geq \text{minsup} \), is derived for every \( cl \) from \( D_{cl} \). Next, for every \( a \in \text{FQFI}(cl) \), the growth rate defined by \( \text{growth rate}_{cl \rightarrow \tilde{D}_{cl}}(a) = \text{support}_{\tilde{D}_{cl}}(a)/\text{support}_{D_{cl}}(a) \) is calculated for each class \( cl \), where \( \tilde{D}_{cl} = D - D_{cl} \) represents the opponent instances of \( cl \). When the growth rate of \( a \) is not less than its threshold \( \rho (\geq 1) \), that is, \( \text{growth rate}_{cl \rightarrow \tilde{D}_{cl}}(a) \geq \rho \), \( a \) is called an emergent pattern (EP) and is selected for a rule body where its head is the class \( cl \), that is, \( a \Rightarrow cl \). Let FEP\( (cl) \) be a set
Apriori of all EPs selected from FQFI(cl) under this measure. The underlying principle here is to select the rule bodies that are strong enough to differentiate the class cl from the others. The strength of an EP a is measured by the relative difference between support\(D_{cl}(a)\) and support\(D_{cl}(a)/(support_{D_{cl}}(a) + support_{D_{cl}}(a))\): support\(D_{cl}(a)/(support_{D_{cl}}(a) + support_{D_{cl}}(a))\). This can be aggregated to define the aggregate score defined by score\(t, cl\) = \(\sum_{a \subseteq t, a \in FEP}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}$ \hat{a}
4.6 Summary

Experimenting with Apriori-like algorithm is the first thing that data miners try to do. In this chapter the basic concepts and algorithms of Apriori family (Apriori, AprioriTid, AprioriAll) were introduced first and then their working mechanisms were explained with illustrative examples, followed by a performance evaluation of Apriori using a typical freely available implementation. Since Apriori is so fundamental and easy to implement, there are many variants of it. The limitation of Apriori approach is discussed and an overview of recent important advancement in frequent pattern mining methodologies is provided. There are other topics that cannot be covered in this chapter. These include use of constraints, colossal patterns, noise handling, and top-k representatives.
4.7 Exercises

1. Prove that Apriori can derive all frequent itemsets from a given transaction database.

2. Prove the following relation:

   \[ \text{support}(X \cup Y \cup Z) \geq \text{support}(X \cup Y) + \text{support}(X \cup Z) - \text{support}(X), \]

   where \( X, Y, \) and \( Z \) are itemsets in a database.

3. Given the database shown in Table 4.5, find all frequent itemsets using Apriori and AprioriTid for \( \text{minsup} = 0.3 \) and compare their efficiency.

4. Explain the relation between a hash-tree and a trie.

5. Draw an FP-tree for the database shown in Table 4.5 and explain how frequent itemsets are derived from the FP-tree.

6. Download and install Weka on your computer, and mine association rules by using Apriori from the Soybean dataset included in the Weka’s package for various metrics to evaluate association rules using the same minimum threshold (fix the other parameters). Then, report how the resulting association rules change according to the metrics.

7. Draw a prefix tree to store the database in Section 4.4 with reference to [10] and explain how the efficiency of frequency counting can be improved in this case.

8. In an FP-tree, items in a transaction are sorted in the order of descending support count, while in a prefix tree for Apriori they are sorted in the order of ascending support count. Discuss the reason why they adopt the different orders.

9. When a transaction database has a small number of very long transactions, Apriori-based algorithms take much time to mine frequent itemsets. Explain the reason why they need so much time and propose an efficient method of mining closed itemsets from such a database.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>T01</td>
<td>Cheese, Milk, Egg</td>
</tr>
<tr>
<td>T02</td>
<td>Apple, Cheese</td>
</tr>
<tr>
<td>T03</td>
<td>Apple, Bread, Cheese, Orange, Grape</td>
</tr>
<tr>
<td>T04</td>
<td>Bread, Egg, Orange</td>
</tr>
<tr>
<td>T05</td>
<td>Cheese, Milk, Grape</td>
</tr>
<tr>
<td>T06</td>
<td>Apple, Cheese, Egg, Orange</td>
</tr>
<tr>
<td>T07</td>
<td>Bread, Cheese, Orange</td>
</tr>
<tr>
<td>T08</td>
<td>Cheese, Egg, Grape</td>
</tr>
<tr>
<td>T09</td>
<td>Bread, Cheese, Egg, Grape</td>
</tr>
<tr>
<td>T10</td>
<td>Bread, Cheese, Grape</td>
</tr>
</tbody>
</table>

© 2009 by Taylor & Francis Group, LLC
10. Given the sequence database shown in Table 4.6, find frequent sequential patterns by AprioriAll for \( \text{minsup} = 0.5 \).

### References

10. C. Borgelt. Efficient implementations of Apriori and Eclat. In *Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI’03)*,


References


© 2009 by Taylor & Francis Group, LLC


