Unit – IV

Association Analysis: Basic Concepts and Algorithms

Introduction

Many business enterprises accumulate large quantities of data from their day-to-day operations. For example, huge amounts of customer purchase data are collected daily at the checkout counters of grocery stores. The table below illustrates an example of such data, commonly known as market basket transactions. Each row in this table corresponds to a transaction, which contains a unique identifier labelled TID and a set of items bought by a given customer.

A methodology known as Association Analysis is useful for discovering interesting relationships hidden in large data sets. The uncovered relationships can be represented in the form of association rules or sets of frequent items. For example, the following rule can be extracted from the data set shown in the above table: {Diapers} -> {Beer}.

The rule suggests that a strong relationship exists between the sale of diapers and beer because many customers who buy diapers also buy beer. Retailers can use this type of rules to help them identify new opportunities for cross-selling their products to the customers.

Besides market basket data, association analysis is also applicable to other application domains such as bioinformatics, medical diagnosis, web mining, and scientific data analysis.

There are two key issues that need to be addressed when applying association analysis to market basket data. First, discovering patterns from a large transaction data set can be computationally expensive. Second, some of the discovered patterns are potentially spurious because they may happen simply by chance.

Problem Definition

Binary Representation: Market basket data can be represented in a binary format as shown in the table below, where each row corresponds to a transaction and each column corresponds to an item.

Table: A binary 0/1 representation of market basket data

TID	Bread	Milk	Diapers	Beer	Eggs	Cola
1	1	1	0	0	0	0
2	1	0	1	1	1	0
3	0	1	1	1	0	1
4	1	1	1	1	0	0
5	1	1	1	0	0	1

An item can be treated as a binary variable whose value is one if the item is present in a transaction and zero otherwise. Because of the presence of an item in a transaction is often considered more important that its absence, an item is an asymmetric binary variable. This representation is perhaps a very simplistic view of real market basket data because it ignores certain important aspects of the data such as the quantity of items sold or the price paid to purchase them.

Itemset and Support Count

Let I = {i_1, i_2, …., i_d} be the set of all items in a market basket data and T = {t_1, t_2, …., t_N} be the set of all transactions. Each transaction t_i contains a subset of items chosen from I. in association analysis, a collection of zero or more items is termed an itemset. If an itemset contains k items, it is called a k-itemset. For instance, {Beer, Diapers, Milk} is an example of a 3-itemset. The null (or empty) set is an itemset that does not contain any items.

The transaction width is defined as the number of items present in a transaction. A transaction t_j is said to contain an itemset X if X is a subset of t_j. For example, the second transaction shown in the table above contains the itemset {Bread, Diapers} but not {Bread, Milk}. An important property of an itemset is its support count, **which refers to the number of transactions that contain a particular itemset**. Mathematically, the support count, σ(X), for an itemset X can be stated as follows:

$$\sigma(\mathrm{X}) = |\{t_i | X \subseteq t_i, t_i \in T\}|$$

where the symbol |.| denote the number of elements in a set. In the data set shown in the above table, the support count for {Beer, Diapers, Milk} is equal to two because there are only two transactions that contain all three items.

Association Rule:

An association rule is an implication expression of the form X → Y, where X and Y are disjoint itemsets, i.e., X ∩ Y =∅. The strength of an association rule can be measured in terms of its support and confidence. Support determines how often a rule is applicable to a given data set, while confidence determines how frequently items in Y appear in transactions that contain X. The formal definitions of these metrics are

$$\text{Support, s (X} \rightarrow \text{Y)} = \frac{\sigma(\mathrm{X\ U\ Y})}{N}$$

$$\text{Confidence, c(X} \rightarrow \text{ Y)} = \frac{\sigma(\mathrm{X\ U\ Y})}{\sigma(\mathrm{X})}$$

Example: Consider the rule, {Milk, Diapers} → {Beer}. Since the support count for {Milk, Diapers, Beer} is 2 and the total number of transactions is 5, the rule's support is 2/5 = 0.4. The rule's confidence is obtained by dividing the support count for {Milk, Diapers, Beer} by

the support count for {Milk, Diapers}. Since there are 3 transactions that contain milk and diapers, the confidence for this rule is 2/3 = 0.67.

Why use Support and Confidence?

Support is an important measure because a rule that has very low support may occur simply by chance. A low support rule is also likely to be uninteresting from a business perspective because it may not be profitable to promote items that customers seldom buy together.

Confidence, on the other hand, measures the reliability of the inference made by a rule. For a given rule, $X \rightarrow Y$, the higher the confidence, the more likely it is for Y to be present in transactions that contain X. Confidence also provides an estimate of the conditional probability of Y given X.

Association analysis results should be interpreted with caution. The inference made by an association rule does not necessarily imply causality. Instead, it suggests a strong co-occurrence relationship between items in the antecedent and consequent of the rule. Causality, on the other hand, requires knowledge about the causal and effect attributes in the data and typically involves relationships occurring over time.

Formulation of Association Rule Mining Problem:

The association rule mining problem can be formally stated as follows:

Definition (Association Rule Discovery): Given a set of transactions T, find all the rules having support $\geq$ *minsup* and confidence $\geq$ *minconf*, where *minsup* and *minconf* are the corresponding support and confidence thresholds.

A brute-force approach for mining association rules is to compute the support and confidence for every possible rule. This approach is expensive because there are exponentially many rules that can be extracted from a dataset.

Therefore a common strategy adopted by many association rule mining algorithms is to decompose the problem into two major subtasks:

1. **Frequent Itemset Generation**, whose objective is to find all the itemsets that satisfy the minsup threshold. These itemsets are called frequent itemsets.
2. **Rule Generation**, whose objective is to extract all the high-confidence rules from the frequent itemsets found in the previous step. These rules are called strong rules.

The computational requirements for frequent itemset generation are generally more expensive than those of rule generation.

Frequent Itemset Generation

A lattice structure can be used to enumerate the list of all possible itemsets. Figure below shows an itemset lattice for I = {a, b, c, d, e}. In general, a data set that contains *k* items can potentially generate up to $2^k - 1$ frequent itemsets, excluding the null set. Because *k* can be very large in many practical applications, the search space of itemsets that need to be explored is exponentially large.

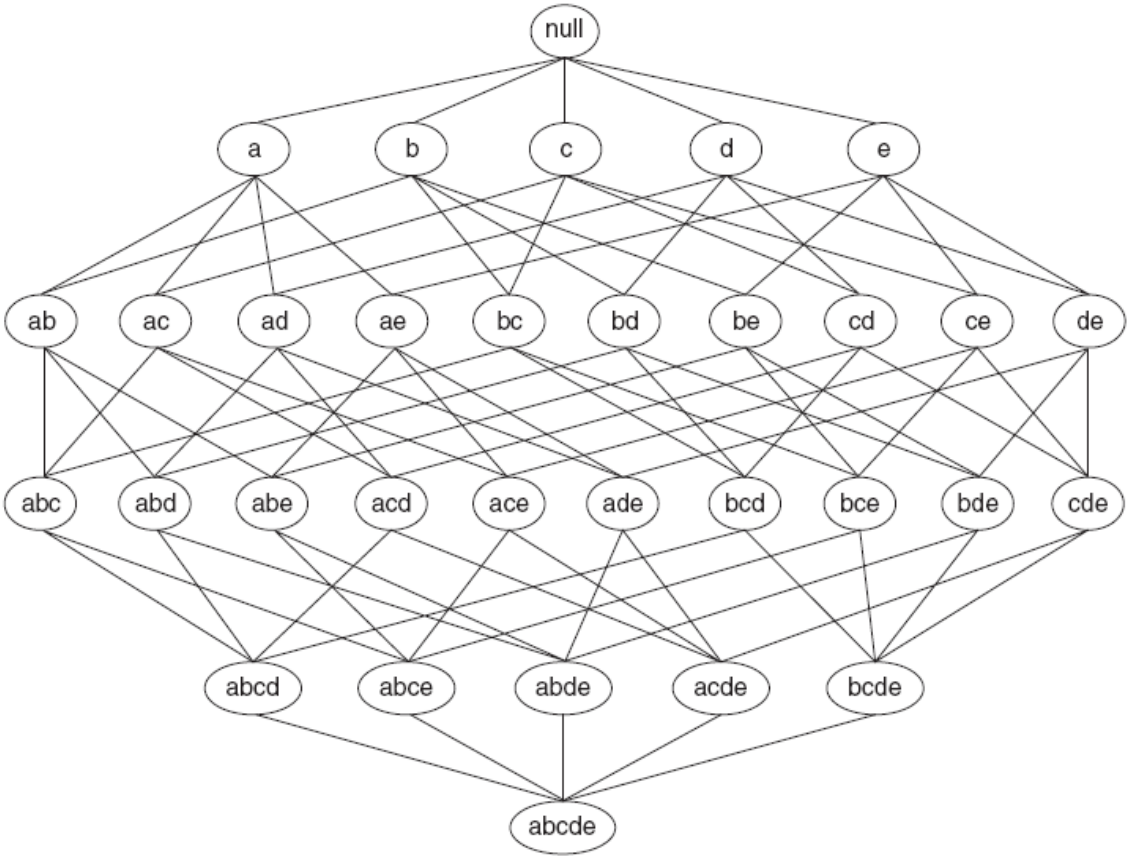


Figure 6.1. An itemset lattice.

A brute-force approach for finding frequent itemsets is to determine the support count for every candidate itemset in the lattice structure. To do this, we need to compare each candidate against every transaction, an operation that is shown in figure below. If the candidate is contained in a transaction, it support count will be incremented. For example, the support for {Bread, Milk} is incremented three times because the itemset is contained in transactions 1, 4, and 5. Such an approach can be very expensive because it requires O (N M w) comparisons, where N is the number of transactions, $M = 2^k - 1$ is the number of candidate itemsets, and w is the maximum transaction width.

There are several ways to reduce the computational complexity of frequent itemset generation.

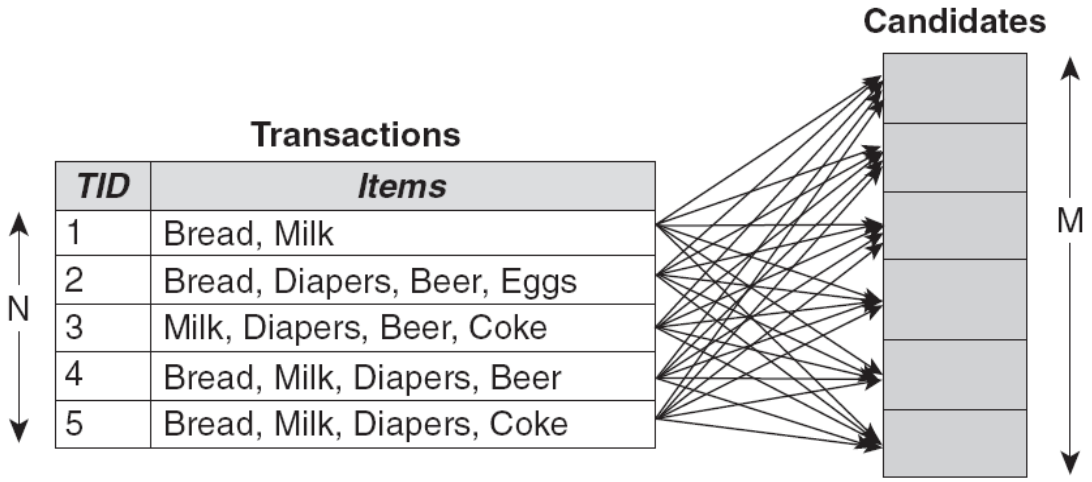


Figure 6.2. Counting the support of candidate itemsets.

1. **Reduce the number of candidate itemsets (M).** The Apriori principle, is an effective way to eliminate some of the candidate itemsets without counting their support values.
2. **Reduce the number of comparisons.** Instead of matching each candidate itemset against every transaction, we can reduce the number of comparisons by using more advanced data structures, either to store the candidate itemsets or to compress the data set.

The Apriori Principle

Theorem (Apriori Principle): If an itemset is frequent, then all of its subsets must also be frequent.

To illustrate the idea behind the Apriori Principle, consider the itemset lattice shown in figure below.

Suppose {c, d, e} is a frequent itemset. Clearly, any transaction that contains {c, d, e} must also contain its subsets, {c, d}, {c, e}, {d, e}, {c}, {d} and {e}. As a result, if {c, d, e} is frequent, then all subsets of {c, d, e} (i.e., the shaded itemsets in the below figure) must also be frequent.

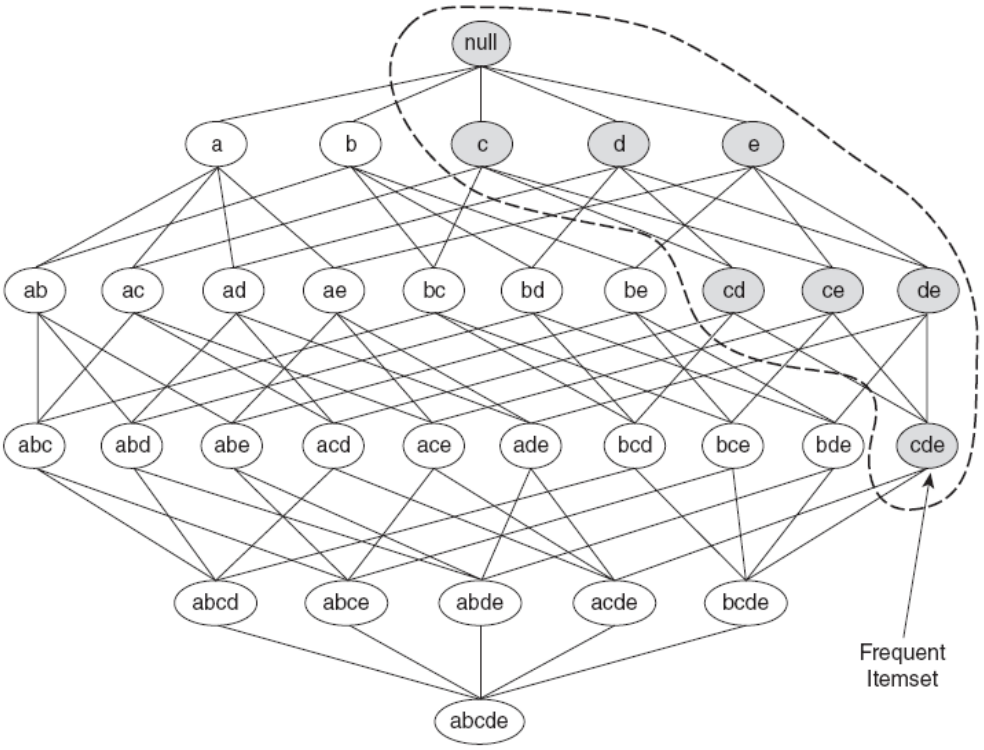


Figure 6.3. An illustration of the *Apriori* principle. If $\{c, d, e\}$ is frequent, then all subsets of this itemset are frequent.

Conversely, if an itemset such as {a, b} is infrequent, then all of its supersets must be infrequent too.

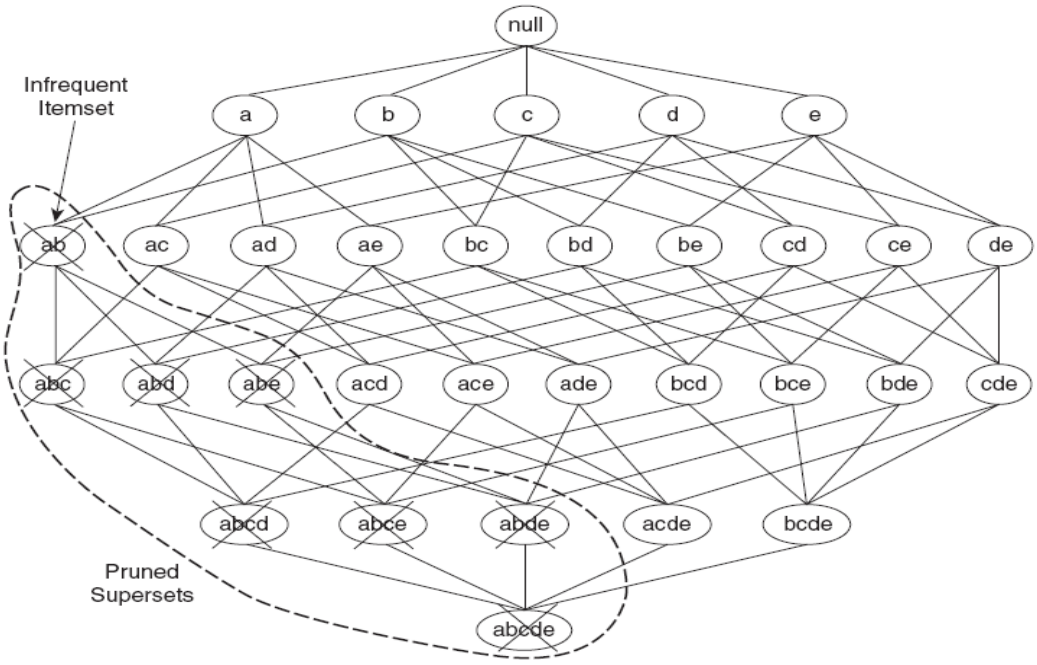


Figure 6.4. An illustration of support-based pruning. If $\{a, b\}$ is infrequent, then all supersets of $\{a, b\}$ are infrequent.

As shown in the above figure, the entire subgraph containing the supersets of {a, b} can be pruned immediately once {a, b} is found to be infrequent. This strategy of trimming the exponential search space based on the support measure is known as support-based pruning. Such a pruning strategy is made possible by a key property of the support measure, namely, that the support for an itemset never exceeds the support for its subsets. This property is also known as the anti-monotone property of the support measure.

Definition (Monotonicity Property): Let I be a set of items, and J = 2^I be the power set of I. A measure f is monotone (or upward closed) if

$$\forall X, Y \in J: (X \subseteq Y) \rightarrow f(X) \leq f(Y),$$

which means that if X is a subset of Y, then f(X) must not exceed f(Y). On the other hand, f is anti-monotone (or downward closed) if

$$\forall X, Y \in J: (X \subseteq Y) \rightarrow f(Y) \leq f(X),$$

which means that if X is a subset of Y, then f(Y) must not exceed f(X).

Any measure that possesses an anti-monotone property can be incorporated directly into the mining algorithm to effectively prune the exponential search space of candidate itemsets.

Frequent Itemset Generation in the Apriori Algorithm

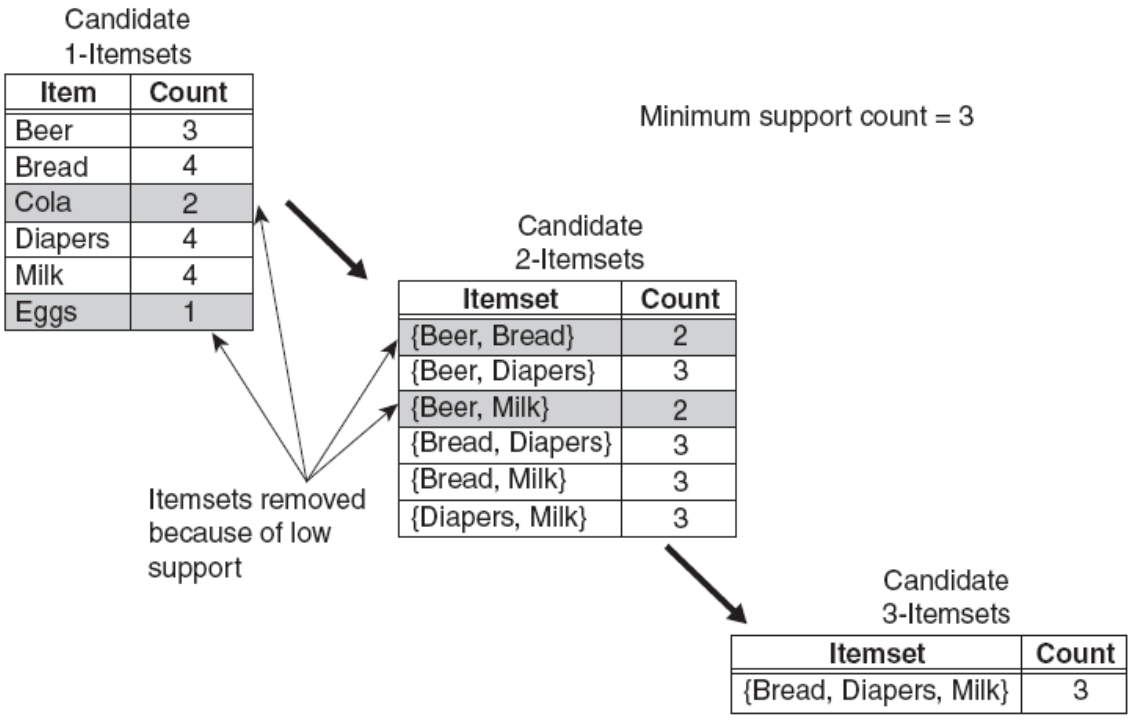


Figure 6.5. Illustration of frequent itemset generation using the *Apriori* algorithm.

Figure above provides a high-level illustration of the frequent itemset generation part of the Apriori algorithm for the transactions shown in the previous table. We assume that the support threshold is 60%, which is equivalent to a minimum support count equal to 3.

Initially, every item is considered as a candidate 1-itemset. After counting their supports, the candidate itemsets {Cola} and {Eggs} are discarded because they appear in fewer than three transactions. In the next iteration, candidate 2-itemsets are generated using only the frequent 1-itemsets because the Apriori principle ensures that all supersets of the infrequent 1-itemsets must be infrequent. Because there are only four frequent 1-itemsets, the number of candidate 2-itemsets generated by the algorithms is $\binom{4}{2} = 6$. Two of these six candidates, {Beer, Bread} and {Beer, Milk} are subsequently found to be infrequent after computing their support values. The remaining four candidates are frequent, and thus will be used to generate candidate 3-itemsets. Without support-based pruning, there are $\binom{6}{3} = 20$ candidate 3-itemsets that can be formed using the six items given in this example. With the Apriori principle, we only need to keep candidate 3-itemsets whose subsets are frequent. The only candidate that has this property is {Bread, Diapers, Milk}.

The effectiveness of the Apriori pruning strategy can be shown by counting the number of candidate itemsets generated. A brute-force strategy of enumerating all itemsets (up to size 3) as candidates will produce

$$\binom{6}{1} + \binom{6}{2} + \binom{6}{3} = 6 + 15 + 20 = 41$$

candidates. With the Apriori principle, this number decreases to

$\binom{6}{1} + \binom{4}{2} + 1 = 6 + 6 + 1 = 13$

candidates, which represents a 68% reduction in the number of candidate itemsets.

The pseudo code for the frequent itemset generation part of the Apriori algorithm is shown in the figure below. Let C_k denote the set of candidate k-itemsets and F_k denote the set of frequent k-itemsets:

- The algorithm initially makes a single pass over the data set to determine the support of each item. Upon completion of this step, the set of all frequent 1-itemsets, F_1, will be known (steps 1 and 2).

- Next, the algorithm will iteratively generate new candidate k-itemsets using the frequent (k – 1) – itemsets found in the previous iteration (step 5). Candidate generation is implemented using a function called apriori-gen.

- To count the support of the candidates, the algorithm needs to make an additional pass over the data set (steps 6 – 10). The subset function is used to determine all the candidate itemsets in C_k that are contained in each transaction t.

- After counting their supports, the algorithm eliminates all candidate itemsets whose support counts are less than minsup (step 12).

- The algorithm terminates where there are no new frequent itemsets generated, i.e., $F_k = \emptyset$ (step 13).

Algorithm: Frequent Itemset generation of the Apriori Algorithm

1. k = 1
2. $F_k = \{i \mid i \in I \wedge \sigma(\{i\}) \geq N \times minsup\}$. {Find all frequent 1-itemsets}
3. repeat
4. k = k + 1
5. C_k = apriori-gen(F_{k-1}). {Generate candidate itemsets}
6. for each transaction t ∈ T do
7. C_t = subset (C_k, t). {Identify all candidates that belong to t}
8. for each candidate itemset c ∈ C_t do
9. σ (c) = σ (c) + 1. {Increment support count}
10. end for
11. end for
12. $F_k = \{ c \mid c \in C_k \wedge \sigma(c) \geq N \times minsup\}$. {Extract the frequent k-itemsets}
13. Until $F_k = \emptyset$
14. Result = $\cup F_k$

The frequent itemset generation part of the Apriori algorithm has two important characteristics. First, it is a level-wise algorithm; i.e., it traverses the itemset lattice one level at a time, from frequent 1-itemsets to the maximum size of frequent itemsets. Second, it employs a generate-and-test strategy for finding frequent itemsets. At each iteration, new candidate itemsets are generated from the frequent itemsets found in the previous iteration. The support for each candidate is then counted and tested against the *minsup* threshold. The total number of iterations needed by the algorithm is $k_{max} + 1$, where k_{max} is the maximum size of the frequent itemsets.

Candidate Generation and Pruning

The apriori-gen function shown in step 5 of the above algorithm generates candidate itemsets by performing the following two operations:

1. **Candidate Generation:** This operation generates new candidate k-itemsets based on the frequent (k – 1) - itemsets found in the previous iteration.
2. **Candidate Pruning:** This operation eliminates some of the candidate k-itemsets using the support-based pruning strategy.

In principle, there are many ways to generate candidate itemsets. The following is a list of requirements for an effective candidate generation procedure:

1. It should avoid generating too many unnecessary candidates. A candidate itemset is unnecessary if at least one of its subsets is infrequent. Such a candidate is guaranteed to be infrequent according to the anti-monotone property of support.
2. It must ensure that the candidate set is complete, i.e., no frequent itemsets are left out by the candidate generation procedure. To ensure completeness, the set of candidate itemsets must subsume the set of all frequent itemsets, i.e., $\forall k: F_k \subseteq C_k$
3. It should not generate the same candidate itemset more than once. For example, the candidate itemset {a, b, c, d} can be generated in many ways – by merging {a, b, c} with {d}, {b, d} with {a, c}, {c} with {a, b, d}, etc. Generation of duplicate candidates leads to wasted computations and thus should be avoided for efficiency reasons.

Several candidate generation procedures, including the one used by the apriori-gen function are discussed below.

Brute-Force Method: The brute-force method considers every k-itemset as a potential candidate and then applies the candidate pruning step to remove any unnecessary candidates (shown in figure below).

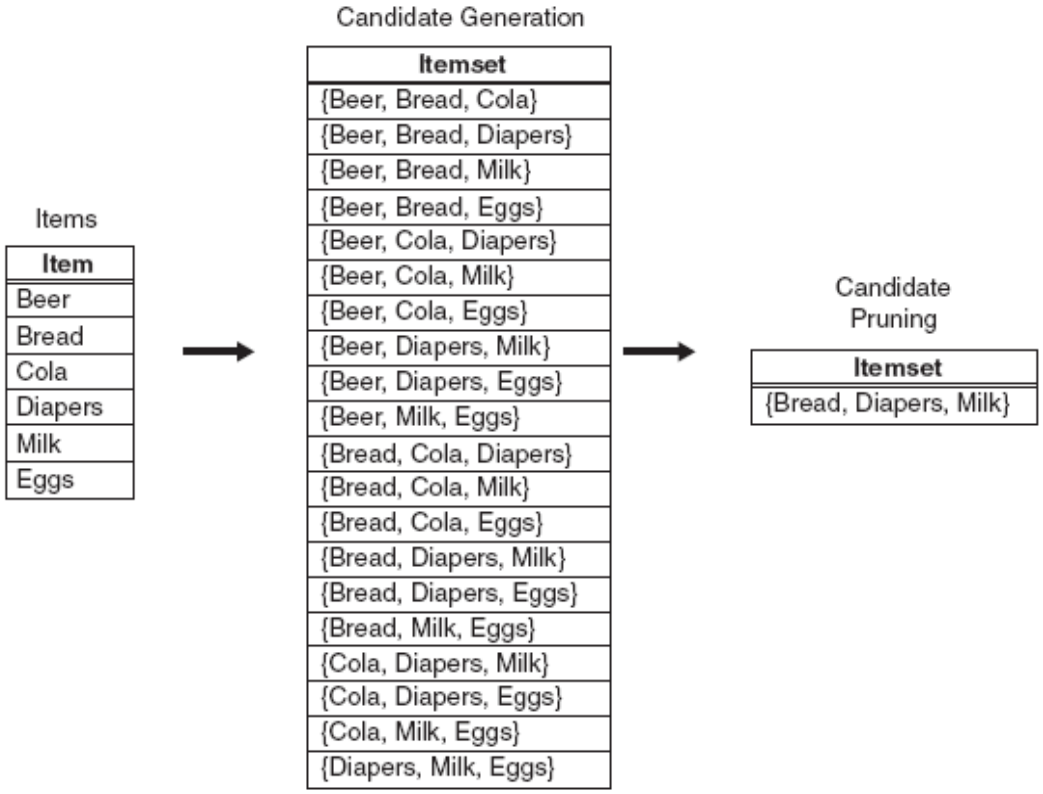


Figure 6.6. A brute-force method for generating candidate 3-itemsets.

The number of candidate itemsets generated at level k is equal to$\binom{d}{k}$, where d is the total number of items. Although candidate generation is rather trivial, candidate pruning becomes extremely expensive because a large number of itemsets must be examined. Given that the amount of computations needed for each candidate is O(k), the overall complexity of this method is $O(\sum_{k=1}^{d} k \times \binom{d}{k}) = O(d \cdot 2^{d-1})$.

F_{k-1} x F_1 Method: An alternative method for candidate generation is to extend each frequent (k – 1)-itemset with other frequent items. Figure below shows how a frequent 2-itemset such as {Beer, Diapers} can be augmented with a frequent item such as Bread to produce a candidate 3-itemset {Beer, Diapers, Bread}. This method will produce O(|Fk – 1|x |F1|) candidate k-itemsets, where |Fj| is the number of frequent j-itemsets. The overall complexity of this step is $O(\sum_k k|Fk-1||F1|)$.

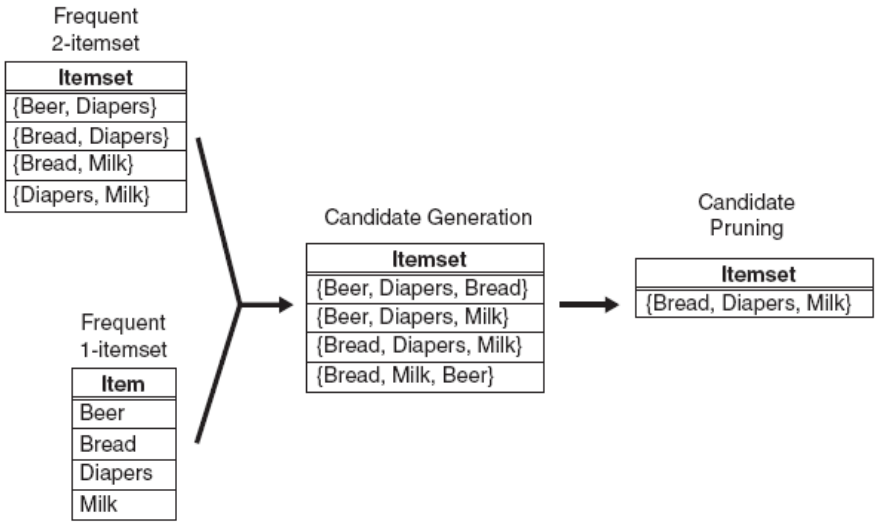


Figure 6.7. Generating and pruning candidate k-itemsets by merging a frequent $(k-1)$-itemset with a frequent item. Note that some of the candidates are unnecessary because their subsets are infrequent.

The procedure is complete because every frequent k-itemset is composed of a frequent (k – 1) itemset and a frequent 1-itemset. Therefore, all frequent k-itemsets are part of the candidate k-itemsets generated by this procedure. This approach, however, does not prevent the same candidate itemset from being generated more than once. For instance, {Bread, Diapers, Milk} can be generated by merging {Bread, Diapers} with {Milk}, {Bread, Milk} with {Diapers}, or {Diapers, Milk} with {Bread}. One way to avoid generating duplicate candidates is by ensuring that the items in each frequent itemset are kept sorted in their lexicographic order. Each frequent (k-1)-itemset X is then extended with frequent items that are lexicographically larger than the items in X. For example, the itemset {Bread, Diapers} can be augmented with {Milk} since Milk is not lexicographically larger than Bread and Diapers. However, we should not augment {Diapers, Milk} with {Bread} nor {Bread, Milk} with {Diapers} because they violate the lexicographic ordering condition.

While this procedure is a substantial improvement over the brute-force method, it can still produce a large number of unnecessary candidates. For example, the candidate itemset obtained by merging {Beer, Diapers} with {Milk} is unnecessary because one of its subsets, {Beer, Milk}, is infrequent. There are several heuristics available to reduce the number of unnecessary candidates. For example, for every candidate k-itemset that survives the pruning step, every item in the candidate must be contained in at least k-1 of the frequent (k-1)-itemsets. Otherwise, the candidate is guaranteed to be infrequent. For example, {Beer, Diapers, Milk} is a viable candidate 3-itemset only if every item in the candidate, including Beer, in contained in at least two frequent 2-itemsets. Since there is only one frequent 2-itemset containing Beer, all candidate itemsets involving Beer must be infrequent.

F_{k-1} x F_{k-1} Method: The candidate generation procedure in the apriori-gen function merges a pair of frequent (k-1)-itemsets only if their first k-2 items are identical. Let A = {a_1, a_2, …, a_{k-1}} and B = {b_1, b_2, …, b_{k-1}} be a pair of frequent (k-1)-itemsets. A and B are merged if they satisfy the following conditions:

$$a_i = b_i \text{ (for } i = 1, 2, \ldots., k - 2) \text{ and } a_{k-1} \neq b_{k-1}$$

In figure below, the frequent itemsets {Bread, Diapers} and {Bread, Milk} are merged to form a candidate 3-itemset {Bread, Diapers, Milk}.

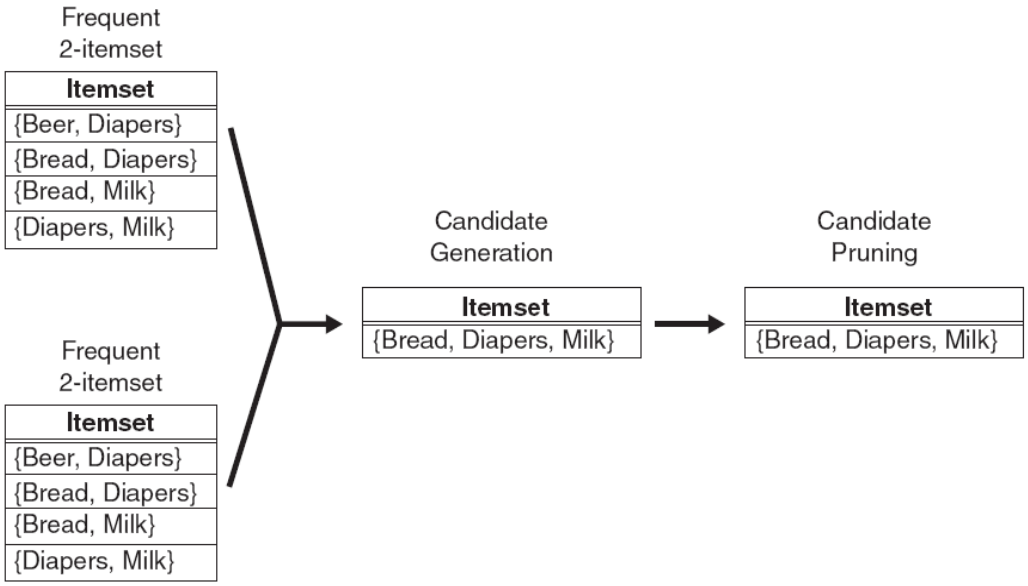


Figure 6.8. Generating and pruning candidate k-itemsets by merging pairs of frequent $(k-1)$-itemsets.

The algorithm does not have to merge {Beer, Diapers} with {Diapers, Milk} because the first item in both itemsets is different. Indeed, if {Beer, Diapers, Milk} is a viable candidate, it would have been obtained by merging {Beer, Diapers} with {Beer, Milk} instead. This example illustrates both the completeness of the candidate generation procedure and the

advantages of using lexicographic ordering to prevent duplicate candidates. However, because each candidate is obtained by merging a pair of frequent (k – 1)-itemsets, an additional candidate pruning step is needed to ensure that the remaining k – 2 subsets of the candidate are frequent.

Support Counting

Support Counting is the process of determining the frequency of occurrence for every candidate itemset that survives the candidate pruning step of the apriori-gen function. Support counting is implemented in steps 6 through 11 of the algorithm. One approach for doing this is to compare each transaction against every candidate itemset and to update the support counts of candidates contained in the transaction. This approach is computationally expensive, especially when the numbers of transactions and candidate itemsets are large.

An alternative approach is to enumerate the itemsets contained in each transaction and use them to update the support counts of their respective candidate itemsets. To illustrate, consider a transaction t that contains 5 items, {1, 2, 3, 5, 6}. There are $\binom{5}{3}$ = 10 itemsets of size 3 contained in this transaction. Some of the itemsets may correspond to the candidate 3-itemsets under investigation, in which case, their support counts are incremented. Other subsets of t that do not correspond to any candidates can be ignored.

Figure below shows a systematic way for enumerating the 3-itemsets contained in t.

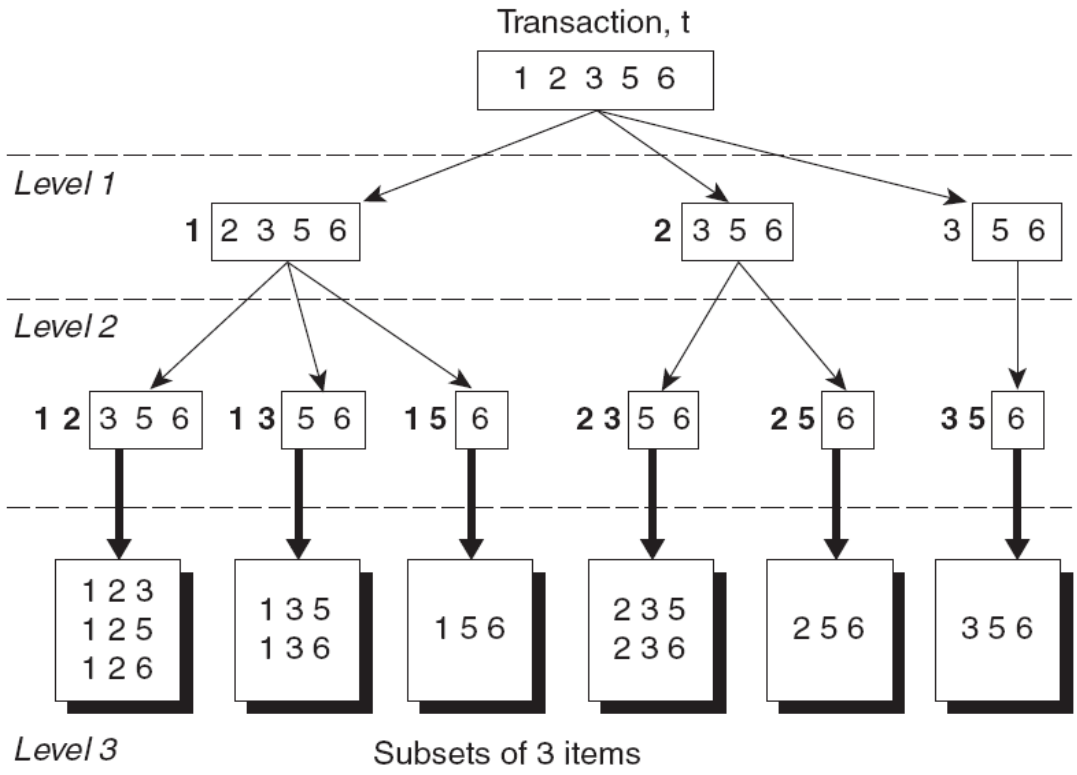


Figure 6.9. Enumerating subsets of three items from a transaction t.

Assuming that each itemset keeps its items in increasing lexicographic order, an itemset can be enumerated by specifying the smallest item first, followed by the larger items. For instance, given t = {1, 2, 3, 5, 6}, all the 3-itemsets contained in t must begin with item 1, 2, or 3. It is not possible to construct a 3-itemset that begins with items 5 or 6 because there are only two items in t whose labels are greater than or equal to 5. The number of ways to specify the first item of a 3-itemset contained in t is illustrated by the Level 1 prefix structures depicted in the figure above. For instance, 1 2 3 5 6 represents a 3-itemset that begins with item 1, followed by two more items chosen from the set {2, 3, 5, 6}.

After fixing the first item, the prefix structures at Level 2 represent the number of ways to select the second item. For example, 1 2 3 5 6 corresponds to itemsets that begin with prefix (1 2) and are followed by items 3, 5, or 6. Finally, the prefix structures at Level 3 represent the complete set of 3-itemsets contained in t. For example, the 3-itemsets that begin with prefix { 1 2} are {1, 2, 3}, {1, 2, 5}, and {1, 2, 6}, while those that begin with prefix {2 3} are {2, 3, 5} and {2, 3, 6}.

The prefix structures shown in the above figure demonstrate how itemsets contained in a transaction can be systematically enumerated, i.e., by specifying their items one by one, from the leftmost item to the rightmost item.

Rule Generation

Each frequent k-itemset, Y, can produce up to $2^k - 2$ association rules, ignoring rules that have empty antecedents or consequents ($\emptyset \rightarrow Y$ or $Y \rightarrow \emptyset$). An association rule can be extracted by partitioning the itemset Y into two non-empty subsets, X and Y – X, such that $X \rightarrow Y - X$ satisfies the confidence threshold. Note that all such rules must have already met the support threshold because they are generated from a frequent itemset.

Example: Let X = {1, 2, 3} be a frequent itemset. There are six candidate association rules that can be generated from X: {1, 2} → {3}, {1, 3} → {2}, {2, 3} → {1}, {1} →{2, 3}, {2} → {1, 3}, and {3} → {1, 2}. As each of their support is identical to the support for X, the rules must satisfy the support threshold.

Computing the confidence of an association rule does not require additional scans of the transaction data set. Consider the rule {1, 2} → {3}, which is generated from the frequent itemset X = {1, 2, 3}. The confidence for this rule is σ({1, 2, 3}) / σ({1, 2}). Because {1, 2, 3} is frequent, the anti-monotone property of support ensures that {1, 2} must be frequent too. Since the support counts for both itemsets were already found during frequent itemset generation, there is no need to read the entire data set again.

Confidence-Based Pruning

Unlike the support measure, confidence does not have any monotone property. For example, the confidence for $X \rightarrow Y$ can be larger, smaller, or equal to the confidence for another rule $X' \rightarrow Y'$, where $X' \subseteq X$ and $Y' \subseteq Y$. Nevertheless if we compare rules generated from the same frequent itemset Y, the following theorem holds for the confidence measure.

Theorem: If a rule $X \rightarrow Y - X$ does not satisfy the confidence threshold, then any rule $X' \rightarrow Y - X'$, where X' is a subset of X, must not satisfy the confidence threshold as well.

To prove this theorem, consider the following two rules: $X' \rightarrow Y - X'$ and $X \rightarrow Y - X$, where $X' \subset X$. The confidence of the rules are $\sigma(Y) / \sigma(X')$ and $\sigma(y) / \sigma(X)$, respectively. Since X' is a subset of X, $\sigma(X') \geq \sigma(X)$. Therefore, the former rule cannot have a higher confidence than the latter rule.

Rule Generation in Apriori Algorithm

The Apriori algorithm uses a level-wise approach for generating association rules, where each level corresponds to the number of items that belong to the rule consequent. Initially, all the high-confidence rules that have only one item in the rule consequent are extracted. These rules are then used to generate new candidate rules. For example, if {acd} → {b} and {abd} → {c} are high confidence rules, then the candidate rule {ad} → {bc} is generated by merging the consequents of both rules. Figure below shows a lattice structure for the association rules generated from the frequent itemset {a, b, c, d}. If any node in the lattice has low confidence, then according to the above theorem, the entire subgraph spanned by the node can be pruned immediately. Suppose the confidence for {bcd} → {a} is low. All the rules containing item a in its consequent, including {cd} → {ab}, {bd} → {ac}, {bc}→{ad} and {d} → {abc} can be discarded.

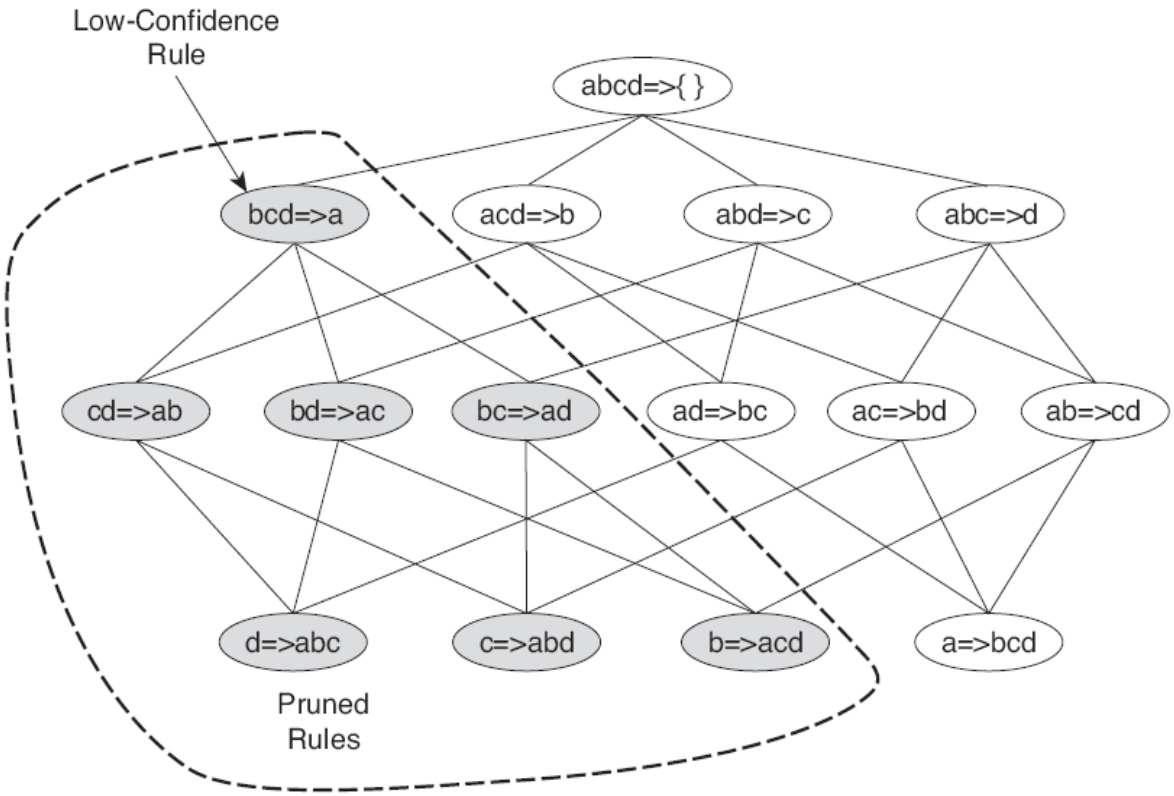


Figure 6.15. Pruning of association rules using the confidence measure.

FP-Growth Algorithm

FP-Growth algorithm takes a radically different approach to discover frequent itemsets. The algorithm encodes the data set using a compact data structure called an FP-tree and extracts frequent itemsets directly from this structure.

FP-Tree Representation

An FP-tree is a compressed representation of the input data. It is constructed by reading the data set one transaction at a time and mapping each transaction onto a path in the FP-tree. As different transactions can have several items in common, their paths may overlap. The more the paths overlap with one another, the more compression we can achieve using the FP-tree structure. If the size of the FP-tree is small enough to fit into main memory, this will allow us to extract frequent itemsets directly from the structure in memory instead of making repeated passes over the data stored on disk.

Figure below shows a data set that contains ten transactions and five items. The structures of the FP-tree after reading the first three transactions are also depicted in the diagram. Each node in the tree contains the label of an item along with a counter that shows the number of transactions mapped onto the given path. Initially, the FP-tree contains only the

root node represented by the null symbol. The FP-tree is subsequently extended in the following way:

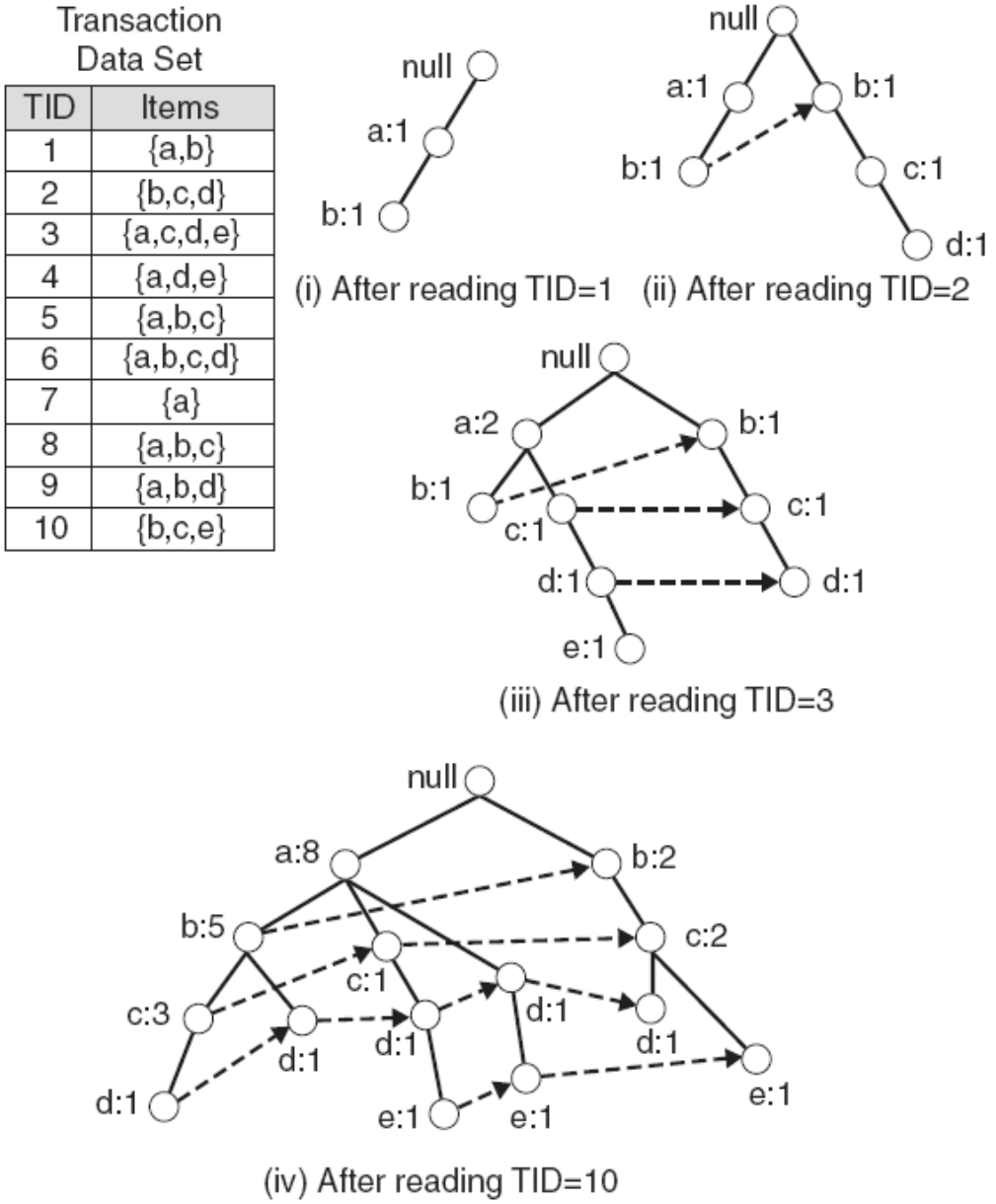


Figure 6.24. Construction of an FP-tree.

1. The data set is scanned once to determine the support count of each item. Infrequent items are discarded, while the frequent items are sorted in decreasing support counts. For the data set shown in the above figure, a is the most frequent item, followed by b, c, d, and e.
2. The algorithm makes a second pass over the data to construct the FP-tree. After reading the first transaction, {a, b}, the nodes labelled as a and b are created. A path is then formed from null → a → b to encode the transaction. Every node along the path has a frequency count of 1.

3. After reading the second transaction, {b, c, d}, a new set of nodes is created for items b, c and d. A path is then formed to represent the transaction by connecting the nodes null → b → c → d. Every node along this path also has a frequency count equal to one. Although the first two transactions have an item in common, which is b, their paths are disjoint because the transactions do not share a common prefix.
4. The third transaction, {a, c, d, e}, shares a common prefix item (which is a) with the first transaction. As a result, the path for the third transaction, null → a → c → d → e, overlaps with the path for the first transaction, null → a → b. Because of their overlapping path, the frequency count for node a is incremented to two, while the frequency counts for the newly created nodes, c, d, and e, are equal to one.
5. This process continues until every transaction has been mapped onto one of the paths given in the FP-tree. The resulting FP-tree after reading all the transactions is shown at the bottom of the above figure.

The size of an FP-tree is typically smaller than the size of the uncompressed data because many transactions in market basket data often share a few items in common. In the best-case scenario, where all the transactions have the same set of items, the FP-tree contains only a single branch of nodes. The worst-case scenario happens when every transaction has a unique set of items. As none of the transactions have any items in common, the size of the FP-tree is effectively the same as the size of the original data. However, the physical storage requirement for the FP-tree is higher because it requires additional space to store pointers between nodes and counters for each item.

The size of an FP-tree also depends on how the items are ordered. If the ordering scheme in the preceding example is reversed, i.e., from lowest to highest support item, the resulting FP-tree is shown in figure below. The tree appears to be denser because the branching factor at the root node has increased from 2 to 5 and the number of nodes containing the high support items such as a and b has increased from 3 to 12.

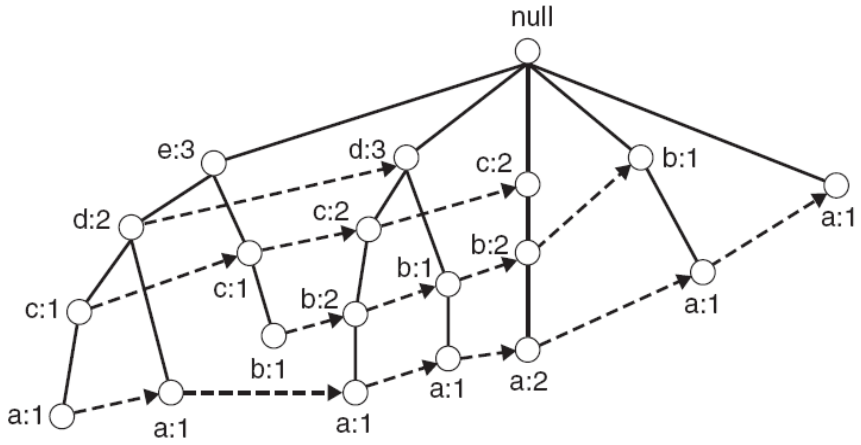


Figure 6.25. An FP-tree representation for the data set shown in Figure 6.24 with a different item ordering scheme.

Nevertheless, ordering by decreasing support counts do not always lead to the smallest tree. For example, suppose we augment the data set given in figure 6.24 with 100 transactions that contain {e}, 80 transactions that contain {d}, 60 transactions that contain {c}, and 40 transactions that contain {b}. Item e is now most frequent, followed by d, c, b, and a. With the augmented transactions, ordering by decreasing support counts will result in an FP-tree similar to the above figure, while a scheme based on increasing support counts produces a smaller FP-tree similar to figure 6.24(iv).

Frequent Itemset Generation in FP-Growth Algorithm

FP-Growth is an algorithm that generates frequent itemsets from an FP-tree by exploring the tree in a bottom-up fashion. Given the example tree shown in fig 6.24, the algorithm looks for frequent itemsets ending in e first, followed by d, c, b, and finally a. This bottom-up strategy for finding frequent itemsets ending with a particular item is equivalent to the suffix-based approach. Since every transaction is mapped onto a path in the FP-tree, we can derive the frequent itemsets ending with a particular item, say, e, by examining only the paths containing node e. These paths can be accessed rapidly using the pointers associated with node e. The extracted paths are shown in fig (a) below.

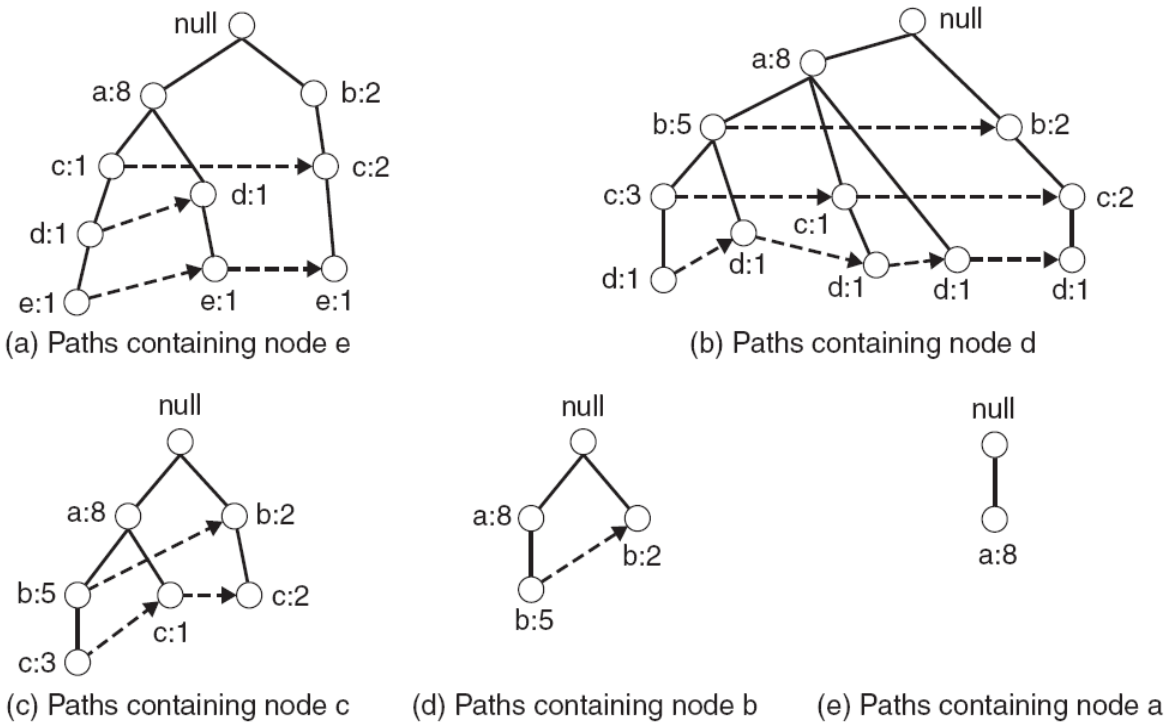


Figure 6.26. Decomposing the frequent itemset generation problem into multiple subproblems, where each subproblem involves finding frequent itemsets ending in e, d, c, b, and a.

After finding the frequent itemsets ending in e, the algorithm proceeds to look for frequent itemsets ending in d by processing the paths associated with node d. the corresponding paths are shown in fig (b) above. This process continues until all the paths associated with

nodes c, b, and finally a, are processed. The paths for these items are shown in fig (c), (d), and (e), while their corresponding frequent itemsets are summarized in table below.

Table: The list of frequent itemsets ordered by their corresponding suffixes

Suffix	Frequent Itemsets
e	{e}, {d, e}, {a, d, e}, {c, e}, {a, e}
d	{d}, {c, d}, {b, c, d}, {a, c, d}, {b, d}, {a, b, d}, {a, d}
c	{c}, {b, c}, {a, b, c}, {a, c}
b	{b}, {a, b}
a	{a}

FP-Growth finds all the frequent itemsets ending with a particular suffix by employing a divide-and-conquer strategy to split the problem into smaller subproblems. For example, suppose we are interested in finding all frequent itemsets ending in e. To do this, we must first check whether the itemset {e} itself is frequent. If it is frequent, we consider the subproblem of finding frequent itemsets ending in de, followed by ce, be, and ae. In turn, each of these subproblems are further decomposed into smaller subproblems. By merging the solutions obtained from the subproblems, all the frequent itemsets ending in e can be found. This divide-and-conquer approach is the key strategy employed by the FP-growth algorithm.

For a more concrete example on how to solve the subproblems, consider the task of finding frequent itemsets ending with e.

1. The first step is to gather all the paths containing node e. These initial paths are called prefix paths and are shown in fig(a) below.
2. From the prefix paths shown in fig(a) below, the support count for e is obtained by adding the support counts associated with node e. Assuming that the minimum support count is 2, {e} is declared a frequent itemset because its support count is 3.
3. Because {e} is frequent, the algorithm has to solve the subproblems of finding frequent itemsets ending in de, ce, be, and ae. Before solving these subproblems, it must first convert the prefix paths into a conditional FP-tree, which is structurally similar to an FP-tree, except it is used to find frequent itemsets ending with a particular suffix. A conditional FP-tree is obtained in the following way:

   (a) First, the support counts along the prefix paths must be updated because some of the counts include transactions that do not contain item e. For example, the rightmost path shown in fig (a), null → b : 2 → c : 2 → e: 1, includes a transaction {b, c} that does not contain item e. The counts along the prefix path must therefore be adjusted to 1 to reflect the actual number of transactions containing {b, c, e}.

(b) The prefix paths are truncated by removing the nodes for e. These nodes can be removed because the support counts along the prefix paths have been updated to reflect only transactions that contain e and the subproblems of finding frequent itemsets ending in de, ce, be and ae no longer need information about node e.

(c) After updating the support counts along the prefix paths, some of the items may no longer be frequent. For example, the node b appears only once and has a support count equal to 1, which means that there is only one transaction that contains both b and e. Item b can be safely ignored from subsequent analysis because all itemsets ending in *be* must be infrequent.

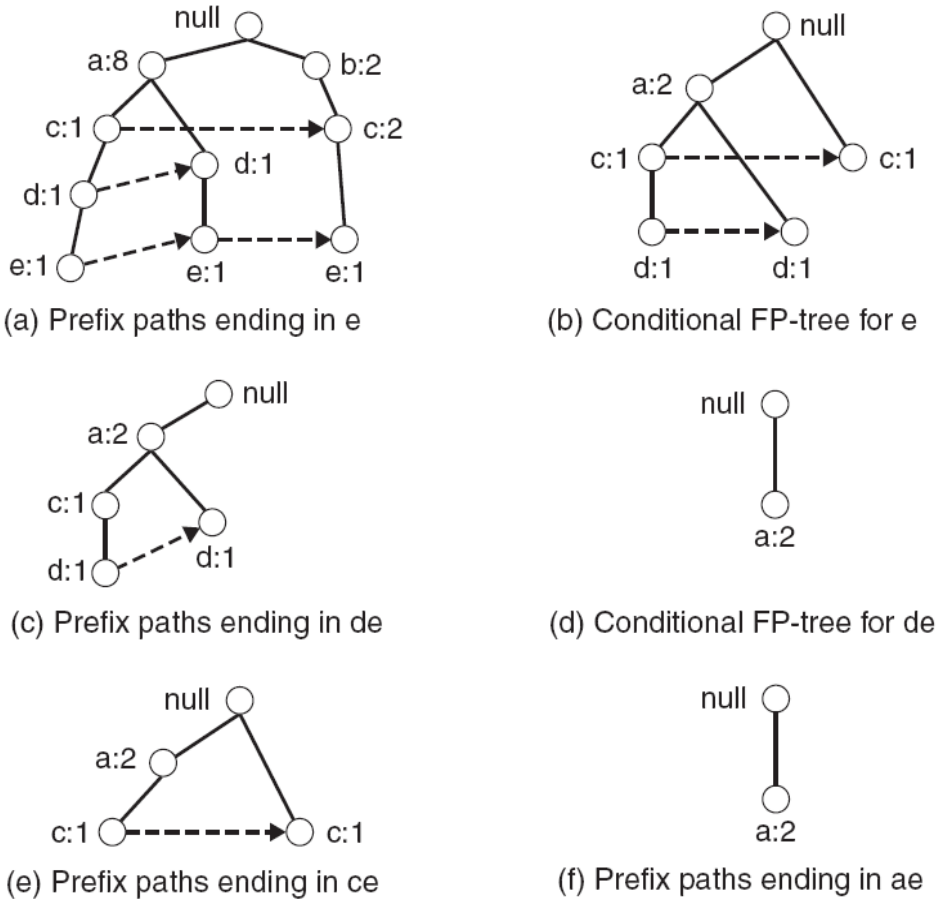


Figure 6.27. Example of applying the FP-growth algorithm to find frequent itemsets ending in e.

The conditional FP-tree for e is shown in figure (b). The tree looks different than the original prefix paths because the frequency counts have been updated and the nodes b and e have been eliminated.

4. FP-growth uses the conditional FP-tree for e to solve the subproblems of finding frequent itemsets ending in de, ce and ae. To find the frequent itemsets ending in de, the prefix paths for d are gathered from the conditional FP-tree for e (fig (c)). By adding the frequency counts associated with node d, we obtain the support count

for {d, e}. Since the support count is equal to 2, {d, e} is declared a frequent itemset. Next, the algorithm constructs the conditional FP-tree for de using the approach described in step 3. After updating the support counts and removing the infrequent item c, the conditional FP-tree for de is shown in fig (d). Since the conditional FP-tree contains only one item, a, whose support is equal to minsup, the algorithm extracts the frequent itemset {a, d, e} and moves on to the next subproblem, which is to generate frequent itemsets ending in ce. After processing the prefix paths for c, only {c, e} is found to be frequent. The algorithm proceeds to solve the next subprogram and found {a, e} to be the only frequent itemset remaining.

This algorithm illustrates the divide-and-conquer approach used in the FP-growth algorithm. At each recursive step, a conditional FP-tree is constructed by updating the frequency counts along the prefix paths and removing all infrequent items. Because the subproblems are disjoint, FP-growth will not generate any duplicate itemsets. In addition, the counts associated with the nodes allow the algorithm to perform support counting while generating the common suffix itemsets.

Clustering Techniques

Clustering or cluster analysis is a machine learning technique, which groups the unlabelled dataset. It can be defined as "**A way of grouping the data points into different clusters, consisting of similar data points. The objects with the possible similarities remain in a group that has less or no similarities with another group**." It does it by finding some similar patterns in the unlabelled dataset such as shape, size, color, behavior, etc., and divides them as per the presence and absence of those similar patterns.

It is an unsupervised learning method, hence no supervision is provided to the algorithm, and it deals with the unlabeled dataset.

After applying this clustering technique, each cluster or group is provided with a cluster-ID. ML system can use this id to simplify the processing of large and complex datasets.

Types of Cluster Analysis Methods

The cluster analysis methods may be divided into the following categories:

Partitional Methods:

Partitional methods obtain a single level partition of objects. These methods usually are based on greedy heuristics that are used iteratively to obtain a local optimum solution. Given n objects, these methods make k <= n clusters of data and use an iterative relocation method. It is assumed that each cluster has at least one object and each object belongs to only one cluster. Objects may be relocated between clusters as the clusters are refined. Often these methods require that the number of clusters be specified apriori and this number usually does not change during the processing.

Hierarchical Methods

Hierarchical methods obtain a nested partition of the objects resulting in a tree of clusters. These methods either start with one cluster and then split into smaller and smaller clusters (called divisive or top down) or start with each object in an individual cluster and then try to merge similar clusters into larger and larger clusters (called agglomerative or bottom up). In this approach, in contrast to partitioning, tentative clusters may be merged or split based on some criteria.

Density-Based Methods

In this class of methods, typically for each data point in a cluster, at least a minimum number of points must exist within a given radius. Density-based methods can deal with arbitrary shape clusters since the major requirement of such methods is that each cluster be a dense region of points surrounded by regions of low density.

Grid-based methods

In this class of methods, the object space rather than the data is divided into a grid. Grid partitioning is based on characteristics of the data and such methods can deal with non-numeric data more easily. Grid-based methods are not affected by data ordering.

Model-based methods

A model is assumed, perhaps based on a probability distribution. Essentially the algorithm tries to build clusters with a high level of similarity within them and a low level of similarity between them. Similarity measurement is based on the mean values and the algorithm tries to minimize the squared-error function.

A simple taxonomy of cluster analysis methods is presented in fig. below.

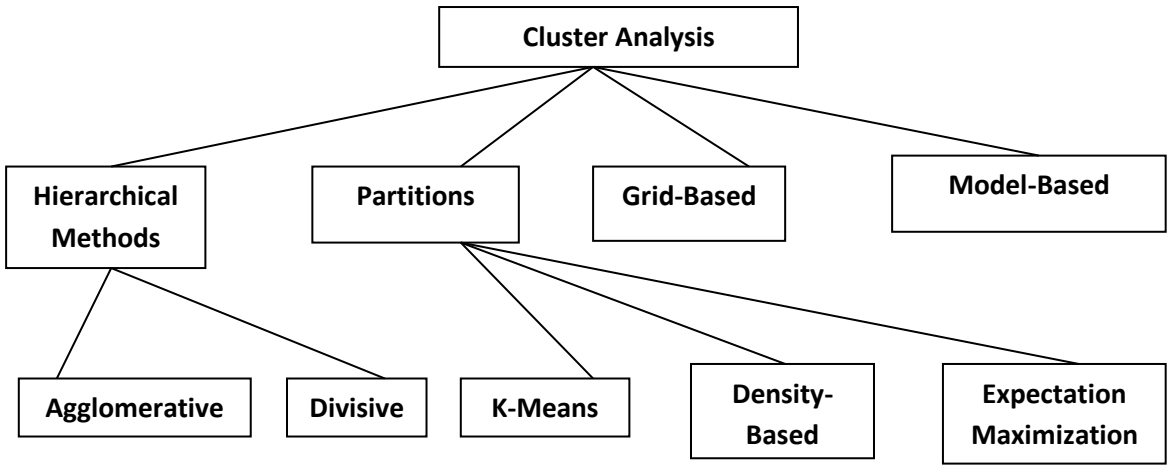


Fig: Taxonomy of cluster analysis methods

Partitional methods are popular since they tend to be computationally efficient and are more easily adapted for very large datasets. The hierarchical methods tend to be computationally more expensive. These methods not only require specifying the number of clusters apriori, they also require the user to normally specify the starting states (or seeds) of the clusters. This may be difficult for a user since the user may not have such knowledge and there is no simple reliable method for finding initial conditions for the clusters.

Partitional Methods

The aim of partitional methods is to reduce the variance within each cluster as much as possible and have large variance between the clusters. Since the partitional methods do not normally explicitly control the inter-cluster variance, heuristics may be used for ensuring large inter-cluster variance. The K-Means method of the partitional method is discussed below.

The K-Means Method

K-Means is the simplest and most popular classical clustering method that is easy to implement. The classical method can only be used if the data about all the objects is located in the main memory. The method is called K-Means since each of the K clusters is represented by the mean of the objects (called the centroid) within it. It is also called the *centroid method* since at each step the centroid point of each cluster is assumed to be known and each of the remaining points are allocated to the cluster whose centroid is closest to it. Once this allocation is completed, the centroids of the clusters are recomputed using simple means and the process of allocating points to each cluster is repeated until there is no change in the clusters (or some other stopping criterion, e.g. no significant reduction in the squared error, is met). The method may also be looked at as a search problem where the aim is essentially to find the optimum clusters given the number of clusters and seeds specified by the user.

The K-Means method uses the Euclidean distance measure, which appears to work well with compact clusters. If instead of the Euclidean distance, the Manhattan distance is used the method is called the K-median method. The K-Median method can be less sensitive to outliers.

The K-Means method may be described as follows:

1. Select the number of clusters. Let this number be k.

2. Pick k seeds as centroids of the k clusters. The seeds may be picked randomly unless the user has some insight into the data

3. Compute the Euclidean distance of each object in the dataset from each of the centroids.

4. Allocate each object to the cluster it is nearest to based on the distances computed in the previous step.

5. Compute the centroids of the clusters by computing the means of the attribute values of the objects in each cluster.

6. Check if the stopping criterion has been met (e.g. the cluster membership is unchanged). If yes, go to step 7. If not, go to step 3.

7. [Optional] One may decide to stop at this stage or to split a cluster or combine two clusters heuristically until a stopping criterion is met.

Example done in class

Starting values for the K-Means method

Specifying the number of clusters and starting seeds is difficult. This problem may be overcome by using an iterative approach. For example, one may first select three clusters and choose three starting seeds randomly. Once the final clusters have been obtained, the process may be repeated with a different set of seeds. Attempts should be made to select seeds that are as far away from each other as possible. Also, during the iterative process if two clusters are found to be close together, it may be desirable to merge them. Also, a large cluster may be split in two if the variance within the cluster is above some threshold value.

Summary of the K-Means method

K-Means is an iterative-improvement greedy method. A number of iterations are normally needed for convergence and therefore the dataset is processed a number of times. If the data is very large and cannot be accommodated in the main memory the process may become inefficient.

Although the K-Means method is most widely known and used, there are a number of issues related to the method that should be understood:

1. The K-Means method needs to compute Euclidean distances and means of the attribute values of objects within a cluster. The classical algorithm therefore is only suitable for continuous data. K-Means variations that deal with categorical data are available but not widely used.

2. The K-Means method implicitly assumes spherical probability distributions.

3. The results of the K-Means method depend strongly on the initial guesses of the seeds.

4. The K-Means method can be sensitive to outliers. If an outlier is picked as a starting seed, it may end up in a cluster of its own. Also, if an outlier moves from one cluster to

another during iterations, it can have a major impact on the clusters because the means of the two clusters are likely to change significantly.

5. Although some local optimum solutions discovered by the K-Means method are satisfactory, often the local optimum is not as good as the global optimum.

6. The K-Means method does not consider the size of the clusters. Some clusters may be large and some very small.

7. The K-Means method does not deal with overlapping clusters.

Hierarchical Methods

Hierarchical methods produce a nested series of clusters as opposed to the partitional methods which produce only a list of clusters. The hierarchical methods attempt to capture the structure of the data by constructing a tree of clusters. This approach allows clusters to be found at different levels of granularity.

Two types of hierarchical approaches are possible. In one approach, called the agglomerative approach for merging groups (or bottom-up approach), each object at the start is a cluster by itself and the nearby clusters are repeatedly merged resulting in larger and larger clusters until some stopping criterion (often a given number of clusters) is met or all the objects are merged into a single large cluster which is the highest level of the hierarchy.

In the second approach, called the divisive approach (or the top-down approach), all the objects are put in a single cluster to start. The method then repeatedly performs splitting of clusters resulting in smaller and smaller clusters until a stopping criterion is reached or each cluster has only one object in it.

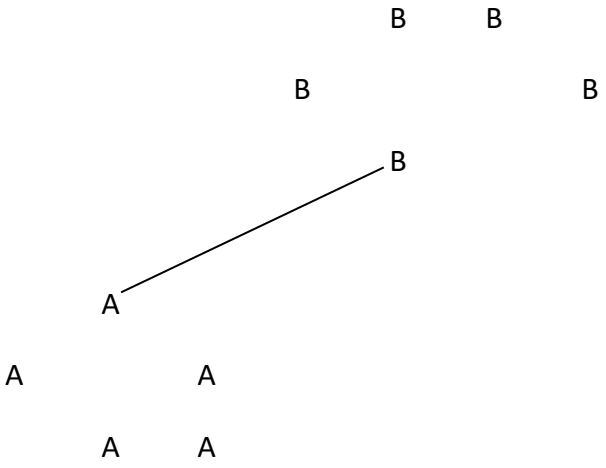
Distance between clusters

The hierarchical clustering methods require distances between clusters to be computed. These distance metrics are often called linkage metrics. The following are some of the methods for computing distances between clusters:

1. Single-link algorithm
2. Complete-link algorithm
3. Centroid algorithm
4. Average-link algorithm
5. Ward's minimum-variance algorithm

Single-link: The single-link (or the nearest neighbour) algorithm is perhaps the simplest algorithm for computing distance between two clusters. The algorithm determines the distance between two clusters as the minimum of the distances between all pairs of points (a, x) where a is from the first cluster and x is from the second. The algorithm therefore requires that all pairwise distances be computed and the smallest distance (or the shortest link) found. The algorithm can form chains and can form elongated clusters. Figure below shows two clusters A and B and the single-link distance between them.

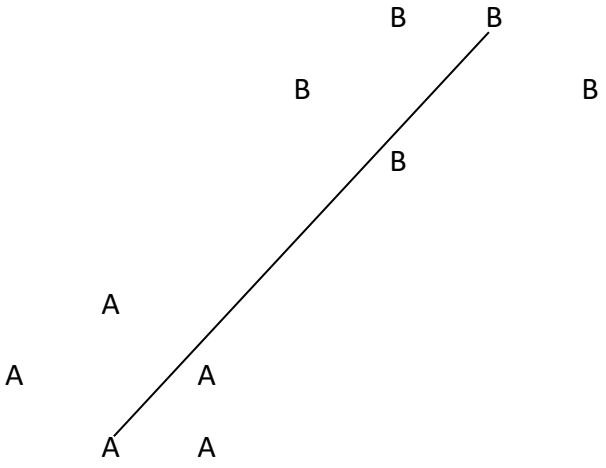
Fig: Single-link distance between two clusters



Complete-link

The complete-link algorithm is also called the farthest neighbour algorithm. In this algorithm, the distance between two clusters is defined as the maximum of the pairwise distances (a, x). Therefore if there are m elements in one cluster and n in the other, all mn pairwise distances therefore must be computed and the largest chosen.

Complete link is strongly biased towards compact clusters. Figure below shows two clusters A and B and the complete link distance between them. Complete link can be distorted by moderate outliers in one or both of the clusters.

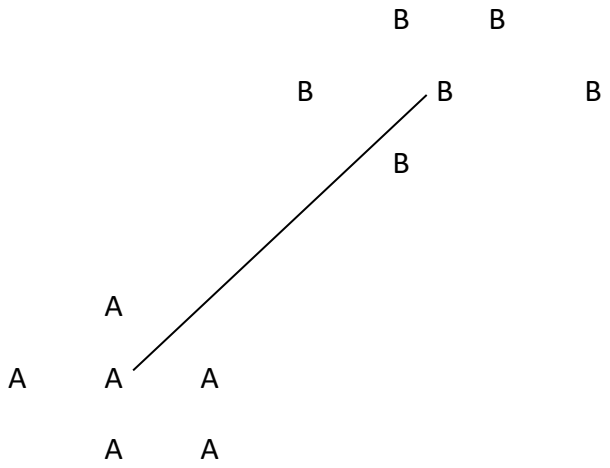


Both single-link and complete-link measures have their difficulties. In the single-link algorithm, each cluster may have an outlier and the two outliers may be nearby and so the distance between the two clusters would be computed to be small. Single-link can form a chain of objects as clusters are combined since there is no constraint on the distance between objects that are far away from each other.

On the other hand, the two outliers may be very far away although the clusters are nearby and the complete-link algorithm will compute the distance as large.

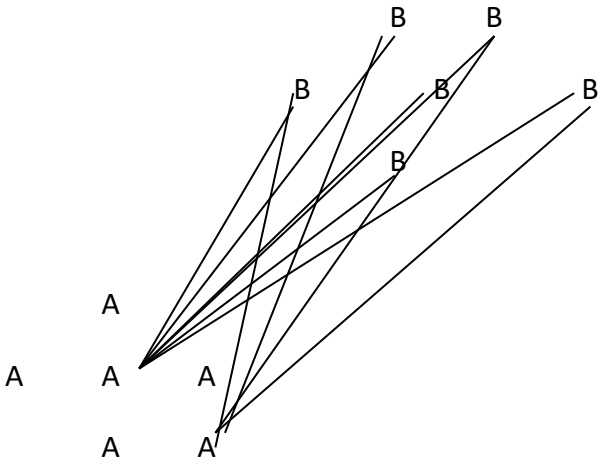
Centroid

In the centroid algorithm, the distance between two clusters is determined as the distance between the centroids of the clusters as shown below. The centroid algorithm computes the distance between two clusters as the distance between the average point of each of the two clusters. Usually the squared Euclidean distance between the centroids is used. This approach is easy and generally works well and is more tolerant of somewhat longer clusters than the complete-link algorithm. Figure below shows two clusters A and B and the centroid distance between them.



Average-link

The average-link algorithm on the other hand computes the distance between two clusters as the average of all pairwise distances between an object from one cluster and another from the other cluster. Therefore if there are m elements in one cluster and n in the other, there are mn distances to be computed, added and divided by mn. This approach also generally works well. It tends to join clusters with small variances although it is more tolerant of somewhat longer clusters than the complete-link algorithm. Figure below shows two clusters A and B and the average-link distance between them.



Ward's minimum-variance method

Ward's minimum-variance distance measure on the other hand is different. The method generally works well and results in creating tight clusters. Ward's distance is the difference between the total within the cluster sum of squares for the two clusters separately and within the cluster sum of squares resulting from merging the two clusters. An expression for Ward's distance may be derived. It may be expressed as follows:

$$D_w(A, B) = N_A N_B D_c(A, B) / (N_A + N_B)$$

Where $D_w(A, B)$ is the Ward's minimum-variance distance between clusters A and B with N_A and N_B objects in them respectively. $D_c(A, B)$ is the centroid distance between the two clusters computed as squared Euclidean distance between the centroids. It has been observed that the Ward's method tends to join clusters with a small number of objects and is biased towards producing clusters with roughly the same number of objects. The distance measure can be sensitive to outliers.

Agglomerative Method

The basic idea of the agglomerative method is to start out with n clusters for n data points, that is, each cluster consisting of a single data point. Using a measure of distance, at each step of the method, the method merges two nearest clusters, thus reducing the number of clusters and building successively larger clusters. The process continues until the required number of clusters has been obtained or all the data points are in one cluster. The agglomerative method leads to hierarchical clusters in which at each step we build larger and larger clusters that include increasingly dissimilar objects. The result is a tree-based representation of the objects, named **dendrogram**.

The agglomerative method is a bottom-up approach which involves the following steps.

1. Allocate each point to a cluster of its own. Thus we start with n clusters for n objects

2. Create a distance matrix by computing distances between all pairs of clusters either using, for example, the single-link metric or the complete-link metric. Some other metric may also be used. Sort these distances in ascending order.

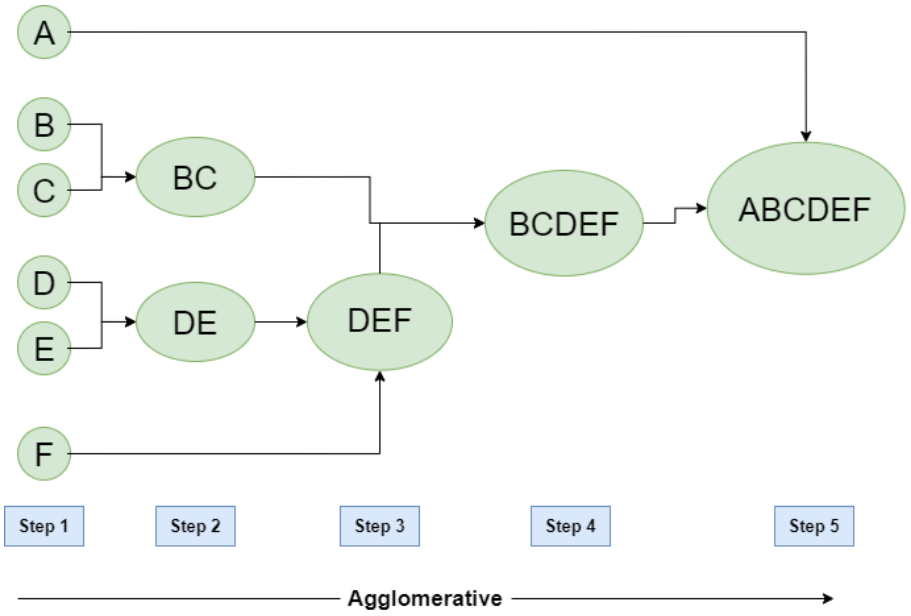
3. Find the two clusters that have the smallest distance between them.

4. Remove the pair of objects and merge them.

5. If there is only one cluster left then stop.

6. Compute all distances from the new cluster and update the distance matrix after the merger and go to step 3.

Example



- Consider each alphabet as a single cluster and calculate the distance of one cluster from all the other clusters.
- In the second step, comparable clusters are merged together to form a single cluster. Let's say cluster (B) and cluster (C) are very similar to each other therefore we merge them in the second step similarly to cluster (D) and (E) and at last, we get the clusters [(A), (BC), (DE), (F)]
- We recalculate the proximity according to the algorithm and merge the two nearest clusters([(DE), (F)]) together to form new clusters as [(A), (BC), (DEF)]
- Repeating the same process; The clusters DEF and BC are comparable and merged together to form a new cluster. We're now left with clusters [(A), (BCDEF)].
- At last, the two remaining clusters are merged together to form a single cluster [(ABCDEF)].

Divisive Hierarchical Method

The divisive method is the opposite of the agglomerative method in that the method starts with the whole dataset as one cluster and then proceeds to recursively divide the cluster into two sub-clusters and continues until each cluster has only one object or some other stopping criterion has been reached. There are two types of divisive methods:

1. Monothetic: It splits a cluster using only one attribute at a time. An attribute that has the most variation could be selected.

2. Polythetic: It splits a cluster using all of the attributes together. Two clusters far apart could be built based on distance between objects.

A typical polythetic divisive method works like the following:

1. Decide on a method of measuring the distance between two objects. Also decide a threshold distance.

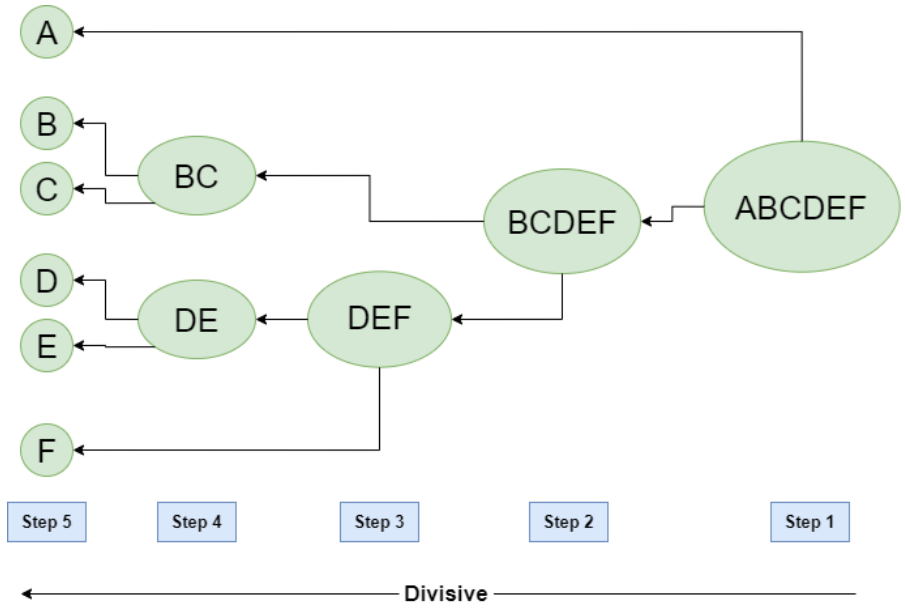
2. Create a distance matrix by computing distances between all pairs of objects within the cluster. Sort these distances in ascending order.

3. Find the two objects that have the largest distance between them. They are the most dissimilar objects.

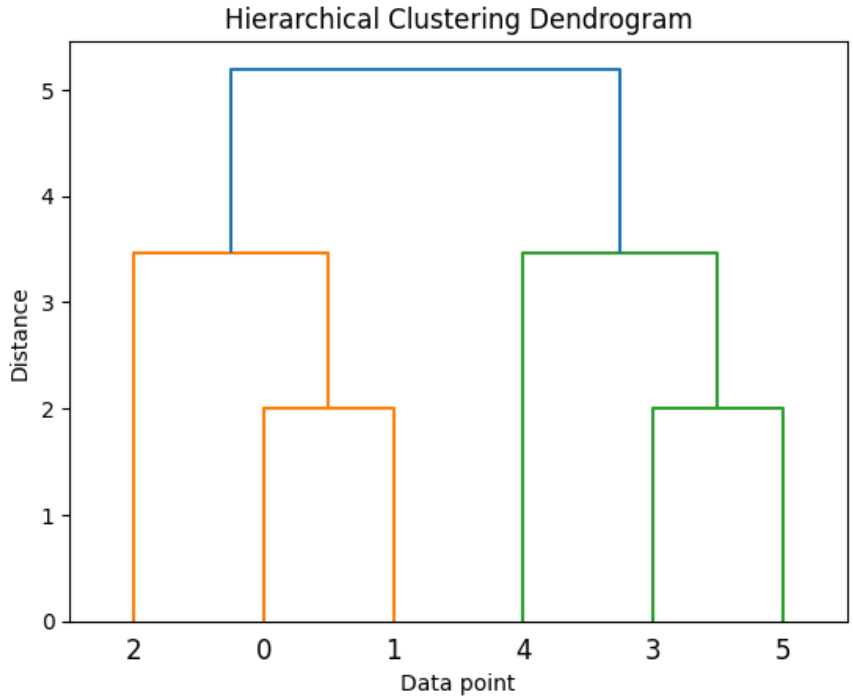
4. If the distance between the two objects is smaller than the pre-specified threshold and there is no other cluster that needs to be divided then stop, otherwise continue.

5. Use the pair of objects as seeds of a K-means method to create two new clusters.

6. If there is only one object in each cluster then stop otherwise continue with Step 2.



Dendrogram Representation



In the above method, we need to resolve the following two issues:

- Which cluster to split next?
- How to split a cluster?

Which cluster to split next?

There are a number of possibilities when selecting the next cluster to split:

1. Split the clusters in some sequential order
2. Split the cluster that has the largest number of objects
3. Split the cluster that has the largest variation within it.

The first two approaches are clearly very simple but the third approach is better since it is based on the sound criterion of splitting a cluster that has the most variance.

How to split a cluster?

A simple approach for splitting a cluster based on distances between the objects in the cluster. A distance matrix is created and the two most dissimilar objects are selected as seeds of two new clusters. The K-Means method is then used to split the cluster.

Advantages of the Hierarchical approach

1. The hierarchical approach can provide more insight into the data by showing a hierarchy of clusters than a flat cluster structure created by a partitioning method like the K-Means method.

2. Hierarchical methods are conceptually simpler and can be implemented easily.

3. In some applications only proximity data is available and then the hierarchical approach may be better.

4. Hierarchical methods can provide clusters at different levels of granularity.

Disadvantages of the hierarchical approach

1. The hierarchical methods do not include a mechanism by which objects that have been incorrectly put in a cluster may be reassigned to another cluster.

2. The time complexity of hierarchical methods can be shown to be $O(n^3)$.

3. The distance matrix requires $O(n^2)$ space and becomes very large for a large number of objects.

4. Different distance metrics and scaling of data can significantly change the results.

Hierarchical methods are quite different from partition-based clustering. In hierarchical methods there is no need to specify the number of clusters and the cluster seeds. The agglomerative approach essentially starts with each object in an individual cluster and then merges clusters to build larger and larger clusters. The divisive approach is the opposite of that. There is no metric to judge the quality of the results.