24 DATA MINING

The secret of success is to know something nobody else knows.

—Aristotle Onassis

Data mining consists of finding interesting trends or patterns in large datasets, in order to guide decisions about future activities. There is a general expectation that data mining tools should be able to identify these patterns in the data with minimal user input. The patterns identified by such tools can give a data analyst useful and unexpected insights that can be more carefully investigated subsequently, perhaps using other decision support tools. In this chapter, we discuss several widely studied data mining tasks. There are commercial tools available for each of these tasks from major vendors, and the area is rapidly growing in importance as these tools gain acceptance from the user community.

We start in Section 24.1 by giving a short introduction to data mining. In Section 24.2, we discuss the important task of counting co-occurring items. In Section 24.3, we discuss how this task arises in data mining algorithms that discover rules from the data. In Section 24.4, we discuss patterns that represent rules in the form of a tree. In Section 24.5, we introduce a different data mining pattern called clustering and consider how to find clusters. In Section 24.6, we consider how to perform similarity search over sequences. We conclude with a short overview of other data mining tasks in Section 24.7.

24.1 INTRODUCTION TO DATA MINING

Data mining is related to the subarea of statistics called *exploratory data analysis*, which has similar goals and relies on statistical measures. It is also closely related to the subareas of artificial intelligence called *knowledge discovery* and *machine learning*. The important distinguishing characteristic of data mining is that the volume of data is very large; although ideas from these related areas of study are applicable to data mining problems, *scalability with respect to data size* is an important new criterion. An algorithm is **scalable** if the running time grows (linearly) in proportion to the dataset size, given the available system resources (e.g., amount of main memory and disk). Old algorithms must be adapted or new algorithms must be developed to ensure scalability.

Finding useful trends in datasets is a rather loose definition of data mining: In a certain sense, all database queries can be thought of as doing just this. Indeed, we have a continuum of analysis and exploration tools with SQL queries at one end, OLAP queries in the middle, and data mining techniques at the other end. SQL queries are constructed using relational algebra (with some extensions); OLAP provides higher-level querying idioms based on the multidimensional data model; and data mining provides the most abstract analysis operations. We can think of different data mining tasks as complex 'queries' specified at a high level, with a few parameters that are user-definable, and for which specialized algorithms are implemented.

In the real world, data mining is much more than simply applying one of these algorithms. Data is often noisy or incomplete, and unless this is understood and corrected for, it is likely that many interesting patterns will be missed and the reliability of detected patterns will be low. Further, the analyst must decide what kinds of mining algorithms are called for, apply them to a well-chosen subset of data samples and variables (i.e., tuples and attributes), digest the results, apply other decision support and mining tools, and iterate the process.

The **knowledge discovery process**, or short **KDD process**, can roughly be separated into four steps. The raw data first undergoes a **data selection** step, in which we identify the target dataset and relevant attributes. Then in a **data cleaning** step, we remove noise and outliers, transform field values to common units, generate new fields through combination of existing fields, and bring the data into the relational schema that is used as input to the data mining activity. The data cleaning step might also involve a denormalization of the underlying relations. In the **data mining** step, we extract the actual patterns. In the final step, the **evaluation** step, we present the patterns in an understandable form to the end user, for example through visualization. The results of any step in the KDD process might lead us back to an earlier step in order to redo the process with the new knowledge gained. In this chapter, however, we will limit ourselves to looking at algorithms for some specific data mining tasks. We will not discuss other aspects of the KDD process further.

24.2 COUNTING CO-OCCURRENCES

We begin by considering the problem of counting co-occurring items, which is motivated by problems such as market basket analysis. A **market basket** is a collection of items purchased by a customer in a single **customer transaction**. A customer transaction consists of a single visit to a store, a single order through a mail-order catalog, or an order at a virtual store on the web. (In this chapter, we will often abbreviate *customer transaction* by *transaction* when there is no confusion with the usual meaning of *transaction* in a DBMS context, which is an execution of a user program.) A common goal for retailers is to identify items that are purchased together. This information can be used to improve the layout of goods in a store or the layout of catalog pages.

transid	*custid*	*date*	*item*	*qty*
111	201	5/1/99	pen	2
111	201	5/1/99	ink	1
111	201	5/1/99	milk	3
111	201	5/1/99	juice	6
112	105	6/3/99	pen	1
112	105	6/3/99	ink	1
112	105	6/3/99	milk	1
113	106	5/10/99	pen	1
113	106	5/10/99	milk	1
114	201	6/1/99	pen	2
114	201	6/1/99	ink	2
114	201	6/1/99	juice	4

Figure 24.1 The Purchases Relation for Market Basket Analysis

24.2.1 Frequent Itemsets

We will use the Purchases relation shown in Figure 24.1 to illustrate frequent itemsets. The records are shown sorted into groups by transaction. All tuples in a group have the same *transid*, and together they describe a customer transaction, which involves purchases of one or more items. A transaction occurs on a given date, and the name of each purchased item is recorded, along with the purchased quantity. Observe that there is redundancy in Purchases: It can be decomposed by storing *transid–custid–date* triples separately and dropping *custid* and *date*; this may be how the data is actually stored. However, it is convenient to consider the Purchases relation as it is shown in Figure 24.1 in order to compute frequent itemsets. Creating such 'denormalized' tables for ease of data mining is commonly done in the data cleaning step of the KDD process.

By examining the set of transaction groups in Purchases, we can make observations of the form: "In 75 percent of the transactions both a pen and ink are purchased together." It is a statement that describes the transactions in the database. Extrapolation to future transactions should be done with caution, as discussed in Section 24.3.6. Let us begin by introducing the terminology of market basket analysis. An **itemset** is a set of items. The **support** of an itemset is the fraction of transactions in the database that contain all the items in the itemset. In our example, we considered the itemset {pen, ink} and observed that the support of this itemset was 75 percent in Purchases. We can thus conclude that pens and ink are frequently purchased together. If we consider the itemset {milk, juice}, its support is only 25 percent. Thus milk and juice are not purchased together frequently.

Usually the number of sets of items that are frequently purchased together is relatively small, especially as the size of the itemsets increases. We are interested in all itemsets whose support is higher than a user-specified minimum support called *minsup*; we call such itemsets **frequent itemsets**. For example, if the minimum support is set to 70 percent, then the frequent itemsets in our example are {pen}, {ink}, {milk}, {pen, ink}, and {pen, milk}. Note that we are also interested in itemsets that contain only a single item since they identify items that are purchased frequently.

We show an algorithm for identifying frequent itemsets in Figure 24.2. This algorithm relies upon a simple yet fundamental property of frequent itemsets:

> **The a priori property**: Every subset of a frequent itemset must also be a frequent itemset.

The algorithm proceeds iteratively, first identifying frequent itemsets with just one item. In each subsequent iteration, frequent itemsets identified in the previous iteration are extended with another item to generate larger candidate itemsets. By considering only itemsets obtained by enlarging frequent itemsets, we greatly reduce the number of candidate frequent itemsets; this optimization is crucial for efficient execution. The a priori property guarantees that this optimization is correct, that is, we don't miss any frequent itemsets. A single scan of all transactions (the Purchases relation in our example) suffices to determine which candidate itemsets generated in an iteration are frequent itemsets. The algorithm terminates when no new frequent itemsets are identified in an iteration.

```
foreach item,                                                    // Level 1
    check if it is a frequent itemset          // appears in > minsup transactions
k = 1
repeat                  // Iterative, level-wise identification of frequent itemsets
    foreach new frequent itemset I_k with k items                // Level k + 1
        generate all itemsets I_{k+1} with k + 1 items, I_k ⊂ I_{k+1}
    Scan all transactions once and check if
    the generated k + 1-itemsets are frequent
    k = k + 1
until no new frequent itemsets are identified
```

Figure 24.2 An Algorithm for Finding Frequent Itemsets

We illustrate the algorithm on the Purchases relation in Figure 24.1, with *minsup* set to 70 percent. In the first iteration (Level 1), we scan the Purchases relation and determine that each of these one-item sets is a frequent itemset: {*pen*} (appears in all four transactions), {*ink*} (appears in three out of four transactions), and {*milk*} (appears in three out of four transactions).

In the second iteration (Level 2), we extend each frequent itemset with an additional item and generate the following candidate itemsets: {*pen, ink*}, {*pen, milk*}, {*pen, juice*}, {*ink, milk*}, {*ink, juice*}, and {*milk, juice*}. By scanning the Purchases relation again, we determine that the following are frequent itemsets: {*pen, ink*} (appears in three out of four transactions), and {*pen, milk*} (appears in three out of four transactions).

In the third iteration (Level 3), we extend these itemsets with an additional item, and generate the following candidate itemsets: {*pen, ink, milk*}, {*pen, ink, juice*}, and {*pen, milk, juice*}. (Observe that {*ink, milk, juice*} is not generated.) A third scan of the Purchases relation allows us to determine that none of these is a frequent itemset.

The simple algorithm presented here for finding frequent itemsets illustrates the principal feature of more sophisticated algorithms, namely the iterative generation and testing of candidate itemsets. We consider one important refinement of this simple algorithm. Generating candidate itemsets by adding an item to an itemset that is already known to be frequent is an attempt to limit the number of candidate itemsets using the a priori property. The a priori property implies that a candidate itemset can only be frequent if all its subsets are frequent. Thus, we can reduce the number of candidate itemsets further—a priori to scanning the Purchases database—by checking whether all subsets of a newly generated candidate itemset are frequent. Only if all subsets of a candidate itemset are frequent do we compute its support in the subsequent database scan. Compared to the simple algorithm, this refined algorithm generates fewer candidate itemsets at each level and thus reduces the amount of computation performed during the database scan of Purchases.

Consider the refined algorithm on the Purchases table in Figure 24.1 with *minsup*= 70 percent. In the first iteration (Level 1), we determine the frequent itemsets of size one: {*pen*}, {*ink*}, and {*milk*}. In the second iteration (Level 2), only the following candidate itemsets remain when scanning the Purchases table: {*pen, ink*}, {*pen, milk*}, and {*ink, milk*}. Since {*juice*} is not frequent, the itemsets {*pen, juice*}, {*ink, juice*}, and {*milk, juice*} cannot be frequent as well and we can eliminate those itemsets a priori, that is, without considering them during the subsequent scan of the Purchases relation. In the third iteration (Level 3), no further candidate itemsets are generated. The itemset {*pen, ink, milk*} cannot be frequent since its subset {*ink, milk*} is not frequent. Thus, the improved version of the algorithm does not need a third scan of Purchases.

24.2.2 Iceberg Queries

We introduce iceberg queries through an example. Consider again the Purchases relation shown in Figure 24.1. Assume that we want to find pairs of customers and items

such that the customer has purchased the item more than five times. We can express this query in SQL as follows:

```
SELECT    P.custid, P.item, SUM (P.qty)
FROM      Purchases P
GROUP BY  P.custid, P.item
HAVING    SUM (P.qty) > 5
```

Think about how this query would be evaluated by a relational DBMS. Conceptually, for each (*custid*, *item*) pair, we need to check whether the sum of the *qty* field is greater than 5. One approach is to make a scan over the Purchases relation and maintain running sums for each (*custid*,*item*) pair. This is a feasible execution strategy as long as the number of pairs is small enough to fit into main memory. If the number of pairs is larger than main memory, more expensive query evaluation plans that involve either sorting or hashing have to be used.

The query has an important property that is not exploited by the above execution strategy: Even though the Purchases relation is potentially very large and the number of (*custid*,*item*) groups can be huge, the output of the query is likely to be relatively small because of the condition in the `HAVING` clause. Only groups where the customer has purchased the item more than five times appear in the output. For example, there are nine groups in the query over the Purchases relation shown in Figure 24.1, although the output only contains three records. The number of groups is very large, but the answer to the query—the tip of the iceberg—is usually very small. Therefore, we call such a query an **iceberg query**. In general, given a relational schema R with attributes A1, A2, ..., Ak, and B and an aggregation function `aggr`, an iceberg query has the following structure:

```
SELECT    R.A1, R.A2, ..., R.Ak, aggr(R.B)
FROM      Relation R
GROUP BY  R.A1, ..., R.Ak
HAVING    aggr(R.B) >= constant
```

Traditional query plans for this query that use sorting or hashing first compute the value of the aggregation function for all groups and then eliminate groups that do not satisfy the condition in the `HAVING` clause.

Comparing the query with the problem of finding frequent itemsets that we discussed in the previous section, there is a striking similarity. Consider again the Purchases relation shown in Figure 24.1 and the iceberg query from the beginning of this section. We are interested in (*custid*, *item*) pairs that have `SUM` (P.qty) > 5. Using a variation of the a priori property, we can argue that we only have to consider values of the *custid* field where the customer has purchased at least five items overall. We can generate such items through the following query:

```
SELECT    P.custid
FROM      Purchases P
GROUP BY P.custid
HAVING    SUM (P.qty) > 5
```

Similarly, we can restrict the candidate values for the *item* field through the following query:

```
SELECT    P.item
FROM      Purchases P
GROUP BY P.item
HAVING    SUM (P.qty) > 5
```

If we restrict the computation of the original iceberg query to (*custid*, *item*) groups where the field values are in the output of the previous two queries, we eliminate a large number of (*custid*, *item*) pairs a priori! Thus, a possible evaluation strategy is to first compute candidate values for the *custid* and *item* fields, and only to use combinations of these values in the evaluation of the original iceberg query. We first generate candidate field values for individual fields and only use those values that survive the a priori pruning step as expressed in the two previous queries. Thus, the iceberg query is amenable to the same bottom-up evaluation strategy that is used to find frequent itemsets. In particular, we can use the a priori property as follows: We only keep a counter for a group if each individual component of the group satisfies the condition expressed in the `HAVING` clause. The performance improvements of this alternative evaluation strategy over traditional query plans can be very significant in practice.

Even though the bottom-up query processing strategy eliminates many groups a priori, the number of (*custid*, *item*) pairs can still be very large in practice—even larger than main memory. Efficient strategies that use sampling and more sophisticated hashing techniques have been developed; the references at the end of the chapter provide pointers to the relevant literature.

24.3 MINING FOR RULES

Many algorithms have been proposed for discovering various forms of rules that succinctly describe the data. We now look at some widely discussed forms of rules and algorithms for discovering them.

24.3.1 Association Rules

We will use the Purchases relation shown in Figure 24.1 to illustrate association rules. By examining the set of transactions in Purchases, we can identify rules of the form:

$$\{pen\} \Rightarrow \{ink\}$$

This rule should be read as follows: "If a pen is purchased in a transaction, it is likely that ink will also be purchased in that transaction." It is a statement that describes the transactions in the database; extrapolation to future transactions should be done with caution, as discussed in Section 24.3.6. More generally, an **association rule** has the form $LHS \Rightarrow RHS$, where both LHS and RHS are sets of items. The interpretation of such a rule is that if every item in LHS is purchased in a transaction, then it is likely that the items in RHS are purchased as well.

There are two important measures for an association rule:

- **Support:** The support for a set of items is the percentage of transactions that contain all of these items. The support for a rule $LHS \Rightarrow RHS$ is the support for the set of items $LHS \cup RHS$. For example, consider the rule $\{pen\} \Rightarrow \{ink\}$. The support of this rule is the support of the itemset $\{pen, ink\}$, which is 75 percent.
- **Confidence:** Consider transactions that contain all items in LHS. The confidence for a rule $LHS \Rightarrow RHS$ is the percentage of such transactions that also contain all items in RHS. More precisely, let $sup(LHS)$ be the percentage of transactions that contain LHS and let $sup(LHS \cup RHS)$ be the percentage of transactions that contain both LHS and RHS. Then the confidence of the rule $LHS \Rightarrow RHS$ is $sup(LHS \cup RHS)$ / $sup(RHS)$. The confidence of a rule is an indication of the strength of the rule. As an example, consider again the rule $\{pen\} \Rightarrow \{ink\}$. The confidence of this rule is 75 percent; 75 percent of the transactions that contain the itemset $\{pen\}$ also contain the itemset $\{ink\}$.

24.3.2 An Algorithm for Finding Association Rules

A user can ask for all association rules that have a specified minimum support *(minsup)* and minimum confidence *(minconf)*, and various algorithms have been developed for finding such rules efficiently. These algorithms proceed in two steps. In the first step, all frequent itemsets with the user-specified minimum support are computed. In the second step, rules are generated using the frequent itemsets as input. We discussed an algorithm for finding frequent itemsets in Section 24.2, thus we concentrate here on the rule generation part.

Once frequent itemsets are identified, the generation of all possible candidate rules with the user-specified minimum support is straightforward. Consider a frequent itemset

X with support s_X identified in the first step of the algorithm. To generate a rule from X, we divide X into two itemsets *LHS* and *RHS*. The confidence of the rule *LHS* $\Rightarrow$ *RHS* is s_X/s_{LHS}, the ratio of the support of X and the support of *LHS*. From the a priori property, we know that the support of *LHS* is larger than *minsup*, and thus we have computed the support of *LHS* during the first step of the algorithm. We can compute the confidence values for the candidate rule by calculating the ratio support(X)/ support(*LHS*) and then check how the ratio compares to *minconf*.

In general, the expensive step of the algorithm is the computation of the frequent itemsets and many different algorithms have been developed to perform this step efficiently. Rule generation—given that all frequent itemsets have been identified—is straightforward.

In the remainder of this section we will discuss some generalizations of the problem.

24.3.3 Association Rules and ISA Hierarchies

In many cases an **ISA hierarchy** or **category hierarchy** is imposed upon the set of items. In the presence of a hierarchy, a transaction contains for each of its items implicitly all the item's ancestors in the hierarchy. For example, consider the category hierarchy shown in Figure 24.3. Given this hierarchy, the Purchases relation is conceptually enlarged by the eight records shown in Figure 24.4. That is, the Purchases relation has all tuples shown in Figure 24.1 in addition to the tuples shown in Figure 24.4.

The hierarchy allows us to detect relationships between items at different levels of the hierarchy. As an example, the support of the itemset {*ink*, *juice*} is 50 percent, but if we replace *juice* with the more general category *beverage*, the support of the resulting itemset {*ink*, *beverage*} increases to 75 percent. In general, the support of an itemset can only increase if an item is replaced by one of its ancestors in the ISA hierarchy.

Assuming that we actually physically add the eight records shown in Figure 24.4 to the Purchases relation, we can use any algorithm for computing frequent itemsets on the augmented database. Assuming that the hierarchy fits into main memory, we can also perform the addition on-the-fly while we are scanning the database, as an optimization.

Figure 24.3 An ISA Category Taxonomy

transid	*custid*	*date*	*item*	*qty*
111	201	5/1/99	stationery	3
111	201	5/1/99	beverage	9
112	105	6/3/99	stationery	2
112	105	6/3/99	beverage	1
113	106	5/10/99	stationery	1
113	106	5/10/99	beverage	1
114	201	6/1/99	stationery	4
114	201	6/1/99	beverage	4

Figure 24.4 Conceptual Additions to the Purchases Relation with ISA Hierarchy

24.3.4 Generalized Association Rules

Although association rules have been most widely studied in the context of market basket analysis, or analysis of customer transactions, the concept is more general. Consider the Purchases relation as shown in Figure 24.5, grouped by *custid*. By examining the set of customer groups, we can identify association rules such as {pen} ⇒ {milk}. This rule should now be read as follows: "If a pen is purchased by a customer, it is likely that milk will also be purchased by that customer." In the Purchases relation shown in Figure 24.5, this rule has both support and confidence of 100 percent.

transid	*custid*	*date*	*item*	*qty*
112	105	6/3/99	pen	1
112	105	6/3/99	ink	1
112	105	6/3/99	milk	1
113	106	5/10/99	pen	1
113	106	5/10/99	milk	1
114	201	5/15/99	pen	2
114	201	5/15/99	ink	2
114	201	5/15/99	juice	4
111	201	5/1/99	pen	2
111	201	5/1/99	ink	1
111	201	5/1/99	milk	3
111	201	5/1/99	juice	6

Figure 24.5 The Purchases Relation Sorted on Customer Id

Similarly, we can group tuples by date and identify association rules that describe purchase behavior on the same day. As an example consider again the Purchases relation. In this case, the rule {pen} ⇒ {milk} is now interpreted as follows: "On a day when a pen is purchased, it is likely that milk will also be purchased."

If we use the *date* field as grouping attribute, we can consider a more general problem called **calendric market basket analysis**. In calendric market basket analysis, the user specifies a collection of **calendars**. A calendar is any group of dates, e.g., *every Sunday in the year 1999,* or *every first of the month.* A rule holds if it holds on every day in the calendar. Given a calendar, we can compute association rules over the set of tuples whose *date* field falls within the calendar.

By specifying interesting calendars, we can identify rules that might not have enough support and confidence with respect to the entire database, but that have enough support and confidence on the subset of tuples that fall within the calendar. On the other hand, even though a rule might have enough support and confidence with respect to the complete database, it might gain its support only from tuples that fall within a calendar. In this case, the support of the rule over the tuples within the calendar is significantly higher than its support with respect to the entire database.

As an example, consider the Purchases relation with the calendar *every first of the month.* Within this calendar, the association rule *pen* ⇒ *juice* has support and confidence of 100 percent, whereas over the entire Purchases relation, this rule only has 50 percent support. On the other hand, within the calendar, the rule *pen* ⇒ *milk* has support of confidence of 50 percent, whereas over the entire Purchases relation it has support and confidence of 75 percent.

More general specifications of the conditions that must be true within a group for a rule to hold (for that group) have also been proposed. We might want to say that all items in the *LHS* have to be purchased in a quantity of less than two items, and all items in the *RHS* must be purchased in a quantity of more than three.

Using different choices for the grouping attribute and sophisticated conditions as in the above examples, we can identify rules that are more complex than the basic association rules discussed earlier. These more complex rules, nonetheless, retain the essential structure of an association rule as a condition over a group of tuples, with support and confidence measures defined as usual.

24.3.5 Sequential Patterns

Consider the Purchases relation shown in Figure 24.1. Each group of tuples, having the same *custid* value, can be thought of as a *sequence* of transactions ordered by *date.* This allows us to identify frequently arising buying patterns over time.

We begin by introducing the concept of a sequence of itemsets. Each transaction is represented by a set of tuples, and by looking at the values in the *item* column, we get a set of items purchased in that transaction. Thus, the sequence of transactions associated with a customer corresponds naturally to a sequence of itemsets purchased by the customer. For example, the sequence of purchases for customer 201 is ⟨{pen,ink,milk,juice}, {pen,ink,juice}⟩.

A **subsequence** of a sequence of itemsets is obtained by deleting one or more itemsets, and is also a sequence of itemsets. We say that a sequence $\langle a_1, \ldots, a_m \rangle$ is **contained** in another sequence S if S has a subsequence $\langle b_1, \ldots, b_m \rangle$ such that $a_i \subseteq b_i$, for $1 \leq i \leq m$. Thus, the sequence ⟨{pen}, {ink,milk}, {pen,juice}⟩ is contained in ⟨{pen,ink}, {shirt}, {juice,ink,milk}, {juice,pen,milk}⟩. Note that the order of items within each itemset does not matter. However, the order of itemsets does matter: the sequence ⟨{pen}, {ink,milk}, {pen,juice}⟩ is not contained in ⟨{pen,ink}, {shirt}, {juice,pen,milk}, {juice,milk,ink}⟩.

The **support** for a sequence S of itemsets is the percentage of customer sequences of which S is a subsequence. The problem of identifying sequential patterns is to find all sequences that have a user-specified minimum support. A sequence $\langle a_1, a_2, a_3, \ldots, a_m \rangle$ with minimum support tells us that customers often purchase the items in set a_1 in a transaction, then in some subsequent transaction buy the items in set a_2, then the items in set a_3 in a later transaction, and so on.

Like association rules, sequential patterns are statements about groups of tuples in the current database. Computationally, algorithms for finding frequently occurring sequential patterns resemble algorithms for finding frequent itemsets. Longer and longer sequences with the required minimum support are identified iteratively in a manner that is very similar to the iterative identification of frequent itemsets.

24.3.6 The Use of Association Rules for Prediction

Association rules are widely used for prediction, but it is important to recognize that such predictive use is not justified without additional analysis or domain knowledge. Association rules describe existing data accurately but can be misleading when used naively for prediction. For example, consider the rule

$$\{pen\} \Rightarrow \{ink\}$$

The confidence associated with this rule is the conditional probability of an ink purchase given a pen purchase *over the given database*; that is, it is a *descriptive* measure. We might use this rule to guide future sales promotions. For example, we might offer a discount on pens in order to increase the sales of pens and, therefore, also increase sales of ink.

However, such a promotion assumes that pen purchases are good indicators of ink purchases in *future* customer transactions (in addition to transactions in the current database). This assumption is justified if there is a *causal link* between pen purchases and ink purchases; that is, if buying pens causes the buyer to also buy ink. However, we can infer association rules with high support and confidence in some situations where there is no causal link between *LHS* and *RHS*! For example, suppose that pens are always purchased together with pencils, perhaps because of customers' tendency to order writing instruments together. We would then infer the rule

$$\{pencil\} \Rightarrow \{ink\}$$

with the same support and confidence as the rule

$$\{pen\} \Rightarrow \{ink\}$$

However, there is no causal link between pencils and ink. If we promote pencils, a customer who purchases several pencils due to the promotion has no reason to buy more ink. Thus, a sales promotion that discounted pencils in order to increase the sales of ink would fail.

In practice, one would expect that by examining a large database of past transactions (collected over a long time and a variety of circumstances) and restricting attention to rules that occur often (i.e., that have high support), we minimize inferring misleading rules. However, we should bear in mind that misleading, noncausal rules will be generated. Therefore, we should treat the generated rules as possibly, rather than conclusively, identifying causal relationships. Although association rules do not indicate causal relationships between the *LHS* and *RHS*, we emphasize that they provide a useful starting point for identifying such relationships, using either further analysis or a domain expert's judgment; this is the reason for their popularity.

24.3.7 Bayesian Networks

Finding causal relationships is a challenging task, as we saw in Section 24.3.6. In general, if certain events are highly correlated, there are many possible explanations. For example, suppose that pens, pencils, and ink are purchased together frequently. It might be the case that the purchase of one of these items (e.g., ink) depends causally upon the purchase of another item (e.g., pen). Or it might be the case that the purchase of one of these items (e.g., pen) is strongly correlated with the purchase of another (e.g., pencil) because there is some underlying phenomenon (e.g., users' tendency to think about writing instruments together) that causally influences both purchases. How can we identify the true causal relationships that hold between these events in the real world?

One approach is to consider each possible combination of causal relationships among the variables or events of interest to us and to evaluate the likelihood of each combina-

tion on the basis of the data available to us. If we think of each combination of causal relationships as a *model* of the real world underlying the collected data, we can assign a score to each model by considering how consistent it is (in terms of probabilities, with some simplifying assumptions) with the observed data. Bayesian networks are graphs that can be used to describe a class of such models, with one node per variable or event, and arcs between nodes to indicate causality. For example, a good model for our running example of pens, pencils, and ink is shown in Figure 24.6. In general, the number of possible models is exponential in the number of variables, and considering all models is expensive, so some subset of all possible models is evaluated.

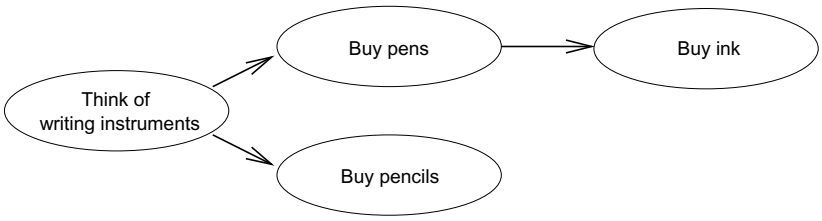

Figure 24.6 Bayesian Network Showing Causality

24.3.8 Classification and Regression Rules

Consider the following view that contains information from a mailing campaign performed by a publishing company:

InsuranceInfo(*age:* `integer`, *cartype:* `string`, *highrisk:* `boolean`)

The InsuranceInfo view has information about current customers. Each record contains a customer's age and type of car as well as a flag indicating whether the person is considered a high-risk customer. If the flag is true, the customer is considered high-risk. We would like to use this information to identify rules that predict the insurance risk of new insurance applicants whose age and car type are known. For example, one such rule could be: "If *age* is between 16 and 25 and *cartype* is either Sports or Truck, then the risk is high."

Note that the rules we want to find have a specific structure. We are not interested in rules that predict the age or type of car of a person; we are only interested in rules that predict the insurance risk. Thus, there is one designated attribute whose value we would like to predict and we will call this attribute the **dependent** attribute. The other attributes are called **predictor** attributes. In our example, the dependent attribute in the InsuranceInfo view is the *highrisk* attribute and the predictor attributes are *age* and *cartype*. The general form of the types of rules we want to discover is:

$$P_1(X_1) \wedge P_2(X_2) \ldots \wedge P_k(X_k) \Rightarrow Y = c$$

The predictor attributes $X_1, \ldots, X_k$ are used to predict the value of the dependent attribute Y. Both sides of a rule can be interpreted as conditions on fields of a tuple. The $P_i(X_i)$ are predicates that involve attribute X_i. The form of the predicate depends on the type of the predictor attribute. We distinguish two types of attributes: **numerical** and **categorical** attributes. For numerical attributes, we can perform numerical computations such as computing the average of two values, whereas for categorical attributes their domain is a set of values. In the InsuranceInfo view, *age* is a numerical attribute whereas *cartype* and *highrisk* are categorical attributes. Returning to the form of the predicates, if X_i is a numerical attribute, its predicate P_i is of the form $li \leq X_i \leq hi$; if X_i is a categorical attribute, P_i is of the form $X_i \in \{v_1, \ldots, v_j\}$.

If the dependent attribute is categorical, we call such rules **classification rules**. If the dependent attribute is numerical, we call such rules **regression rules**.

For example, consider again our example rule: "If *age* is between 16 and 25 and *cartype* is either Sports or Truck, then *highrisk* is true." Since *highrisk* is a categorical attribute, this rule is a classification rule. We can express this rule formally as follows:

$$(16 \leq age \leq 25) \wedge (cartype \in \{\text{Sports, Truck}\}) \Rightarrow highrisk = \texttt{true}$$

We can define support and confidence for classification and regression rules, as for association rules:

- **Support:** The support for a condition C is the percentage of tuples that satisfy C. The support for a rule $C1 \Rightarrow C2$ is the support for the condition $C1 \wedge C2$.
- **Confidence:** Consider those tuples that satisfy condition $C1$. The confidence for a rule $C1 \Rightarrow C2$ is the percentage of such tuples that also satisfy condition $C2$.

As a further generalization, consider the right-hand side of a classification or regression rule: $Y = c$. Each rule predicts a value of Y for a given tuple based on the values of predictor attributes $X1, \ldots, Xk$. We can consider rules of the form:

$$P_1(X_1) \wedge \ldots \wedge P_k(X_k) \Rightarrow Y = f(X_1, \ldots, X_k)$$

where f is some function. We will not discuss such rules further.

Classification and regression rules differ from association rules by considering continuous and categorical fields, rather than only one field that is set-valued. Identifying such rules efficiently presents a new set of challenges and we will not discuss the general case of discovering such rules. We will discuss a special type of such rules in Section 24.4.

Classification and regression rules have many applications. Examples include classification of results of scientific experiments, where the type of object to be recognized

depends on the measurements taken; direct mail prospecting, where the response of a given customer to a promotion is a function of his or her income level and age; and car insurance risk assessment, where a customer could be classified as risky depending on his age, profession, and car type. Example applications of regression rules include financial forecasting, where the price of coffee futures could be some function of the rainfall in Colombia a month ago, and medical prognosis, where the likelihood of a tumor being cancerous is a function of measured attributes of the tumor.

24.4 TREE-STRUCTURED RULES

In this section, we discuss the problem of discovering classification and regression rules from a relation, but we consider only rules that have a very special structure. The type of rules that we discuss can be represented by a tree, and typically the tree itself is the output of the data mining activity. Trees that represent classification rules are called **classification trees** or **decision trees** and trees that represent regression rules are called **regression trees**.

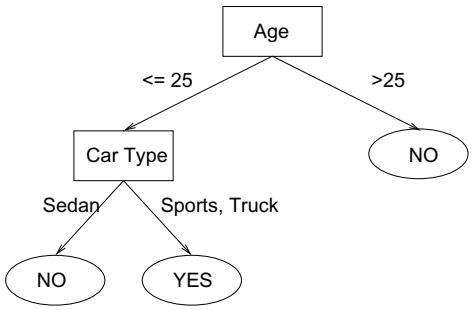

Figure 24.7 Insurance Risk Example Decision Tree

As an example, consider the decision tree shown in Figure 24.7. Each path from the root node to a leaf node represents one classification rule. For example, the path from the root to the leftmost leaf node represents the classification rule: "If a person is 25 years or younger and drives a sedan, then he is likely to have a low insurance risk." The path from the root to the right-most leaf node represents the classification rule: "If a person is older than 25 years, then he is likely to have a low insurance risk."

Tree-structured rules are very popular since they are easy to interpret. Ease of understanding is very important, since the result of any data mining activity needs to be comprehensible by nonspecialists. In addition, studies have shown that despite limitations in structure, tree-structured rules are very accurate. There exist efficient

algorithms to construct tree-structured rules from large databases. We will discuss a sample algorithm for decision tree construction in the remainder of this section.

24.4.1 Decision Trees

A decision tree is a graphical representation of a collection of classification rules. Given a data record, the tree directs the record from the root to a leaf. Each internal node of the tree is labeled with a predictor attribute. This attribute is often called a **splitting attribute**, because the data is 'split' based on conditions over this attribute. The outgoing edges of an internal node are labeled with predicates that involve the splitting attribute of the node; every data record entering the node must satisfy the predicate labeling exactly one outgoing edge. The combined information about the splitting attribute and the predicates on the outgoing edges is called the **splitting criterion** of the node. A node with no outgoing edges is called a **leaf node**. Each leaf node of the tree is labeled with a value of the dependent attribute. We only consider binary trees where internal nodes have two outgoing edges, although trees of higher degree are possible.

Consider the decision tree shown in Figure 24.7. The splitting attribute of the root node is *age*, the splitting attribute of the left child of the root node is *cartype*. The predicate on the left outgoing edge of the root node is $age \leq 25$, the predicate on the right outgoing edge is $age > 25$.

We can now associate a classification rule with each leaf node in the tree as follows. Consider the path from the root of the tree to the leaf node. Each edge on that path is labeled with a predicate. The conjunction of all these predicates makes up the left hand side of the rule. The value of the dependent attribute at the leaf node makes up the right-hand side of the rule. Thus, the decision tree represents a collection of classification rules, one for each leaf node.

A decision tree is usually constructed in two phases. In phase one, the **growth phase**, an overly large tree is constructed. This tree represents the records in the input database very accurately; for example, the tree might contain leaf nodes for individual records from the input database. In phase two, the **pruning phase**, the final size of the tree is determined. The rules represented by the tree constructed in phase one are usually overspecialized. By reducing the size of the tree, we generate a smaller number of more general rules that are better than a very large number of very specialized rules. Algorithms for tree pruning are beyond our scope of discussion here.

Classification tree algorithms build the tree greedily top-down in the following way. At the root node, the database is examined and the locally 'best' splitting criterion is computed. The database is then partitioned, according to the root node's splitting criterion, into two parts, one partition for the left child and one partition for the

Input: node n, partition D, split selection method $\mathcal{S}$
Output: decision tree for D rooted at node n

Top-Down Decision Tree Induction Schema:
BuildTree(Node n, data partition D, split selection method $\mathcal{S}$)
(1) Apply $\mathcal{S}$ to D to find the splitting criterion
(2) **if** (a good splitting criterion is found)
(3) Create two children nodes n_1 and n_2 of n
(4) Partition D into D_1 and D_2
(5) BuildTree(n_1, D_1, $\mathcal{S}$)
(6) BuildTree(n_2, D_2, $\mathcal{S}$)
(7) **endif**

Figure 24.8 Decision Tree Induction Schema

right child. The algorithm then recurses on each child. This schema is depicted in Figure 24.8.

The splitting criterion at a node is found through application of a **split selection method**. A split selection method is an algorithm that takes as input (part of) a relation and outputs the locally 'best' splitting criterion. In our example, the split selection method examines the attributes *cartype* and *age*, selects one of them as splitting attribute, and then selects the splitting predicates. Many different, very sophisticated split selection methods have been developed; the references provide pointers to the relevant literature.

age	*cartype*	*highrisk*
23	Sedan	false
30	Sports	false
36	Sedan	false
25	Truck	true
30	Sedan	false
23	Truck	true
30	Truck	false
25	Sports	true
18	Sedan	false

Figure 24.9 The InsuranceInfo Relation

24.4.2 An Algorithm to Build Decision Trees

If the input database fits into main memory we can directly follow the classification tree induction schema shown in Figure 24.8. How can we construct decision trees when the input relation is larger than main memory? In this case, step (1) in Figure 24.8 fails, since the input database does not fit in memory. But we can make one important observation about split selection methods that helps us to reduce the main memory requirements.

Consider a node of the decision tree. The split selection method has to make two decisions after examining the partition at that node: (i) It has to select the splitting attribute, and (ii) It has to select the splitting predicates for the outgoing edges. Once decided on the splitting criterion, the algorithm is recursively applied to each of the children of the node. Does a split selection method actually need the complete database partition as input? Fortunately, the answer is no.

Split selection methods that compute splitting criteria that involve a single predictor attribute at each node evaluate each predictor attribute individually. Since each attribute is examined separately, we can provide the split selection method with aggregated information about the database instead of loading the complete database into main memory. Chosen correctly, this aggregated information is sufficient to compute the 'best' splitting criterion—the same splitting criterion as if the complete database would reside in main memory.

Since the split selection method examines all predictor attributes, we need aggregated information about each predictor attribute. We call this aggregated information the **AVC set** of the predictor attribute. The AVC set of a predictor attribute X at node n is the projection of n's database partition onto X and the dependent attribute where counts of the individual values in the domain of the dependent attribute are aggregated. (The acronym AVC stands for **A**ttribute-**V**alue, **C**lasslabel, because the values of the dependent attribute are often called **class labels**.) For example, consider the InsuranceInfo relation as shown in Figure 24.9. The AVC set of the root node of the tree for predictor attribute *age* is the result of the following database query:

```
SELECT    R.age, R.highrisk, COUNT (*)
FROM      InsuranceInfo R
GROUP BY  R.age, R.highrisk
```

The AVC set for the left child of the root node for predictor attribute *cartype* is the result of the following query:

```
SELECT    R.cartype, R.highrisk, COUNT (*)
FROM      InsuranceInfo R
```

```
WHERE      R.age <= 25
GROUP BY R.cartype, R.highrisk
```

The two AVC sets of the root node of the tree are shown in Figure 24.10.

Car type	highrisk	
	true	false
Sedan	0	4
Sports	1	1
Truck	2	1

Age	highrisk	
	true	false
18	0	1
23	1	1
25	2	0
30	0	3
36	0	1

Figure 24.10 AVC Group of the Root Node for the InsuranceInfo Relation

We define the **AVC group** of a node n to be the set of the AVC sets of all predictor attributes at node n. In our example of the InsuranceInfo relation, there are two predictor attributes; therefore, the AVC group of any node consists of two AVC sets.

How large are AVC sets? Note that the size of the AVC set of a predictor attribute X at node n depends only on the number of distinct attribute values of X and the size of the domain of the dependent attribute. For example, consider the AVC sets shown in Figure 24.10. The AVC set for the predictor attribute *cartype* has three entries, and the AVC set for predictor attribute *age* has five entries, although the InsuranceInfo relation as shown in Figure 24.9 has nine records. For large databases, the size of the AVC sets is independent of the number of tuples in the database, except if there are attributes with very large domains, e.g., a real-valued field that is recorded at a very high precision with many digits after the decimal point.

If we make the simplifying assumption that all the AVC sets of the root node together fit into main memory, then we can construct decision trees from very large databases as follows: We make a scan over the database and construct the AVC group of the root node in memory. Then we run the split selection method of our choice with the AVC group as input. After the split selection method computes the splitting attribute and the splitting predicates on the outgoing nodes, we partition the database and recurse. Note that this algorithm is very similar to the original algorithm shown in Figure 24.8; the only modification necessary is shown in Figure 24.11. In addition, this algorithm is still independent of the actual split selection method involved.

24.5 CLUSTERING

In this section we discuss the **clustering problem**. The goal is to partition a set of records into groups such that records within a group are similar to each other and

Input: node n, partition D, split selection method $\mathcal{S}$
Output: decision tree for D rooted at node n

Top-Down Decision Tree Induction Schema:
BuildTree(Node n, data partition D, split selection method $\mathcal{S}$)
(1a) Make a scan over D and construct the AVC group of n in-memory
(1b) Apply $\mathcal{S}$ to the AVC group to find the splitting criterion

Figure 24.11 Classification Tree Induction Refinement with AVC Groups

records that belong to two different groups are dissimilar. Each such group is called a **cluster** and each record belongs to exactly one cluster.[1] Similarity between records is measured computationally by a **distance function**. A distance function takes two input records and returns a value that is a measure of their similarity. Different applications have different notions of similarity and there is no one measure that works for all domains.

As an example, consider the schema of the CustomerInfo view:

CustomerInfo(*age:* `int`, *salary:* `real`)

We can plot the records in the view on a two-dimensional plane as shown in Figure 24.12. The two coordinates of a record are the values of the record's *salary* and *age* fields. We can visually identify three clusters: Young customers who have low salaries, young customers with high salaries, and older customers with high salaries.

Usually, the output of a clustering algorithm consists of a **summarized representation** of each cluster. The type of summarized representation depends strongly on the type and shape of clusters the algorithm computes. For example, assume that we have spherical clusters as in the example shown in Figure 24.12. We can summarize each cluster by its *center* (often also called the *mean*) and its *radius* which are defined as follows. Given a collection of records $r_1, \ldots, r_n$, their **center** C and **radius** R are defined as follows:

$$C = \frac{1}{n} \sum_{i=1}^{n} r_i, \text{ and } R = \sqrt{\frac{\sum_{i=1}^{n} (r_i - C)}{n}}$$

There are two types of clustering algorithms. A **partitional** clustering algorithm partitions the data into k groups such that some criterion that evaluates the clustering quality is optimized. The number of clusters k is a parameter whose value is specified

[1] There are clustering algorithms that allow overlapping clusters, where a record can potentially belong to several clusters.

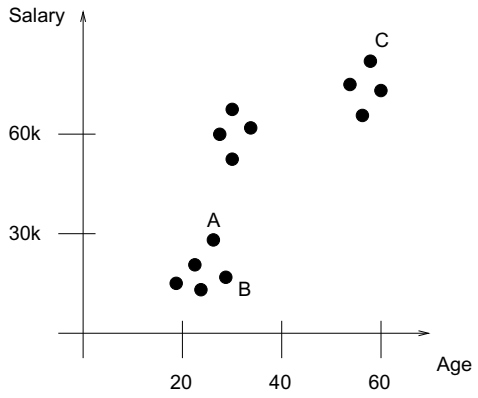

Figure 24.12 Records in CustomerInfo

by the user. A **hierarchical** clustering algorithm generates a sequence of partitions of the records. Starting with a partition in which each cluster consists of one single record, the algorithm merges two partitions in each step until only one single partition remains in the end.

24.5.1 A Clustering Algorithm

Clustering is a very old problem and numerous algorithms have been developed to cluster a collection of records. Traditionally, the number of records in the input database was assumed to be relatively small and the complete database was assumed to fit into main memory. In this section we describe a clustering algorithm called BIRCH that handles very large databases. The design of BIRCH reflects the following two assumptions:

- The number of records is potentially very large and therefore we want to make only one scan over the database.
- We have only a limited amount of main memory available.

A user can set two parameters to control the BIRCH algorithm. The first parameter is a threshold on the amount of main memory available. This main memory threshold translates into a maximum number of cluster summaries k that can be maintained in memory. The second parameter ϵ is an initial threshold for the radius of any cluster. The value of ϵ is an upper bound on the radius of any cluster and controls the number of clusters that the algorithm discovers. If ϵ is small, we discover many small clusters; if ϵ is large, we discover very few clusters, each of which is relatively large. We say that a cluster is **compact** if its radius is smaller than ϵ.

BIRCH always maintains k or fewer cluster summaries (C_i, R_i) in main memory, where C_i is the center of cluster i and R_i is the radius of cluster i. The algorithm always maintains compact clusters, i.e., the radius of each cluster is less than ϵ. If this invariant cannot be maintained with the given amount of main memory, ϵ is increased as described below.

The algorithm reads records from the database sequentially and processes them as follows:

1. Compute the distance between record r and each of the existing cluster centers. Let i be the cluster index such that the distance between r and C_i is the smallest.
2. Compute the value of the new radius R_i' of the ith cluster under the assumption that r is inserted into it. If $R_i' \leq \epsilon$, then the ith cluster remains compact and we assign r to the ith cluster by updating its center and setting its radius to R_i'. If $R_i' > \epsilon$, then the ith cluster is no longer compact if we insert r into it. Therefore, we start a new cluster containing only the record r.

The second step above presents a problem if we already have the maximum number of cluster summaries, k. If we now read a record that requires us to create a new cluster, we don't have the main memory required to hold its summary. In this case, we increase the radius threshold ϵ—using some heuristic to determine the increase—in order to *merge* existing clusters: An increase of ϵ has two consequences. First, existing clusters can accommodate 'more' records, since their maximum radius has increased. Second, it might be possible to merge existing clusters such that the resulting cluster is still compact. Thus, an increase in ϵ usually reduces the number of existing clusters.

The complete BIRCH algorithm uses a balanced in-memory tree, which is similar to a B+ tree in structure, to quickly identify the closest cluster center for a new record. A description of this data structure is beyond the scope of our discussion.

24.6 SIMILARITY SEARCH OVER SEQUENCES

A lot of information stored in databases consists of sequences. In this section, we introduce the problem of similarity search over a collection of sequences. Our query model is very simple: We assume that the user specifies a **query sequence** and wants to retrieve all data sequences that are similar to the query sequence. Similarity search is different from 'normal' queries in that we are not only interested in sequences that match the query sequence exactly, but also in sequences that differ only slightly from the query sequence.

We begin by describing sequences and similarity between sequences. A **data sequence** X is a series of numbers $X = \langle x_1, \ldots, x_k \rangle$. Sometimes X is also called a **time series**. We call k the **length** of the sequence. A **subsequence** $Z = \langle z_1, \ldots, z_j \rangle$ is obtained