17

OLAP and Data Mining

This chapter is an introduction to the concepts and techniques from the fields of *online analytical processing* (OLAP), *data warehousing*, and *data mining*. Recall from our discussion in Section 1.4 that while online transaction processing (OLTP) is concerned with using a database to maintain an accurate model of some real-world situation, OLAP and data mining are concerned with using the information in a database to guide strategic decisions. OLAP is concerned with obtaining specific information, while data mining can be viewed as knowledge discovery. Data warehouses are used to accumulate and store the information needed for OLAP and data mining queries.

In Sections 17.2 and 17.3, you will learn about the multidimensional model for OLAP and related notions, such as CUBE and ROLLUP. In Sections 17.7 and beyond, you will be introduced to a number of important techniques, including the a priori algorithm for computing associations, the ID3 and C4.5 algorithms for training decision trees using the information gain measure, the perceptron and back propagation learning algorithms for neural nets, and the K-means and hierarchical algorithms (using dendrograms) for clustering.

17.1 OLAP and Data Warehouses—Old and New

Why is so much free material available on the Internet—all for just filling out a form? In fact, the goodies you receive are not free—you are paying for them by providing information about yourself, and, when you buy on the Internet, you are providing even more information about yourself—your buying habits.

You also provide information about yourself when you purchase items in a department store or supermarket with your credit card. In these cases, you are inputting information into a transaction processing system, and the system is saving that information for future use.

What is done with this information? In many situations, it is combined with what is known about you from other sources, stored in a database, and then

- It might be combined with information about the purchases of other people to help an enterprise plan its inventory, advertising, or other aspects of its future strategy.

- It might be used to produce an individualized profile of your buying (or browsing) habits so that an enterprise can target its marketing to you through the mail or in other ways. Perhaps in the future, the people in your zip code area will see different TV commercials based on information about their purchasing habits. Or perhaps you will see TV commercials personalized for you.

These trends in information gathering and assimilation have serious implications for personal privacy. Do you want strangers to be able to access this information and use it in ways of which you might not approve? However, that is not our concern in this text. Our concern is to understand the techniques that might be used to analyze this data.

The applications that use data of this type are referred to as **online analytic processing**, or **OLAP**, in contrast with **online transaction processing**, or **OLTP**. The two types of applications have different goals and different technical requirements.

- The goal of OLTP is to maintain a database that is an accurate model of some real-world enterprise. The system must provide sufficiently large transaction throughput and low response time to keep up with the load and avoid user frustration. OLTP applications are characterized by
  - Short, simple transactions
  - Relatively frequent updates
  - Transactions that access only a tiny fraction of the database
- The goal of OLAP is to use the information in a database to guide strategic decisions. The databases involved are usually very large and often need not be completely accurate or up to date. Nor is fast response always required. OLAP applications are characterized by
  - Complex queries
  - Infrequent updates
  - Transactions that access a significant fraction of the database

One might say that OLTP is *operational* in that it deals with the everyday operations of the enterprise, while OLAP is *decisional* in that it deals with decision making by the managers of the enterprise.

The example of an OLAP application in Section 1.4 involved managers of a supermarket chain who want to make one-time (not preprogrammed) queries to the database to gather information they need in order to make a specific decision. This illustrates the traditional use of OLAP—ad hoc queries, often made by people who are not highly technical.

The OLAP examples at the beginning of this section describe some of the newer uses. Businesses are using preprogrammed queries against OLAP databases on an ongoing operational basis to customize marketing and other aspects of their business. These queries are often complex and, since they are key to the business and used operationally (perhaps daily or weekly), are designed and implemented by professionals.

In traditional OLAP applications, the information in the OLAP database is often just the data the business happens to gather during day-to-day operations—perhaps

in its OLTP systems. In newer applications, the business often makes an active effort to gather—perhaps even to purchase—the additional information needed for its planned application.

As the *A* in OLAP implies, the goal of an OLAP application is to *analyze* data for use in some application. Thus, there are often two separate but related subjects.

- *The analysis to be performed.* For example, a company wants to decide the mix of products to manufacture during the next accounting period. It develops an analysis procedure that requires as input the sales for the last period and the history of sales for the equivalent periods over the past five years.
- *The methods to efficiently obtain the large amounts of data required for the analysis.* For example, how can the company extract the required sales data from databases in its subsidiary departments? In what form should it store this data in the OLAP database? How can it retrieve the data efficiently when needed for the analysis?

The first issue, analysis, is not a database problem since it requires algorithms specific to the particular business in which the company engages. Our interest is primarily in the second issue—database support for these analytical procedures. For our purposes, we assume that the retrieved data is simply displayed on the screen. However, in many situations—particularly in newer applications—this data is input to sophisticated analysis procedures.

Data warehouses. OLAP databases are usually stored in special OLAP servers, often called **data warehouses**, which are structured to support the OLAP queries that will be made against them. OLAP queries are often so complex that if they were run in an OLTP environment, they would slow down OLTP transactions to an unacceptable degree.

We will discuss some of the issues involved in populating a data warehouse in Section 17.6. First, we will look at the kinds of data we might want to store in the warehouse.

17.2 A Multidimensional Model for OLAP Applications

Fact tables and dimension tables. Many OLAP applications are similar to the supermarket example of Section 1.4: analysis of sales of different products in different supermarkets over different time periods. We might describe this sales data with a relational table such as that shown in Figure 17.1. `Market_Id` identifies a particular supermarket, `Product_Id` identifies a particular product, `Time_Id` identifies a particular time interval, and `Sales_Amt` identifies the dollar value of the sales of that product at that supermarket in that time period. Such a table is called a **fact table** because it contains all of the facts about the data to be analyzed.

We can view this data as **multidimensional**. The `Market_Id`, `Product_Id`, and `Time_Id` attributes are the dimensions and correspond to the arguments of a function. The `Sales_Amt` attribute corresponds to the value of the function.

SALES	Market_Id	Product_Id	Time_Id	Sales_Amt
	M1	P1	T1	1000
	M1	P2	T1	2000
	M1	P3	T1	1500
	M1	P4	T1	2500
	M2	P1	T1	500
	M2	P2	T1	800
	M2	P3	T1	0
	M2	P4	T1	3333
	M3	P1	T1	5000
	M3	P2	T1	8000
	M3	P3	T1	10
	M3	P4	T1	3300
	M1	P1	T2	1001
	M1	P2	T2	2001
	M1	P3	T2	1501
	M1	P4	T2	2501
	M2	P1	T2	501
	M2	P2	T2	801
	M2	P3	T2	1
	M2	P4	T2	3334
	M3	P1	T2	5001
	M3	P2	T2	8001
	M3	P3	T2	11
	M3	P4	T2	3301
	M1	P1	T3	1002
	M1	P2	T3	2002
	M1	P3	T3	1502
	M1	P4	T3	2502
	M2	P1	T3	502
	M2	P2	T3	802
	M2	P3	T3	2
	M2	P4	T3	333
	M3	P1	T3	5002
	M3	P2	T3	8002
	M3	P3	T3	12
	M3	P4	T3	3302

FIGURE 17.1 The fact table for the supermarket application.

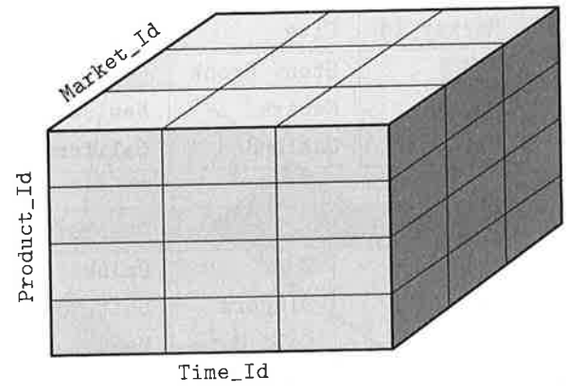

FIGURE 17.2 Three-dimensional cube for the supermarket application.

We can also think of the data in a fact table as being arranged in a multidimensional cube. Thus, in the supermarket example, the data is arranged in the three-dimensional cube shown in Figure 17.2, where the dimensions of the cube are `Market_Id`, `Product_Id`, and `Time_Id` and the vertices, or **cells**, of the cube contain the corresponding `Sales_Amt`. Such a multidimensional view can be an intuitive way to think about OLAP queries and their results.

Additional information about the dimensions can be stored in **dimension tables**, which describe dimension attributes. For the supermarket example, these tables might be called MARKET, PRODUCT, and TIME, as shown in Figure 17.3. The MARKET table describes the market: its city, state, and region. In a more realistic example, the MARKET table would contain a row for each supermarket in the chain, which might include many markets in each city, many cities in each state, and many states in each region.

Star schema. The relations corresponding to the supermarket example can be displayed in a diagram, as in Figure 17.4. The figure suggests a star, with the fact table at the center and the dimension tables radiating from it. This type of schema, called a **star schema**, is very common in OLAP applications. It is interesting to note that a star schema corresponds to a very common fragment of an entity-relation diagram, where the fact table is a relationship and the dimension tables are entities.

If the dimension tables are normalized (so that each might become several tables), the figure gets a bit more complex and is called a **snowflake schema**. However, for two reasons, dimension tables are rarely normalized:

1. They are so small compared with the fact table that the space saved due to the elimination of redundancy is negligible.
2. They are updated so infrequently that update anomalies are not an issue. Moreover, in this situation, decomposing the relations into 3NF or BCNF might lead to significant query overhead, as explained in Section 6.13.

MARKET	Market_Id	City	State	Region
	M1	Stony Brook	New York	East
	M2	Newark	New Jersey	East
	M3	Oakland	California	West

PRODUCT	Product_Id	Name	Category	Price
	P1	Beer	Drink	1.98
	P2	Diapers	Soft Goods	2.98
	P3	Cold Cuts	Meat	3.98
	P4	Soda	Drink	1.25

TIME	Time_Id	Week	Month	Quarter
	T1	Wk-1	January	First
	T2	Wk-24	June	Second
	T3	Wk-52	December	Fourth

FIGURE 17.3 Dimension tables for the supermarket application.

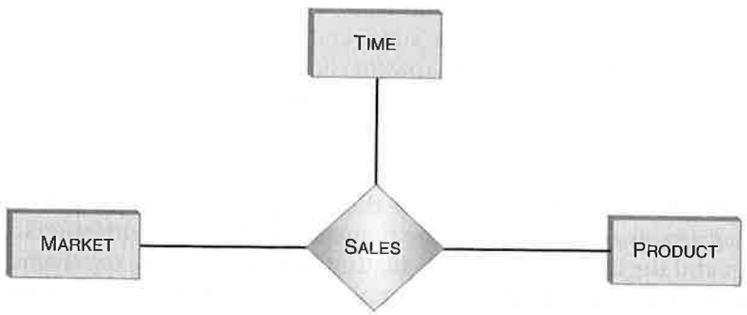

FIGURE 17.4 Star schema for the supermarket example.

Instead of a star schema, many OLAP applications use a **constellation schema**, which consists of several fact tables that might share one or more dimension tables. For example, the supermarket application might maintain a fact table called INVENTORY, with dimension tables WAREHOUSE, PRODUCT, and TIME, as shown in Figure 17.5. Note that the PRODUCT and TIME dimension tables are shared with the SALES fact table, whereas the WAREHOUSE table, which describes where the inventory is stored, is not shared.

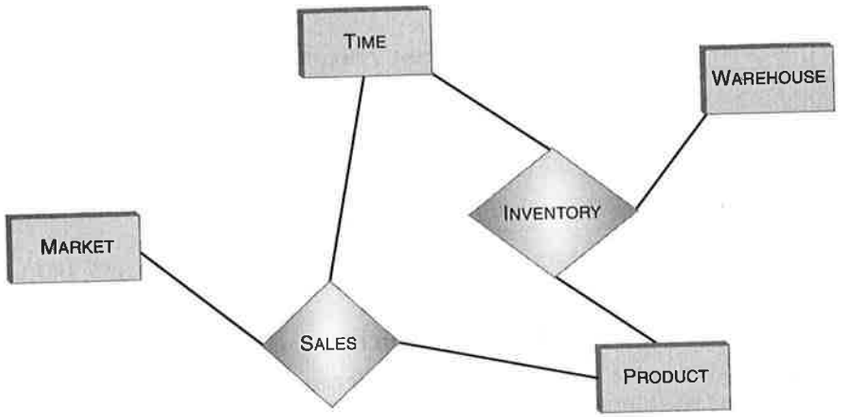

FIGURE 17.5 Constellation schema for the expanded supermarket example.

SUM(Sales_Amt)		Market_Id		
		M1	M2	M3
Product_Id	P1	3003	1503	15003
	P2	6003	2402	24003
	P3	4503	3	33
	P4	7503	7000	9903

FIGURE 17.6 Query result that aggregates `Sales_Amt` on the time dimension.

17.3 Aggregation

Many OLAP queries involve **aggregation** of the data in the fact table. For example, a query that produces the total sales (over time) of each product in each market can be expressed with the SQL statement

```
SELECT      S.Market_Id, S.Product_Id, SUM(S.Sales_Amt)
FROM        SALES S
GROUP BY    S.Market_Id, S.Product_Id
```

which returns the result table shown in Figure 17.6. Here we depict the result table as a two-dimensional cube with the values of the aggregation over `Sales_Amt` placed in the cells. Since this aggregation is over the entire time dimension (i.e., the result does not depend on the time coordinate), it produces a reduced-dimensional view of the data—two dimensions instead of three.

SUM(Sales_Amt)		Region			
		North	South	East	West
Product_Id	P1	0	0	4506	15003
	P2	0	0	8405	24003
	P3	0	0	4506	33
	P4	0	0	14503	9903

FIGURE 17.7 Query result that drills down on regions.

17.3.1 Drilling, Slicing, Rolling, and Dicing

Some dimension tables represent an **aggregation hierarchy**. For example, the MARKET table represents the hierarchy

```
Market_Id → City → State → Region
```

meaning that supermarkets are in cities, cities are in states, and states are in regions. We can perform queries at different levels of a hierarchy, as shown here:

```
SELECT      S.Product_Id, M.Region, SUM(S.Sales_Amt)
FROM        SALES S,    MARKET M
WHERE       M.Market_Id = S.Market_Id                    17.1
GROUP BY    S.Product_Id, M.Region
```

This produces the table of Figure 17.7, which aggregates total sales per product for each region over all time.

When we execute a sequence of queries that move down a hierarchy—from general to specific, such as moving from aggregation over regions to aggregation over states—we are said to be **drilling down**. Drilling down, of course, requires access to more specific information than is contained in the result of a more general query. Thus, in order to aggregate over states, we must either use the fact table or a previously computed table that aggregates over cities. Thus, we might use the fact table to drill down to states with

```
SELECT      S.Product_Id, M.State, SUM(S.Sales_Amt)
FROM        SALES S,    MARKET M
WHERE       M.Market_Id = S.Market_Id                    17.2
GROUP BY    S.Product_Id, M.State
```

When we move up the hierarchy (for example, from aggregation over states to aggregation over regions), we are said to be **rolling up**. For example, if we were to save the result of executing the query (17.2) as a table called STATE_SALES, then we could roll up the hierarchy using the query

SUM(Sales_Amt)		Quarter			
		First	Second	Third	Fourth
Product_Id	P1	6500	6503	0	6506
	P2	10800	10803	0	10806
	P3	1510	1513	0	1516
	P4	9133	9136	0	6137

FIGURE 17.8 Query result that presents total product sales for each quarter.

```
SELECT      T.Product_Id, M.Region, SUM(T.Sales_Amt)
FROM        STATE_SALES T,  MARKET M
WHERE       R.State = T.State
GROUP BY    T.Product_Id, R.Region
```

While this is not an efficient way to aggregate over regions, it demonstrates the ability to use previously computed results when rolling up. This is an important optimization and has motivated the inclusion in SQL of special features, which we will discuss shortly. It is very common to roll up or drill down using the time dimension—for example, to summarize sales on a daily, monthly, or quarterly basis.

Here is a bit more OLAP terminology. When we view the data in the form of a multidimensional cube and then select a subset of the axes, we are said to be performing a **pivot** (we are reorienting the multidimensional cube). The selected axes correspond to the list of attributes in the GROUP BY clause. Pivoting is usually followed by aggregation on the remaining axes.

As an example, the following query performs a pivot of the multidimensional cube to view it from the product and time dimensions. It finds the total sales (over all markets) of each product for each quarter (of the current year) and produces the table of Figure 17.8:

```
SELECT      S.Product_Id, T.Quarter,  SUM(S.Sales_Amt)
FROM        SALES S,  TIME T
WHERE       T.Time_Id = S.Time_Id                        17.3
GROUP BY    S.Product_Id, T.Quarter
```

If we next ask the same query, but use the GROUP BY clause to group by years instead of by quarters, we are rolling up the time hierarchy.

```
SELECT      S.Product_Id, T.Year, SUM(S.Sales_Amt)
FROM        SALES S,  TIME T
WHERE       T.Time_Id = S.Time_Id
GROUP BY    S.Product_Id, T.Year
```

SQL:1999/2003 and some OLAP vendors support a new SQL clause, ROLLUP, to simplify this process (see Section 17.3.2). However, notice that a corresponding drill-down clause is usually *not* provided. The reason is that rollup is not only a convenience but also an optimization device. If the user first asks to aggregate over quarters and then rolls up the result to years, the OLAP system does not need to compute from scratch but can aggregate to the year using the previously computed aggregation results for each quarter. No such optimization is possible for drilling down. However, OLAP systems typically let the user precompute and cache certain aggregations in order to speed up the drilling down process. For instance, if we are expected to drill down to weeks and quarters, we might ask the system to precompute aggregations of the data cube grouping by weeks in the time dimension. Then when we drill down to weeks or quarters we will not need to sum up the sales figures for each particular value of the `Time_Id` attribute. Instead, when we group by `Week` or `Quarter` we will be summing up the already precomputed weekly sales figures—a much smaller number of items.

Rollup can be optimized through the reuse of the results of previous requests. Drilling down can be sped up by precomputing and caching certain carefully selected aggregations.

Not all aggregation hierarchies are linear, as is the location hierarchy. The time hierarchy shown in Figure 17.9, for example, is a lattice. Weeks are not fully contained in months—the same week can fall on the boundary for two different months. Thus, we can roll up days into either weeks or months, but we can only roll up weeks into quarters.

Note that all of the above queries access a significant fraction of the data in the fact table. By contrast, the OLTP query to the database at your local supermarket *How many cans of tomato juice are in stock*? accesses only a single tuple.

Slicing and dicing. We can imagine that the hierarchy for each dimension partitions the multidimensional cube into subcubes. Thus, for example, the `Quarter`

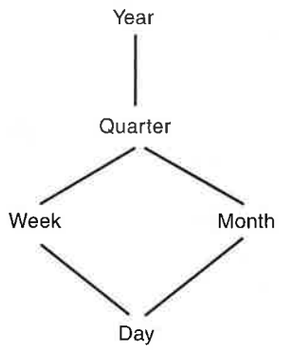

FIGURE 17.9 Time hierarchy as a lattice.

level of the time dimension partitions the cube into subcubes, one for each quarter. Queries that return information about those subcubes are said to **slice and dice**.

- When we pivot, that is, use a GROUP BY clause in a query to specify a level in a hierarchy, we are partitioning the multidimensional cube into subcubes: all the elements in the contained level are grouped together. For example, if we group by product Id and quarter, as in the query (17.3), all transactions for the same product in the same quarter are grouped together. Thus, pivoting creates the effect of **dicing** the data cube into subcubes.
- When we use a WHERE clause that equates a dimension attribute to a constant, we are specifying a particular value for that dimension and so are performing a **slice**.

Typically pivoting and slicing are used together, so it has become known as "slicing and dicing." For example, a query that requests the total product sales in each market in the first quarter

```
SELECT      S.Product_Id, SUM(S.Sales_Amt)
FROM        SALES S, TIME T
WHERE       T.Time_Id = S.Time_Id AND T.Quarter = 'First'
GROUP BY    S.Product_Id
```

is an example of a slice in the time dimension and dice in the product dimension. Thus, pivoting "dices" (partitions) the cube into subcubes along the specified dimensions and slicing selects a cross section that cuts across these subcubes.

17.3.2 The CUBE Operator

Many OLAP queries use the aggregate functions and the GROUP BY clause of the SELECT statement to perform aggregation. However, the standard options for the SELECT statement limit the types of OLAP queries that can be easily formulated in SQL. A number of OLAP vendors (as well as SQL:1999/2003) extend SQL with additional aggregate functions, and some vendors even allow programmers to specify their own aggregate functions.

One extension in this direction is the ROLLUP operator; another is the CUBE operator introduced in [Gray et al. 1997]. Suppose that we want to obtain a table such as that in Figure 17.10. This table is similar to the one in Figure 17.6 except that, in addition, it has totals for each row and each column. To construct such a table, we would need to use four standard SQL SELECT statements to retrieve the necessary information. The following statement returns the data needed for the table entries (without the totals).

```
SELECT      S.Market_Id, S.Product_Id, SUM(S.Sales_Amt)
FROM        SALES S
GROUP BY    S.Market_Id, S.Product_Id
```

SUM(Sales_Amt)		Market_Id			
		M1	M2	M3	Total
Product_Id	P1	3003	1503	15003	19509
	P2	6003	2402	24003	32408
	P3	4503	3	33	4539
	P4	7503	7000	9903	24406
	Total	21012	10908	48942	80862

FIGURE 17.10 Query result for the sales application in the form of a spreadsheet.

The next statement computes the row totals:

```
SELECT    S.Product_Id, SUM(S.Sales_Amt)
FROM      SALES S
GROUP BY  S.Product_Id
```

This statement computes the totals for the columns:

```
SELECT    S.Market_Id, SUM(S.Sales_Amt)
FROM      SALES S
GROUP BY  S.Market_Id
```

and the last statement computes the grand total of 80862 in the lower right corner:

```
SELECT    SUM(S.Sales_Amt)
FROM      SALES S
```

Four statements are required because the table needs four aggregations—by time, by product Id and time, by market Id and time, and by all attributes together. Each such aggregation is produced by a different GROUP BY clause.

Computing all of these queries *independently* is wasteful of both time and computing resources. The first query does much of the work needed for the other three queries, so, if we save the result and then use it to aggregate over Market_Id and Product_Id, we can compute the second and third queries more efficiently. Efficient computation of such "data cubes" is important in OLAP, and much research has been dedicated to this issue. See, for example, [Agrawal et al. 1996; Harinarayan et al. 1996; Ross and Srivastava 1997; Zhao et al. 1998].

Economy of scale is the main motivation for the CUBE clause [Gray et al. 1997], which is included in the SQL:1999/2003 standard. When CUBE is used in a GROUP BY clause,


```
GROUP BY CUBE(v1, v2, . . . , vn)
```

it is equivalent to a collection of GROUP BYs, one for each of the $2^n - 1$ non-empty subsets of $v1, v2, \ldots, vn$, plus the query that does not have the GROUP BY clause, which corresponds to the empty subset. For example, the statement

```
SELECT    S.Market_Id, S.Product_Id, SUM(S.Sales_Amt)
FROM      SALES S
GROUP BY CUBE(S.Market_Id, S.Product_Id)
```

returns the result set of Figure 17.11, which is equivalent to all four of the above SELECT statements and includes all of the values required for the table of Figure 17.10. Note the NULL entries in the columns that are being aggregated. For example, the

RESULT SET	Market_Id	Product_Id	Sales_Amt
	M1	P1	3003
	M1	P2	6003
	M1	P3	4503
	M1	P4	7503
	M2	P1	1503
	M2	P2	2402
	M2	P3	3
	M2	P4	7000
	M3	P1	15003
	M3	P2	24003
	M3	P3	33
	M3	P4	9903
	M1	NULL	21012
	M2	NULL	10908
	M3	NULL	48942
	NULL	P1	19509
	NULL	P2	32408
	NULL	P3	4539
	NULL	P4	24406
	NULL	NULL	80862

FIGURE 17.11 Result set returned with the CUBE operator.

first NULL in the Product_Id column means that the sales for market M1 are being aggregated over all products.[1]

ROLLUP is similar to CUBE except that instead of aggregating all subsets of its arguments, it creates subsets by moving from right to left. Like CUBE, the ROLLUP option to the GROUP BY clause is included in SQL:1999/2003.

Consider the above SELECT statement in which CUBE has been replaced with ROLLUP:

```
SELECT   S.Market_Id, S.Product_Id, SUM(S.Sales_Amt)
FROM     SALES S                                          17.4
GROUP BY ROLLUP(S.Market_Id, S.Product_Id)
```

The syntax here says that aggregation should be computed first with the finest granularity, using GROUP BY S.Market_Id, S.Product_Id, and then with the next level of granularity, using GROUP BY S.Market_Id. Finally, the grand total is computed, which corresponds to the empty GROUP BY clause. The result set is depicted in Figure 17.12. In a larger example (with more attributes in the ROLLUP clause), the result set table would contain some rows with NULL in the last column, some with NULL in the last two columns, some with NULL in the last three columns, and so on.

Note that the ROLLUP operator of OLAP-extended SQL is a generalization of the idea of rolling up the aggregation hierarchy described on page 718. Moreover, the cost savings from reusing the results of fine-grained aggregations to compute coarser levels of aggregation apply to the general ROLLUP operator. For instance, in query (17.4), aggregations computed for the clause GROUP BY S.Market_Id, S.Product_Id can be reused in the computation of aggregates for the clause GROUP BY S.Market_Id. These aggregates can then be reused in the computation of the grand total.

Materialized views using the CUBE operator. The CUBE operator can be used to precompute aggregations on all dimensions of a fact table and then save them for use in future queries. Thus, the statement

```
SELECT   S.Market_Id, S.Product_Id, SUM(S.Sales_Amt)
FROM     SALES S
GROUP BY CUBE(S.Market_Id, S.Product_Id, S.Time_Id)
```

produces a result set that is the table of Figure 17.1 with the addition of rows corresponding to aggregations on all subsets of the dimensions. The *additional* rows

[1] The use of SQL NULL in this context might be confusing because in this context NULL actually means "all" (it is the aggregation of one of the dimensions).

RESULT SET	Market_ Id	Product_Id	Sales_Amt
	M1	P1	3003
	M1	P2	6003
	M1	P3	4503
	M1	P4	7503
	M2	P1	1503
	M2	P2	2402
	M2	P3	3
	M2	P4	7000
	M3	P1	15003
	M3	P2	24003
	M3	P3	33
	M3	P4	9903
	M1	NULL	21012
	M2	NULL	10908
	M3	NULL	48092
	NULL	NULL	80862

FIGURE 17.12 Result set returned with the ROLLUP operator.

are shown in Figure 17.13. If this result set is saved as a materialized view (see Section 5.2.9), it can speed up subsequent queries.

Several materialized views can be prepared (with or without the CUBE operator) and used to speed up queries throughout the entire OLAP application. Of course, each such materialized view requires additional storage space, so there is some limit on the number of materialized views that can be constructed. Since updates are infrequent, the view update problem is not an issue.

17.4 ROLAP and MOLAP

We have been assuming that the OLAP data is stored in a relational database as one (or more) star schemas. Such an implementation is referred to as **relational OLAP** (ROLAP).

Some vendors provide OLAP servers that implement the fact table as a **data cube** using some sort of multidimensional (nonrelational) implementation, often with a substantial amount of precomputed aggregation. Such implementations are referred to as **multidimensional OLAP** (MOLAP). Note that in ROLAP implementations, a data cube is a way to think about the data; in MOLAP implementations, the data is actually stored in some representation of a data cube.

RESULT SET	Market_Id	Product_Id	Time_Id	Sales_Amt
	. . .	. . .	. . .	. . .
	NULL	P1	T1	6500
	NULL	P2	T1	10800
	NULL	P3	T1	1510
	NULL	P4	T1	9133
	NULL	P1	T2	6503
	NULL	P2	T2	10803
	NULL	P3	T2	1513
	NULL	P4	T2	9136
	NULL	P1	T3	6506
	NULL	P2	T3	10806
	NULL	P3	T3	1516
	NULL	P4	T3	6137
	M1	NULL	T1	7000
	M2	NULL	T1	4633
	M3	NULL	T1	4610
	M1	NULL	T2	7004
	M2	NULL	T2	4634
	M3	NULL	T2	16314
	M1	NULL	T3	7008
	M2	NULL	T3	1639
	M3	NULL	T3	16318
	M1	P1	NULL	3003
	M1	P2	NULL	6003
	M1	P3	NULL	4503
	M1	P4	NULL	7503
	M2	P1	NULL	1503
	M2	P2	NULL	2402
	M2	P3	NULL	3
	M2	P4	NULL	7000
	M3	P1	NULL	15003
	M3	P2	NULL	24003
	M3	P3	NULL	33
	M3	P4	NULL	9903
	NULL	NULL	T1	16243
	NULL	NULL	T2	27952

FIGURE 17.13 Tuples added to the fact table by the CUBE operator.

RESULT SET	Market_Id	Product_Id	Time_Id	Sales_Amt
	NULL	NULL	T3	24967
	NULL	P1	NULL	19509
	NULL	P2	NULL	32408
	NULL	P3	NULL	4539
	NULL	P4	NULL	24406
	M1	NULL	NULL	31012
	M2	NULL	NULL	10908
	M3	NULL	NULL	48942
	NULL	NULL	NULL	80862

FIGURE 17.13 (continued)

One use of the CUBE operator is to compute the aggregations needed to load a MOLAP database from an SQL database. Many MOLAP systems also allow the user to specify certain other aggregations that are to be stored as materialized views. Their databases provide efficient implementations of certain (perhaps nonrelational) operations often used in OLAP, such as aggregations at different levels of a hierarchy.

There is no standard query language for MOLAP implementations, but a number of MOLAP (and ROLAP) vendors provide proprietary, sometimes visual, languages that allow technically unsophisticated users to compute tables such as that in Figure 17.10 with a single query, and then to pivot, drill down, or roll up on any table dimension, sometimes with a single click of a mouse.

Not all commercial decision support applications use ROLAP (with star schemas) or MOLAP database servers. Many use conventional relational databases with schemas designed for their particular application. For example, several complex SQL queries used throughout this book can be viewed as OLAP queries (e.g., *List all professors who have taught all courses* . . .). Indeed, any query that uses complicated joins or nested SELECT statements is probably useful only for analysis since its execution time is too long for an OLTP system.

17.5 Implementation Issues

Most of the specialized implementation techniques for OLAP systems are derived from the key technical characteristic of OLAP applications:

> **OLAP applications deal with very large amounts of data, but that data is relatively static and updates are infrequent.**

Moreover, many of these techniques involve precomputing partial results or indices, which makes them particularly appropriate when queries are known in advance—for example, when they are embedded into an operational OLAP application. They can also be used for ad hoc (nonprogrammed) queries if the database designer or administrator has some idea as to what those queries will be.

One technique is to precompute some often-used aggregations and store them in the database. These include aggregations over some of the dimension hierarchies. Since the data does not change often, the overhead of maintaining the aggregation values is small.

Another technique is to use indices particularly oriented toward the queries that will be made. Since data updates are infrequent, the usual overhead of index maintenance is minimal. Two examples of such indices are *join* and *bitmap* indices.

Star joins and join indices. A join of the relations in a star schema, called a **star join**, can be optimized using a special index structure, called a **join index**, as discussed in Section 9.7.2. All recent releases of major commercial DBMSs are capable of recognizing and optimizing star joins.

Bitmap indices. Bitmap indices, introduced in Section 9.7.1, are particularly useful for indexing attributes that can take only a small number of values. Such attributes occur frequently in OLAP applications. For example, `Region` in the MARKET table might take only four values: `North`, `South`, `East`, and `West`. If the MARKET table has a total of 10,000 rows, a bitmap index on `Region` contains four bit vectors, with a total storage requirement of 40,000 bits or 5K bytes. An index of this size can easily fit in main memory and can provide quick access to records with corresponding values.

17.6 Populating a Data Warehouse

Data for both OLAP and data mining is usually stored in a special database often called a **data warehouse**. Data warehouses are usually very large, perhaps containing terabytes of data that have been gathered at different times from a number of sources, including databases from different vendors and with different schemas. Merging such data into a single OLAP database is not trivial. Additional problems arise when that data has to be periodically updated.

Two important operations must be performed on the data before it can be loaded into the warehouse.

1. *Transformation*. The data from the different source DBMSs must be transformed, both syntactically and semantically, into the common format required by the warehouse.
   (a) *Syntactic transformation*. The syntax used by the different DBMSs to represent the same data might be different. For example, the schema in one DBMS might represent Social Security numbers with the attribute `SSN` while another might use `SSnum`. One might represent it as a character string, another as an integer.
   (b) *Semantic transformation*. The semantics used by the different DBMSs to represent the same data might be different. For example, the warehouse might summarize sales on a daily basis, while one DBMS summarizes them on an hourly basis and another does not summarize sales at all but merely provides information about individual transactions.
2. *Data cleaning*. The data must be examined to correct any errors and missing information. We might think that data obtained from an OLTP database should be correct, but experience indicates otherwise. Moreover, some erroneous data might have been obtained from sources other than an OLTP database—for example, an incorrect zip code on a form filled in on the Internet.

Often the term "data cleaning" is used to describe both types of operations. Although there is no general design theory for performing these operations, a number of vendors supply tools that do a decent job for concrete domains such as postal addresses, product descriptions, and the like.

If no data cleaning is necessary and the sources are relational databases that have schemas sufficiently similar to that of the warehouse, the data can sometimes be extracted from the sources and inserted into the warehouse with a single SQL statement. For example, assume that each store, M, in the supermarket chain has an M_SALES table with schema M_SALES(`Product_Id, Time_Id, Sales_Amt`), which records M's sales of each product for each time period. Then, after time period `T4`, we can update the fact table (Figure 17.1) stored in the data warehouse with the sales information for market M in time period `T4` with the statement

```
INSERT INTO SALES(Market_Id, Product_Id, Time_Id, Sales_Amt)
    SELECT Market_Id = 'M', S.Product_Id, S.Time_Id, S.Sales_Amt
    FROM M_SALES  S
    WHERE S.Time_Id = 'T4'
```

If data cleaning or reformatting is needed, the data to be extracted can be represented as nonmaterialized views over the source databases. A cleansing program can then retrieve the data through the views (without requiring knowledge of the individual database schemas) for further processing before inserting it into the warehouse database.

As with other types of databases, an OLAP database must include a **metadata repository** containing information about the physical and logical organization of the data, including its schema, indices, and the like. For data warehouses, the repository must also include information about the source of all data and the dates on which it was loaded and refreshed.

The large volume of data in an OLAP database makes loading and updating a significant task. For the sake of efficiency, updating is usually incremental. Different parts of the database are updated at different times. Unfortunately, however, incremental updating might leave the database in an inconsistent state. It might

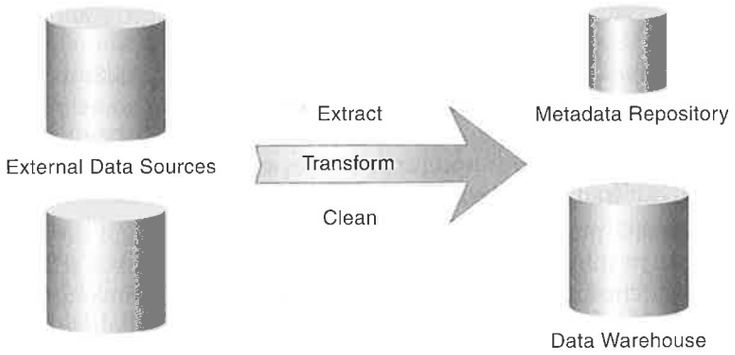

FIGURE 17.14 Loading data into an OLAP database.

not satisfy certain integrity constraints or it might not exactly describe the current state of the enterprise at the current instant. Usually, this is not an important issue for OLAP queries because much of the data analysis involves only summaries and statistical analyses, which are not significantly affected by such inconsistencies.

Figure 17.14 summarizes the processes involved in loading an OLAP database.

17.7 Data Mining Tasks

Data mining is an attempt at knowledge discovery—searching for patterns and structure in large data sets, as contrasted with requesting specific information. If OLAP is about confirming the known, we might say that data mining is about exploring the unknown.

Data mining uses techniques from many disciplines, such as statistical analysis, pattern recognition, machine learning, and artificial intelligence. Our main interest is in understanding these techniques and how they are used to process large data sets. Data mining with its associated data warehousing, data displaying, etc. is sometimes called *Knowledge Discovery in Databases* (KDD).

Among the goals of data mining are

- *Association.* Finding patterns in data that associate instances of that data with other related instances of that data. For example, *Amazon.com* associates the information about books purchased by its customers so that if a particular customer purchases some book, it then suggests other books that this customer might also want to purchase.
- *Classification.* Finding patterns in data that can be used to classify that data (and possibly the people it describes) into certain interesting categories. For example, a company might classify its customers based on their past purchases as either "high-end buyers" or "low-end buyers." This information might then be used to target specific advertisements to those customers.

One important application of classification is for *prediction*. For example, a bank might gather data about the customers who did or did not default on their mortgages over the past five years: their net worth, their income, their marital status, etc. and use that data to classify each customer as a *defaulter* or a *nondefaulter* based on this data. When new customers apply for a mortgage, the bank might use the data on their net worth, income, and marital status to predict whether or not they would default on their mortgage if the application were approved.

- *Clustering.* As with classification, clustering involves finding patterns in data that can be used to classify that data (and possibly the people it describes) into certain interesting categories. However, in contrast with classification, in which the categories are specified by the analyst, in clustering, the categories are discovered by the clustering algorithm.

17.8 Mining Associations

One of the more important applications of data mining is finding associations. An **association** is a correlation between certain values in the database. We gave an example of such a correlation in Section 1.4:

> *In a convenience store in the early evening, a high percentage of customers who bought diapers also bought beer.*

This association can be described using the **association rule**

$$Purchase\_diapers \Rightarrow Purchase\_beer \qquad \textbf{17.5}$$

An association can involve more than two items. For example, it might assert that, if a customer buys cream cheese and lox, she is likely to also buy bagels (if the customer buys only cream cheese, she might be planning to use it for a different purpose and therefore not buy bagels).

$$Purchase\_creamcheese \text{ AND } Purchase\_lox \Rightarrow Purchase\_bagels$$

To see how the association (17.5) might have been discovered, assume that the convenience store maintains a PURCHASES table, shown in Figure 17.15, which it computes from its OLTP system. Based on this table, the data mining system can compute two measures:

1. *The* **confidence** *for an association.* The percentage of transactions that contain the items on the right side of the association among the transactions that contain the items on the left side of the association. The first three transactions of Figure 17.15 contain diapers, and of these, the first two also contain beer. Hence the confidence for association (17.5) is 66.66%.

PURCHASES	Transaction_Id	Product
	001	diapers
	001	beer
	001	popcorn
	001	bread
	002	diapers
	002	cheese
	002	soda
	002	beer
	002	juice
	003	diapers
	003	cold cuts
	003	cookies
	003	napkins
	004	cereal
	004	beer
	004	cold cuts

FIGURE 17.15 PURCHASES table used for data mining.

2. *The* **support** *for an association.* The percentage of transactions that contain all items (on both the left and right sides) of the association. Two of the four transactions in Figure 17.15 contain both items. Hence the support for association (17.5) is 50%. We also define the support for a single item as the percentage of transactions that contain that item.

The purpose of the confidence factor is to certify that there is certain probability that if a transaction includes all items on the left side of the association—*Purchase_diapers* in (17.5)—then the item on the right side will appear as well—*Purchase_beer*. If the confidence factor is high enough, the convenience store manager might want to put a beer display at the end of the diaper aisle.

However, confidence alone might not provide reliable information. We need to make sure that the correlation it represents is statistically significant. For instance, the confidence for the association *Purchase_cookies* $\Rightarrow$ *Purchase_napkins* is 100%, but there is only one transaction where napkins and cookies are involved, so this association is most likely not statistically significant. The support of an association deals with this issue by measuring the fraction of transactions in which the association is actually demonstrated.

To assert that the association exists, both of the above measures must be above a certain threshold. Selecting appropriate thresholds is part of the discipline of statistical analysis and is beyond the scope of this book.

It is relatively easy for the system to compute the support and confidence for a particular association. That is an OLAP query. However, it is much more difficult for the system to return all possible associations for which the confidence and support are above a certain threshold. That is a data mining query. The idea of mining for association rules and some early algorithms were first introduced in [Agrawal et al. 1993].

We present an efficient algorithm for retrieving the data needed to determine all associations for which the support is larger than a given threshold, T. As we will see, once we have found those associations it is easy to determine which of them has a confidence factor greater than some given threshold.

Assume that we are trying to find all associations $A \Rightarrow B$ for which the support is greater than T. The naive approach is to compute the support for $A_i \Rightarrow B_j$ for all pairs of distinct items, A_i and B_j. However, if there are n items, $n(n-1)$ pairs have to be tried. This is usually too costly. The situation is even more difficult if we are interested in associations in which the left side contains more than one item, for example A AND $C \Rightarrow B$. Now we would have to compute the support for all triples of items, A_i, B_i, and C_i. It would be still more difficult if we are interested in associations with *any* number of items on the left side.

We call such sets of items (A_i, B_i, . . .) **itemsets.** Our goal is to find all itemsets for which the support is greater than T. The plan is to first find all single items for which the support is greater than T, then use that information to find all pairs of items with support greater than T, and so on. We consider only the case of associations among two items, but the same ideas generalize to associations with more than one item on the left side

The algorithm we use, called the **a priori algorithm**, is based on the following observation, which follows from the fact that if A and B appear together in R rows, then A and B each appear in at least R rows—and perhaps even more.

> *If the support for an association* A $\Rightarrow$ B *(or an itemset* A, B*) is larger than* T*, then the support for both* A *and* B *separately must be larger than* T.

Based on this observation, the a priori algorithm for pairs of items can be described as follows:

1. *Find all individual items whose support is greater than* T. This requires examining n items. The number of items with high enough support, m, is likely to be much less than n.
2. *Among these* m *items, find all distinct pairs of items whose support is greater than* T. This requires examining $m * (m-1)$ pairs of items. Assume the number of such pairs with high enough support is p.
3. *Compute the confidence factor for these* p *associations.*

If we want to find associations with more than two items, we can use the same ideas. First find single items that have support greater then T, then pairs of items, then triples, and so on.

17.9 Classification and Prediction Using Decision Trees

Classification involves finding patterns in data items that can be used to place those items in specific categories. That categorization can then be used to predict future outcomes.

For example, a bank might gather data from the application forms of past customers who applied for a mortgage and then, based on whether or not those customers later defaulted on their mortgage payments, classify those customers as *defaulters* or *nondefaulters*. When new customers apply for a mortgage, the bank might use the information on their application forms to predict whether or not they might default on their mortgage.

As a simple example, suppose the bank used only three types of information: whether or not the applicant was married, whether or not the applicant had ever defaulted on another mortgage, and the applicant's income at the time of the mortgage application. The information about the past customers would then be stored in a table such as Figure 17.16. The column labels `Married`, `PrevDefault`, and `Income` are called the *attributes* of the table, and the column label `Default` is called the *outcome*. Of course the actual table would have many more attributes and many more rows for different customers, but for the purpose of this example, we assume the table has only these three attributes and these twenty rows.

The bank's goal is to use the information in this table to classify their customers as to whether or not they are *defaulters*. One approach to performing this classification is to make a *decision tree*, such as the one in Figure 17.17.

Classification rules. A decision tree implies a number of **classification rules**. Each rule corresponds to a path from the root of the tree to one of its leaves. Two of the rules for the tree of Figure 17.17 are

```
((PrevDefault = yes) AND (Married = yes)) ⇒ (Default = no)

((PrevDefault = no) AND (Married = yes) AND (Income < 30))
        ⇒ (Default = yes)
```

You might have observed that the entries for `Income` in the table are given as numbers, but the entries for `Income` in the decision tree are for ranges. We will discuss later how we determine appropriate ranges into which to put the `Income` data.

When a table such as Figure 17.16 is used to make a decision tree, the table is called a **training set** because the data in that table is being used to "train" the system as to whether future applicants for a mortgage should be accepted.

CUSTOMER	Id	Married	PrevDefault	Income	Default
	C1	yes	no	50	no
	C2	yes	no	100	no
	C3	no	yes	135	yes
	C4	yes	no	125	no
	C5	yes	no	50	no
	C6	no	no	30	no
	C7	yes	yes	10	no
	C8	yes	no	10	yes
	C9	yes	no	75	no
	C10	yes	yes	45	no
	C11	yes	no	60	yes
	C12	no	yes	125	yes
	C13	yes	yes	20	no
	C14	no	no	15	no
	C15	no	no	60	no
	C16	yes	no	15	yes
	C17	yes	no	35	no
	C18	no	yes	160	yes
	C19	yes	no	40	no
	C20	yes	no	30	no

FIGURE 17.16 The table used to gather information about whether or not the bank's past customers defaulted on their mortgage payments. This table can be used as a training set to make a decision tree.

When a decision tree is developed based on such a table, some of the classification rules derived from the decision tree might not be consistent with the information in the table. For example, the decision tree in Figure 17.16 makes one "mistake." It predicts that

```
((PrevDefault = no) AND (Married = yes) AND (Income ≥ 30))
    ⇒ Default = no)
```

but customer C11 is not correctly classified by that rule—she defaulted. It is naive to expect that the behavior of all the bank's many customers over the past several years could be described by a simple decision tree or a small set of classification rules. The idea is to produce the best possible decision tree (and associated classification rules)

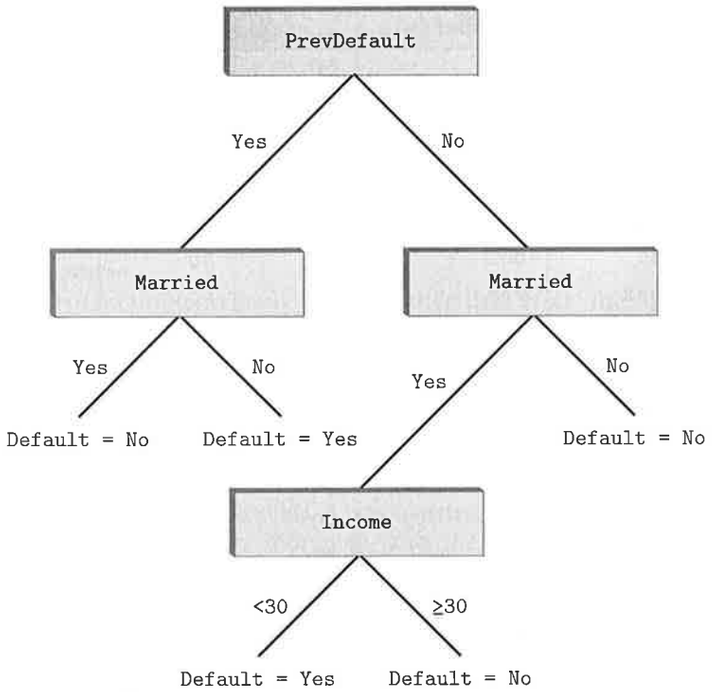

FIGURE 17.17 A decision tree for predicting defaults on a mortgage.

from the training set so that those rules can be used to predict, hopefully with a high success rate, the future behavior of new customers.

One measure of the quality of the decision tree is the percentage of errors that occur when the tree is used to make decisions using all the entries in the training set table, the so-called **training-set error**. Often, some of the historical entries in such a table are not used as part of the training set but are used as a **test set**. After the decision tree has been constructed, the data in the test set is used to test the tree. The percentage of errors using these entries is called the **test-set error**.

Sometimes the test-set error is significantly larger than the training-set error. One reason for this might be that the classification rules implied by the resulting decision tree **overfit** the training data by being tuned to some features peculiar to the particular training set that happened to be selected and not present in the general data. One approach to dealing with such overfitting is to ignore some of the data in the training set—particularly the data that provides the least information about the decision (see the discussion in the next few paragraphs). This can result in the decision tree being **pruned**, that is, all the nodes below a given node (or nodes) are removed, and the best possible decision is made at each such node, again based on the information provided by the remaining data in the training set. We discuss

how such pruning might be done later in this section. As we shall see, frequently this pruning is accomplished with still another subset of the historical entries in the table, called the **validation set**, which is disjoint from both the training set and the test set.

Induction on decision trees. A number of algorithms have been developed for producing a decision tree from a training set, and many of these algorithms have been implemented in commercial systems. We discuss one of these algorithms, **ID3** (induction of decision trees) [Quinlan 1986]. The ID3 algorithm is "top down." It starts by selecting an attribute to be used at the top level of the tree to make the first decision and thus produce the second-level nodes in the tree. Each value of that attribute starts a new level of the tree, and the process repeats on each level. A key part of the algorithm is how to pick the attribute on each level that best differentiates the items in the training set.

Intuitively we want to pick the attribute that maximizes the "purity" of the selection (as the devotees of the field would say). A selection is "pure" if it contains mainly instances of one outcome, and it is "impure" if it contains many instances of items that correspond to both outcomes ("yes" and "no" in the example). The measure of purity that is used in the ID3 algorithm is **entropy**, which is defined as

$$-\sum_{i=1}^{n} p_i \log_2 p_i$$

where p_i (the probability that an item has the outcome i) is approximated by the fraction of items that have outcome i. This measure is borrowed from the field of information theory, where it is used to measure randomness or lack of information (the more random a set of data is, the less information it implies). Thus, again intuitively, we can say that we want to select the attribute that gives us the most information about the final decision.

Suppose that half of the entries in a training table have an outcome of "yes" and half have an outcome of "no." Then the entropy for the table would be

$$-(1/2 \log_2 1/2 + 1/2 \log_2 1/2) = 1$$

which is maximally random and corresponds to zero information. By contrast if all the entries have an outcome of "no," the entropy for the table would be

$$-(1 \log_2 1) = 0$$

which is maximally nonrandom and implies a maximum amount of information (that the only decision is "no").

In the table of Figure 17.16, six entries have an outcome of "yes" and fourteen have an outcome of "no," so the entropy of the table is

$$-(6/20 \log_2 6/20 + 14/20 \log_2 14/20) = .881$$

Now we begin to make the decision tree using the table in Figure 17.16 as a training set. Our initial goal is to find what attribute should be used to make the

CUSTOMER	Id	Married	Income	Default
	C3	no	135	yes
	C7	yes	10	no
	C10	yes	45	no
	C12	no	125	yes
	C13	yes	20	no
	C18	no	160	yes

FIGURE 17.18 The table used as the training set at the second level of the tree when `PrevDefault` is "yes."

decision at the top level in the tree. We investigate, one at a time, each of the attributes—`PrevDefault`, `Married`, and `Income`—to see which attribute provides the most information about the outcome.

If the topmost attribute were selected to be `PrevDefault`, the second level of the tree would consist of two nodes, one corresponding to when `PrevDefault` is "yes" and the other to when `PrevDefault` is "no." Each of these nodes would be the topmost node of a subtree of the decision tree. The training sets for these two subtrees would be the two tables: Figure 17.18 (for the subtree where `PrevDefault` is "yes") and Figure 17.19 (for the subtree where `PrevDefault` is "no"). Each of these tables is a subset of the table in Figure 17.16 consisting of those entries where `PrevDefault` has the specified value and with the column for `PrevDefault` omitted.

Now let us investigate how much information would be gained by using `PrevDefault` as the topmost attribute. We start by computing the entropy of each of the resulting subtables. When `PrevDefault` is "yes," four of the six outcomes are "yes" and two are "no," so the entropy of that subtable is

$$-(4/6\ log_2\ 4/6 + 2/6\ log_2\ 2/6) = .918$$

When `PrevDefault` is "no," two of the fourteen outcomes are "yes" and twelve are "no," so the entropy of that subtable is

$$-(2/14\ log_2\ 2/14 + 12/14\ log_2\ 12/14) = .592$$

Since the first subtable has six entries and the second has fourteen entries, the weighted average entropy of the two subtables is

$$(6/20 * .918) + (14/20 * .592) = .690$$

The measure used by the ID3 algorithm to determine which attribute to select is called the **information gain**. The algorithm compares the information gain implied by using each of the attributes and selects the attribute with the largest information gain. The information gain when using some attribute, A, as the topmost node in

CUSTOMER	Id	Married	Income	Default
	C1	yes	50	no
	C2	yes	100	no
	C4	yes	125	no
	C5	yes	50	no
	C6	no	30	no
	C8	yes	10	yes
	C9	yes	75	no
	C11	yes	60	yes
	C14	no	15	no
	C15	no	60	no
	C16	yes	15	yes
	C17	yes	35	no
	C19	yes	40	no
	C20	yes	30	no

FIGURE 17.19 The table used as the training set at the second level of the tree when `PrevDefault` is "no."

the tree constructed using a table, T, is the entropy of T minus the average entropy of the subtables determined by A:

Information gain(A,T) = *entropy*(T) − *average*(*entropy*(T_i)**weight*(T_i))

The average in the computation of the information gain is taken over all subtables T_i of T that are determined by the values of the attribute A. The weight of each subtable is the relative contribution of the rows of that subtable to the pool of rows in T: *weight*(T_i) = $|T_i|/|T|$, where $|\cdot|$ denotes the number of rows in a table. Thus the information gain for the attribute `PrevDefault` is

$$.881 - .690 = .191$$

Note that the attribute that provides the largest information gain is always the attribute whose subtables have the smallest average entropy (since for all calculations of information gain for different attributes, the average entropy corresponding to each attribute is subtracted from the same value of entropy for the complete table).

Now we compute the information gain implied by each of the other attributes. If we repeat the information gain calculation assuming the topmost attribute of the

tree is Married, the information gain would be .056.[2] Clearly PrevDefault provides more information gain and would be a better choice for the topmost attribute.

Now consider the possibility of using Income as the topmost attribute. There is a complication since Income has continuous values, whereas a decision tree makes its decision based on discrete values. Therefore, to use an attribute with continuous values, we have to divide its possible values into discrete ranges. This process is called **discretization**. (Actually this approach to dealing with continuous values is not in ID3 but in its extension C4.5 [Quinlan 1986].)

If we decide to divide Income into two ranges, Income $< X$ and Income $\geq X$, we have two choices. We can just pick *some* value of X based on our intuitive knowledge of the application, or we can try *all* values for X that are mentioned in the table and compute the entropy for each to see if any of those ranges provide more information gain than PrevDefault. The second choice is usually preferable. For example, if we try $X = 50$, the information gain would be .035. In fact, it turns out that no value of X provides more information gain than does PrevDefault.

Thus, based on the information-gain measure, we decide to use PrevDefault as the toplevel attribute of the tree. We therefore label the topmost node in the tree PrevDefault and construct a branch descending from that node for each of the possible values of PrevDefault going to a node at the second level. These second-level nodes are each the topmost node of a subtree. The tables of Figures 17.18 and 17.19 are the training sets for these subtrees.

Now we repeat the entire procedure on each of these subtrees. When we use the information gain measure on each of the training tables for these subtrees to determine the attributes to be used to make the decisions at the second level, either of the following situations might (or might not) happen.

1. The attribute selected might be different for the two tables.
2. The range selected for some attributes with continuous values (such as the Income attribute in the example) might be different for the two tables (and different from that in analysis at the topmost level).

In our example, neither of these situations occur, and the attribute with the largest information gain for both second-level nodes is Married. Thus we label both of the second-level nodes Married and construct branches going to nodes at the third level of the tree. The tables to be used as the training sets for these third-level nodes are shown in Figures 17.20, 17.21, 17.22, and 17.23.

Note that in the tables shown in Figures 17.20, 17.21, and 17.23, all entries have the same outcome and therefore the tables have an entropy of 0. Thus these tables correspond to leaf nodes of the decision tree, each labelled with its unique outcome.

[2] The subtable determined by Married = yes has 11 tuples with Default = no and 3 tuples with Default = yes. Therefore, this table's entropy is 0.7496. The subtable determined by Married = no has 3 tuples with Default = no and 3 tuples with Default = yes. Its entropy is therefore 1. The average weighted by the relative number of tuples in these tables ($14/20 = 0.7$ and $6/20 = 0.3$) is thus $0.7 * 0.7496 + 0.3 * 1 = 0.8247$. Therefore, the information gain is $.881 - .8247 = .0563$.

CUSTOMER	Id	Income	Default
	C7	10	no
	C10	45	no
	C13	20	no

FIGURE 17.20 The table used as the training set at the third level of the tree when PrevDefault is "yes" and Married is "yes."

CUSTOMER	Id	Income	Default
	C3	135	yes
	C12	125	yes
	C18	160	yes

FIGURE 17.21 The table used as the training set at the third level of the tree when PrevDefault is "yes" and Married is "no."

CUSTOMER	Id	Income	Default
	C1	50	no
	C2	100	no
	C4	125	no
	C5	50	no
	C8	10	yes
	C9	75	no
	C11	60	yes
	C16	15	yes
	C17	35	no
	C19	40	no
	C20	30	no

FIGURE 17.22 The table used as the training set at the third level of the tree when PrevDefault is "no" and Married is "yes."

We then repeat the procedure on the remaining table, Figure 17.22. The only remaining attribute is Income. When we investigate all the possible ranges for Income, we find that the maximum information (minimum entropy) is obtained when X is 30. For that value of X, the average entropy is .412. Thus we decide

CUSTOMER	Id	Income	Default
	C6	30	no
	C14	15	no
	C15	60	no

FIGURE 17.23 The table used as the training set at the third level of the tree when `PrevDefault` is "no" and `Married` is "no."

CUSTOMER	Id	Default
	C8	yes
	C16	yes

FIGURE 17.24 The table used as the training set at the fourth level of the tree when `PrevDefault` is "no" and `Married` is "yes" and `Income` is less than 30.

CUSTOMER	Id	Default
	C1	no
	C2	no
	C4	no
	C5	no
	C9	no
	C11	yes
	C17	no
	C19	no
	C20	no

FIGURE 17.25 The table used as the training set at the fourth level of the tree when PrevDefault is "no" and `Married` is "yes" and `Income` is greater than or equal to 30.

to use the ranges, `Income` < 30 and `Income` ≥ 30, and obtain the tables shown in Figures 17.24 and 17.25.

The table shown in Figure 17.24 has an entropy of 0 and corresponds to a leaf node labelled with outcome "yes."

However, the table shown in Figure 17.25 does not have an entropy of 0 because all the outcomes are not the same. However, we have run out of attributes and cannot continue the procedure. Thus we say that this table corresponds to a leaf with outcome "no" (because "no" is the most frequently occurring outcome in this

table). We have to settle for the fact that this leaf does not make the correct decision for customer C11. That completes the design of the decision tree.

Summary of the ID3 algorithm. The ID3 procedure can be described recursively as follows. Given a table, T, to be used as a training set to construct a decision tree d:

1. If T has entropy equal to 0, construct d with only a single node, which is a leaf node labelled with the (only) outcome in T.
2. If T has no attributes, construct d with only a single node, which is a leaf node labelled with the most frequently occurring outcome in T.
3. Otherwise
   (a) Find the attribute, A, in T that provides the maximum information gain. Specifically, find the attribute, A, that divides T into n subtables, $T_1, T_2, \ldots, T_n$ (assuming that A has n possible values, $v_1, \ldots, v_n$), such that the information gain is the largest of all the information gains obtained from all the other attributes in T. Each subtable T_i is $\sigma_{a=v_i}(T)$ with the column corresponding to the attribute A projected out.
   (b) Label the topmost node in d with the name A.
   (c) Construct n branches descending from that node, where each branch corresponds to one of the n values of A and is labelled with that value.
   (d) At the end of each branch, for example, the branch corresponding to the value v_i of A, start a subtree, d_i, and use T_i as its training set.
   (e) Repeat this procedure on each of the subtrees, $d_1, \ldots, d_n$, starting at step 1.

Brain Teaser: The algorithm ID3 is guaranteed to terminate. Why?

If we are concerned about possible overfitting of the data and want to consider the possibility of pruning the decision tree, one approach is to use the validation set, which (as you might recall) is a subset of the historical data that is disjoint from both the training set and the test set. We first construct the complete decision tree using the training set and compute the training-set error. Then we test the tree with the validation set. If the validation-set error is significantly greater than the training-set error, we can assume that some overfitting has taken place. We then make a series of pruned versions of the tree by deleting each of the leaf nodes. We apply the validation set to each of these pruned versions. If we find that the validation set error has been decreased, we assume that some overfitting has taken place. We then continue the pruning process until the new pruned versions do not have less validation-set error. When we are done, we apply the test set to the final version to compute the test-set error.

The information-gain measure is not the only measure than can be used to produce a decision tree from a training set. Two other measures that have been proposed and used in commercial products are

- **Gain ratio** [Quinlan 1986]. The **gain ratio** is defined as follows:

$$Gain\ Ratio = (Information\ Gain)/SplitInfo$$

where information gain is as defined earlier and

$$SplitInfo = -\sum_{i=1}^{n} |T_i| / |T| \ log_2(|T_i| / |T|)$$

where $|T|$ is the number of entries in the table being decomposed by the attribute and $|T_i|$ is the number of entries in the i^{th} table produced by the decomposition. The idea is to normalize the information-gain measure to compensate for the fact that it favors attributes that have a large number of values.

Since the information gain obtained when using `PrevDefault` is .191, and since there are six entries in the table of Figure 17.18 and fourteen entries in the table of Figure 17.19, the gain ratio for the `PrevDefault` attribute is

$$Gain\ Ratio = .191/(6/20\ log_2(6/20) + 14/20\ log_2(14/20)) = .217$$

- **Gini index** [Breiman et al. 1984]. The **Gini index** is defined as

$$Gini = 1 - \sum_{i=1}^{k} p_i^2$$

where p_i is the probability that a tuple in the training set table has outcome i. Thus, if all of the entries in the training table had outcome "no," the Gini index would be 0, and if half of the entries had outcome "yes" and half had outcome "no," the Gini index would be 1/2.

Since the number of "yes" outcomes in the table of Figure 17.16 is six and the number of "no" outcomes is fourteen, the Gini index of that table is

$$Gini = 1 - ((6/20)^2 + (14/20)^2)) = .42$$

Each measure has its advocates and its share of successes in specific applications.

17.10 Classification and Prediction Using Neural Nets

OPTIONAL

One might say that the decision tree algorithm just discussed is a learning algorithm that *learns* how to make predictions based on the data in its training set. The field of machine learning, which is a subfield of artificial intelligence, provides a number of other techniques that are useful in classification and prediction. Suppose that the mortgage lender wants to determine which applicants are likely to default on their mortgage but believes that the classification depends on a larger number of factors than in the previous example and that these factors should be weighted differently.

To see how a bank might use weights in making a decision, assume it wants to consider only two factors: `PrevDefault` and `Married`. Then it might associate a weight w_1 with the predicate `PrevDefault = yes` and a weight w_2 with the predicate `Married = yes`. The bank might then evaluate the expression

$$w_1 * x_1 + w_2 * x_2$$

where x_1 has value 1 if `PrevDefault = yes` is true and 0 otherwise; x_2 is defined similarly using the predicate `Married = yes`. A customer is considered a bad risk if the value of that expression exceeds some threshold, t, that is, if

$$w_1 * x_1 + w_2 * x_2 \geq t.$$

If $w_1 * x_1 + w_2 * x_2 < t$, the customer is considered a good risk. In practice, the lender might want to include a number of other possible factors in this computation. The question is, how should the weights and the threshold be determined?

A technique called **neural nets** allows the lender to use the information in a training set derived from an OLAP database about past customers to "learn" a set of weights that would have predicted their behavior and thus will (hopefully) predict the behavior of new customers. By "learning" we mean that the system uses examples of the characteristics of past customers who did or did not default on their loans to incrementally adjust the weights to give a better prediction of whether or not customers will default.

The above inequality can be viewed as modeling the behavior of a primitive **neuron** (or nerve cell). In general, a neuron can have any number of inputs, $x_1, \ldots, x_n$, and each input has a weight, w_i. The neuron is said to be **activated** if the weighted sum of its inputs, $\Sigma_{i=1}^{n} w_i * x_i$, exceeds or equals some threshold, t, which can be specific to that particular neuron. When a neuron is activated, it **emits** the value 1; otherwise it is said to emit the value 0.

Our discussion of neurons can be simplified if we introduce w_0 so that $w_0 = t$ and rewrite the equation

$$\sum_{i=1}^{n} w_i * x_1 \geq t$$

as

$$\sum_{i=1}^{n} w_i * x_i - w_0 * 1 \geq 0$$

(We can assume there is a new input x_0, which always has a value of -1.) The expression $\sum_{i=1}^{n} w_i * x_i - w_0 * 1$ is sometimes called the **normalized weighted input**. The **activation function** of the neuron is a monotonic function that takes the normalized weighted input, X, and returns a real number, $f(X)$, such that $0 \leq f(X) \leq 1$.

A typical activation function (and the one used above) is a **step function** where $f(X) = 0$, if $X < 0$, and $f(X) = 1$, if $X \geq 0$. The step activation function tells us when the neuron is "active" (emits 1) or "inactive" (emits 0). In general, however, the activation function can be continuous, such as the *sigmoid* function depicted in Figure 17.27, which will be discussed shortly. In such a case the neuron can emit

OPTIONAL

any real number between 0 and 1, and the activation function indicates the "degree of activation" of the neuron.

The perceptron learning algorithm. Based on this notation, we can define a learning algorithm for a single neuron that has the step function activation. This algorithm is sometimes called the **perceptron learning algorithm** because the authors of that algorithm referred to such neuron models as perceptrons.

1. Initially set the values of all the weights and the threshold to some small random number.
2. Apply the inputs corresponding to each item in the training set one at a time to the neuron model. For each input, compute the output of the neuron.
3. If the desired output of the neuron for that input is d and the actual output is y, change each weight, w_i, by Δw_i where

$$\Delta w_i = \eta * x_i * (d - y)$$

(assuming $x_0 = -1$) where η is some small positive number called the **learning rate**. Note that if for this input, the neuron does not make an error (the desired output equals the actual output), no weights are changed. If the neuron emits a value higher than d, then the weight w_i is decreased in order to try to lower the emitted value (observe that the activation function is monotonically growing). If the emitted value is less than d, then w_i is increased in order to raise the emitted value.

4. Continue the training until some termination condition is met. For example, the data in the training set has been used some fixed number of times, the number of errors has stopped decreasing significantly, the weights have stopped changing significantly, or the number of errors reaches some predetermined level.

If the neuron has n inputs, each of which can be 1 or 0, then there are 2^n possible combinations of these inputs. If we assume that the training set includes each of these 2^n combinations of inputs, the perceptron learning algorithm has the property that if the decision can *always* be correctly made by a single neuron, the values of the weights and threshold will converge to correct values after only a bounded number of weight adjustments [Novikoff 1962].

Neural networks: the sigmoid function. In practice the perceptron learning algorithm is not very useful because for most applications the required decisions cannot be made (even approximately) by a single neuron. Therefore, a network of neurons, such as the one depicted in Figure 17.26, is used.

The network shown has three layers: the input layer, the middle or hidden layer, and the output layer. The input layer does not consist of neurons that can adjust their weights. It just gathers the inputs and presents them to the neurons in the middle layer. The neurons in the middle layer make some intermediate decisions and then send those decisions to the neurons in the output layer, which makes the final decisions.

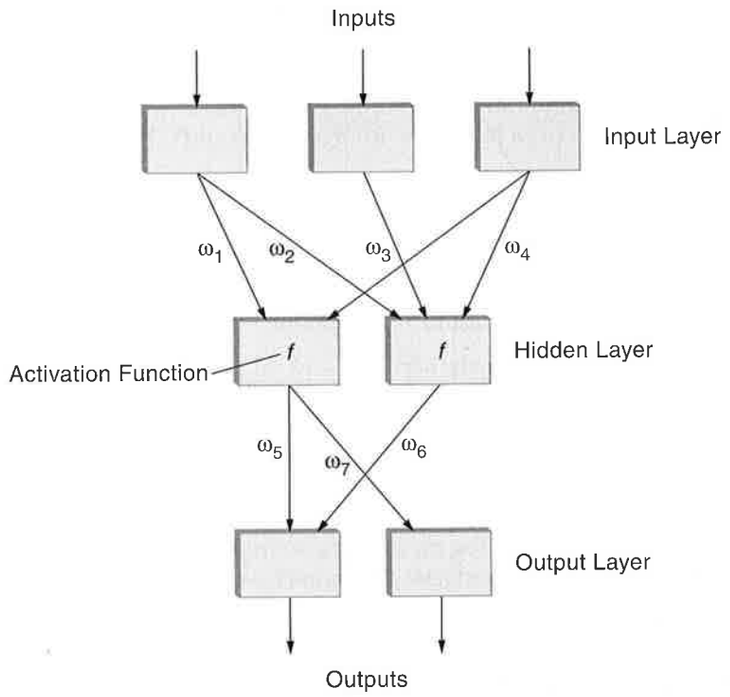

FIGURE 17.26 A neural net.

Learning algorithms for such neural networks are more complex than those for a single neuron because it is not immediately apparent how to adjust the weights of the neurons in the middle layer when one or more of the neurons in the output layer gives an output different than its desired output. More specifically it is not apparent how each such weight in the middle layer affects the output of the network. A mathematical analysis of this situation is difficult because such an analysis usually requires taking derivatives of the activation function, and the step function that we used so far is discontinuous and does not have a derivative. For this reason, neural networks usually use differentiable activation functions.

A commonly used activation function is the **sigmoid function**, shown in Figure 17.27, which is defined as

$$1/(1 + e^{-X})$$

where X denotes the normalized weighted input to the neuron:

$$X = \sum_{i=1}^{n} w_i * x_i - w_0 * 1$$

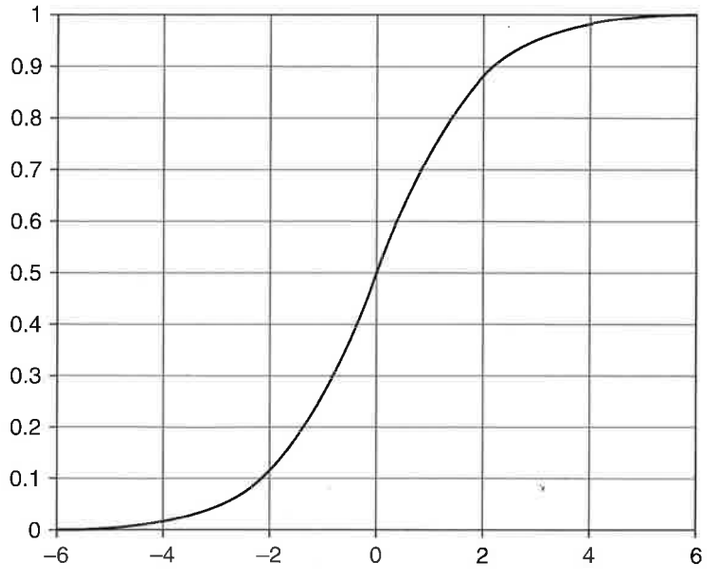

FIGURE 17.27 The sigmoid function.

Note that the value of the sigmoid function is 1/2 for $X = 0$. It becomes asymptotic to 1 for large positive values of X and asymptotic to 0 for large negative values of X. Thus it is a continuous (and differentiable) approximation of the step activation function.

The sigmoid function has an interesting property that we will use later: if the output of the neuron is denoted as y, so that

$$y = 1/(1 + e^{-X})$$

then the partial derivative of y with respect to X is

$$\frac{\partial y}{\partial X} = e^{-X}/(1 + e^{-X})^2 = (1/(1 + e^{-X})) * (1 - (1/(1 + e^{-X}))) = y * (1 - y)$$

and then if we use the definition of X given above, the partial derivative of y with respect to any particular weight, w_i, is

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial X} * \frac{\partial X}{\partial w_i} = y * (1 - y) * x_i \qquad \textbf{17.6}$$

Using this result, we can now derive the learning procedure for a single neuron that uses the sigmoid activation function. Assume that for some input in the training set, the output of the neuron is y and the desired output is d. The error is then $(d - y)$, and the squared error is $(d - y)^2$. (We could also consider the mean squared error for all the inputs, but now we are just considering a single input.) Consider the squared

error as an n-dimensional function of the weights $w_1, \ldots, w_n$. The plan is to take the partial derivative of that function with respect to each of the weights. We will then adjust each weight by some small fraction, η, of the negative of that partial derivative. This is called the **gradient descent** approach to finding the minimum of that function. Thus we change the weight of w_i to $w_i + \Delta w_i$ where

$$\Delta w_i = -\eta \frac{\partial (d - y)^2}{\partial w_i}$$

To see why gradient descent is a good heuristic for finding a local minimum of a function, we need to observe two things:

1. If we change w_i to $w_i + \Delta w_i$, we are moving towards a local minimum of the function $(d - y)^2$. Indeed, if this function is increasing at this value of w_i, the above derivative is positive and so we need to decrease w_i to approach the local minimum. In this case $\Delta w_i < 0$ and $w_i + \Delta w_i$ is an adjustment in the right direction. If the derivative is negative, it means that the function is decreasing, so we need to increase w_i in order to approach the local minimum. In this case $\Delta w_i > 0$ and, again, $w_i + \Delta w_i$ is an adjustment in the right direction. We can imagine that, as we change the weights, we are sliding down a hill.
2. As w_i approaches the local minimum, the value of the derivative decreases to zero and therefore the adjustment step, Δw_i, becomes smaller and smaller. In this way, we avoid overshooting the local minimum by large amounts and the process eventually converges.

The derivative of the square of the error with respect to w_i is

$$\frac{\partial (d - y)^2}{\partial w_i} = -2 * (d - y) * \frac{\partial y}{\partial w_i} = -2 * (d - y) * y * (1 - y) * x_i$$

Thus, to adjust the weights and decrease the error, the learning algorithm needs to change w_i by some fraction of the negative of that derivative.

$$\Delta w_i = \eta * x_i * y * (1 - y) * (d - y) \qquad \textbf{17.7}$$

(Note that we have incorporated the constant 2 that appears in the derivative into the learning rate η. Note also that we have changed the order of the multipliers to what is common in the literature.)

The back-propagation learning algorithms for neural networks. Next we present a learning algorithm for networks of neurons, each described by the sigmoid activation function. Specifically we discuss one of the most popular learning algorithms for such neural networks: the **back-propagation algorithm**. We present the algorithm for three-layer networks, such as that in Figure 17.26, but this algorithm can be adapted for networks with any number of layers.

It is called the back-propagation algorithm because, for each input in the training set, the algorithm first goes forward to compute the output of each neuron in

OPTIONAL

the output layer. Then it goes backward to adjust the weights in each layer one at a time. It initially adjusts the weights of the neurons in the output layer, and then it uses the result of that adjustment to adjust the weights of the neurons in the middle layer (and, if there are more layers, it goes further backward to adjust the weights in those layers one at a time).

1. Initially set the values of all the weights and thresholds of all the neurons in the network to some small random number.
2. Apply the inputs corresponding to each item in the training set one at a time to the network. For each input, compute the output of each neuron in the output layer.
3. Adjust the value of the weights in each neuron in the output layer. Consider one such neuron, v_{out}, and assume it emits y^{out} (for the given input) while the desired output is d^{out}. We can use the same reasoning for v_{out} as we did for the case of a single neuron earlier. Thus we can apply the equation (17.7) to v_{out} and adjust each weight, w_i^{out}, associated with input x_i^{out}, according to the formula

$$\Delta w_i^{out} = \eta * x_i^{out} * y^{out} * (1 - y^{out}) * (d^{out} - y^{out})$$

For reasons that will become clear in the next step, it is convenient to rewrite this formula as

$$\Delta w_i^{out} = \eta * x_i^{out} * \delta^{out}$$

where

$$\delta^{out} = y^{out} * (1 - y^{out}) * (d^{out} - y^{out}) \qquad \textbf{17.8}$$

4. Adjust the value of the weights in each middle-layer neuron. Consider one such neuron—let us denote it v_{mid}. Assume its output for the given input is y^{mid}. A problem is that we do not know what is the *desired* output of v_{mid}. However we do know the desired outputs of the output-layer neurons to which v_{mid} is connected. We are interested in determining how the input weights in the middle-layer neuron v_{mid} affect the output of the outer-layer neurons. As a simple example, suppose our middle-layer neuron is connected to only one output-layer neuron, such as v_{out} above, and the weight of that connection is $w^{mid/out}$. Suppose that for the given training set input, the output neuron v_{out} emits y^{out} while its desired output is d^{out}. Using the same reasoning as before, for each input weight w_i^{mid} of the middle-layer neuron v_{mid} with input x_i^{mid}, we want to adjust w_i^{mid} by some fraction of the negative of the derivative of $(d^{out} - y^{out})^2$ with respect to w_i^{mid}

$$\Delta w_i^{mid} = -\eta * \frac{\partial (d^{out} - y^{out})^2}{\partial w_i^{mid}}$$

Then we note that

$$\frac{\partial (d^{out} - y^{out})^2}{\partial w_i^{mid}} = \frac{\partial (d^{out} - y^{out})^2}{\partial X^{out}} * \frac{\partial X^{out}}{\partial w_i^{mid}}$$

where X^{out} is the X of the output-layer neuron. As before

$$\frac{\partial (d^{out} - y^{out})^2}{\partial X^{out}} = -2 * (d^{out} - y^{out}) * y^{out} * (1 - y^{out})$$

Then we observe that the inputs x_k^{out} of the output neuron v_{out} are the outputs of the middle-layer neurons connected to v_{out}. Therefore, we have

$$X^{out} = \sum_k w_k^{mid/out} * y_k^{mid}$$

where the y_k^{mid} are the outputs of all the middle-layer neurons connected to v_{out}, and the $w_k^{mid/out}$ are the corresponding weights of the connection. Note that since v_{mid} is connected to v_{out}, $w^{mid/out}$ is one of these $w_k^{mid/out}$ and y^{mid} is one of these y_k^{mid}. Returning to our derivatives, we can see that $\frac{\partial y_k^{mid}}{\partial w_i^{mid}} \neq 0$ only when y_k^{mid} is y^{mid}, since w_i^{mid} in an input weight to v_{mid} and thus it affects v_{mid}'s output only. We also know from (17.6) that $\frac{\partial y^{mid}}{\partial w_i^{mid}} = x_i^{mid} * y^{mid} * (1 - y^{mid})$. Therefore we can then write

$$\frac{\partial X^{out}}{\partial w_i^{mid}} = w^{mid/out} * \frac{\partial y^{mid}}{\partial w_i^{mid}} = w^{mid/out} * x_i^{mid} * y^{mid} * (1 - y^{mid})$$

Putting all this together we get

$$\frac{\partial (d^{out} - y^{out})^2}{\partial w_i^{mid}} =$$

$$-2 * (d^{out} - y^{out}) * y^{out} * (1 - y^{out}) * w^{mid/out} * x_i^{mid} * y^{mid} * (1 - y^{mid})$$

Therefore, we can use the following learning rule for our middle-layer neuron v_{mid} connected to the single output-layer neuron v_{out}. For each weight w_i^{mid} of v_{mid} associated with input x_i^{mid}, adjust that weight using the formula

$$\Delta w_i^{mid} = \eta * x_i^{mid} * \delta^{mid}$$

where δ^{mid} is defined as

$$\delta^{mid} = y^{mid} * (1 - y^{mid}) * w^{mid/out} * y^{out} * (1 - y^{out}) * (d^{out} - y^{out})$$

The formula for δ^{mid} can be rewritten as

$$\delta^{mid} = y^{mid} * (1 - y^{mid}) * w^{mid/out} * \delta^{out}$$

where δ^{out} was previously computed for the output neuron v_{out} in (17.8). Therefore, δ^{mid} can be computed from δ^{out}, whence the name back propagation.

If the v_{mid} is connected to several output-layer neurons, we can use the same reasoning based on the negative of the derivative of the sum of the squares of the errors of all the output-layer neurons to which v_{mid} is connected:

$$\Delta w_i^{mid} = -\eta * \frac{\partial \sum_j (d_j^{out} - y_j^{out})^2}{\partial w_i^{mid}}$$

Here y_j^{out} and d_j^{out} are, respectively, the outputs and the desired outputs of all the output-layer neurons that receive input from v_{mid}. The computations are a bit more complex, but the result is that the formula for Δw_i^{mid} is the same as when v_{mid} is connected to a single output-layer neuron except that the formula for δ^{mid} involves the weighted sum of the δ^{out}s of all the output-layer neurons in question:

$$\Delta w_i^{mid} = \eta * x_i^{mid} * \delta^{mid}$$

where

$$\delta^{mid} = y^{mid} * (1 - y^{mid}) * \sum_j (w_j^{wid/out} * \delta_j^{out})$$

5. Continue the training until some termination condition is met. For example, the data in the training set has been used some fixed number of times, the number of errors has stopped decreasing significantly, the weights have stopped changing significantly, or the number of errors reaches some predetermined level.

17.11 Clustering

Suppose we examine the addresses of all the people in the United States who have a certain form of lung cancer. We might find that many of those addresses are clustered in a few areas that are near certain chemical plants. We might then conclude that those chemical plants are somehow involved in causing that cancer.

In a more general situation, suppose we are given a set of data items, each with certain attributes, and a similarity measure based on those attributes. In the above example, the items contain information about cancer patients, the attribute is `Address`, and the similarity measure is *nearness* (as measured by Euclidean distance). **Clustering** involves placing those data items into *clusters* such that the items in each cluster are similar to each other and the items in different clusters are less similar. The clusters are usually disjoint. Thus in the example, we are finding clusters of cancer patients who are similar in that they live near each other.

Note that clustering is different than the classification rules and decision trees that we discussed earlier because in those cases the categories into which the data items are to be classified are known in advance. In clustering, the categories are determined by the clustering algorithm.

An important issue in clustering is the similarity measure that is used. When the attributes are numeric, a similarity measure based on Euclidean distance is often used. Thus if two cancer patients have addresses that can be represented using location coordinates as x_1, y_1 and x_2, y_2, the Euclidean distance between them is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. If the attributes are not numeric, an analyst must develop an appropriate similarity measure. Sometimes this involves converting a nonnumeric attribute into a numeric attribute.

The K-means algorithm. Many algorithms have been proposed for clustering. We first present one of the most popular, the **K-means** algorithm. The inputs to the algorithm are: the dataset of items to be clustered, the desired number of clusters k, and the similarity measure. The algorithms proceeds as follows:

1. Select k of the items at random as the centers of the (initial versions) of the k clusters.
2. Put each item in the dataset into the cluster for which that item is closest to the center, based on the similarity measure. (Initially, when the cluster has only one item, the center is the location of that member.)
3. Recalculate the center of each cluster as the mean of the locations (similarity measures) of all the items in that cluster.
4. Repeat the procedure starting at step 2 until there is no change in the membership in all clusters.

Brain Teaser: The K-means algorithm is guaranteed to eventually terminate. Why?

The final version of the clusters produced by the K-means algorithm is not necessarily unique—there can be several states where equilibrium is achieved. Hence the final version of the clusters might depend on the initial selection of the items in step 1.

As a simple example, consider the table of students' ages and GPAs shown in Figure 17.28. Suppose we are interested in investigating whether older students do better or worse in college than younger ones. As a part of that investigation, we want to cluster these items by age and then see the average GPA in each cluster. Note that the similarity measure (age) is just one-dimensional, so the calculation of distances is particularly easy.

Suppose we want to place the students into two clusters: cluster 1 (younger students) and cluster 2 (older students), so we set k equal to 2. Then suppose in step 1 of the algorithm, we randomly select students `S_1` and `S_4` as the centers of our initial clusters. Thus the (initial) centers are at ages 17 and 20.

STUDENT	Id	Age	GPA
	S_1	17	3.9
	S_2	17	3.5
	S_3	18	3.1
	S_4	20	3.0
	S_5	23	3.5
	S_6	26	3.6

FIGURE 17.28 Table of student ages and GPAs for clustering example.

Then we examine each student row and place it in one of the clusters. For example, student S_2 is at a distance of 0 from cluster 1 and at a distance of 3 from cluster 2, so it is placed in cluster 1. On the other hand, student S_5 is at a distance of 6 from cluster 1 and at a distance of 3 from cluster 2, so it is placed in cluster 2. Thus the initial version of the clusters is

Cluster 1: S_1, S_2, S_3
Cluster 2: S_4, S_5, S_6

The new centers of these clusters are

Cluster 1: $(17 + 17 + 18)/3 = 17.333$
Cluster 2: $(20 + 23 + 26)/3 = 23.0$

Then we recompute in which cluster each student is to be placed. The only interesting computation is for student S_4, who is at a distance of 2.677 from the center of cluster 1 and a distance of 3 from the center of cluster 2 and so is placed in cluster 1 (perhaps counter-intuitively since this student was chosen as the initial center of cluster 2). The other students remain in their original clusters. Thus the second version of the clusters is

Cluster 1: S_1, S_2, S_3, S_4
Cluster 2: S_5, S_6

If we now repeat step 2 of the algorithm, the clusters remain the same, and so the algorithm has completed. The average GPA of the students in each cluster is then:

Cluster 1 (younger students): $(3.9 + 3.5 + 3.1 + 3.0)/4 = 3.375$
Cluster 2 (older students): $(3.5 + 3.6)/2 = 3.55$

Whether or not that is a significant difference is a subject for further analysis.

Since the final value of the clusters might depend on the initial selection of items in step 1, some authors suggest that the algorithm be repeated with different initial selections or that a nonrandom selection be made of items that are far apart. Other authors suggest that a better set of final clusters is often obtained if, in step 2, the items are moved one at a time and the cluster centers are recalculated after each move.

In some applications, it might not be obvious what value to use for k. One approach is to try different values of k and calculate the average distance to the center of each cluster as k increases. Usually the average decreases rapidly until the "correct" value of k has been reached and then decreases more slowly.

The hierarchical algorithm. Another algorithm for clustering, in which the value of k need not be selected in advance, is the **hierarchical** algorithm (sometimes called the **agglomerative** hierarchical algorithm). The algorithm proceeds as follows:

1. Start with each item in the dataset as a separate cluster.
2. Select two clusters to merge into a single cluster. The goal is to pick the two clusters that are "closest." Various measures have been proposed for closeness. One measure, and the one we will use, is that the distance between clusters is the distance between their centers. The center of a cluster is the mean (the numeric average) of the locations of all the items in the cluster. We therefore merge the two clusters for which the centers are closest. (Another measure is that the distance between groups is the distance between the "nearest neighbors," the closest two items in each group.)
3. Repeat step 2 until some termination condition is reached. One condition is that some predetermined number of k clusters has been obtained. Another condition might be to continue as long as the average distance to the center of each cluster is decreasing rapidly and terminate when it begins to decrease more slowly. Still another, as we shall see below, is to continue until there is only one cluster and then analyze the result to select an appropriate set of clusters.

One way to implement this algorithm is with a matrix of all the pairwise distances between the clusters. Initially, the matrix contains the pairwise distances between the individual items in the data set. The matrix is then used in step 2 to determine which clusters to merge. After this determination is made, the matrix is updated by inactivating (or deleting) one of the clusters being merged and updating the information about the other cluster (now representing the new cluster) with the distances between that cluster and the other clusters. If there are n items in the dataset, this implementation requires $O(n^2)$ space and $O(n^3)$ time.

If we use this algorithm on the items in the table of Figure 17.28, we would get the following sequence of clusters. Here we denote each cluster by its age attribute and separate different clusters with space.

17 17 18 20 23 26

17, 17 18 20 23 26

17, 17, 18 20 23 26

17, 17, 18, 20 23 26

17, 17, 18, 20 23, 26

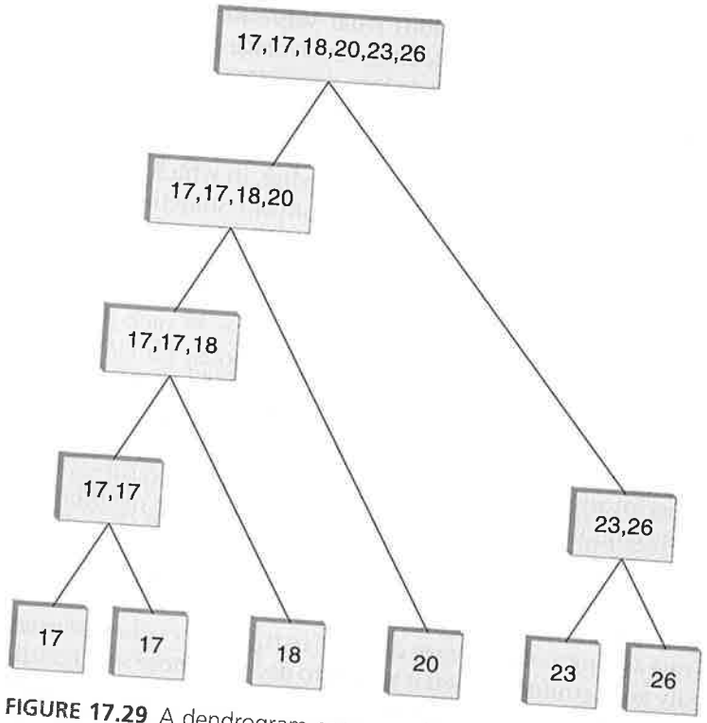

FIGURE 17.29 A dendrogram corresponding to the example of the hierarchical clustering algorithm.

The first row depicts clusters where each item is in a cluster of its own. In the last row we have only two clusters. If we stop at this point, the final set of clusters is the same as with the K-means algorithm. If we continue one more step, we would reduce the set of clusters to a single cluster

17, 17, 18, 20, 23, 26

One way to analyze the results of the hierarchical clustering algorithm is with a tree, usually called a **dendrogram**, that represents the progress of the algorithm, assuming it continues until there is only one cluster. The dendrogram for the above example is shown in Figure 17.29. The cluster generated by each step in the algorithm is denoted by a node in the tree.

Any set of nodes whose children include all the leaves in the tree exactly once represents a possible set of clusters. For example the three nodes denoted as

17, 17, 18 20 23, 26

is a set of clusters that did not appear at any time in the hierarchical algorithm but might be appropriate for certain applications.

The analyst can then analyze such a dendrogram in the light of her knowledge of the application and select a set of clusters that is appropriate for that application.

BIBLIOGRAPHIC NOTES

The term "OLAP" was coined by Codd in [Codd 1995]. A good survey of OLAP appears in [Chaudhuri and Dayal 1997]. A collection of articles on applications and current research in data mining can be found in [Fayyad et al. 1996]. The CUBE operator was introduced in [Gray et al. 1997]. Efficient computation of data cubes is discussed in [Agrawal et al. 1996; Harinarayan et al. 1996; Ross and Srivastava 1997; Zhao et al. 1998]. The idea of mining for association rules and some early algorithms was first introduced in [Agrawal et al. 1993]. The ID3 algorithm for decision trees, including both the information gain and gain ratio measures, was introduced in [Quinlan 1986]. The Gini index was introduced in [Breiman et al. 1984]. A textbook-style coverage of data mining can be found in [Han and Kamber 2001] and [Hand et al. 2001].

EXERCISES

17.1 Is a typical fact table in BCNF? Explain.

17.2 Explain why, in an E-R model of a star schema, the fact table is a relationship and the dimension tables are entities.

17.3 Design another fact table and related dimension tables that a supermarket might want to use for an OLAP application.

17.4 Explain why it is not appropriate to model the database for the Student Registration System as a star schema.

17.5 Design SQL queries for the supermarket example that will return the information needed to make a table similar to that of Figure 17.10, except that markets are aggregated by state, and time is aggregated by months.

a. Use CUBE or ROLLUP operators.
b. Do not use CUBE or ROLLUP operators.
c. Compute the result table.

17.6 a. Design a query for the supermarket example that will return the total sales (over time) for each supermarket.
b. Compute the result table.

17.7 Suppose that an application has four dimension tables, each of which contains 100 rows.

a. Determine the maximum number of rows in the fact table.
b. Suppose that a one-dimension table has an attribute that can take on 10 values. Determine the size in bytes of a bit index on that attribute.

c. Determine the maximum number of tuples in a join index for a join between one of the dimension tables and the fact table.
d. Suppose that we use the CUBE operator on this fact table to perform aggregations on all four dimensions. Determine the number of rows in the resulting table.
e. Suppose that we use the ROLLUP operator on this fact table. Determine the number of rows in the resulting table.

17.8 Design a query evaluation algorithm for the ROLLUP operator. The objective of such an algorithm should be that the results of the previously computed aggregations are *reused* in subsequent aggregations and *not* recomputed from scratch.

17.9 Design a query evaluation algorithm for the CUBE operator that uses the results of the previously computed aggregations to compute new aggregations. (*Hint*: Organize the GROUP BY clauses used in the computation of a data cube into a lattice, that is, a partial order with the least upper bound and the greatest lower bound for each pair of elements. Here is an example of such a partial order: GROUP BY A > GROUP BY A,B > GROUP BY A,B,C and GROUP BY B > GROUP BY B,C > GROUP BY A,B,C. Describe how aggregates computed for the lower parts of the lattice can be used in the computation of the upper parts.)

17.10 Suppose that the fact table of Figure 17.1 has been cubed and the result has been stored as a view, SALES_V1. Design queries against SALES_V1 that will return the tables of Figure 17.10 and Figure 17.7.

17.11 We are interested in building an OLAP application with which we can analyze the grading at our university, where grades are represented as integers from 0 to 4 (4 representing an A). We want to ask questions about average grades for different courses, professors, and departments during different semesters and years. Design a star schema for this application.

17.12 Discuss the difference in storage requirements for a data cube implemented as a multidimensional array and a fact table.

17.13 Give examples, different from those in the text, of syntactic and semantic transformations that might have to be made while loading data into a data warehouse.

17.14 Perform the a priori algorithm on the table of Figure 17.15 to determine all reasonable two-item associations.

17.15 Show that, when evaluating possible associations, the confidence is always larger than the support.

17.16 Apply the gain ratio measure to the table in Figure 17.16 to design a decision tree.

17.17 Apply the Gini measure to the table in Figure 17.16 to design a decision tree.

17.18 Show how the K-mean clustering algorithm would have worked on the table of Figure 17.28 if the original choice for cluster centers had been 17 and 18.

17.19 a. Give an example of a set of three items on a straight line for which the K-mean algorithm, with $k = 2$, would give a different answer for the two clusters depending on the choice of items for the initial clusters.

b. Which of these final clusters would have been obtained by the hierarchical algorithm?

17.20 Suppose the table of Figure 17.16 is stored in a relational database. Use SQL to compute the probabilities needed to compute the information gain when using the `PrevDefault` attribute as the topmost attribute of a decision tree based on that table.