Chapter 4
Introduction to Data Analysis

4.1 What Is Data Analysis?

Data Analysis is the set of tools and techniques used to extract information from data. This information comes in the form of patterns, formulas, or rules that describe properties of the dataset. Because there are many types of information that can be learned from data, data analysis is a vast and complex subject with an abundant bibliography, especially about *Data Mining* and *Machine Learning* [5, 13, 14, 16]. In this chapter, we are going to concentrate on a few, simple methods that can be implemented in SQL without excessive complexity.

Most techniques described apply to tabular data (although there are also specific techniques for text and for graphs, which we also briefly describe). In these techniques, we have a set of records (records, objects, events, observations), each represented by n attributes (observations, features) $x_1, \ldots, x_n$. In the case of *supervised* learning techniques (also called *predictive* techniques), each object also has some additional feature y of interest, usually called the *dependent variable* or *response variable* in Statistics (the *label* in Machine Learning). We want to predict the value of y from the values of $x_1, \ldots, x_n$ (called the *independent, or predictor, variables* in Statistics, and simply the *features* in Machine Learning). We start with a set of records for which the values of y are known; this is called *labeled* data. We will use such values to *train* an algorithm. The supervision here refers to the fact that we supply the algorithm with examples of what we want to know through the labels, so that the algorithm can learn form these. The label can be either categorical (in which case it describes a class out of a finite set of possible classes, and the task is called *classification*) or numerical (normally a real value, in which case the task is called *regression analysis*) [4, 16].

Supervised learning is conventionally used when we have a certain analysis in mind, or a certain hypothesis that we want to examine. Having this means that we have identified a dependent or response variable among all attributes present in the dataset. Once this is done, we start by splitting data into the *training* data and the

A. Badia, *SQL for Data Science*, Data-Centric Systems and Applications,
https://doi.org/10.1007/978-3-030-57592-2_4

test data. The training data, with labels included, is supplied to the algorithm. Once the algorithm is trained, it is run on the test data (with labels withheld). We thus can compare the answers that the algorithm provides on the test data with the labels for such data to determine how well the algorithm is performing. For this to work well, the data should be split randomly, to avoid providing the algorithm with training data that is biased in some way. There are some smart ways to split data, but here we will use a simple method that relies on a random number generator.

Sometimes we do not have any particular attribute to serve as the dependent variable; this is the *unsupervised* learning case (also called *knowledge discovery*). In this case, we want to explore the data and find patterns on it, without much in the way of assumptions. Unsupervised learning can sometimes be used as part of the pre-processing of data, since it can help us learn more about the dataset and its lack of assumptions fits this stage of analysis well. For instance, a very common task is to discover whether the data records we have can be divided into groups based on their similarities to each other. This is called *clustering*, and the groups of similar objects we find are called *clusters*. Other types of unsupervised learning include finding *association rules* and discovering *latent factors* (also called *dimensionality reduction*).

There are also other approaches, like *semi-supervised* learning. There are methods that combine more than one tool, called *ensemble* models.

Here we are going to cover only some basic ideas that can be implemented in SQL with relatively limited effort. Knowing these ideas is quite useful: they can serve as an introduction to more complex approaches. From a practical perspective, it is important to start analysis with simple tools and only use more complex methods once the data (and the problem) are well understood. For many real-life problems, a simple method may provide approximate but useful results.

4.2 Supervised Approaches

In supervised approaches, we have labeled data that we need to split into training data and test data. One good way to do this is to choose randomly; as we have seen (see Sect. 3.4), most systems have a way to generate random numbers that can be used for this. Recall that in Postgres, for instance, the function `random()` generates a random number between 0 and 1 each time it is called (so it generates multiple random numbers, if called several times in the same query), and that MySQL has the exact same function. It is common to devote most data (between 75% and 90%) to training and a small part (between 25% and 10%) to testing. A general procedure to split dataset into training and test would be written (in Postgres or MySQL) as follows:

```
CREATE TABLE new-data
as SELECT *, random() as split
FROM Data;
```

```
CREATE TABLE training-data
as SELECT *
FROM new-data
WHERE split >= .1;

CREATE TABLE testing-data
as SELECT *
FROM new-data
WHERE split < .1;
```

Remember that `random()` simply generates a random value each time it is called; and in the first query above, it is called once for each row in the `Data` table. Thus, the table `new-data` is simply rows of data with a random number between 0 and 1 attached to each row. By changing the constant in the definition of table `training-data` (and adjusting the constant in the definition of `testing-data`), we can split the data as needed.

4.2.1 Classification: Naive Bayes

In classification, we assume that each record has n predictive attributes $(A_1, \ldots, A_n)$ and belongs to one of several classes $C_1, \ldots, C_m$. In the training data, each record has an attribute `Class` giving the class C_i of that record, so the schema of the training data is $(A_1, \ldots, A_n, \texttt{Class})$. We create the training data as shown above and train our algorithm on it. We then take out attribute `Class` from the test data and run our trained algorithm in this data to see which class it assigns to each testing record. We then compare the predicted class to the real, withheld class to see how well our algorithm did.

Classification is one of the most studied problems in Data Science, and there are many algorithms to attack it. One of the simplest is *Naive Bayes*. In spite of its simplicity, it can be surprisingly effective, and in simple situations it can be written in SQL.

Assume we have a table `training-data`$(A_1, \ldots, A_n, \texttt{Class})$, where each A_i is an attribute (feature) that we are going to use for classification. The idea of Naive Bayes is this:

1. For each class C_i, the *a priori probability* of the class, $P(C_i)$, is computed as the percentage of records in the dataset that belong to this class. We can easily compute this for each class; we compute also the raw count of records for each class because it will be useful in the next step:

```
CREATE TABLE classPriors AS
SELECT class, sum(1.0 / total) as classProb,
              count(*) as rawclass
FROM training-data,
     (SELECT count(*) AS total FROM training-data) AS T
GROUP BY class;
```

This yields a table with schema `(class, classProb)`, with one row for each value of `Class`; row (C_i, p) means that class C_i has probability p (p will always be a real number between 0 and 1).

2. For each predictor attribute A_i, class C_j, we compute the conditional probability $P(C_j|A_i = a_i)$ that, if attribute A_i has value a_i, the record is of class C_j. This is done for each value of A_i, so again it is a simple grouping:

```
CREATE TABLE AiPriors
SELECT Ai, class, sum (1.0 / rawclass) AS classProb
FROM training-data, classPriors
WHERE training-data.class = Priors.class
GROUP by Ai, class;
```

This yields a table with schema $(A_i$`,class,classProb`$)$ with tuples (a_i, C_j, p) reflecting that $P(C_j|A_i = a_i) = p$ (as before, $0 \leq p \leq 1$). Note that we divide by the number of records in the class because what matters to us is what is the influence of the fact that attribute A_i has value a_i in the fact of the record being of class C_i; therefore, we need to take all and only the records of class C_i as reference.

A similar table is created for each predictor attribute, so we get a table for A_1, a table for $A_2, \ldots$, and a table for A_n.

3. Given all these prior probabilities, we can now apply our predictor as follows: to predict the probability that a record $r = (a_1, \ldots, a_n)$ in the testing set belongs to class C_i, we use

$$P(C_i|r) = P(C_i|a_1, \ldots, a_n) = \Pi_j P(C_i|A_j = a_j)P(c_i).$$

Note that this assumes that the probability of each single attribute denoting a class is independent of all other attributes, a strong assumption that does not always hold (hence the 'naive' in 'naive Bayes'). In spite of this, the approach works unexpectedly well in many situations, including some with dependencies among the predictors.

This calculation is carried out for each class, that is, in each training record r we get a probability $P(C_i|r)$ for each of $C_1, \ldots, C_m$. We then choose the class with the highest probability as the class of r. The process is repeated for each record in the testing data:

```
CREATE TABLE results as
SELECT test-data.*, classPrior.class,
    classPrior.prior * A1prior.prob * A2prior.prob ...
    as ClassProb
FROM classPriors, A1Prior, ..., AnPrior, test-data
WHERE test-data.A1 = A1Prior.value
      and classPrior.class = A1Prior.class and
      test-data.A2 = A1Prior.value
      and classPrior.class = A2Prior.class and
      ...
GROUP BY test-data.*, classPrior.class;
```

We have used '*' here as a shortcut; it is necessary to enter all attributes of `test-data` to make the query legal. That is, the schema here is `(`A_1`,...,`A_n`,Class, ClassProb)`, with each test record $r = (a_1, \ldots, a_n)$ generating m records of the form $(a_1, \ldots, a_n, C_i, p)$ (one for each C_i), meaning that $P(C_i|r) = p$. This step is done for each record in the testing set and each class; now we chose our final results:

```
SELECT test-data.*, classPrior.class
FROM results R1
WHERE ClassProb = (SELECT max(ClassProb)
                   FROM results R2
                   WHERE R2.test-data.* = R1.test-data.*)
```

Again, the '*' stands for all predictor attributes $A_1,\ldots,A_n$.

Once this is done, the results are compared with the real class of those records; the percentage of correct predictions is a good indication of how good our Naive Bayes classifier is.

Example: Naive Bayes in SQL

Assume a dataset where we have information about several patients and their eyesight issues.[1] In particular, a table `RawData` has schema `(id, age, prescription, astigmatic, tears, lens)`. We are going to predict attribute 'lens' from attributes 'age,' 'prescription,' 'astigmatic,' and 'tears.'

```
%split data into training and testing, as indicated
CREATE TABLE FullData AS
SELECT *, rand() as split
FROM RawData;

CREATE TABLE TrainData AS
SELECT id, age, prescription, astigmatic, tears, lens
FROM FullData
WHERE split >= .01;

CREATE TABLE TestData AS
SELECT id, age, prescription, astigmatic, tears, lens
FROM FullData
WHERE split >= .01;

%calculate class priors
CREATE TABLE Priors AS
SELECT lens, count(*) as rawclass, sum(1.0 /t.total) as prior
FROM TrainData,
     (SELECT count(*) as total FROM TrainData) AS t
GROUP BY lens;

%calculate conditional probabilities per attribute and class.
```

[1] This example is adapted from the one at https://sqldatamine.blogspot.com/.

```
CREATE TABLE AgeCondProbs AS
SELECT age as value, TrainData.lens,
       sum(1.0/rawclass) as condprobs
FROM TrainData, Priors
WHERE TrainData.lens = Priors.lens
GROUP BY age, lens;

CREATE TABLE PrescriptionCondProbs AS
SELECT prescription as value, TrainData.lens,
       sum(1.0 / rawclass) as condprobs
FROM TrainData, Priors
WHERE TrainData.lens = Priors.lens
GROUP BY prescription, lens;

CREATE TABLE AstigCondProbs AS
SELECT astigmatic as value, TrainData.lens,
       sum(1.0/ rawclass) as condprobs
FROM TrainData, Priors
WHERE TrainData.lens = Priors.lens
GROUP BY astigmatic, lens

CREATE TABLE TearsCondProbs AS
SELECT tears as value, TrainData.lens,
       sum(1.0/rawclass) as condprobs
FROM TrainData, Priors
WHERE TrainData.lens = Priors.lens
GROUP BY tears, lens;

%using probabilities for class and attributes,
%compute the class of each record in training data
CREATE TABLE Results as
SELECT id, Priors.lens,
       (A.condprobs * B.condprobs * C.condprobs * D.condprobs
       * Priors.prior)  as classProb
FROM TrainData, Priors
     AgeCondProbs A, PrescriptionCondProbs B,
     AstigCondProbs C, TearsCondProbs D
WHERE TrainData.age = A.value and
      TrainData.prescription = B.value and
      TrainData.astigmatic = C.value and
      TrainData.tears = D.value and
      Priors.lens = A.lens and Priors.lens = B.lens and
      Priors.lens = C.lens and Priors.lens = D.lens
GROUP BY id, Prior.lens;

%we chose the class here as the one with highest probability
CREATE TABLE Eval as
SELECT Results.id, Results.lens
FROM  Results R1
WHERE classProb = (SELECT max(classProb)
                   FROM Result.R2
                   WHERE R1.id = R2.id)

%evaluation of results: compare to real class (ground truth)
```

```
SELECT sum(case when Eval.lens = TestData.lens
           then 1 else 0 end)/count(*)
FROM TestData, Eval
WHERE TestData.id = Eval.id;
```

Exercise 4.1 One of the most common exercises in all of Machine Learning is to apply some simple algorithms (like Naive Bayes) to the *iris dataset*, a very famous (and simple) dataset available in many places on the Internet.[2] Find a copy of the dataset, load it into Postgres or MySQL, and implement the Naive Bayes algorithm over it (what features are the predictive ones and which one is the class to be predicted will be obvious once you learn about the dataset). Pick 90% of the data randomly for training and 10% for validation.

While the implementation above is straightforward, there is a practical issue to consider: when the number of predictive attributes is high, this requires quite a bit of tables (and queries). Note that the priors of the classes can be computed with a single query, regardless of the number of classes; this is due to the fact that what we want (the particular classes) are in the data, while the (predictive) attributes are part of the schema. Therefore, in some approaches the calculated table is 'pivoted' (see Sect. 3.4.1) so that all per-attribute probabilities can be calculated in a single query (albeit a pretty long one) and put in a single table. However, this means that all probabilities will fall under one attribute. In this case, since in SQL there is no multiplication aggregate, we cannot write

```
SELECT attribute, mult(probs)
...
GROUP BY attribute;
```

We instead use the trick described earlier (see Sect. 3.1): we add logs instead of multiplying probabilities and then take the obtained value as an exponential. That is, instead of multiplying numbers $r_1 \times \ldots \times r_n$, we add $log(r_1) + \ldots + log(r_n)$ (which uses the aggregate `sum()`) and then compute e^r (for r the result of the sum), which in most systems is done with function `exp()`. Thus, we write

```
SELECT attribute, exp(sum(log(probs)))
...
GROUP BY attribute;
```

instead of the above.

As stated earlier, we must realize that the calculated value is an approximation due to numerical representation limits; to minimize loss of accuracy, it is important that the value be represented as a real number with the biggest precision available in the system. Even then, very small numbers (as probabilities tend to be, since they

[2] See https://en.wikipedia.org/wiki/Iris_flower_data_set for a description and pointers to several repositories of the data.

are normalized to between 0 and 1) may sometimes render inaccurate results. This is especially the case with large datasets where the class data is sparse (i.e. each class appears infrequently in the data); this makes the class priors to be very small numbers.

Exercise 4.2 Compute the Naive Bayes of the example, but this time create a single table `AttributeCondProbs(attribute, value, lens, condprob)`, where `attribute` is one of the attribute names ("age," "prescription," "astigmatic," "tears"), `value` is the value of the given attribute, `lens` is one of the classes, and `condprob` is the conditional probability that when attribute `attribute` takes value `value` in a record, and the record is of class `lens`. Hint: you can union the individual tables computed above. Another hint: when you do this, you can apply the trick of adding logs to compute the final conditional probability.

We mention an additional feature of Naive Bayes. When the model has many features, it could be that the estimated probability of one (or several) of the features for some class is zero (i.e. there is no record where attribute A_i has a certain value a_i for some class C_j). The problem with this is that it yields a conditional probability of zero; as we have seen above, we are going to multiply probabilities, so if one of them is zero, that will bring the whole product to zero. For this reason, it is common to use a technique called *additive smoothing* or *Laplace smoothing* when calculating probabilities by counting. Without going into the technical details, the basic idea is to add a very small value (usually 1) to each count, while adding a corrective factor (usually n, for n the total number of possible values) to the denominator of the fractions that compute probabilities (otherwise, the probabilities for the values of the class may not add up to 1). Thus, for attribute A_i, we would compute $P(C_j|A_i = a_i)$ as follows:

```
CREATE TABLE AiPriors
SELECT Ai, class,
       sum (1.0 / (rawclass + norm)) + 1 AS classProb
FROM training-data, classPriors,
     (SELECT count(distinct Ai) as norm FROM training-data)
      as T
WHERE training-data.class = Priors.class
GROUP by Ai, class;
```

Recall that the fact that we do not have a multiplicative aggregate in SQL forces us to use sums of logs of probabilities. While the sum has no problem with zeros, the log function is undefined for a value of zero, so this issue does not go away in SQL; thus, this technique may have to be applied in certain problems anyways.

Exercise 4.3 Redo the exercise on the Iris dataset, but this time apply Laplace smoothing. Check the difference between your result here and without the smoothing. Did the algorithm perform better with or without smoothing?

4.2.2 Linear Regression

Regression is similar to classification, in that we have, in the data, independent variables or predictors, and we want to predict values of a dependent variable or response. However, in regression the response is a numerical value. For example, assessing the risk of a borrower for a loan based on attributes like age, occupation, expenses, credit history, etc., is a typical regression task, since we give each borrower a numerical *score* that represents how high/low of a risk it is to loan money to that individual. The predictors themselves can be numerical or categorical; whenever a predictor is categorical, it is transformed into a numerical one by creating a *dummy variable*, as seen in Sect. 3.3.1.1.

The simplest type of regression is *linear* regression, where one attribute (independent variable) A is related to attribute (dependent variable) B with a linear relation

$$B = f(A) = \alpha_0 + \alpha_1 A,$$

where α_1, α_2 are the parameters we need to estimate from the data (traditionally, α_2 is called the *intercept* and α_1 is called the *slope*). With n independent variables, we have a linear relation

$$B = f(A_1, \ldots, A_n) = \alpha_0 + \alpha_1 A_1 + \ldots + \alpha_n A_n$$

and try to estimate $\alpha_0, \ldots, \alpha_n$. We can start with a guess and then work to minimize the error, that is, the difference between predicted and actual value. Since this is supervised learning, on each record r we know $r.B$ (the value of B at r) , which we compare with $f(r.A)$ (the value derived from A at r): the error in the record r is usually taken to be $r.B - f(r.A)$, since we are dealing with numerical values. We want to minimize the total error for the whole dataset; usually, this is expressed as the *sum of square differences*. For the one variable case, this is given by

$$SSE = \Sigma_i (f(r_i.A) - r_i.B)^2 = \Sigma_i ((\alpha_0 + \alpha_1 r_i.A) - r_i.B)^2,$$

where we sum over all records in the dataset. We simply choose the values of α_0, α_1 that minimize this. Because this is a very simple expression, we can determine what are the optimal values for these parameters:

$$\alpha_1 = \frac{\Sigma_i (r_i.A - \bar{A})(r_i.B - \bar{B})}{\Sigma_i (r_i.A - \bar{A})^2}, \tag{4.1}$$

$$\alpha_0 = \bar{B} - \alpha_1 \bar{A}, \tag{4.2}$$

where $\bar{A}$ is the mean of the A values and $\bar{B}$ is the mean of the B values. There are equivalent formulas for the slope:

$$\alpha_1 = \frac{(\Sigma_i r_i.A\ r_i.B) - (N\bar{A}\bar{B})}{(\Sigma_i r_i.A^2) - (N\bar{A}^2)} \tag{4.3}$$

or

$$\alpha_1 = \frac{N\Sigma_i(r_i.A\ r_i.B) - (\Sigma_i r_i.A)(\Sigma_i r_i.B)}{N\Sigma_i(r_i.A^2) - (\Sigma_i r_i.A)^2} \tag{4.4}$$

where N is the number of data records in the dataset. There are also equivalent formulas for the intercept:

$$\alpha_0 = \frac{1}{N}(\Sigma_i r_i.B - \alpha_1 \Sigma_i r_i.A). \tag{4.5}$$

Looking at formula 4.1 for α_1, it is clear that the top is the covariance of A and B, and the bottom is the variance of A, that is, α_1 can also be expressed as

$$\alpha_1 = \frac{Cov(A, B)}{Var(A)} = Corr(A, B)\frac{stdev(B)}{stdev(A)}. \tag{4.6}$$

All these formulas can be expressed quite easily in SQL: in a system like Postgres, where Covariance and Variance come as standard functions, formula 4.6 is also the easiest. But even in systems without these functions, writing the formulas in SQL is a matter of putting together their pieces:

- $\Sigma_i(r_i.A - \bar{A})$ is simply the variance, the sum of the differences between values of A and their mean.
- $\Sigma_i(r_i.A - \bar{A})(r_i.B - \bar{B})$ is simply the product of the variances, or the sum of the product of the differences between values of A and their mean and values of B and their mean.
- $(\Sigma_i r_i.A)$ simply requires us to sum the values of A.
- $(\Sigma_i r_i.A\ r_i.B)$ requires us to multiply, on each row, the value of A times the value of B, and sum the results.
- $(\Sigma_i r_i.A^2)$ requires that we square the value of A and add the results—note that this is different from $(\Sigma_i r_i.A)^2$, where we first add the values of A and then square the result.

For simplicity, we can do these calculations in the FROM clause and compute α_1 before α_0 since the value of α_0 can be derived from that of α_1. Our dataset consists of records that contain attributes A and B. For Definitions 4.3 and 4.5, we get

```
SELECT ((SB * SAA) - (SA * SAB)) /
         ((N * (SAA)) - (SA * SA)) AS intercept,
       ((N * SAB) - (SA * SB))  /
```

```
         ((N * SAA) - (SA * SA)) AS slope
FROM (SELECT sum(A) AS SA,
             sum(B) AS SB,
             sum(A * A) AS SAA,
             sum(A * B) AS SAB,
             count(*) AS N
      FROM Data);
```

Example: Simple Linear Regression

Assume a real estate dataset with schema:

`(Id, Address, size, price, num-beds, num-baths)`

with an Id and an address for each property, together with the size in square feet and the price paid for it the last time it was sold, as well as the number of bedrooms and the number of bathrooms in the house.

We believe that the price of a house is a direct result of its size. Then we could calculate a simple regression with the size as the predictor and the price as the dependent variable.

```
SELECT
  ((SumPrice * sumPriceSq) - (SumSize * sumSizePrice)) /
    ((N * (sumPriceSq)) - (SumSize * SumSize)) AS intercept,
  ((N * sumSizePrice) - (SumSize * SumPrice))  /
    ((N * sumPriceSq) - (SumSize * SumSize)) AS slope
FROM (SELECT sum(size) AS SumSize,
             sum(price) AS SumPrice,
             sum(size * size) AS sumPriceSq,
             sum(size * price) AS sumSizePrice,
             sum(price * price) AS SumPriceSq,
             count(*) AS N
      FROM Data);
```

Exercise 4.4 Modify the previous example to compute α_1 and α_0 using Definitions 4.1 and 4.2. Apply it to the New York real estate sales dataset using `GROSS SQUARE FEET` and `SALE PRICE`.

Exercise 4.5 Modify the previous example to compute α_1 and α_0 using Definitions 4.4 and 4.2. Apply it to the New York real estate sales dataset using `GROSS SQUARE FEET` and `SALE PRICE`.

What if we want to use more than one independent variable? Unfortunately, the formula for the general case is quite complex. We show here the 2-variable case, for which it is still possible to give a somewhat reasonable SQL query. In this case, we are looking at an equation:

$$B = \alpha_0 + \alpha_1 A_1 + \alpha_2 A_2,$$

where A_1 and A_2 are the predicting attributes (independent variables), B the predicted attribute (dependent variable), and $\alpha_0, \alpha_1, \alpha_2$ the parameters to be learned from the data. To calculate these parameters, we proceed in three steps:

1. Compute $SUM(A_1)$, $SUM(A_2)$, $SUM(A_1^2)$, $SUM(A_2^2)$, $SUM(B)$, $SUM(A_1 \cdot B)$, $SUM(A_2 \cdot B)$, and $SUM(A_1 \cdot A_2)$.
2. Using the previous results, compute
   - $SA_1^2 = SUM(A_1^2) - \frac{SUM(A_1)^2}{n}$;
   - $SA_2^2 = SUM(A_2^2) - \frac{SUM(A_2)^2}{n}$;
   - $SA_1B = SUM(A_1 \cdot B) - \frac{SUM(A_1)SUM(B)}{n}$;
   - $SA_2B = SUM(A_2 \cdot B) - \frac{SUM(A_2)SUM(B)}{n}$;
   - $SA_1A_2 = SUM(A_1 \cdot A_2) - \frac{SUM(A_1)SUM(A_2)}{n}$;
3. Finally, compute

$$\alpha_1 = \frac{(SA_2^2)(SA_1B) - (SA_1A_2 \cdot SA_2B)}{(SA_1^2)(SA_2^2) - (SA_1A_2)^2},$$

$$\alpha_2 = \frac{(SA_1^2)(SA_2B) - (SA_1A_2 \cdot SA_2B)}{(SA_1^2)(SA_2^2) - (SA_1A_2)^2},$$

$$\alpha_0 = SUM(B - \alpha_1 A_1 - \alpha_2 A_2).$$

Note that, in spite of the complexity of the formula, many factors are reused.

As before, this can be put into an SQL query by using subqueries in FROM to carry out computations in a step-by-step manner.

Exercise 4.6 Write an SQL query on real estate dataset to apply linear regression to number of bedrooms and number of bathrooms in a house to predict its price.

Linear regression is a well-known, simple technique, so it is often tried first on many datasets. However, it has some severe limitations of which we should be aware. First and foremost, it will always 'work,' in the sense that it will always yield an answer (the one that minimizes the error, as defined above). However, even this answer may not be very good; that is, the error it produces may still be quite large. This is because there are two strong assumptions built into linear regression: first that there is a *linear* relation between independent and dependent variables, not a more complex one.[3] Second, it assumes that the errors (and there will always be errors; perfect fits are extremely unlikely with real data[4]) behave very nicely (technically speaking, errors must be independent and identically distributed with

[3]This is a very strong assumption: see https://en.wikipedia.org/wiki/Anscombe's_quartet.

[4]In fact, perfect fits are usually a cause for suspicion and one of the reasons some cheaters have been caught.

a normal distribution). Regression will not warn us that the relationship we are looking for is not there or is not linear. How do we know that our regression result is any good?

Recall that we minimized the sum of square errors, SSEs, in our calculation. Another measure of error is the *regression sum of squares*:

$$SSR = \Sigma_i (f(r_i.A) - \bar{B})^2.$$

This measure tells us how far our estimates are from the average of the real data value (the variance of the estimates). But to make sense of this, we need to know the variance of the real values:

$$SSTO = \Sigma_i (r_i.B - \bar{B})^2.$$

The key is to note that $SSTO = SSR + SSE$. Because we can think of SSE as the variance of the data, we can test to see how much of it we account for with the regression; this value is usually called *R-square* (R^2) and defined as

$$R^2 = \frac{SSR}{SSTO} = 1 - \frac{SSE}{SSTO}.$$

The value of R^2 is between 0 and 1 (although they are many times expressed as a percentage). A high value means that most of the variance on the predicted attribute B is accounted for by the variance in the predictor attribute A. In the context of linear regression, it means that the slope and intercept obtained fit the data quite well.

An important note: the Pearson *correlation coefficient r* that we saw in Sect. 3.2.2 turns out to be the square root of R-square, $r = \sqrt{R^2}$.

Exercise 4.7 Compute the R-squared value for your results predicting the price of a house from its size on the New York real estate dataset (any version).

Another issue with linear regression is that it requires that we normalize all our data before trying it; if an attribute is of much larger magnitude than others, it will dominate the calculations, creating the biggest differences—hence, it will be minimized even at the expense of other factors. For instance, assume we are again trying to predict the price at which a house will sell and that we decide to use, this time, the size, number of bedrooms, and number of bathrooms. It is clear that the number of bedrooms and bathrooms are going to be very small numbers (from 1 to 5 or so), while the size will be a number in the hundreds (if expressed in square meters) or the thousands (if expressed in square feet), perhaps even more. If we try to use all these attributes to predict the price (and it would seem reasonable to do so), we need to normalize the size attribute or it will dominate the calculations (this is sometimes called *feature scaling* in Machine Learning).

Finally, linear regression is very sensitive to outliers. An extreme value creates a very large error and, as in the case of large magnitude attributes, the approach will

try to minimize this error even at the expense of other values or other attributes. Therefore, it is very important to check the data for outliers before applying linear regression.

4.2.3 Logistic Regression

In spite of its name, logistic regression is actually a classification algorithm. However, we describe it after linear regression because it uses linear regression at its core.

The idea of logistic regression is to estimate a linear regression but to interpret it as the *odds* of the probability that the response variable is in a certain class. In the simplest case, assume our response variable is binary, so it can belong to one of two classes, 0 or 1 (for example, whether a person is a good or bad risk for loans, or whether a student will pass or fail an exam, or a patient will survive a certain procedure, etc.). If p is the probability of being of class 1 (so that $1 - p$ is the probability of class 0), the odds of being of class 1 is $\frac{p}{1-p}$. Linear regression with predictors $A_1, \ldots, A_n$ aims to approximate $Pr(1|A_1, \ldots, A_n)$. Instead, logistic calculates the log of the odds:

$$log \frac{Pr(1|A_1, \ldots, A_n)}{(1 - Pr(1|A_1, \ldots, A_n))}.$$

It does this by assuming that these odds are the product of a linear regression. In simple form,

$$log \frac{p}{1-p} = \alpha_0 + \alpha_1 A_1 + \alpha_2 A_2 + \ldots + \alpha_n A_n,$$

where, as before, the $\alpha_0, \ldots, \alpha_n$ are the parameters we want to estimate for the predictor attributes $A_1, \ldots, A_n$. Note that if we can find out the values of $\alpha_0, \ldots, \alpha_n$, then we can find out p simply by reversing the above, which comes out (after a bit of algebra) as

$$p = Pr(1|A_1, \ldots, A_n) = \frac{1}{1 + exp(-\alpha_0 - \alpha_1 A_1 - \ldots - \alpha_n A_n)}.$$

For the case of one variable,

$$p = Pr(1|A) = \frac{1}{1 + exp(-\alpha_0 - \alpha_1 A)}.$$

Thus, the idea is to compute the parameters α_0 (intercept) and α_1 (slope) by using linear regression and then use them in the formula above. This formula is also called the *sigmoid* function, and it gives values between 0 and 1. Usually, this

approach is used in binary classification (i.e. deciding between two classes), with values between 0 and 0.5 are interpreted as denoting one of the classes, and values between 0.5 and 1 denoting the other class. Clearly, this can be added to our previous computation of b (intercept) and a (slope):

```
SELECT 1.0 / (1.0 + exp(-intercept * -slope * A)
FROM ...
```

where A is the attribute we are using to predict our result, and `slope` and `intercept` are the values computed using linear regression in the previous subsection.

Exercise 4.8 Assume that all we want to know about the real estate dataset is whether a house will sell for a high price (defined as more than $300,000) or a low price (defined as equal to or less than $300,000). Apply logistic regression to this problem using only size as predictor. Hint: first add a variable `classprice` to the dataset with values 1 (for high) and 0 (for low) depending on whether the house's price is above or below our threshold of $300,000 (we have seen how to do this in Sect. 3.3.1.1). Then pick some data randomly as training and apply linear regression to get a slope and intercept, which are then finally fed to the sigmoid function, as shown above.

4.3 Unsupervised Approaches

In unsupervised approaches, we are given a dataset without labels. The goal is to discover pattern or structure in the data without any guidance. For instance, instead of classifying records (called *data points* in this context) into a prefixed list of classes, we check to see if a set of potential classes emerges from the data itself. Classes obtained from the analysis of data are called *clusters*, and the task of finding such clusters is, naturally, called *clustering*. Note that this can be harder than classification, since we do not know anything a priori: the number of classes, or their nature. In general, unsupervised approaches are considered weaker than supervised ones, as it is to be expected since unsupervised approaches have less information to work with. However, since they do not require labeled data, unsupervised techniques are very important, as they can be applied to any dataset. They can also be very helpful in understanding the data, and some authors classify techniques like clustering and dimensionality reduction with EDA.

4.3.1 Distances and Clustering

Many approaches rely on defining how similar (or dissimilar) two records are and comparing all records in a dataset. Then, records that are similar enough to each

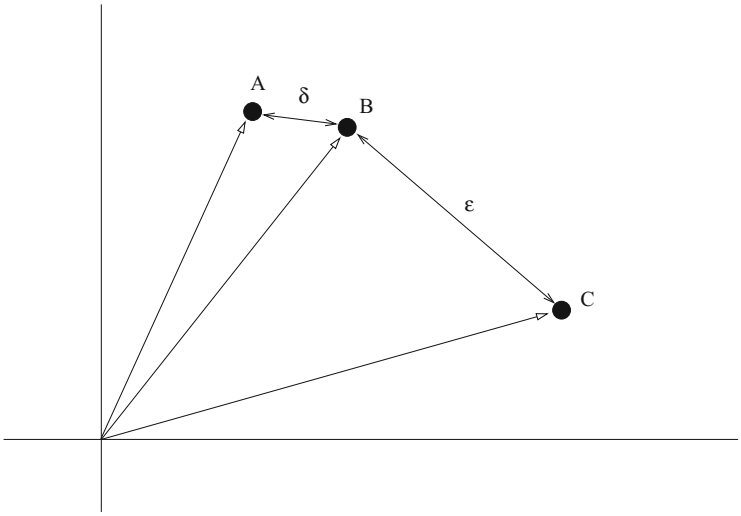


Fig. 4.1 Items A, B, and C as points in a (2-dimensional) space (or, equivalently, vectors); δ is the Euclidean distance between A and B and ϵ the Euclidean distance between B and C

other are grouped together to form a cluster. The intuitive idea is that similar records end in the same cluster, and dissimilar records end up in different clusters.

To implement this, we rely on the idea of *distance*, a function that assigns a number to each pair of records/data points. This function abstracts from the familiar idea of distance in geometry, with records seen as points in a space (hence the name; see Fig. 4.1). Intuitively, records/points that are 'close' to each other (the distance between them is small) are similar, while those that are far away according to the distance are dissimilar. Formally, a distance $D(x, y)$ is any function that applies to two data points x and y and fulfills the following:

- $D(x, y) \geq 0$ (the result is always positive), with $D(x, y) = 0$ only if $x = y$;
- $D(x, y) = D(y, x)$ (the function is symmetric; that is, order of arguments is not important);
- $D(x, y) \leq D(x, z) + D(z, y)$ (the 'triangle inequality.' Intuitively, the shortest distance from x to y is a straight line).

Distances between records/data points are usually obtained by combining distances between attributes. The typical distance for numerical attributes is the difference: the distance between numbers n_1 and n_2 is $\mid n_1 - n_2 \mid$. For categorical attributes, it is difficult to establish a meaningful distance. In some contexts, string similarity (as seen in Sect. 3.3.1.2) works well, but in many others the only distance that can be used with categorical attributes is the *trivial* distance:

$$D(s_1, s_2) = \begin{cases} 0 \text{ if } s_1 = s_2 \\ 1 \text{ otherwise} \end{cases}.$$

Distances are mostly used in the context of records with all attributes being numerical.

To combine individual distances into a distance between the two records, a very common approach is the Minkowski distance, defined as follows: for two data points x and y defined over attributes/features $A_1, \ldots, A_n$ (so that each data point is a record of values $(a_1, \ldots, a_n)$), their Minkowski distance is

$$d(x, y) = \sqrt[k]{\Sigma_{i=1}^{n}(x.A_i - y.A_i)^k},$$

where we use $x.A_i$ to denote the value of record/data point x for A_i, and likewise for $y.A_i$. Here, the number $k > 0$ is a parameter; in actuality, the Minkowski distance is a family of distance functions, one for each value of k. Some very well-known examples are the *Manhattan* distance, which uses $k = 1$:

$$d(x, y) = \Sigma_{i=1}^{n}(x.A_i - y.A_i)$$

and the *Euclidean* distance, which uses $k = 2$ (again, see Fig. 4.1):

$$d(x, y) = \sqrt{\Sigma_{i=1}^{n}(x.A_i - y.A_i)^2}.$$

These distances are quite useful, but they only work well under certain circumstances. First, they require that all the attributes have been scaled; otherwise, if one of them is much larger than others, it will dominate the distance. Recall the real estate example: a dataset with information about houses, including size, number of bedrooms, and number of bathrooms. If we are trying to determine how 'similar' two houses are, the Euclidean distance is not a bad idea, but before using it we need to normalize our data. Otherwise, any distance that combines all these attributes and does not pre-process them to remove this scale will be almost exclusively based on size and will be unable to distinguish between two houses of similar size but one with only one bathroom and another one with two or three.

The second problem with these distances is that they work better when the attributes are independent (or at least, uncorrelated). This problem is addressed by another famous distance, the *Mahalanobis* distance. This distance tries to normalize the differences between the values in each attribute by factoring in the covariances of the attributes. The idea is that if the covariance is high in absolute value, the attributes are highly correlated and any similarity should be discounted, as is to be expected. In contrast, if the covariance value is low, the attributes are closed to independent, so their distance should be considered important. To fully compute the Mahalanobis distance requires the covariance matrix of the dataset (a matrix whose element in the (i, j) position is the covariance between the i-th and j-th attributes of the dataset). This is hard (but not impossible) to compute in SQL; but then Mahalanobis requires the inverse of this matrix, which is not computable in SQL without using functions and some advanced tricks (and even then it is a real pain). But note that the covariance of any element with itself is simply its variance;

therefore, the diagonal of the covariance matrix is simply the variance of each attribute. Thus, one way to approximate the idea is to divide the distance between two values of an attribute by the standard deviation of that attribute (which is the square root of the variance, and for which a function exists in most SQL systems). Thus, we have a 'poor person' Mahalanobis distance:

$$d(x, y) = \Sigma_{i=1}^{n} \sqrt{\frac{(x.A_i - y.A_i)^2}{stddev(A_i)}}.$$

This is also called the *standardized Euclidean distance*, since it is equivalent to a Euclidean distance that uses standardized (z-score) values.

Another famous distance function is *cosine similarity*, inspired by linear algebra. Consider again each data point as literally a point in a space with n dimensions (one per attribute)—or, equivalently, as a vector (a vector is an object with a magnitude and a direction; these are usually depicted as arrows, with the length being the magnitude and arrowhead giving the direction): think of an arrow from the 'origin' of the space (the point (0,0,...)) to the point itself. Two points represent two vectors with a common origin; thus, they create an angle between them. The idea is that two similar points are very close to each other, so the angle between their vectors is very small (see Fig. 4.2). If they are pointing in exactly the same direction, the angle is 0° (so the cosine of the angle is 1); if they are at 90 degrees from each other, the angle is 90° (so the cosine is 0); if they are in the same 'line' but opposite directions, the

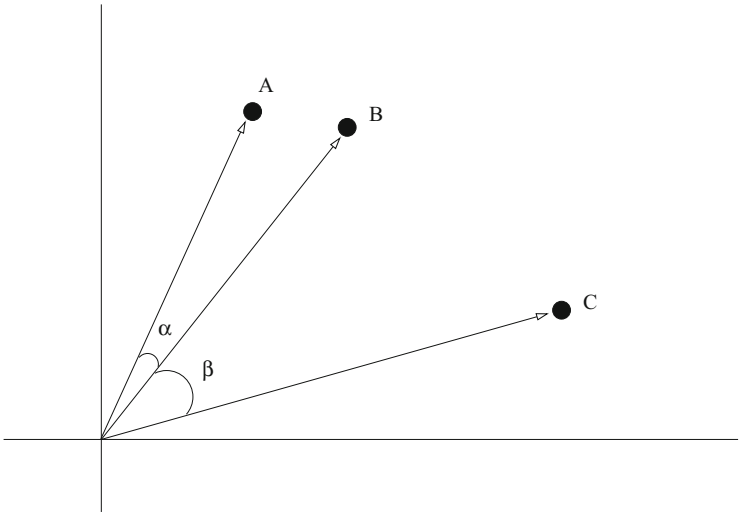


Fig. 4.2 Distance between A and B this time as cosine of angle α; distance between B and C as cosine of angle β

angle is 180° (so the cosine is −1). The cosine is calculated as

$$\frac{\Sigma_i^n (x.A_i\, y.A_i)}{\sqrt{\Sigma_i^n (x.A_i)^2}\sqrt{\Sigma_i^n (y.A_i)^2}}.$$

This distance ranges from −1 (meaning exactly opposite) to 1 (meaning exactly the same); 0 means 'orthogonal' or non-correlated. It is a popular distance used in several contexts, as we will see.

To implement any distance in SQL, the general pattern is: given a dataset `Data(id,dim1,dim2,...)`,

```
SELECT D1.id, D2.id, distance(D1.*, D2.*)
FROM Data D1, Data D2;
```

where `distance(D1.*, D2.*)` is the calculation of the chosen distance function using the attributes in the schema (or whatever attributes we deem relevant). For instance, the Euclidean distance over attributes `dim1, dim2, ...` becomes

```
SELECT  sqrt(pow(D1.dim1 - D2.dim1, 2) +
             pow(D1.dim2 - D2.dim2, 2) + ...)
FROM Data D1, Data D2;
```

Observe that the FROM clause generates a Cartesian product; this allows us to compare each point to each other point. However, it also creates a problem: in a dataset of size n (that is, with n data points), the number of comparisons is n^2. This is too much for even medium-sized datasets (set $n = 100,000$). It is possible to cut comparisons in half by adding

```
WHERE D1.id <= D2.id;
```

because of symmetry of distances. However, this is just a bit of relief. In Sect. 5.3 we will see a more efficient way to compute some simple distances.

Exercise 4.9 Give an SQL query to compute the Manhattan distance over a generic `Data` table.

Exercise 4.10 Give an SQL query to compute the cosine distance over a generic `Data` table.

4.3.1.1 K-Means Clustering

Clustering algorithms use distances between data points to group them together into clusters, or set of points such that the distance between any two points in a cluster is smaller than the distance between two points in any two clusters. Several strategies can be used to accomplish this. The *k-means clustering algorithm* uses the following approach:

1. Fix the number of clusters, k.
2. Pick k random data points and create the clusters by putting each point alone in a cluster. Set the *mean* (also called the *centroid*) of each cluster to be this (unique) point.
3. For each data point p, calculate the distance between p and each one of the cluster centroids; assign p to the cluster with the shortest distance.
4. After this is done, on each cluster recompute the *mean* or *centroid*.
5. Repeat the assignment of data points p to clusters by again computing the distance of p to each cluster, using the new centroid.
6. Repeat the last two steps until the clusters do not change, or until each cluster is *cohesive* enough, or for a fixed number of iterations.

The *cohesion* of a cluster can be measured in a number of ways; typical ones include taking the average of all distances between pairs of points in the cluster—sometimes, the maximum or the minimum is used instead of the average.

Clearly, this is an *iterative* algorithm, and the only way to implement it in SQL is with recursion. Recursion in SQL was explained in Sect. 4.6; here we explain the elements needed to write the final query. The following tables are needed, besides the table `Data(point-id,...)` storing the data:

- A table to store the means/centroids for each one of the clusters. We will also add a number to point out at which iteration the means were computed, since they will change over time. Thus, we have table `centroids(iteration, cluster-id, mean)`, which will be initialized with tuples `(1, 1, point)`, `(1, 2, point2)`, `...(1, k, pointk)`, where `point1, ..., pointk` are k randomly chosen data points.
- A table to keep track of the clustering, that is, the assignment of a cluster to each point. As before, we will use an iteration number to control the updates. Table `Clustering(iteration, point-id, cluster-id)` is sufficient. At initialization, all points in the dataset are included with an iteration number of 0 and a cluster id value of null.
- As an auxiliary table, we need to compute the distance between each point and each cluster, in order to establish to which cluster a point will go. A table `Distances(iteration, point-id, cluster-id, distance)` is not strictly necessary, but it will simplify the computation. This table can be started empty.

At each step, we need to compute a new assignment for each data point using the current mean (the one computed at the latest iteration) by updating the distances and updating the cluster centroids in turn. Thus, we have

```
INSERT INTO Distances
SELECT C.iter+1, D.point-id, C.cluster-id,
       d(D.attributes, C.mean)
FROM Data D, Centroids C
WHERE iter = (SELECT max(iter) FROM Centroids);
```

where `d(...)` is whatever distance function we have decided to use, and we restrict ourselves the most recent means by selecting the latest (highest) iteration—note also that we increase this latest iteration by one when inserting into the table, to signify that we are currently executing another iteration of the algorithm. After this is done, the new assignment of points to clusters can be done:

```
INSERT INTO Clustering
SELECT C.iter+1, D.point-id, D.cluster-id
FROM Distances D, Clustering C
WHERE  D.point-id = C.point-id
 and distance = (SELECT min(distance)
                 FROM Distances D2
                 WHERE D2.point-id = D.point-id)
 and iter = (SELECT max(iter) FROM C);
```

where we assign each point to the closest (minimum distance to mean) cluster. Finally, the centroids can then be recomputed:

```
INSERT INTO TABLE Centroids
 SELECT iter+1, cluster-id, value
 FROM (SELECT cluster-id, avg(dist) as value
       FROM Distances D, Clustering C
       WHERE D.cluster-id = C.cluster-id
       GROUP BY cluster-id) as TEMP,
       Centroids as C
 WHERE C.cluster-id = TEMP.cluster-id);
```

The iteration can be controlled by making sure that the iteration number does not exceed a prefixed value; this makes sure that the computation ends—something of importance in a database system!

4.3.2 The kNN Algorithm

k Nearest Neighbors is a simple and powerful algorithm, also based on using distances. Given a dataset D, a distance d on it, and a new data point p, the algorithm finds the k closest points (shortest distance) to p in D according to d. These points are the *neighbors* of p. Usually, k is set to be a small number, from 3 to 10, although this can change with datasets. Once this is done, kNN can be used for:

- Classification: if points are labeled with a class, a new point p can be assigned the class of its 'closest' (according to d) neighbor, or the most frequent class among p's k closest neighbors.
- Prediction: if an attribute y needs to be predicted from $A_1, \ldots, A_n$ on a new data point p, we can base this prediction on the y values of the k 'closest' (according to d) neighbors of p.

For classification, given dataset `Data(point, class)` and new point `pt`, we can compute `pt`'s class as follows (using the rule of selecting the majority's class):

```
WITH kNN AS (SELECT point, class
             FROM Data
             ORDER BY distance(point, pt)
             LIMIT k)
SELECT class
FROM (SELECT class, count(*) as freq
      FROM kNN
      GROUP BY class) as classFreq
WHERE freq = (SELECT max(freq) FROM classFreq);
```

We can use weighed counts for the class or use other aggregates.

Exercise 4.11 Write the kNN algorithm using the distance of neighbor p' as the weight of p' in the computation for the majority class.

For prediction, given dataset `Data(X1,...,Xn,Y)` and new data point `(X1',...,Xn')`, we can predict the `Y` for the new data point as follows:

```
SELECT avg(Y)
FROM (SELECT Y
      FROM Data
      ORDER BY distance(X1,..,Xn,X1',...,Xn')
      LIMIT K) as KNN;
```

Again, we can weigh the average by distance or use another aggregate.

Exercise 4.12 Write the kNN algorithm for prediction using the distance of neighbor p' as the weight of p' in the computation for the majority class.

Example: Using kNN Algorithm for Prediction

Assume table `Data(x,y,...)` with both attributes numerical. Assume we want to predict the value of `y` when `x=6.5`, but there is no point in the table with that value of `x`. We use kNN with $k = 2$, and the Manhattan distance on `x`.

```
SELECT avg(y)
FROM
 (SELECT y FROM Data ORDER BY abs(6.5-x) LIMIT 2) as KNN;
```

The value of the parameter k is set per problem and it reflects a trade-off. Low ks are sensitive to outliers; larger ks are robust, but more expensive to compute. The kNN algorithm, like all algorithms based on distance, requires that dimensions be standardized so that larger numerical values do not dominate. Also, note that all attributes used to compute the distance must be numerical. Categorical variables can be handled by creating dummy variables.

4.3.3 Association Rules

Association rules are a way to look at relations between *values* of an attribute. In some datasets, we have events that involve a set of items; in databases, such sets are usually called *transactions*.[5] The idea is to determine whether there are connections among the components of transactions; in particular, to determine whether certain components appear frequently together.

Association rules first appeared in the analysis of 'market baskets,' that is, of retail transactions. In this context, an *itemset* (i.e. a set of items) refers to the products involved in a transaction, as each transaction represents a customer purchasing several products at once—here, the event is the shopping transaction, and the components are the products bought. For instance, a SALES table could have schema *(transaction-id, product, quantity, price)* to specify that in a certain transaction (identified by its id) consisted of the purchase of two (amount) loafs of French bread (product) at $1.50 each (price); three cans of lentil soup, at $2.50 each; and one jug of milk for $3.00. A transaction would be represented by listing the components (product, in this case) row by row:

Transaction-id	Product	Quantity	Price
1	"French bread"	2	1.50
1	"Lentil soup"	3	2.50
1	"Milk"	1	3

The goal is to analyze the products purchased within transactions (items in the itemset) to see if this reveals any interesting pattern. Even though association rules found their first application in market analysis, they can be used in any scenario where a 'transaction' event can be identified involving several 'items.'

An association rule of the form $X \rightarrow Y$, where X and Y are itemsets, tells us that 'transactions' that contain values X are likely to also contain values Y' (i.e. $\{bread, soup\} \rightarrow \{milk\}$). X is called the *left-hand side* of the rule and Y is called the *right-hand side*. It is difficult to search for association rules because there are many possible combinations of values in a dataset (in our example, there are many items that can be purchased from your average supermarket); enumerating all combinations is tremendously costly. However, we are only interested on certain associations. In particular, we want the association to be frequent in the dataset, that is, to have a large presence, so that the chances of it being just an accident are small. Also, we want the association to be strong so that, again, we are not seeing something due merely to chance. The *support* of $X \rightarrow Y$ (in symbols, $|X, Y|$) is defined as the set of items (rows) that contain both values X and Y (in our example, the number of transactions that contain bread, soup, and milk among the products purchased). We want to focus on itemsets with high support (if such transactions are

[5] In this subsection, we use the term 'item' to refer to product or, in general, aspect of a transaction, as this terminology is deeply entrenched (i.e. see the idea of 'itemset' later).

infrequent, any pattern we find in them may be due to noise, or not very relevant). Among these, we want those where the association is strong. The *confidence* of $X \rightarrow Y$ is $\frac{|X|}{|X,Y|}$, that is, the fraction of 'transactions' that contain X among those that contain X and Y (in our example, we count which percentage contain milk and divide this by the number of transactions that involve bread, soup, and milk. This establishes how often the association is true, as a measure of the strength of the relationship among items.

Example: Association Rules

Using the table `SALES(transaction-id, product, quantity, price)`, we compute rules involving products by counting, for each pair of products A and B, the support and confidence of rule $A \rightarrow B$. For this, we first have to produce the pairs (A,B) of products in the same transaction; we do this computation first and produce a temporary table `Pairs`.

```
CREATE TABLE Pairs AS
SELECT  transaction-id, LeftHand.product as Left,
                        RightHand.Product as Right
FROM (SELECT DISTINCT transaction-id, product FROM SALES)
     AS LeftHand,
     (SELECT DISTINCT transaction-id, product FROM SALES)
     AS RightHand,
WHERE LeftHand.transaction-id = RightHand.transaction-id and
     LeftHand.product <> RightHand.product;
```

We now can count support and confidence easily; note that this is done over the original ORDERS table, since we need to include all data, even transactions with just a single product, which did not make it into PAIRS. We then join these results with the previous one:

```
SELECT Left, Right, both / (total * 1.0) as support
                    both / (countLeft * 1.0) as confidence
FROM (SELECT Left, Right, count(*) as both
      FROM Pairs
      GROUP BY Left, Right) AS Supports,
     (SELECT product, count(*) as countLeft
      FROM ORDERS
      GROUP BY product) as LeftCounts,
     (SELECT count(DISTINCT transaction-id) as total
      FROM ORDERS) AS all
WHERE Pairs.Left = Supports.Left and
      Pairs.Right = Supports.Right and
      Pairs.Left = countLeft.product
GROUP BY Pairs.Left, Pairs.Right;
```

Note that the support is expressed as a percentage (i.e. how many, out of all transactions, contain both items) since this is usually much more informative than the raw number. Note also that this will list all possible pairs, which can be very numerous. It is traditional to demand a minimal support (and sometimes, a minimal

confidence also). This threshold can be added easily to the query by adding a condition with a HAVING clause:

```
HAVING support > .1;
```

and likewise for a threshold on confidence.

Note that a rule $X \rightarrow Y$ and a rule $Y \rightarrow X$ are different: clearly, they both have the same support, but they may have very different confidences. When it is not known which way an association works, calculating both possibilities and picking out the one with the higher confidence is a good idea.

This still leaves open the fact that X and Y are sets of values: we could have rules like $\{beer, bread\} \rightarrow \{diapers, milk, eggs\}$. However, there are too many sets of values for us to calculate confidence and support for each pair of sets. Each element of the set involves a self-join of SALES with itself, so a rule like $\{beer, bread\} \rightarrow \{diapers, milk, eggs\}$ would require us to join SALES with itself 4 times to have lists of 5 elements, which then we need to break into two sides (left and right), which in itself can be done in several different ways (from only one element on the left and 4 on the side to the other way around). The challenge then is to extend the approach to more than one value per side.

A nice property of support is that a set X can only have high support if each $Z \subset X$ also has high support. In our example, for instance, the set $\{beer, bread\}$ can only have high support if both $\{beer\}$ and $\{bread\}$ have high support—since the support of $\{beer, bread\}$ is at most the smallest of the supports of $\{beer\}$ and $\{bread\}$. This is the idea behind the *a priori* algorithm: first, check the support of individual values and discard those below an appropriate threshold. Next, build sets of pairs of attributes using only those that survived the filter, compute the support for these pairs of attributes, and again filter those with low support (since even if each value individually has high support, the pair may not, as they may appear in very few common transactions). The process then continues: merge pairs of attributes with high support to create a three-element set and check the support of the result, discarding those results below the threshold. At each step, sets with low support can be thrown away since they cannot 'grow' into high support sets. For instance, the support of $\{diapers\}$ is greater or equal to the support of $\{diapers, milk\}$, which is greater than or equal to the support of $\{diapers, milk, egg\}$. Support goes down as the set grows, so we eventually run out of sets to consider; depending on the dataset, this may happen relatively early.

This suggests an improvement to the strategy above: start by computing the support of individual items and then compute support and confidence only for rules involving individual items with support above a threshold. In large databases, this can make quite a bit of difference.

Example: Efficient Association Rules

In our previous example, we computed all possible pairs of items in table Pairs. Instead, we can start with

```
CREATE TABLE Candidates AS
SELECT product
FROM SALES
GROUP BY product
HAVING count(*) > threshold;
```

and run query Pairs over table Candidates joined with Sales (we need to make sure that pairs of products are in the same transaction, using `transaction-id`).

```
CREATE TABLE Pairs AS
SELECT  transaction-id, LeftHand.product as Left,
                        RightHand.Product as Right
FROM (SELECT product as Left FROM Candidates) as C1,
     (SELECT product as Right FROM Candidates) as C2,
     (SELECT DISTINCT transaction-id, product FROM SALES)
      AS LeftHand,
     (SELECT DISTINCT transaction-id, product FROM SALES)
      AS RightHand,
WHERE LeftHand.transaction-id = RightHand.transaction-id and
     LeftHand.product <> RightHand.product and
     LeftHand.product = C1.product and
     RightHand.product = C2.product;
```

Even though this query is more complex than the previous one, the table Candidates may be considerably smaller than Sales (depending on the threshold and the dataset), so this result will also be smaller than before.

Also, note that the computation of support may be pushed to the table Pairs, and only those pairs of attributes with support above the threshold are then left to compute confidence.

Exercise 4.13 Rewrite the SQL for the creation of table `Pairs` keeping only pairs of attributes with support greater than 0.2.

Using this idea, we can compute larger sets of items by using only those in Candidates and joining such items with itemsets that are themselves large. For instance, in the example above, we can join Candidates and Pairs to generate 3-element sets. This is a simplification of *A priori* but is still more efficient than an exhaustive consideration of all items.

Example: Complex Association Rules

As stated, we can generate triplets of items using the Candidates and Pairs from earlier:

```
CREATE TABLE Triplets AS
SELECT S1.transaction-id, product1, product2, product3
FROM (SELECT product as product1 FROM Candidate)
      AS C,
     (SELECT Left as product2, Right as product3 FROM Pairs)
      AS P,
     (SELECT DISTINCT transaction-id, product FROM SALES)
      AS S1,
     (SELECT DISTINCT transaction-id, product FROM SALES)
      AS S2,
     (SELECT DISTINCT transaction-id, product FROM SALES)
      AS S3,
WHERE S1.transaction-id = S2.transaction-id and
      S2.transaction-id = S3.transaction-id and
      S1.product = product1 and
      S2.product = product2 and
      S3.product = product3;
```

As stated above, this table can be further trimmed by checking for appropriate support, i.e. counting the number of transactions with all 3 products.

Exercise 4.14 Rewrite the SQL to create table Triplets by trimming all triplets with a support lower than 0.1.

But we still face two problems: first, even if (A,B) and (C) have good support, this does not guarantee that (A,B,C) has good support, so we need to check this. Second, even if (A,B,C) has good support, there are six rules that can be generated from this set: $A \rightarrow B, C$; $B \rightarrow A, C$; $C \rightarrow A, B$; $A, B \rightarrow C$; $A, C \rightarrow B$; $B, C \rightarrow A$. We can, of course, generate all of them and check their confidence (they all share the same support, which we just checked). However, going beyond 3 attributes quickly becomes a combinatorial nightmare.

Exercise 4.15 Write SQL queries using table Triplets to compute, for a triple (A,B,C), confidence for rules $A \rightarrow B, C$; $B \rightarrow A, C$; and $C \rightarrow A, B$, and keep the rule with the highest confidence.

Even using the *A priori* algorithm in its full generality has a very high cost, as it generates a large number of sets that are later disregarded. Therefore, it is customary to investigate rules involving only small sets (2 or 3 items on each set, left and right). The good news is that, for the reasons just seen, very large sets are highly unlikely to have large supports.

4.4 Dealing with JSON/XML

As we saw in Sect. 2.3.1, there are two ways of dealing with semistructured data in databases. The first one was flattening, that is, transforming the hierarchical structure into a 'flat' one that leaves the data in a table. After flattening, all the algorithms that we have seen so far apply to this data too. The second method is to create a table with a column of type XML or JSON and store the data there in one of these formats. Unfortunately, there are very few algorithms that deal with data in this format. Hence, the typical approach is to flatten XML or JSON data into a tabular format and analyze the resulting table.

Most systems have functions that allow them to extract data from an XML or JSON column and represent it as a (plain) table; unfortunately, the names and types of such functions can vary considerably from system to system, as not all of them follow the SQL standard faithfully. The basic ideas behind these (and other) functions are always the same:

- If a value a associated with several values $b_1, \ldots, b_n$ (as a sub-element, in XML; as an array, in JSON), this is flattened into tuples $(a, b_1), \ldots, (a, b_n)$. If several levels exist, the process is repeated for each level.
- If we want to retrieve part of a complex element or object, a *path* is indicated to locate the part of interest. A path represents the location of a part by giving directions to 'navigate' the complex element or object from its root.[6] In XML notation, paths are indicated with forward slashes (as in "element/subelement/...") and optionally conditions or functions in square brackets (for instance, the value of an XML attribute is accessed using '[@attribute_name]').[7] In JSON, a path is built by using the dot notation ('.') to denote attributes inside an object and the square brackets ('[]') to denote an element inside an array.
- The schema of the target table (i.e. the target to be created) is sometimes given implicitly to the function used (some functions take parameters that indicate the names and type of attributes to be created), and sometimes given explicitly (in a separate clause). To accommodate the irregular nature of XML and JSON data, attributes not mentioned (explicitly or implicitly) but present in the data are ignored; and attributed mentioned (explicitly or implicitly) but not present in the data are given a default value (by the user or by the system).

In the following examples, we illustrate these ideas by showing some of the basic functionality in Postgres and MySQL.

[6] Recall that all hierarchical data can be seen as a 'tree,' as described in Sect. 1.2; a path means simply a description of the 'route' from the root to the given element—which in a tree is always unique.

[7] There is a whole language, XPath, devoted to denoting paths in XML, as they can become quite complex expressions. We do not discuss it here.

In Postgres, the main function for flattening XML data is called *xmltable*. Its (simplified) format is

```
xmltable(
   row_expression PASSING document_expression
       COLUMNS name { type [PATH column_expression]
                           [DEFAULT default_expression]
                           [NOT NULL | NULL]
                           | FOR ORDINALITY })
```

The xmltable function produces a table based on arguments:

- `document_expression`, which provides the XML document to operate on. The argument must be a well-formed XML document; fragments/forests are not accepted.
- `row_expression`, which is an XPath expression that is evaluated against the supplied XML document to obtain an ordered sequence of XML nodes. This sequence is what `xmltable` transforms into output rows.
- An optional set of column definitions, specifying the schema of the output table (if the COLUMNS clause is omitted, the rows in the result set contain a single column of type xml containing the data matched by `row_expression`). If COLUMNS is specified, each entry gives a single column name and type (other clauses are optional). A column marked FOR ORDINALITY will be populated with row numbers matching the order in which the output rows appeared in the original input XML document. At most one column may be marked FOR ORDINALITY. The `column_expression` for a column is an XPath expression that is evaluated for each row, relative to the result of the `row_expression`, to find the value of the column.

Note that in XML not all elements may have all attributes; this will result in a table with nulls unless a DEFAULT value is specified.

Example: XML Flattening in Postgres

Suppose we have the following data in XML:[8]

```
CREATE TABLE xmldata AS SELECT
xml $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>AU</COUNTRY_ID>
    <COUNTRY_NAME>Australia</COUNTRY_NAME>
  </ROW>
  <ROW id="5">
    <COUNTRY_ID>JP</COUNTRY_ID>
    <COUNTRY_NAME>Japan</COUNTRY_NAME>
    <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
```

[8]This example is taken from the Postgres documentation.

```
    <SIZE unit="sq_mi">145935</SIZE>
  </ROW>
  <ROW id="6">
    <COUNTRY_ID>SG</COUNTRY_ID>
    <COUNTRY_NAME>Singapore</COUNTRY_NAME>
    <SIZE unit="sq_km">697</SIZE>
  </ROW>
</ROWS>
$$ AS data;
```

Note that the above creates a table called `xmldata` with a single attribute called `data`, of type XML, on it.

```
SELECT xmltable.*
FROM xmldata,
     XMLTABLE('//ROWS/ROW' PASSING data
              COLUMNS id int PATH '@id',
                      order FOR ORDINALITY,
                      "COUNTRY_NAME" text,
                      ct_id text PATH 'COUNTRY_ID',
                      szsqkm float
                           PATH 'SIZE[@unit = "sq_km"]',
                      size_other text
            PATH 'concat(SIZE[@unit = "sq_km"], " ",
                         SIZE[@unit != "sq_km"]/@unit),
                    premier_name text PATH 'PREMIER_NAME'
                           DEFAULT 'not specified');
```

The above query transforms the data in `xmldata` into the following table:

```
id|order|COUNTRY_NAME|ct_id|szsqkm|size_other |premier_name
--+-----+------------+-----+------+-----------+----------+------
1 |   1 | Australia  | AU  |      |           | not specified
5 |   2 | Japan      | JP  |      |145935 sq_mi| Shinzo Abe
6 |   3 | Singapore  | SG  |  697 |           | not specified
```

Note how an attribute in the XML data (`SIZE`) has been split into two attributes in the table, depending on the value of XML attribute `unit`. Note also that there are missing values that are not explicitly marked (in attributes `size_sq_km` and `size_other`) and missing values explicitly marked (in attribute `premier_name`).

Exercise 4.16 The above practice of dealing with missing data in several ways is unwise. Modify the query above so that all absent data is marked by the string 'NA.'

As for JSON data, Postgres provides a set of functions that can be combined to flatten a JSON collection into a table:

- `JSON_each(JSON)` expands the outermost JSON object into a set of key/value pairs; each pair becomes a row in the resulting table.

```
SELECT *
FROM JSON_each('{"a":"foo", "b":"bar"}')

 key | value
-----+-------
 a   | "foo"
 b   | "bar"
```

- To find values inside a complex object, `JSON_extract_path(from_JSON JSON, VARIADIC path_elems text[])` returns the JSON value found following the argument `path_elems`:

```
JSON_extract_path('{"f2":{"f3":1},
                    "f4":{"f5":99,"f6":"foo"}}','f4')

{"f5":99,"f6":"foo"}
```

  Here 'f4' is the path.
- `JSON_populate_recordset(base anyelement, from_JSON JSON)` expands the outermost array of objects in the second argument to a set of rows whose columns match the record type defined by the first argument.

```
SELECT *
FROM JSON_populate_recordset(null::myrowtype,
                          '[{"a":1,"b":2},{"a":3,"b":4}]')

 a | b
---+---
 1 | 2
 3 | 4
```

- `JSON_to_recordset(JSON)` builds a table (set of records) from a JSON array of objects. The schema of the table (i.e. the structure of the record) is defined with AS clause.

```
SELECT *
FROM JSON_to_recordset('[{"a":1,"b":"foo"},
                         {"a":"2","c":"bar"}]')
     as x(a int, b text);

 a |  b
---+-----
 1 | foo
 2 |
```

  Note that elements not mentioned in the AS clause are ignored; elements mentioned but not present in the JSON data have an empty string to denote the missing value.

In MySQL, the function to deal with XML is called `ExtractValue`, and it takes a JSON object and a path into the object; it extracts the value of the path in the object. To break down a JSON object into parts, we call `ExtractValue` repeatedly with the same object and the paths leading to the different parts.

```
SELECT ExtractValue('<a>ccc<b>ddd</b></a>', '/a') AS val1,
       ExtractValue('<a>ccc<b>ddd</b></a>', '/a/b') AS val2,
       ExtractValue('<a>ccc<b>ddd</b></a>', '//b') AS val3,
       ExtractValue('<a>ccc<b>ddd</b></a>', '/b') AS val4,
       ExtractValue('<a>ccc<b>ddd</b><b>eee</b></a>', '//b')
         AS val5;

+------+------+------+------+---------+
| val1 | val2 | val3 | val4 | val5    |
+------+------+------+------+---------+
| ccc  | ddd  | ddd  |      | ddd eee |
+------+------+------+------+---------+
```

As it can be seen in the previous examples, the process is laborious due to the need to specify parts of the XML/JSON objects by giving paths into the parts of the object that are of interest. Unfortunately, there is no work-around this—but since most Data Mining and Machine Learning tools will not work directly with XML or JSON data, one must become familiar with these functions in order to transform the data as appropriate.

4.5 Text Analysis

The last type of data is unstructured, or text, data. There are, roughly speaking, three levels of text analysis:

- *Information Retrieval (IR)*: IR sees documents as *bags of words*: the semantics of a document are characterized by the words it contains. IR systems support *keyword search*, the retrieval of some documents in a collection by using a list of keywords. Documents containing those keywords are retrieved. The idea is that the user will pick keywords that documents of interest are likely to use. The retrieved documents are *ranked* to signify how relevant the documents are for the given keywords. This is the technique behind web search, with each web page is seen as a document. Most relational databases nowadays support keyword search, and this is the focus of this section. Even though this is the simplest form of text analysis, other tasks like *sentiment analysis* can be performed using IR techniques.
- *Information Extraction (IE)*: IE tries to extract snippets of information from text; such snippets are usually described by rows in certain tables. The schema of the tables depends on what information can be obtained from the text. For instance, the sentence "Paris is the largest city in France, and also its capital" can result in an entry *(Paris, France)* in a table with schema *(capital, country)*. Note that not all information presented in the sentence is extracted. However, what is extracted is now in a database-like form and can be analyzed with SQL queries. The extraction relies on a number of techniques, from simple pattern matching to neural networks. IE is usually not implemented in databases due to its complexity.

- *Natural Language Processing (NLP)*: NLP analyzes both the syntax and semantics of each sentence. A *parser* breaks a sentence into its components, and a semantic analyzer then uses this result to extract all the information from the sentence. For instance, the sentence of our previous example would be parsed as a complex sentence that can be seen as the conjunction of two simple sentences: "Paris is the largest city in France," and "Paris is the capital of France." Within each sentence, "Paris" is determined to be the subject, "is" the main verb, and what follows it is a direct complement that can in turn be broken into parts. NLP goes 'deeper' into analysis than IE: besides getting more information, the information retrieved is usually represented in a richer format (typically, the example would be expressed with a logic formula stating that for all cities x of France, if y is the population of x, then the population of Paris is greater than or equal to y). NLP analysis can get very complex and is usually carried out using deep learning techniques. As a result, NLP is usually also not implemented in databases.

In summary, IE and NLP are complex, specialized fields that rely heavily on machine learning; IR, while simpler, provides some basic tools that can be profitably used for analysis. We explain here how IR can be carried out in a database.

IR sees documents as *bags of words*; the semantics of a document can be characterized by its *content words*. *Non-content words* or *stopwords* (also called *function words* [1, 3, 12]) are words that carry no informational content (usually, articles and prepositions: *a, the, of*). They are present in almost any document, so they have no discriminatory value.[9] If we eliminate them, we are left with the content words. Note that no syntax or semantics is used; only word appearance is important. Not even order is used: `the cat is on the mat` and `the mat is on the cat` are exactly the same in IR (except for indices that keep word offsets, see below). Beyond the order of words in a sentence, we also throw away relationships among sentences, and any logical structure in the document (to build arguments, etc.).

Documents are *tokenized*, i.e. divided into discrete units or tokens. This is accomplished through a series of steps.

- *Determining canonical words*: words that are strongly related may be converted to a common term. For instance, verb forms (past, present, tense) may all be converted to a root. A particular example of this is *stemming*, getting rid of word inflections (prefixes, suffixes) to link several words to a common root (for instance, transforming *running* to *run*). Common stemming methods are based on morphological knowledge (and hence are language dependent).[10] Note that stemming introduces some risk: Porter's algorithm, one of the best known stemmers for English, stems *university* and *universal* to *univers*. Some stemmers also transform the case of letters (all to lowercase or uppercase).

[9] Most systems provide a list of stopwords, also sometimes called a *negative dictionary*.

[10] Obviously this only applies to languages where words can be declined.

- Another improvement is to accept *phrases*. Technically, phrases are n-grams (i.e. n terms adjacent in the text) that together have a meaning different from the separate terms, e.g. *operating system*. In systems that accept them, phrases are terms of their own right. However, to discover phrases may be complicated. There are basically two approaches, syntactical and statistical. In the syntactic approach, allowed combinations are listed by syntactic categories, i.e. *noun + noun*. However, this approach is weak, as most rules allow non-phrases. The statistical approach consists of looking at the number of occurrences of n terms $t_1, t_2, \ldots, t_n$ together and determining if this number is higher than it could be expected if the terms were independent (i.e. higher than the product of their individual occurrences). Recognizing phrases in general is a complex problem; in a language like English, phrases can become quite large and complex (*simulated back-propagation neural network*, *apples and oranges*).
- *Approximate string matching* is required in order to deal with typos, misspellings, etc., which can be quite frequent in some environments (e.g. the web) due to the lack of editorial control. There are two ways to attack the problem: one is by using methods like the ones shown in Sect. 3.3.1.2. The other approach is to break down each word into n-grams, or sequences of n characters, and compare the overlap of sequences between two given words. The parameter n depends on the language characteristics, like typical syllable size. Because approximate term matching may be expensive, it is typically not used by default in any IR approach.
- Another technique is to have a *thesaurus* or a similar resource to catch *synonyms* and choose a term to stand for all synonyms. This reduces the number of terms to deal with and allows the user to denote the same concept with different terms. However, thesaurus are usually built by hand and therefore their quality and coverage may vary substantially.

To explain how databases support IR, we introduce some terminology. Let D be a collection of m documents, that is, $|D| = m$ (D is sometimes called a *corpus*). Let T be the collection of all terms in D; for any $t \in T$, we denote by D_t the set of documents where t appears. For term t and document d, we denote with $tf(t, d)$ the occurrence frequency of t in d. The intuition behind this number is that if the term occurs frequently in a document, then it is likely to be very significant (and vice versa: if a term is only mentioned once or twice, it may be a mention in passing, meaning that the term does not represent the contents of the document). Note that since tf is an absolute value (not normalized) we may need to normalize it: as is, it tends to favor larger documents over short ones. tf can be normalized by the sum of term counts:

$$tf(t, d) = \frac{tf(t, d)}{\sum_{d' \in D} tf(t, d')}$$

or by the largest sum:

$$tf(t,d) = \frac{tf(t,d)}{max_{d' \in D} tf(t,d')}.$$

The *inverse document frequency (idf)* of t is computed as $idf(t, D) = \log \frac{|D|}{|D_t|} = \log \frac{m}{n}$ (note: when $n = m$, idf is 0; when $n = 1$, idf will be as large as possible). The idf tries to account for the fact that terms occurring in many documents are not good discriminators. The number $|D|$ acts as a normalization factor; we could also use $max_{t' \in T} D_{t'}$. A very commonly used normalization factor is $|D| - |D_t|$, i.e. the number of documents *not* containing the term.

Note that this is a property of the corpus a whole, not just of a document!

The *term frequency (tf)* of a term is the number of documents where the term occurs, i.e. $|D_t|$. In general, terms with high document frequencies are poor at discriminating among documents, since their appearance may not be significant.[11] However, terms that appear very rarely are also of limited help, as they do not tell us much about the corpus as a whole. It has been found that the best terms for searching are those that have medium document frequencies (not too high, not too low) (also, among terms occurring on the same number of documents, those with a higher variance are better).

A *tf-idf weight* is a weight assigned to a term in a document, obtained by combining the tf and the idf. Simple multiplication can be used, especially with the normalized tf:

$$TFIDF(t,d) = TF(t,d) \times idf(t,D).$$

To compute this weight, most systems create an *inverted (full text) index*. This is a list of all words in D and, for each word, a list of the documents where they appear. An inverted index could be represented by a table with schema `WORDS(term, docid)`. With a table like this, tf and idf can be computed in SQL:

```
SELECT term, docid, count(*)/len as tf
FROM WORDS,
     (SELECT  docid, count(*) as len
      FROM WORDS
      GROUP BY docid) as Temp
WHERE WORDS.docid = Temp.docid
GROUP BY term, docid;


SELECT term, total/count(distinct docid) as idf
FROM WORDS,
     (SELECT count(distinct docid) as total FROM WORDS)
     as Temp
GROUP BY term;
```

[11] Recall that ubiquitous presence was the reason to get rid of stopwords, but stopwords have high frequency too.

Most systems compute tf and idf from the index and use it for ranking results when keyword search is used, as explained next.

Keyword search consists of searching, among a corpus of documents, for the ones relevant for a certain goal; the search is based on a list of words, called *keywords*, that we expect to appear in any such document. As stated, most database systems support keyword search. If a table is created with at least one attribute of type `Text`, this attribute is considered to contain a corpus, with each row's value for the attribute being a document. It is possible to create an inverted index in such an attribute and then carry out keyword search on it. We now describe how keyword search is supported in MySQL and Postgres.

MySQL comes with its own list of stopwords, but it can be overwritten by a user, as follows: when using the `InnoDB` engine,

```
CREATE TABLE my_stopwords(value VARCHAR(30));
Query OK, 0 rows affected (0.01 sec)

INSERT INTO my_stopwords(value) VALUES ('or');
Query OK, 1 row affected (0.00 sec)

SET GLOBAL innodb_ft_server_stopword_table = 'my_stopwords';
Query OK, 0 rows affected (0.00 sec)
```

With the `MyISAM` engine, one creates a stopword file and then calls MySQL with option:

`-ft-stopword-file=file_name`

in the command line.

Keyword search in MySQL is based on the predicate

`MATCH(columns) AGAINST string`

`MATCH()` takes a comma-separated list that names the columns to be searched; `AGAINST` takes a string to search for, and an optional modifier to indicate type of search. For each row in the table, `MATCH()` returns a relevance score (based on $tf - idf$). There are three types of searches:

- Natural language search: searches for the string as is. A phrase that is enclosed within double quote (") characters matches only rows that contain the phrase literally, as it was typed. Without quotes, the system searches for the words in no particular order.
- Query Expansion search: after natural language search, words from the most relevant documents are added to the query, and the search is repeated with these additional words.
- Boolean search: the string is interpreted as a pattern, and the system searches for matches. The types of patterns allowed are described below.

A quick example shows how this is done:

```
CREATE TABLE articles (
   id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
   title VARCHAR(200),
   body TEXT,
```

```
    FULLTEXT (title,body)
) ENGINE=InnoDB;

SELECT *
FROM articles
WHERE MATCH (title,body)
      AGAINST ('database' IN NATURAL LANGUAGE MODE);
```

`MATCH()` can be used in the `SELECT` clause and in the `WHERE` clause. When `MATCH()` is used in a `WHERE` clause, rows are returned automatically sorted with the highest relevance first. When used in `SELECT`, the score assigned to each row is retrieved, but the returned rows are not ordered. For instance, the following returns the rows scored but in no particular order:

```
SELECT id, MATCH (title,body)
FROM articles
AGAINST ('Tutorial' IN NATURAL LANGUAGE MODE) AS score
```

To get the results ordered, we can repeat the `MATCH()` predicate in the `WHERE` clause:

```
SELECT id, body, MATCH (title,body) AGAINST
    ('Security implications of running MySQL as root'
    IN NATURAL LANGUAGE MODE) AS score
FROM articles
WHERE MATCH (title,body) AGAINST
    ('Security implications of running MySQL as root'
    IN NATURAL LANGUAGE MODE);
```

We can also sort results by relevance by using an ORDER BY clause:

```
SELECT id, title, body, MATCH (title,body)
       AGAINST ('database' IN BOOLEAN MODE) AS score
FROM articles
ORDER BY score DESC;
```

For Boolean search, several operators are supported:

- + acts like AND: all words must be present. If nothing is used, OR (some words must be present) is the default.
- − acts like NOT: word must be absent.
- @distance requires that word appear within a certain distance of each other (usually a 'distance' of k here means 'k words apart').
- " (quotes): literal phrase.

The following table gives examples of how Boolean search can be used[12]

'apple banana'	Find rows that contain at least one of the two words
'+apple +juice'	Find rows that contain both words
'+apple macintosh'	Find rows that contain the word "apple" but rank rows higher if they also contain "macintosh"
'+apple −macintosh'	Find rows that contain the word "apple" but not "macintosh"
'+apple macintosh'	Find rows that contain the word "apple," but if the row also contains the word "macintosh," rate it lower than if row does not
'+apple +(>turnover <strudel)'	Find rows that contain the words "apple" and "turnover," or "apple" and "strudel" (in any order), but rank "apple turnover" higher than "apple strudel."
'apple*'	Find rows that contain words such as "apple," "apples," "applesauce," or "applet."
'"some words"'	Find rows that contain the exact phrase "some words."

Postgres also has an inverted ('full text') index for documents, used to support keyword search. Like MySQL, Postgres uses a dictionary of stopwords. The text search operator in Postgres is represented by the '@@' symbol. It operates of what Postgres calls a 'tsvector' (a representation of the document, with all words normalized) and a 'tsquery' (a list of keywords representing the search criteria, also normalized). There are functions `to_tsquery, plainto_tsquery` and `phraseto_tsquery` that are helpful in converting user-written text into a proper tsquery (there is also `to_tsvector` for tsvectors). The tsquery may combine multiple terms using AND ('&'), OR ('|'), NOT ('!'), and FOLLOWED BY ('$<-> $') operators. The AND/OR/NOT operators are interpreted differently when they are used within the arguments of the FOLLOWED BY, since within FOLLOWED BY the exact position of the match is significant. Let a, b, c be keywords:

- The tsquery '$!a$' matches only documents that do not contain a anywhere, but '$!a <-> b$' is interpreted as "no a immediately after a b (but okay somewhere else in the document)";
- The tsquery '$a\&b$' normally requires that a and b both appear somewhere in the document, but '$(a\&b) <-> c$' requires a and b to appear immediately before a c.

Example: Keyword Search in Postgres

The following are examples of keyword searches in Postgres; they all return True except the second one, which is False:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@
       'cat & rat'::tsquery;
```

[12]From MYSQL's documentation.

```
SELECT 'fat & cow'::tsquery @@
       'a fat cat sat on a mat and ate a fat rat'::tsvector;

SELECT to_tsvector('fat cats ate fat rats') @@
                    to_tsquery('fat & rat');

SELECT to_tsvector('fatal error') @@
                   to_tsquery('fatal <-> error');
```

Another tool of text analysis is to generate (and often, count) the number of *n-grams* in a document. An n-gram is simply a sequence of n words; the most common case is $n = 2$ (called *bigrams*) and $n = 3$ (called *trigrams*). For instance, in the sentence "Mary had a little lamb," the bigrams are "Mary had," "had a," "a little," "little lamb."

To generate bigrams, we must break down a text into a sequence of words, with each word's position in the sequence explicitly marked. That is, we want to go from "Mary had a little lamb" to a set of pairs ("Mary," 1), ("had," 2), ("a," 3), ("little," 4), ("lamb," 5). In some systems, there are functions that do this for us; when such functions are not present, the process may be quite elaborate. Here we illustrate how this can be achieved in Postgres.

Assume table `user_comments(id int, comments text)` like this:

```
 comment_id |      comments
------------+-----------------------------
          1 | "i dont think this sam i am"
          2 | "mary had a little lamb"
(2 rows)
```

The breakdown process works in 3 steps. First, we make the comments into arrays of words:

```
CREATE TABLE word_list as (
SELECT id as comment_id,
        string_to_array(
          regexp_replace(
            lower(comment),
            E'[^a-z0-9_]+', ' ', 'g'),
        ' ') as word_array
FROM user_comments);
```

First we use regexp_replace to clean up the text, converting all the characters we do not care about to spaces. The 'g' at the end tells Postgres to replace all the matches, not just the first. Then we use string_to_array with a space (' ') as its split parameter to convert the cleaned comments into arrays. At the same time we will select the id of the original comment as that will be helpful later. This creates the following result:

```
 comment_id |          word_array
------------+------------------------------
          1 | {i,dont,think,this,sam,i,am}
          2 | {mary,had,a,little,lamb}
(2 rows)
```

Second, we break down the arrays into rows and keep the order:

```
CREATE TABLE word_indexes as (
SELECT comment_id, word_array,
       generate_subscripts(word_array, 1) as word_id
FROM word_list);
```

This uses the function `generate_subscript`, which generates a sequence 1, 2, ..., m, where m is the size of an array passed as first argument. This generates the table

```
 comment_id |          word_array          | word_id
------------+------------------------------+---------
          1 | {i,dont,think,this,sam,i,am} |       1
          1 | {i,dont,think,this,sam,i,am} |       2
          1 | {i,dont,think,this,sam,i,am} |       3
          1 | {i,dont,think,this,sam,i,am} |       4
          1 | {i,dont,think,this,sam,i,am} |       5
          1 | {i,dont,think,this,sam,i,am} |       6
          1 | {i,dont,think,this,sam,i,am} |       7
          2 | {mary,had,a,little,lamb}     |       1
          2 | {mary,had,a,little,lamb}     |       2
          2 | {mary,had,a,little,lamb}     |       3
          2 | {mary,had,a,little,lamb}     |       4
          2 | {mary,had,a,little,lamb}     |       5
```

Third, we use the array index to get individual words out, together with their position:

```
CREATE TABLE numbered_words AS
SELECT comment_id, word_array[word_id] word, word_id as pos
FROM word_indexes);
```

which yields the table

```
comment_id |  word  | pos
------------+--------+-----
          1 | i      |   1
          1 | dont   |   2
          1 | think  |   3
          1 | this   |   4
          1 | sam    |   5
          1 | i      |   6
          1 | am     |   7
          2 | mary   |   1
          2 | had    |   2
          2 | a      |   3
          2 | little |   4
          2 | lamb   |   5
```

Now we can make bigrams by self-join this table with itself:

```
SELECT  nw1.word,  nw2.word
FROM numbered_words nw1
     join numbered_words nw2 on
        nw1.word_id = nw2.word_id - 1
        and nw1.comment_id = nw2.comment_id;
```

From here we can do further analysis; for instance, we can get the bigram frequencies:

```
SELECT nw1.word, nw2.word, count(*)
FROM numbered_words nw1
     join numbered_words nw2 on
       nw1.word_id = nw2.word_id - 1
       and nw1.comment_id = nw2.comment_id
GROUP BY nw1.word, nw2.word
ORDER BY count(*) desc;
```

Clearly, trigrams (and n-grams) in general can also be obtained using 2 (or $n-1$) self-joins.

Another type of analysis that has become very popular with text is *sentiment analysis* (also called `sentiment detection`) [12]. Given a text (document) T and a target t (which can be a product, or a person, or an idea), we assume that T expresses some opinions about t, either in a *positive* (favorable, supportive) or *negative* (critical) way. While sentiment analysis can be quite tricky, a rough approximation can be achieved as follows: first, we come up with a list giving certain words (mostly, adjectives) a positive or negative score (for instance, 'good' or 'great' would have a positive score, while 'terrible,' 'harmful' would have a negative score). Then we create a score for each document T by adding up the scores of words in T that are in our list. Assume, for instance, that we have a table `Sentiment(word,score)`, where `score` is a number between n and $-n$, that gives the 'sentiment value' of the word. Then we can break each document into a list of words as we saw previously when dealing with bigrams; given table `Words(docid, word)` we can estimate a score per document:

```
SELECT W.docid, sum(S.score) as sentiment
FROM Sentiment S, Words W
WHERE S.word = W.word
GROUP BY W.docid;
```

However, it should be clear that this analysis is very approximate; for instance, positive words within the scope of a negation actually represent a negative sentiment ("this product was not good at all"). There are also other subtle problems, like irony. More sophisticated NLP techniques are currently used to extract sentiment; but when dealing with large collections, the above can be a good first step to focus on a smaller set of documents.

Exercise 4.17 As an improvement over the approach proposed, create a table from `Words(docid,word,position)` where all words that are preceded by an negation

("non," "no," "isn't," "wasn't," "don't") have their weight changed from m to $-m$ (note that this will turn positive words into negative words and negative words into positive ones).

4.6 Graph Analytics: Recursive Queries

Graphs can be analyzed in many different ways, but most analyses look for connectivity (what paths exist in the graph) and patterns (is there a part of the graph that has this links?), since one of the fundamental characteristics of networks (or graphs in general) is connectivity. We might want to know how to go from A to B, or how two people are connected, and we also want to know how many "hops" separate two nodes—in networks, 'distance' usually refers to the length of the shortest path between two nodes and is also called 'degree of separation.' For instance, social networks like LinkedIn show our connections or search results sorted by degree of separation, and trip planning sites show how many flights you have to take to reach your destination, usually listing direct connections first.

We start with paths: assume the standard representation of graphs with two tables called `NODES(id, ....)` and `EDGES(source, dest, weight)` as introduced in Sect. 2.3.2. Listing the nodes directly connected to a given node i (that is, connected by a path of length 1) is very simple:

```
SELECT *
FROM nodes N JOIN edges E ON N.id = E.dest
WHERE e.source = i;
```

or, in the case of undirected edges:

```
SELECT * FROM nodes WHERE id IN (
  SELECT source FROM edges WHERE dest = i
  UNION
  SELECT dest FROM edges WHERE source = i);
```

Nodes connected by a 2-step path are also easy to get:

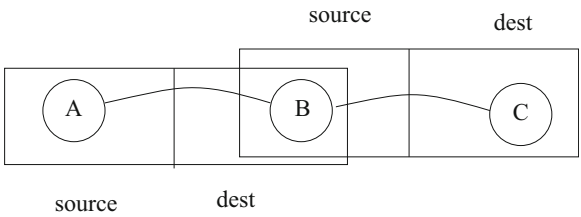
```
SELECT E1.source, E2.dest
FROM edges E1 JOIN edges E2 ON (E1.dest = E2.source);
```

Every step requires a join. To get all nodes connected by a 3-step path, we use

```
SELECT E1.source, E3.dest
FROM edges E1 JOIN edges E2 ON (E1.dest = E2.source)
     JOIN edges E3 ON (E2.dest = E3.source);
```

Note that we are joining the `EDGES` table with itself to create paths, since this is how paths are expressed in this representation (see Fig. 4.3). In general, finding a path with length n requires $n - 1$ joins.

Fig. 4.3 Paths as self-joins



source	dest
A	B
B	C
...	...

The problem is that finding arbitrary paths requires more flexibility, since we do not know in advance how long a path may be, and therefore we cannot fix the number of steps. In order to fully analyze the graph, we need to use recursive queries.

One way to do this is to create a temporary table holding all the possible paths between two nodes. This is called the *transitive closure* of the graph and can be done in a single statement as follows:

```
WITH RECURSIVE
transitive_closure(source, dest, distance, path_string) AS
(SELECT source, dest, 1 AS distance,
        source || '.' || dest || '.' AS path_string
 FROM edges
UNION ALL
 SELECT tc.source, e.dest, tc.distance + 1,
        tc.path_string || e.dest || '.' AS path_string
FROM transitive_closure AS TC JOIN edges AS E
     ON TC.dest = E.source
WHERE TC.path_string NOT LIKE '%' || E.source || '.%')
SELECT * FROM transitive_closure
ORDER BY source, dest, distance;
```

We now describe the example in detail:

- We start with the `WITH RECURSIVE` statement. In some systems, the keyword `RECURSIVE` does not need to be used; simply `WITH` will result in a recursive query as the system detects the general pattern of such queries (explained next);

- The query itself is a UNION of two SELECT statements, which works as follows: both SELECT statements are computed, and their results put together (for this to work, both SELECT statements must produce tables with the same schema; the UNION operation is explained in more detail in Sect. 5.4).
- The second SELECT statement uses the table `transitive_closure`, which is the table being defined by the WITH statement. This is what makes this a *recursive* statement: the table being defined is used in the definition. The system computes the result of the WITH statement in stages, as follows: in the first stage, the table `transitive_closure` is created. At this point, it is an empty table (it contains no data). Hence, when the UNION statement is executed, the second query (which uses `transitive_closure` on its FROM clause, to be joined with data table `edges`) yields nothing, since the join of two tables, one of which is empty, yields an empty table. However, the first SELECT statement of the UNION, which simply uses data table `edges` can be (and is) executed. The result is that, after this first stage, the UNION picks all the results from the first SELECT statement and deposits them in table `transitive_closure`; the second SELECT statement does not contribute anything. But now the system repeats the computation: it executes the UNION statement again, but this time table `transitive_closure` has data on it (the result of the first stage). On this new computation, the first SELECT statement again grabs data from table `edges` to add to `transitive_closure`, but this is the same data that we previously added, so this is ignored. However, the second SELECT statement this time can actually be carried out and it does, taking the join of `transitive_closure` and `edges`. Whatever is produced by this second SELECT statement is now the result of the UNION and is added to `transitive_closure`. Once this is done, the system repeats the computation again. This time (all times except the very first one) the first SELECT statement in the UNION brings nothing new and so is discarded, while the second SELECT statement may (or may not) yield additional tuples. As far as the UNION adds data to `transitive_closure`, the system will keep on repeating the computation. When, at some point, the UNION yields nothing new, no data is added to `transitive_closure`, and the whole computation ends. Intuitively, the first stage adds existing edges (1-step paths) to `transitive_closure` (essentially copying `edges` in `transitive_closure`); the second stage joins `transitive_closure` with `edges` (and, since at this point `transitive_closure` is a copy of `edges`, it joins `edges` with itself); the third stage again joins `transitive_closure` with `edges`—but since now `transitive_closure` contains all 2-step paths, it produces 3-step paths. The process continues adding one more step to each path that can be extended, until we run out of paths.
- Notice that in the WHERE condition of the second SELECT there is a check that stops the recursion in the presence of loops. As we go adding more steps to `transitive_closure`, we also add a string representing the path created: the statement

```
source || '.' || dest || '.' AS path_string
```

creates an initial string with the first two nodes, separated by dots (recall that || is string concatenation), while the statement

```
tc.path_string || e.dest || '.' AS path_string
```

adds, to the existing string, a new node reached on each recursive step. The condition `NOT LIKE` makes sure that the node we are about to add to the path is not already in there. This is very important to avoid the system looping without end.

Once we have the transitive closure, we can find if any two arbitrary nodes are connected or not, and if so, what path(s) exist between them.

Example: Connectivity in Flights

Assume a table `Flight(src, dst, price, ...)` that lists direct flights between airport `src` and airport `dst`, together with their price and other information. Suppose a customer is interested in flying from Boston to Los Angeles. There may be direct flights or there may not, but sometimes it is actually cheaper not to fly direct, so even if direct flights exist we may want to check alternatives. We write the query

```
WITH RECURSIVE
  travel(src, dst, total_price, itinerary, num_stops) AS
(SELECT src, dst, price, src ||'-'|| dst, 0)
 FROM Flights
 WHERE src = 'BOS'
UNION ALL
 SELECT T.src, F.dst, total_price + F.price,
        T.itinerary || '-' || F.dst,
        T.num_stops + 1
 FROM travel T, Flights F
 WHERE T.dst = F.src and
       position(F.src in T.itinerary) = 0)
SELECT *
FROM travel
WHERE src = 'BOS' and dst = 'LAX';
```

Note that we only copy flights that start at Boston airport (code 'BOS') in the first stage, since it makes no sense to *start* the trip somewhere else. However, we do not stop as soon as we find the Los Angeles airport (code 'LAX'), since we may find a 3-leg flight that is cheaper than another 2-leg flight; hence we do not want to stop searching as soon as we have reached LAX in some way. However, in the end we only retrieve flights that end at 'LAX.' Note also that at each stage, the total price (which was initialized with the price of the initial leg) is increased by the price of the last leg, the itinerary (which was initialized with the source and destination of the first flight) is enlarged with the new destination, and the number of stops (which was started at 0) is increased by 1.

Exercise 4.18 Modify the query in the previous example so that it does not return flights with more than 3 legs. Apply your query to the `ny_flights` dataset to get all flights from NYC (any airport) to Los Angeles ('LAX') in 3 or fewer legs.[13]

Exercise 4.19 A more realistic example would check that the departure time of a flight is within a reasonable margin (say, 2 h) of the arrival time of the previous flight. Assume there are attributes `arrival_time` and `departure_time` in `Flights` and modify the query in the previous exercise to only add legs to the itinerary if they fulfill this condition.

As for patterns, we are usually interested in finding a subset of nodes such that they are connected in a certain way. A typical example is the search for *triangles*, sets of three nodes with each one connected to the other two. Counting triangles is a basic tool for graph analysis (used, for instance, to spot fake users in social media). The following query counts triangles in our graph:

```
SELECT e1.source, Count(*)
FROM edges E1 join edges E2 on E1.dest = E2.source
     join edges E3 on E2.dest = E3.source
     and E3.dest = E1.source and E2.source <> E3.source
GROUP BY E1.source;
```

Another common pattern is to look for nodes that are not directly connected but have many common neighbors. Suppose we have computed the transitive closure `TC(source, dest, distance)` of a social network while keeping the distances between nodes, as above. Note that we can identify all the *neighbors* of a give node—nodes that are connected directly, so they have a distance of 0 (in fact, we can determine neighborhoods of any radius [any number of steps] for any given node). We want to find pairs of nodes a, b, such that they are not neighbors of each other, but they have more than n common neighbors (note that the fact that they have common neighbors implies that a and b are at a distance of 1):

```
SELECT TC1.source, TC1.dest, count(distinct TC2.dest) as cn
FROM TC as TC1, TC as TC2, TC as TC3
WHERE TC1.distance = 1 and
      TC2.source = TC1.source and TC2.distance = 1 and
      TC3.source = TC1.dest and TC3.distance = 1 and
      TC2.dest = TC3.dest
GROUP BY TC1.source, TC1.dest
HAVING cn > n;
```

In this query, we are using renaming of the table to compare 3 copies of it: one, to make sure the nodes of interest are directly connected, and two other copies, one for the neighbors of each node, which are then compared to each other. Finally, the grouping allows us to count how many such common neighbors (`'nc'`) we have found.

[13] Note: this query will take some time even in a powerful PC! Make sure your database system has plenty of memory.

The idea here is first to compute the transitive closure of the graph to gather information about the connectivity in the network. We can gather distances (as above) to distinguish nodes that are directly connected from those that are not, and also paths, in order to determine commonalities between nodes. There are many other patterns of interest that can be examined with this approach.

Finally, we consider the case where a graph is stored as an adjacency matrix, that is, as a table `Matrix(row, column, value)`, where the nodes of the graph have been numbered $1, \ldots, n$, and the entry `(i,j,v)` indicates that there is an edge between node `i` and node `j` with associated label or value `v`. Finding paths in the original graph can be achieved by multiplying the graph by itself, transposed. This is a simple operation in SQL, since multiplying matrices is straightforward and transposing the graph simply means using the row position as the column position and the column position as the row position.

First, assume we have two matrices $M1$ and $M2$ that can be multiplied (that is, the number of columns in $M1$ is the same as the number of rows in $M2$); then the product $M1 \times M2$ is simply

```
SELECT M1.row, M2.column, sum(M1.value*M2.value)
FROM M1, M2
WHERE M1.column = M2.row
GROUP BY M1.row, M2.column;
```

Exercise 4.20 Write an SQL query that produces the *sum* of matrices $M1$ and $M2$, assuming that they can be added (i.e. they have the same dimensions).

As noted, transposing is trivial:

```
SELECT  M1.column as row, M1.row as column, M1.value
FROM M1;
```

Multiplying matrix M by itself can be accomplished by using this schema with two copies of M and using the indices so that they represent transposition:

```
SELECT M1.row, M2.column, sum(M1.value*M2.value)
FROM M as M1,
     (SELECT M.column as row, M.row as column, M.value as value
      FROM M) as M2
WHERE M1.column = M2.row
GROUP BY M1.row, M2.column;
```

Exercise 4.21 Rewrite the query above to get rid of the subquery in the FROM clause. Hint: to do this, simply change how the indices are used.

Example: Boolean Matrices and Paths

Assume we want to find all paths of length up to k in a graph G and that G has been stored as the Boolean adjacency matrix M. We can compute M^k (M multiplied by itself k times), as follows.

```
WITH RECURSIVE PATHS(nodea int, nodeb int, steps int) AS
(SELECT a, b, 1 FROM MATRIX
 UNION
 SELECT a, b, steps + 1
 FROM MATRIX M, PATHS P
 WHERE M.b = P.nodea and steps < k)
SELECT *
FROM PATHS;
```

Note that if there are n nodes in G, we can compute all paths by using condition:

```
steps < (SELECT max(row) FROM MATRIX)
```

since `max(row)` (or `max(column)`, as this is a square matrix) is n, and (without loops) we cannot have any path longer than $n - 1$ steps.

4.7 Collaborative Filtering

Collaborative filtering is a family of techniques used by recommender systems. The basic idea is to filter information for an agent u using data about what other agents have seen/used/liked. A similarity distance between users establishes which users are similar to u; then their preferences are used to make recommendations for u. There are many variants of this idea.

Assume a table `Data(userid, itemid, rating)` with rows *(u,i,r)* if user u has given a rating r to item i. Then the simplest recommendation is: for a given user u and item i,

- item-item: find closest item i' to i, recommend i' to u.
- user-user: find closes user u' to u, recommend to u whatever u' likes.

To define 'closest' we need a distance. We introduced several distances in Sect. 4.3.1. Cosine is very popular among recommenders, so we use it in an example of item-item.

```
WITH similar_items(itemid, distance) AS
 SELECT D2.itemid, sum(D1.rating * D2.rating) /
                (sqrt(sum(D1.rating))*sqrt(sum(D2.ratings))
 FROM Data D1, Data D2
 WHERE D1.itemid = 'i' and D1.itemid <> D2.itemid
 GROUP BY D2.itemid
SELECT itemid
FROM similar_items
WHERE distance = (SELECT max(distance)
                  FROM similar_items);
```

This is the most similar item to item 'i.'

Exercise 4.22 Write an SQL query to determine the closest user to a given user u using cosine similarity over generic table `Data`.

Slope One is a family of algorithms for collaborative filtering [10] that uses the item-item approach. It uses an average of rating differences as distance, normalized by the number of common users. While this is a very simple measure (which makes it easy to implement and quite efficient), it sometimes performs on a par with more sophisticated approaches.

We again assume a table `Data(userid, itemid, rating)` and start by computing, for each pair of items, the average difference between their ratings as well as the number of common ratings. Using this, we can compute a recommendation for a user u on item i by applying the differences between i and other items and modifying u's rating on those other items accordingly.

```
WITH Diffs(itemid1, itemid2, freq, diff) AS
 (SELECT D1.itemid, D2.itemid, count(*),
        (sum(ud1.rating - ud2.rating))/count(*),
  FROM Data D1 join Data D2 on
        D1.userid = D2.userid and D1.itemid > D2.itemid
  GROUP BY D1.itemid, D2.itemid)
SELECT itemid2,  sum(freq) as freq,
    sum(freq*(diff + rating)) as pref,
    sum(freq*(diff + rating)) /sum(freq) as rating
FROM Diffs
WHERE itemid1 = 'i'
GROUP BY itemid2
ORDER BY rating
LIMIT 1;
```

This is the item closest to i according to Slope One.

Example: Slope One

We use the example from Wikipedia to illustrate the approach and predict Lucy's rating for item A. The example data (in a tidy format) is

Customer	Item	Rating
John	A	5
John	B	3
John	C	2
Mark	A	3
Mark	B	4
Lucy	B	2
Lucy	C	5

The table Diffs shows this result:

Item1	Item2	Count	Diff
A	B	2	0.5
A	C	1	3
B	C	2	−1

The 2 cases for A and B came from John (who gave A a 5 and B 3, for a difference of 2) and Mark (who gave A a 3 and B a 4 , for a difference of −1); both differences add up to 1, which divided by 2 cases yields a 0.5 average difference. Note that the symmetric entries (items A and B versus items B and A) are skipped by condition `itemid1 > itemid2`.

To predict Lucy's rate for item A, we use the difference between A and B (0.5) and add that to Lucy's rate for B (obtaining 2.5) and the difference between A and C (3) and add that to Lucy's rate for C (obtaining 8); these are weighted by the number of common ratings for A and B (2) and for A and C (1), to get $\frac{2(2.5)+1(8)}{2+1} = 4.33$, which is indeed the result of the query above over this data.

Exercise 4.23 Using the same data as the Wikipedia example, write a query to predict Mark's rating for item C.