8

Finding Patterns – Market Basket Analysis Using Association Rules

Think back to your last impulse purchase. Maybe in the grocery store checkout lane you bought a pack of chewing gum or a candy bar. Perhaps on a late-night trip for diapers and formula you picked up a caffeinated beverage or a six-pack of beer. You might have even bought this book on a bookseller's recommendation. These impulse buys are no coincidence, as retailers use sophisticated data analysis techniques to identify patterns that will drive retail behavior.

In years past, such recommendations were based on the subjective intuition of marketing professionals and inventory managers. Now, barcode scanners, inventory databases, and online shopping carts have generated transactional data that machine learning can use to learn purchasing patterns. The practice is commonly known as **market basket analysis** due to the fact that it has been so frequently applied to supermarket data.

Although the technique originated with shopping data, it is also useful in other contexts. By the time you finish this chapter, you will be able to apply market basket analysis techniques to your own tasks, whatever they may be. Generally, the work involves:

- Using simple performance measures to find associations in large databases
- Understanding the peculiarities of transactional data
- Knowing how to identify the useful and actionable patterns

The goal of a market basket analysis is to discover actionable patterns. Thus, as we apply the technique, you are likely to identify applications to your work even if you have no affiliation with a retail chain.

Understanding association rules

The building blocks of a market basket analysis are the items that may appear in any given transaction. Groups of one or more items are surrounded by brackets to indicate that they form a set, or more specifically, an **itemset** that appears in the data with some regularity. Transactions are specified in terms of itemsets, such as the following transaction that might be found in a typical grocery store:

$$\{\text{bread}, \text{peanut butter}, \text{jelly}\}$$

The result of a market basket analysis is a collection of **association rules** that specify patterns found in the relationships among items in the itemsets. Association rules are always composed from subsets of itemsets and are denoted by relating one itemset on the left-hand side (LHS) of the rule to another itemset on the right-hand side (RHS) of the rule. The LHS is the condition that needs to be met in order to trigger the rule, and the RHS is the expected result of meeting that condition. A rule identified from the preceding example transaction might be expressed in the form:

$$\{\text{peanut butter}, \text{jelly}\} \rightarrow \{\text{bread}\}$$

In plain language, this association rule states that if peanut butter and jelly are purchased together, then bread is also likely to be purchased. In other words, "peanut butter and jelly imply bread."

Developed in the context of retail transaction databases, association rules are not used for prediction, but rather for unsupervised knowledge discovery in large databases. This is unlike the classification and numeric prediction algorithms presented in previous chapters. Even so, you will find that association rule learners are closely related to and share many features of the classification rule learners presented in *Chapter 5*, *Divide and Conquer – Classification Using Decision Trees and Rules*.

Because association rule learners are unsupervised, there is no need for the algorithm to be trained; data does not need to be labeled ahead of time. The program is simply unleashed on a dataset in the hope that interesting associations are found. The downside, of course, is that there isn't an easy way to objectively measure the performance of a rule learner, aside from evaluating it for qualitative usefulness—typically, an eyeball test of some sort.

Although association rules are most often used for market basket analysis, they are helpful for finding patterns in many different types of data. Other potential applications include:

- Searching for interesting and frequently occurring patterns of DNA and protein sequences in cancer data
- Finding patterns of purchases or medical claims that occur in combination with fraudulent credit card or insurance use
- Identifying combinations of behavior that precede customers dropping their cellular phone service or upgrading their cable television package

Association rule analysis is used to search for interesting connections among a very large number of elements. Human beings are capable of such insight quite intuitively, but it often takes expert-level knowledge or a great deal of experience to do what a rule learning algorithm can do in minutes or even seconds. Additionally, some datasets are simply too large and complex for a human being to find the needle in the haystack.

The Apriori algorithm for association rule learning

Just as large transactional datasets create challenges for humans, these datasets also present challenges for machines. Transactional datasets can be large in both the number of transactions as well as the number of items or features that are recorded. The problem is that the number of potential itemsets grows exponentially with the number of features. Given k items that can appear or not appear in a set, there are 2^k possible itemsets that could be potential rules. A retailer that sells only 100 different items could have on the order of *2^100 = 1.27e+30* itemsets that an algorithm must evaluate—a seemingly impossible task.

Rather than evaluating each of these itemsets one by one, a smarter rule learning algorithm takes advantage of the fact that in reality, many of the potential combinations of items are rarely, if ever, found in practice. For instance, even if a store sells both automotive items and women's cosmetics, a set of *{motor oil, lipstick}* is likely to be extraordinarily uncommon. By ignoring these rare (and perhaps less important) combinations, it is possible to limit the scope of the search for rules to a more manageable size.

Much work has been done to identify heuristic algorithms for reducing the number of itemsets to search. Perhaps the most widely-used approach for efficiently searching large databases for rules is known as **Apriori**. Introduced in 1994 by Rakesh Agrawal and Ramakrishnan Srikant, the Apriori algorithm has since become somewhat synonymous with association rule learning. The name is derived from the fact that the algorithm utilizes a simple prior (that is, *a priori*) belief about the properties of frequent itemsets.

Before we discuss that in more depth, it's worth noting that this algorithm, like all learning algorithms, is not without its strengths and weaknesses. Some of these are listed as follows:

Strengths	Weaknesses
• Capable of working with large amounts of transactional data • Results in rules that are easy to understand • Useful for data mining and discovering unexpected knowledge in databases	• Not very helpful for small datasets • Takes effort to separate the true insight from the common sense • Easy to draw spurious conclusions from random patterns

As noted earlier, the Apriori algorithm employs a simple *a priori* belief as a guideline for reducing the association rule search space: all subsets of a frequent itemset must also be frequent. This heuristic is known as the **Apriori property**. Using this astute observation, it is possible to dramatically limit the number of rules to search. For example, the set *{motor oil, lipstick}* can only be frequent if both *{motor oil}* and *{lipstick}* occur frequently as well. Consequently, if either motor oil or lipstick is infrequent, then any set containing these items can be excluded from the search.

For additional details on the Apriori algorithm, refer to *Fast Algorithms for Mining Association Rules*, *Agrawal, R*, *Srikant, R*, *Proceedings of the 20th International Conference on Very Large Databases*, *1994*, *pp. 487-499*.

To see how this principle can be applied in a more realistic setting, let's consider a simple transaction database. The following table shows five completed transactions at an imaginary hospital's gift shop:

transaction ID	items purchased
1	{flowers, get well card, soda}
2	{plush toy bear, flowers, balloons, candy bar}
3	{get well card, candy bar, flowers}
4	{plush toy bear, balloons, soda}
5	{flowers, get well card, soda}

Figure 8.1: Itemsets representing five transactions in a hypothetical hospital's gift shop

By looking at the sets of purchases, one can infer that there are a couple of typical buying patterns. A person visiting a sick friend or family member tends to buy a get well card and flowers, while visitors to new mothers tend to buy plush toy bears and balloons. Such patterns are notable because they appear frequently enough to catch our interest; we simply apply a bit of logic and subject matter experience to explain the rule.

In a similar fashion, the Apriori algorithm uses statistical measures of an itemset's "interestingness" to locate association rules in much larger transaction databases. In the sections that follow, we will discover how Apriori computes such measures of interest, and how they are combined with the Apriori property to reduce the number of rules to be learned.

Measuring rule interest – support and confidence

Whether or not an association rule is deemed interesting is determined by two statistical measures: support and confidence. By providing minimum thresholds for each of these metrics and applying the Apriori principle, it is easy to drastically limit the number of rules reported, perhaps even to the point where only the obvious or common sense rules are identified. For this reason, it is important to carefully understand the types of rules that are excluded under these criteria.

The **support** of an itemset or rule measures how frequently it occurs in the data. For instance, the itemset *{get well card, flowers}* has the support of *3 / 5 = 0.6* in the hospital gift shop data. Similarly, the support for *{get well card}* → *{flowers}* is also *0.6*. Support can be calculated for any itemset or even a single item; for instance, the support for *{candy bar}* is *2/5 = 0.4*, since candy bars appear in 40 percent of purchases. A function defining support for itemset *X* could be defined as:

$$\text{confidence}(X \rightarrow Y) = \frac{\text{support}(X, Y)}{\text{support}(X)}$$

Here, *N* is the number of transactions in the database and *count(X)* is the number of transactions containing itemset *X*.

A rule's **confidence** is a measurement of its predictive power or accuracy. It is defined as the support of the itemset containing both *X* and *Y* divided by the support of the itemset containing only *X*:

$$\text{confidence}(X \rightarrow Y) = \frac{\text{support}(X, Y)}{\text{support}(X)}$$

Essentially, the confidence tells us the proportion of transactions where the presence of item or itemset *X* results in the presence of item or itemset *Y*. Keep in mind that the confidence that *X* leads to *Y* is not the same as the confidence that *Y* leads to *X*. For example, the confidence of *{flowers} → {get well card}* is *0.6 / 0.8 = 0.75*. In comparison, the confidence of *{get well card} → {flowers}* is *0.6 / 0.6 = 1.0*. This means that a purchase of flowers also includes the purchase of a get well card 75 percent of the time, while a purchase of a get well card also includes flowers 100 percent of the time. This information could be quite useful to the gift shop management.

You may have noticed similarities between support, confidence, and the Bayesian probability rules covered in *Chapter 4*, *Probabilistic Learning – Classification Using Naive Bayes*. In fact, *support(A, B)* is the same as $P(A \cap B)$ and *confidence(A → B)* is the same as $P(B \mid A)$. It is just the context that differs.

Rules like *{get well card} → {flowers}* are known as **strong rules** because they have both high support and confidence. One way to find more strong rules would be to examine every possible combination of items in the gift shop, measure the support and confidence, and report back only those rules that meet certain levels of interest. However, as noted before, this strategy is generally not feasible for anything but the smallest of datasets.

In the next section, you will see how the Apriori algorithm uses minimum levels of support and confidence with the Apriori principle to find strong rules quickly by reducing the number of rules to a more manageable level.

Building a set of rules with the Apriori principle

Recall that the Apriori principle states that all subsets of a frequent itemset must also be frequent. In other words, if *{A, B}* is frequent, then *{A}* and *{B}* must both be frequent. Recall also that, by definition, the support metric indicates how frequently an itemset appears in the data. Therefore, if we know that *{A}* does not meet a desired support threshold, there is no reason to consider *{A, B}* or any itemset containing *{A}*; it cannot possibly be frequent.

The Apriori algorithm uses this logic to exclude potential association rules prior to actually evaluating them. The process of creating rules then occurs in two phases:

1. Identifying all the itemsets that meet a minimum support threshold.
2. Creating rules from these itemsets using those meeting a minimum confidence threshold.

The first phase occurs in multiple iterations. Each successive iteration involves evaluating the support of a set of increasingly large itemsets. For instance, iteration one involves evaluating the set of 1-item itemsets (1-itemsets), iteration two evaluates the 2-itemsets, and so on. The result of each iteration *i* is a set of all the *i*-itemsets that meet the minimum support threshold.

All the itemsets from iteration *i* are combined in order to generate candidate itemsets for evaluation in iteration *i* + *1*. But the Apriori principle can eliminate some of them even before the next round begins. If *{A}*, *{B}*, and *{C}* are frequent in iteration one, while *{D}* is not frequent, then iteration two will consider only *{A, B}*, *{A, C}*, and *{B, C}*. Thus, the algorithm needs to evaluate only three itemsets rather than the six that would have been evaluated if the sets containing *D* had not been eliminated *a priori*.

Continuing this thought, suppose during iteration two it is discovered that *{A, B}* and *{B, C}* are frequent, but *{A, C}* is not. Although iteration three would normally begin by evaluating the support for *{A, B, C}*, this step is not necessary. Why not? The Apriori principle states that *{A, B, C}* cannot possibly be frequent, since the subset *{A, C}* is not. Therefore, having generated no new itemsets in iteration three, the algorithm may stop.

iteration	must evaluate	frequent itemsets	infrequent itemsets
1	{A}, {B}, {C}, {D}	{A}, {B}, {C}	{D}
2	{A, B}, {A, C}, {B, C} ~~{A, D}, {B, D}, {C, D}~~	{A, B}, {B, C}	{A, C}
3	~~{A, B, C}~~		
4	~~{A, B, C, D}~~		

Figure 8.2: The Apriori algorithm only needs to evaluate seven of the 12 potential itemsets

At this point, the second phase of the Apriori algorithm may begin. Given the set of frequent itemsets, association rules are generated from all possible subsets. For instance, *{A, B}* would result in candidate rules for *{A}* → *{B}* and *{B}* → *{A}*. These are evaluated against a minimum confidence threshold, and any rule that does not meet the desired confidence level is eliminated.

Example – identifying frequently purchased groceries with association rules

As noted in this chapter's introduction, market basket analysis is used behind the scenes for the recommendation systems used in many brick-and-mortar and online retailers. The learned association rules indicate the combinations of items that are often purchased together. Knowledge of these patterns provides insight into new ways a grocery chain might optimize the inventory, advertise promotions, or organize the physical layout of the store. For instance, if shoppers frequently purchase coffee or orange juice with a breakfast pastry, it may be possible to increase profit by relocating pastries closer to coffee and juice.

In this tutorial, we will perform a market basket analysis of transactional data from a grocery store. However, the techniques could be applied to many different types of problems, from movie recommendations, to dating sites, to finding dangerous interactions among medications. In doing so, we will see how the Apriori algorithm is able to efficiently evaluate a potentially massive set of association rules.

Step 1 – collecting data

Our market basket analysis will utilize purchase data from one month of operation at a real-world grocery store. The data contains 9,835 transactions, or about 327 transactions per day (roughly 30 transactions per hour in a 12-hour business day), suggesting that the retailer is not particularly large, nor is it particularly small.

The dataset used here was adapted from the `Groceries` dataset in the `arules` R package. For more information, see *Implications of Probabilistic Data Modeling for Mining Association Rules*, *Hahsler, M*, *Hornik, K*, *Reutterer, T*, *2005*. In *From Data and Information Analysis to Knowledge Engineering*, *Gaul W*, *Vichi M*, *Weihs C*, *Studies in Classification, Data Analysis, and Knowledge Organization*, *2006*, *pp. 598–605*.

A typical grocery store offers a huge variety of items. There might be five brands of milk, a dozen types of laundry detergent, and three brands of coffee. Given the moderate size of the retailer in this example, we will assume that it is not terribly concerned with finding rules that apply only to a specific brand of milk or detergent. With this in mind, all brand names have been removed from the purchases. This reduced the number of groceries to a more manageable 169 types, using broad categories such as chicken, frozen meals, margarine, and soda.

If you hope to identify highly specific association rules—such as whether customers prefer grape or strawberry jelly with their peanut butter—you will need a tremendous amount of transactional data. Large chain retailers use databases of many millions of transactions in order to find associations among particular brands, colors, or flavors of items.

Do you have any guesses about which types of items might be purchased together? Will wine and cheese be a common pairing? Bread and butter? Tea and honey? Let's dig into this data and see if these guesses can be confirmed.

Step 2 – exploring and preparing the data

Transactional data is stored in a slightly different format than we have used previously. Most of our prior analyses utilized data in a matrix where rows indicated example instances and columns indicated features. In the matrix format, all examples must have exactly the same set of features.

In comparison, transactional data is more freeform. As usual, each row in the data specifies a single example—in this case, a transaction. However, rather than having a set number of features, each record comprises a comma-separated list of any number of items, from one to many. In essence, the features may differ from example to example.

To follow along with this analysis, download the `groceries.csv` file from the Packt Publishing website and save it in your R working directory.

The first five rows of the raw `groceries.csv` file are as follows:

```
citrus fruit,semi-finished bread,margarine,ready soups
tropical fruit,yogurt,coffee
whole milk
pip fruit,yogurt,cream cheese,meat spreads
other vegetables,whole milk,condensed milk,long life bakery product
```

These lines indicate five separate grocery store transactions. The first transaction included four items: citrus fruit, semi-finished bread, margarine, and ready soups. In comparison, the third transaction included only one item: whole milk.

Suppose we tried to load the data using the `read.csv()` function as we did in prior analyses. R would happily comply and read the data into matrix format as follows:

	V1	V2	V3	V4
1	citrus fruit	semi-finished bread	margarine	ready soups
2	tropical fruit	yogurt	coffee	
3	whole milk			
4	pip fruit	yogurt	cream cheese	meat spreads
5	other vegetables	whole milk	condensed milk	long life bakery product

Figure 8.3: Transactional data incorrectly loaded into matrix format

You will notice that R created four columns to store the items in the transactional data: `V1`, `V2`, `V3`, and `V4`. Although this may seem reasonable, if we use the data in this form, we will encounter problems later on. R chose to create four variables because the first line had exactly four comma-separated values. However, we know that grocery purchases can contain more than four items; in the four-column design, such transactions will be broken across multiple rows in the matrix. We could try to remedy this by putting the transaction with the largest number of items at the top of the file, but this ignores another more problematic issue.

By structuring the data this way, R has constructed a set of features that record not just the items in the transactions, but also the order they appear. If we imagine our learning algorithm as an attempt to find a relationship among `V1`, `V2`, `V3`, and `V4`, then the whole milk in `V1` might be treated differently than the whole milk appearing in `V2`. Instead, we need a dataset that does not treat a transaction as a set of positions to be filled (or not filled) with specific items, but rather as a market basket that either contains or does not contain each particular item.

Data preparation – creating a sparse matrix for transaction data

The solution to this problem utilizes a data structure called a sparse matrix. You may recall that we used a sparse matrix for processing text data in *Chapter 4*, *Probabilistic Learning – Classification Using Naive Bayes*. Just as with the preceding dataset, each row in the sparse matrix indicates a transaction. However, the sparse matrix has a column (that is, feature) for every item that could possibly appear in someone's shopping bag. Since there are 169 different items in our grocery store data, our sparse matrix will contain 169 columns.

Why not just store this as a data frame as we did in most of our prior analyses? The reason is that as additional transactions and items are added, a conventional data structure quickly becomes too large to fit in the available memory. Even with the relatively small transactional dataset used here, the matrix contains nearly 1.7 million cells, most of which contain zeros (hence the name "sparse" matrix—there are very few non-zero values).

Since there is no benefit to storing all these zeros, a sparse matrix does not actually store the full matrix in memory; it only stores the cells that are occupied by an item. This allows the structure to be more memory efficient than an equivalently sized matrix or data frame.

In order to create the sparse matrix data structure from transactional data, we can use functionality provided by the `arules` (association rules) package. Install and load the package using the `install.packages("arules")` and `library(arules)` commands.

For more information on the `arules` package, refer to: *arules - A Computational Environment for Mining Association Rules and Frequent Item Sets*, *Hahsler, M, Gruen, B, Hornik, K, Journal of Statistical Software, 2005, Vol. 14.*

Because we're loading transactional data, we cannot simply use the `read.csv()` function used previously. Instead, `arules` provides a `read.transactions()` function that is similar to `read.csv()` with the exception that it results in a sparse matrix suitable for transactional data. The parameter `sep = ","` specifies that items in the input file are separated by a comma. To read the `groceries.csv` data into a sparse matrix named `groceries`, type the following line:

```
> groceries <- read.transactions("groceries.csv", sep = ",")
```

To see some basic information about the `groceries` matrix we just created, use the `summary()` function on the object:

```
> summary(groceries)
transactions as itemMatrix in sparse format with
 9835 rows (elements/itemsets/transactions) and
 169 columns (items) and a density of 0.02609146
```

The first block of information in the output (as shown previously) provides a summary of the sparse matrix we created. The output `9835 rows` refers to the number of transactions, and `169 columns` indicates each of the 169 different items that might appear in someone's grocery basket. Each cell in the matrix is a `1` if the item was purchased for the corresponding transaction, or `0` otherwise.

The **density** value of `0.02609146` (2.6 percent) refers to the proportion of non-zero matrix cells. Since there are *9,835 * 169 = 1,662,115* positions in the matrix, we can calculate that a total of *1,662,115 * 0.02609146 = 43,367* items were purchased during the store's 30 days of operation (ignoring the fact that duplicates of the same items might have been purchased). With an additional step, we can determine that the average transaction contained *43,367 / 9,835 = 4.409* distinct grocery items. Of course, if we look a little further down the output, we'll see that the mean number of items per transaction has already been provided.

The next block of `summary()` output lists the items that were most commonly found in the transactional data. Since *2,513 / 9,835 = 0.2555*, we can determine that whole milk appeared in 25.6 percent of transactions. Other vegetables, rolls/buns, soda, and yogurt round out the list of other common items, as follows:

```
most frequent items:
      whole milk other vegetables       rolls/buns
            2513             1903             1809
            soda           yogurt          (Other)
            1715             1372            34055
```

Finally, we are presented with a set of statistics about the size of the transactions. A total of 2,159 transactions contained only a single item, while one transaction had 32 items. The first quartile and median purchase size are two and three items respectively, implying that 25 percent of the transactions contained two or fewer items and about half contained three items or fewer. The mean of 4.409 items per transaction matches the value we calculated by hand.

```
element (itemset/transaction) length distribution:
sizes
   1    2    3    4    5    6    7    8    9   10   11   12
2159 1643 1299 1005  855  645  545  438  350  246  182  117
  13   14   15   16   17   18   19   20   21   22   23   24
  78   77   55   46   29   14   14    9   11    4    6    1
  26   27   28   29   32
   1    1    1    3    1

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   2.000   3.000   4.409   6.000  32.000
```

The `arules` package includes some useful features for examining transaction data. To look at the contents of the sparse matrix, use the `inspect()` function in combination with R's vector operators. The first five transactions can be viewed as follows:

```
> inspect(groceries[1:5])
  items
1 {citrus fruit,
   margarine,
   ready soups,
   semi-finished bread}
```

```
2 {coffee,
   tropical fruit,
   yogurt}
3 {whole milk}
4 {cream cheese,
   meat spreads,
   pip fruit,
   yogurt}
5 {condensed milk,
   long life bakery product,
   other vegetables,
   whole milk}
```

These transactions match our look at the original CSV file. To examine a particular item (that is, a column of data), use the `[row, column]` matrix notion. Using this with the `itemFrequency()` function allows us to see the proportion of transactions that contain the specified item. For instance, to view the support level for the first three items in the grocery data, use the following command:

```
> itemFrequency(groceries[, 1:3])
abrasive cleaner artif. sweetener   baby cosmetics
    0.0035587189     0.0032536858     0.0006100661
```

Notice that the items in the sparse matrix are sorted in columns by alphabetical order. Abrasive cleaner and artificial sweeteners are found in about 0.3 percent of the transactions, while baby cosmetics are found in about 0.06 percent of the transactions.

Visualizing item support – item frequency plots

To present these statistics visually, use the `itemFrequencyPlot()` function. This creates a bar chart depicting the proportion of transactions containing specified items. Since transactional data contains a very large number of items, you will often need to limit those appearing in the plot in order to produce a legible chart.

If you would like to display items that appear in a minimum proportion of transactions, use `itemFrequencyPlot()` with the `support` parameter:

```
> itemFrequencyPlot(groceries, support = 0.1)
```

As shown in the following plot, this results in a histogram showing the eight items in the groceries data with at least 10 percent support:

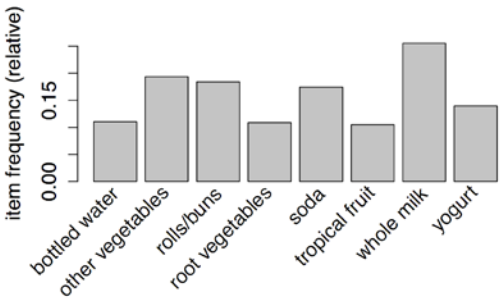

Figure 8.4: Support levels for all grocery items in at least 10 percent of transactions

If you would rather limit the plot to a specific number of items, use `itemFrequencyPlot()` with the `topN` parameter:

```
> itemFrequencyPlot(groceries, topN = 20)
```

The histogram is then sorted by decreasing support, as shown in the following diagram for the top 20 items in the groceries data:

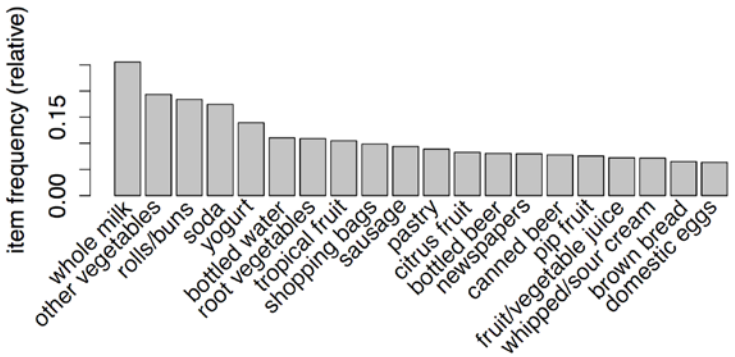

Figure 8.5: Support levels for the top 20 grocery items

Visualizing the transaction data – plotting the sparse matrix

In addition to looking at specific items, it's also possible to obtain a bird's-eye view of the entire sparse matrix using the `image()` function. Of course, because the matrix itself is very large, it is usually best to request a subset of the entire matrix. The command to display the sparse matrix for the first five transactions is as follows:

```
> image(groceries[1:5])
```

The resulting diagram depicts a matrix with five rows and 169 columns, indicating the five transactions and 169 possible items we requested. Cells in the matrix are filled with black for transactions (rows) where the item (column) was purchased.

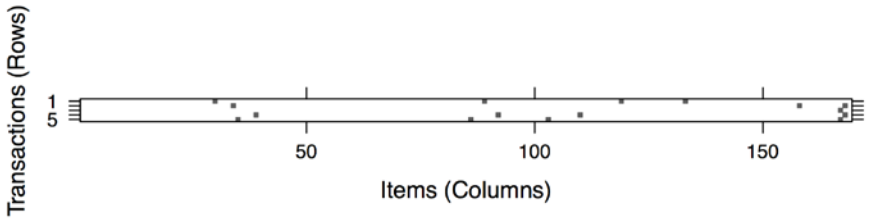

Figure 8.6: A visualization of the sparse matrix for the first five transactions

Although the preceding diagram is small and may be slightly hard to read, you can see that the first, fourth, and fifth transactions contained four items each, since their rows have four cells filled in. On the right side of the diagram, you can also see that rows three and five, and rows two and four, share an item in common.

This visualization can be a useful tool for exploring the transactional data. For one, it may help with the identification of potential data issues. Columns that are filled all the way down could indicate items that are purchased in every transaction—a problem that could arise, perhaps, if a retailer's name or identification number was inadvertently included in the transaction dataset.

Additionally, patterns in the diagram may help reveal interesting segments of transactions and items, particularly if the data is sorted in interesting ways. For example, if the transactions are sorted by date, patterns in the black dots could reveal seasonal effects in the number or types of items purchased. Perhaps around Christmas or Hanukkah, toys are more common; around Halloween, perhaps candies become popular. This type of visualization could be especially powerful if the items were also sorted into categories. In most cases, however, the plot will look fairly random, like static on a television screen.

Keep in mind that this visualization will not be as useful for extremely large transaction databases because the cells will be too small to discern. Still, by combining it with the `sample()` function, you can view the sparse matrix for a randomly sampled set of transactions. The command to create a random selection of 100 transactions is as follows:

```
> image(sample(groceries, 100))
```

This creates a matrix diagram with 100 rows and 169 columns:

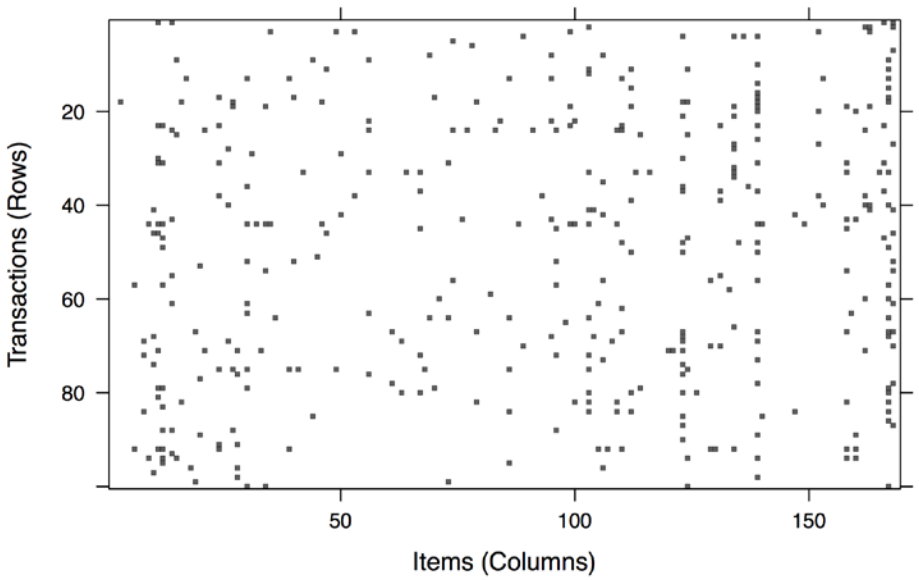

Figure 8.7: A visualization of the sparse matrix for 100 randomly-selected transactions

A few columns seem fairly heavily populated, indicating some very popular items at the store. However, the distribution of dots seems overall fairly random. Given nothing else of note, let's continue with our analysis.

Step 3 – training a model on the data

With data preparation complete, we can now work at finding associations among shopping cart items. We will use an implementation of the Apriori algorithm in the `arules` package we've been using for exploring and preparing the groceries data. You'll need to install and load this package if you have not done so already. The following table shows the syntax for creating sets of rules with the `apriori()` function:

Association rule syntax

using the `apriori()` function in the `arules` package

Finding association rules:

```
myrules <- apriori(data = mydata, parameter =
  list(support = 0.1, confidence = 0.8, minlen = 1))
```

- `data` is a sparse item matrix holding transactional data
- `support` specifies the minimum required rule support
- `confidence` specifies the minimum required rule confidence
- `minlen` specifies the minimum required rule items

The function will return a rules object storing all rules that meet the minimum criteria.

Examining association rules:

```
inspect(myrules)
```

- `myrules` is a set of association rules from the `apriori()` function

This will output the association rules to the screen. Vector operators can be used on myrules to choose a specific rule or rules to view.

Example:

```
groceryrules <- apriori(groceries, parameter =
 list(support = 0.01, confidence = 0.25, minlen = 2))
inspect(groceryrules[1:3])
```

Although running the `apriori()` function is straightforward, there can sometimes be a fair amount of trial and error needed to find the `support` and `confidence` parameters that produce a reasonable number of association rules. If you set these levels too high, then you might find no rules, or might find rules that are too generic to be very useful. On the other hand, a threshold too low might result in an unwieldy number of rules. Worse, the operation might take a very long time or run out of memory during the learning phase.

On the groceries data, using the default settings of `support = 0.1` and `confidence = 0.8` results in a set of zero rules:

```
> apriori(groceries)
set of 0 rules
```

Obviously, we need to widen the search a bit.

If you think about it, this outcome should not have been terribly surprising. Because `support = 0.1` by default, in order to generate a rule, an item must have appeared in at least *0.1 * 9,385 = 938.5* transactions. Since only eight items appeared this frequently in our data, it's no wonder we didn't find any rules.

One way to approach the problem of setting a minimum support is to think about the smallest number of transactions needed before you would consider a pattern interesting. For instance, you could argue that if an item is purchased twice a day (about 60 times in a month of data) then it may be important. From there, it is possible to calculate the support level needed to find only rules matching at least that many transactions. Since 60 out of 9,835 equals 0.006, we'll try setting the support there first.

Setting the minimum confidence involves a delicate balance. On the one hand, if the confidence is too low, then we might be overwhelmed with a large number of unreliable rules—such as dozens of rules indicating items commonly purchased with batteries. How would we know where to target our advertising budget then? On the other hand, if we set the confidence too high, then we will be limited to rules that are obvious or inevitable—like the fact that a smoke detector is always purchased in combination with batteries. In this case, moving the smoke detectors closer to the batteries is unlikely to generate additional revenue, since the two items were already almost always purchased together.

The appropriate minimum confidence level depends a great deal on the goals of your analysis. If you start with a conservative value, you can always reduce it to broaden the search if you aren't finding actionable intelligence.

We'll start with a confidence threshold of 0.25, which means that in order to be included in the results, the rule has to be correct at least 25 percent of the time. This will eliminate the most unreliable rules, while allowing some room for us to modify behavior with targeted promotions.

We are now ready to generate some rules. In addition to the minimum `support` and `confidence` parameters, it is helpful to set `minlen = 2` to eliminate rules that contain fewer than two items. This prevents uninteresting rules from being created simply because the item is purchased frequently, for instance, `{} => whole milk`. This rule meets the minimum support and confidence because whole milk is purchased in over 25 percent of transactions, but it isn't a very actionable insight.

The full command for finding a set of association rules using the Apriori algorithm is as follows:

```
> groceryrules <- apriori(groceries, parameter = list(support =
                                0.006, confidence = 0.25, minlen = 2))
```

This saves our rules in a rules object, which we can peek into by typing its name:

```
> groceryrules
set of 463 rules
```

Our `groceryrules` object contains a set of 463 association rules. To determine whether any of them are useful, we'll have to dig deeper.

Step 4 – evaluating model performance

To obtain a high-level overview of the association rules, we can use `summary()` as follows. The rule length distribution tells us how many rules have each count of items. In our rule set, 150 rules have only two items, while 297 have three, and 16 have four. The summary statistics associated with this distribution are also provided in the output:

```
> summary(groceryrules)
set of 463 rules

rule length distribution (lhs + rhs):sizes
  2   3   4
150 297  16

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   2.000   3.000   2.711   3.000   4.000
```

As noted in the previous output, the size of the rule is calculated as the total of both the left-hand side (`lhs`) and right-hand side (`rhs`) of the rule. This means that a rule like `{bread} => {butter}` is two items and `{peanut butter, jelly} => {bread}` is three.

Next, we see the summary statistics of the rule quality measures: `support`, `confidence`, and `lift`. The `support` and `confidence` measures should not be very surprising, since we used these as selection criteria for the rules. We might be alarmed if most or all of the rules had `support` and `confidence` very near the minimum thresholds, as this would mean that we may have set the bar too high.

This is not the case here, as there are many rules with much higher values of each:

```
summary of quality measures:
    support           confidence          lift
 Min.   :0.006101   Min.   :0.2500   Min.   :0.9932
 1st Qu.:0.007117   1st Qu.:0.2971   1st Qu.:1.6229
 Median :0.008744   Median :0.3554   Median :1.9332
 Mean   :0.011539   Mean   :0.3786   Mean   :2.0351
 3rd Qu.:0.012303   3rd Qu.:0.4495   3rd Qu.:2.3565
 Max.   :0.074835   Max.   :0.6600   Max.   :3.9565
```

The third column is a metric we have not considered yet. The **lift** of a rule measures how much more likely one item or itemset is to be purchased relative to its typical rate of purchase, given that you know another item or itemset has been purchased. This is defined by the following equation:

$$\text{lift}(X \rightarrow Y) = \frac{\text{confidence}(X \rightarrow Y)}{\text{support}(Y)}$$

Unlike confidence, where the item order matters, *lift(X → Y)* is the same as *lift(Y → X)*.

For example, suppose at a grocery store most people purchase milk and bread. By chance alone, we would expect to find many transactions with both milk and bread. However, if *lift(milk → bread)* is greater than one, this implies that the two items are found together more often than expected by chance alone. A large lift value is therefore a strong indicator that a rule is important and reflects a true connection between the items.

In the final section of the `summary()` output, we receive mining information, telling us about how the rules were chosen. Here, we see that the groceries data, which contained 9,835 transactions, was used to construct rules with a minimum support of 0.006 and minimum confidence of 0.25:

```
mining info:
      data  transactions support confidence
 groceries          9835   0.006       0.25
```

We can take a look at specific rules using the `inspect()` function. For instance, the first three rules in the `groceryrules` object can be viewed as follows:

```
> inspect(groceryrules[1:3])
```

```
  lhs                 rhs                    support confidence     lift
1 {potted plants} => {whole milk}       0.006914082  0.4000000 1.565460
2 {pasta}         => {whole milk}       0.006100661  0.4054054 1.586614
3 {herbs}         => {root vegetables} 0.007015760  0.4312500 3.956477
```

The first rule can be read in plain language as "if a customer buys potted plants, they will also buy whole milk." With a support of about 0.007 and confidence of 0.400, we can determine that this rule covers about 0.7 percent of transactions and is correct in 40 percent of purchases involving potted plants. The lift value tells us how much more likely a customer is to buy whole milk relative to the average customer, given that he or she bought a potted plant. Since we know that about 25.6 percent of customers bought whole milk (`support`), while 40 percent of customers buying a potted plant bought whole milk (`confidence`), we can compute the lift as *0.40 / 0.256 = 1.56*, which matches the value shown.

Note that the column labeled `support` indicates the support for the rule, not the support for the `lhs` or `rhs` alone.

In spite of the fact that the confidence and lift are high, does *{potted plants} → {whole milk}* seem like a very useful rule? Probably not, as there doesn't seem to be a logical reason why someone would be more likely to buy milk with a potted plant. Yet our data suggests otherwise. How can we make sense of this fact?

A common approach is to take the association rules and divide them into the following three categories:

- Actionable
- Trivial
- Inexplicable

Obviously, the goal of a market basket analysis is to find **actionable** rules that provide a clear and useful insight. Some rules are clear and others are useful; it is less common to find a combination of both of these factors.

So-called **trivial** rules include any rules that are so obvious that they are not worth mentioning—they are clear, but not useful. Suppose you are a marketing consultant being paid large sums of money to identify new opportunities for cross-promoting items. If you report the finding that *{diapers} → {formula}*, you probably won't be invited back for another consulting job.

Trivial rules can also sneak in disguised as more interesting results. For instance, say you found an association between a particular brand of children's cereal and a certain DVD movie. This finding is not very insightful if the movie's main character is on the front of the cereal box.

Rules are **inexplicable** if the connection between the items is so unclear that figuring out how to use the information is impossible or nearly impossible. The rule may simply be a random pattern in the data, for instance, a rule stating that *{pickles} → {chocolate ice cream}* may be due to a single customer whose pregnant wife had regular cravings for strange combinations of foods.

The best rules are the hidden gems—the undiscovered insights that only seem obvious once discovered. Given enough time, one could evaluate each and every rule to find the gems. However, the data scientists working on the analysis may not be the best judge of whether a rule is actionable, trivial, or inexplicable. Consequently, better rules are likely to arise via collaboration with the domain experts responsible for managing the retail chain, who can help interpret the findings. In the next section, we'll facilitate such sharing by employing methods for sorting and exporting the learned rules so that the most interesting results float to the top.

Step 5 – improving model performance

Subject matter experts may be able to identify useful rules very quickly, but it would be a poor use of their time to ask them to evaluate hundreds or thousands of rules. Therefore, it's useful to be able to sort the rules according to different criteria, and get them out of R in a form that can be shared with marketing teams and examined in more depth. In this way, we can improve the performance of our rules by making the results more actionable.

Sorting the set of association rules

Depending upon the objectives of the market basket analysis, the most useful rules might be those with the highest support, confidence, or lift. The `arules` package includes a `sort()` function that can be used to reorder the list of rules so that those with the highest or lowest values of the quality measure come first.

To reorder the `groceryrules` object, we can `sort()` while specifying a value of `"support"`, `"confidence"`, or `"lift"` to the `by` parameter. By combining the sort with vector operators, we can obtain a specific number of interesting rules. For instance, the best five rules according to the `lift` statistic can be examined using the following command:

```
> inspect(sort(groceryrules, by = "lift")[1:5])
```

The output is as follows:

```
  lhs                     rhs                      support confidence     lift
1 {herbs}              => {root vegetables}    0.007015760  0.4312500 3.956477
2 {berries}            => {whipped/sour cream} 0.009049314  0.2721713 3.796886
3 {other vegetables,
   tropical fruit,
   whole milk}         => {root vegetables}    0.007015760  0.4107143 3.768074
4 {beef,
   other vegetables}   => {root vegetables}    0.007930859  0.4020619 3.688692
5 {other vegetables,
   tropical fruit}     => {pip fruit}          0.009456024  0.2634561 3.482649
```

These rules appear to be more interesting than the ones we looked at previously. The first rule, with a `lift` of about 3.96, implies that people who buy herbs are nearly four times more likely to buy root vegetables than the typical customer—perhaps for a stew of some sort? Rule two is also interesting. Whipped cream is over three times more likely to be found in a shopping cart with berries versus other carts, suggesting perhaps a dessert pairing?

By default, the sort order is decreasing, meaning the largest values come first. To reverse this order, add an additional parameter `decreasing = FALSE`.

Taking subsets of association rules

Suppose that given the preceding rule, the marketing team is excited about the possibilities of creating an advertisement to promote berries, which are now in season. Before finalizing the campaign, however, they ask you to investigate whether berries are often purchased with other items. To answer this question, we'll need to find all the rules that include berries in some form.

The `subset()` function provides a method for searching for subsets of transactions, items, or rules. To use it to find any rules with berries appearing in the rule, use the following command. This will store the rules in a new object named `berryrules`:

```
> berryrules <- subset(groceryrules, items %in% "berries")
```

We can then inspect the rules as we had done with the larger set:

```
> inspect(berryrules)
```

The result is the following set of rules:

```
  lhs          rhs                     support confidence     lift
1 {berries} => {whipped/sour cream} 0.009049314  0.2721713 3.796886
2 {berries} => {yogurt}             0.010574479  0.3180428 2.279848
3 {berries} => {other vegetables}   0.010269446  0.3088685 1.596280
4 {berries} => {whole milk}         0.011794611  0.3547401 1.388328
```

There are four rules involving berries, two of which seem to be interesting enough to be called actionable. In addition to whipped cream, berries are also purchased frequently with yogurt—a pairing that could serve well for breakfast or lunch, as well as dessert.

The `subset()` function is very powerful. The criteria for choosing the subset can be defined with several keywords and operators:

- The keyword `items`, explained previously, matches an item appearing anywhere in the rule. To limit the subset to where the match occurs only on the left-hand side or right-hand side, use `lhs` or `rhs` instead.
- The operator `%in%` means that at least one of the items must be found in the list you defined. If you wanted any rules matching either berries or yogurt, you could write `items %in% c("berries", "yogurt")`.
- Additional operators are available for partial matching (`%pin%`) and complete matching (`%ain%`). Partial matching allows you to find both citrus fruit and tropical fruit using one search: `items %pin% "fruit"`. Complete matching requires that all listed items are present. For instance, items `%ain% c("berries", "yogurt")` finds only rules with both berries and yogurt.
- Subsets can also be limited by `support`, `confidence`, or `lift`. For instance, `confidence > 0.50` would limit the rules to those with confidence greater than 50 percent.
- Matching criteria can be combined with standard R logical operators such as AND (`&`), OR (`|`), and NOT (`!`).

Using these options, you can limit the selection of rules to be as specific or general as you would like.

Saving association rules to a file or data frame

To share the results of your market basket analysis, you can save the rules to a CSV file with the `write()` function. This will produce a CSV file that can be used in most spreadsheet programs, including Microsoft Excel:

```
> write(groceryrules, file = "groceryrules.csv",
          sep = ",", quote = TRUE, row.names = FALSE)
```

Sometimes it is also convenient to convert the rules into an R data frame. This can be accomplished using the `as()` function, as follows:

```
> groceryrules_df <- as(groceryrules, "data.frame")
```

This creates a data frame with the rules in factor format, and numeric vectors for support, confidence, and lift:

```
> str(groceryrules_df)
'data.frame':    463 obs. of 4 variables:
 $ rules     : Factor w/ 463 levels "{baking powder} => {other
vegetables}",..: 340 302 207 206 208 341 402 21 139 140 ...
 $ support   : num  0.00691 0.0061 0.00702 0.00773 0.00773 ...
 $ confidence: num  0.4 0.405 0.431 0.475 0.475 ...
 $ lift      : num  1.57 1.59 3.96 2.45 1.86 ...
```

Saving the rules to a data frame may be useful if you want to perform additional processing on the rules or need to export them to another database.

Summary

Association rules are used to find useful insight in the massive transaction databases of large retailers. As an unsupervised learning process, association rule learners are capable of extracting knowledge from large databases without any prior knowledge of what patterns to seek. The catch is that it takes some effort to reduce the wealth of information into a smaller and more manageable set of results. The Apriori algorithm, which we studied in this chapter, does so by setting minimum thresholds of interestingness, and reporting only the associations meeting these criteria.

We put the Apriori algorithm to work while performing a market basket analysis for a month's worth of transactions at a modestly sized supermarket. Even in this small example, a wealth of associations was identified. Among these, we noted several patterns that may be useful for future marketing campaigns. The same methods we applied are used at much larger retailers on databases many times this size, and can also be applied to projects outside of a retail setting.

In the next chapter, we will examine another unsupervised learning algorithm. Just like association rules, it is intended to find patterns within data. But unlike association rules that seek groups of related items or features, the methods in the next chapter are concerned with finding connections and relationships among the examples.