12 Association Rules for Market Basket Analysis

Many firms compile records of customer transactions. These data sets take diverse forms including products that are purchased together, services that are tracked over time in a customer relationship management (CRM) system, sequences of visits and actions on a Web site, and records of customer support calls. These records are very valuable to marketers and inform us about customers' purchasing patterns, ways in which we might optimize pricing or inventory given the purchase patterns, and relationships between the purchases and other customer information.

Such records may comprise an enormous number of data points yet with relatively little information in each observation. This means that simple analyses such as correlation and linear regression are not applicable because those methods assume complete or near-complete measurement for each case. For example, consider the number of products in a typical supermarket. Most items are not purchased with most other items in any transaction because there are so many possible combinations.

In this chapter we examine a strategy to extract insight from transactions and co-occurrence data: *association rule* mining. Association rule analysis attempts to find sets of informative patterns from large, sparse data sets. We demonstrate association rules using a real data set of more than 80,000 market basket transactions with 16,000 unique items [20]. We then examine how rule mining is potentially useful with non-transactional data and we use association rules to explore patterns in the subscription data from Chap. 5.

We develop the methods here from an *exploratory* point of view, to gain insight and form hypotheses about relationships in the data. Although it is out of scope for this chapter, if one is interested to demonstrate that the insights apply to new data or are stable over time, the same methods might be used with split samples and replication techniques (see Kuhn and Johnson [97] for an introduction to such approaches in general).

C. Chapman, E.M. Feit, *R for Marketing Research and Analytics*, Use R!,
DOI 10.1007/978-3-319-14436-8_12

12.1 The Basics of Association Rules

The basic idea of association rule mining is this: when events occur together more often than one would expect from their individual rates of occurrence, such co-occurrence is an interesting pattern. For example, consider sales of sweet relish and hot dogs (summertime treats in the USA). Imagine that hot dogs are sold in 5 % of supermarket transactions during a summer month, while relish is sold in 3 % of transactions. Are they related?

Suppose we just take the data for every sale that includes hot dogs, which is 5 % of transactions. If the proportion of those hot dog sales that have relish is 3 %, then there is no relationship because that is what we would expect for relish from the overall data, regardless of what else is sold. However, if relish is sold in 25 % of the transactions that have hot dogs, that is quite different than the base rate and is evidence of an association.

There are some terms to understand for association rules. An *association* is simply the co-occurrence of two or more things. Hot dogs might be positively associated with relish, hot dog buns, soda, potato chips, and ketchup. An association is not necessarily strong. In a store such as Costco that sells everything from hot dogs to (sometimes) grand pianos, everything sold is associated with everything else but most of those associations are weak. A *set of items* is a group of one or more items, and might be written as {item1, item2, ...}. For instance, a set might be {relish} or {hot dogs, soda, potato chips}.

A *transaction* is a set of items that co-occur in an observation. In marketing, a common transaction unit is the *market basket*, the set of things that are purchased or considered for purchase at one time. Any data points that co-occur are considered to be a transaction, even if using the term "transaction" seems unusual in the context. For example, the set of web pages that a user visits during a session would be a transaction in this sense.

A *rule* expresses the incidence across transactions of one set of items as a *condition* of another set of items. The association of relish, conditional on hot dogs, is expressed in the rule {relish} $\Rightarrow$ {hot dogs}. Rules may express the relationship of multiple items; for instance, {relish, ketchup, mustard, potato chips} $\Rightarrow$ {hot dogs, hamburger patties, hot dog buns, soda, beer}. A condition in this sense does not imply a causal relationship, only an association of some strength, whether strong or weak.

12.1.1 Metrics

Association rules are expressed with a few common metrics that reflect the rules of conditional probability. The *support* for a set of items is the proportion of all transactions that contain the set. If {hot dogs, soda} appears in 10 out of 200 transactions, then $support(\{hotdogs, soda\}) = 0.05$. It does not matter if those 10 transactions contain other items; support is defined separately for every unique set of items.

Confidence is the support for the co-occurrence of all items in a rule, conditional on the support for the left-hand set alone. Thus, $confidence(X \Rightarrow Y) = support(X \cap Y)/support(X)$ (where "$\cap$" means "*and*"). How does that work? Consider the rule {relish} $\Rightarrow$ {hot dogs}. If {relish} occurs in 1 % of transactions (in other words, $support(\{relish\}) = 0.01$) and {relish, hot dogs} appears in 0.5 %, then $confidence(\{relish\} \Rightarrow \{hotdogs\}) = 0.005/0.1 = 0.5$. In other words, hot dogs appear alongside relish 50 % of the time that relish appears.

Note that "confidence" in this context carries no implication about hypothesis testing, confidence intervals, or the like; it is only a measure of conditional association. Confidence is also not symmetric; unless $support(X) = support(Y)$, $confidence(X \Rightarrow Y) \neq confidence(Y \Rightarrow X)$.

Perhaps the most popular measure is *lift*, the support of a set conditional on the joint support of each element, or $lift(X \Rightarrow Y) = support(X \cap Y)/(support(X)support(Y))$. To continue the hot dog example, if $support(\{relish\}) = 0.01$, $support(\{hotdogs\}) = 0.01$, and $support(\{relish, hotdogs\}) = 0.005$, then $lift(\{relish \Rightarrow hotdogs\}) = 0.005/(0.01 * 0.01) = 50$. In other words, the combination {relish, hot dogs} occurs 50 times more often than we would expect if the two items were independent.

These three measures tell us different things. When we search for rules we wish to exceed a minimum threshold on each: to find item sets that occur relatively frequently in transactions (*support*), that show strong conditional relationships (*confidence*), and that are more common than chance (*lift*). As we will see, in practice an analyst sets the level of required support to a value such as 0.01, 0.10, 0.20, or so forth as is meaningful and useful for the business in consideration of the data characteristics (such as the size of the item set). Similarly, the level of required confidence might be high (such as 0.8) or low (such as 0.2) depending on the data and business. For lift, higher values are generally better and certainly should be above 1.0, although one must be mindful of outliers with huge lift.

We use the R package `arules` to illustrate association rules [71]. `arules` encapsulates many popular methods for mining associations and provides extensions for visualization [69]. Readers who are interested in the algorithms that generate association rules should review the references in the primary `arules` documentation [70, 71].

12.2 Retail Transaction Data: Market Baskets

The first two data sets we examine contain supermarket transaction data. We first examine a small data set that is included with the `arules` package. This data set is useful despite its small size because the items are labeled with category names, making them easier to read. Then we turn to a larger data set from a supermarket chain whose data is disguised but is more typical of large data sets.

12.2.1 Example Data: `Groceries`

We illustrate the general concepts of association rules with the `Groceries` data set in the `arules` package. This data set comprises lists of items purchased together (that is, market baskets), where the individual items have been recorded as category labels instead of product names. You should install the `arules` and `arulesViz` packages before proceeding.

We load the package and data, and then check the data as follows:

```
> library(arules)
> data("Groceries")
> summary(Groceries)
transactions as itemMatrix in sparse format with
 9835 rows (elements/itemsets/transactions) and
 169 columns (items) and a density of 0.02609146
...
> inspect(head(Groceries, 3))
  items
1 {citrus fruit,
   semi-finished bread,
   margarine,
   ready soups}
2 {tropical fruit,
   yogurt,
   coffee}
3 {whole milk}
```

The `summary()` shows us that the data comprise 9,835 transactions with 169 unique items. Using `inspect(head(Groceries))` we see a few examples from the baskets. For example, the second transaction includes fruit, yogurt, and coffee, while the third transaction is just a container of milk. In this output, notice that the item sets are structured with brackets, a visual clue that they reflect a new "transactions" data type that we examine in more detail below.

We now use `apriori(data, parameters=...)` to find association rules with the "apriori" algorithm [17, 71]. At a conceptual level, the apriori algorithm searches through the item sets that occur frequently in a list of transactions. For each item set, it evaluates the various possible rules that express associations among the items at or above a particular level of support, and then retains the rules that show confidence above some threshold value [16].

To control the extent that `apriori()` searches, we use the `parameter=list()` control to instruct the algorithm to search rules that have a minimum `support` of 0.01 (1 % of transactions) and extract the ones that further demonstrate a minimum `confidence` of 0.3. The resulting rule set is assigned to the `groc.rules` object:

```
> groc.rules <- apriori(Groceries, parameter=list(supp=0.01, conf=0.3,
+                                                  target="rules"))

parameter specification:
 confidence minval smax arem  aval originalSupport support minlen maxlen target
      ext
        0.3    0.1    1 none FALSE            TRUE    0.01      1     10  rules
     FALSE

algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

apriori - find association rules with the apriori algorithm
version 4.21 (2004.05.09)        (c) 1996-2004   Christian Borgelt
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[169 item(s), 9835 transaction(s)] done [0.01s].
sorting and recoding items ... [88 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 3 4 done [0.00s].
writing ... [125 rule(s)] done [0.00s].
creating S4 object  ... done [0.00s].
```

The rules have been found and saved to an object that we shall inspect in a moment. Note that the values for the `support` and `confidence` parameters are found largely by experience (in other words, by trial and error) and should be expected to vary from industry to industry and data set to data set. We arrived at the values of `support=0.01` and `confidence=0.3` after finding that they resulted in a modest number of rules suitable for an example. In real cases, you would adapt those values to your data and business case (we will say more about this as we examine additional data sets).

To interpret the results of `apriori()` above, there are two key things to examine. First, check the number of *items* going into the rules, which is shown on the output line "`sorting and recoding items ...`" and in this case tells us that the rules found are using 88 of the total number of items. If this number is too small (only a tiny set of your items) or too large (almost all of them), then you might wish to adjust the support and confidence levels.

Next, check the number of rules found, as indicated on the "`writing ...`" line. In this case, the algorithm found 125 rules. Once again, if this number is too low, it suggests the need to lower the support or confidence levels; if it is too high (such as many more rules than items), you might increase the support or confidence levels.

Once we have a rule set from `apriori()`, we use `inspect(rules)` to examine the association rules. The complete list of 125 from above is too long to examine here, so we select a `subset` of them with high lift, `lift > 3`. We find that five of the rules in our set have lift greater than 3.0:

```
> inspect(subset(groc.rules, lift > 3))
  lhs                     rhs                    support confidence     lift
1 {beef}              => {root vegetables}  0.01738688  0.3313953 3.040367
2 {citrus fruit,
   root vegetables}  => {other vegetables} 0.01037112  0.5862069 3.029608
3 {citrus fruit,
   other vegetables} => {root vegetables}  0.01037112  0.3591549 3.295045
4 {tropical fruit,
   root vegetables}  => {other vegetables} 0.01230300  0.5845411 3.020999
5 {tropical fruit,
   other vegetables} => {root vegetables}  0.01230300  0.3427762 3.144780
```

The first rule tells us that if a transaction contains {beef} then it is also relatively more likely to contain {root vegetables}—a category that we assume includes items such as potatoes and onions. That combination appears in 1.7 % of baskets ("support"), and the lift tells us that combination is 3× more likely to occur together than one would expect from the individual rates of incidence alone.

A store might form several ideas on the basis of such information. For instance, the store might create a display for potatoes and onions near the beef counter to encourage shoppers who are examining beef to purchase those vegetables or consider recipes with them. It might also suggest putting coupons for beef in the root vegetable area, or featuring recipe cards somewhere in the store. We will see other ways to inspect such data and develop ideas later in this chapter.

12.2.2 Supermarket Data

We now investigate associations in a larger set of retail transaction data from a Belgian supermarket chain. This data set comprises market baskets of items purchased together, where each record includes arbitrarily numbered item numbers without item descriptions (to protect the chain's proprietary data). This data set is made publicly available by Brijs et al. [20].

First we use `readLines(url)` to get the data from the website where it is hosted:

```
> retail.raw <- readLines("http://fimi.ua.ac.be/data/retail.dat")
```

An alternative location on this book's website is the following (see Appendix D for more options):

```
> retail.raw <- readLines("http://goo.gl/FfjDAO")
```

As always, we check the `head`, `tail`, and `summary`:

```
> head(retail.raw)
[1] "0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 ... "
[2] "30 31 32 "
...
> tail(retail.raw)
...
```

```
[5] "39 48 2528 "
[6] "32 39 205 242 1393 "
> summary(retail.raw)
   Length     Class      Mode
    88162 character character
```

Each row in this object represents a single market basket of items purchased together. Within each row, the items have been assigned arbitrary numbers that simply start at 0 in the first transaction and add new item numbers as needed for all later transactions. The data comprise 88,162 transactions, where the first basket has 30 items (numbered 0–29, some truncated in the output here), the second has 3 items, and so forth. In the `tail()`, we see that the last market basket had 5 items, most of which—items 32, 39, 205, and 242—have low numbers reflecting that those particular items first appeared in transactions early in the data set.

In this text format, the data are not ready to mine; we must first split each of the transaction text lines into individual items. To do this, we use `strsplit(lines, " ")`. This command splits each line wherever there is a blank space character (`" "`) and saves the results to a list:

```
> retail.list <- strsplit(retail.raw, " ")
```

To label the individual transactions, we assign descriptive names using `names()` and `paste()`:

```
> names(retail.list) <- paste("Trans", 1:length(retail.list), sep="")
```

As usual, we check the data format again. Finally, we remove the `retail.raw` object that is no longer needed:

```
> str(retail.list)
List of 88162
 $ Trans1    : chr [1:30] "0" "1" "2" "3" ...
 $ Trans2    : chr [1:3] "30" "31" "32"
...
> library(car)
> some(retail.list)  # note: random sample; your results may vary
$Trans3742
[1] "488"  "1588" "2750" "2832" "4099"
...
> rm(retail.raw)
```

Using `str()` we confirm that the list has 88,162 entries and that individual entries look appropriate. `some()` samples a few transactions throughout the larger set for additional confirmation.

The transaction list could be used to find rules at this point, but we take an additional step to convert it to a formal *transactions* object, which enhances the ways we can work with the data and speeds up `arules` operations. To convert from a list to transactions, we cast the object using `as(..., "transactions")`:

```
> retail.trans <- as(retail.list, "transactions")   # takes a few seconds
> summary(retail.trans)
```

```
transactions as itemMatrix in sparse format with
 88162 rows (elements/itemsets/transactions) and
 16470 columns (items) and a density of 0.0006257289

most frequent items:
     39      48      38      32      41 (Other)
  50675   42135   15596   15167   14945  770058

element (itemset/transaction) length distribution:
sizes
   1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
3016 5516 6919 7210 6814 6163 5746 5143 4660 4086 3751 3285 2866 2620 2310
...
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    4.00    8.00   10.31   14.00   76.00
...
> rm(retail.list)  # no longer needed
```

Looking at the `summary()` of the resulting object, we see that the transaction-by-item matrix is 88,162 rows by 16,470 columns. Of those 1.4 billion intersections, only 0.06 % have positive data (*density*) because most items are not purchased in most transactions. Item 39 appears the most frequently and occurs in 50,675 baskets or more than half of all transactions. 3,016 of the transactions contain only a single item ("sizes" = 1) and the median basket size is 8 items.

12.3 Finding and Visualizing Association Rules

With the data in transaction format, we are ready to find rules. As we have seen briefly already, the `apriori(data, parameters=...)` command finds association rules [17]. For the Belgian supermarket data, we specify `parameter=list(...)` with values of minimum `support = 0.001` and minimum `confidence = 0.4`. We assign the resulting rules to a new object:

```
> retail.rules <- apriori(retail.trans, parameter=list(supp=0.001, conf=0.4))
parameter specification:
 confidence minval smax arem  aval originalSupport support minlen maxlen target
        ext
        0.4    0.1    1 none FALSE            TRUE   0.001      1     10  rules
      FALSE
...
set transactions ...[16470 item(s), 88162 transaction(s)] done [0.12s].
sorting and recoding items ... [2117 item(s)] done [0.02s].
creating transaction tree ... done [0.06s].
checking subsets of size 1 2 3 4 5 6 done [0.16s].
writing ... [5944 rule(s)] done [0.01s]. ...
```

This finds a set of 5,944 rules that exceed the required levels of support and confidence.

To get a sense of the rule distribution, we load the `arulesViz` package and then `plot()` the rule set, which charts the rules according to confidence (Y axis) by support (X axis) and scales the darkness of points to indicate lift. The commands are simply:

```
> library(arulesViz)
> plot(retail.rules)
```

The resulting chart is shown in Fig. 12.1. In that chart, we see that most rules involve item combinations that occur infrequently (that is, they have low support) while confidence is relatively smoothly distributed.

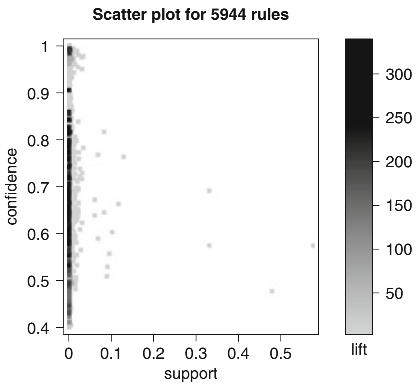


Fig. 12.1. Plotting a large set of rules for confidence (*Y axis*) by support (*X axis*) and lift (*shade*). There are a few rules in the upper left with exceptionally high confidence and lift.

Simply showing points is not very useful, and a key feature with `arulesViz` is *interactive plotting*. In Fig. 12.1 there are some rules in the upper left with high lift. We can use interactive plotting to inspect those rules. To do this, add `interactive=TRUE` to the `plot()` command:

```
> plot(retail.rules, interactive=TRUE)
```

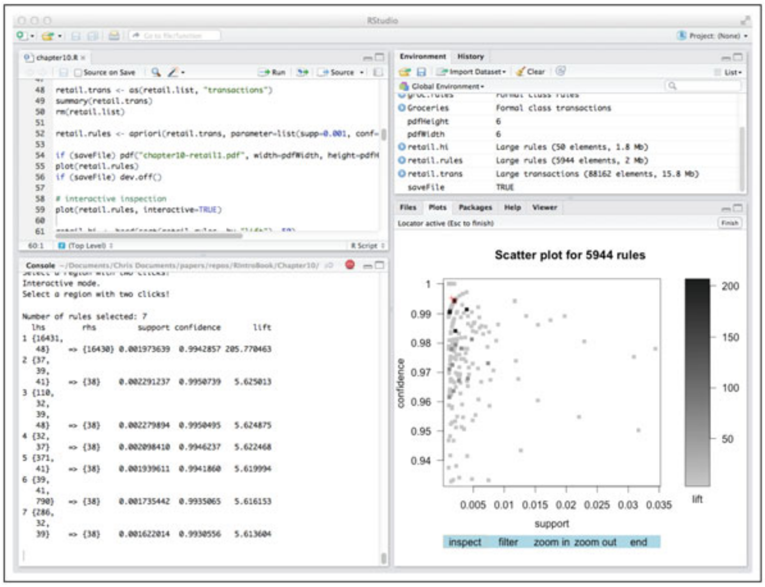
In interactive mode, you can examine regions of rules. To do so, click once in the plot window at one corner of the area of interest, and then click again at the opposite corner. You can use `zoom in` to magnify that region, or `inspect` to list the rules in the region. When finished, click `end`.

Figure 12.2 shows an interactive plotting session in RStudio where we seek rules with high lift. To get Fig. 12.2 we previously selected the upper left region as was shown in Fig. 12.1 and zoomed in on that region. Then we selected a few rules from the zoomed-in area and clicked `inspect` to display them in the console. There were seven rules in that subregion, as shown in the lower left console window. This revealed one exceptionally high lift rule:

```
  lhs         rhs          support  confidence       lift
1 {16431,
   48}     => {16430} 0.001973639   0.9942857 205.770463
```

This rule tells us that the combination {16431, 48} occurs in about 0.2 % of baskets (support=0.00197), and when it occurs it almost always includes {16430} (confidence=0.99). The combination occurs 200 times more often than we would expect from the individual incidence rates of {16431, 48} and {16430} considered separately (lift=205).

Such information could be used in various ways. If we pair the transactions with customer information, we could use this for targeted mailings or email suggestions. For items often sold together, we could adjust the price and margins together; for instance, to put one item on sale while increasing the price on the other. Or perhaps—only somewhat facetiously—the cashiers might ask customers, "Would you like a 16,430 with that?"



```
retail.trans <- as(retail.list, "transactions")
summary(retail.trans)
rm(retail.list)

retail.rules <- apriori(retail.trans, parameter=list(supp=0.001, conf=

if (saveFile) pdf("chapter10-retail1.pdf", width=pdfWidth, height=pdfH
plot(retail.rules)
if (saveFile) dev.off()

# interactive inspection
plot(retail.rules, interactive=TRUE)
```

```
Groceries        Formal class transactions
pdfHeight        6
pdfWidth         6
retail.hi        Large rules (50 elements, 1.8 Mb)
retail.rules     Large rules (5944 elements, 2 Mb)
retail.trans     Large transactions (88162 elements, 15.8 Mb)
saveFile         TRUE
```

```
Interactive mode.
Select a region with two clicks!

Number of rules selected: 7
  lhs          rhs         support confidence       lift
1 {16431,
   48}      => {16430} 0.001973639  0.9942857 205.770463
2 {37,
   39,
   41}      => {38}    0.002291237  0.9950739   5.625013
3 {110,
   32,
   39,
   48}      => {38}    0.002279894  0.9950495   5.624875
4 {32,
   37}      => {38}    0.002098410  0.9946237   5.622468
5 {371,
   41}      => {38}    0.001939611  0.9941860   5.619994
6 {39,
   41,
   790}     => {38}    0.001735442  0.9935065   5.616153
7 {286,
   32,
   39}      => {38}    0.001622014  0.9930556   5.613604
```

Fig. 12.2. Using `plot(..., interactive=TRUE)` to inspect rules of interest in interactive mode. In this screenshot from RStudio, we zoomed into a small region to `inspect()` a subgroup of the complete rule set. This reveals the selected rules in the console (*lower left window*).

12.3.1 Finding and Plotting Subsets of Rules

A common goal in market basket analysis is to find rules with high lift. We can find such rules easily by sorting the larger set of rules by lift. We extract the 50 rules with highest lift using `sort()` to order the rules by `lift` and taking 50 from the `head()`:

```
> retail.hi <- head(sort(retail.rules, by="lift"), 50)
> inspect(retail.hi)
   lhs       rhs         support  confidence     lift
1  {696}  => {699}  0.001032191  0.5833333 338.3410
2  {699}  => {696}  0.001032191  0.5986842 338.3410
3  {1818,
    3311,
    795}  => {1819} 0.001088905  0.9056604 318.1069
4  {3402} => {3535} 0.001417844  0.7062147 305.2024
...
```

Support and lift are identical for an item set regardless of the items' order within a rule (left-hand or right-hand side of the rule). Thus the first two rules—which include the same two items {696} and {699} on opposite sides of the conditional arrow—are identical for support and lift. However, confidence reflects direction because it computes occurrence the right-hand set conditional on the left-hand side set, and differs slightly for the first two rules.

A *graph* display of rules may be useful to seek higher level themes and patterns. We chart the top 50 rules by lift with `plot(..., method="graph")` and display rules as the intersection of items by adding the graph option, `control=list(type="item"))`:

```
> plot(retail.hi, method="graph", control=list(type="items"))
```

The resulting chart is shown in Fig. 12.3. Positioning of items on the graph may differ for your system, but the item clusters should be similar. Each circle there represents a rule with inbound arrows coming from items on the left-hand side of the rule and outbound arrows going to the right-hand side. The size (area) of the circle represents the rule's support, and shade represents lift (darker indicates higher lift).

Figure 12.3 shows several patterns of interest. Items 696 and 699 form a tight set; there are item clusters for {3402, 3535, 3537}, {309, 1080, 1269, 1378, 1379, 1380}, and so forth; and item 39 appears as a key item in two sets of items that otherwise do not overlap. By exploring sets of rules with various levels of lift and support, and with specific subsets of items (see the usage of `%in%` in `arules` help), an analyst may be able to find patterns that suggest interesting hypotheses and trends. We will see a further example of this for non-transactional data in Sect. 12.4 below.

12.3.2 Using Profit Margin Data with Transactions: An Initial Start

An analyst will often wish to combine market basket transactions and rules with other data; for instance, one might have information on item profitability (margin) or purchaser characteristics. In this section, we consider how to combine information on item cost and margin with transaction data.

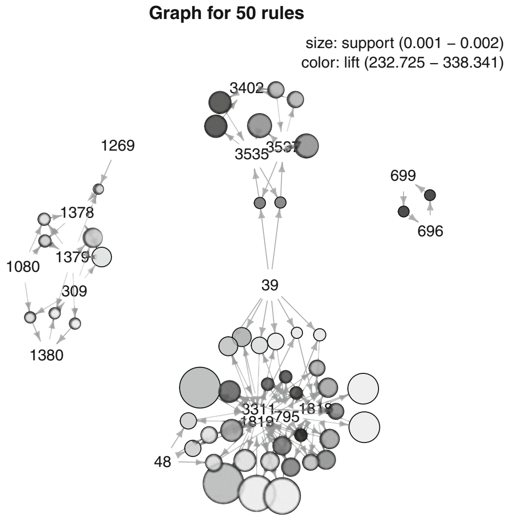


Fig. 12.3. A graph using `arulesViz` of the top 50 association rules mined from the retail market basket data set. There are four distinct sets of rules (*arrows and circular nodes*), each relating a set of 2–6 items (the integer ID numbers). These rules have *lift* of 232× or more in the retail shopping data.

How can we find the profit for a transaction? The answer may be complex and depend on the details of a firm and its transaction data. Because the Belgian supermarket data set does not include item price or cost, we simulate margin by items for illustration purposes. We assume that each item has a single margin value; if we had access to a firm's complete data, it would be better to use information about costs and prices by date, along with discounts and other adjustments to estimate margin more accurately.

To simulate per-item margin data we first compile a list of the item names that we need. We do this by converting the complete transaction set to `list` format and then using `unlist()` to gather the individual items from the many transactions into a single vector. We take the `unique()` values to remove duplicates, and then `sort()` them:

```
> retail.itemnames <- sort(unique(unlist(as(retail.trans, "list"))))
> head(retail.itemnames); tail(retail.itemnames)
[1] "0"     "1"     "10"    "100"   "1000"  "10000"
[1] "9994" "9995" "9996" "9997" "9998" "9999"
```

Items are not in numeric order because the item labels are character data and `sort()` orders them alphabetically; this poses no problem here.

Next we generate the simulated margin data with one value for each item, using `rnorm()` with a mean and standard deviation of 0.30 currency units (such as €0.30):

```
> set.seed(03870)
> retail.margin <- data.frame(margin=rnorm(length(retail.itemnames),
+                                          mean=0.30, sd=0.30))
> quantile(retail.margin$margin)
        0%        25%        50%        75%       100%
-1.1090452  0.1045897  0.3026245  0.5050533  1.5542344
```

We make those values indexable by item name by adding the list of items from above as the `rownames()` for the random numbers:

```
> rownames(retail.margin) <- retail.itemnames
> head(retail.margin); tail(retail.margin)
          margin
0      0.88340359
1      0.52964087
...
9999  0.6850124
> library(car); some(retail.margin)
          margin
12336  0.18504274
...
```

In this format, we can look up the margin for an item—or set of items—using the relevant item names. For example, we find the item margins and then their sum for the basket $\{39, 48\}$ as follows:

```
> retail.margin[c("39", "48"), ]
[1]  0.1217833 -0.2125105
> sum(retail.margin[c("39", "48"), ])
[1] -0.09072725
```

Item 39 has margin of 0.12, and the basket $\{39, 48\}$ has total margin of -0.09.

To find the margin for a complete transaction—in this case, transaction #3 from the Belgian data—there is one more step. We have to convert the transaction to *list* form to find the items in it using `as(..., "list")`, at which point we can look up the margins for those items:

```
> (basket.items <- as(retail.trans[3], "list")[[1]])
[1] "33" "34" "35"
> retail.margin[basket.items, ]
[1] 0.3817115 0.6131403 0.1979879
> sum(retail.margin[basket.items, ])
[1] 1.19284
```

12.3.3 Language Brief: A Function for Margin Using an Object's `class`*

This optional section expands on the margin example by writing a more complex function. Along the way we will see one way to use objects' *classes* and how to write more error-resistant code. If you do not wish to dive deeply into programming, you may safely skip this section.

12.3.3.1 Motivation

Using a simple index to look up the margin for items as we did above is not very satisfactory because it depends on the exact format of the data, such as the fact that it is given in a `list` format. If we ever change the format or wish to explore margin for some other kind of data, it is necessary to find any code where data is treated as a list and alter it. That process would be tedious and likely to introduce errors.

A better solution is to write a function to look up margins. With a function, we can perform more complex logic such as date lookups and volume or customer discounts. It also localizes all logic to a single place; if we call the function in each place that we need a margin lookup, we only need to change the procedure in one place.

In this section we create an initial working version of a more general lookup function. We also enhance the simple lookup capability in an important way: we make it work for transactions and rule sets as well as item names. A user may call the function with any of those data types and it will handle the data properly.

One way to make a function work for different kinds of input data is to use the R *class* system to determine the data type. (More advanced programmers may note that the approach here is a simple solution; a more complete solution—but well beyond the scope of this book—is to implement *S3* or *S4* methods for each data class that a function supports. For details on the various object-oriented programming paradigms in R, see [22, 62, 156, 163].)

Our function takes the form `retail.margsum(items, itemMargins)`, where `items` may be any of the following:

- A character vector of item names such as `c("39", "48")`, of class “`character`”
- One or more transactions such as `retail.trans` in our example above, of class “`transactions`”
- A set of rules such as `retail.hi` in our example above, of class “`rules`”

By checking the `class()`, our function is able to extract items appropriately from the data that a user provides, so the user will not have to extract item names from different kinds of objects.

Before inspecting the `retail.margsum()` code, we note that it has three key sections:

1. Convert the data we’re given to a list of item name sets
2. Check that those item names are in our margin data (`itemMargins`)
3. Look up the margins and sum them

Here is the complete code:

```
retail.margsum <- function(items, itemMargins) {
  # Input: "items" == item names, rules or transactions in arules format
  #        "itemMargins", a data frame of profit margin indexed by name
  # Output: look up the item margins, and return the sum
  library(arules)

  # check the class of "items" and coerce appropriately to an item list
  if (class(items) == "rules") {
    tmp.items <- as(items(items), "list")       # rules ==> item list
  } else if (class(items) == "transactions") {
    tmp.items <- as(items, "list")              # transactions ==> item list
  } else if (class(items) == "list") {
    tmp.items <- items                          # it's already an item list!
  } else if (class(items) == "character") {
    tmp.items <- list(items)                    # characters ==> item list
  } else {
    stop("Don't know how to handle margin for class ", class(items))
  }
  # make sure the items we found are all present in itemMargins
  good.items <- unlist(lapply(tmp.items, function (x)
                      all(unlist(x) %in% rownames(itemMargins))))

  if (!all(good.items)) {
    warning("Some items not found in rownames of itemMargins. ",
            "Lookup failed for element(s):\n",
            which(!good.items), "\nReturning only good values.")
    tmp.items <- tmp.items[good.items]
  }

  # and add them up
  return(unlist(lapply(tmp.items, function(x) sum(itemMargins[x, ]))))
}
```

We explain the code in detail below, but first let's see how it works. One way to use it is to find margin for an item set with simple item names:

```
> retail.margsum(c("39", "48"), retail.margin)
[1] -0.09072725
```

Another use is to find margin for each entry in a list with multiple, separate item sets:

```
> retail.margsum(list(t1=c("39", "45"), t2=c("31", "32")), retail.margin)
       t1        t2
0.9664982 0.2733963
```

It accepts one or more transaction objects:

```
> retail.margsum(retail.trans[101:103], retail.margin)
 Trans101  Trans102  Trans103
0.7171411 4.8989272 4.9470372
```

It also accepts sets of rules, such as our `retail.hi` set of the 50 highest list rules:

```
> retail.margsum(retail.hi, retail.margin)
 [1] 0.9609471 0.9609471 1.9327917 0.7084729 0.7084729 1.9327917 ...
...
[45] 0.1624291 0.5067865 0.5067865 0.5442604 0.5442604 0.6285698
```

It includes error detection. For instance, it gives an error in case of incorrect item names:

```
> retail.margsum(c("hello", "world"), retail.margin)  # error!
NULL
Warning message:
In retail.margsum(c("hello", "world"), retail.margin) :
  Some items not found in rownames of itemMargins. ...
```

In the above case, it returns a value of `NULL` as shown on the first line of the output because there was nothing valid to look up. However, if some of the data is bad while other parts are good, it finds whatever is possible:

```
> retail.margsum(list(a=c("39", "45"), b=c("hello", "world"), c=c("31", "32")),
+                retail.margin)    # only the first and third are OK
        a         c
0.9664982 0.2733963
Warning message:
...
```

In this case, the second element in the input is bad, so the function omits that and returns the sums for the other two item sets "a" and "c."

Now let's look at the function in detail to see how it works. In the first part of the code we convert the `items` input to proper types, by checking the `class()` and then applying an appropriate conversion:

```
# [ function excerpt, don't run on its own ]
  if (class(items) == "rules") {
    tmp.items <- as(items(items), "list")       # rules ==> item list
  } else if (class(items) == "transactions") {
    tmp.items <- as(items, "list")              # transactions ==> item list
...
  } else {
    stop("Don't know how to handle margin for class ", class(items))
  }
```

In this part of the code, we use the `if ... else if ...` construct in R to check types successively. It ends with a final `else` clause in case the data is a type the function cannot handle. In that case, it calls `stop(message)` to issue an error message to the user and exit the function.

The second part of our code checks that the sets of items are present in the `itemMargins` data:

```
# [ function excerpt, don't run on its own ]
  good.items <- unlist(lapply(tmp.items, function (x)
                     all(unlist(x) %in% rownames(itemMargins))))
  if (!all(good.items)) {
    warning("Some items not found in rownames of itemMargins. ",
            "Lookup failed for element(s):\n",
            which(!good.items), "\nReturning only good values.")
    tmp.items <- tmp.items[good.items]
  }
```

This short code block has a few crucial elements. First it uses an anonymous function to check that item names are present in `itemMargins`. It uses `%in%` to look up each name from a single list element (with the names extracted by `unlist(x)`) and then uses `all()` to make sure that every one of the names is found successfully (that is, that `all` of the `%in%` matches are `TRUE`). The result of this is a flag whether a given element of `tmp.items` is good or not.

Then we use the `unlist()` function a second time to convert the individual results from `lapply()` to a master vector, which indicates whether each individual element of `tmp.items` is good or not. Finally, if any of the individual item sets has an item that was not found (and therefore, using `!` for binary negation, `!all(good.items)` is `TRUE`) then we issue a `warning()` to the user, and retain only the good items for further processing. Unlike `stop()`, a function continues after a `warning()` to the user.

The third and final part of our code looks up the items and returns the sum of their margins:

```
# [ function excerpt, don't run on its own ]
  return(unlist(lapply(tmp.items, function(x) sum(itemMargins[x, ]))))
```

That line unpacks as follows, starting from the innermost part. An anonymous function looks up rows in `itemMargins`, and then sums them. Those rows `x` are determined by the surrounding `lapply()` that iterates over the individual sets of items that form the list `tmp.items`. Each member set of `tmp.items` has its items' margins summed. Finally, the line calls `unlist()` in order to convert the `lapply()` result—which is a list—to a more convenient vector.

But wait! That final, single line effectively delivers the whole purpose of the function. Why did we have to write so much else in the function? Isn't that needless complexity?

The answer depends on the circumstance, but this function exemplifies a common issue in programming: handling exceptions and doing error-checking is often the most complex part of a programming task. Just as getting data into shape is often the bulk of an analyst's work, much of a programmer's effort is to anticipate potential data problems when writing code. It is a good practice to include error-checking as we've done here. Don't assume your data will always be good; check it! You'll avoid many headaches for yourself and your colleagues.

Once the skeleton of a profit margin function is in place, an analyst will find many uses for it. For example, one might use it on transactions to find the most valuable customers, to find potential loss-leading items that are associated with other, higher margin items, to find money-losing associations, and so forth. A simple function of the kind here would be a proof of concept; a next step might be to increase its precision by including time series data, discounts, and other important factors specific to a firm and category.

12.4 Rules in Non-Transactional Data: Exploring Segments Again

There are many uses of association rules beyond retail transactions such as we considered above. The idea of a "transaction" broadly speaking is simply an observation of one or more data points that co-occur. For instance, when a user visits one or more web pages during a browsing session, the pages would constitute a transaction in this sense.

In the most general sense, one can consider any data points that occur together in a record—such as any variables observed for a customer, user, or survey respondent—to be a transaction. This means that association rules can be applied to other kinds of data such as general data frames (with some limitations that we'll discuss). In this section, we examine association rules as a way to explore consumer segmentation.

We use the simulated consumer segmentation data from Sect. 5.1.4. If you saved the data in that chapter (page 120), reload it now. We suggested a file destination as `file="~/segdf-Rintro-Ch5.RData"`. If you saved there, you can retrieve the data with:

```
> load("~/segdf-Rintro-Ch5.RData")
```

Alternatively, run the code in that chapter (Sects. 5.1.1–5.1.4) or download the file from this book's website:

```
> seg.df <- read.csv("http://goo.gl/qw303p")
```

After loading the data, check that it matches expectations:

```
> summary(seg.df)
      age            gender          income              kids          ownHome ...
 Min.   :19.26   Female:157   Min.    : -5183   Min.   :0.00   ownNo :159 ...
 1st Qu.:33.01   Male  :143   1st Qu.: 39656   1st Qu.:0.00   ownYes:141 ...
```

12.4.1 Language Brief: Slicing Continuous Data with `cut()`

Association rules work with *discrete* data yet `seg.df` includes three continuous (or quasi-continuous) variables: `age`, `income`, and `kids`. It's necessary to convert those to discrete factors to use with association rules in the `arules` package.

We could add factor variables as new columns appended to the original data frame. However, we use that data frame elsewhere in this book and thus prefer instead to make a copy and alter it:

```
> seg.fac <- seg.df
```

Now we replace `age`, `income`, and `kids` with recoded factors (specifically, using the `ordered` factor class to code these data as ordinal values). `cut(data, breaks, labels)` transforms numeric data to a factor variable. `breaks=...` specifies either the number of bins or specific cut points, and `labels=...` specifies the text for a factor's category labels. We transform `age` as follows:

```
> seg.fac$age <- cut(seg.fac$age,
+                    breaks=c(0,25,35,55,65,100),
+                    labels=c("19-24", "25-34", "35-54", "55-64", "65+"),
+                    right=FALSE, ordered_result=TRUE)
```

This recodes `age` from an integer value into an ordered factor with five levels: 19–24, 25–34, and so forth. The argument `right=FALSE` ensures that continuous values have closed intervals on the left, giving us $[25-34)$ instead of $(25-34]$. We set `ordered_result=TRUE` to specify that the resulting factor is ordinal. We check the data and see that the recode was successful:

```
> summary(seg.fac$age)
19-24 25-34 35-54 55-64   65+
   38    58   152    38    14
```

Next we convert `income` and `kids` similarly:

```
> seg.fac$income <- cut(seg.fac$income,
+                       breaks=c(-100000, 40000, 70000, 1000000),
+                       labels=c("Low", "Medium", "High"),
+                       right=FALSE, ordered_result=TRUE)
> seg.fac$kids <- cut(seg.fac$kids,
+                       breaks=c(0, 1, 2, 3, 100),
+                       labels=c("No kids", "1 kid", "2 kids", "3+ kids"),
+                       right=FALSE, ordered_result=TRUE)
> summary(seg.fac)
    age          gender        income          kids        ownHome     subscribe
 19-24: 38   Female:157   Low    : 77   No kids:121   ownNo :159   subNo :260
 25-34: 58   Male  :143   Medium:183   1 kid  : 70   ownYes:141   subYes: 40
...
```

All variables are now coded as categorical factors and the `seg.fac` data frame is suitable for exploring associations.

12.4.2 Exploring Segment Associations

A data frame in suitable discrete (factor) format can be converted to use in `arules` by using `as(..., "transactions")` to code it as transaction data:

```
> library(arules)
> library(arulesViz)
> seg.trans <- as(seg.fac, "transactions")
> summary(seg.trans)
transactions as itemMatrix in sparse format with
 300 rows (elements/itemsets/transactions) and
 22 columns (items) and a density of 0.3181818
...
```

Rules are generated in the same way as for market basket data. We use `apriori()` and specify `support=0.1` and `conf=0.4`. This finds 579 association rules:

```
> seg.rules <- apriori(seg.trans, parameter=list(support=0.1, conf=0.4,
+                                                 target="rules"))
> summary(seg.rules)
set of 579 rules ...
```

A default plot of the resulting `seg.rules` object is:

```
> plot(seg.rules)
```

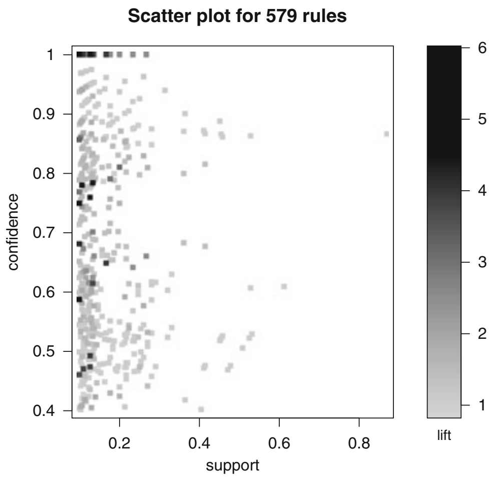


Fig. 12.4. The distribution of rules inferred from the segmentation data set.

This products Fig. 12.4, where we see a few rules with high confidence shown in the upper left region. If we add the `interactive=TRUE` option for `plot()` (not shown; see Sect. 12.3 for an explanation), we could explore those interactively to find the following rules with both high confidence and high lift:

```
> plot(seg.rules, interactive=T)
...
   lhs                        rhs                    support  confidence     lift
1  {age=19-24}             => {Segment=Urban hip} 0.1266667   1.0000000 6.000000
2  {age=19-24,
    income=Low}            => {Segment=Urban hip} 0.1266667   1.0000000 6.000000
3  {age=19-24,
    ownHome=ownNo}       => {Segment=Urban hip} 0.1000000  1.0000000 6.000000
4  {age=19-24,
    subscribe=subNo}     => {Segment=Urban hip} 0.1000000  1.0000000 6.000000
...
```

These show an association of `age` and other variables with membership in the Urban hip segment.

A graph plot visualizes clusters of rules to reveal higher-level patterns. We extract the top 35 highest-lift rules and visualize them as a `graph`:

```
> seg.hi <- head(sort(seg.rules, by="lift"), 35)
> inspect(seg.hi)
   lhs                    rhs                   support confidence     lift
1  {age=19-24}         => {Segment=Urban hip} 0.1266667  1.0000000 6.000000
...
> plot(seg.hi, method="graph", control=list(type="items")) # orientation varies
```

The resulting chart is shown in Fig. 12.5 (orientation of the chart may vary for you). There are two dominant clusters: a large cluster with many rules and relatively high lift that involve ages 19–24, no home ownership, lower income, and so forth; and a smaller cluster involving late middle-age consumers without kids in the travelers segment.

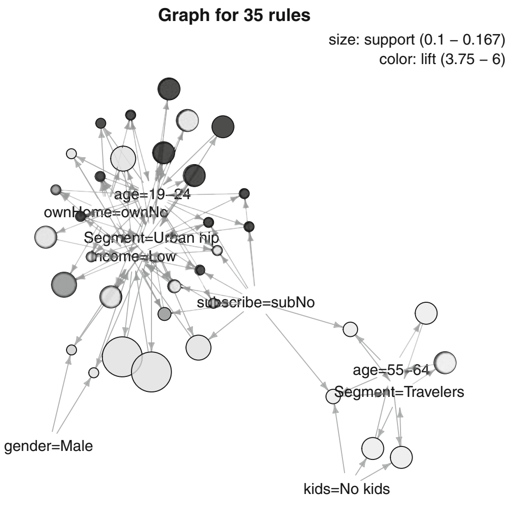


Fig. 12.5. Example of using a `graph` plot to explore rule clusters for the segmentation data set.

One might do further explorations by selecting additional sets of rules beyond the `head()` of the sorted rules. To do this, `sort()` the rules by lift (or other parameter as desired) and then index the rules you want. To examine the next 25 rules after the first 35 considered above:

```
> seg.next <- sort(seg.rules, by="lift")[36:60]
> plot(seg.next, method="graph", control=list(type="items")) # not shown
```

We omit the resulting chart in this case, which shows patterns involving factors such as the suburban mix segment and home ownership.

The patterns demonstrate that association rules can be useful to seek patterns in such non-transactional data. A key point is that this is primarily an *exploratory* exercise. It is useful if it reveals interesting patterns for further investigation. One should confirm any such inferences before drawing final conclusions.

12.5 Learning More*

An approachable text for association rules is Tan et al [149]. In that text, Chap. 6 discusses the fundamental concepts and algorithms of association rules, and Chap. 7 develops more advanced concepts and applications. Vipin Kuman, one of that text's authors, has published online materials related to the book and association rules, at `http://www-users.cs.umn.edu/~kumar/dmbook/index.php`.

The `arules` package is notable for its rich ecosystem of tools such as the `arulesViz` package that we used for charting. Other options include sequence mining and naive Bayes algorithms in addition to the standard apriori algorithm. For an overview of the `arules` ecosystem, see [70] and the vignettes that come with `arules`. The latest developments are available from the first author Michael Hahsler's site, `http://michael.hahsler.net/`.

12.6 Key Points

Association rules are a powerful way to explore the relationships in a data set. The following points summarize some key suggestions from this chapter.

- Association rules are commonly used with sparse data sets that have many observations but little information per observation. In marketing, this is typical of market baskets and similar transaction data. (Sect. 12.1)
- The `arules` package is the standard R package for association rules. `arules` provides support for handling sparse data and finding rules and the `arulesViz` package provides visualization methods.
- Core metrics for evaluating association rules are *support* (frequency), *confidence* (co-occurrence), and *lift* (co-occurrence above the rate of association by pure chance). There is no absolute value required of them except that lift should be somewhat greater than 1.0 (or possibly very much less than 1.0, showing that the *non*-association is unexpected, as in fraud detection). Interpretation depends on experience with similar data and the usefulness for a particular business question. (Sect. 12.1)

- A typical workflow for association rules (Sects. 12.2.1 and 12.2.2) is:
  - Import the raw data and use `as(data, "transactions")` to transform it to a transactions object for better performance.
  - Use `apriori(transactions, support= , confidence= , target="rules")` to find a set of association rules.
  - Plot the resulting rule with `plot(..., interactive=TRUE)` and inspect the rules (Sect. 12.3)
  - Look for patterns by selecting subsets of rules, such as those with highest lift, and use `plot(..., method="graph")` for visualization (Sect. 12.3.1)
- Data such as item profit margin may be used to extend analyses and look at the potential business impact of acting on particular rules (Sect. 12.3.2)
- Association rule mining can also be a useful exploratory technique for mining non-transactional data such as consumer segmentation data (Sect. 12.4).
- We used R functions `cut()` to slice continuous data (Sect. 12.4.1) and `class()` to determine an object's data type (Sect. 12.3.3)
- When you write a custom function, use `warning()` to report potential issues and violations of data assumptions (Sect. 12.3.3), and use `stop()` when a condition means that the function should not continue.