Lab 08: Association Rules

PSTAT 131/231, Winter 2018

Learning Objectives

- Mine association rules by `apriori()` in package `arules`
- Sort rules by confidence and lift
- Delete the redundant rules
- Target the most interested rules and items
- Visualize rules

---

1. Obtain Groceries dataset and install `arulesViz` package

Association analysis can be done in R by using the package `arules`. Instead of installing it directly, we use `arulesViz` package, which automatically loads other needed packages like `arules` (Hahsler et al. 2010) for handling and mining association rules. We will perform association analysis on the Groceries dataset, which can be found in `arules`. The Groceries dataset contains 1 month (30 days) of real-world point-of-sale transaction data from a typical local grocery outlet. The data set contains 9835 transactions and the items are aggregated to 169 categories. The goal is to identify the most frequent-purchased groceries.

```r
# install.packages("arulesViz")
library("arulesViz")

# Load data: 9835 transactions and 169 categories
data("Groceries")

# Save the transaction dataset by the following command
groceries = as(Groceries, "transactions")

# Summarise the basic statistics of the data set
summary(groceries)
```

```
## transactions as itemMatrix in sparse format with
##  9835 rows (elements/itemsets/transactions) and
##  169 columns (items) and a density of 0.02609146
##
## most frequent items:
##       whole milk other vegetables       rolls/buns             soda
##             2513             1903             1809             1715
##           yogurt          (Other)
##             1372            34055
##
## element (itemset/transaction) length distribution:
## sizes
##    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
## 2159 1643 1299 1005  855  645  545  438  350  246  182  117   78   77   55
##   16   17   18   19   20   21   22   23   24   26   27   28   29   32
##   46   29   14   14    9   11    4    6    1    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
##
## includes extended item information - examples:
##        labels  level2           level1
## 1 frankfurter sausage meat and sausage
## 2     sausage sausage meat and sausage
## 3  liver loaf sausage meat and sausage
```

From the above output, we see the dataset is rather sparse with a density just above 2.6%, that `whole milk` is the most popular item and that the average transaction contains less than 5 items.

To display associations and transactions as well as other information formatted for inspection and visualize the frequency of items, **`inspect()`** and **`itemFrequencyPlot()`** are often used.

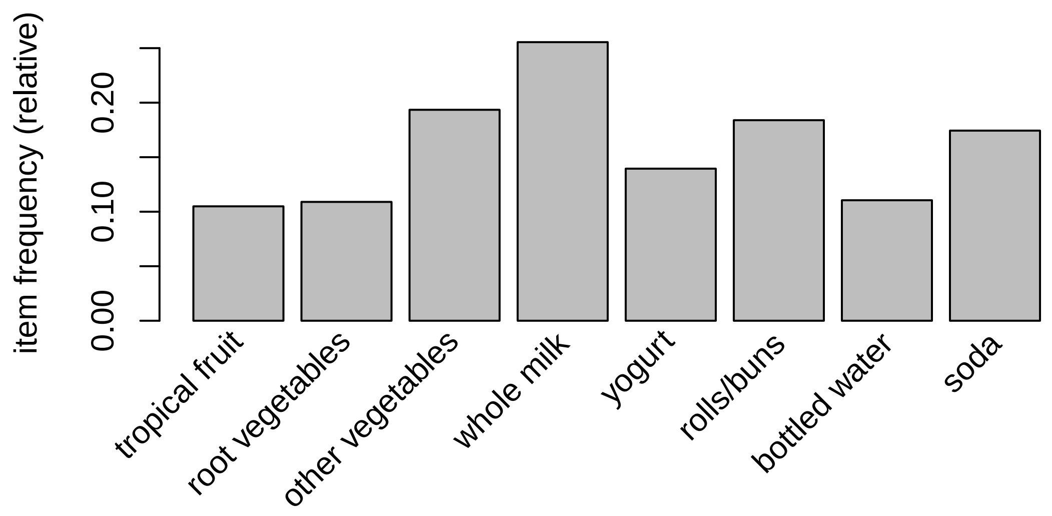
Inspect the first 3 transactions by

```r
inspect(groceries[1:3])
```

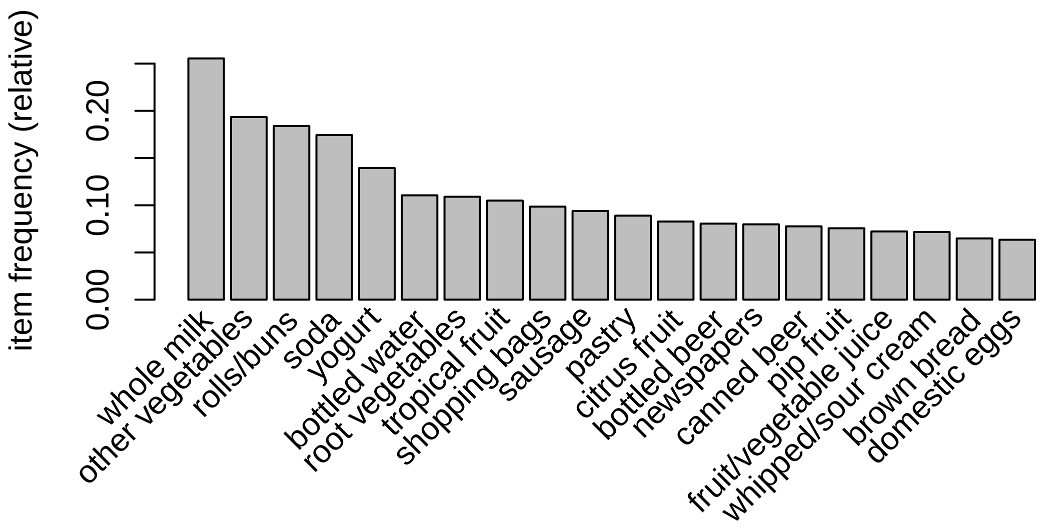
```
##     items
## [1] {citrus fruit,
##      semi-finished bread,
##      margarine,
##      ready soups}
## [2] {tropical fruit,
##      yogurt,
##      coffee}
## [3] {whole milk}
```

Check the histogram satisfying filtering conditions

```r
# Plot the frequency of items
itemFrequencyPlot(groceries, support = 0.1)
```



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



In the above function `itemFrequencyPlot()`, `support=0.1` instructs R only to display items which have a support at least 0.1. `topN` instructs R to only plot 20 items with the highest item frequency. The items are plotted ordered by descending support.

2. Association analysis

We mine association rules using the Apriori algorithm by function `apriori()`. The Apriori algorithm employs level-wise search for frequent item sets.

(a). Train a model on the data

`apriori()` function:

- ***parameter*** instructs R how to mine the association rule. You can specify `support`, `confidence`, `maxlen` (of items), and `maxtime` to control the mining process. If `parameter` is not specified, by default it mines rules with a minimum support of 0.1, minimum confidence of 0.8, maximum of 10 items, and a maximal time for subset checking of 5 seconds.
- ***apperance*** restricts the item appearance. By default all items can appear unrestricted.
- ***control*** takes charge of the algorithmic performance of the mining algorithm (item sorting, report progress (verbose), etc).

To begin with, let's mine the rules using `apriori` with the default settings

```
# Mine the rules with the default setting
# i.e., min support = 0.1, min confidence = 0.8, maximum of 10 items,
# and maximal time for subset checking = 5 seconds.
apriori(groceries)
```

```
## Apriori
##
## Parameter specification:
##  confidence minval smax arem  aval originalSupport maxtime support minlen
##         0.8    0.1    1 none FALSE            TRUE       5     0.1      1
##  maxlen target   ext
##      10  rules FALSE
##
## Algorithmic control:
##  filter tree heap memopt load sort verbose
##     0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 983
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
## sorting and recoding items ... [8 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 done [0.00s].
## writing ... [0 rule(s)] done [0.00s].
## creating S4 object  ... done [0.00s].

## set of 0 rules
```

Look at the bottom line in the output, the default settings result in **zero** rules learnt, which is not informative at all, thus we set better parameters to learn more rules. This time we simultaneously set 0.001 as the minimum support and 0.5 as the minimum confidence. (Notice here the usage of `list` function)

```r
# Mine the rules with min support = 0.001 and min confidence = 0.5
rules = apriori(groceries, parameter=list(support=0.001, confidence=0.8))
```

```
## Apriori
##
## Parameter specification:
##  confidence minval smax arem  aval originalSupport maxtime support minlen
##         0.8    0.1    1 none FALSE            TRUE       5   0.001      1
##  maxlen target   ext
##      10  rules FALSE
##
## Algorithmic control:
##  filter tree heap memopt load sort verbose
##     0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 9
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
## sorting and recoding items ... [157 item(s)] done [0.01s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 done [0.01s].
## writing ... [410 rule(s)] done [0.00s].
## creating S4 object  ... done [0.00s].
```

(b) Check the model results

- Recall that **`inspect()`** function can display associations and transactions as well as other information formatted for inspection.

  Look at the first 3 rules as well as the parameters `support`, `confidence` and `lift`. `options()` is to round all paramters to have 2 digits.

```r
options(digits=2)
inspect(rules[1:3])
```

```
##     lhs                           rhs             support confidence lift
## [1] {liquor,red/blush wine} => {bottled beer} 0.0019  0.90       11.2
## [2] {curd,cereals}          => {whole milk}   0.0010  0.91        3.6
## [3] {yogurt,cereals}        => {whole milk}   0.0017  0.81        3.2
##     count
## [1] 19
## [2] 10
## [3] 17
```

This result reads easily, for example: if someone buys `yogurt` and `cereals`, they are 81% likely to buy `whole milk` too.

- We can also get `summary` information

```r
summary(rules)
```

```
## set of 410 rules
##
## rule length distribution (lhs + rhs):sizes
##   3   4   5   6
##  29 229 140  12
##
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     3.0     4.0     4.0     4.3     5.0     6.0
##
## summary of quality measures:
##     support          confidence        lift           count
##  Min.   :0.00102   Min.   :0.80   Min.   : 3.1   Min.   :10.0
##  1st Qu.:0.00102   1st Qu.:0.83   1st Qu.: 3.3   1st Qu.:10.0
##  Median :0.00122   Median :0.85   Median : 3.6   Median :12.0
##  Mean   :0.00125   Mean   :0.87   Mean   : 4.0   Mean   :12.3
##  3rd Qu.:0.00132   3rd Qu.:0.91   3rd Qu.: 4.3   3rd Qu.:13.0
##  Max.   :0.00315   Max.   :1.00   Max.   :11.2   Max.   :31.0
##
## mining info:
##       data ntransactions support confidence
##  groceries          9835   0.001        0.8
```

As in the output above:

- The number of rules generated: 410
- The distribution of rules by length: Most rules are 4 items long
- The summary of quality measures: interesting to see ranges of support, lift, and confidence.
- The information on the data mined: total data mined, and minimum parameters.

(c) Sort the rules

The first issue we see by `inspect(rules[1:3])` is that the rules are not sorted. Often we will want the most relevant rules first. That is to say, we want the rules are ordered in some certain way. This can easily be done by using `sort()` function.

`sort()`, built in `arules()`, is helpful to sort elements in `rules` (which are the rules obtained by `apriori()` previously). Specifying `by="lift"`, we use the lift quality measure to sort `rules`. You can easily sort the rules by one of `lift`, `confidence` and `support`.

Let's say we wanted to have the most likely rules, so we use `confidence` to sort the rules by executing the following code. Due to too many transactions to be displayed, we use `head()` nested in `inspect()` in order to check only the first three transactions.

```
# The first 3 rules with respect to the confidence (in a decreaseing manner)
rules.sorted = sort(rules, by="confidence", decreasing=TRUE)
inspect(head(rules.sorted, 3))
```

```
##     lhs                                rhs          support confidence lift
## [1] {rice,sugar}                    => {whole milk} 0.0012  1          3.9
## [2] {canned fish,hygiene articles} => {whole milk} 0.0011  1          3.9
## [3] {root vegetables,butter,rice}  => {whole milk} 0.0010  1          3.9
##     count
## [1] 12
## [2] 11
## [3] 10
```

We can also sort the mined rules by `lift` using the following code.

```
# The first 5 rules with respect to the lift measure
inspect(head(sort(rules, by="lift"),4))
```

```
##     lhs                         rhs                support confidence lift count
## [1] {liquor,
##      red/blush wine}        => {bottled beer}     0.0019       0.90 11.2    19
## [2] {citrus fruit,
##      other vegetables,
##      soda,
##      fruit/vegetable juice} => {root vegetables}  0.0010       0.91  8.3    10
## [3] {tropical fruit,
##      other vegetables,
##      whole milk,
##      yogurt,
##      oil}                   => {root vegetables}  0.0010       0.91  8.3    10
## [4] {citrus fruit,
##      grapes,
##      fruit/vegetable juice} => {tropical fruit}   0.0011       0.85  8.1    11
```

Note: $lift$ is the ratio of Confidence to Expected Confidence. The larger the lift, the more significant the association. To explain it more, if some rules had $lift = 1$, it would imply that the occurrence of the antecedent and that of the consequent are independent of each other, then no rule can be drawn involving those two events. If the lift is $lift > 1$, it tells the degree to which those two occurrences are dependent on one another, and makes those rules potentially more useful for predicting the consequent in future observations.

```
# Sort the rules as lift increases
# (change the number of displayed rules to see the increasing trend in lift)
inspect(head(sort(rules, by="lift", decreasing=FALSE), 3))
```

```
##     lhs                    rhs           support confidence lift count
## [1] {herbs,
##      rolls/buns}       => {whole milk}  0.0024        0.8  3.1    24
## [2] {butter,
##      yogurt,
##      soft cheese}      => {whole milk}  0.0012        0.8  3.1    12
## [3] {frankfurter,
##      other vegetables,
##      frozen meals}     => {whole milk}  0.0012        0.8  3.1    12
```

(d) Redundancies

Once the rules have been created, a researcher can then review and filter the rules down to a manageable subset (since some rules will repeat). Redundancy indicates that one itemset might be a subset of another itemset(s). As an analyst you can elect to drop the former itemset from the dataset.

We can find all redundant rules by using `is.subset()`, `colSums()` and `which()`. The logic works in the following order:

- With `proper=TRUE` in `is.subset()`, we evaluate if all rules are **proper** subsets of each other. A dot in the sparse matrix indicates a `FALSE` and a vertical bar (|) indicates a `TRUE`. In other words, if a vertical bar (|) occurs in the matrix, it means the itemset of that column is redundant. Name the resulting matrix from `is.subset()` as `subset.matrix`.
- To carry out the above idea in coding, we calculate the column sums of `subset.matrix`. If a column sum is 0, then this column is not redundant; however if it is greater than 0 (equivalently ≥ 1), then the cooresponding column is redundant.
- Use `which()` to retrun the redundant itemset indices.

Let's do it step by step.

- Find whether each itemset is a subset of another by `is.subset()`. `subset.matrix` is a 410 by 410 sparse matrix with elements `|` and `.` in each entry.
- Drop the redundant rules.

  ```
  subset.matrix = is.subset(rules.sorted, proper=TRUE)
  ```

- Record redundant itemsets. Redundant itemsets are columns with column sum greater than or equal to 1 in `subset.matrix`. `redundant` is a 410 by 1 named-logical vector with entries `TRUE` and `FALSE`.

  ```
  redundant = colSums(subset.matrix)  >= 1
  redundant[1:5]
  ```

  ```
  ##                              {whole milk,rice,sugar}
  ##                                                FALSE
  ##              {whole milk,canned fish,hygiene articles}
  ##                                                FALSE
  ##               {root vegetables,whole milk,butter,rice}
  ##                                                 TRUE
  ## {root vegetables,whole milk,whipped/sour cream,flour}
  ##                                                FALSE
  ##         {whole milk,butter,soft cheese,domestic eggs}
  ##                                                FALSE
  ```

- Obtain the indices of redundant itemsets by `which()`. In plain English, if one itemset is with `TRUE` in `redundant`, then `which()` tells you the index of this itemset. Thus we know this rule is redundant and should be dropped.

  ```
  which(redundant)
  ```

  Due to the length of `which(redundant)`, we show the first 5 entries.

  ```
  which(redundant)[1:5]
  ```

  ```
  ##                          {root vegetables,whole milk,butter,rice}
  ##                                                                 3
  ```

```
##       {tropical fruit,grapes,other vegetables,whole milk,yogurt}
##                                                               14
##           {ham,tropical fruit,pip fruit,other vegetables,yogurt}
##                                                               15
##       {ham,tropical fruit,pip fruit,other vegetables,whole milk}
##                                                               16
## {root vegetables,other vegetables,whole milk,butter,white bread}
##                                                               19
```

- Delete redundancies. Before removing redundancy, we had 410 rules, but now we have 341 rules.

```
rules.pruned = rules.sorted[!redundant]
summary(rules.pruned)
```

```
## set of 341 rules
##
## rule length distribution (lhs + rhs):sizes
##   3   4   5   6
##  29 216  95   1
##
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     3.0     4.0     4.0     4.2     5.0     6.0
##
## summary of quality measures:
##     support          confidence        lift           count
##  Min.   :0.00102   Min.   :0.80   Min.   : 3.1   Min.   :10.0
##  1st Qu.:0.00102   1st Qu.:0.82   1st Qu.: 3.3   1st Qu.:10.0
##  Median :0.00122   Median :0.85   Median : 3.6   Median :12.0
##  Mean   :0.00127   Mean   :0.86   Mean   : 3.9   Mean   :12.5
##  3rd Qu.:0.00132   3rd Qu.:0.91   3rd Qu.: 4.3   3rd Qu.:13.0
##  Max.   :0.00315   Max.   :1.00   Max.   :11.2   Max.   :31.0
##
## mining info:
##       data ntransactions support confidence
##  groceries          9835   0.001        0.8
```

(e) Two ways to find the most interested subsets of rules

Now that we know how to generate rules, limit the output, let's say we wanted to target several items to generate rules. Suppose we like `whole milk`. There are three types of targets we might be interested in:

- What are customers likely to buy before buying `whole milk`?
- What are customers likely to buy if they purchase `whole milk`?
- What are the rules containing `whole milk`?

This essentially means we can restrict either the left-hand-side (lhs) or right-hand-side (rhs) in all mined rules and select those that are satisfying. In R, we have two choices to do so, either use `subset()` function, or use `apriori()` function. These two methods must yield the same results.

- Succinctly, we use `subset()` and some value matching operator in R.
  - Find all the rules including `whole milk`.

```
milk = subset(rules.pruned, items %in% "whole milk")
inspect(tail(milk,3))
```

```
##     lhs                        rhs                  support confidence lift count
## [1] {sausage,
##      citrus fruit,
##      other vegetables,
##      whipped/sour cream} => {whole milk}        0.0012         0.8  3.1    12
## [2] {sausage,
##      citrus fruit,
##      whole milk,
##      whipped/sour cream} => {other vegetables}  0.0012         0.8  4.1    12
## [3] {citrus fruit,
##      tropical fruit,
##      yogurt,
##      whipped/sour cream} => {whole milk}        0.0012         0.8  3.1    12
```

– Find what goods customers are likely to buy BEFORE buying `whole milk`. `%in%` is used to select all rules with `whole milk` in the left-hand-side.

```
before.milk = subset(rules.pruned, lhs %in% "whole milk")
inspect(head(before.milk,3))
```

```
##     lhs                         rhs                support confidence lift count
## [1] {whole milk,
##      rolls/buns,
##      soda,
##      newspapers}          => {other vegetables}  0.0010       1.00  5.2    10
## [2] {root vegetables,
##      whole milk,
##      yogurt,
##      oil}                 => {other vegetables}  0.0014       0.93  4.8    14
## [3] {citrus fruit,
##      whole milk,
##      whipped/sour cream,
##      domestic eggs}       => {other vegetables}  0.0012       0.92  4.8    12
```

– Find what goods customers are likely to buy IF they buy `whole milk`. `%pin%` uses partial matching for all items corresponding to the variable `whole milk` in the right-hand-side.

```
after.milk = subset(rules.pruned, rhs %pin% "whole milk")
# Sort pmilk by lift measure
inspect(head(sort(after.milk, by = "lift"), 3))
```

```
##     lhs                         rhs          support confidence lift count
## [1] {rice,
##      sugar}               => {whole milk}  0.0012          1  3.9    12
## [2] {canned fish,
##      hygiene articles}    => {whole milk}  0.0011          1  3.9    11
## [3] {root vegetables,
##      whipped/sour cream,
##      flour}               => {whole milk}  0.0017          1  3.9    17
```

- Alternatively, we can use `apriori()` function and change its default settings. Recall that ***apperance*** restricts the item appearance and ***control*** takes charge of the algorithmic performance of the mining algorithm (item sorting, report progress (verbose), etc).

  – Find what goods customers are likely to buy BEFORE buying `whole milk`.

```
# Find what goods customers are likely to buy BEFORE buying whole milk
milk = apriori(data=groceries,
               parameter=list(supp=0.001,conf = 0.8),
               appearance = list(default="lhs",rhs="whole milk"),
               control = list(verbose=F))
```

We can sort the rules by confidence.

```
# Sort the rules by decreasing confidence
milk.sort = sort(milk, decreasing=TRUE, by="confidence")
# Check the first 5 ruls
inspect(milk.sort[1:3])
```

```
##     lhs                                 rhs          support confidence lift
## [1] {rice,sugar}                     => {whole milk} 0.0012  1          3.9
## [2] {canned fish,hygiene articles} => {whole milk} 0.0011  1          3.9
## [3] {root vegetables,butter,rice}  => {whole milk} 0.0010  1          3.9
##     count
## [1] 12
## [2] 11
## [3] 10
```

* Find what goods customers are likely to buy AFTER they buy `whole milk`. Likewise, we can set the left hand side to be `whole milk` and find its antecedents.

  Note: we set a minimum length of 2 to avoid empty left hand side items; we set minumum confidence of 0.15 because we got no rules with 0.8.

```
pmilk = apriori(data=groceries, parameter=list(supp=0.001,conf = 0.15,minlen=2),
                appearance = list(default="rhs",lhs="whole milk"),
                control = list(verbose=F))
pmilk.sort = sort(pmilk, decreasing=TRUE,by="lift")
inspect(pmilk.sort[1:3])
```

```
##     lhs             rhs               support confidence lift count
## [1] {whole milk} => {root vegetables} 0.049   0.19       1.8  481
## [2] {whole milk} => {tropical fruit}  0.042   0.17       1.6  416
## [3] {whole milk} => {yogurt}          0.056   0.22       1.6  551
```

3. Visualize the rules

We use the built-in `plot()` function in the package `arulesViz` to visualize association rules and item sets.

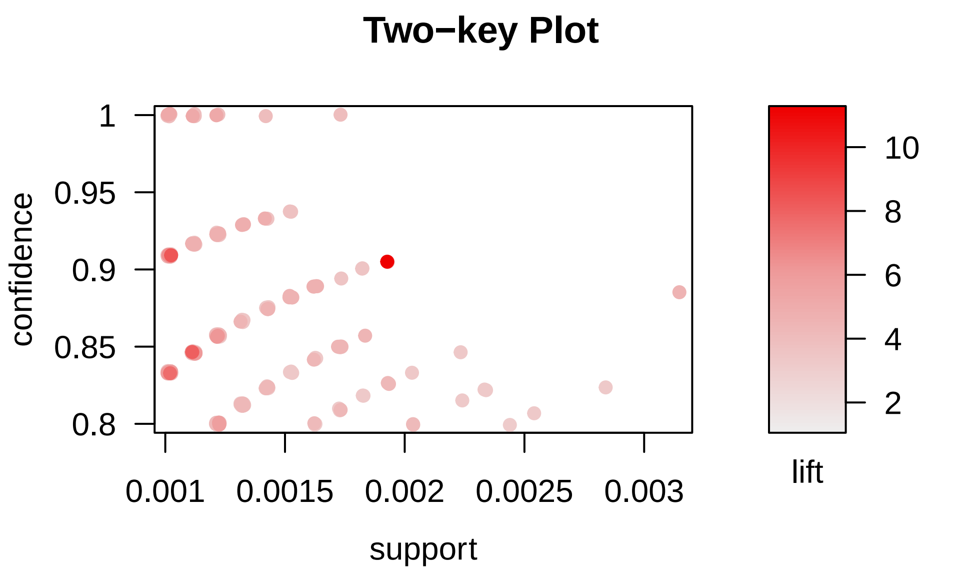
(a) Scatter plots

One popular visualization method is scatter plots with shading, often referred to two-key plots.

```
# Get help file of plot in arulesViz
# ?arulesViz::plot
```

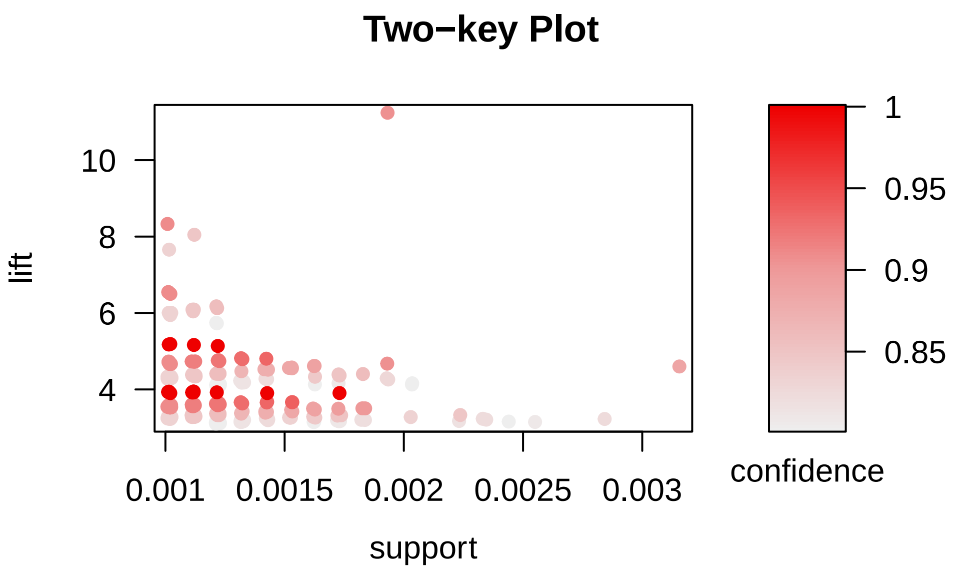
Note: the double colon or triple colon is for accessing exported and internal variables in R. The basic usage is `pkg::name` or `pkg:::name`. For example, `?arulesViz::plot` is to find the help page for the built-in `plot` function in the package `arulesViz`

```
# Scatter plot (two key plot)
# By default, use measure=c("support", "confidence") for visualization
plot(rules.pruned, main="Two-key Plot")
```



Now we can specify the measure of interest used for visualization is *support* and *lift* and *confidence* is of our interest to color the dots (shading).

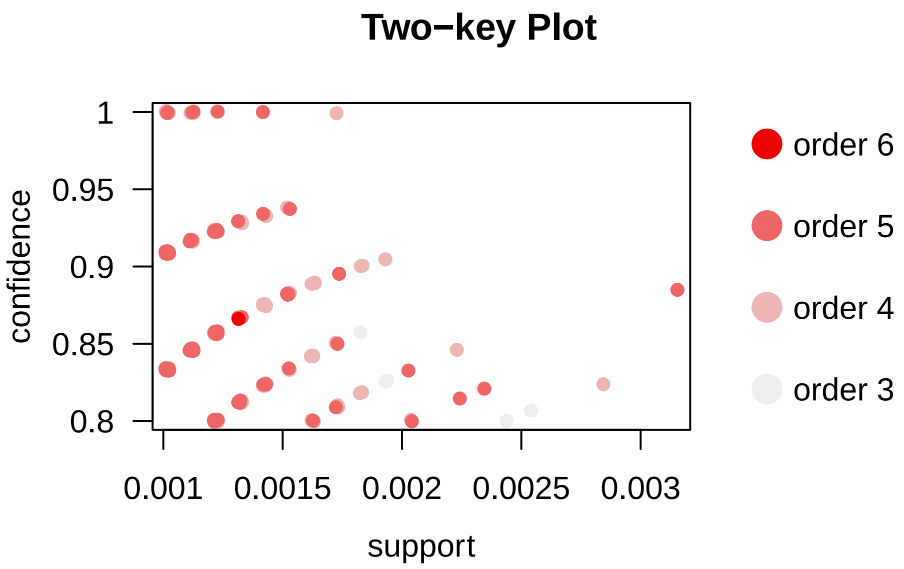
```
# Specify the measure of interest, here we
plot(rules.pruned, measure=c("support","lift"), shading="confidence",main="Two-key Plot")
```



From above plot, we see that rules with high *lift* have typically a relatively low *support*.

By `shading="order"`, we instructs R to color the dots by the number of items contained in rules. It's not hard to see there is a strong inverse relationship between *order* and *support*.

```r
plot(rules.pruned, shading="order", main="Two-key Plot")
```



We can also enable interactive exploration of the plot by `interactive=TRUE`. Try it by selecting different areas using your mouse :}

```r
# interactive plot
plot(rules.pruned, measure=c("support", "lift"), shading="confidence", main="Two-key Plot", interactive
```

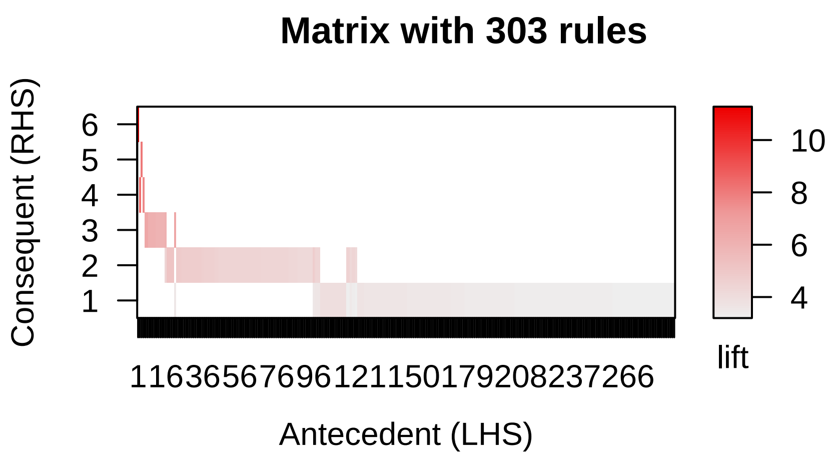
(b) Matrix-based visualizations

This can be used to explore different antecedents which have a similar impact on the same consequent in terms of the measure used in the plot

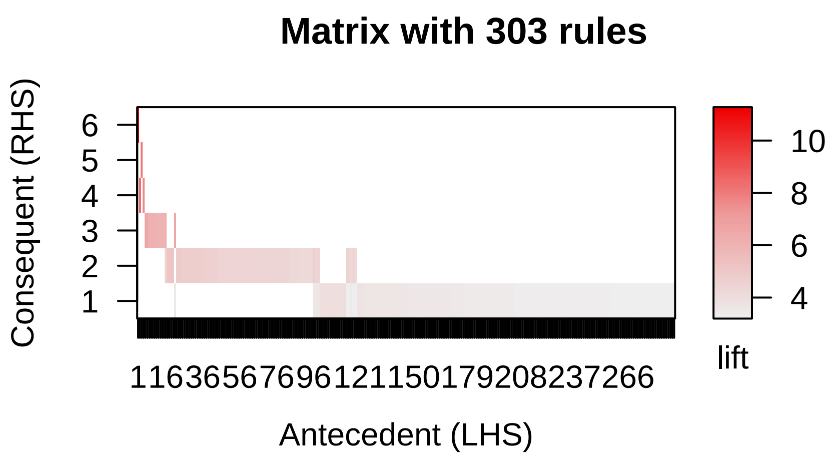
```r
# We reduce the number of rules here by filtering out all rules with a higher confidence score
subrules = rules.pruned[quality(rules.pruned)$confidence > 0.8]
subrules
```

```
## set of 303 rules
```

```r
# By default, plot function uses "scatterplot" as the visualization "method",
# you can now change it to "matrix"
plot(subrules, method="matrix", measure=c("lift", "confidence"))
```



```
plot(subrules, method="matrix", measure=c("lift", "confidence"),control=list(reorder=TRUE))
```



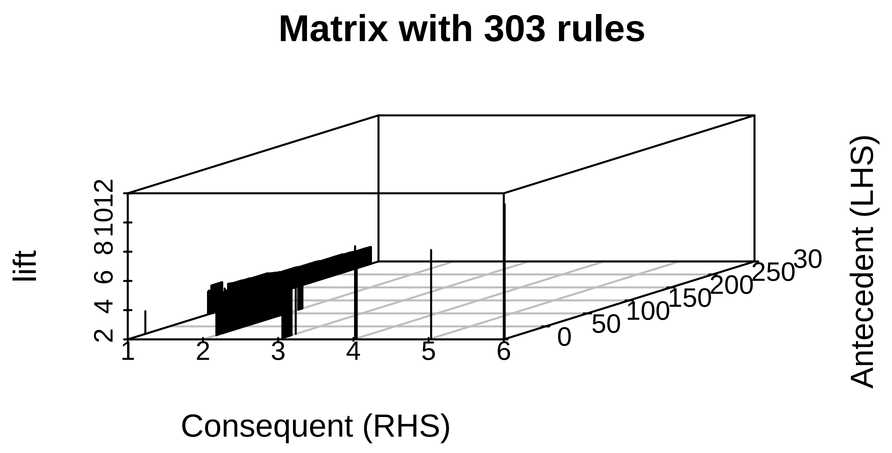
High confidence/high support rules can be identified in the plot as hot/red (high confidence) and dark/intense (high support).

```
plot(subrules, method="matrix3D", measure="lift")
```

```
## Warning in plot.rules(subrules, method = "matrix3D", measure = "lift"):
## method 'matrix3D' is deprecated use method 'matrix' with engine '3d'
```

```
plot(subrules, method="matrix3D", measure="lift", control=list(reorder=TRUE))
```

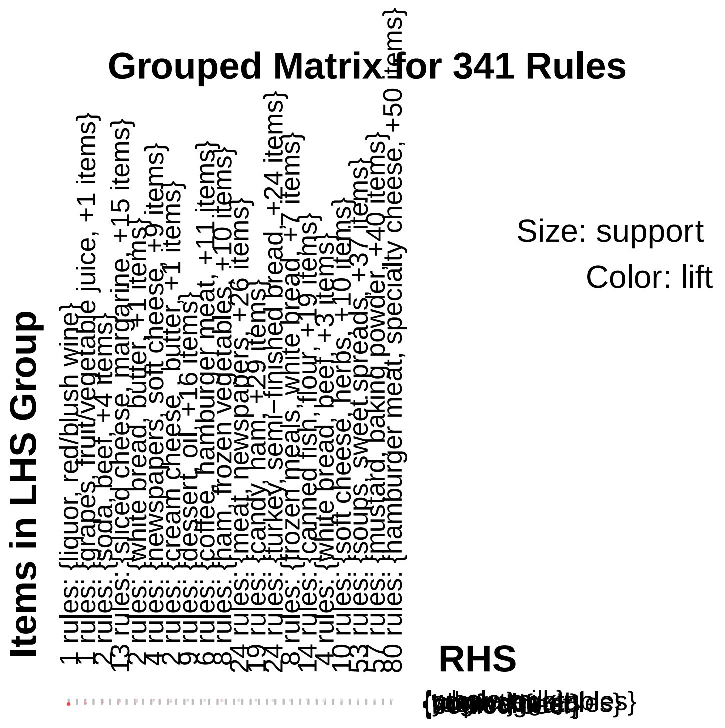
```
## Warning in plot.rules(subrules, method = "matrix3D", measure = "lift",
## control = list(reorder = TRUE)): method 'matrix3D' is deprecated use method
## 'matrix' with engine '3d'
```



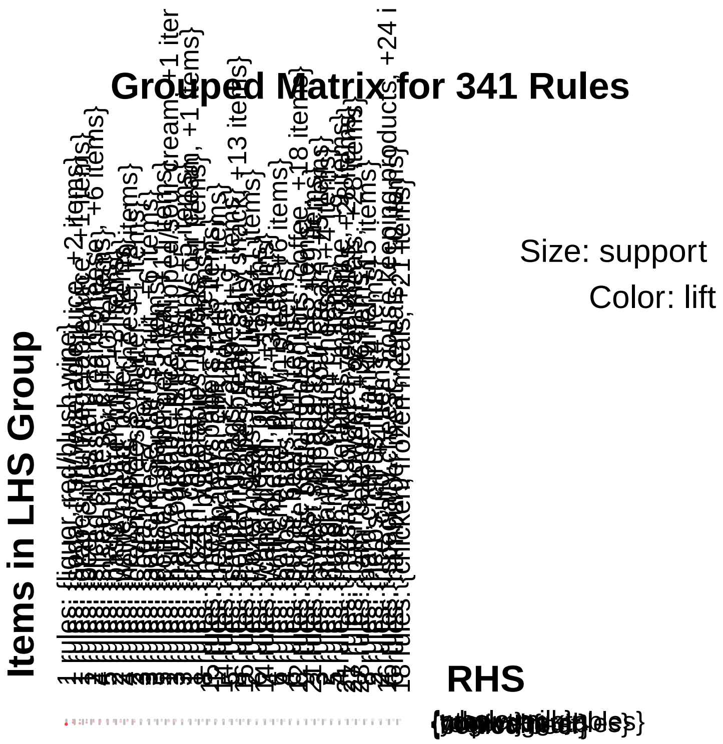
(c) Grouped matrix-based visualization

Matrix-based visualization is limited in the number of rules it can visualize effectively since large sets of rules typically also have large sets of unique antecedents/consequents. Here we introduce a new visualization techniques that enhances matrix-based visualization using grouping of rules via clustering to handle a larger number of rules.

```r
plot(rules.pruned, method="grouped")
```



```r
plot(rules.pruned, method="grouped", control=list(k=50))
```

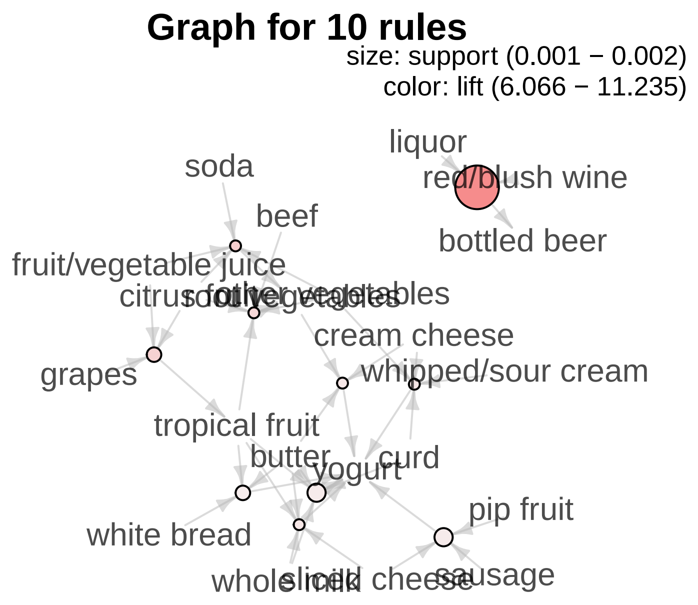


```r
# Try this interacitve plot by yourself
# sel = plot(rules.pruned, method="grouped", interactive=TRUE)
```

(d) Graph-based visualizations

Graph-based visualization offers a very clear representation of rules but they tend to easily become cluttered and thus are only viable for very small sets of rules. Vertices typically represent items or item sets and edges indicate relationship in rules.

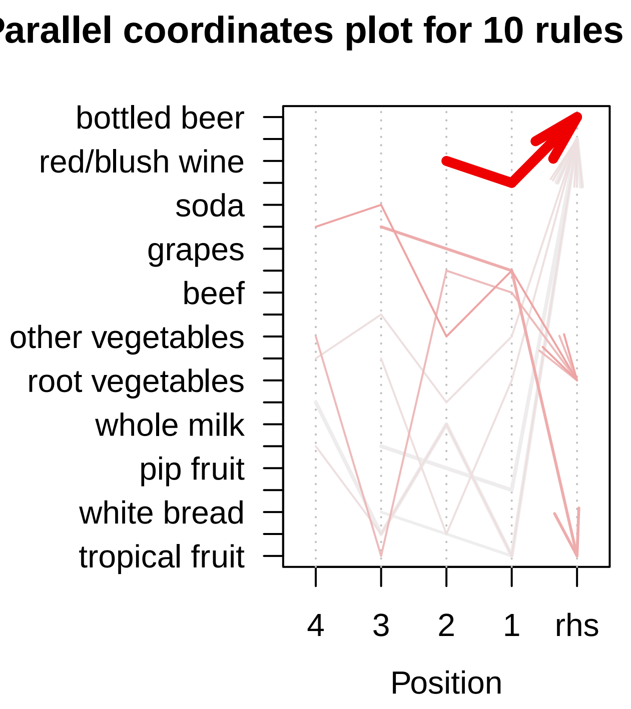
```r
# Graph-based visualizations
subrules2 = head(sort(rules.pruned, by="lift"), 10)
plot(subrules2, method="graph")
```


Graph for 10 rules
size: support (0.001 − 0.002)
color: lift (6.066 − 11.235)
liquor
red/blush wine
soda
beef
bottled beer
fruit/vegetable juice
citrus fruit
root vegetables
other vegetables
cream cheese
grapes
whipped/sour cream
tropical fruit
butter
curd
yogurt
pip fruit
white bread
whole milk
sliced cheese
sausage

(e) Parallel coordinates plot

The width of the arrows represents support and the intensity of the color represent confidence. It is obvious that for larger rule sets visual analysis becomes difficult since with an increasing number of rules also the number of crossovers between the lines increases

```r
# Parallel coordinates plot
plot(subrules2, method="paracoord")
```



Your turn

```r
# Find the rules that involving "soda" on the left hand side and "whole milk" on the right
# hand side using subset function and logical operator "&"

# Hint: use lhs and rhs argument inside of subset function

# rules: all the association rules learnt from the data
```

Credit: adopted from http://www.salemmarafi.com/code/market-basket-analysis-with-r/