R for Data Science

Learn and explore the fundamentals of data science with R

Dan Toomey
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Data Mining Patterns"
- A synopsis of the book’s content
- More information on R for Data Science

About the Author

Dan Toomey has been developing applications for over 20 years. He has worked in a variety of industries and companies in different roles, from a sole contributor to VP and CTO. For the last 10 years or so, he has been working with companies in the eastern Massachusetts area.

Dan has been contracting under Dan Toomey Software Corporation as a contractor developer in the area.
R for Data Science

R is a software package that provides a language and an environment for data manipulation and statistics calculation. The resulting statistics can be displayed graphically as well.

R has the following features:

- A lean syntax to perform operations on your data
- A set of tools to load and store data in a variety of formats, both local and over the Internet
- Consistent syntax for operating on datasets in memory
- A built-in and an open source collection of tools for data analysis
- Methods to generate on-the-fly graphics and store graphical representations to disk.

What This Book Covers

Chapter 1, Data Mining Patterns, covers data mining in R. In this instance, we will look for patterns in a dataset. This chapter will explore examples of using cluster analysis using several tools. It also covers anomaly detection, and the use of association rules.

Chapter 2, Data Mining Sequences, explores methods in R that allow you to discover sequences in your data. There are several R packages available that help you to determine sequences and portray them graphically for further analysis.

Chapter 3, Text Mining, describes several methods of mining text in R. We will look at tools that allow you to manipulate and analyze the text or words in a source. We will also look into XML processing capabilities.

Chapter 4, Data Analysis – Regression Analysis, explores different ways of using regression analysis on your data. This chapter has methods to run simple and multivariate regression, along with subsequent displays.

Chapter 5, Data Analysis – Correlation, explores several correlation packages. The chapter analyzes data using basic correlation and covariance as well as Pearson, polychoric, tetrachoric, heterogeneous, and partial correlation.

Chapter 6, Data Analysis – Clustering, explores a variety of references for cluster analysis. The chapter covers k-means, PAM, and a number of other clustering techniques. All of these techniques are available to an R programmer.

Chapter 7, Data Visualization – R Graphics, discusses a variety of methods of visualizing your data. We will look at the gamut of data from typical class displays to interaction with third-party tools and the use of geographic maps.
Chapter 8, *Data Visualization – Plotting*, discusses different methods of plotting your data in R. The chapter has examples of simple plots with standardized displays as well as customized displays that can be applied to plotting data.

Chapter 9, *Data Visualization – 3D*, acts as a guide to creating 3D displays of your data directly from R. We will also look at using 3D displays for larger datasets.

Chapter 10, *Machine Learning in Action*, discusses how to use R for machine learning. The chapter covers separating datasets into training and test data, developing a model from your training data, and testing your model against test data.

Chapter 11, *Predicting Events with Machine Learning*, uses time series datasets. The chapter covers converting your data into an R time series and then separating out the seasonal, trend, and irregular components. The goal is to model or predict future events.

Chapter 12, *Supervised and Unsupervised Learning*, explains the use of supervised and unsupervised learning to build your model. It covers several methods in supervised and unsupervised learning.
A common use of data mining is to detect patterns or rules in data.

The points of interest are the non-obvious patterns that can only be detected using a large dataset. The detection of simpler patterns, such as market basket analysis for purchasing associations or timings, has been possible for some time. Our interest in R programming is in detecting unexpected associations that can lead to new opportunities.

Some patterns are sequential in nature, for example, predicting faults in systems based on past results that are, again, only obvious using large datasets. These will be explored in the next chapter.

This chapter discusses the use of R to discover patterns in datasets' various methods:

- **Cluster analysis**: This is the process of examining your data and establishing groups of data points that are similar. Cluster analysis can be performed using several algorithms. The different algorithms focus on using different attributes of the data distribution, such as distance between points, density, or statistical ranges.

- **Anomaly detection**: This is the process of looking at data that appears to be similar but shows differences or anomalies for certain attributes. Anomaly detection is used frequently in the field of law enforcement, fraud detection, and insurance claims.

- **Association rules**: These are a set of decisions that can be made from your data. Here, we are looking for concrete steps so that if we find one data point, we can use a rule to determine whether another data point will likely exist. Rules are frequently used in market basket approaches. In data mining, we are looking for deeper, non-obvious rules that are present in the data.
Cluster analysis
Cluster analysis can be performed using a variety of algorithms; some of them are listed in the following table:

<table>
<thead>
<tr>
<th>Type of model</th>
<th>How the model works</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connectivity</td>
<td>This model computes distance between points and organizes the points based on closeness.</td>
</tr>
<tr>
<td>Partitioning</td>
<td>This model partitions the data into clusters and associates each data point to a cluster. Most predominant is k-means.</td>
</tr>
<tr>
<td>Distribution Models</td>
<td>This model uses a statistical distribution to determine the clusters.</td>
</tr>
<tr>
<td>Density</td>
<td>This model determines closeness of data points to arrive at dense areas of distribution. The common use of DBSCAN is for tight concentrations or OPTICS for more sparse distributions.</td>
</tr>
</tbody>
</table>

Within an algorithm, there are finer levels of granularity as well, including:

- **Hard or soft clustering**: It defines whether a data point can be part of more than one cluster.
- **Partitioning rules**: Are rules that determine how to assign data points to different partitions. These rules are as follows:
  - **Strict**: This rule will check whether partitions include data points that are not close
  - **Overlapping**: This rule will check whether partitions overlap in any way
  - **Hierarchical**: This rule checks whether the partitions are stratified

In R programming, we have clustering tools for:

- K-means clustering
- K-medoids clustering
- Hierarchical clustering
- Expectation-maximization
- Density estimation
K-means clustering

K-means clustering is a method of partitioning the dataset into $k$ clusters. You need to predetermine the number of clusters you want to divide the dataset into. The k-means algorithm has the following steps:

1. Select $k$ random rows (centroids) from your data (you have a predetermined number of clusters to use).
2. We are using Lloyd's algorithm (the default) to determine clusters.
3. Assign each data point according to its closeness to a centroid.
4. Recalculate each centroid as an average of all the points associated with it.
5. Reassign each data point as closest to a centroid.
6. Continue with steps 3 and 4 until data points are no longer assigned or you have looped some maximum number of times.

This is a heuristic algorithm, so it is a good idea to run the process several times. It will normally run quickly in R, as the work in each step is not difficult. The objective is to minimize the sum of squares by constant refining of the terms.

Predetermining the number of clusters may be problematic. Graphing the data (or its squares or the like) should present logical groupings for your data visually. You can determine group sizes by iterating through the steps to determine the cutoff for selection (we will use that later in this chapter). There are other R packages that will attempt to compute this as well. You should also verify the fit of the clusters selected upon completion.

Using an average (in step 3) shows that k-means does not work well with fairly sparse data or data with a larger number of outliers. Furthermore, there can be a problem if the cluster is not in a nice, linear shape. Graphical representation should prove whether your data fits this algorithm.

Usage

K-means clustering is performed in R programming with the `kmeans` function. The R programming usage of k-means clustering follows the convention given here (note that you can always determine the conventions for a function using the inline help function, for example, `?kmeans`, to get this information):

```r
kmeans(x, centers, iter.max = 10, nstart = 1, algorithm = c("Hartigan-Wong",
```

---

Chapter 1
The various parameters are explained in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>This is the data matrix to be analyzed</td>
</tr>
<tr>
<td>centers</td>
<td>This is the number of clusters</td>
</tr>
<tr>
<td>iter.max</td>
<td>This is the maximum number of iterations (unless reassignment stops)</td>
</tr>
<tr>
<td>nstart</td>
<td>This is the number of random sets to use</td>
</tr>
<tr>
<td>algorithm</td>
<td>This can be of one of the following types: Hartigan-Wong, Lloyd, Forgy, or MacQueen algorithms</td>
</tr>
<tr>
<td>trace</td>
<td>This gives the present trace information as the algorithm progresses</td>
</tr>
</tbody>
</table>

Calling the `kmeans` function returns a `kmeans` object with the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster</td>
<td>This contains the cluster assignments</td>
</tr>
<tr>
<td>centers</td>
<td>This contains the cluster centers</td>
</tr>
<tr>
<td>totss</td>
<td>This gives the total sum of squares</td>
</tr>
<tr>
<td>withinss</td>
<td>This is the vector of within sum of squares, per cluster</td>
</tr>
<tr>
<td>tot.withinss</td>
<td>This contains the total (sum of withinss)</td>
</tr>
<tr>
<td>betweens</td>
<td>This contains the between-cluster sum of squares</td>
</tr>
<tr>
<td>size</td>
<td>This contains the number of data points in each cluster</td>
</tr>
<tr>
<td>iter</td>
<td>This contains the number of iterations performed</td>
</tr>
<tr>
<td>u commentators'</td>
<td>This contains the expert diagnostic</td>
</tr>
</tbody>
</table>

**Example**

First, generate a hundred pairs of random numbers in a normal distribution and assign it to the matrix `x` as follows:

```r
>x <- rbind(matrix(rnorm(100, sd = 0.3), ncol = 2),
            matrix(rnorm(100, mean = 1, sd = 0.3), ncol = 2))
```

We can display the values we generate as follows:

```r
>x

[,1]    [,2]
[10] ---
The resultant `kmeans` object values can be determined and displayed (using 10 clusters) as follows:

```r
> fit <- kmeans(x, 10)
> fit
K-means clustering with 10 clusters of sizes 4, 12, 10, 7, 13, 16, 8, 13, 8, 9

Cluster means:

```
   [,1]      [,2]
1  0.59611989  0.77213527
2  1.09064550  1.02456563
3 -0.01095292  0.41255130
4  0.07613688 -0.48816360
5  1.04043914  0.78864770
6  0.04167769 -0.05023832
7  0.47920281 -0.05528244
8  1.03305030  1.28488358
9  1.47791031  0.90185427
10 -0.28881626 -0.26002816
```

Clustering vector:

```
[1]  7 10 10  6  7  6  3  3  7 10  4  7  4  7  6  7  6  6  4  3 10
 [55]  9  5  5  2  5  8  9  8  1  2  5  9  5  2  5  8  1  5  8  2  8
  8  5  5  8  1  1  5  8  9  9  8  5  2  5  8  2  2  9  2  8  2  8
  9
```

Within cluster sum of squares by cluster:

```
[1] 0.09842712 0.23620192 0.47286373 0.30604945 0.21233870 0.47824982
 0.36380678 0.58063931 0.67803464 0.28407093
```

(between_SS / total_SS =  94.6 %)

Available components:

```
[1] "cluster"  "centers"  "totss"  "withinss"  "tot.withinss"  "betweenss"  "size"  "iter"  "ifault"
```

---

[11]
If we look at the results, we find some interesting data points:

- The **Cluster means** shows the breakdown of the means used for the cluster assignments.
- The **Clustering vector** shows which cluster each of the 100 numbers was assigned to.
- The **Cluster sum of squares** shows the \( \text{totss} \) value, as described in the output.
- The percentage value is the \( \text{betweenss} \) value divided as a percentage of the \( \text{totss} \) value. At 94.6 percent, we have a very good fit.

We chose an arbitrary cluster size of 10, but we should verify that this is a good number to use. If we were to run the `kmeans` function a number of times using a range of cluster sizes, we would end up with a graph that looks like the one in the following example.

For example, if we ran the following code and recorded the results, the output will be as follows:

```r
results <- matrix(nrow=14, ncol=2, dimnames=list(2:15,c("clusters","sumsquares")))
for(i in 2:15) {
  fit <- kmeans(x,i)
  results[i-1,1] <- i
  results[i-1,2] <- fit$totss
}
plot(results)
```

![Sum of Squares](chart.png)
If the data were more distributed, there would be a clear demarcation about the maximum number of clusters, as further clustering will show no improvement in the sum of squares. However, since we used very smooth data for the test, the number of clusters could be allowed to increase.

Once your clusters have been determined, you should be able to gather a visual representation, as shown in the following plot:

**K-medoids clustering**

K-medoids clustering is another method of determining the clusters in a dataset. A medoid is an entity of the dataset that represents the group to which it was inserted. K-means works with centroids, which are artificially created to represent a cluster. So, a medoid is actually part of the dataset. A centroid is a derived amount.

When partitioning around medoids, make sure that the following points are taken care of:

- Each entity is assigned to only one cluster
- Each entity is assigned to the medoid that defines its cluster
- Exactly \( k \) clusters are defined
The algorithm has two phases with several steps:

- **Build phase**: During the build phase, we come up with initial estimates for the clusters:
  1. Choose random $k$ entities to become medoids (the $k$ entities may be provided to the algorithm).
  2. Calculate the dissimilarity matrix (compute all the pairwise dissimilarities (distances) between observations in the dataset) so that we can find the distances.
  3. Assign every entity to the closest medoid.

- **Swap phase**: In the swap phase, we fine-tune our initial estimates given the rough clusters determined in the build phase:
  1. Search each cluster for the entity that lowers the average dissimilarity coefficient the most and therefore makes it the medoid for the cluster.
  2. If any medoid has changed, start from step 3 of the build phase again.

**Usage**

K-medoid clustering is calculated in R programming with the *pam* function:

```r
pam(x, k, diss, metric, medoids, stand, cluster.only, do.swap, keep.diss, keep.data, trace.lev)
```

The various parameters of the *pam* function are explained in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>This is the data matrix or dissimilarity matrix (based on the diss flag)</td>
</tr>
<tr>
<td>k</td>
<td>This is the number of clusters, where 0 is less than $k$ which is less than the number of entities</td>
</tr>
<tr>
<td>diss</td>
<td>The values are as follows:</td>
</tr>
<tr>
<td></td>
<td>* FALSE if $x$ is a matrix</td>
</tr>
<tr>
<td></td>
<td>* TRUE if $x$ is a dissimilarity matrix</td>
</tr>
<tr>
<td>metric</td>
<td>This is a string metric to be used to calculate the dissimilarity matrix. It can be of the following types:</td>
</tr>
<tr>
<td></td>
<td>* euclidean for Euclidean distance</td>
</tr>
<tr>
<td></td>
<td>* manhattan for Manhattan distance</td>
</tr>
<tr>
<td>medoids</td>
<td>If the NULL value is assigned, it means a set of medoids is to be developed. Otherwise, it is a set of initial medoids.</td>
</tr>
</tbody>
</table>
Parameter | Description
---|---
stand | If \( x \) is the data matrix, then measurements in \( x \) will be standardized before computing the dissimilarity matrix.
cluster.only | If the value set is `TRUE`, then only clustering will be computed and returned.
do.swap | This contains a Boolean value to decide whether swap should occur.
keep.diss | This contains a Boolean value to decide whether dissimilarity should be kept in the result.
keep.data | This contains a Boolean value to decide whether data should be kept in the result.
trace.lev | This contains an integer trace level for diagnostics, where 0 means no trace information.

The results returned from the `pam` function can be displayed, which is rather difficult to interpret, or the results can be plotted, which is intuitively more understandable.

**Example**

Using a simple set of data with two (visually) clear clusters as follows, as stored in a file named `medoids.csv`:

<table>
<thead>
<tr>
<th>Object</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Let's use the `pam` function on the `medoids.csv` file as follows:

```r
# load pam function
> library(cluster)

# load the table from a file
```
Data Mining Patterns

```r
> x <- read.table("medoids.csv", header=TRUE, sep=" ","

#execute the pam algorithm with the dataset created for the example
> result <- pam(x, 2, FALSE, "euclidean")
Looking at the result directly we get:
> result
Medoids:
   ID Object x  y
   [1,]  2      2 2 11
   [2,]  7      7 2  6
Clustering vector:
[1] 1 1 1 2 2 2 2 2
Objective function:
   build swap
1.564722 1.564722
Available components:
[1] "medoids" "id.med" "clustering" "objective" "isolation"
[6] "clusinfo" "silinfo" "diss" "call" "data"

Evaluating the results we can see:

• We specified the use of two medoids, and row 3 and 6 were chosen
• The rows were clustered as presented in the clustering vector (as expected, about half in the first medoid and the rest in the other medoid)
• The function did not change greatly from the build phase to the swap phase (looking at the Objective function values for build and swap of 1.56 versus 1.56)

Using a summary for a clearer picture, we see the following result:

> summary(result)
Medoids:
   ID Object x  y
   [1,]  2      2 2 11
   [2,]  7      7 2  6
Clustering vector:
[1] 1 1 1 2 2 2 2
Objective function:
   build swap
1.564722 1.564722

Numerical information per cluster:
```
Chapter 1

sizemax_dissav_diss diameter separation
[1,] 4 2.236068 1.425042 3.741657 5.744563
[2,] 5 3.000000 1.676466 4.898979 5.744563

Isolated clusters:
L-clusters: character(0)
L*-clusters: [1] 1 2

Silhouette plot information:
cluster neighbor sil_width
  2       1        2 0.7575089
  3       1        2 0.6864544
  1       1        2 0.6859661
  4       1        2 0.6315196
  8       2        1 0.7310922
  7       2        1 0.6872724
  6       2        1 0.6595811
  9       2        1 0.6374808
  5       2        1 0.5342637

Average silhouette width per cluster:
[1] 0.6903623 0.6499381

Average silhouette width of total data set:
[1] 0.6679044

36 dissimilarities, summarized:
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
1.4142  2.3961  6.2445  5.2746  7.3822  9.1652

Metric : euclidean
Number of objects : 9

Available components:
[1] "medoids" "id.med" "clustering" "objective" "isolation"
[6] "clusinfo" "silinfo" "diss" "call" "data"

Download the example code
You can download the example code files from your account at
http://www.packtpub.com for all the Packt Publishing books you
have purchased. If you purchased this book elsewhere, you can visit
http://www.packtpub.com/support and register to have the
files e-mailed directly to you.

The summary presents more details on the medoids and how they were selected.
However, note the dissimilarities as well.
Plotting the data, we can see the following output:

```r
#plot a graphic showing the clusters and the medoids of each cluster
> plot(result$data, col = result$clustering)
```

The resulting plot is as we expected it to be. It is good to see the data clearly broken into two medoids, both spatially and by color demarcation.

**Hierarchical clustering**

Hierarchical clustering is a method to ascertain clusters in a dataset that are in a hierarchy.

Using hierarchical clustering, we are attempting to create a hierarchy of clusters. There are two approaches of doing this:

- **Agglomerative (or bottom up):** In this approach, each entity starts as its own cluster and pairs are merged as they move up the hierarchy
- **Divisive (or top down):** In this approach, all entities are lumped into one cluster and are split as they are moved down the hierarchy
The resulting hierarchy is normally displayed using a tree/graph model of a dendogram.

Hierarchical clustering is performed in R programming with the `hclust` function.

**Usage**

The `hclust` function is called as follows:

```
hclust(d, method = "complete", members = NULL)
```

The various parameters of the `hclust` function are explained in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>This is the matrix.</td>
</tr>
<tr>
<td>method</td>
<td>This is the agglomeration method to be used. This should be (a distinct abbreviation of) one of these methods: ward.D, ward.D2, single, complete, average (= UPGMA), mcquitty (= WPGMA), median (= WPGMC), or centroid (= UPGMC).</td>
</tr>
<tr>
<td>members</td>
<td>This could be NULL or d, the dissimilarity matrix.</td>
</tr>
</tbody>
</table>

**Example**

We start by generating some random data over a normal distribution using the following code:

```r
> dat <- matrix(rnorm(100), nrow=10, ncol=10)

> dat
[1,] 1.4811953 -1.0882253 -0.47659922 0.22344983 -0.74227899 0.2835530
[2,] -0.6414931 -1.0103688 -0.55213606 -0.48812235 1.41763706 0.8337524
[3,] 0.2638638 0.2535630 -0.53310519 2.27778665 -0.09526058 1.9579652
[4,] -0.50307726 -0.3873578 -1.54407287 -0.1503834 1.9579652
Then, we calculate the hierarchical distribution for our data as follows:
```
```r
> hc <- hclust(dist(dat))
> hc
Call:
\hspace{1em} \texttt{hclust(d = dist(dat))}

Cluster method : complete  
Distance : euclidean  
Number of objects: 10

The resulting data object is very uninformative. We can display the hierarchical cluster using a dendogram, as follows:

\hspace{1em} \texttt{>plot(hc)}

The dendogram has the expected shape. I find these diagrams somewhat unclear, but if you go over them in detail, the inference will be as follows:

- Reading the diagram in a top-down fashion, we see it has two distinct branches. The implication is that there are two groups that are distinctly different from one another. Within the two branches, we see 10 and 3 as distinctly different from the rest. Generally, it appears that we have determined there are an even group and an odd group, as expected.
- Reading the diagram bottom up, we see closeness and similarity over a number of elements. This would be expected from a simple random distribution.
Expectation-maximization

Expectation-maximization (EM) is the process of estimating the parameters in a statistical model.

For a given model, we have the following parameters:

- $x$: This is a set of observed data
- $z$: This is a set of missing values
- $\theta$: This is a set of unknown parameters that we should apply to our model to predict $z$

The steps to perform expectation-maximization are as follows:

1. Initialize the unknown parameters ($\theta$) to random values.
2. Compute the best missing values ($z$) using the new parameter values.
3. Use the best missing values ($z$), which were just computed, to determine a better estimate for the unknown parameters ($\theta$).
4. Iterate over steps 2 and 3 until we have a convergence.

This version of the algorithm produces hard parameter values ($z$). In practice, soft values may be of interest where probabilities are assigned to various values of the parameters ($z$). By hard values, I mean we are selecting specific $z$ values. We could instead use soft values where $z$ varies by some probability distribution.

We use EM in R programming with the `Mclust` function from the `mclust` library. The full description of `Mclust` is the normal mixture modeling fitted via EM algorithm for model-based clustering, classification, and density estimation, including Bayesian regularization.

**Usage**

The `Mclust` function is as follows:

```r
Mclust(data, G = NULL, modelNames = NULL,
       prior = NULL, control = emControl(),
       initialization = NULL, warn = FALSE, ...)
```
The various parameters of the `Mclust` function are explained in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data</code></td>
<td>This contains the matrix.</td>
</tr>
<tr>
<td><code>G</code></td>
<td>This contains the vector of number of clusters to use to compute BIC. The default value is 1:9.</td>
</tr>
<tr>
<td><code>modelName</code></td>
<td>This contains the vector of model names to use.</td>
</tr>
<tr>
<td><code>prior</code></td>
<td>This contains the optional conjugate prior for means.</td>
</tr>
<tr>
<td><code>control</code></td>
<td>This contains the list of control parameters for EM. The default value is <code>List</code>.</td>
</tr>
<tr>
<td><code>initialization</code></td>
<td>This contains <code>NULL</code> or a list of one or more of the following components:</td>
</tr>
<tr>
<td></td>
<td>• <code>hcPairs</code>: This is used to merge pairs</td>
</tr>
<tr>
<td></td>
<td>• <code>subset</code>: This is to be used during initialization</td>
</tr>
<tr>
<td></td>
<td>• <code>noise</code>: This makes an initial guess at noise</td>
</tr>
<tr>
<td><code>warn</code></td>
<td>This contains which warnings are to be issued. Default is none.</td>
</tr>
</tbody>
</table>

**List of model names**

The `Mclust` function uses a model when trying to decide which items belong to a cluster. There are different model names for univariate, multivariate, and single component datasets. In each, the idea is to select a model that describes the data, for example, VII will be used for data that is spherically displaced with equal volume across each cluster.

<table>
<thead>
<tr>
<th>Model</th>
<th>Type of dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Univariate mixture</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>equal variance (one-dimensional)</td>
</tr>
<tr>
<td>V</td>
<td>variable variance (one-dimensional)</td>
</tr>
<tr>
<td>Multivariate mixture</td>
<td></td>
</tr>
<tr>
<td>EII</td>
<td>spherical, equal volume</td>
</tr>
<tr>
<td>VII</td>
<td>spherical, unequal volume</td>
</tr>
<tr>
<td>EEI</td>
<td>diagonal, equal volume and shape</td>
</tr>
<tr>
<td>VEI</td>
<td>diagonal, varying volume, equal shape</td>
</tr>
<tr>
<td>EVI</td>
<td>diagonal, equal volume, varying shape</td>
</tr>
<tr>
<td>VVI</td>
<td>diagonal, varying volume and shape</td>
</tr>
<tr>
<td>EEE</td>
<td>ellipsoidal, equal volume, shape, and orientation</td>
</tr>
<tr>
<td>EEV</td>
<td>ellipsoidal, equal volume and equal shape</td>
</tr>
</tbody>
</table>
### Model Type of dataset

<table>
<thead>
<tr>
<th>Model</th>
<th>Type of dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEV</td>
<td>ellipsoidal, equal shape</td>
</tr>
<tr>
<td>VVV</td>
<td>ellipsoidal, varying volume, shape, and orientation</td>
</tr>
<tr>
<td>XXX</td>
<td>ellipsoidal, equal shape</td>
</tr>
<tr>
<td>XXX</td>
<td>ellipsoidal, equal shape</td>
</tr>
<tr>
<td>XXX</td>
<td>ellipsoidal, equal shape</td>
</tr>
</tbody>
</table>

### Example

First, we must load the library that contains the `mclust` function (we may need to install it in the local environment) as follows:

```r
> install.packages("mclust")
> library(mclust)
```

We will be using the `iris` data in this example, as shown here:

```r
```

Now, we can compute the best fit via EM (note capitalization of `Mclust`) as follows:

```r
> fit <- Mclust(data)
```

We can display our results as follows:

```r
> fit
'Mclust' model object:
 best model: ellipsoidal, equal shape (VEV) with 2 components

> summary(fit)
----------------------------------------------------
Gaussian finite mixture model fitted by EM algorithm
----------------------------------------------------
Mclust VEV (ellipsoidal, equal shape) model with 2 components:

<table>
<thead>
<tr>
<th>log.likelihood</th>
<th>n</th>
<th>df</th>
<th>BIC</th>
<th>ICL</th>
</tr>
</thead>
<tbody>
<tr>
<td>-121.1459</td>
<td>149</td>
<td>37</td>
<td>-427.4378</td>
<td>-427.4385</td>
</tr>
</tbody>
</table>
```

Clustering table:

```
1 2
49 100
```
Simple display of the fit data object doesn’t tell us very much, it shows just what was used to compute the density of the dataset.

The summary command presents more detailed information about the results, as listed here:

- **log.likelihood (-121)**: This is the log likelihood of the BIC value
- **n (149)**: This is the number of data points
- **df (37)**: This is the distribution
- **BIC (-427)**: This is the Bayesian information criteria; this is an optimal value
- **ICL (-427)**: Integrated Complete Data Likelihood—a classification version of the BIC. As we have the same value for ICL and BIC we classified the data points.

We can plot the results for a visual verification as follows:

```r
> plot(fit)
```

You will notice that the `plot` command for EM produces the following four plots (as shown in the graph):

- The BIC values used for choosing the number of clusters
- A plot of the clustering
- A plot of the classification uncertainty
- The orbital plot of clusters

The following graph depicts the plot of density.

The first plot gives a depiction of the BIC ranges versus the number of components by different model names; in this case, we should probably not use VEV, for example:
This second plot shows the comparison of using each of the components of the data feed against every other component of the data feed to determine the clustering that would result. The idea is to select the components that give you the best clustering of your data. This is one of those cases where your familiarity with the data is key to selecting the appropriate data points for clustering.

In this case, I think selecting X5.1 and X1.4 yield the tightest clusters, as shown in the following graph:
The third plot gives another iteration of the clustering affects of the different choices highlighting the main cluster by eliminating any points from the plot that would be applied to the main cluster, as shown here:

The final, fourth plot gives an orbital view of each of the clusters giving a highlight display of where the points might appear relative to the center of each cluster, as shown here:
Density estimation

Density estimation is the process of estimating the probability density function of a population given in an observation set. The density estimation process takes your observations, disperses them across a number of data points, runs a FF transform to determine a kernel, and then runs a linear approximation to estimate density.

Density estimation produces an estimate for the unobservable population distribution function. Some approaches that are used to produce the density estimation are as follows:

- **Parzen windows**: In this approach, the observations are placed in a window and density estimates are made based on proximity
- **Vector quantization**: This approach lets you model the probability density functions as per the distribution of observations
- **Histograms**: With a histogram, you get a nice visual showing density (size of the bars); the number of bins chosen while developing the histogram decide your density outcome

Density estimation is performed via the `density` function in R programming. Other functions for density evaluation in R are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBSCAN</td>
<td>This function determines clustering for fixed point clusters</td>
</tr>
<tr>
<td>OPTICS</td>
<td>This function determines clustering for wide distribution clusters</td>
</tr>
</tbody>
</table>

**Usage**

The `density` function is invoked as follows:

```r
density(x, bw = "nrd0", adjust = 1,
        kernel = c("gaussian", "epanechnikov",
                   "rectangular",
                   "triangular", "biweight",
                   "cosine", "optcosine"),
        weights = NULL, window = kernel, width,
        give.Rkern = FALSE,
        n = 512, from, to, na.rm = FALSE, ...
)```
The various parameters of the density function are explained in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>This is the matrix.</td>
</tr>
<tr>
<td>bw</td>
<td>This is the smoothing bandwidth to be used.</td>
</tr>
<tr>
<td>adjust</td>
<td>This is the multiplier to adjust bandwidth.</td>
</tr>
<tr>
<td>kernel</td>
<td>This is the smoother kernel to be used. It must be one of the following kernels:</td>
</tr>
<tr>
<td></td>
<td>• gaussian</td>
</tr>
<tr>
<td></td>
<td>• rectangular</td>
</tr>
<tr>
<td></td>
<td>• triangular</td>
</tr>
<tr>
<td></td>
<td>• epanechnikov</td>
</tr>
<tr>
<td></td>
<td>• biweight</td>
</tr>
<tr>
<td></td>
<td>• cosine</td>
</tr>
<tr>
<td></td>
<td>• optcosine</td>
</tr>
<tr>
<td>weights</td>
<td>This is a vector of observation weights with same length as x.</td>
</tr>
<tr>
<td>window</td>
<td>This is the kernel used.</td>
</tr>
<tr>
<td>width</td>
<td>This is the $S$ compatibility parameter.</td>
</tr>
<tr>
<td>give.Rkern</td>
<td>If the value of this parameter is TRUE, no density is estimated.</td>
</tr>
<tr>
<td>N</td>
<td>This is the number of density points to estimate.</td>
</tr>
<tr>
<td>from, to</td>
<td>These are the left and right-most points to use.</td>
</tr>
<tr>
<td>na.rm</td>
<td>If the value of this parameter is TRUE, missing values are removed.</td>
</tr>
</tbody>
</table>

The available bandwidths can be found using the following commands:

```r
bw.nrd0(x)
```

```r
bw.nrd(x)
```

```r
bw.ucv(x, nb = 1000, lower = 0.1 * hmax, upper = hmax, tol = 0.1 * lower)
```

```r
bw.bcv(x, nb = 1000, lower = 0.1 * hmax, upper = hmax, tol = 0.1 * lower)
```

```r
bw.SJ(x, nb = 1000, lower = 0.1 * hmax, upper = hmax, method = c("ste", "dpi"), tol = 0.1 * lower)
```
The various parameters of the \texttt{bw} function are explained in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x}</td>
<td>This is the dataset</td>
</tr>
<tr>
<td>\texttt{nb}</td>
<td>This is the number of bins</td>
</tr>
<tr>
<td>lower, upper</td>
<td>This is the range of bandwidth which is to be minimized</td>
</tr>
<tr>
<td>\texttt{method}</td>
<td>The \texttt{ste} method is used to solve the equation or the \texttt{dpi} method is used for direct plugin</td>
</tr>
<tr>
<td>\texttt{tol}</td>
<td>This is the convergence tolerance for \texttt{ste}</td>
</tr>
</tbody>
</table>

**Example**

We can use the \texttt{iris} dataset as follows:

```r
The density of the X5.1 series (sepal length) can be computed as follows:
> d <- density(data$X5.1)
> d
density.default(x = data$X5.1)
 Call:
density.default(x = data$X5.1)
 Data: data$X5.1 (149 obs.); Bandwidth 'bw' = 0.2741
  x      y
 Min.:3.478 Min. :0.0001504
 1st Qu.:4.789 1st Qu.:0.0342542
 Median :6.100 Median :0.1538908
 Mean :6.100 Mean :0.1904755
 3rd Qu.:7.411 3rd Qu.:0.3765078
 Max. :8.722 Max. :0.3987472
```
We can plot the density values as follows:

```r
> plot(d)
```

![Density plot](image)

The plot shows most of the data occurring between 5 and 7. So, sepal length averages at just under 6.

**Anomaly detection**

We can use R programming to detect anomalies in a dataset. Anomaly detection can be used in a number of different areas, such as intrusion detection, fraud detection, system health, and so on. In R programming, these are called outliers. R programming allows the detection of outliers in a number of ways, as listed here:

- Statistical tests
- Depth-based approaches
- Deviation-based approaches
- Distance-based approaches
- Density-based approaches
- High-dimensional approaches
Show outliers

R programming has a function to display outliers: `identify` (in `boxplot`).

The `boxplot` function produces a box-and-whisker plot (see following graph). The `boxplot` function has a number of graphics options. For this example, we do not need to set any.

The `identify` function is a convenient method for marking points in a scatter plot. In R programming, box plot is a type of scatter plot.

Example

In this example, we need to generate a 100 random numbers and then plot the points in boxes.

Then, we mark the first outlier with it's identifier as follows:

```r
> y <- rnorm(100)
> boxplot(y)
> identify(rep(1, length(y)), y, labels = seq_along(y))
```

![Boxplot with outlier](image)

Notice the 0 next to the outlier in the graph.
Data Mining Patterns

Example

The boxplot function automatically computes the outliers for a set as well.

First, we will generate a 100 random numbers as follows (note that this data is randomly generated, so your results may not be the same):

```r
> x <- rnorm(100)
```

We can have a look at the summary information on the set using the following code:

```r
> summary(x)
Min. 1st Qu. Median  Mean  3rd Qu.    Max.
-2.12000 -0.74790 -0.20060 -0.01711  0.49930  2.43200
```

Now, we can display the outliers using the following code:

```r
> boxplot.stats(x)$out
[1] 2.420850 2.432033
```

The following code will graph the set and highlight the outliers:

```r
> boxplot(x)
```

![Boxplot Image]
Notice the 0 next to the outlier in the graph.

We can generate a box plot of more familiar data showing the same issue with outliers using the built-in data for cars, as follows:

```r
boxplot(mpg~cyl, data=mtcars, xlab="Cylinders", ylab="MPG")
```

Another anomaly detection example

We can also use box plot’s outlier detection when we have two dimensions. Note that we are forcing the issue by using a union of the outliers in \( x \) and \( y \) rather than an intersection. The point of the example is to display such points. The code is as follows:

```r
x <- rnorm(1000)
y <- rnorm(1000)
f <- data.frame(x,y)
a <- boxplot.stats(x)$out
b <- boxplot.stats(y)$out
list <- union(a,b)
plot(f)
px <- f[f$x %in% a,]
```
Data Mining Patterns

```r
> py <- f[f$y %in% b,]
> p <- rbind(px,py)
> par(new=TRUE)
> plot(p$x, p$y,cex=2,col=2)
```

While R did what we asked, the plot does not look right. We completely fabricated the data; in a real use case, you would need to use your domain expertise to determine whether these outliers were correct or not.

Calculating anomalies

Given the variety of what constitutes an anomaly, R programming has a mechanism that gives you complete control over it: write your own function that can be used to make a decision.

Usage

We can use the `name` function to create our own anomaly as shown here:

```r
name <- function(parameters,…) {
  # determine what constitutes an anomaly
  return(df)
}
```
Here, the parameters are the values we need to use in the function. I am assuming we return a data frame from the function. The function could do anything.

**Example 1**

We will be using the *iris* data in this example, as shown here:

```r
```

If we decide an anomaly is present when sepal is under 4.5 or over 7.5, we could use a function as shown here:

```r
> outliers <- function(data, low, high) {
>  outs <- subset(data, data$X5.1 < low | data$X5.1 > high)
>  return(outs)
> }
```

Then, we will get the following output:

```r
> outliers(data, 4.5, 7.5)
X5.1 X3.5 X1.4 X0.2   Iris.setosa
  8  4.4  2.9  1.4  0.2   Iris-setosa
 13  4.3  3.0  1.4  0.1   Iris-setosa
 38  4.4  3.0  1.3  0.2   Iris-setosa
 42  4.4  3.2  1.3  0.2   Iris-setosa
 105 7.6  3.0  6.6  2.1   Iris-virginica
 117 7.7  2.6  6.9  2.3   Iris-virginica
 118 7.7  2.8  6.7  2.0   Iris-virginica
 122 7.7  2.8  6.7  2.0   Iris-virginica
 131 7.9  3.8  6.4  2.0   Iris-virginica
 135 7.7  3.0  6.1  2.3   Iris-virginica
```

This gives us the flexibility of making slight adjustments to our criteria by passing different parameter values to the function in order to achieve the desired results.

**Example 2**

Another popular package is *DMwR*. It contains the `lofactor` function that can also be used to locate outliers. The `DMwR` package can be installed using the following command:

```r
> install.packages("DMwR")
> library(DMwR)
```
We need to remove the species column from the data, as it is categorical against it data. This can be done by using the following command:

```r
> nospecies <- data[,1:4]
```

Now, we determine the outliers in the frame:

```r
> scores <- lofactor(nospecies, k=3)
```

Next, we take a look at their distribution:

```r
> plot(density(scores))
```

One point of interest is if there is some close equality amongst several of the outliers (that is, density of about 4).

### Association rules

Association rules describe associations between two datasets. This is most commonly used in market basket analysis. Given a set of transactions with multiple, different items per transaction (shopping bag), how can the item sales be associated? The most common associations are as follows:

- **Support**: This is the percentage of transactions that contain A and B.
- **Confidence**: This is the percentage (of time that rule is correct) of cases containing A that also contain B.
- **Lift**: This is the ratio of confidence to the percentage of cases containing B. Please note that if lift is 1, then A and B are independent.
Mine for associations
The most widely used tool in R from association rules is \texttt{apriori}.

Usage
The \texttt{apriori} rules library can be called as follows:

\begin{verbatim}
  apriori(data, parameter = NULL, appearance = NULL, control = NULL)
\end{verbatim}

The various parameters of the \texttt{apriori} library are explained in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>This is the transaction data.</td>
</tr>
<tr>
<td>parameter</td>
<td>This stores the default behavior to mine, with support as 0.1, confidence as 0.8, and maxlen as 10. You can change parameter values accordingly.</td>
</tr>
<tr>
<td>appearance</td>
<td>This is used to restrict items that appear in rules.</td>
</tr>
<tr>
<td>control</td>
<td>This is used to adjust the performance of the algorithm used.</td>
</tr>
</tbody>
</table>

Example
You will need to load the \texttt{apriori} rules library as follows:

\begin{verbatim}
  > install.packages("arules")
  > library(arules)
\end{verbatim}

The market basket data can be loaded as follows:

\begin{verbatim}
\end{verbatim}

Then, we can generate rules from the data as follows:

\begin{verbatim}
  > rules <- apriori(data)
\end{verbatim}

parameter specification:
\begin{verbatim}
  confidence minval maxaremaval originalSupport support minlen maxlen
target 0.8 0.1 1 none FALSE TRUE 0.1 1
\end{verbatim}

10 rules
ext
FALSE

algorithmic control:
filter tree heap memopt load sort verbose
There are several points to highlight in the results:

- As you can see from the display, we are using the default settings (confidence 0.8, and so on)
- We found 15,000 transactions for three items (picked from the 655 total items available)
- We generated five rules

We can examine the rules that were generated as follows:

```r
> rules

set of 5 rules
> inspect(rules)

lhs rhs  support  confidence lift
1 {semi.finished.bread=} => {margarine=} 0.2278522          1
     2.501226
2 {semi.finished.bread=} => {ready.soups=} 0.2278522          1
     1.861385
3 {margarine=}          => {ready.soups=} 0.3998039          1
     1.861385
4 {semi.finished.bread=, 
margarine=}           => {ready.soups=} 0.2278522          1
     1.861385
5 {semi.finished.bread=, 
ready.soups=}         => {margarine=} 0.2278522          1
     2.501226
```

The code has been slightly reformatted for readability.

Looking over the rules, there is a clear connection between buying bread, soup, and margarine—at least in the market where and when the data was gathered.
If we change the parameters (thresholds) used in the calculation, we get a different set of rules. For example, check the following code:

```r
> rules <- apriori(data, parameter = list(supp = 0.001, conf = 0.8))
```

This code generates over 500 rules, but they have questionable meaning as we now have the rules with 0.001 confidence.

**Questions**

**Factual**

- How do you decide whether to use `kmeans` or `kdemoids`?
- What is the significance of the boxplot layout? Why does it look that way?
- Describe the underlying data produced in the outliers for the `iris` data, given the density plot.
- What are the extract rules for other items in the market dataset?

**When, how, and why?**

- What is the risk of not vetting the outliers that are detected for the specific domain? Shouldn't the calculation always work?
- Why do we need to exclude the `iris` category column from the outlier detection algorithm? Can it be used in some way when determining outliers?
- Can you come up with a scenario where the market basket data and rules we generated were not applicable to the store you are working with?

**Challenges**

- I found it difficult to develop test data for outliers in two dimensions that both occurred in the same instance using random data. Can you develop a test that would always have several outliers in at least two dimensions that occur in the same instance?
- There is a good dataset on the Internet regarding passenger data on the Titanic. Generate the rules regarding the possible survival of the passengers.
Summary
In this chapter, we discussed cluster analysis, anomaly detection, and association rules. In cluster analysis, we use k-means clustering, k-medoids clustering, hierarchical clustering, expectation-maximization, and density estimation. In anomaly detection, we found outliers using built-in R functions and developed our own specialized R function. For association rules, we used the apriori package to determine the associations amongst datasets.

In the next chapter, we will cover data mining for sequences.
Where to buy this book

You can buy R for Data Science from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.