11

Segmentation: Clustering and Classification

In this chapter, we tackle a canonical marketing research problem: finding, assessing, and predicting customer segments. In previous chapters we've seen how to assess relationships in the data (Chap. 4), compare groups (Chap. 5), and assess complex multivariate models (Chap. 10). In a real segmentation project, one would use those methods to ensure that data has appropriate multivariate structure, and then begin segmentation analysis.

Segmentation is not a well-defined process and analysts vary in their definitions of segmentation as well as their approaches and philosophy. The model in this chapter demonstrates our approach using basic models in R. As always, this should be supplemented by readings that we suggest at the end of the chapter.

We start with a warning: we have definite opinions about segmentation and what we believe are common misunderstandings and poor practices. We hope you'll be convinced by our views—but even if not, the methods here will be useful to you.

11.1 Segmentation Philosophy

The general goal of market segmentation is to find groups of customers that differ in important ways associated with product interest, market participation, or response to marketing efforts. By understanding the differences among groups, a marketer can make better strategic choices about opportunities, product definition, and positioning, and can engage in more effective promotion.

11.1.1 The Difficulty of Segmentation

The definition of segmentation above is a textbook description and does not reflect what is most difficult in a segmentation project: finding actionable business

© Springer International Publishing Switzerland 2015

C. Chapman, E.M. Feit, *R for Marketing Research and Analytics*, Use R!,
DOI 10.1007/978-3-319-14436-8_11

outcomes. It is not particularly difficult to find *groups* within consumer data; indeed, in this chapter we see several ways to do this, all of which "succeed" according to one statistical criterion or another. Rather, the difficulty is to ensure that the outcome is *meaningful* for a particular business need.

It is outside the range of this book to address the question of business need in general. However, we suggest that you ask a few questions along the following lines. If you were to find segments, what would you do about them? Would anyone in your organization use them? Why and how? Are the differences found large enough to be meaningful for your business? Among various solutions you might find, are there organizational efforts or politics that would make one solution more or less influential than another?

There is no magic bullet to find the "right" answer. In computer science the *no free lunch theorem* says that "for both static and time-dependent optimization problems, the average performance of any pair of algorithms across all possible problems is identical" [167]. For segmentation this means that there is no all-purpose method or algorithm that is a priori preferable to others. This does not mean that the choice of a method is irrelevant or arbitrary; rather, one cannot necessarily determine in advance which approach will work best for a novel problem. As a form of optimization, segmentation is likely to require an iterative approach that successively tests and improves its answer to a business need.

Segmentation is like slicing a pie, and any pie might be sliced in an infinite number of ways. Your task as an analyst is to consider the infinity of possible data that might be gathered, the infinity of possible groupings of that data, and the infinity of possible business questions that might be addressed. Your goal is to find a solution within those infinities that represents real differences in the data and that informs and influences real business decisions.

Statistical methods are only part of the answer. It often happens that a "stronger" statistical solution poses complexity that makes it impossible to implement in a business context while a slightly "weaker" solution illuminates the data with a clear story and fits the business context so well that it can have broad influence.

To maximize chances of finding such a model, we recommend that an analyst expects—and prepares management to understand—the need to iterate analyses. A segmentation project is not a matter of "running a segmentation study" or "doing segmentation analysis on the data." Rather, it is likely to take multiple rounds of data collection and analysis to determine the important data that should be collected in the first place, to refine and test the solutions, and to conduct rounds of interpretation with business stakeholders to ensure that the results are actionable.

11.1.2 Segmentation as Clustering and Classification

In this chapter, we demonstrate several methods in R that will help you start with segmentation analysis. We explore two distinct yet related areas of statistics:

clustering or *cluster analysis* and *classification*. These are the primary branches of what is sometimes called *statistical learning*, i.e., learning from data through statistical model fitting.

A key distinction in statistical learning is whether the method is *supervised* or *unsupervised*. In *supervised learning*, a model is presented with observations whose outcome status (dependent variable) is known, with a goal to predict that outcome from the independent variables. For example, we might use data from previous direct marketing campaigns—with a known outcome of whether each target responded or not, plus other predictor variables—to fit a model that predicts likelihood of response in a new campaign. We refer to this process as *classification*.

In *unsupervised learning* we do not know the outcome groupings but attempt to discover them from structure in the data. For instance, we might explore a direct marketing campaign and ask, "Are there groups that differ in how and when they respond to offers? If so, what are the characteristics of those groups?" We use the term *clustering* for this approach.

Clustering and classification are both useful in segmentation projects. Stakeholders often view segmentation as discovering groups in the data in order to derive new insight about customers. This obviously suggests clustering approaches because the possible customer groups are unknown. Still, classification approaches are also useful in such projects for at least two reasons: there may be outcome variables of interest that are known (such as observed in-market response) that one wishes to predict from segment membership, and if you use clustering to discover groups you will probably want to predict (i.e., classify) future responses into those groups. Thus, we view clustering and classification as complementary approaches.

A topic we do not address is how to determine what data to use for clustering, the observed *basis variables* that go into the model. That is primarily a choice based on business need, strategy, and data availability. Still, you can use the methods here to evaluate different sets of such variables. If you have a large number of measures available and need to determine which ones are most important, the *variable importance* assessment method we review in Sect. 11.4.3 might assist. Aside from that, we assume in this chapter that the basis variables have been determined (and we use the customer relationship data from Chap. 5).

There are hundreds of books, thousands of articles, and scores of R packages for clustering and classification methods, all of which propose hundreds of approaches with—as we noted above—no single "best" method. This chapter cannot cover clustering or classification in a comprehensive way, but we can give an introduction that will get you started, teach you the basics, accelerate your learning, and help you avoid some traps. As you will see, in most cases the process of fitting such models in R is extremely similar from model to model.

11.2 Segmentation Data

We use the segmentation data (object `seg.df`) from Chap. 5. If you saved that data in Sect. 5.1.4, you can reload it:

```
> load("~/segdf-Rintro-Ch5.RData")
> seg.raw <- seg.df
> seg.df  <- seg.raw[ , -7]   # remove the known segment assignments
```

Otherwise, you could download the data set from the book website:

```
> seg.raw <- read.csv("http://goo.gl/qw303p")
> seg.df  <- seg.raw[ , -7]     # remove the known segment assignments
```

As you may recall from Chap. 5, this is a simulated data set with four identified segments of customers for a subscription product, and contains a few variables that are similar to data from typical consumer surveys. Each observation has the simulated respondent's age, gender, household income, number of kids, home ownership, subscription status, and assigned segment membership. In Chap. 5, we saw how to simulate this data and how to examine group differences within it. Other data sources that are often used for segmentation are customer relationship management (CRM) records, attitudinal surveys, product purchase and usage, and more generally, any data set with observations about customers.

The original data `seg.raw` contains "known" segment assignments that have been provided for the data from some other source (as might occur from some human coding process). Because our task here is to discover segments, we create a copy `seg.df` that omits those assignments (omitting column 7), so we don't accidentally include the known values when exploring applying segmentation methods. (Later, in the classification section, we will use the correct assignments because they are needed to train the classification models.)

We check the data after loading:

```
> 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 ...
```

We use the subscription segment data in this chapter for two purposes: to examine clustering methods that find intrinsic groupings (unsupervised learning), and to show how classification methods learn to predict group membership from known cases (supervised learning).

11.3 Clustering

We examine four clustering procedures that are illustrative of the hundreds of available methods. You'll see that the general procedure for finding and evaluating clusters in R is similar across the methods.

To begin, we review two *distance-based* clustering methods, `hclust()` and `kmeans()`. Distance-based methods attempt to find groups that minimize the distance between members within the group, while maximizing the distance of members from other groups. `hclust()` does this by modeling the data in a tree structure, while `kmeans()` uses group centroids (central points).

Then we examine *model-based* clustering methods, `Mclust()` and `poLCA()`. Model-based methods view the data as a mixture of groups sampled from different distributions, but whose original distribution and group membership has been "lost" (i.e., is unknown). These methods attempt to model the data such that the observed variance can be best represented by a small number of groups with specific distribution characteristics such as different means and standard deviations. `Mclust()` models the data as a mixture of Gaussian (normal) variables, while `poLCA()` uses a latent class model with categorical (nominal) variables.

11.3.1 The Steps of Clustering

Clustering analysis requires two stages: finding a proposed cluster solution and evaluating that solution for one's business needs. For each method we go through the following steps:

- Transform the data if needed for a particular clustering method; for instance, some methods require all numeric data (e.g., `kmeans()`, `mclust()`) or all categorical data (e.g., `poLCA()`).
- Compute a distance matrix if needed; some methods require a precomputed matrix of similarity in order to group observations (e.g., `hclust()`).
- Apply the clustering method and save its result to an object. For some methods this requires specifying the number (K) of groups desired (e.g., `kmeans()`, `poLCA()`).
- For some methods, further parse the object to obtain a solution with K groups (e.g., `hclust()`).
- Examine the solution in the model object with regard to the underlying data, and consider whether it answers a business question.

As we've already argued, the most difficult part of that process is the last step: establishing whether a proposed statistical solution answers a business need. Ultimately, a cluster solution is largely just a vector of purported group assignments for each observation, such as "1, 1, 4, 3, 2, 3, 2, 2, 4, 1, 4 ...." It is up to you to figure out whether that tells a meaningful story for your data.

11.3.1.1 A Quick Check Function

We recommend that you think hard about how you would know whether the solution—assignments of observations to groups—that is proposed by a clustering method is useful for your business problem. Just because some grouping is proposed by an algorithm does not mean that it will help your business. One way we often approach this is to write a simple function that summarizes the data and allows quick inspection of the high-level differences between groups.

A segment inspection function may be complex depending on the business need and might even include plotting as well as data summarization. For purposes here we use a simple function that reports the mean by group. We use `mean` here instead of a more robust metric such as `median` because we have several binary variables and `mean()` easily shows the mixture proportion for them (i.e., 1.5 means a 50 % mix of 1 and 2). A very simple function is:

```
> seg.summ <- function(data, groups) {
+   aggregate(data, list(groups), function(x) mean(as.numeric(x)))
+ }
```

This function first splits the data by reported group (`aggregate(..., list (groups), ...)`). An anonymous function (`function(x) ...`) then converts all of a group's data to numeric (`as.numeric(x)`) and computes its `mean()`. Here's an example using the known segments from `seg.raw`:

```
> seg.summ(seg.df, seg.raw$Segment)
     Group.1      age gender   income     kids  ownHome subscribe
1  Moving up 36.33114   1.30 53090.97 1.914286 1.328571     1.200
2 Suburb mix 39.92815   1.52 55033.82 1.920000 1.480000     1.060
3  Travelers 57.87088   1.50 62213.94 0.000000 1.750000     1.125
4  Urban hip 23.88459   1.60 21681.93 1.100000 1.200000     1.200
```

This simple function will help us to inspect cluster solutions efficiently. It is not intended to be a substitute for detailed analysis—and it takes shortcuts such as treating categorical variables as numbers, which is inadvisable except for analysts who understand what they're doing—yet it provides a quick first check of whether there is something interesting (or uninteresting) occurring in a solution.

With a summary function of this kind we are easily able to answer the following questions related to the business value of a proposed solution:

- Are there obvious differences in group means?
- Does the differentiation point to some underlying story to tell?
- Do we see immediately odd results such as a mean equal to the value of one data level?

Why not just use a standard R function such as `by()` or `aggregate()`? There are several reasons. Writing our own function allows us to minimize typing by providing a short command. By providing a consistent and simple interface, it reduces risk of error. And it is extensible; as an analysis proceeds, we might decide to add to the function, expanding it to report variance metrics or to plot results, without needing to change how we invoke it.

11.3.2 Hierarchical Clustering: `hclust()` Basics

Hierarchical clustering is a popular method that groups observations according to their similarity. The `hclust()` method is one way to perform this analysis in R. `hclust()` is a distance-based algorithm that operates on a *dissimilarity* matrix, an N-by-N matrix that reports a metric for the *distance* between each pair of observations.

The hierarchical clustering method begins with each observation in its own cluster. It then successively joins neighboring observations or clusters one at a time according to their distances from one another, and continues this until all observations are linked. This process of repeatedly joining observations and groups is known as an *agglomerative* method. Because it is both very popular and exemplary of other methods, we present hierarchical clustering in more detail than the other clustering algorithms.

The primary information in hierarchical clustering is the *distance* between observations. There are many ways to compute distance, and we start by examining the best-known method, the *Euclidean distance*. For two observations (vectors) X and Y, the Euclidean distance d is:

$$d = \sqrt{\sum (X - Y)^2}. \tag{11.1}$$

For single pairs of observations, such as $X = \{1,2,3\}$ and $Y = \{2,3,2\}$ we can compute the distance easily in R:

```
> c(1,2,3) - c(2,3,2)                  # vector of differences
[1] -1 -1  1
> sum((c(1,2,3) - c(2,3,2))^2)         # the sum of squared differences
[1] 3
> sqrt(sum((c(1,2,3) - c(2,3,2))^2))   # root sum of squares
[1] 1.732051
```

When there are many pairs, this can be done with the `dist()` function. Let's check it first for the simple X,Y example, using `rbind()` to group these vectors as observations (rows):

```
> dist(rbind(c(1,2,3), c(2,3,2)))
         1
2 1.732051
```

The row and column labels tell us that `dist()` is returning a matrix for observation 1 (column) by observation 2 (row).

A limitation is that Euclidean distance is only defined when observations are numeric. In our data `seg.df` it is impossible to compute the distance between `Male` and `Female` (a fact many people suspect even before studying statistics). If we did not care about the factor variables, then we could compute Euclidean distance using only the numeric columns.

For example, we can select the three numeric columns in `seg.df`, calculate the distances, and then look at a matrix for just the first five observations as follows:

```
> d <- dist(seg.df[, c("age", "income", "kids")])
> as.matrix(d)[1:5, 1:5]
          1         2         3         4         5
1     0.000 13936.531  5313.626 31559.178 29870.205
2 13936.531     0.000  8622.906 45495.698 43806.727
3  5313.626  8622.906     0.000 36872.800 35183.828
4 31559.178 45495.698 36872.800     0.000  1688.977
5 29870.205 43806.727 35183.828  1688.977     0.000
```

As expected, the distance matrix is symmetric, and the distance of an observation from itself is 0.

For `seg.df` we cannot assume that factor variables are irrelevant to our cluster definitions; it is better to use *all* the data. The `daisy()` function in the `cluster` package [108] works with mixed data types by rescaling the values, so we use that instead of Euclidean distance:

```
> library(cluster)                  # daisy works with mixed data types
> seg.dist <- daisy(seg.df)
```

We inspect the distances computed by `daisy()` by coercing the resulting object to a matrix and selecting the first few rows and columns:

```
> as.matrix(seg.dist)[1:5, 1:5]
          1         2         3         4         5
1 0.0000000 0.2532815 0.2329028 0.2617250 0.4161338
2 0.2532815 0.0000000 0.0679978 0.4129493 0.3014468
3 0.2329028 0.0679978 0.0000000 0.4246012 0.2932957
4 0.2617250 0.4129493 0.4246012 0.0000000 0.2265436
5 0.4161338 0.3014468 0.2932957 0.2265436 0.0000000
```

The distances look reasonable (zeroes on the diagonal, symmetric, scaled [0, 1]) so we proceed to the hierarchical cluster method itself, invoking `hclust()` on the dissimilarity matrix:

```
> seg.hc <- hclust(seg.dist, method="complete")
```

We use the *complete* linkage method, which evaluates the distance between every member when combining observations and groups.

A simple call to `plot()` will draw the `hclust` object:

```
> plot(seg.hc)
```

The resulting tree for all $N = 300$ observations of `seg.df` is shown in Fig. 11.1.

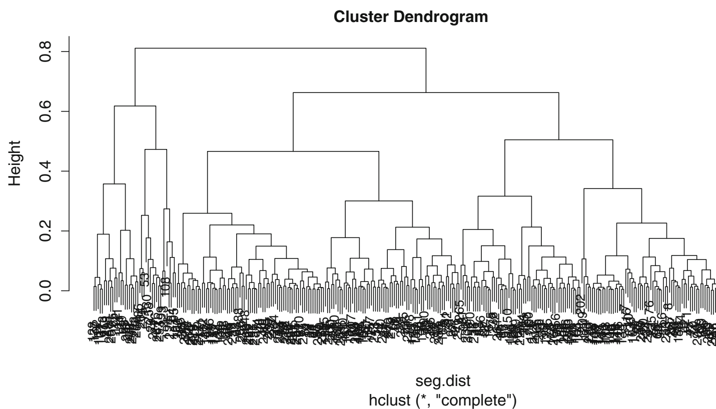

Fig. 11.1. Complete dendrogram for the segmentation data, using `hclust()`.

A hierarchical dendrogram is interpreted primarily by height and where observations are joined. The height represents the dissimilarity between elements that are joined. At the lowest level of the tree in Fig. 11.1 we see that elements are combined into small groups of 2–10 that are relatively similar, and then those groups are successively combined with less similar groups moving up the tree. The horizontal ordering of branches is not important; branches could exchange places with no change in interpretation.

Figure 11.1 is difficult to read, so it is helpful to zoom in on one section of the chart. We can cut it at a specified location and plot just one branch as follows. We coerce it to a dendrogram object (`as.dendrogram(...)`), cut it at a certain height (`h=...`), and select the resulting branch that we want (`...$lower[[1]]`).

```
> plot(cut(as.dendrogram(seg.hc), h=0.5)$lower[[1]])
```

The result is shown in Fig. 11.2, where we are now able to read the observation labels (which defaults to the row names—usually the row numbers—of observations in the data frame). Each node at the bottom represents one customer, and the brackets show how each has been grouped progressively with other customers.

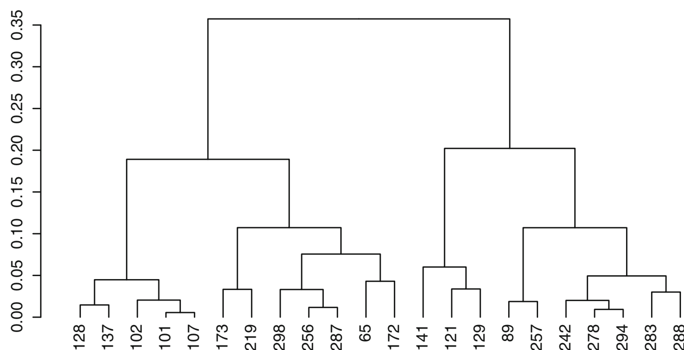

Fig. 11.2. A close up view of the left-most branch from Fig. 11.1.

We can check the similarity of observations by selecting a few rows listed in Fig. 11.2. Observations 101 and 107 are represented as being quite similar because they are linked at a very low height, as are observations 278 and 294. On the other hand, observations 173 and 141 are only joined at the highest level of this branch and thus should be relatively dissimilar. We can check those directly:

```
> seg.df[c(101, 107), ]  # similar
         age gender   income kids ownHome subscribe
101 24.73796   Male 18457.85    1   ownNo    subYes
107 23.19013   Male 17510.28    1   ownNo    subYes
> seg.df[c(278, 294), ]  # similar
         age gender   income kids ownHome subscribe
278 36.23860 Female 46540.88    1   ownNo    subYes
294 35.79961 Female 52352.69    1   ownNo    subYes
> seg.df[c(173, 141), ]  # less similar
         age gender   income kids ownHome subscribe
173 64.70641   Male 45517.15    0   ownNo    subYes
141 25.17703 Female 20125.80    2   ownNo    subYes
```

The first two sets—observations that are neighbors in the dendrogram—are similar on all variables (age, gender, income, etc.). The third set—observations taken from widely separated branches—differs substantially on the first four variables.

Finally, we might check one of the goodness-of-fit metrics for a hierarchical cluster solution. One method is the *cophenetic correlation* coefficient (CPCC), which assesses how well a dendrogram (in this case `seg.hc`) matches the true distance metric (`seg.dist`) [145]. We use `cophenetic()` to get the distances from the dendrogram, and compare it to the `dist()` metrics with `cor()`:

```
> cor(cophenetic(seg.hc), seg.dist)
[1] 0.7682436
```

CPCC is interpreted similarly to Pearson's r. In this case, CPCC > 0.7 indicates a relatively strong fit, meaning that the hierarchical tree represents the distances between customers well.

11.3.3 Hierarchical Clustering Continued: Groups from `hclust()`

How do we get specific segment assignments? A dendrogram can be cut into clusters at any height desired, resulting in different numbers of groups. For instance, if Fig. 11.1 is cut at a height of 0.7, there are $K = 2$ groups (draw a horizontal line at 0.7 and count how many branches it intersects; each cluster below is a group), while cutting at height of 0.4 defines $K = 7$ groups.

Because a dendrogram can be cut at any point, the analyst must specify the number of groups desired. We can see where the dendrogram would be cut by overlaying its `plot()` with `rect.hclust()`, specifying the number of groups we want (`k=...`):

```
> plot(seg.hc)
> rect.hclust(seg.hc, k=4, border="red")
```

The $K = 4$ solution is shown in Fig. 11.3.

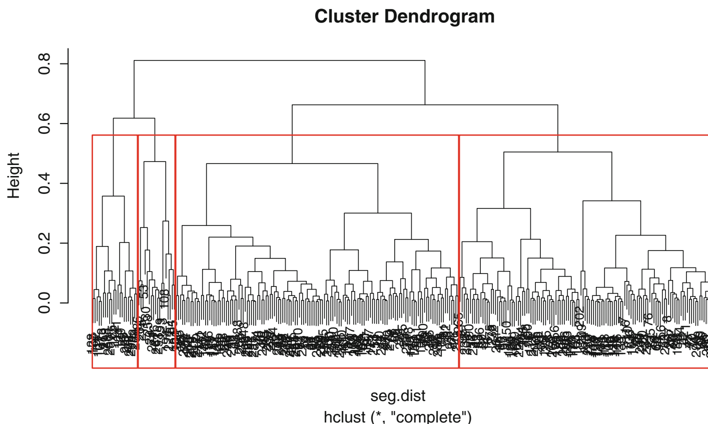

Fig. 11.3. The result of cutting Fig. 11.1 into $K = 4$ groups.

We obtain the assignment vector for observations using `cutree()`:

```
> seg.hc.segment <- cutree(seg.hc, k=4)      # membership vector for 4 groups
> table(seg.hc.segment)
seg.hc.segment
  1   2   3   4
124 136  18  22
```

We see that groups 1 and 2 dominate the assignment. Note that the class labels (1, 2, 3, 4) are in arbitrary order and are not meaningful in themselves. `seg.hc.segment` is the vector of group assignments.

We use our custom summary function `seg.summ()`, defined above, to inspect the variables in `seg.df` with reference to the four clusters:

```
> seg.summ(seg.df, seg.hc.segment)
  Group.1      age   gender   income     kids  ownHome subscribe
1       1 40.78456 2.000000 49454.08 1.314516 1.467742         1
2       2 42.03492 1.000000 53759.62 1.235294 1.477941         1
3       3 44.31194 1.388889 52628.42 1.388889 2.000000         2
4       4 35.82935 1.545455 40456.14 1.136364 1.000000         2
```

We see that groups 1 and 2 are distinct from 3 and 4 due to subscription status. Among those who do not subscribe, group 1 is all male (`gender=2` as in `levels(seg.df$gender)`) while group 1 is all female. Subscribers are differentiated into those who own a home (group 3) or not (group 4).

Is this interesting from a business point of view? Probably not. Imagine describing the results to a set of executives: "Our advanced hierarchical analysis in R examined consumers who don't yet subscribe and found two segments to target! The segments are known as 'Men' and 'Women.'" Such insight is unlikely to win the analyst a promotion.

We confirm this with a quick plot of `gender` by `subscribe` with all of the observations colored by segment membership. To do this, we use a trick: we convert the factor variables to numeric, and call the `jitter()` function to add a bit of noise and prevent all the cases from being plotted at the same positions (namely at exactly four points: $(1,1)$, $(1,2)$, $(2,1)$, and $(2,2)$). We color the points by segment with `col=seg.hc.segment`, and label the axes with more meaningful labels:

```
> plot(jitter(as.numeric(seg.df$gender)) ~
+      jitter(as.numeric(seg.df$subscribe)),
+      col=seg.hc.segment, yaxt="n", xaxt="n", ylab="", xlab="")
> axis(1, at=c(1, 2), labels=c("Subscribe: No", "Subscribe: Yes"))
> axis(2, at=c(1, 2), labels=levels(seg.df$gender))
```

The resulting plot is shown in Fig. 11.4, where we see clearly that the non-subscribers are broken into two segments (colored red and black) that are perfectly correlated with gender. We should point out that such a plot is a quick hack, which we suggest only for rapid inspection and debugging purposes.

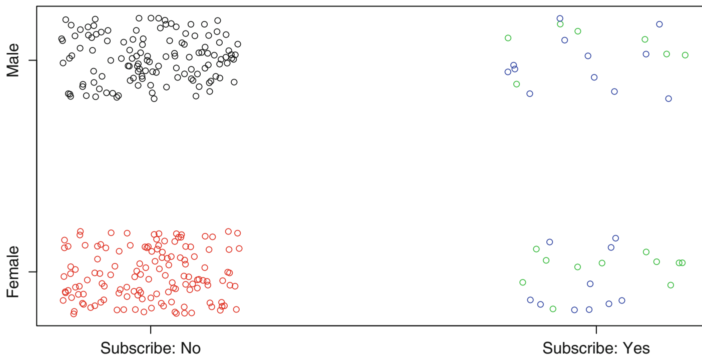

Fig. 11.4. Plotting the 4-segment solution from `hclust()` by gender and subscription status, with color representing segment membership. We see the uninteresting result that non-subscribers are simply divided into two segments purely on the basis of gender.

Why did `hclust()` find a result that is so uninteresting? That may be answered in several ways. For one thing, machine learning techniques often take the path of least resistance and serve up obvious results. In this specific case, the scaling in `daisy()` rescales variables to $[0,1]$ and this will make two-category factors (gender, subscription status, home ownership) more influential. Overall, this demonstrates why you should expect to try several methods and iterate in order to find something useful.

11.3.4 Mean-Based Clustering: `kmeans()`

K-means clustering attempts to find groups that are most compact, in terms of the mean sum-of-squares deviation of each observation from the multivariate center (*centroid*) of its assigned group. Like hierarchical clustering, k-means is a very popular approach.

Because it explicitly computes a mean deviation, k-means clustering relies on Euclidean distance. Thus it is only appropriate for numeric data or data that can be reasonably coerced to numeric. In our `seg.df` data, we have a mix of numeric and binary factors. Unlike higher-order categorical variables, binary factors can be coerced to numeric with no alteration of meaning.

Although it is not optimal to cluster binary values with k-means, given that we have a mixture of binary and numeric data, we might attempt it. Our first step is to create a variant of `seg.df` that is recoded to numeric. We make a copy of `seg.df` and use `ifelse()` to recode the binary factors:

```
> seg.df.num <- seg.df
> seg.df.num$gender    <- ifelse(seg.df$gender=="Male", 0, 1)
> seg.df.num$ownHome   <- ifelse(seg.df$ownHome=="ownNo", 0, 1)
> seg.df.num$subscribe <- ifelse(seg.df$subscribe=="subNo", 0, 1)
```

```
> summary(seg.df.num)
      age            gender            income            kids           ownHome
 Min.   :19.26   Min.   :0.0000   Min.   : -5183   Min.   :0.00   Min.   :0.00
 1st Qu.:33.01   1st Qu.:0.0000   1st Qu.: 39656   1st Qu.:0.00   1st Qu.:0.00
 Median :39.49   Median :0.0000   Median : 52014   Median :1.00   Median :0.00
 ...
```

There are several ways to recode data, but `ifelse()` is simple and explicit for binary data.

We now run the `kmeans()` algorithm, which specifically requires specifying the number of clusters to find. We ask for four clusters with `centers=4`:

```
> set.seed(96743)
> seg.k <- kmeans(seg.df.num, centers=4)
```

We use our custom function `seg.summ()` to do a quick check of the data by proposed group, where cluster assignments are found in the `$cluster` vector inside the `seg.k` model:

```
> seg.summ(seg.df, seg.k$cluster)
  Group.1      age   gender   income      kids  ownHome subscribe
1       1 56.37245 1.428571 92287.07 0.4285714 1.857143  1.142857
2       2 29.58704 1.571429 21631.79 1.0634921 1.301587  1.158730
3       3 44.42051 1.452632 64703.76 1.2947368 1.421053  1.073684
4       4 42.08381 1.454545 48208.86 1.5041322 1.528926  1.165289
```

Unlike with `hclust()` we now see some interesting differences; the groups appear to vary by age, gender, kids, income, and home ownership. For example, we can visually check the distribution of income according to segment (which `kmeans()` stored in `seg.k$cluster`) using `boxplot()`:

```
> boxplot(seg.df.num$income ~ seg.k$cluster, ylab="Income", xlab="Cluster")
```

The result is Fig. 11.5, which shows substantial differences in income by segment. Note that in clustering models, the group labels are in arbitrary order, so don't worry if your solution shows the same pattern with different labels.

We visualize the clusters by plotting them against a dimensional plot. `clusplot()` will perform dimensional reduction with principal components or multidimensional scaling as the data warrant, and then plot the observations with cluster membership identified (see Chap. 8 to review principal component analysis and plotting.) We use `clusplot` from the `cluster` package with arguments to `color` the groups, `shade` the ellipses for group membership, label only the groups (not the individual points) with `labels=4`, and omit distance lines between groups (`lines=0`):

```
> library(cluster)
> clusplot(seg.df, seg.k$cluster, color=TRUE, shade=TRUE,
+          labels=4, lines=0, main="K-means cluster plot")
```

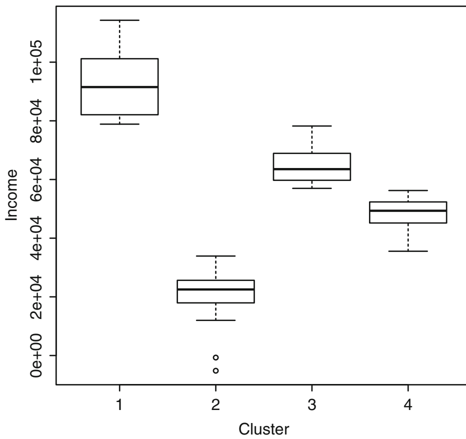

Fig. 11.5. Boxplot of income by cluster as found with `kmeans()`.

The code produces the plot in Fig. 11.6, which plots cluster assignment by color and ellipses against the first two principal components of the predictors (see Sect. 8.2.2). Groups 3 and 4 are largely overlapping (in this dimensional reduction) while group 1 and especially group 2 are modestly differentiated.

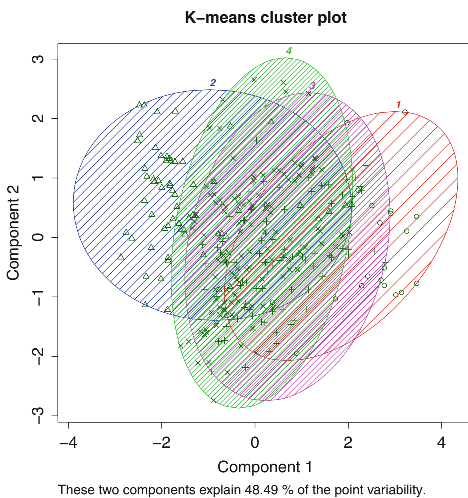

Fig. 11.6. Cluster plot created with `clusplot()` for the four group solution from `kmeans()`. This shows the observations on a multidimensional scaling plot with group membership identified by the ellipses.

Overall, this is a far more interesting cluster solution for our segmentation data than the `hclust()` proposal. The groups here are clearly differentiated on key variables such as age and income. With this information, an analyst might cross-reference the group membership with key variables (as we did using our `seg.summ()` function and then look at the relative differentiation of the groups (as in Fig. 11.6).

This may suggest a business strategy. In the present case, for instance, we see that group 1 is modestly well differentiated, and has the highest average income. That may make it a good target for a potential campaign. Many other strategies are possible, too; the key point is that the analysis provides interesting options to consider.

A limitation of k-means analysis is that it requires specifying the number of clusters, and it can be difficult to determine whether one solution is better than another. If we were to use k-means for the present problem, we would repeat the analysis for $k = 3$, 4, 5, and so forth, and determine which solution gives the most useful result for our business goals.

One might wonder whether the algorithm itself can suggest how many clusters are in the data. Yes! To see that, we turn next to model-based clustering.

11.3.5 Model-Based Clustering: `Mclust()`

The key idea for model-based clustering is that observations come from groups with different statistical distributions (such as different means and variances). The algorithms try to find the best set of such underlying distributions to explain the observed data. We use the `mclust` package [53, 54] to demonstrate this.

Such models are also known as "mixture models" because it is assumed that the data reflect a mixture of observations drawn from different populations, although we don't know which population each observation was drawn from. We are trying to estimate the underlying population parameters and the mixture proportion. `mclust` models such clusters as being drawn from a mixture of normal (also known as *Gaussian*) distributions.

As you might guess, because `mclust` models data with normal distributions, it uses only numeric data. We use the numeric data frame `seg.df.num` that we adapted for `kmeans()` in Sect. 11.3.4; see that section for the code if needed. The model is estimated with `Mclust()` (note the capital letter for the fitting function, as opposed to the package name):

```
> library(mclust)
> seg.mc <- Mclust(seg.df.num)
> summary(seg.mc)
----------------------------------------------------
Gaussian finite mixture model fitted by EM algorithm
----------------------------------------------------

Mclust EEV (ellipsoidal, equal volume and shape) model with 3 components:

 log.likelihood   n df       BIC       ICL
      -5256.222 300 71 -10917.41 -10955.48

Clustering table:
  1   2   3
111 115  74
```

This tells us that the data are estimated to have three clusters (components) with the sizes as shown in the table. `Mclust()` compared a variety of different mixture shapes and concluded that an ellipsoidal model (modeling the data as multivariate ellipses) fit best.

We also see log-likelihood information, which we can use to compare models. We try a 4-cluster solution by telling `Mclust()` the number of clusters we want with the `G=4` argument:

```
> seg.mc4 <- Mclust(seg.df.num, G=4)
> summary(seg.mc4)
...
Mclust EEI (diagonal, equal volume and shape) model with 4 components:

 log.likelihood   n df       BIC       ICL
      -5455.346 300 33 -11098.92 -11131.54

Clustering table:
  1   2   3   4
 45  54  23 178
```

Forcing it to find four clusters resulted in quite a different model, with lower log-likelihood, a different multivariate pattern (diagonal), and no obvious correspondence in the cluster table (for instance, it's not simply that one of the groups in the 3-cluster solution was split into two).

11.3.6 Comparing Models with BIC()

We compare the 3-cluster and 4-cluster models using the Bayesian information criterion (BIC) [129] with `BIC(model1, model2)`:

```
> BIC(seg.mc, seg.mc4)
        df      BIC
seg.mc  71 10917.41
seg.mc4 33 11098.92
```

The difference between the models is 181. The key point to interpreting BIC is to remember this: the *lower* the value of BIC, on an infinite number line, the better. BIC of $-1{,}000$ is better than BIC of -990; and BIC of 60 is better than BIC of 90.

There is one important note when interpreting BIC in R: unfortunately, some functions return the *negative* of BIC, which would then have to be interpreted in the opposite direction. We see above that `BIC()` reports positive values while `Mclust()` returns the same values in the negative direction. If you are ever unsure of the direction to interpret, use the `BIC()` function and interpret as noted (lower values are better). Alternatively, you could also check the log-likelihood values, where *higher* log-likelihood values are better (e.g., $-1{,}000$ is better than $-1{,}100$).

With that in mind, differences in BIC may be interpreted as shown in Table 11.1. Comparing the present models, we see that the `Mclust()` solution with three clusters (BIC = 10,917) is a much stronger fit than the model with 4 clusters (BIC = 11,098) because it is lower by 181. That doesn't mean that the 3-cluster model is *correct*; there's no absolute standard for such a statement. Rather, it means that between just these two models, as found by `Mclust()`, the 3-cluster solution has much stronger evidence in the data.

Table 11.1. Interpretation of the Bayesian information criterion (BIC) when comparing two models

BIC difference	Odds of model superiority (%)	Strength of the evidence
0–2	50–75	Weak
2–6	75–95	Positive
6–10	95–99	Strong
>10	>99	Very strong

Lower BIC is better, and the difference in BIC indicates the strength of evidence. Adapted from Raftery [129, p. 139]

Will the 3-cluster solution provide useful insight for the business? We check the quick summary and plot the clusters:

```
> seg.summ(seg.df, seg.mc$class)
  Group.1      age   gender   income     kids  ownHome subscribe
1       1 37.11167 1.405405 53142.71 1.747748 1.099099  1.270270
2       2 33.67599 1.530435 41315.55 1.626087 1.634783  1.000000
3       3 59.02380 1.500000 62578.80 0.000000 1.770270  1.135135
> library(cluster)
> clusplot(seg.df, seg.mc$class, color=TRUE, shade=TRUE,
+          labels=4, lines=0, main="Model-based cluster plot")
```

The plot is shown in Fig. 11.7.

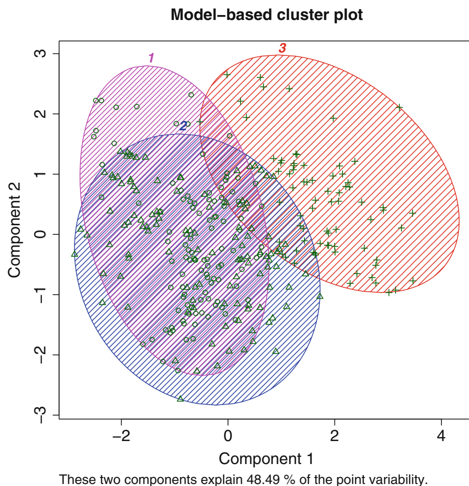

Fig. 11.7. A cluster plot using `clusplot()` for the 3-cluster model from `Mclust()`. Group 3 is highly differentiated on the first two multivariate dimensional components (*X* and *Y* axes).

When we compare the `Mclust()` solution to the one found by `kmeans()`, there are arguments for and against each. The 4-cluster k-means solution had much crisper differentiation on demographics (Sect. 11.3.4). On the other hand, the most clearly differentiated segment (segment 2; cf. Fig. 11.6) had the lowest income and thus might be more difficult to sell to (or not—it depends on the product or service).

Looking closely at the `Mclust()` solution, we see that one of the groups is nicely differentiated (group 3), but the demographic differences reported in `seg.summ()` are not particularly interesting. Everyone in Group 2 is a non-subscriber, Group 3 has no kids, and Group 1 mixes everyone else. As always, the ultimate value depends more on one'e strategy, business case, and modes available to target respondents than it does on the statistical solution in itself. The statistics provide information about how customers are similar and different, not a definitive answer.

11.3.7 Latent Class Analysis: `poLCA()`

Latent class analysis (LCA) is similar to mixture modeling in the assumption that differences are attributable to unobserved groups that one wishes to uncover. In this section we take a look at the `poLCA` package for *po*lytomous (i.e., categorical) LCA [105].

Whereas `mclust` and `kmeans()` work with numeric data, and `hclust()` depends on the distance measure, `poLCA` uses only categorical variables. To demonstrate it here, we adopt an opposite strategy from our procedure with k-means and `mclust` and convert our data `seg.df` to be all categorical data before analyzing it.

There are several approaches to convert numeric data to factors, but for purposes here we simply recode everything as binary with regard to a specified cutting point (for instance, to recode as 1 for income below some cutoff and 2 above that). In the present case, we split each variable at the `median()` and recode using `ifelse()` and `factor()` (we'll see a more general approach to recoding numeric values with `cut()` in Sect. 12.4.1):

```
> seg.df.cut <- seg.df
> seg.df.cut$age    <- factor(ifelse(seg.df$age < median(seg.df$age), 1, 2))
> seg.df.cut$income <- factor(ifelse(seg.df$income < median(seg.df$income),
+                                    1, 2))
> seg.df.cut$kids   <- factor(ifelse(seg.df$kids < median(seg.df$kids), 1, 2))
> summary(seg.df.cut)
 age        gender     income   kids      ownHome      subscribe
 1:150   Female:157   1:150   1:121   ownNo :159   subNo :260
 2:150   Male  :143   2:150   2:179   ownYes:141   subYes: 40
```

With the data in place, we specify the model that we want to fit. `poLCA` can estimate complex models with covariates, but for the present analysis we only wish

to examine the effect of cluster membership alone. Thus, we model the dependent variables (all the observed columns) with respect to the model intercepts (i.e., the cluster positions). We use `with()` to save typing, and ~1 to specify a formula with intercepts only:

```
> seg.f <- with(seg.df.cut,
+               cbind(age, gender, income, kids, ownHome, subscribe)~1)
```

Next we fit `poLCA` models for $K = 3$ and $K = 4$ clusters using `poLCA(formula, data, nclass=K)`:

```
> library(poLCA)
> set.seed(02807)
> seg.LCA3 <- poLCA(seg.f, data=seg.df.cut, nclass=3)
...
> seg.LCA4 <- poLCA(seg.f, data=seg.df.cut, nclass=4)
...
```

`poLCA()` displays voluminous information by default, which we have omitted.

Which model is better? We use `str(seg.LCA3)` to discover the `bic` value within the object (as shown in the printed output from `poLCA()`). Comparing the two models:

```
> seg.LCA4$bic
[1] 2330.043
> seg.LCA3$bic
[1] 2298.767
```

The 3-cluster model shows a lower BIC by 32 and thus a substantially stronger fit to the data (see Table 11.1). As we've seen, that is not entirely conclusive as to business utility, so we also examine some other indicators such as the quick summary function and cluster plots:

```
> seg.summ(seg.df, seg.LCA3$predclass)
  Group.1      age   gender   income      kids  ownHome subscribe
1       1 28.22385 1.685714 30075.32 1.1285714 1.285714  1.271429
2       2 54.44407 1.576923 60082.47 0.3846154 1.769231  1.105769
3       3 37.47652 1.277778 54977.08 2.0793651 1.325397  1.079365
> table(seg.LCA3$predclass)
  1   2   3
 70 104 126

> clusplot(seg.df, seg.LCA3$predclass, color=TRUE, shade=TRUE,
+          labels=4, lines=0, main="LCA plot (K=3)")

> seg.summ(seg.df, seg.LCA4$predclass)
  Group.1      age   gender   income      kids  ownHome subscribe
1       1 36.62554 1.349593 52080.13 2.1951220 1.349593  1.113821
2       2 53.64073 1.535714 60534.17 0.5178571 1.785714  1.098214
3       3 30.22575 1.050000 41361.81 0.0000000 1.350000  1.000000
```

```
4       4 27.61506 1.866667 28178.70 1.1777778 1.066667  1.333333
> table(seg.LCA4$predclass)
  1   2   3   4
123 112  20  45
> clusplot(seg.df, seg.LCA4$predclass, color=TRUE, shade=TRUE,
+          labels=4, lines=0, main="LCA plot (K=4)")
```

The resulting plots from `clusplot()` are shown in Fig. 11.8.

We interpret the LCA results by looking first at the cluster plots (Fig. 11.8). At a high level, it appears that "Group 2" is similar in both solutions. The primary difference is that "Group 3" buried inside the overlapping ellipses in the 4-cluster solution could be viewed as being largely carved out of two larger groups (Groups "2" and "3" as labeled in the 3-cluster solution). This is an approximate interpretation of the data visualization, not a perfect correspondence.

Does the additional group in the 4-cluster solution add anything to our interpretation? Turning to the quick summary from `seg.summ()` in the code block, we see good differentiation of groups in both models. One argument in favor of the 4-cluster solution is that Group 3 has no subscribers (as shown by the mean in the `seg.summ()` results) and is relatively well identified (mostly younger women with no kids); that might make it an appealing group either for targeting or exclusion, depending on one's strategy.

As a final note on model-based clustering (and many other clustering methods such as `kmeans()`), the solutions are partially dependent on the random number seed. It can be useful to run the models with different random seeds and compare the results. This brings us to our next topic: comparing cluster solutions.

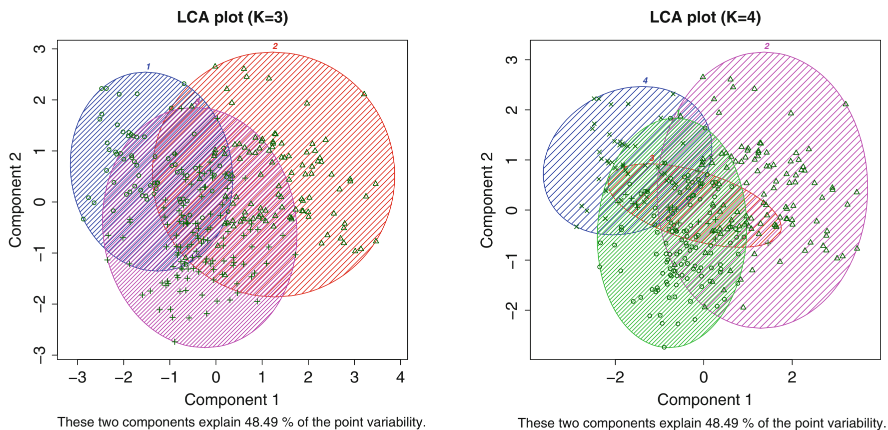

Fig. 11.8. 3-cluster and 4-cluster latent class solutions for `seg.df` found by `poLCA()`.

11.3.8 Comparing Cluster Solutions

One question we've avoided until now is this: given that we know the real group membership in `seg.df`, how does it compare to the clustering methods' results? The question is not as simple as counting agreement for two reasons. First, it is not obvious how to match one cluster solution to another because the order of group labels is arbitrary. "Group 1" in one solution might well be called "Group 2" or "Group C" in another solution.

Second, if we solve the matching problem we still need to adjust for chance agreement. Is an agreement rate of 90 % good? It depends on the base rate. If you are attempting to predict the gender of each person in a random sample of Japanese citizens, then 90 % accuracy is much better than chance (which would be roughly 51 %, the proportion of women). On the other hand, if you are attempting to predict whether each respondent speaks Japanese, then 90 % accuracy is terrible (just assigning everyone to "Yes" would achieve nearly perfect prediction, because the true rate is over 99 %).

The `mclust` package provides tools to solve both issues. `mapClass()` solves the matching problem. It examines all permutations of how two sets of class assignments might be related and selects a mapping that maximizes agreement between the two assignment schemes. `adjustedRandIndex()` likewise matches two assignment schemes and then computes the degree of agreement over and above what might be attributed to "chance" by simply assigning all observations to the largest group [81, 131]. Its magnitude may be interpreted similarly to a standard r correlation coefficient.

We use `table()` to look at the cross-tabs between the LCA 3-cluster and 4-cluster solutions found above:

```
> table(seg.LCA3$predclass, seg.LCA4$predclass)

      1   2   3   4
  1  13   0  12  45
  2   0 104   0   0
  3 110   8   8   0
```

It would appear that observations assigned to "Group 1" in the 3-cluster solution are split between Groups 1, 3, and 4 in the 4-cluster solution, while "Group 3" maps closely to "Group 1" (in the 4 class solution) and "Group 2" is predominantly the same in both. However, matching groups manually is sometimes unclear and generally error-prone. Instead, we use `mapClass(a, b)` and `adjustedRandIndex(a, b)` to compare agreement between the two solutions:

```
> library(mclust)
> mapClass(seg.LCA3$predclass, seg.LCA4$predclass)
$aTOb
```

```
$aTOb$`1`
[1] 4
$aTOb$`2`
[1] 2
$aTOb$`3`
[1] 1
... # [similarly for mapping b to a, omitted]
> adjustedRandIndex(seg.LCA3$predclass, seg.LCA4$predclass)
[1] 0.7288822
```

This tells us that “1” in the LCA3 model (`a`) maps best to “4” in the LCA4 model (`b`), and so forth. The adjusted Rand index of 0.729 indicates that the match between the two assignment lists is much better than chance. From a business perspective, it also tells us that the 3-cluster and 4-cluster differ modestly from one another, which provides another perspective on choosing between them.

By comparison, R makes it easy to see what happens if we were to test a random assignment scheme:

```
> set.seed(11021)
> random.data <- sample(4, length(seg.LCA4$predclass),
    replace=TRUE)
> adjustedRandIndex(random.data, seg.LCA4$predclass)
[1] 0.002292031
```

In this case, the adjusted Rand index is near zero, because the match between the clusters is no better than random chance.

Finally we compare the LCA 4-cluster solution to the true segments in `seg.raw`:

```
> table(seg.raw$Segment, seg.LCA4$predclass)
              1  2  3  4
  Moving up   50  4  8  8
  Suburb mix  62 29  2  7
  Travelers    0 79  1  0
  Urban hip   11  0  9 30

> adjustedRandIndex(seg.raw$Segment, seg.LCA4$predclass)
[1] 0.3513031
```

With a Rand index of 0.35, the LCA solution matches the true segment assignments moderately better than chance alone. In many cases, of course, one would not have identified clusters for comparison; but when they are available from other projects or previous efforts, it is helpful to examine correspondence in this way.

11.3.9 Recap of Clustering

We've covered four statistical methods to identify potential groups of observations in a data set. In the next section we examine the problem of how to predict (classify) observations into groups after those groups have been defined. Before we move to that problem, there are two points that are crucial for success in segmentation projects:

- Different methods are likely to yield different solutions, and in general there is no absolute "right" answer. We recommend to try multiple clustering methods with different potential numbers of clusters.
- The results of segmentation are primarily about business value, and solutions should be evaluated in terms of both model fit (e.g., using `BIC()`) *and* business utility. Although model fit is an important criterion and should not be overlooked, it is ultimately necessary that an answer can be communicated to and used by stakeholders.

11.4 Classification

Whereas clustering is the process of *discovering* group membership, classification is the *prediction* of membership. In this section we look at two examples of classification: predicting segment membership, and predicting who is likely to subscribe to a service.

Classification uses observations whose status is *known* to derive predictors, and then applies those predictors to new observations. When working with a single data set it is typically split into a *training* set that is used to develop the classification model, and a *test* set that is used to determine performance. It is crucial not to assess performance on the same observations that were used to develop the model.

A classification project typically includes the following steps at a minimum:

- A data set is collected in which group membership for each observation is known or assigned (e.g., assigned by behavioral observation, expert rating, or clustering procedures).
- The data set is split into a training set and a test set. A common pattern is to select 50–80 % of the observations for the training set (67 % seems to be particularly common), and to assign the remaining observations to the test set.
- A prediction model is built, with a goal to predict membership in the training data as well as possible.

- The resulting model is then assessed for performance using the test data. Performance is assessed to see that it exceeds chance (base rate). Additionally one might assess whether the method performs better than a reasonable alternative (and simpler or better-known) model.

Classification is an even more complex area than clustering, with hundreds of methods and hundreds of R packages, thousands of academic papers each year, and enormous interest with technology and data analytics firms. Our goal is not to cover all of that but to demonstrate the common patterns in R using two of the best-known and most useful classification methods, the naive Bayes and random forest classifiers.

11.4.1 Naive Bayes Classification: `naiveBayes()`

A simple yet powerful classification method is the *Naive Bayes* (NB) classifier. Naive Bayes uses training data to learn the probability of class membership as a function of each predictor variable considered independently (hence "naive"). When applied to new data, class membership is assigned to the category considered to be most likely according to the joint probabilities assigned by the combination of predictors. Several R packages provide NB methods; we use the `e1071` package from the Vienna University of Technology (TU Wien) [114].

The first step in training a classifier is to split the data into *training* and *test* data, which will allow one to check whether the model works on the test data (or is instead overfitted to the training data). We select 65 % of the data to use for training with the `sample()` function, and keep the unselected cases as holdout (test) data. Note that we select the training and test cases *not* from `seg.df`, which omitted the previously known segment assignments, but from the full `seg.raw` data frame. Classification requires known segment assignments in order to learn how to assign new values.

```
> set.seed(04625)
> train.prop  <- 0.65
> train.cases <- sample(nrow(seg.raw), nrow(seg.raw)*train.prop)
> seg.df.train <- seg.raw[train.cases, ]
> seg.df.test  <- seg.raw[-train.cases, ]
```

We then train a naive Bayes classifier to predict `Segment` membership from all other variables in the training data. This is a very simple command:

```
> library(e1071)
> (seg.nb <- naiveBayes(Segment ~ ., data=seg.df.train))
...
Y
 Moving up Suburb mix  Travelers  Urban hip
 0.2512821  0.3025641  0.2615385  0.1846154
```

```
Conditional probabilities:
...
            gender
Y                Female      Male
  Moving up  0.6530612 0.3469388
  Suburb mix 0.4576271 0.5423729
  Travelers  0.4705882 0.5294118
  Urban hip  0.3333333 0.6666667
...
```

Examining the summary of the model object `seg.nb`, we see how the NB model works. First, the a priori likelihood of segment membership—i.e., the estimated odds of membership before any other information is added—is 25.1 % for the Moving up segment, 30.2 % for the Suburb mix segment, and so forth. Next we see the probabilities conditional on each predictor. In the code above, we show the probabilities for `gender` conditional on segment. A member of the Moving up segment has a 65.3 % probability of being female in the training data.

The NB classifier starts with the observed probabilities of gender, age, etc., *conditional on segment* found in the training data. It then uses Bayes' Rule to compute the *probability of segment*, conditional on gender, age, etc. This can then be used to estimate segment membership in new observations such as the test data. You have likely seen a description of how Bayes' Rule works, and we will not repeat it here. For details, refer to a general text on Bayesian methods such as Kruschke [94].

Using the classifier model object `seg.nb` we can predict segment membership in the test data `seg.df.test` with `predict()`:

```
> (seg.nb.class <- predict(seg.nb, seg.df.test))
  [1] Suburb mix Travelers  Suburb mix Suburb mix Suburb mix Suburb mix
  [7] Moving up  Suburb mix Suburb mix Suburb mix Travelers  Moving up
...
```

We examine the frequencies of predicted membership using `table()` and `prop.table()`:

```
> prop.table(table(seg.nb.class))
seg.nb.class
 Moving up Suburb mix  Travelers  Urban hip
 0.2285714  0.3047619  0.3428571  0.1238095
```

A cluster plot of these segments against their principal components is created with the following code and shown in Fig. 11.9. In this case we remove the known segment assignments from the data using `[ , -7]` because we are using the NB classifications:

```
> clusplot(seg.df.test[, -7], seg.nb.class, color=TRUE, shade=TRUE,
+          labels=4, lines=0,
+          main="Naive Bayes classification, holdout data")
```

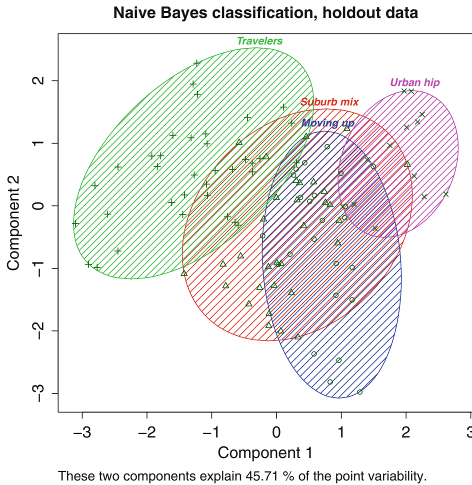

Fig. 11.9. A cluster plot for the naive Bayes classifier for segment membership predicted in holdout (test) data.

How well did the model perform? We compare the predicted membership to the known segments for the 35 % holdout (test) data. First we see the raw agreement rate, which is 80 % agreement between predicted and actual segment membership:

```
> mean(seg.df.test$Segment==seg.nb.class)
[1] 0.8
```

As we saw in Sect. 11.3.8, instead of raw agreement, one should assess performance above chance. In this case, we see that NB was able to recover the segments in the test data imperfectly but substantially better than chance:

```
> library(mclust)
> adjustedRandIndex(seg.nb.class, seg.df.test$Segment)
[1] 0.5626787
```

We compare performance for each category using `table()`. The resulting table is known in machine learning as a *confusion matrix*:

```
> table(seg.nb.class, seg.df.test$Segment)

seg.nb.class Moving up Suburb mix Travelers Urban hip
  Moving up         13         10         0         1
  Suburb mix         3         29         0         0
  Travelers          5          2        29         0
  Urban hip          0          0         0        13
```

The NB prediction (shown in the rows) was correct for a majority of observations in each segment, as shown in the diagonal. When we examine individual categories, we see that NB was correct for every proposed member of the Urban hip segment (13 correct out of 13 proposed), and for nearly 90 % of the Suburb mix proposals

(29 correct out of 32). However, it incorrectly classified 12 of the actual 41 Suburb mix respondents into other segments, and similarly failed to identify 1 of the true Urban hip segment.

This demonstrates the asymmetry of positive prediction (making a correct claim of *inclusion*) vs. negative prediction (making a correct claim of *exclusion*). There is likely to be a different business gain for identifying true positives and true negatives, versus the costs of false positives and false negatives. If you have estimates of these costs, you can use the confusion matrix to compute a custom metric for evaluating your classification results.

As we did for clustering, we check the predicted segments' summary values using our summary function. However, because we now have labeled test data, we can also compare that to the summary values using the true membership:

```
> # summary data for proposed segments in the test data
> seg.summ(seg.df.test, seg.nb.class)
     Group.1      age   gender   income      kids  ownHome subscribe  Segment
1  Moving up 34.29258 1.125000 51369.52 2.2916667 1.416667  1.250000 1.541667
2 Suburb mix 41.24653 1.562500 58095.10 2.1875000 1.562500  1.000000 1.906250
3  Travelers 55.08669 1.444444 58634.10 0.0000000 1.666667  1.166667 2.666667
4  Urban hip 23.36047 1.461538 22039.69 0.8461538 1.307692  1.153846 4.000000

> seg.summ(seg.df.test, seg.df.test$Segment)
     Group.1      age   gender   income      kids  ownHome subscribe Segment
1  Moving up 36.88989 1.190476 53582.16 1.4761905 1.333333  1.190476       1
2 Suburb mix 39.61984 1.487805 56341.99 2.2439024 1.585366  1.048780       2
3  Travelers 58.57245 1.448276 59869.24 0.0000000 1.689655  1.206897       3
4  Urban hip 23.71537 1.428571 22700.06 0.9285714 1.357143  1.142857       4
```

Overall, we see that the summary of demographics for the proposed segments (the first summary above) is very similar to the values in the true segments (the second summary). Thus, although NB assigned some observations to the wrong segments, its overall model of the segment descriptive values—at least at the mean values—is similar for the proposed and true segments. By making such a comparison using the test data, we gain confidence that although assignment is not perfect on a case-by-case basis, the overall group definitions are quite similar.

For naive Bayes models, `predict()` can estimate not only the most likely segment but also the odds of membership in each segment, using the `type="raw"` argument:

```
> predict(seg.nb, seg.df.test, type="raw")
          Moving up   Suburb mix    Travelers     Urban hip
  [1,] 4.070780e-01 5.928052e-01 4.848358e-05  6.832328e-05
  [2,] 2.715183e-04 2.422066e-03 9.973064e-01  6.143554e-32
  [3,] 2.671393e-01 7.326897e-01 1.710510e-04  2.844967e-40
  [4,] 2.237216e-01 7.746457e-01 1.632613e-03  7.568258e-37
  [5,] 2.255663e-01 7.740280e-01 4.057610e-04  9.030641e-11
...
```

This tells us that Respondent 1 is estimated to be about 59 % likely to be a member of Suburb mix, yet 40 % likely to be in Moving up. Respondent 2 is estimated nearly 100 % likely to be in Travelers. This kind of individual-level detail can suggest which individuals to target according to the difficulty of targeting and the degree of certainty. For high-cost campaigns, we might target only those most certain to be in a segment; whereas for low-cost campaigns, we might target people for second-best segment membership in addition to primary segment assignment.

We conclude that the naive Bayes model works well for the data analyzed here, with performance much better than chance, overall 80 % accuracy in segment assignment, and demographics that are similar between the proposed and actual segments. It also provides interpretable individual-level estimation of membership likelihood.

Of course there are times when naive Bayes may not perform well, and it's always a good idea to try multiple methods. For an alternative, we next examine random forest models.

11.4.2 Random Forest Classification: `randomForest()`

A random forest (RF) classifier does not attempt to fit a single model to data but instead builds an *ensemble* of models that jointly classify the data [19, 104]. RF does this by fitting a large number of classification trees. In order to find an assortment of models, each tree is optimized to fit only *some* of the observations (in our case, customers) using only *some* of the predictors. The ensemble of all trees is the *forest*.

When a new case is predicted, it is predicted by every tree and the final decision is awarded to the *consensus* value that receives the most votes. In this way, a random forest avoids dependencies on precise model specification while remaining resilient in the face of difficult data conditions, such as data that are collinear or wide (more columns than rows). Random forest models perform well across a wide variety of data sets and problems [48].

In R, a random forest may be created with code very similar to that for naive Bayes models. We use the same `seg.df.train` training data as in Sect. 11.4.1, and call `randomForest()` from the (surprise!) `randomForest` package to fit the classifier:

```
> library(randomForest)
> set.seed(98040)
> (seg.rf <- randomForest(Segment ~ ., data=seg.df.train, ntree=3000)
    )
...
        OOB estimate of  error rate: 24.1%
Confusion matrix:
           Moving up Suburb mix Travelers Urban hip class.error
Moving up         29         19         0         1  0.40816327
```

```
Suburb mix        20         35          3          1   0.40677966
Travelers          0          3         48          0   0.05882353
Urban hip          0          0          0         36   0.00000000
```

There are two things to note about the call to `randomForest()`. First, random forests are random to some extent, as the name says. They select variables and subsets of data probabilistically. Thus, we use `set.seed()` before modeling. Second, we added an argument `ntree=3000` to specify the number of trees to create in the forest. It is sometimes suggested to have 5–10 trees per observation for small data sets like the present one.

`randomForest()` returns a confusion matrix of its own based on the *training* data. How can it do that? Remember that RF fits many trees, where each tree is optimized for a portion of the data. It uses the remainder of the data—known as "out of bag" or OOB data—to assess the tree's performance more generally. In the confusion matrix, we see that the Travelers and Urban hip segments fit well, while the Moving up and Suburb mix segments had 40 % error rates in the OOB data. This is an indicator that we may see similar patterns in our holdout data.

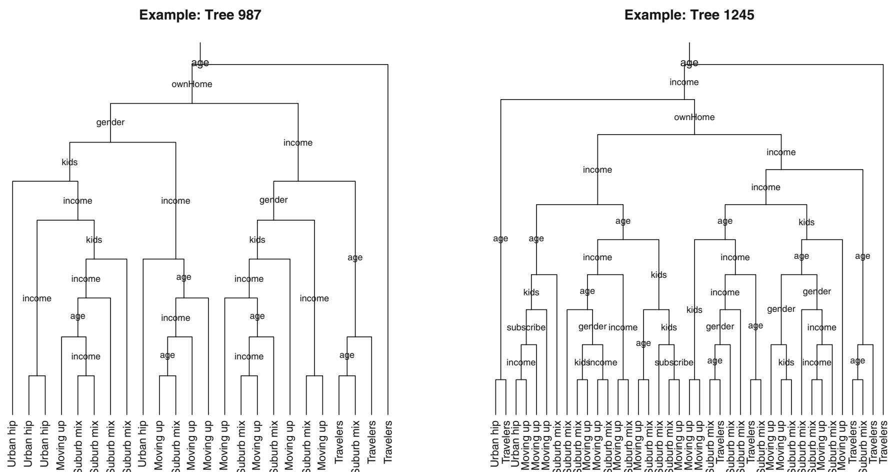

Fig. 11.10. Two examples among the 3,000 trees in the ensemble found by `randomForest()` for segment prediction in `seg.df`. The trees differ substantially in structure and variable usage. No single tree is expected to be a particularly good predictor in itself, yet the ensemble of all trees may predict well in aggregate by voting on the assignment of observations to outcome groups.

What does a random forest look like? Figure 11.10 shows two trees among those we fit above (using visualization code from Caldon [21]). The complete forest comprises 3,000 such trees that differ in structure and the predictors used. When an observation is classified, it is assigned to the group that is predicted by the greatest number of trees within the ensemble.

A cluster plot of predicted segment membership is shown in Fig. 11.11, where we omit the known segment assignments in column 7 of `seg.df.test` because we want to see the differences on the baseline variables on the basis of segments identified in the RF model:

```
> library(cluster)
> clusplot(seg.df.test[, -7], seg.rf.class, color=TRUE, shade=TRUE,
+          labels=4, lines=0, main="Random Forest classification, holdout data"
+          )
```

The RF clusters in Fig. 11.11 are quite similar in shape to those found by naive Bayes in Fig. 11.9, with respect to the principal components axes.

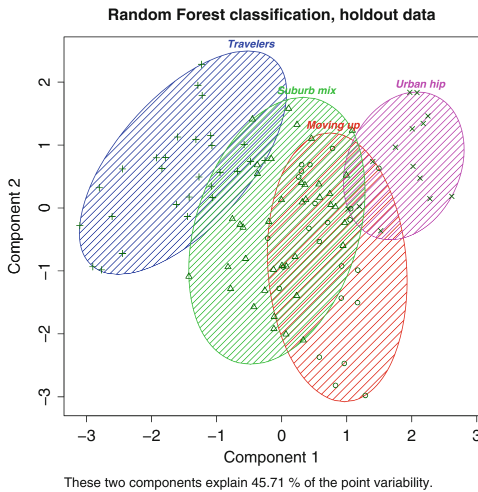

Fig. 11.11. A cluster plot for the random forest solution for segment membership predicted in holdout (test) data.

It is possible to inspect the distribution of predictions for individual cases. Add the `predict.all=TRUE` argument to the `predict()` call to get the estimate of every tree for every case in the test data. We then `apply()` the `table()` function to summarize a few of these:

```
> seg.rf.class.all <- predict(seg.rf, seg.df.test, predict.all=TRUE)
> apply(seg.rf.class.all$individual[1:5, ], 1, table) / 3000
                    2           3         4         6     7
Moving up   0.42066667 0.076333333 0.1886667 0.1223333 0.217
Suburb mix  0.47266667 0.485000000 0.6930000 0.8526667 0.340
Travelers   0.02966667 0.436333333 0.1173333 0.0240000 0.050
Urban hip   0.07700000 0.002333333 0.0010000 0.0010000 0.393
```

We divide the results of `table()` by 3,000 to get the percentage of votes across all trees. Cases 2, 3, 4, and 6 are each assigned to the Suburb mix segment as the most likely class (by tiny margins for cases 2 and 3), although only cases 4 and

6 are assigned with an overall majority of the votes. Case 7 would be assigned to the Urban hip segment as the most likely, although with only an estimated 39 % likelihood of that being its true class.

The proposed and actual segments are quite similar in the mean values of the variables in our summary function:

```
> seg.summ(seg.df.test, seg.rf.class)  # proposed segments
     Group.1      age   gender   income       kids  ownHome subscribe  Segment
1  Moving up 34.60317 1.130435 52288.38 2.39130435 1.434783  1.260870 1.608696
2 Suburb mix 40.83221 1.500000 57652.19 1.65000000 1.550000  1.000000 1.850000
3  Travelers 59.26118 1.464286 59812.04 0.03571429 1.714286  1.214286 2.892857
4  Urban hip 24.37450 1.500000 21842.73 1.00000000 1.285714  1.142857 3.857143

> seg.summ(seg.df.test, seg.df.test$Segment)  # actual segments
     Group.1      age   gender   income      kids  ownHome subscribe Segment
1  Moving up 36.88989 1.190476 53582.16 1.4761905 1.333333  1.190476       1
2 Suburb mix 39.61984 1.487805 56341.99 2.2439024 1.585366  1.048780       2
3  Travelers 58.57245 1.448276 59869.24 0.0000000 1.689655  1.206897       3
4  Urban hip 23.71537 1.428571 22700.06 0.9285714 1.357143  1.142857       4
```

As suggested by the OOB assessment we saw above for the training data, a confusion matrix reveals which segments were predicted more accurately:

```
> mean(seg.df.test$Segment==seg.rf.class)
[1] 0.7428571
> table(seg.df.test$Segment, seg.rf.class)
            seg.rf.class
             Moving up Suburb mix Travelers Urban hip
  Moving up         11          9         1         0
  Suburb mix        11         28         1         1
  Travelers          0          3        26         0
  Urban hip          1          0         0        13
```

The segment comparison using `mean(.. == ..)` calculates that RF correctly assigned 72 % of cases to their segments, and the confusion matrix using `table()` shows that incorrect assignments were mostly in the Moving up and Suburb mix segments.

Finally, we note that the RF model performed substantially better than chance:

```
> library(mclust)
> adjustedRandIndex(seg.df.test$Segment, seg.rf.class)
[1] 0.4659346
```

11.4.3 Random Forest Variable Importance

Random forest models are particularly good for one common marketing problem: estimating the importance of classification variables. Because each tree

uses only a subset of variables, RF models are able to handle very *wide* data where there are more—even many, many more—predictor variables than there are observations.

An RF model assesses the importance of a variable in a simple yet powerful way: for each variable, it randomly permutes (sorts) the variable's values, computes the model accuracy in OOB data using the permuted values, and compares that to the accuracy with the real data. If the variable is important, then its performance will degrade when its observed values are randomly permuted. If, however, the model remains just as accurate as it is with real data, then the variable in question is not very important [19].

To estimate importance, run `randomForest()` with the `importance=TRUE` argument. We reset the random seed and run RF again:

```
> set.seed(98040)
> (seg.rf <- randomForest(Segment ~ ., data=seg.df.train, ntree=3000,
                          importance=TRUE))
 ...
> importance(seg.rf)
          Moving up Suburb mix   Travelers Urban hip
age       61.386693  44.653251 121.9187436 86.345025
gender    13.065763  -4.266584  -1.6609796  8.409029
income    23.712016  17.428848  15.9978527 77.258853
kids      18.476067  14.248174  53.8039237  6.308172
ownHome    5.212246 -11.539183  23.5491524 20.667305
subscribe 16.625874   9.118376   0.8989833 -3.194460
          MeanDecreaseAccuracy MeanDecreaseGini
age                 130.712724        62.399834
gender                7.382935         3.354667
income               68.809768        36.439804
kids                 52.404913        20.081438
ownHome              16.063356         4.898022
subscribe            16.023871         2.965571
```

The upper block shows the variable importance by segment. We see, for example, that `age` is important for all segments, while `gender` is not very important. The lower block shows two overall measures of variable importance, the permutation measure of impact on accuracy (MeanDecreaseAccuracy), and an assessment of the variable's ability to assist classification better than chance labeling (MeanDecreaseGini, a measure of *Gini impurity* [19]).

The `randomForest` package includes `varImpPlot()` to plot variable importance:

```
> varImpPlot(seg.rf, main="Variable importance by segment")
```

The result is Fig. 11.12. The most important variables in this data set are `age`, `income`, and `kids`.

We plot the importance for variables by *segment* with information from `importance(MODEL)`. The variable-by-segment data are in the first four columns of that object (as shown in the code output above). We transpose it to put segments on the rows and use `heatmap.2()` to plot the values with color:

```
> library(gplots)
> library(RColorBrewer)
> heatmap.2(t(importance(seg.rf)[ , 1:4]),
+           col=brewer.pal(9, "Blues"),
+           dend="none", trace="none", key=FALSE,
+           margins=c(10, 10),
+           main="Variable importance by segment"
+ )
```

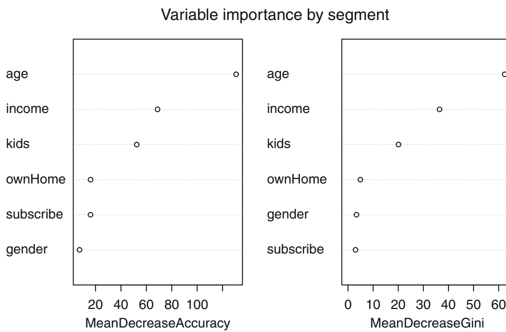

Fig. 11.12. Variable importance for segment classification from `randomForest()`.

The result is Fig. 11.13. We used the `gplots` package for `heatmap.2()`, and `RColorBrewer` to get a color palette. In the call to `heatmap.2()`, we specified `col=brewer.pal(9, "Blues")` to get nine shades of blue, `dend="none", trace="none", key=FALSE` to turn off some plot options we didn't want (dendrograms and a legend), and `margins=c(10, 10)` to adjust the margins and make the axes more readable.

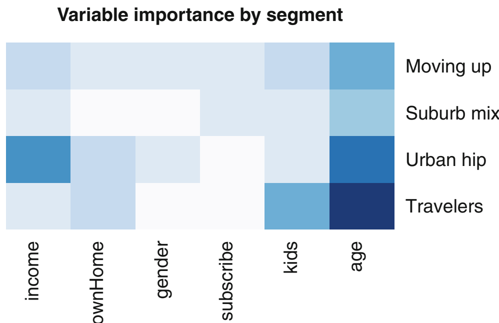

Fig. 11.13. A heatmap of variable importance by segment, produced with `randomForest()` and `heatmap.2()`. *Darker shades* signify higher importance for the variable (column) in differentiating a segment (row).

Figure 11.13 highlights the importance of age in predicting all of the segments, the importance of income to predict Urban hip, of kids to predict Travelers, and the relatively low importance of the other predictors.

11.5 Prediction: Identifying Potential Customers*

We now turn to another use for classification: to predict potential customers. An important business question—especially in high-churn categories such as mobile subscriptions—is how to reach new customers. If we have data on past prospects that includes potential predictors such as demographics, and an outcome such as purchase, we can develop a model to identify customers for whom the outcome is most likely among new prospects. In this section, we use a random forest model and attempt to predict subscription status from our data set seg.df.

As usual with classification problems, we split the data into a training sample and a test sample:

```
> set.seed(92118)
> train.prop  <- 0.65
> train.cases <- sample(nrow(seg.df), nrow(seg.df)*train.prop)
> sub.df.train <- seg.df[train.cases, ]
> sub.df.test  <- seg.df[-train.cases, ]
```

Next, we wonder how difficult it will be to identify potential subscribers. Are subscribers in the training set well differentiated from non-subscribers? We use clusplot() to check the differentiation, removing subscribe from the data with [, -6] and using it instead as the cluster identifier:

```
> clusplot(sub.df.train[, -6], sub.df.train$subscribe, color=TRUE, shade=TRUE,
+          labels=4, lines=0, main="Subscriber clusters, training data")
```

The result in Fig. 11.14 shows that the subscribers and non-subscribers are not well differentiated when plotted against principal components (which reflect almost 56 % of the variance in the data). This suggests that the problem will be difficult!

We fit an initial RF model to predict subscribe:

```
> library(randomForest)
> set.seed(11954)
> (sub.rf <- randomForest(subscribe ~ ., data=sub.df.train,
+                         ntree=3000))
...
        OOB estimate of  error rate: 14.87%
Confusion matrix:
       subNo subYes class.error
subNo    166      4  0.02352941
subYes    25      0  1.00000000
```

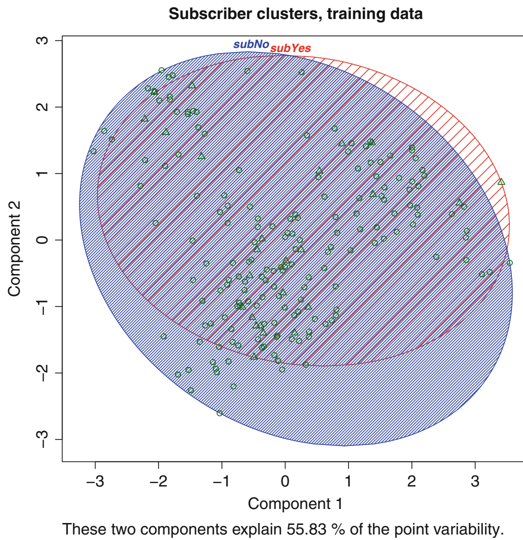

Fig. 11.14. Cluster plot for the subscribers and non-subscribers. The two groups show little differentiation on the principal components, which suggests that classifying respondents into the groups and predicting subscribers could be difficult.

The results are not encouraging. Although the error rate might initially sound good at 14.9 %, we have 100 % error in predicting subscribers (`subYes`) with all 25 misclassified in the OOB data.

Why? This demonstrates the *class imbalance* problem in machine learning. When one category dominates the data, it is very difficult to learn to predict other groups. This frequently arises with small-proportion problems, such as predicting the comparatively rare individuals who will purchase a product, who have a medical condition, who are security threats, and so forth.

A general solution is to balance the classes by sampling more from the small group. In RF models, this can be accomplished by telling the classifier to use a balanced group when it samples data to fit each tree. We use `sampsize=c(25, 25)` to draw an equal number of subscribers and non-subscribers when fitting each tree (selecting $N = 25$ each because we have that many subscribers in the training data; these are sampled with replacement so trees are not all identical):

```
> set.seed(11954)
> (sub.rf <- randomForest(subscribe ~ ., data=sub.df.train, ntree=3000,
+                          sampsize=c(25, 25)) )

Call:
 randomForest(formula = subscribe ~ ., data = sub.df.train, ntree
      = 3000,       sampsize = c(25, 25))
               Type of random forest: classification
                     Number of trees: 3000
No. of variables tried at each split: 2

        OOB estimate of  error rate: 30.77%
Confusion matrix:
       subNo subYes class.error
```

```
subNo    127     43   0.2529412
subYes    17      8   0.6800000
```

Although our overall error rate is substantially higher at 31.79 %, we are successfully predicting 32 % (i.e., $1 - 0.68$) of the subscribers in the OOB data, which is greatly improved over zero.

We use `predict()` to apply the RF model to the holdout data and examine the confusion matrix:

```
> sub.rf.sub <- predict(sub.rf, sub.df.test)
> table(sub.rf.sub, sub.df.test$subscribe)
sub.rf.sub subNo subYes
    subNo     79      9
    subYes    11      6
```

The model correctly predicts 6 of the 15 subscribers in the holdout data, at a cost of incorrectly predicting 11 others as subscribers who are not. That may be an acceptable tradeoff if we are trying to identify prospects who are worth an effort to reach. For instance, in the present case, calling *all* prospects would result in 15/105 successes (14 % success rate), while calling the suggested ones would result in 6/17 successes (35 %). The ultimate value of each strategy—to call of them or not—depends on the cost of calling vs. the value of successful conversion.

Another way to look at the result is this: those that the model said were non-subscribers were almost 90 % correct (79 correct out of 88). If the cost to target customers is high, it may be very useful to predict those *not* to target with high accuracy.

Is the model predicting better than chance? We use `adjustedRandIndex()` to find that performance is modestly better than chance, and we confirm this with `cohen.kappa()` in the `psych` package, which provides confidence intervals:

```
> adjustedRandIndex(sub.rf.sub, sub.df.test$subscribe)
[1] 0.1928668

> library(psych)
> cohen.kappa(cbind(sub.rf.sub, sub.df.test$subscribe))
Call: cohen.kappa1(x = x, w = w, n.obs = n.obs, alpha = alpha)

Cohen Kappa and Weighted Kappa correlation coefficients and confidence
     boundaries
                 lower estimate upper
unweighted kappa 0.025     0.26   0.5
weighted kappa   0.025     0.26   0.5
```

With an adjusted Rand Index = 0.19 and Cohen's kappa = 0.26 (confidence interval 0.025–0.50), the model identifies subscribers in the test data modestly better than chance.

How could we further improve prediction? We would expect to improve predictive ability if we had more data: additional observations of the subscriber group and additional predictor variables. We have described prediction using a random forest model, but there are many other approaches such as logistic regression (Sect. 9.2) and other machine learning algorithms (see Sect. 11.6).

With a difficult problem—predicting a low incidence group, in data where the groups are not well-differentiated, and with a small sample—the random forest model performs modestly yet perhaps surprisingly well. There are no magic bullets in predictive modeling, but if you use the many tools available in R, avoid pitfalls such as class imbalance, and interpret results in terms of the business action, you will have good odds to achieve positive results.

11.6 Learning More*

We covered the basics of clustering and classification in this chapter. There are many places to learn more about those methods and related statistical models. A recommended introduction to the field of statistical learning is James et al., *An Introduction to Statistical Learning* (ISL) [86]. A more advanced treatment of the topics in ISL is Hastie et al., *The Elements of Statistical Learning* [75].

For cluster analysis, a readable text is Everitt et al., *Cluster Analysis* [44]. An introduction to latent class analysis is Collins and Lanza, *Latent Class and Latent Transition Analysis* [30].

R has support for a vast number of clustering algorithms that we cannot cover here, but a few are worth mentioning. Mixture modeling is an area with active and exciting work. In addition to `mclust` that we covered above, another package of note is `flexmix` [66, 102], which fits more generalized models, and which finds mixtures using a variety of models (normal, Poisson, multinomial, Markov, and others). For very large data sets, the `clara` algorithm in the standard `cluster` package is a good starting point.

For classification and especially prediction, in addition to ISL noted above, an applied, practitioner-friendly text is Kuhn and Johnson's *Applied Predictive Modeling*. If you do classification in R, you owe it to yourself and your stakeholders to examine the `caret` package from Kuhn et al [98]. `caret` provides a uniform interface to 149 machine learning and classification algorithms (as of writing time) along with tools to assess performance and streamline other common tasks.

A resource for data when practicing clustering and classification is the `mlbench` package [103]. `mlbench` provides data sets from a variety of applications in agriculture, forensics, politics, economics, genomics, engineering, and other areas (although not marketing).

Marketing segmentation has developed approaches and nuances that differ from the typical description in statistics texts. For instance, in addition to the static, cross-sectional models considered in this chapter (where segmentation examines data at just one point in time), one might wish to consider dynamic models that take into account customer lifestyle changes over time. An overview of diverse approaches in marketing is Wedel and Kamakura, *Market Segmentation: Conceptual and Methodological Foundations* [160].

There are various ways to model changes in class membership over time. One approach is latent transition analysis (LTA), described in Collins and Lanza [30]. At the time of writing, LTA was not supported by a specific package in R. Another approach is a finite state model such as Markov chain model (cf. Ross [134]). An alternative when change over time is metric (i.e., is conceptualized as change in a *dimension* rather than change between *groups*) is to use longitudinal structural equation modeling or latent growth curve models. A starting point is the `growth()` function in the `lavaan` package that we examined in Chap. 10.

Finally, a generalized approach that is popular for clustering is cluster ensemble analysis. An ensemble method creates multiple solutions and determines group membership by likelihood or consensus among the solutions. Cluster ensembles are conceptually similar to random forest models for classification that we examined in this chapter. A package for cluster ensemble analysis is `clue` [77].

11.7 Key Points

We addressed segmentation through the lenses of clustering and classification, each of which is a large area of statistics with active research. We examined several varieties of clustering methods and compared them. Once segments or groups are identified, classification methods can help to predict group membership status for new observations.

- The most crucial question in a segmentation project is the business aspect: will the results be useful for the purpose at hand? Will they inspire new strategies for marketing to customers? It is important to try multiple methods and evaluate the utility of their results (cf. Sect. 11.1.1).
- Distance-based clustering methods attempt to group similar observations. We examined `hclust()` for hierarchical clustering (Sect. 11.3.2) and `kmeans()` for k-means grouping (Sect. 11.3.4). Distance-based measures rely on having a way to express metric distance, which is a challenge for categorical data.

- Model-based clustering methods attempt to model an underlying distribution that the data express. We examined `mclust` for model-based clustering of data assumed to be a mix of normal distributions (Sect. 11.3.5), and `poLCA` for latent-class analysis with categorical data (Sect. 11.3.7).

- A feature of some model-based methods is that they propose the *number* of clusters, unlike distance-based measures in which the analyst must choose a number. We saw how to estimate the number of groups using the `mclust` procedure (Sect. 11.3.5).

- BIC can compare models with the best statistical fit (Sect. 11.3.5). We recommend that the ultimate decision to use a model's solution be made on the grounds of both statistics (i.e., excellent fit) and the business applicability of the solution (i.e., actionable implications).

- With classification models, data should be split into training and test groups, and models validated on the test (holdout) data (Sect. 11.4).

- We examined naive Bayes models (`naiveBayes()`, Sect. 11.4.1) and random forest models (`randomForest()`, Sect. 11.4.2). These—and many other classification methods—have quite similar syntax, making it easy to try and compare models.

- A useful feature of random forest models is their ability to determine variable importance for prediction, even when there are a large number of variables (Sect. 11.4.3).

- A common problem in classification is class imbalance, where one group dominates the observations and makes it difficult to predict the other group. We saw how to correct this for random forest models with the `sampsize` argument, resulting in a more successful predictive model (Sect. 11.5).