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

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.

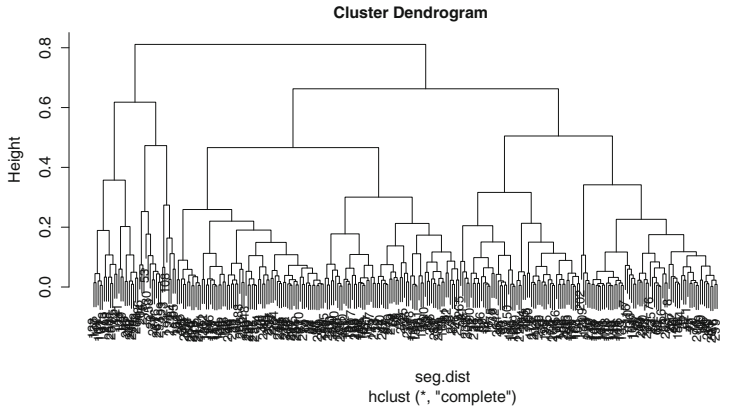

Fig. 11.1. Complete dendrogram for the segmentation data, using `hclust()`.

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.

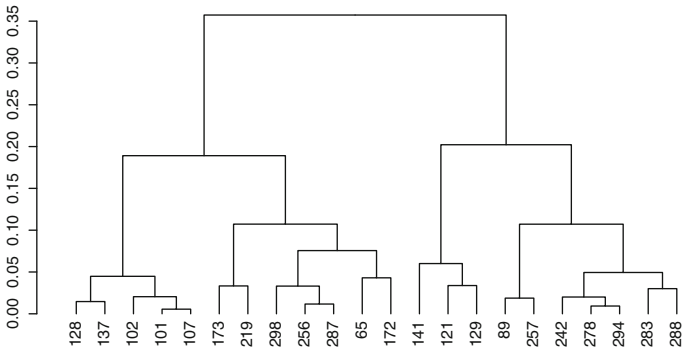

Fig. 11.2. A close up view of the left-most branch from Fig. 11.1.

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.

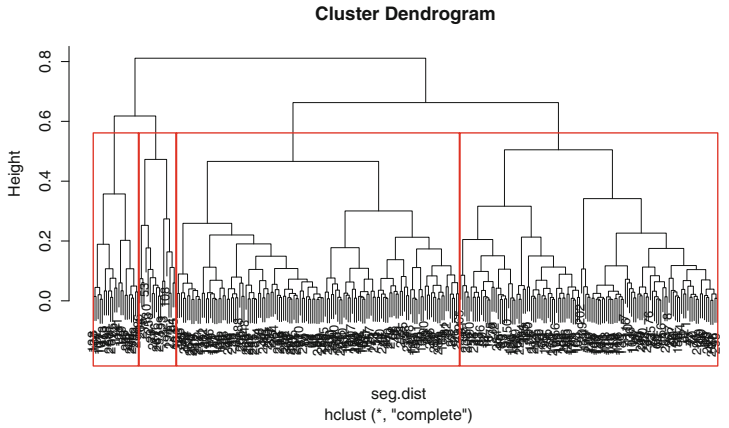

Fig. 11.3. The result of cutting Fig. 11.1 into $K = 4$ groups.

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.

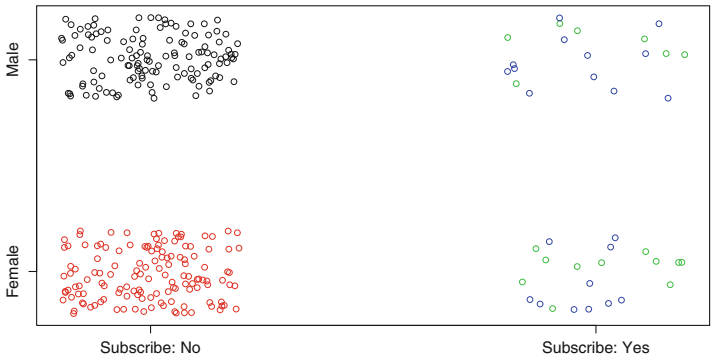

Fig. 11.4. Plotting the 4-segment solution from `hclust()` by gender and subscription status, with color representing segment membership. We see the uninteresting result that non-subscribers are simply divided into two segments purely on the basis of gender.

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")
```

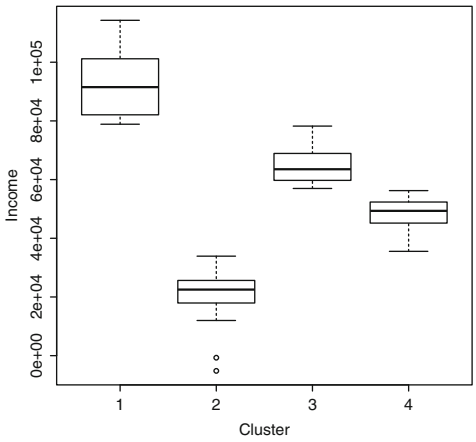

Fig. 11.5. Boxplot of income by cluster as found with `kmeans()`.

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.

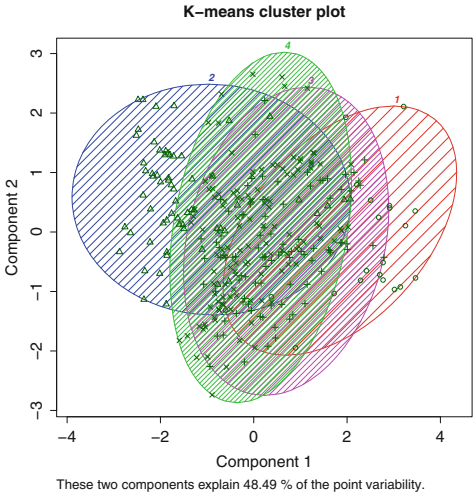

Fig. 11.6. Cluster plot created with `clusplot()` for the four group solution from `kmeans()`. This shows the observations on a multidimensional scaling plot with group membership identified by the ellipses.

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
```