CHAPTER 15

Behavioural profiles, distance metrics and cluster analysis

What you will learn from this chapter:

This chapter presents the Behavioural Profiles approach, which involves the comparison of contextual features of words or constructions in a corpus. The chapter also discusses several clustering algorithms, which are based on different distance metrics. Cluster analysis is a family of techniques that can help you discover groups of similar objects in the data. Several popular methods of cluster validation and diagnostics are discussed, which involve the computation of average silhouette widths and multiscale bootstrap resampling. The chapter also demonstrates how to interpret clusters with the help of the snake plot and effect size measures. In addition, you will learn to create and interpret scree plots, which are useful for determining the optimal number of clusters.

15.1 What are Behavioural Profiles?

The method of Behavioural Profiles (BP) is based on the ideas of Atkins (1987) and Hanks (1996), which have been more recently elaborated by Dagmar Divjak, Stefan Th. Gries and other linguists in a series of studies (e.g. Divjak 2003; Divjak & Gries 2006; Gries 2006). The approach is especially revealing for sets of near synonyms or different senses of one word. It is also particularly convenient for analysis of verb semantics because it can be used to incorporate the semantic and syntactic properties of verb arguments (e.g. the giver, the recipient and the object of transfer in the ditransitive construction, as in *I gave him the book*). For instance, Divjak & Gries (2006) compared the BP vectors of verbs of trying in Russian. The same approach can be applied to word senses, as Gries (2006) did in his study of the polysemous verb *run* (see Gries [2012] for more examples). The BP method typically requires many instances of a construction or a word coded for a number of semantic, syntactic and other categorical variables that characterize the local context, which is usually defined at the level of the sentence where the word occurs. Next, the data are represented as vectors of proportions of each value in a variable. These are BP vectors. The numerical differences between the vectors can be transformed into distances between the objects (words or word senses). Next, one can investigate semantic relationships between the objects by clustering them and exploring the common and distinctive features of clusters and individual words.

15.2 Behavioural Profiles of English analytic causatives

15.2.1 Data and theoretical background

To perform this case study, you will need the data and functions from several add-on packages. These packages should be first installed and then loaded.

```
> install.packages(c("cluster", "pvclust", "vcd"))
> library(Rling); library(cluster); library(pvclust); library(vcd)
```

Causation is a basic concept in human cognition and language (e.g. Talmy 2000). Languages have a wide range of strategies for expressing causal relationships, from monolexical verbs (e.g. *kill*, which expresses 'cause to die') to causal connectives (P *because* Q; Q, *therefore*, P). In this case study, we will examine English analytic causatives (see Chapter 12 for a description of this construction type). The English analytic causatives have received much attention in the literature (e.g. Stefanowitsch 2001; Wierzbicka 2006; Gilquin 2010; Levshina et al. 2013). The constructions that we address in this study contain the causative verbs *make*, *have*, *get* and *cause*. These verbs combine with different forms of the Effected Predicates, which include the active or passive infinitive and past or present participles. Table 15.1 shows the constructional patterns that will be considered in this study.

Table 15.1 Constructional patterns of English analytic causatives

Constructional Pattern	Example
make_V	*She made him leave.*
be_made_toV	*He was made to resign* (*by the opposition*).
cause_toV	*The high interest rates caused the currency to collapse.*
have_V	*They had a draughtsman prepare the plans.*
have_Ved	*He had his hair cut* (*by a hairdresser*).
have_Ving	*The band will have you rocking in your seat.*
get_toV	*She got the minister to sign the papers.*
get_Ved	*They tried to get their plan accepted* (*by the community*).
get_Ving	*The new government got the economy going.*

For this case study, a random sample of 450 observations was collected from the BNC. Each of the nine constructional patterns was represented by 50 observations. The observations were manually coded for six categorical variables shown in Table 14.1 (see Chapter 14). These data are available as the data frame `caus` in the `Rling` package

```
> data(caus)
> str(caus)
```

```
'data.frame': 450 obs. of 7 variables:
$ Cx: Factor w/ 9 levels "be_made_toV",..: 1 1 1 1 1 1 1 1 1 …
$ CrSem: Factor w/ 2 levels "Anim","Inanim": 1 1 1 1 1 1 1 1 1 …
$ CeSem: Factor w/ 2 levels "Anim","Inanim": 1 1 1 1 1 1 1 1 1 …
$ CdEv: Factor w/ 3 levels "Ment","Phys",..: 3 3 3 1 2 3 3 1 2…
$ Neg: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 …
$ Coref: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 …
$ Poss: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 …
```

The main question of this case study is whether the distributional properties of the constructional patterns with the same causative auxiliaries are highly similar, or such patterns exhibit idiosyncratic properties.

To construct and analyse BP vectors, one can also use a ready-made package BehavioralProfiles written by Stefan Th. Gries (available upon request), but, since the primary goal of this textbook is to teach how one can use freely available R functions, the analyses will be performed and explained stepwise. The main steps are as follows:

Step 1. Creation of numeric BP vectors from categorical data.
Step 2. Computation of the matrix of distances between the BP vectors.
Step 3. Cluster analysis and identification of the optimal number of clusters.
Step 4. Interpretation of the cluster solution in terms of the distinctive features of clusters.
Step 5. Validation of the cluster solution.

In the remaining part of the chapter, these steps will be addressed one by one.

15.2.2 Computation of numeric BP vectors from the categorical data

To create BP vectors of nine causative constructions, one can first make subsets of the data frame, which correspond to the individual constructional patterns. The patterns are the levels of the factor in the first column:

```
> levels(caus$Cx)
[1] "be_made_toV" "cause_toV"    "get_toV"    "get_Ved"    "get_
Ving""have_V""have_Ved"   "have_Ving"   "make_V"
```

To create a subset with all observations that correspond to one pattern, we will use the following code:

```
> be_made_toV <- caus[caus$Cx == "be_made_toV", -1]
> cause_toV <- caus[caus$Cx == "cause_toV", -1]
> get_toV <- caus[caus$Cx == "get_toV", -1]
> get_Ved <- caus[caus$Cx == "get_Ved", -1]
> get_Ving <- caus[caus$Cx == "get_Ving", -1]
> have_V <- caus[caus$Cx == "have_V", -1]
```

```
> have_Ved <- caus[caus$Cx == "have_Ved", -1]
> have_Ving <- caus[caus$Cx == "have_Ving", -1]
> make_V <- caus[caus$Cx == "make_V", -1]
```

Note that we also discard the first column with the constructional types, since it becomes redundant. After having repeated the procedure for all patterns, we have nine separate small data frames. Now we are ready to create BP vectors. A BP vector contains the proportions of each value of every variable. To transform the frequencies into proportions, one can use `prop.table()`, which was introduced in Chapter 4. For instance, the proportions of animate and inanimate Causers for *make_V* can be obtained as follows:

```
> prop.table(table(make_V$CrSem))
Anim  Inanim
0.56  0.44
```

For the next variable, *CeSem*, the proportions are as follows:

```
> prop.table(table(make_V$CeSem))
Anim  Inanim
0.7   0.3
```

We could combine these proportions in one vector manually. However, to save the time, you can use the special function `bp()` in the `Rling` package. It also adds the names of variables to the levels:

```
> make_V.bp <- bp(make_V)
> make_V.bp
CrSem.Anim   CrSem.Inanim   CeSem.Anim   CeSem.Inanim   CdEv.Ment
0.56         0.44           0.70         0.30           0.34
CdEv.Phys    CdEv.Soc       Neg.No       Neg.Yes        Coref.No
0.32         0.34           0.98         0.02           1.00
Coref.Yes    Poss.No        Poss.Yes
0.00         1.00           0.00
```

The resulting numeric vector consists of 13 elements. The procedure should be repeated for the remaining constructions:

```
> be_made_toV.bp <- bp(be_made_toV)
> cause_toV.bp <- bp(cause_toV)
> get_toV.bp <- bp(get_toV)
> get_Ved.bp <- bp(get_Ved)
> get_Ving.bp <- bp(get_Ving)
> have_V.bp <- bp(have_V)
> have_Ved.bp <- bp(have_Ved)
> have_Ving.bp <- bp(have_Ving)
```

Finally, we can combine these vectors as rows in a matrix.

```
> caus.bp <- rbind(be_made_toV.bp, cause_toV.bp, get_toV.bp, get_
Ved.bp, get_Ving.bp, have_V.bp, have_Ved.bp, have_Ving.bp, make_V.
bp)
```

The matrix has nine rows and thirteen columns:

```
> dim(caus.bp)
[1] 9 13
```

Finally, we will replace the row names of the matrix with the original level names from `caus$Cx`. Short and clear labels are preferable when we explore clustering solutions.

```
> rownames(caus.bp) <- levels(caus$Cx)
```

A faster way of building BP vectors

A more efficient method with slightly more advanced code would take the following three steps. First, we split the data frame `caus` into a list of nine data frames that correspond to the values of *Cx*:

```
> caus.split <- split(caus, caus$Cx)
> str(caus.split)
List of 9
$ be_made_toV:'data.frame': 50 obs. of 7 variables:
..$ Cx: Factor w/ 9 levels "be_made_toV",..: 1 1 1 1 1 1 1 1 1 1 …
..$ CrSem: Factor w/ 2 levels "Anim","Inanim": 1 1 1 1 1 1 1 1
1 1 …
..$ CeSem: Factor w/ 2 levels "Anim","Inanim": 1 1 1 1 1 1 1 1
1 1 …
..$ CdEv: Factor w/ 3 levels "Ment","Phys",..: 3 3 3 1 2 3 3 1
2 1 …
..$ Neg: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 …
..$ Coref: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 …
..$ Poss: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 …
[output omitted]
> length(caus.split)
[1] 9
```

(*Continued*)

Next, we remove the first column *Cx* in each data frame, since this column is no longer needed:

```
> caus.split <- lapply(caus.split, function(x) x = x[, -1])
```

Finally, we apply the `bp` function from `Rling` to each data frame with the help of `lapply()`:

```
> caus.split.bp <- lapply(caus.split, bp)
> caus.split.bp[1] # to check the first element
$be_made_toV
CrSem.Anim  CrSem.Inanim  CeSem.Anim  CeSem.Inanim
0.96        0.04          0.90        0.10
CdEv.Ment   CdEv.Phys     CdEv.Soc    Neg.No
0.14        0.18          0.68        0.94
Neg.Yes     Coref.No      Coref.Yes   Poss.No
0.06        1.00          0.00        1.00
Poss.Yes
0.00
```

and combine the vectors from the list as rows in a matrix by using `do.call()`. The resulting matrix is identical to the previously created `caus.bp`:

```
> caus.bp1 <- do.call(rbind, caus.split.bp)
> identical(caus.bp, caus.bp1) #to demonstrate that the resulting
matrices are identical
[1] TRUE
```

This approach may be particularly useful when the number of BP vectors is large.

15.2.3 Distance matrix

One way to explore the similarities and differences between the constructions is to eyeball the proportions in the matrix and compare the values for different constructions. Of course, this does not sound like a very reliable method. When dealing with multivariate data, it is often best to aggregate the information over many variables and try to find structure in the data. One can do so by computing a matrix of distances between the BP vectors and performing a cluster analysis.

The distances will reflect how (dis)similar the constructions are with regard to the proportions of values of the contextual variables that their BP vectors contain. The more similar their vectors, the smaller the distances. Conversely, the more dissimilar the vectors, the greater the distances. Computation of distances in R is very easy. The main function is `dist()`. Distances can be computed in several ways. The default method is the Euclidean distance, which is similar to our everyday idea of

the distance between two objects. The distance between two vectors is the square root of the summed squared differences between all pairs of numbers in the vectors. Another popular measure is the Manhattan distance. In BP studies, Canberra distance has been used, because BP vectors often contain values around zero for rare categories. The differences between these small values are treated proportionally, not absolutely, which increases their contribution to the distance metric. See more information about various distances in the box 'Distance metrics'. The reader would be well-advised to experiment with several distances in order to obtain the most interpretable solution.

Distance metrics

When we think of the distance between two points A and B, we usually imagine the shortest route, 'as the crow flies'. This distance is called **Euclidean**. Imagine that you ask a taxi driver to bring you from A to B, as shown in Figure 15.1. The driver who uses the Euclidean distance will take the direct route. Another popular measure is the **Manhattan**, or **city-block** distance. A Manhattan distance taxi driver will choose a route that takes you first down the street and then continue on the avenue orthogonal to the street, and so on, until you finally reach B. Yet another metric is called the **maximum** distance. In that case, the taxi driver will take you only by the longest side of the triangle, and you will have to walk the remaining part yourself.

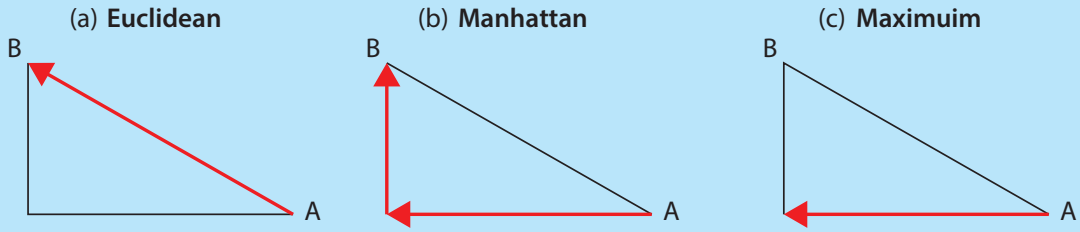

Figure 15.1. Distance metrics: (a) Euclidean, (b) Manhattan, (c) maximum

Finally, there is the Canberra distance, which is slightly more complex to interpret. It zooms in on the differences between small values, and zooms out from the differences

(*Continued*)

between large values. Therefore, it is very sensitive to small changes near zero, and less influenced by variables with large values.

In R, the Euclidean metric is set by default:

```
> dist(data) # do not run
```

To compute Manhattan distances, use the following code:

```
> dist(data, method = "manhattan") # do not run
```

To compute the maximum or Canberra distance, use `method = "maximum"` or `method = "canberra"`, respectively.

These distances can only be applied to numeric vectors. However, it is possible to compute distances between rows in a dataset with categorical variables by counting the proportion of non-overlapping values in two factors. This measure is called the **Gower** distance. It is available in the `daisy()` function in the `cluster` package. See Chapter 17 for an example. Another useful measure is the **Levenshtein** distance, which measures the number of editing operations that are needed to get one string from another string. For example, to turn *water* into *wine*, one needs three operations: replace *a* with *i*, *t* with *n*, and remove *r*. This measure is widely used in dialectometry and typology for comparison and identification of cognates. To compute it, you can use the function `adist()`, as in the following example:

```
> adist("water", "wine")
     [,1]
[1,]  3
```

Finally, one should mention the **cosine** as a measure of (dis)similarity. It is employed in vector space models of distributional semantics, which are discussed in the next chapter.

```
> caus.dist <- dist(caus.bp, method = "canberra")
> round(caus.dist, 2)
           be_made_toV  cause_toV  get_toV  get_Ved  get_Ving.
cause_toV  4.93
get_toV    3.13         5.07
get_Ved    5.93         6.24       4.41
get_Ving   5.02         4.62       5.27     6.63
[output omitted]
```

The distance matrix shows the pairwise distances between the BP vectors. For convenience, the `round()` function limits the desired number of decimal places after the point by only two. The distances between a vector and itself are not shown because they equal zero. The repeated values above the diagonal are not displayed, either (in the default settings).

One can examine the matrix and compare the distances. For example, the maximum distance is between *get_Ving* and *get_Ved* (6.63), whereas the minimum distance is between *have_V* and *get_toV* (1.3):

```
> max(caus.dist)
[1] 6.632803
> min(caus.dist)
[1] 1.300911
```

The results already indicate that the distributional and therefore semantic (dis)similarities are probably not motivated by the causative verb alone, since the constructions with the same auxiliary *get* are so dissimilar. We can explore these differences in a more systematic way by performing a cluster analysis.

15.2.4 Hierarchical cluster analysis

15.2.4.1 *Identifying the clusters*

There exist many kinds of cluster analysis. In this section, we will discuss hierarchical agglomerative clustering. It represents all objects as leaves or branches of a clustering tree. This tree is called a dendrogram. Unlike a normal tree, it 'grows' not from the root to the branches, but the other way round. In the beginning, each object (in our case, a constructional profile vector) represents its own cluster, or a 'leaf'. Next, the most similar objects (the ones for which the distance between the objects is the smallest) are merged. This procedure is repeated again and again. In the end, all leaves and branches are merged into one tree.

The function that we will need is `hclust()`. It offers many different options. We will use `method = "ward.D2"`, which usually produces compact and interpretable clusters. See Box "How do cluster trees grow?" on this and other possibilities. The dendrogram can be inspected visually in a plot.

```
> caus.hc <- hclust(caus.dist, method = "ward.D2")
> plot(caus.hc, hang = -1)
```

The resulting dendrogram can be seen in Figure 15.2. The argument `hang = -1` enables us to 'hang' all labels at the same height below the rest of the plot. The lower two elements are merged on the tree, the more similar the merged elements. The higher the merge, the more dissimilar the merged elements. The lowest merge is that of *have_V* and *get_toV*, which have the smallest distance, as you might recall.

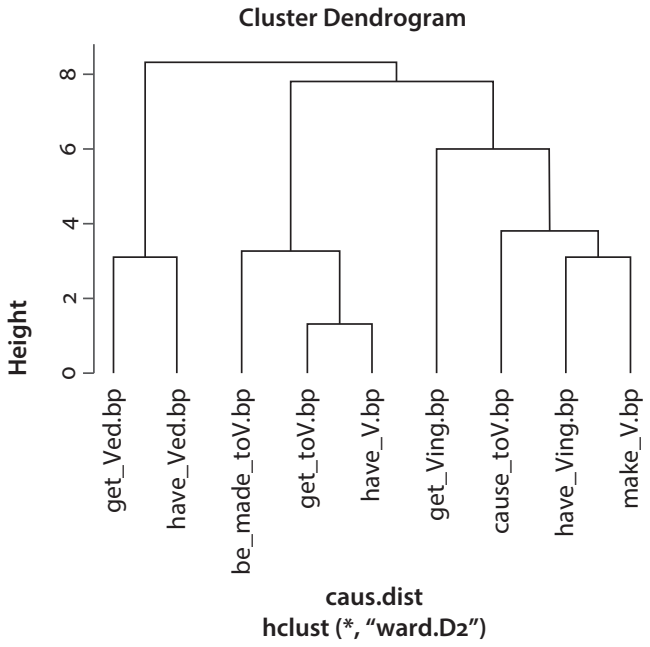

Figure 15.2. Hierarchical cluster dendrogram of the English causatives

How do cluster trees grow?

Cluster trees can grow in two ways: from roots to leaves, and from leaves to roots. The first method is called divisive clustering, whereas the second one is labelled agglomerative clustering. Agglomerative clustering, which is more popular, is implemented in `hclust()`. There are many different methods of agglomerative clustering, depending on how clusters are merged by the algorithm. The main ones are as follows:

- **Complete** (`method = "complete"`). The algorithm compares the **farthest neighbours** in all pairs of clusters and merges those clusters whose farthest neighbours are the closest. In Figure 15.3 (left) point X joins cluster A because the distance between X and the farthest member in cluster A is smaller than the distance between X and the farthest member in cluster B.
- **Single** (`method = "single"`). The algorithm compares the **nearest neighbours** in all pairs of clusters and merges those clusters whose nearest neighbours are the closest. In Figure 15.3 (centre) point X joins cluster B because the distance between X and the nearest member in cluster B is smaller than the distance between X and the nearest member in cluster A.
- **Average** (`method = "average"`). The algorithm compares the **average distances** between all pairs of clusters and merges those two clusters whose members have

the smallest average distance. In Figure 15.3 (right) Point X joins cluster B because the average distance from X to the members of cluster B is smaller than the average distance from X to the members of cluster A.

– **Ward** (`method = "ward.D2"`). The algorithm tries to minimize the increase in the variance in the distances between the members of clusters. This method usually produces compact clusters.

The merge procedure is repeated until all clusters are merged.

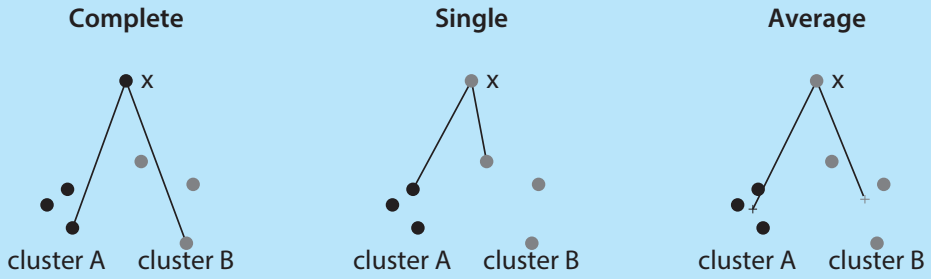

Figure 15.3. Different clustering methods. Left: complete. Centre: single. Right: average

For interpretation, it is often convenient to find the optimal number of clusters in a clustering solution. A useful statistic is the so-called average silhouette width. It shows the average well-formedness of the clusters in a given solution. Well-formedness means that the members of one cluster are close to one another and far away from the members of the other clusters. Average silhouette width ranges from 0 (no cluster structure in the data) to 1 (perfect separation of all clusters from one another). According to a rule of thumb, the average silhouette width below 0.2 should be interpreted as a lack of substantial cluster structure in the data (Kaufman & Rousseeuw 1990). We will use the function `silhouette()` from the package `cluster` to extract average silhouette widths. One can specify the number of clusters with the help of `cutree()` function. For example, this is how one can "cut" the tree in two large clusters:

```
> test.clust <- cutree(caus.hc, k = 2)
> test.clust
be_made_toV.bp   cause_toV.bp   get_toV.bp    get_Ved.bp
1                1              1             2
get_Ving.bp      have_V.bp      have_Ved.bp   have_Ving.bp
1                1              2             1
make_V.bp
1
```

The function cuts the tree into two clusters: one with *get_Ved* and *have_Ved* (value '2'), and the other with the remaining constructions (value '1').

We will begin with two clusters and then repeat the procedure until we have eight clusters. A one-cluster solution, which includes all objects, is not interesting and will not be tested. Nine clusters are theoretically possible if every causative construction forms its own cluster, but this case is not interesting, either. To see which number of clusters is optimal, we will need the function `silhouette()` from the package `cluster`.

```
> summary(silhouette(test.clust, caus.dist))$avg.width
[1] 0.2709663
```

To compute the silhouette widths for the number of clusters from 2 to 8, you can use the following code:

```
> asw <- sapply(2:8, function(x) summary(silhouette(cutree(caus.hc,
k = x), caus.dist))$avg.width)
> asw
[1] 0.2709663 0.3427721 0.3685310 0.2951836 0.2399533 0.2083129
[7] 0.1206189
```

The greatest silhouette width is approximately 0.369. It belongs to a four-cluster solution. One can explore the clustering information as we did above for the two-cluster solution, or, more conveniently, add the rectangles that correspond to the clusters directly to the plot:

```
> plot(caus.hc, hang = -1) # run if you have already closed the
graphics device
> rect.hclust(caus.hc, k = 4)
```

The result is shown in Figure 15.4. This solution also looks the most intuitive, although the three-cluster solution, which would merge *get_Ving* with the rest of the large left-hand cluster, has only a slightly smaller average silhouette width.

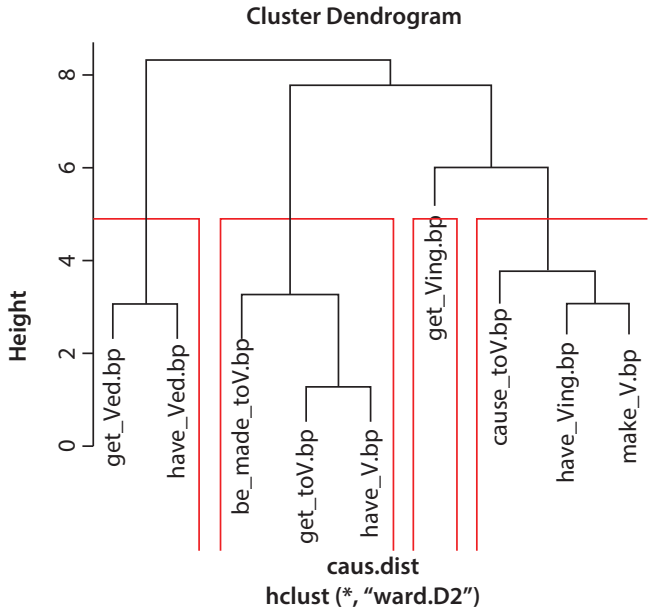

Figure 15.4. Rectangles with the optimal number of clusters, according to the average silhouette widths

15.2.4.2 *Interpretation of the cluster solution: Snake plots and effect size measures*

Let us begin with the cluster that contains *get_Ved* and *have_Ved* and compare it with the other constructions. How can we identify the features that are distinctive of this cluster? One of the options is to explore the absolute differences between the scores. First, we create two matrices, one with *get_Ved* and *have_Ved*, and the other one with all other constructions.

```
> c1 <- caus.bp[c(4,7),]
> c2 <- caus.bp[-c(4,7),]
```

Next, we compute the average proportion for every feature by using `colMeans()`.

```
> c1.bp <- colMeans(c1)
> c2.bp <- colMeans(c2)
```

Now we can compute the differences between the average values in both clusters and sort them. The function `sort()` can sort vectors in ascending (the default) and descending order:

```
> diff <- c1.bp - c2.bp
> sort(diff, decreasing = TRUE)
CrSem.Anim   CeSem.Anim    Poss.Yes     CdEv.Phys    Coref.Yes
0.32000000   0.25142857   0.16000000   0.09285714   0.06857143
CdEv.Soc     Neg.No       Neg.Yes      Coref.No     CdEv.Ment
0.03285714   0.03285714   -0.03285714  -0.06857143  -0.12571429
Poss.No      CeSem.Inanim CrSem.Inanim
-0.16000000  -0.25142857  -0.32000000
```

For binary variables the results will be symmetric, e.g. 0.32 for `CrSem.Anim` and –0.32 for `CrSem.Inanim`. To visualize the differences, one can use the so-called snake plot (Divjak & Gries 2009). The *x*-axis shows the sorted scores (in ascending order), and the *y*-axis displays the variables according to their ranks. To create a plot with text labels, one should first create an empty plot (`type = "n"`), and next add the text as shown below:

```
> plot(sort(diff)*1.2, 1:length(diff), type = "n", xlab = "cluster
2 <--> cluster 1", yaxt = "n", ylab = "")
> text(sort(diff), 1:length(diff), names(sort(diff)))
```

The result can be seen in Figure 15.5. Some clarifications are necessary. The first argument of `plot()` is the sorted vector with differences arranged in ascending order. This vector contains the values plotted on the horizontal axis. The values are magnified slightly by multiplying them by 1.2 to ensure that there is enough space for the text labels. Alternatively, one can use `xlim`. The second argument tells R to plot the variables at different heights from 1 to 13, according to their ranks. The argument `type = "n"` is used when we want to create an empty plot first and to add text labels later. Next, `xlab` gives the name of the *x* axis; `yaxt = "n"` suppresses plotting any ticks on the y axis; and `ylab = ""` suppresses adding any labels for the y axis.

The next line of code adds text labels to the plot. The first two arguments specify the coordinates of the labels. They are identical to the first arguments of the `plot()` function above, except for the multiplication by 1.2. The text labels are the names of the values in the sorted vector with differences. Finally, `cex = 0.8` specifies the relative size of text labels (by default, `cex = 1`). See Appendix 2 for other graphical options.

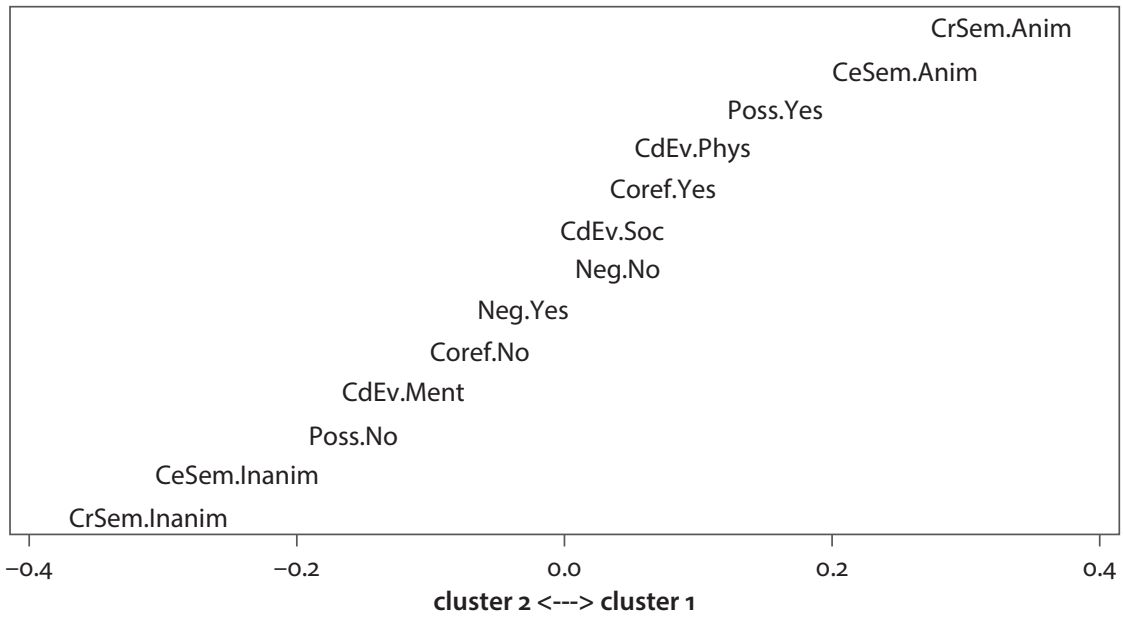

Figure 15.5. A snake plot of the differences between *get_Ved* and *have_Ved* ('cluster 1') and all other constructions ('cluster 2')

The snake plot demonstrates that *get_Ved* and *have_Ved* have a much higher proportion of animate Causers and Causees, as well as markers of the Causer's possession, and a higher proportion on non-mental effected predicates in comparison with the other causative constructions. This perfectly fits the semantics of these constructions as the ones related to inducive interpersonal causation, which is often related to service encounters, as in (1):

(1) a. *I had my car repaired* (*by a mechanic*).
b. *I got my hair done.*

A more traditional way of identification of the most distinctive variables is by using effect size measures for contingency tables (see Chapter 9). To create a contingency table with two clusters as rows and the values of a contextual feature as columns, one should first create a factor with values '1' (*get_Ved* and *have_Ved*) and '2' (all other constructions). You can use any other values if you like.

```
> cluster <- as.character(caus$Cx)
> cluster[cluster == "get_Ved"|cluster == "have_Ved"] = "1"
> cluster[cluster != "1"] = "2"
> cluster <- as.factor(cluster)
```

Now one can cross-tabulate `cluster` with any variable and compute the effect size (we will use Cramér's *V* because the size of the table may vary) with the help of `assocstats` from the package `vcd`:

```
> assocstats(table(cluster, caus$CrSem))
                    X^2 df   P(> X^2)
Likelihood Ratio    66.191   1 4.4409e-16
Pearson             42.604   1 6.7038e-11

Phi-Coefficient     :0.308
Contingency Coeff. :0.294
Cramer's V          :0.308
```

By repeating the procedure, one will find that the semantics of the Causer is indeed the variable with the strongest effect size. This finding supports the conclusions that we have made on the basis of the snake plot. One can also use multiple logistic regression and conditional inference trees and random forests with the cluster as the response variable to test if the selected variables remain significantly distinctive when the other variables are taken into account.

15.2.4.3 *Validation of a cluster solution*

We have used average silhouette widths to choose the best number of clusters, but how stable are the clusters and how well are they supported by the data? These questions can be answered with the help of multiscale bootstrap resampling in the package `pvclust`. There is an important difference: one has to use a transposed version of the data, so that each BP vector is represented by a column. It is the columns that are clustered. To transpose a matrix or a data frame, one can use the function `t()`. One has to specify the clustering method and the distance metric again. Note that computation may take some time. The results may slightly differ from one run to another.

```
> caus.pvc <- pvclust(t(caus.bp), method.hclust = "ward.D2", method.
dist = "canberra")
Bootstrap (r = 0.46)… Done.
Bootstrap (r = 0.54)… Done.
Bootstrap (r = 0.69)… Done.
Bootstrap (r = 0.77)… Done.
Bootstrap (r = 0.85)… Done.
Bootstrap (r = 1.0)… Done.
Bootstrap (r = 1.08)… Done.
Bootstrap (r = 1.15)… Done.
Bootstrap (r = 1.23)… Done.
Bootstrap (r = 1.38)… Done.
```

The essence of bootstrapping was discussed in Chapter 7. The algorithm takes a random sample with replacement from the original sample and computes the statistics of interest. The procedure is repeated many times. In the case of `pvclust()`, the default number of resamplings is 1000, but it can be changed if more precise results are needed by adding, for example, `nboot = 10000`. The computation can take quite some time if the number of iterations is very large.

The results can be seen in the plot (Figure 15.6):

```
> plot(caus.pvc, hang = -1)
```

The numbers are the probability values of obtaining the clusters in a bootstrap. You may have slightly different values every time you run `pvclust()`, unless you use the same random seed (see Chapter 14). The probability number on the left is the so-called AU (Approximately Unbiased) *p*-value, the other one is the BP (Bootstrap Probability) value. Note that the interpretation of the *p*-value is different here from our previous case studies: the closer the *p*-value to 1, the more empirical support the cluster has. The AU value is considered to be the more precise measure. We can visualize the *p*-values and highlight the clusters that have a *p*-value above 0.95 with the help of the following command:

```
> pvrect(caus.pvc, alpha = 0.95)
```

As one can see from Figure 15.6, only one cluster which contains *get_Ved* and *have_Ved* is supported by the data at the level of 0.95, although the other clusters are not particularly

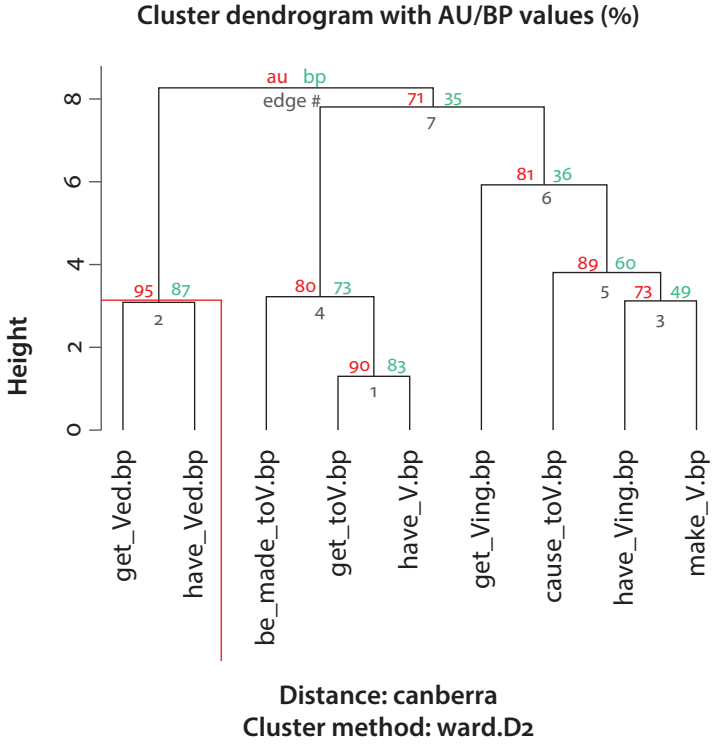

Figure 15.6. Hierarchical clustering with multiscale bootstrap *p*-values

weak, either. We have thus reasons to believe that this cluster may be observed if we use a different sample.

15.2.5 Partitioning methods

15.2.5.1 *General introduction*

Non-hierarchical cluster analysis groups objects into a pre-specified number of clusters. The process usually begins with a random classification of the objects. Next, the classification is refined by an iterative algorithm by reallocating the objects from one cluster to another until no further improvement can be made. Probably the best-known non-hierarchical method is *k*-means clustering. In this algorithm, observations are clustered around cluster centroids ('means'). These centroids are abstract 'prototypes', which are usually not identical to any particular member of their cluster. Another approach is partitioning around medoids, where the medoid is the most centrally located exemplar of a cluster, average distance from which to all other members of the cluster is minimal. Unlike centroids, medoids are always members of their clusters. The *k*-medoids algorithm is considered to be more robust that *k*-means with regard to outliers and noise. Below you can find information about both methods.

15.2.5.2 *Partitioning around centroids* (*k-means*)

The *k*-means algorithm is implemented in the function `kmeans()`. It requires two main arguments: a matrix with data (the clustered objects should be represented by the rows) and the number of clusters. It is also recommended to try several random sets for selecting initial means (one can simply choose a random number). For example, we can partition the BP vectors of the causatives into four clusters as follows:

```
> test.clust <- kmeans(caus.bp, 4, nstart = 25)
```

To see how the cluster membership, one can use the following code:

```
> test.clust$cluster
be_made_toV  cause_toV  get_toV     get_Ved  get_Ving
1            2          1           1        3
have_V       have_Ved   have_Ving   make_V
1            1          4           2
```

The classification is very different from the one obtained in the previous section. The main explanation is that *k*-means uses Euclidean distances, whereas we used Canberra distances. Therefore, high-frequency features matter more than low-frequency features. *K*-means clustering is also sensitive to outliers and often creates singleton clusters, as we see here (clusters 3 and 4).

How to choose the optimal number of clusters? In *k*-means, the main goodness-of-fit criterion is the so-called within-cluster sum of squares (WCSS). The algorithm tries to

reallocate observations from one cluster to another in such a way as to minimize the sum of squared Euclidean distances between the members of the same cluster and the cluster mean. One can also compare the sum WCSS scores produced by different solutions with different numbers of clusters. Of course, the greater the number of clusters, the smaller the total WCSS. However, it is possible to select the number of clusters such that partitioning the data into more clusters will not bring any significant decrease in WCSS. One can compare different WCSS and decide on the number of clusters with the help of a **scree plot**. To obtain WCSS for a specific cluster solution, one can do the following:

```
> test.clust$tot.withinss
[1] 0.43024
```

If we repeat the procedure for the number of clusters from 1 (no partitioning) to 8 (the total number of objects minus one), we will obtain the following values, which can be plotted as shown in Figure 15.7. To speed up the procedure, you can use the following code with `sapply()`, which returns a vector of WCSS:

```
> wcss <- sapply(1:8, function(x) kmeans(caus.bp, x, nstart =
25)$tot.withinss)
> wcss
[1] 3.02124444 1.39204000 0.66037333 0.43024000 0.26853333
[6] 0.08213333 0.03813333 0.01600000

> plot(1:8, wcss, type = "b", main = "Scree plot of WCSS for n
clusters", xlab = "n of clusters", ylab = "WCSS")
```

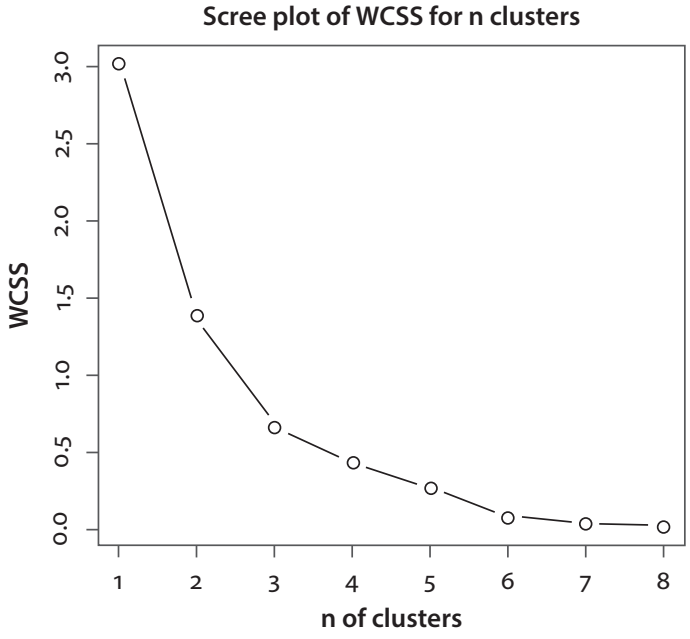

Figure 15.7. Within-cluster sums of squares for different numbers of clusters produced by the *k*-means clustering algorithm

How to interpret a scree plot? The rule of thumb is to find the point where the line 'elbows'. That means that after this point, adding new clusters does not help decrease WCSS very significantly. It seems that the best candidate is the point at $n = 3$. Of course, this is only a heuristic, and the ultimate judgment should be made on the basis of theoretical considerations. We will use this method for diagnostics in different modifications in the subsequent chapters, as well.

15.2.5.3 *Partitioning around medoids*

In this subsection, we will discuss the *k*-medoids algorithm. As has been mentioned already, it is considered more robust than *k*-means. In addition, the function `pam()` in the `cluster` package, where this algorithm is implemented, accepts both an initial matrix with numeric vectors and a distance matrix. This enables us to use any kind of distance. Let us use the Canberra distance matrix from our previous analysis and partition the data into four clusters, as above:

```
> caus.pam <- pam(caus.dist, k = 4)
```

If you want to use the original matrix, you only have choice between the Euclidean (the default) and Manhattan distances, e.g. `pam(caus.bp, k = 2, metric = "Manhattan")`.

To see the cluster membership, you can do the following:

```
> caus.pam$clustering
be_made_toV   cause_toV   get_toV    get_Ved   get_Ving
1             2           1          3         4
have_V        have_Ved    have_Ving  make_V
1             3           2          2
```

This solution coincides with the four-cluster solution in the hierarchical clustering. To obtain the medoids, one can use the following command:

```
> caus.pam$medoids
[1] "have_V"  "make_V"      "have_Ved" "get_Ving"
```

These are the prototypical members of the clusters. In our case, two clusters have only one or two members, so their medoids are not particularly informative.

Since the members of the four PAM-clusters are identical to the members of the four clusters discussed in the section about hierarchical cluster analysis, the average silhouette width values in both models should be identical. However, for didactic purposes, we will demonstrate how one can compute average silhouette widths for PAM-clusters:

```
> caus.pam$silinfo$avg.width
[1] 0.368531
```

To compare several solutions, you can use the following code:

```
> asw <- sapply(2:8, function(x) pam(caus.dist, k = x)$silinfo$avg.
width)
> asw
[1] 0.2191445 0.3427721 0.3685310 0.2951836 0.2399533 0.2083129
0.1206189
```

To summarize, the main conclusion that we can draw from this case study is that the distributional properties of the periphrastic causatives are not related directly to the auxiliary verbs only. Rather, we deal with full-fledged constructions with their individual semantic characteristics.

Hierarchical or non-hierarchical?

How can one choose, which clustering approach to use? As Manning & Schütze (1999: 500) suggest, hierarchical clustering is preferable for detailed data analysis, as it provides more information than non-hierarchical methods. Non-hierarchical partitioning approaches are convenient when the dataset is large. The *k*-means algorithm is conceptually simple, but it is restricted in its applications because it assumes a Euclidean space and does not allow one to use other types of distances. It is also less robust than *k*-medoids with regard to outliers and other noise.

15.3 Summary

In this chapter we have discussed Behavioural Profile vectors, different distance metrics and clustering methods. You have got acquainted with hierarchical and partitioning cluster algorithms, and have learnt how to decide on the optimal number of clusters with the help of average silhouette widths and scree plots. In addition, we have discussed how to validate a hierarchical clustering solution by using multiscale bootstrap.

It is important to realize that the techniques discussed in this chapter are exploratory. Different distance metrics and clustering methods may produce very different results. This is why it is recommended to try different methods, which enable one to see the data from different perspectives. Although some solutions can reflect the data structure better than others, they are still only heuristic methods that should be complemented by hypothesis-testing methods, such as regression analysis.

Reporting the results of a cluster analysis

When presenting the results of a cluster analysis, it is crucial that you are very clear about the clustered objects, the criteria for their comparison, as well as the distance metric and clustering method. It is also important that you can justify the number of clusters that you consider optimal, and, if available, report the results of cluster validation and the average silhouette width of the solution. If necessary, the dendrogram can be presented horizontally. See also `help(hclust)` for a few graphical options, such as how to avoid plotting the labels, or how to prune the tree up to *n* large clusters.

More on clustering methods

There exist many more clustering algorithms. It is worthwhile to explore the clustering functions in the `cluster` package. You can find such functions as `diana()`, which does divisive hierarchical clustering, `funny()`, which performs fuzzy clustering, and more. For creation of phylogenetic trees of language families, the neighbour-joining algorithm is particularly popular. See `nj()` in the package `ape`. You can represent different kinds of phylogenetic trees (unrooted, radial, cladograms, etc.). See `help(plot.phylo)`, the same package. An extensive discussion of clustering methods for historical linguistics and cladistics is offered in Johnson (2008: Ch. 6). For better general understanding of different clustering approaches, see Everitt et al. (2011).