```
> trainSet = ALL3gf[, trainInd]
> testSet = ALL3gf[, testInd]
```

Exercise 9.18
Now use the KNN procedure to make class predictions. Can you estimate the class-conditional error rates? Can you control the procedure so that the class-conditional error rates are treated equally?

Exercise 9.19
Repeat the above classification using random forests.

10

Unsupervised Machine Learning

R. Gentleman and V. J. Carey

Abstract

In this chapter we explore the use of unsupervised machine learning, or clustering. We cover distances, dimension reduction techniques, and a variety of unsupervised machine learning methods including hierarchical clustering, k-means clustering, and specialized methods, such as those in the **hopach** package.

10.1 Preliminaries

Cluster analysis is also known as unsupervised machine learning, and has a long and extensive history. There are many good references that cover some of the topics discussed here in more detail, such as Gordon (1999), Kaufman and Rousseeuw (1990), Ripley (1996), Venables and Ripley (2002), and Pollard and van der Laan (2005). Unsupervised machine learning is also sometimes referred to as class discovery. One of the major differences between unsupervised machine learning and supervised machine learning is that there is no training set for the former and hence, no obvious role for cross-validation. A second important difference is that although most clustering algorithms are phrased in terms of an optimality criterion there is typically no guarantee that the globally optimal solution has been obtained. The reason for this is that typically one must consider all partitions of the data, and for even moderate sample sizes this is not possible, so some heuristic approach is taken. Thus we recommend that where possible you should use different starting parameters.

The prerequisites to performing unsupervised machine learning are the selection of samples, or items to cluster, the selection of features to be used in the clustering, the choice of similarity metric for the comparison of samples, and the choice of an algorithm to use. In this chapter we consider

F. Hahne et al., *Bioconductor Case Studies*, DOI: 10.1007/978-0-387-77240-0_10,
© Springer Science+Business Media, LLC 2008

the problem of clustering samples, but most of the methods would apply equally well to the problem of clustering genes.

There are two basic clustering strategies: `hierarchical clustering` and 2) `partitioning`, as well as some hybrid methods. Hierarchical clustering can be further divided into two flavors, *agglomerative* and *divisive*. In agglomerative clustering, each object starts as its own single-element cluster and at each stage the two closest clusters are combined into a new, bigger cluster. This procedure is iterated until all objects are in one cluster. The result of this process is a tree, which is often plotted as a dendrogram (see Figure 10.3). To obtain a clustering with a desired number of clusters, one simply cuts the dendrogram at the desired height. On the other hand, divisive hierarchical clustering begins with all objects in a single cluster. At each step of the iteration, the most heterogeneous cluster is divided into two, and this process is repeated until all objects are in their own cluster. The result is again a tree.

Partitioning algorithms typically require the number of clusters to be specified in advance. Then, samples are assigned to clusters, in some fashion, and a series of iterations, where (typically) single sample exchanges or moves are proposed and the resulting change in some clustering criteria computed; changes that improve the criteria are accepted. The process is repeated until either nothing changes or some number of iterations is made.

10.1.1 Data

First we load the necessary packages and load the dataset we use for the examples and exercises.

We use the ALL dataset, from the ALL package for this chapter. It is described more completely in Chapter 1. Our goal is to demonstrate how one can use various clustering methods, so we ignore the sample data. We reduce the data to a manageable size by selecting those samples that correspond to B-cell ALL and where the molecular biology phenotype is either BCR/ABL or NEG. The code for selecting the appropriate subset is given below; more details on the steps involved are given in Chapter 1.

```
> library("ALL")
> data(ALL)
> bcell = grep("^B", as.character(ALL$BT))
> moltyp = which(as.character(ALL$mol.biol)
      %in% c("NEG", "BCR/ABL"))
> ALL_bcrneg = ALL[, intersect(bcell, moltyp)]
> ALL_bcrneg$mol.biol = factor(ALL_bcrneg$mol.biol)
> ALLfilt_bcrneg = nsFilter(ALL_bcrneg, var.cutoff=0.75)$eset
```

The filtering has selected 2638 genes that we consider of interest for further investigation. This will still be too many genes for most applications

Table 10.1. GO molecular function categories that correspond to transcription factors.

GO Identifier	Description
GO:0003700	Transcription factor activity
GO:0003702	RNA polymerase II transcription factor activity
GO:0003709	RNA polymerase III transcription factor activity
GO:0016563	Transcriptional activator activity
GO:0016564	Transcriptional repressor activity

and often one will want to use other criteria to further reduce the genes under study. Here, we focus on transcription factors; these are important regulators of gene expression. As it turns out, finding the set of known transcription factors for any species is not such an easy problem. We use the GO identifiers in Table 10.1 that were used by Kummerfeld and Teichmann (2006) as their reference set of known transcription factors.

For each annotation of a gene to a GO category, there is an evidence code that indicates the basis for mapping that gene to the category. We drop all those that correspond to `IEA`, which stands for inferred from electronic annotation. We show the code for this task below.

```
> GOTFfun = function(GOID) {
      x = hgu95av2GO2ALLPROBES[[GOID]]
      unique(x[ names(x) != "IEA"])
  }
> GOIDs = c("GO:0003700", "GO:0003702", "GO:0003709",
      "GO:0016563", "GO:0016564")
> TFs = unique(unlist(lapply(GOIDs, GOTFfun)))
> inSel = match(TFs, featureNames(ALLfilt_bcrneg), nomatch=0)
> es2 = ALLfilt_bcrneg[inSel,]
```

This leaves us with 249 transcription factor coding genes for our machine learning exercises.

10.2 Distances

As we noted in the other machine learning exercise, no machine learning can take place without some notion of distance. It is not possible to cluster or classify samples without some way to say what it means for two things to be similar. For this reason, we again begin by considering distances. The `dist` function in R, the **bioDist** package, and the function `daisy` in the **cluster** package all provide different distances that you can use. It is always worth spending some time considering what it means for two objects to be similar and to then select a distance measure that reflects your belief.

Many machine learning methods have a built-in distance, often not obvious and difficult to alter, and if you want to use those methods you may need to use their metric. But it is important to realize that if you do use different measures of distance, they will have an impact on your analysis.

We begin by making use of the Manhattan metric; you might choose a different metric to compute distances between samples. Because we have no a priori belief that any one gene is more important than any other, we first center and scale the gene expression values before computing distances. Finally, we produce a heatmap based on the computed between-sample distances (Figure 10.1). There are no obvious groupings of samples based on this heatmap. We choose colors for our heatmap from a palette in the **RColorBrewer** package. Because the palette goes from red to blue, but we want high values to be red, we must reverse the palette, as is done in the code below.

```
> iqrs = esApply(es2, 1, IQR)
> gvals = scale(t(exprs(es2)), rowMedians(es2),
      iqrs[featureNames(es2)])
> manDist = dist(gvals, method="manhattan")
> hmcol = colorRampPalette(brewer.pal(10, "RdBu"))(256)
```

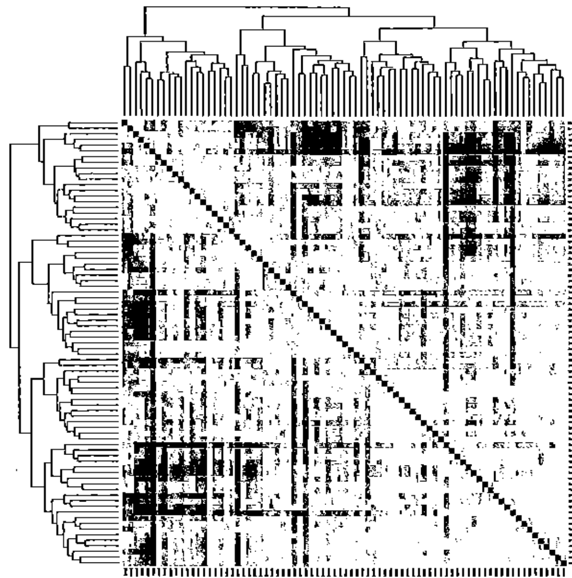

Figure 10.1. A heatmap of the distances between samples. Blue corresponds to small distances, red to large.


```
> hmcol = rev(hmcol)
> heatmap(as.matrix(manDist), sym=TRUE, col=hmcol,
      distfun=function(x) as.dist(x))
```

Another popular visualization method for distance matrices is to use multidimensional scaling to reduce the dimensionality to two or three, and to then plot the resulting data. There are several different methods available, from the classical `cmdscale` function to Sammon mapping via the `sammon` function in the **MASS** package. Again we see little evidence of any grouping of the samples (Figure 10.2).

```
> cols = ifelse(es2$mol.biol == "BCR/ABL", "black",
      "goldenrod")
> sam1 = sammon(manDist, trace=FALSE)
> plot(sam1$points, col=cols, xlab="Dimension 1",
      ylab="Dimension 2")
```

Exercise 10.1

a. *In the code above we obtained a two-dimensional reduction. Obtain a three-dimensional reduction, and if you have it installed, view this using the rgl package, so that you can rotate the points in three dimensions.*

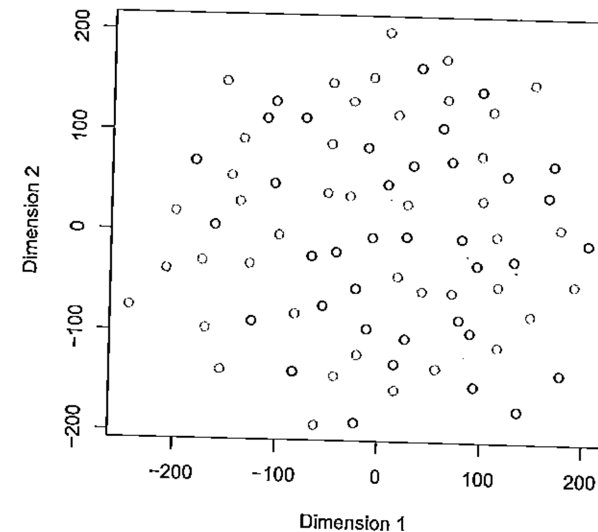

Figure 10.2. The two-dimensional projection of the between-sample distances obtained using Sammon mapping. Samples with the BCR/ABL phenotype are indicated with dark circles, and those with the NEG phenotype by light circles.

b. *Repeat the example using classical multidimensional scaling.*

c. *Repeat the exercise, but first restrict the genes you use to the 50 genes that best separate the two groups via their t-statistics.*

10.3 How many clusters?

Now that we have decided on a distance to use, we can ask one of the more fundamental questions that arises in any application of unsupervised machine learning: How many clusters are there? And unfortunately, even after a lot of research there is no definitive answer. The references given above provide some methods, and there are newer results as well, but none has been found to be broadly useful. We recommend visualizing your data, as much as possible, for instance, by using dimension reduction methods such as multidimensional scaling, as well as special-purpose tools such as the silhouette plot of Kaufman and Rousseeuw (1990); (see Section 10.8).

Another popular method is to examine the dendrogram that is produced by some hierarchical clustering algorithm to see if it suggests a particular number of clusters. Unfortunately, this procedure is not really a good idea. If you compare the four dendrograms in Figure 10.3 they do not convey a coherent message. The third from the top suggests that there might be three clusters, but the other three are much less suggestive.

The **hopach** package contains two functions that can be used to estimate the number of clusters. They are based on approaches that are related to the silhouette plot that is described in Section 10.8. In the code chunk below we demonstrate their use, on both the samples and the genes from our example dataset.

```
> mD = as.matrix(manDist)
> silEst = silcheck(mD, diss=TRUE)
> silEst
[1] 2.000 0.163
> mssEst = msscheck(mD)
> mssEst
[1] 4.0000 0.0777
> d2 = as.matrix(dist(t(gvals), method="man"))
> silEstG = silcheck(d2, diss=TRUE)
> silEstG
[1] 3.000 0.107
> mssEstG = msscheck(d2)
> mssEstG
[1] 6.0000 0.0489
```


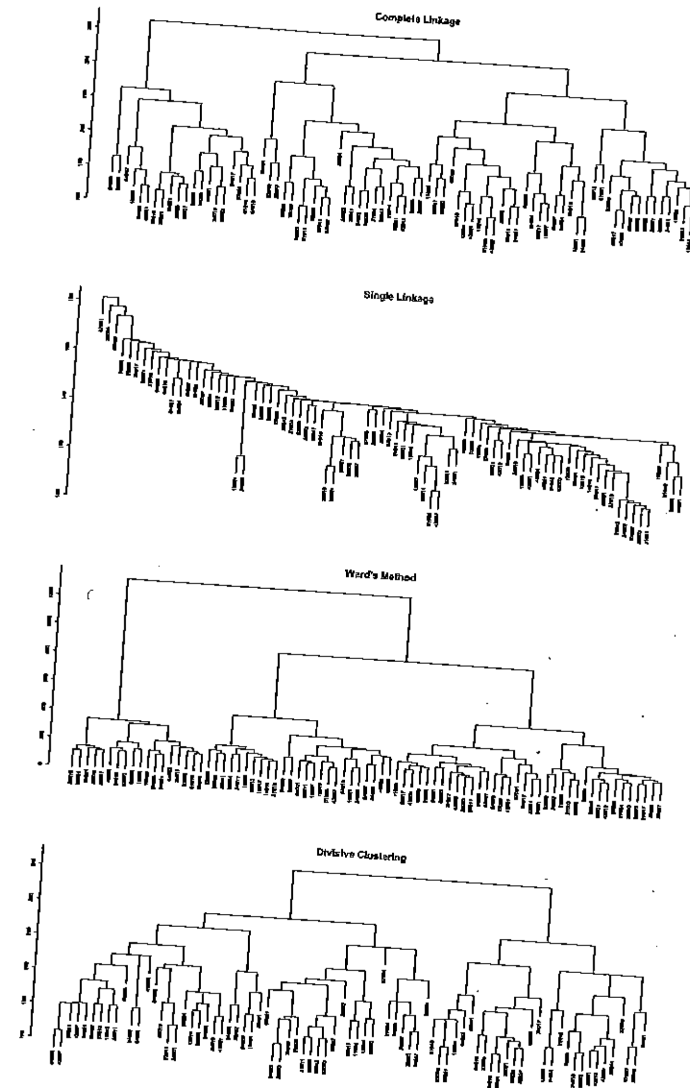

Figure 10.3. Four different dendrograms, clustering samples. The first three (top to bottom) were computed using agglomerative hierarchical clustering with different linkage methods, the bottom one used divisive hierarchical clustering.

The `silcheck` function returns a vector of length two; the first element is the recommended number of clusters, whereas the second element is the average silhouette for that number of clusters. The return value of `msscheck`, is also of length two, the first value again being the recommended number of clusters, and for this function the second value is the median split silhouette (MSS).

We can see that `silcheck` recommends two, whereas `mssEst` recommends four. If instead, we consider clustering genes then the two methods recommend three and six clusters, respectively. These estimates could be used when we consider the partitioning methods described in Section 10.5.

Exercise 10.2
Repeat the exercise of assessing how many clusters there are, using another distance measure.

10.4 Hierarchical clustering

We now briefly discuss different hierarchical clustering methods. There are two basic strategies that can be used in hierarchical clustering. Divisive clustering begins with all objects in one cluster and at each step splits one cluster to increase the number of clusters by one. Agglomerative clustering starts with all objects in their own cluster and at each stage combines two clusters, so that there is one less cluster. Agglomerative clustering is one of the very few clustering methods that have a deterministic algorithm, and this may explain its popularity. There are many variants on agglomerative clustering, and the manual page for the function `hclust` provides some details. Divisive hierarchical clustering can be performed by using the `diana` function from the **cluster** package.

We first compute the clusterings and then show how to plot them and manipulate the outputs. The `hclust` function returns an instance of the *hclust* class, and `diana` returns an object of class *diana*. These are S3 classes, and hence the objects are lists, with certain named components.

```
> hc1 = hclust(manDist)
> hc2 = hclust(manDist, method="single")
> hc3 = hclust(manDist, method="ward")
> hc4 = diana(manDist)
```

We can plot the resulting dendrograms, and the results are shown in Figure 10.3.


```
> par(mfrow=c(4,1))
> plot(hc1, ann=FALSE)
> title(main="Complete Linkage", cex.main=2)
> plot(hc2, ann=FALSE)
> title(main="Single Linkage", cex.main=2)
> plot(hc3, ann=FALSE)
> title(main="Ward's Method", cex.main=2)
> plot(hc4, ann=FALSE, which.plots=2)
> title(main="Divisive Clustering", cex.main=2)
> par(mfrow=c(1,1))
```

The order in which the leaves are plotted (from left to right) is stored in the slot `order`. For example, `hc1$order` is the leaf order in the dendrogram `hc1` and `hc1$labels[hc1$order]` yields the sample labels in the order in which they appear.

Dendrograms can be manipulated using the `cutree` function. You can specify the number of clusters via the parameter `k` and the function will cut the dendrogram at the appropriate height and return the elements of the clusters. Alternatively, you can directly specify the height at which to cut via the parameter `h`.

Exercise 10.3
Cut each of the different clusterings into three clusters. Compare the outputs using, for example, the `table` *function.*

Although the dendrogram has been widely used to represent distances between objects, it should not be considered a visualization method. Dendrograms do not necessarily expose structure that exists in the data. In many cases they impose a preconceived structure (a tree) on the data, and when that is the case it is dangerous to interpret the observed structure.

Hierarchical clustering creates a new set of between-object distances, corresponding to the path lengths between the leaves of the dendrogram. It is interesting to ask whether these new distances reflect the distances that were used as inputs to the hierarchical clustering algorithm. The cophenetic correlation (e.g., Sneath and Sokal (1973, p. 278)), implemented in the function `cophenetic`, can be used to measure the association between these two distance measures.

In the code below we show how to compute the cophenetic correlation for complete linkage hierarchical clustering.

```
> cph1 = cophenetic(hc1)
> cor1 = cor(manDist, cph1)
> cor1
[1] 0.524
> plot(manDist, cph1, pch="|", col="blue")
```

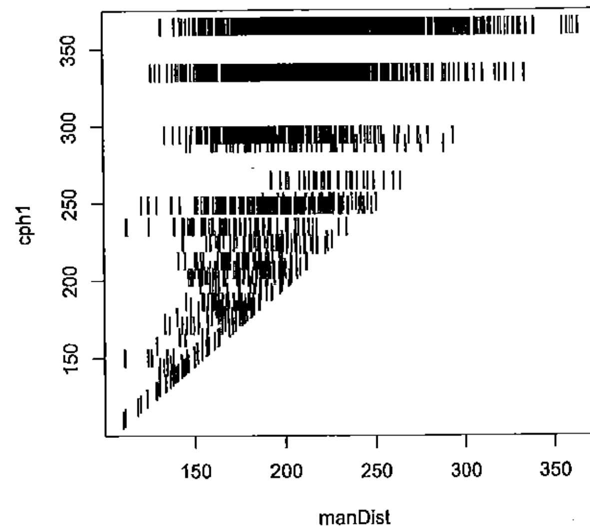

Figure 10.4. A scatterplot of actual distances (on the x-axis) versus the cophenetic distances (on the y-axis) for the hierarchical clustering hc1.

The result is shown in Figure 10.4. The bands in the y direction are due to the discrete nature of the between-sample distances based on trees. For tree-based distances all objects in two subtrees are the same distance from each other.

Exercise 10.4
Compute the cophenetic correlation for the other three dendrograms and comment on which, if any of them, seem to have a particularly good or particularly bad fit.

10.5 Partitioning methods

Let us now turn to partitioning methods. Typically, the algorithms require us to specify the number of clusters into which they should partition the data. There is no generally reliable method for choosing this number, although we may use the estimates we obtained in Section 10.3. Partitioning algorithms have a stochastic element: they depend on an essentially arbitrary choice of a starting partition, which they iteratively update to try to find a good solution.

A simple implementation of a partitioning clustering algorithm, k-means clustering, is provided by the function kmeans. The k-means method attempts to partition the samples into k groups such that the sum

of squared distances from the samples to the assigned cluster centers is minimized. The implementation allows you to supply either the location of the cluster centers, or the number of clusters using the centers parameter. It is often a good idea to try multiple choices of random starting partitions, which can be specified by the nstart parameter. The function returns the partition with the best objective function (the smallest sum of squared distances), but that does not mean that there is not a better partition that has not been tested.

In the code below, we call kmeans twice; in both cases we request two groups, but we try 5 different random starts with the first call, and 25 with the second.

```
> km2 = kmeans(gvals, centers=2, nstart=5)
> kmx = kmeans(gvals, centers=2, nstart=25)
```

Exercise 10.5
What values are returned by kmeans*? Do the two calls find the same clusters?*

Exercise 10.6
Which one of the categorical phenotypic variables for our expression set best aligns with the output of the k-means clustering algorithm?

10.5.1 PAM

Partitioning around mediods (PAM) is based on the search for k representative objects, or medoids, among the samples. Then k clusters are constructed by assigning each observation to the nearest medoid with a goal of finding k representative objects that minimize the sum of the dissimilarities of the observations to their closest representative object. This method is implemented by the pam function, from the **cluster** package. It is much more flexible than the kmeans function in that one can specify different distance metrics to use or supply a distance matrix to use, rather than a data matrix.

```
> pam2 = pam(manDist, k=2, diss=TRUE)
> pam3 = pam(manDist, k=3, diss=TRUE)
```

We can compare the two clusterings, but need to do a little checking to ensure that the orderings are the same.

```
> all(names(km2$cluster) == names(pam2$clustering))
[1] TRUE
> pam2km = table(km2$cluster, pam2$clustering)
> pam2km
     1  2
  1 62  0
  2  3 14
```

Exercise 10.7
How many items are classified in the same way by the two methods (k-means and PAM)? How many are classified differently? Can you determine which ones they are? And, then using the cluster centers, as reported by the different methods, which of the two methods is better? How do the cluster centers for the two methods compare?

Exercise 10.8
Repeat Exercise 10.6 for PAM clustering into three groups.

10.6 Self-organizing maps

self-organizing maps (SOMs)[Self-organizing maps (SOMs)] were proposed by Kohonen (1995) as a simple method for allowing data to be sorted into groups. The basic idea is to lay out the data on a grid, and to then iteratively move observations (and the centers of the groups) around on that grid, slowly decreasing the amount that centers are moved, and slowly decreasing the number of points considered in the neighborhood of a grid point. For our examples we use a four-by-four grid, so that there are at most 16 groups.

We examine two implementations, one in the **kohonen** package, and the `SOM` function in the package **class**. The second of these is described in more detail in Venables and Ripley (2002). We do note that there are others, such as that provided by the **som** package and readers might want to consider that version as well. Unfortunately the default values, calling sequences and return values for these different implementations tend to vary and so you as a user will need to use some caution in comparing them.

First we demonstrate the use of SOMs using the **kohonen** package. We fit three different models: the first uses the default values, and the next two calls change some of these. We set the seed for the random number generator to ensure that readers get the same answers we do.


```
> set.seed(123)
> s1 = som(gvals, grid=somgrid(4,4))
> names(s1)
 [1] "data"          "grid"          "codes"
 [4] "changes"       "alpha"         "radius"
 [7] "toroidal"      "unit.classif" "distances"
[10] "method"
> s2 = som(gvals, grid=somgrid(4,4), alpha=c(1,0.1),
      rlen=1000)
> s3 = som(gvals, grid=somgrid(4,4, topo="hexagonal"),
      alpha=c(1,0.1), rlen=1000)
> whGP = table(s3$unit.classif)
> whGP
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
 1  9  2  1  1  1  1  1  7 10  1 19  1  1 15  8
```

The output is an instance of the `kohonen` class. And the last call (to `table`) in the code above, tells us which sample is assigned to which of the 16 possible groups. There are two groups with more than ten observations in them, and nine with only one. The groups with only one are problematic, and although they may represent clusters, it is not clear that they do.

Exercise 10.9
Read the man page for the `kohonen` class. What are the components of this class? How do the results of using the default values compare to the other two?

Another important aspect of understanding the data would be to consider the samples in the different groups and to visualize them.

Exercise 10.10
We choose two of the larger clusters in the output of the first clustering. Create a heatmap comparing those in cluster 13 to those in cluster 14.

Next we consider the `SOM` from the **class** package. This function returns the grid that the map was laid out on, as well as a matrix of representatives; one then uses the `knn1` function to match a sample to its nearest representative. We begin by setting the seed for the random number generator to ensure that readers get the same output as we do.

```
> set.seed(777)
> s4 = SOM(gvals, grid=somgrid(4,4, topo="hexagonal"))
> SOMgp = knn1(s4$code, gvals, 1:nrow(s4$code))
> table(SOMgp)
```

```
SOMgp
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
 0  1  9  5  1  1  2  0  2 11  1 17  1  3  6 19
```

Now we can see that there are some groups that have no values in them, whereas others tend to have between 10 and 15. To further refine the clusters, down to just a few, we might next ask whether any of the cluster centroids are close to each other, suggesting that merging of the clusters might be worthwhile. We compute the distance matrix comparing cluster centers next, and from that computation we see that clusters (1, 2, 5, 6) can be collapsed, as can (3, 10), (4,15), (7,9), (8, 11, 13, 14). We make this observation based on the zero entries in the distance matrix, cD, computed below.

```
> cD = dist(s4$code)
> cD
       1     2     3     4     5     6     7     8
2  0.000
3  0.857 0.857
4  1.573 1.573 1.813
5  0.000 0.000 0.857 1.573
6  0.000 0.000 0.857 1.573 0.000
7  0.839 0.839 1.132 1.571 0.839 0.839
8  1.182 1.182 1.558 1.796 1.182 1.182 1.219
9  0.839 0.839 1.132 1.571 0.839 0.839 0.000 1.219
10 0.857 0.857 0.000 1.813 0.857 0.857 1.132 1.558
11 1.182 1.182 1.558 1.796 1.182 1.182 1.219 0.000
12 2.669 2.669 3.132 3.046 2.669 2.669 2.888 2.565
13 1.182 1.182 1.558 1.796 1.182 1.182 1.219 0.000
14 1.182 1.182 1.558 1.796 1.182 1.182 1.219 0.000
15 1.573 1.573 1.813 0.000 1.573 1.573 1.571 1.796
16 2.176 2.176 2.445 2.648 2.176 2.176 2.167 2.290
       9    10    11    12    13    14    15
2
3
4
5
6
7
8
9
10 1.132
11 1.219 1.558
12 2.888 3.132 2.565
13 1.219 1.558 0.000 2.565
14 1.219 1.558 0.000 2.565 0.000
15 1.571 1.813 1.796 3.046 1.796 1.796
16 2.167 2.445 2.290 3.788 2.290 2.290 2.648
```


So, we can then remove the redundant codes and remap the data into clusters using the knn1 function as above.

```
> newCodes = s4$code[-c(2,5,6,10, 15, 9, 11, 13, 14),]
> SOMgp2 = knn1(newCodes, gvals, 1:nrow(newCodes))
> names(SOMgp2) = row.names(gvals)
> table(SOMgp2)
SOMgp2
 1  2  3  4  5  6  7
 3 20 11  4  5 17 19
> cD2 = dist(newCodes)
> cmdSOM = cmdscale(cD2)
```

As we see there are now four reasonably large groups, and three smaller ones.

Exercise 10.11
Compare this clustering with k-means output with k set to 4. What happens if you remove the arrays that correspond to the small clusters and redo the k-means analysis?

10.7 Hopach

The hopach package (Pollard and van der Laan, 2005) uses a hybrid approach to clustering. It makes use of both a hierarchical approach as well as a partitioning method. In Section 10.3 we introduced two functions from this package for assessing how many clusters are in the data. In this section we use hopach to cluster our samples.

```
>   samp.hobj = hopach(gvals, dmat = manDist)
>   samp.hobj$clust$k
[1] 3
```

This suggests that there are three clusters. We should first see what sizes they are.

```
> samp.hobj$clust$sizes
[1] 24 28 27
```

We next consider hopach clustering for the genes. In this case, we follow the advice in Pollard and van der Laan (2005) and use the cosangle distance between genes.

```
> gene.dist = distancematrix(t(gvals), d = "cosangle")
> gene.hobj = hopach(t(gvals), dmat = gene.dist)
> gene.hobj$clust$k
[1] 40
```

So we see that hopach is suggesting that there are 40 clusters of genes and their sizes are shown in the output below.

```
> gene.hobj$clust$sizes
 [1]  1  1  1  2  1  1  3  1  1  1  1  3  3  1  1  1 11
[18] 57 53 28  1  1  2  1  4  1  1  1  4  1  1  1  1  3
[35]  1  5  1  1  1 45
```

Next one can try to identify important, possibly functional relationships for the genes in the different clusters. A fairly straightforward process would be to use the **GOstats** package to perform an analysis on these genes.

10.8 Silhouette plots

Silhouette plots can be produced using the `silhouette` function in the **cluster** package. It can be defined for virtually any clustering algorithm, and provides a nice way to visualize the output.

The silhouette for a given clustering, C, is calculated as follows. For each item j, calculate the average dissimilarity $\bar{d}_{jl}$ of item j with other genes in the cluster C_l, for all l. Thus, if there are L clusters, we would compute L values for each item. If item j is assigned to cluster l^* then let $a_j = \bar{d}_{jl^*}$, and let $b_j = \min_{l \neq l^*} \bar{d}_{jl}$. The silhouette of item j is defined by the formula: $S_j = (b_j - a_j)/\max(a_j, b_j)$. Heuristically, the silhouette measures how similar an object is to the other objects in its own cluster versus those in some other cluster. Values for S_j range from 1 to -1, with values close to 1 indicating that the item is well clustered (is similar to the other objects in its group) and values near -1 indicating it is poorly clustered, and that assignment to some other group would probably improve the overall results.

We revisit the PAM clusterings, because there are plotting methods for them. These are shown in Figure 10.5.

```
> silpam2 = silhouette(pam2)
> plot(silpam2, main="")
```


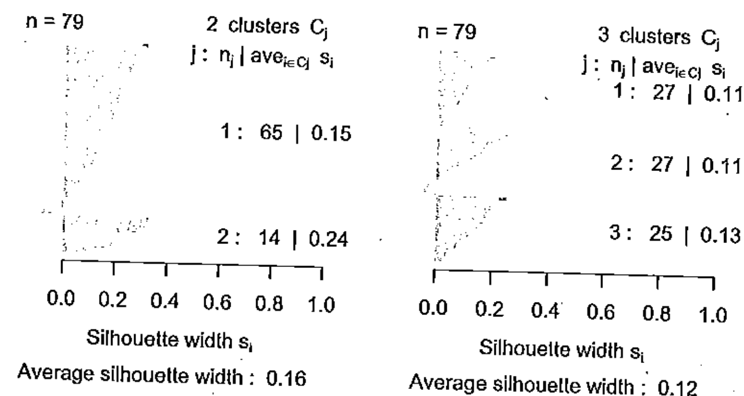

Figure 10.5. Silhouette plots for the PAM clustering of the ALL data: left: two clusters; right: three clusters.

```
> silpam3 = silhouette(pam3)
> plot(silpam3, main="")
```

We can see that there are several samples which have negative silhouette values, and fairly natural questions include "Which samples are these?" and "To what cluster are they closer?" This can be easily determined from the output of the `silhouette` function.

```
> silpam3[silpam3[,"sil_width"] < 0,]
      cluster neighbor sil_width
12026       2        1 -3.97e-05
27003       2        3 -3.21e-02
43004       2        3 -3.30e-02
43001       2        1 -4.15e-02
28031       2        3 -7.80e-02
```

Exercise 10.12
How many samples have negative silhouette widths for the `pam2` *clustering?*

Exercise 10.13
For one of the hierarchical clustering algorithms, divide the data into four clusters and produce a silhouette plot for those four clusters. You will need to read the manual page for the `silhouette` *function to see how to provide the necessary input data.*

10.9 Exploring transformations

Cluster discovery can be aided by the use of variable transformations. We have mentioned multidimensional scaling above in connection with distance assessment. The principal components transformation of a data matrix re-expresses the features using linear combinations of the original variables. The first principal component is the linear combination chosen to possess maximal variance, the second is the linear combination orthogonal to the first possessing maximal variance among all orthogonal combinations, and further principal components are defined (up to p for a rank p matrix) in like fashion. Principal components are readily computed using the singular value decomposition (see the R function `svd`) of the data matrix, and the `prcomp` function will compute them directly. We illustrate the process using the following filtering of the ALL data to 50 genes.

```
> rtt = rowttests(ALLfilt_bcrneg, "mol.biol")
> ordtt = order(rtt$p.value)
> esTT = ALLfilt_bcrneg[ordtt[1:50],]
```

With the raw variables, a five-gene pairwise display is easy to make; we color it with class labels even though we are describing tasks for unsupervised learning (Figure 10.6).

```
> pairs(t(exprs(esTT)[1:5,]),
      col=ifelse(esTT$mol.biol=="NEG", "green", "blue"))
```

Here is how we compute the principal components. We transpose the expression matrix so that gene expression levels are regarded as features of sample objects. In this unsupervised re-expression of the data, clusters corresponding to the different phenotypes are more readily distinguished than they are in the pairwise scatterplot of raw gene expression values (Figure 10.7).

```
> pc = prcomp(t(exprs(esTT)))
```

```
> pairs(pc$x[,1:5], col=ifelse(esTT$mol.biol=="NEG",
      "green", "blue"))
```


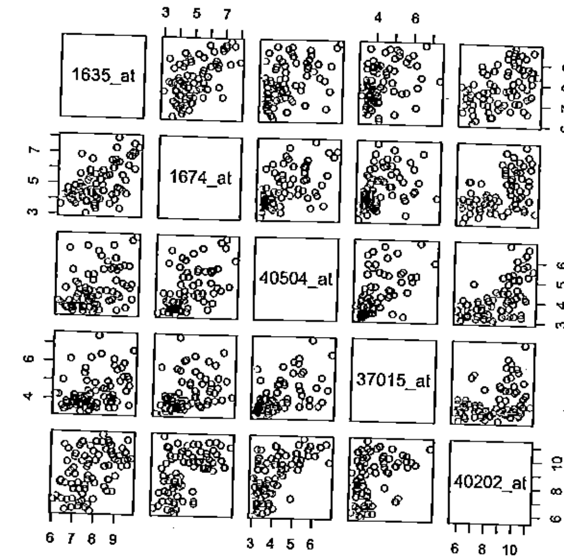

Figure 10.6. A pairs plot for the first five genes in the filtered ALL data.

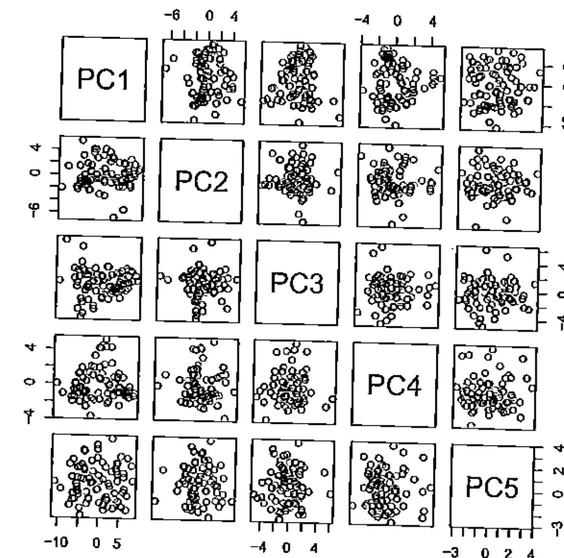

Figure 10.7. A pairs plot for the first five principal components computed from the filtered ALL data.

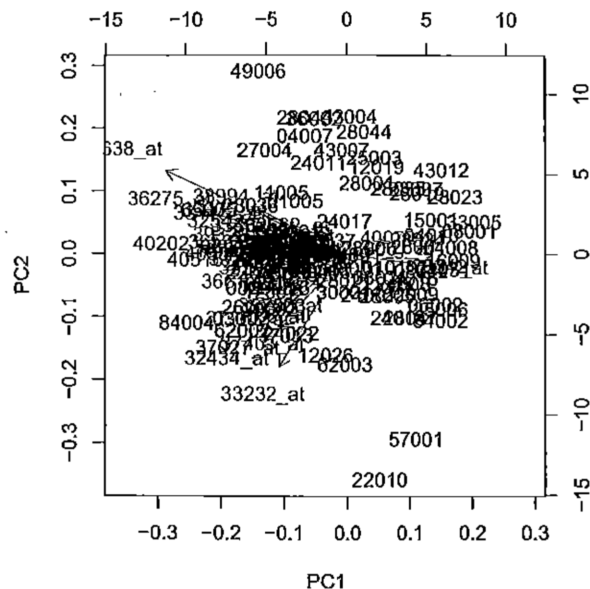

Figure 10.8. A biplot for the first two principal components computed from the filtered ALL data.

Biplots enhance the pairwise principal components display by providing information on directions in which the original variables are transformed to create principal components (Figure 10.8).

```
> biplot(pc)
```

Exercise 10.14
Certain probe set names are prominent in the biplot. Using two-sample tests, explain their roles in discriminating the two phenotypes.

Exercise 10.15
Create a less stringent filtering of the ALL data and generate the associated pairs and biplot displays.

10.10 Remarks

We have given a rudimentary view of the tools available in R for unsupervised machine learning. Most of the ones we have discussed have substantially more capabilities than we have considered and there are many, that are worthwhile that we have not been able to present.

Furthermore, it seems that there is still a great deal of research that can be done in this area. Current topics that need to be addressed are the detection of outlying items and the development of tools that can use additional genomic information in developing and devising the clustering (we gave a simple example, because we concentrated on features obtained from specific GO categories).