Microarray

The fourth case study is from the area of bioinformatics. Namely, we will address the problem of classifying microarray samples into a set of alternative classes. More specifically, given a microarray probe that describes the gene expression level of a patient, we aim to classify this patient into a pre-defined set of genetic mutations of acute lymphoblastic leukemia. This case study addresses several new data mining topics. The main focus, given the characteristics of this type of datasets, is on feature selection, that is, how to reduce the number of features that describe each observation. In our approach to this particular application we will illustrate several general methods for feature selection. Other new data mining topics addressed in this chapter include k-nearest neighbors classifiers, leave one out cross-validation, and some new variants of ensemble models.

5.1 Problem Description and Objectives

Bioinformatics is one of the main areas of application of R. There is even an associated project based on R, with the goal of providing a large set of analysis tools for this domain. The project is called Bioconductor.[1] This case study will use the tools provided by this project to address a supervised classification problem.

5.1.1 Brief Background on Microarray Experiments

One of the main difficulties faced by someone coming from a nonbiological background is the huge amount of "new" terms used in this field. In this very brief background section, we try to introduce the reader to some of the "jargon" in this field and also to provide some mapping to more "standard" data mining terminology.

The analysis of differential gene expression is one of the key applications of DNA microarray experiments. Gene expression microarrays allow us to characterize a set of samples (e.g., individuals) according to their expression levels

[1] http://www.bioconductor.org.

on a large set of genes. In this area a sample is thus an observation (case) of some phenomenon under study. Microarray experiments are the means used to measure a set of "variables" for these observations. The variables here are a large set of genes. For each variable (gene), these experiments measure an expression value. In summary, a dataset is formed by a set of samples (the cases) for which we have measured expression levels on a large set of genes (the variables). If these samples have some disease state associated with them, we may try to approximate the unknown function that maps gene expression levels into disease states. This function can be approximated using a dataset of previously analyzed samples. This is an instantiation of supervised classification tasks, where the target variable is the disease type. The observations in this problem are samples (microarrays, individuals), and the predictor variables are the genes for which we measure a value (the expression level) using a microarray experiment. The key hypothesis here is thus that different disease types can be associated with different gene expression patterns and, moreover, that by measuring these patterns using microarrays we can accurately predict what the disease type of an individual is.

There are several types of technologies created with the goal of obtaining gene expression levels on some sample. Short oligonucleotide arrays are an example of these technologies. The output of oligonucleotide chips is an image that after several pre-processing steps can be mapped into a set of gene expression levels for quite a large set of genes. The bioconductor project has several packages devoted to these pre-processing steps that involve issues like the analysis of the images resulting from the oligonucleotide chips, normalization tasks, and several other steps that are necessary until we reach a set of gene expression scores. In this case study we do not address these initial steps. The interested reader is directed to several informative sources available at the bioconductor site as well as several books (e.g., Hahne et al. (2008)).

In this context, our starting point will be a matrix of gene expression levels that results from these pre-processing steps. This is the information on the predictor variables for our observations. As we will see, there are usually many more predictor variables being measured than samples; that is, we have more predictors than observations. This is a typical characteristic of microarray data sets. Another particularity of these expression matrices is that they appear transposed when compared to what is "standard" for data sets. This means that the rows will represent the predictors (i.e., genes), while the columns are the observations (the samples). For each of the samples we will also need the associated classification. In our case this will be an associated type of mutation of a disease. There may also exist information on other co-variates (e.g., sex and age of the individuals being sampled, etc.).

5.1.2 The ALL Dataset

The dataset we will use comes from a study on acute lymphoblastic leukemia (Chiaretti et al., 2004; Li, 2009). The data consists of microarray

samples from 128 individuals with this type of disease. Actually, there are two different types of tumors among these samples: T-cell ALL (33 samples) and B-cell ALL (95 samples).

We will focus our study on the data concerning the B-cell ALL samples. Even within this latter group of samples we can distinguish different types of mutations. Namely, ALL1/AF4, BCR/ABL, E2A/PBX1, p15/p16 and also individuals with no cytogenetic abnormalities. In our analysis of the B-cell ALL samples we will discard the p15/p16 mutation as we only have one sample. Our modeling goal is to be able to predict the type of mutation of an individual given its microarray assay. Given that the target variable is nominal with 4 possible values, we are facing a supervised classification task.

5.2 The Available Data

The ALL dataset is part of the bioconductor set of packages. To use it, we need to install at least a set of basic packages from bioconductor. We have not included the dataset in our book package because the dataset is already part of the R "universe".

To install a set of basic bioconductor packages and the ALL dataset, we need to carry out the following instructions that assume we have a working Internet connection:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite()
> biocLite("ALL")
```

This only needs to be done for the first time. Once you have these packages installed, if you want to use the dataset, you simply need to do

```
> library(Biobase)
> library(ALL)
> data(ALL)
```

These instructions load the `Biobase` (Gentleman et al., 2004) and the `ALL` (Gentleman et al., 2010) packages. We then load the `ALL` dataset, that creates an object of a special class (`ExpressionSet`) defined by Bioconductor. This class of objects contains significant information concerning a microarray dataset. There are several associated methods to handle this type of object. If you ask R about the content of the `ALL` object, you get the following information:

```
> ALL
```

```
ExpressionSet (storageMode: lockedEnvironment)
assayData: 12625 features, 128 samples
  element names: exprs
phenoData
  sampleNames: 01005, 01010, ..., LAL4  (128 total)
  varLabels and varMetadata description:
    cod:  Patient ID
    diagnosis:  Date of diagnosis
    ...: ...
    date last seen:  date patient was last seen
    (21 total)
featureData
  featureNames: 1000_at, 1001_at, ..., AFFX-YEL024w/RIP1_at  (12625 total)
  fvarLabels and fvarMetadata description: none
experimentData: use 'experimentData(object)'
  pubMedIds: 14684422 16243790
Annotation: hgu95av2
```

The information is divided in several groups. First we have the assay data with the gene expression levels matrix. For this dataset we have 12,625 genes and 128 samples. The object also contains a lot of meta-data about the samples of the experiment. This includes the `phenoData` part with information on the sample names and several associated co-variates. It also includes information on the features (i.e., genes) as well as annotations of the genes from biomedical databases. Finally, the object also contains information that describes the experiment.

There are several methods that facilitate access to all information in the `ExpressionSet` objects. We give a few examples below. We start by obtaining some information on the co-variates associated to each sample:

```
> pD <- phenoData(ALL)
> varMetadata(pD)
```

```
                                                                labelDescription
cod                                                                   Patient ID
diagnosis                                                       Date of diagnosis
sex                                                          Gender of the patient
age                                                  Age of the patient at entry
BT                                  does the patient have B-cell or T-cell ALL
remission       Complete remission(CR), refractory(REF) or NA. Derived from CR
CR                                                        Original remisson data
date.cr                                    Date complete remission if achieved
t(4;11)         did the patient have t(4;11) translocation. Derived from citog
t(9;22)         did the patient have t(9;22) translocation. Derived from citog
cyto.normal                     Was cytogenetic test normal? Derived from citog
citog          original citogenetics data, deletions or t(4;11), t(9;22) status
mol.biol                                                      molecular biology
fusion protein                     which of p190, p210 or p190/210 for bcr/able
mdr                                                        multi-drug resistant
kinet                                        ploidy: either diploid or hyperd.
ccr                             Continuous complete remission? Derived from f.u
```

```
relapse                                                 Relapse? Derived from f.u
transplant   did the patient receive a bone marrow transplant? Derived from f.u
f.u                                                       follow up data available
date last seen                                          date patient was last seen
```

```
> table(ALL$BT)

 B B1 B2 B3 B4  T T1 T2 T3 T4
 5 19 36 23 12  5  1 15 10  2

> table(ALL$mol.biol)

ALL1/AF4  BCR/ABL E2A/PBX1      NEG   NUP-98  p15/p16
      10       37        5       74        1        1

> table(ALL$BT, ALL$mol.bio)

     ALL1/AF4 BCR/ABL E2A/PBX1 NEG NUP-98 p15/p16
  B          0       2        1   2      0       0
  B1        10       1        0   8      0       0
  B2         0      19        0  16      0       1
  B3         0       8        1  14      0       0
  B4         0       7        3   2      0       0
  T          0       0        0   5      0       0
  T1         0       0        0   1      0       0
  T2         0       0        0  15      0       0
  T3         0       0        0   9      1       0
  T4         0       0        0   2      0       0
```

The first two statements obtain the names and descriptions of the existing co-variates. We then obtain some information on the distribution of the samples across the two main co-variates: the `BT` variable that determines the type of acute lymphoblastic leukemia, and the `mol.bio` variable that describes the cytogenetic abnormality found on each sample (`NEG` represents no abnormality).

We can also obtain some information on the genes and samples:

```
> featureNames(ALL)[1:10]

 [1] "1000_at"   "1001_at"   "1002_f_at" "1003_s_at" "1004_at"
 [6] "1005_at"   "1006_at"   "1007_s_at" "1008_f_at" "1009_at"

> sampleNames(ALL)[1:5]

[1] "01005" "01010" "03002" "04006" "04007"
```

This code shows the names of the first 10 genes and the names of the first 5 samples.

As mentioned before, we will focus our analysis of this data on the B-cell ALL cases and in particular on the samples with a subset of the mutations, which will be our target class. The code below obtains the subset of data that we will use:

```
> tgt.cases <- which(ALL$BT %in% levels(ALL$BT)[1:5] &
+                    ALL$mol.bio %in% levels(ALL$mol.bio)[1:4])
> ALLb <- ALL[,tgt.cases]
> ALLb

ExpressionSet (storageMode: lockedEnvironment)
assayData: 12625 features, 94 samples
  element names: exprs
phenoData
  sampleNames: 01005, 01010, ..., LAL5  (94 total)
  varLabels and varMetadata description:
    cod:  Patient ID
    diagnosis:  Date of diagnosis
    ...: ...
    date last seen:  date patient was last seen
    (21 total)
featureData
  featureNames: 1000_at, 1001_at, ..., AFFX-YEL024w/RIP1_at  (12625 total)
  fvarLabels and fvarMetadata description: none
experimentData: use 'experimentData(object)'
  pubMedIds: 14684422 16243790
Annotation: hgu95av2
```

The first statement obtains the set of cases that we will consider. These are the samples with specific values of the `BT` and `mol.bio` variables. Check the calls to the `table()` function we have shown before to see which ones we are selecting. We then subset the original `ALL` object to obtain the 94 samples that will enter our study. This subset of samples only contains some of the values of the `BT` and `mol.bio` variables. In this context, we should update the available levels of these two factors on our new `ALLb` object:

```
> ALLb$BT <- factor(ALLb$BT)
> ALLb$mol.bio <- factor(ALLb$mol.bio)
```

The `ALLb` object will be the dataset we will use throughout this chapter. It may eventually be a good idea to save this object in a local file on your computer, so that you do not need to repeat these pre-processing steps in case you want to start the analysis from scratch:

```
> save(ALLb, file = "myALL.Rdata")
```

5.2.1 Exploring the Dataset

The function `exprs()` allows us to access the gene expression levels matrix:

```
> es <- exprs(ALLb)
> dim(es)

[1] 12625    94
```

The matrix of our dataset has 12,625 rows (the genes/features) and 94 columns (the samples/cases).

In terms of dimensionality, the main challenge of this problem is the fact that there are far too many variables (12,625) for the number of available cases (94). With these dimensions, most modeling techniques will have a hard time obtaining any meaningful result. In this context, one of our first goals will be to reduce the number of variables, that is, eliminate some genes from our analysis. To help in this task, we start by exploring the expression levels data.

The following instruction tells us that most expression values are between 4 and 7:

```
> summary(as.vector(es))

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.985   4.122   5.469   5.624   6.829  14.040
```

A better overview of the distribution of the expression levels can be obtained graphically. We will use a function from package `genefilter` Gentleman et al. (2010). This package must be installed before using it. Please notice that this is a Bioconductor package, and these packages are not installed from the standard R repository. The easiest way to install a Bioconductor package is through the script provided by this project for this effect:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("genefilter")
```

The first instruction loads the script and then we use it do download and install the package. We can now proceed with the above-mentioned graphical display of the distribution of the expression levels:

```
> library(genefilter)
> hist(as.vector(es),breaks=80,prob=T,
+      xlab='Expression Levels',
+      main='Histogram of Overall Expression Levels')
> abline(v=c(median(as.vector(es)),
+            shorth(as.vector(es)),
+            quantile(as.vector(es),c(0.25,0.75))),
+        lty=2,col=c(2,3,4,4))
> legend('topright',c('Median','Shorth','1stQ','3rdQ'),
+        lty=2,col=c(2,3,4,4))
```

The results are shown in Figure 5.1. We have changed the default number of intervals of the function `hist()` that obtains histograms. With the value 80 on the parameter `breaks`, we obtain a fine-grained approximation of the distribution, which is possible given the large number of expression levels we have. On top of the histogram we have plotted several lines showing the median, the first and third quartiles, and the shorth. This last statistic is a robust

estimator of the centrality of a continuous distribution that is implemented by the function `shorth()` of package `genefilter`. It is calculated as the mean of the values in a central interval containing 50% of the observations (i.e., the inter-quartile range).

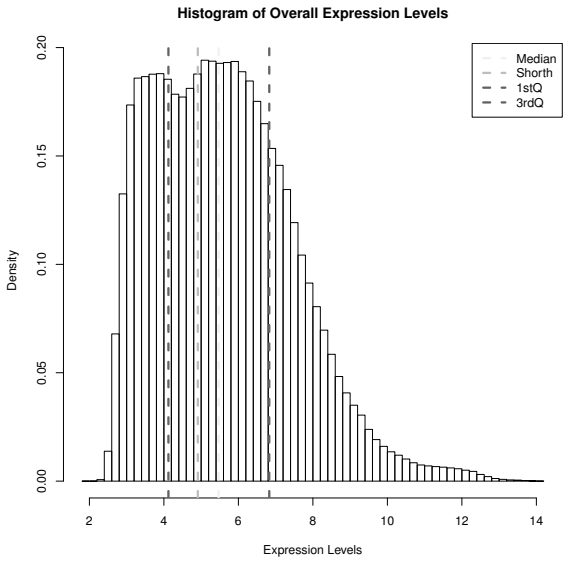

FIGURE 5.1: The distribution of the gene expression levels.

Are the distributions of the gene expression levels of the samples with a particular mutation different from each other? The following code answers this question:

```
> sapply(levels(ALLb$mol.bio), function(x) summary(as.vector(es[,
+     which(ALLb$mol.bio == x)])))

        ALL1/AF4 BCR/ABL E2A/PBX1    NEG
Min.       2.266   2.195    2.268  1.985
1st Qu.    4.141   4.124    4.152  4.111
Median     5.454   5.468    5.497  5.470
Mean       5.621   5.627    5.630  5.622
3rd Qu.    6.805   6.833    6.819  6.832
Max.      14.030  14.040   13.810 13.950
```

As we see, things are rather similar across these subsets of samples and, moreover, they are similar to the global distribution of expression levels.

5.3 Gene (Feature) Selection

Feature selection is an important task in many data mining problems. The general problem is to select the subset of features (variables) of a problem that is more relevant for the analysis of the data. This can be regarded as an instantiation of the more general problem of deciding the weights (importance) of the features in the subsequent modeling stages. Generally, there are two types of approaches to feature selection: (1) filters and (2) wrappers. As mentioned in Section 3.3.2 the former use statistical properties of the features to select the final set, while the latter include the data mining tools in the selection process. Filter approaches are carried out in a single step, while wrappers typically involve a search process where we iteratively look for the set of features that is more adequate for the data mining tools we are applying. Feature wrappers have a clear overhead in terms of computational resources. They involve running the full filter+model+evaluate cycle several times until some convergence criteria are met. This means that for very large data mining problems, they may not be adequate if time is critical. Yet, they will find a solution that is theoretically more adequate for the used modeling tools. The strategies we use and describe in this section can be seen as filter approaches.

5.3.1 Simple Filters Based on Distribution Properties

The first gene filtering methods we describe are based on information concerning the distribution of the expression levels. This type of experimental data usually includes several genes that are not expressed at all or show very small variability. The latter property means that these genes can hardly be used to differentiate among samples. Moreover, this type of microarray usually has several control probes that can be safely removed from our analysis. In the case of this study, which uses Affymetric U95Av2 microarrays, these probes have their name starting with the letters "AFFX".

We can get an overall idea of the distribution of the expression levels of each gene across all individuals with the following graph. We will use the median and inter-quartile range (IQR) as the representatives of these distributions. The following code obtains these scores for each gene and plots the values producing the graph in Figure 5.2:

```
> rowIQRs <- function(em)
+   rowQ(em,ceiling(0.75*ncol(em))) - rowQ(em,floor(0.25*ncol(em)))
> plot(rowMedians(es),rowIQRs(es),
+      xlab='Median expression level',
+      ylab='IQR expression level',
+      main='Main Characteristics of Genes Expression Levels')
```

The function `rowMedians()` from package `Biobase` obtains a vector of the

medians per row of a matrix. This is an efficient implementation of this task. A less efficient alternative would be to use the function `apply()`.[2] The `rowQ()` function is another efficient implementation provided by this package with the goal of obtaining quantiles of a distribution from the rows of a matrix. The second argument of this function is an integer ranging from 1 (that would give us the minimum) to the number of columns of the matrix (that would result in the maximum). In this case we are using this function to obtain the IQR by subtracting the 3rd quartile from the 1st quartile. These statistics correspond to 75% and 25% of the data, respectively. We have used the functions `floor()` and `ceiling()` to obtain the corresponding order in the number of values of each row. Both functions take the integer part of a floating point number, although with different rounding procedures. Experiment with both to see the difference. Using the function `rowQ()`, we have created the function `rowIQRs()` to obtain the IQR of each row.

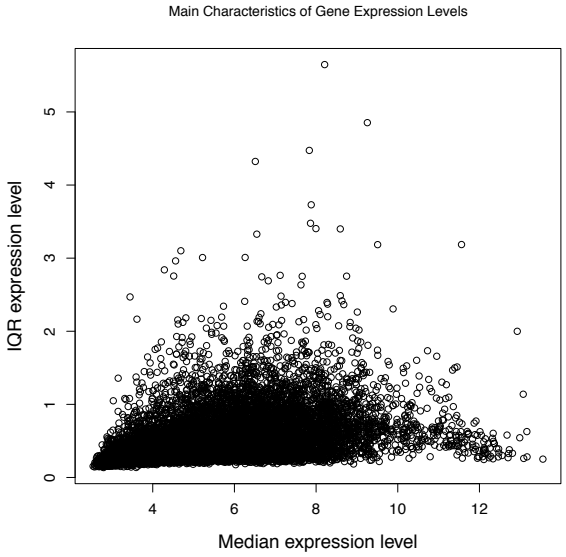

FIGURE 5.2: The median and IQR of the gene expression levels.

Figure 5.2 provides interesting information. Namely, we can observe that a large proportion of the genes have very low variability (IQRs near 0). As mentioned above, if a gene has a very low variability across all samples, then it is reasonably safe to conclude that it will not be useful in discriminating among the different types of mutations of B-cell ALL. This means that we can safely remove these genes from our classification task. We should note

[2]As an exercise, time both alternatives using function `system.time()` to observe the difference.

that there is a caveat on this reasoning. In effect, we are looking at the genes individually. This means that there is some risk that some of these genes with low variability, when put together with other genes, could actually be useful for the classification task. Still, the gene-by-gene approach that we will follow is the most common for these problems as exploring the interactions among genes with datasets of this dimension is not easy. Nevertheless, there are methods that try to estimate the importance of features, taking into account their dependencies. That is the case of the RELIEF method (Kira and Rendel, 1992; Kononenko et al., 1997).

We will use a heuristic threshold based on the value of the IQR to eliminate some of the genes with very low variability. Namely, we will remove any genes with a variability that is smaller than 1/5 of the global IQR. The function `nsFilter()` from the package `genefilter` can be used for this type of filtering:

```
> library(genefilter)
> ALLb <- nsFilter(ALLb,
+                  var.func=IQR,
+                  var.cutoff=IQR(as.vector(es))/5,
+                  feature.exclude="^AFFX")
> ALLb

$eset
ExpressionSet (storageMode: lockedEnvironment)
assayData: 4035 features, 94 samples
  element names: exprs
phenoData
  sampleNames: 01005, 01010, ..., LAL5  (94 total)
  varLabels and varMetadata description:
    cod:  Patient ID
    diagnosis:  Date of diagnosis
    ...: ...
    mol.bio:  molecular biology
    (22 total)
featureData
  featureNames: 41654_at, 35430_at, ..., 34371_at  (4035 total)
  fvarLabels and fvarMetadata description: none
experimentData: use 'experimentData(object)'
  pubMedIds: 14684422 16243790
Annotation: hgu95av2

$filter.log
$filter.log$numLowVar
[1] 4764

$filter.log$numDupsRemoved
[1] 2918

$filter.log$feature.exclude
[1] 19
```

```
$filter.log$numRemoved.ENTREZID
[1] 889
```

As you see, we are left with only 4,035 genes from the initial 12,625. This is a rather significant reduction. Nevertheless, we are still far from a dataset that is "manageable" by most classification models, given that we only have 94 observations.

The result of the `nsFilter()` function is a list with several components. Among these we have several containing information on the removed genes, and also the component `eset` with the "filtered" object. Now that we have seen the result of this filtering, we can update our `ALLb` and `es` objects to contain the filtered data:

```
> ALLb <- ALLb$eset
> es <- exprs(ALLb)
> dim(es)

[1] 4035   94
```

5.3.2 ANOVA Filters

If a gene has a distribution of expression values that is similar across all possible values of the target variable, then it will surely be useless to discriminate among these values. Our next approach builds on this idea. We will compare the mean expression level of genes across the subsets of samples belonging to a certain B-cell ALL mutation, that is, the mean conditioned on the target variable values. Genes for which we have high statistical confidence of having the same mean expression level across the groups of samples belonging to each mutation will be discarded from further analysis.

Comparing means across more than two groups can be carried out using an ANOVA statistical test. In our case study, we have four groups of cases, one for each of the gene mutations of B-cell ALL we are considering. Filtering of genes based on this test is rather easy in R, thanks to the facilities provided by the `genefilter` package. We can carry out this type of filtering as follows:

```
> f <- Anova(ALLb$mol.bio, p = 0.01)
> ff <- filterfun(f)
> selGenes <- genefilter(exprs(ALLb), ff)

> sum(selGenes)

[1] 752

> ALLb <- ALLb[selGenes, ]
> ALLb
```

```
ExpressionSet (storageMode: lockedEnvironment)
assayData: 752 features, 94 samples
  element names: exprs
phenoData
  sampleNames: 01005, 01010, ..., LAL5  (94 total)
  varLabels and varMetadata description:
    cod:  Patient ID
    diagnosis:  Date of diagnosis
    ...: ...
    mol.bio:  molecular biology
    (22 total)
featureData
  featureNames: 266_s_at, 33047_at, ..., 40698_at  (752 total)
  fvarLabels and fvarMetadata description: none
experimentData: use 'experimentData(object)'
  pubMedIds: 14684422 16243790
Annotation: hgu95av2
```

The function `Anova()` creates a new function for carrying out ANOVA filtering. It requires a factor that determines the subgroups of our dataset and a statistical significance level. The resulting function is stored in the variable `f`. The `filterfun()` function works in a similar manner. It generates a filtering function that can be applied to an expression matrix. This application is carried out with the `genefilter()` function that produces a vector with as many elements as there are genes in the given expression matrix. The vector contains logical values. Genes that are considered useful according to the ANOVA statistical test have the value `TRUE`. As you can see, there are only 752. Finally, we can use this vector to filter our `ExpressionSet` object.

Figure 5.3 shows the median and IQR of the genes selected by the ANOVA test. The figure was obtained as follows:

```
> es <- exprs(ALLb)
> plot(rowMedians(es),rowIQRs(es),
+      xlab='Median expression level',
+      ylab='IQR expression level',
+      main='Distribution Properties of the Selected Genes')
```

The variability in terms of IQR and median that we can observe in Figure 5.3 provides evidence that the genes are expressed in different scales of values. Several modeling techniques are influenced by problems where each case is described by a set of variables using different scales. Namely, any method relying on distances between observations will suffer from this type of problem as distance functions typically sum up differences between variable values. In this context, variables with a higher average value will end up having a larger influence on the distance between observations. To avoid this effect, it is usual to standardize (normalize) the data. This transformation consists of subtracting the typical value of the variables and dividing the result by a measure of

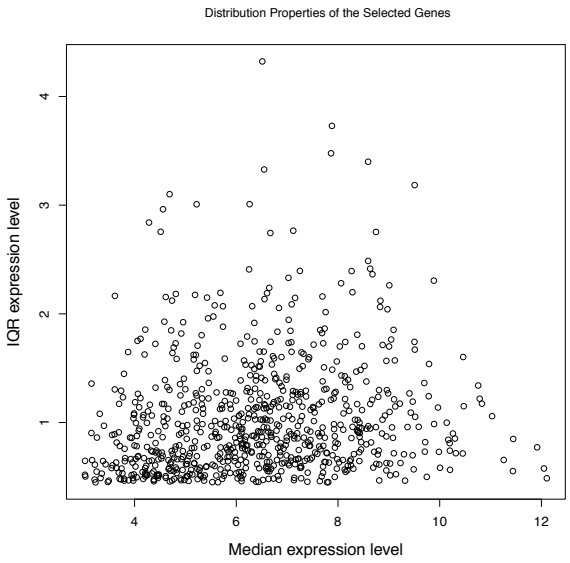

FIGURE 5.3: The median and IQR of the final set of genes.

spread. Given that not all modeling techniques are affected by this data characteristic we will leave this transformation to the modeling stages, making it depend on the tool to be used.

5.3.3 Filtering Using Random Forests

The expression level matrix resulting from the ANOVA filter is already of manageable size, although we still have many more features than observations. In effect, in our modeling attempts described in Section 5.4, we will apply the selected models to this matrix. Nevertheless, one can question whether better results can be obtained with a dataset with a more "standard" dimensionality. In this context, we can try to further reduce the number of features and then compare the results obtained with the different datasets.

Random forests can be used to obtain a ranking of the features in terms of their usefulness for a classification task. In Section 3.3.2 (page 112) we saw an example of using random forests to obtain a ranking of importance of the variables in the context of a prediction problem.

Before proceeding with an illustration of this approach, we will change the names of the genes. The current names are non-standard in terms of what is expected in data frames that are used by many modeling techniques. The function `make.names()` can be used to "solve" this problem as follows:

```
> featureNames(ALLb) <- make.names(featureNames(ALLb))
> es <- exprs(ALLb)
```

The function `featureNames()` provides access to the names of the genes in an `ExpressionSet`.

Random forests can be used to obtain a ranking of the genes as follows,

```
> library(randomForest)
> dt <- data.frame(t(es), Mut = ALLb$mol.bio)
> rf <- randomForest(Mut ~ ., dt, importance = T)
> imp <- importance(rf)
> imp <- imp[, ncol(imp) - 1]
> rf.genes <- names(imp)[order(imp, decreasing = T)[1:30]]
```

We construct a training set by adding the mutation information to the transpose of the expression matrix.[3] We then obtain a random forest with the parameter `importance` set to `TRUE` to obtain estimates of the importance of the variables. The function `importance()` is used to obtain the relevance of each variable. This function actually returns several scores on different columns, according to different criteria and for each class value. We select the column with the variable scores measured as the estimated mean decrease in classification accuracy when each variable is removed in turn. Finally, we obtain the genes that appear at the top 30 positions of the ranking generated by these scores.

We may be curious about the expression levels distribution of theses 30 genes across the different mutations. We can obtain the median level for these top 30 genes as follows:

```
> sapply(rf.genes, function(g) tapply(dt[, g], dt$Mut, median))
```

```
         X40202_at X1674_at X1467_at X1635_at X37015_at X34210_at
ALL1/AF4  8.550639 3.745752 3.708985 7.302814  3.752649  5.641130
BCR/ABL   9.767293 5.833510 4.239306 8.693082  4.857105  9.204237
E2A/PBX1  7.414635 3.808258 3.411696 7.562676  6.579530  8.198781
NEG       7.655605 4.244791 3.515020 7.324691  3.765741  8.791774
         X32116_at X34699_at X40504_at X41470_at X41071_at X36873_at
ALL1/AF4  7.115400  4.253504  3.218079  9.616743  7.698420  7.040593
BCR/ABL   7.966959  6.315966  4.924310  5.205797  6.017967  3.490262
E2A/PBX1  7.359097  6.102031  3.455316  3.931191  6.058185  3.634471
NEG       7.636213  6.092511  3.541651  4.157748  6.573731  3.824670
         X35162_s_at X38323_at X1134_at X32378_at X1307_at X1249_at
ALL1/AF4    4.398885  4.195967 7.846189  8.703860 3.368915 3.582763
BCR/ABL     4.924553  4.866452 8.475578  9.694933 4.945270 4.477659
E2A/PBX1    4.380962  4.317550 8.697500 10.066073 4.678577 3.257649
NEG         4.236335  4.286104 8.167493  9.743168 4.863930 3.791764
         X33774_at X40795_at X36275_at X34850_at X33412_at X37579_at
ALL1/AF4  6.970072  3.867134  3.618819  5.426653 10.757286  7.614200
```

[3]Remember that expression matrices have genes (variables) on the rows.

```
BCR/ABL   8.542248  4.544239  6.259073  6.898979  6.880112  8.231081
E2A/PBX1  7.385129  4.151637  3.635956  5.928574  5.636466  9.494368
NEG       7.348818  3.909532  3.749953  6.327281  5.881145  8.455750
          X37225_at X39837_s_at X37403_at X37967_at X2062_at X35164_at
ALL1/AF4  5.220668    6.633188  5.888290  8.130686 9.409753  5.577268
BCR/ABL   3.460902    7.374046  5.545761  9.274695 7.530185  6.493672
E2A/PBX1  7.445655    6.708400  4.217478  8.260236 7.935259  7.406714
NEG       3.387552    6.878846  4.362275  8.986204 7.086033  7.492440
```

We can observe several interesting differences between the median expression level across the types of mutations, which provides a good indication of the discriminative power of these genes. We can obtain even more detail by graphically inspecting the concrete expression values of these genes for the 94 samples:

```
> library(lattice)
> ordMut <- order(dt$Mut)
> levelplot(as.matrix(dt[ordMut,rf.genes]),
+           aspect='fill', xlab='', ylab='',
+           scales=list(
+             x=list(
+               labels=c('+','-','*','|')[as.integer(dt$Mut[ordMut])],
+               cex=0.7,
+               tck=0)
+             ),
+           main=paste(paste(c('"+"','"-"','"*"','"|"),
+                            levels(dt$Mut)
+                           ),
+                      collapse='; '),
+           col.regions=colorRampPalette(c('white','orange','blue'))
+           )
```

The graph obtained with this code is shown in Figure 5.4. We observe that there are several genes with marked differences in expression level across the different mutations. For instance, there are obvious differences in expression level at gene `X36275_at` between `ALL1/AF4` and `BCR/ABL`. To obtain this graph we used the function `levelplot()` of the `lattice` package. This function can plot a color image of a matrix of numbers. In this case we have used it to plot our expression level matrix with the samples ordered by type of mutation.

5.3.4 Filtering Using Feature Clustering Ensembles

The approach described in this section uses a clustering algorithm to obtain 30 groups of variables that are supposed to be similar. These 30 variable clusters will then be used to obtain an ensemble classification model where m models will be obtained with 30 variables, each one randomly chosen from one of the 30 clusters.

Ensembles are learning methods that build a set of predictive models and

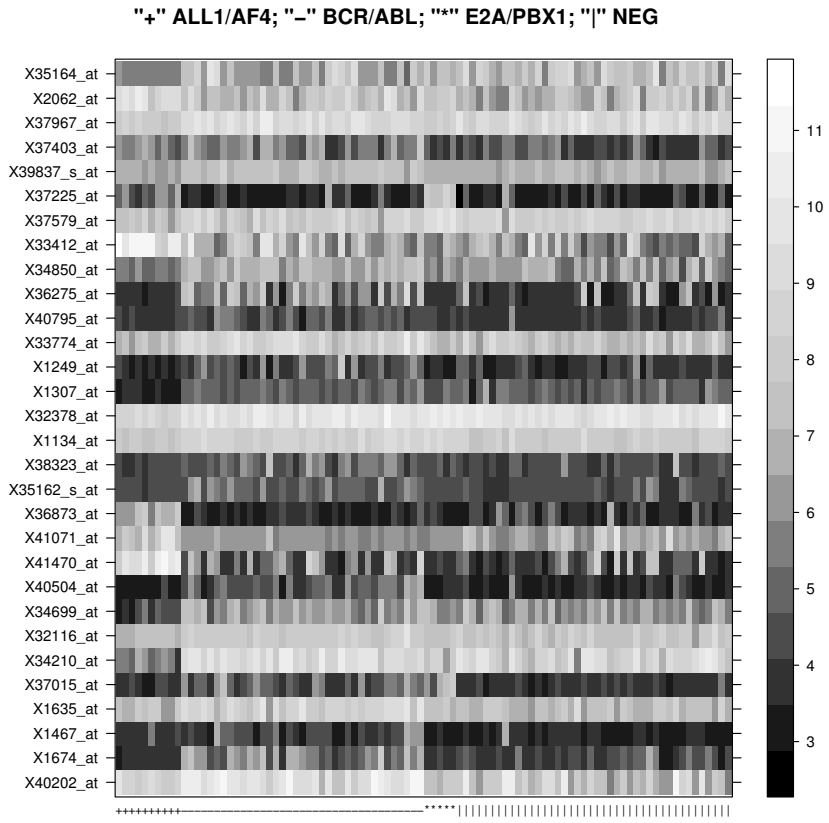

FIGURE 5.4: The expression level of the 30 genes across the 94 samples.

then classify new observations using some form of averaging of the predictions of these models. They are known for often outperforming the individual models that form the ensemble. Ensembles are based on some form of diversity among the individual models. There are many forms of creating this diversity. It can be through different model parameter settings or by different samples of observations used to obtain each model, for instance. Another alternative is to use different predictors for each model in the ensemble. The ensembles we use in this section follow this latter strategy. This approach works better if the pool of predictors from which we obtain different sets is highly redundant. We will assume that there is some degree of redundancy on our set of features generated by the ANOVA filter. We will try to model this redundancy by clustering the variables. Clustering methods are based on distances, in this case distances between variables. Two variables are near (and thus similar) each other if their expression values across the 94 samples are similar. By clustering the variables we expect to find groups of genes that are similar to each other. The `Hmisc` package contains a function that uses a hierarchical clustering algorithm to cluster the variables of a dataset. The name of this function is `varclus()`. We can use it as follows:

```
> library(Hmisc)
> vc <- varclus(t(es))
> clus30 <- cutree(vc$hclust, 30)
> table(clus30)

clus30
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
27 26 18 30 22 18 24 46 22 20 24 18 56 28 47 32 22 31 18 22 18 33 20 20 21
26 27 28 29 30
17  9 19 30 14
```

We used the function `cutree()` to obtain a clustering formed by 30 groups of variables. We then checked how many variables (genes) belong to each cluster. Based on this clustering we can create sets of predictors by randomly selecting one variable from each cluster. The reasoning is that members of the same cluster will be similar to each other and thus somehow redundant.

The following function facilitates the process by generating one set of variables via randomly sampling from the selected number of clusters (defaulting to 30):

```
> getVarsSet <- function(cluster,nvars=30,seed=NULL,verb=F)
+ {
+   if (!is.null(seed)) set.seed(seed)
+
+   cls <- cutree(cluster,nvars)
+   tots <- table(cls)
+   vars <- c()
+   vars <- sapply(1:nvars,function(clID)
+     {
```

```
+       if (!length(tots[clID])) stop('Empty cluster! (',clID,')')
+       x <- sample(1:tots[clID],1)
+       names(cls[cls==clID])[x]
+     })
+   if (verb)  structure(vars,clusMemb=cls,clusTots=tots)
+   else       vars
+ }
> getVarsSet(vc$hclust)

 [1] "X41346_at"   "X33047_at"   "X1044_s_at"  "X38736_at"   "X39814_s_at"
 [6] "X649_s_at"   "X41672_at"   "X36845_at"   "X40771_at"   "X38370_at"
[11] "X36083_at"   "X34964_at"   "X35228_at"   "X40855_at"   "X41038_at"
[16] "X40495_at"   "X40419_at"   "X1173_g_at"  "X40088_at"   "X879_at"
[21] "X39135_at"   "X34798_at"   "X39649_at"   "X39774_at"   "X39581_at"
[26] "X37024_at"   "X32585_at"   "X41184_s_at" "X33305_at"   "X41266_at"

> getVarsSet(vc$hclust)

 [1] "X40589_at"   "X33598_r_at" "X41015_at"   "X38999_s_at" "X37027_at"
 [6] "X32842_at"   "X37951_at"   "X35693_at"   "X36874_at"   "X41796_at"
[11] "X1462_s_at"  "X31751_f_at" "X34176_at"   "X40855_at"   "X1583_at"
[16] "X38488_s_at" "X32542_at"   "X32961_at"   "X32321_at"   "X879_at"
[21] "X38631_at"   "X37718_at"   "X948_s_at"   "X38223_at"   "X34256_at"
[26] "X1788_s_at"  "X38271_at"   "X37610_at"   "X33936_at"   "X36899_at"
```

Each time we call this function, we will get a "new" set of 30 variables. Using this function it is easy to generate a set of datasets formed by different predictors and then obtain a model using each of these sets. In Section 5.4 we present a function that obtains ensembles using this strategy.

Further readings on feature selection

Feature selection is a well-studied topic in many disciplines. Good overviews and references of the work in the area of data mining can be obtained in Liu and Motoda (1998), Chizi and Maimon (2005), and Wettschereck et al. (1997).

5.4 Predicting Cytogenetic Abnormalities

This section describes our modeling attempts for the task of predicting the type of cytogenetic abnormalities of the B-cell ALL cases.

5.4.1 Defining the Prediction Task

The data mining problem we are facing is a predictive task. More precisely, it is a classification problem. Predictive classification consists of obtaining models

designed with the goal of forecasting the value of a nominal target variable using information on a set of predictors. The models are obtained using a set of labeled observations of the phenomenon under study, that is, observations for which we know both the values of the predictors and of the target variable.

In this case study our target variable is the type of cytogenetic abnormality of a B-cell ALL sample. In our selected dataset, this variable will take four possible values: `ALL1/AF4`, `BCR/ABL`, `E2A/PBX1`, and `NEG`. Regarding the predictors, they will consist of a set of selected genes for which we have measured an expression value. In our modeling attempts we will experiment with different sets of selected genes, based on the study described in Section 5.3. This means that the number of predictors (features) will vary depending on these trials. Regarding the number of observations, they will consist of 94 cases of B-cell ALL.

5.4.2 The Evaluation Metric

The prediction task is a multi-class classification problem. Predictive classification models are usually evaluated using the error rate or its complement, the accuracy. Nevertheless, there are several alternatives, such as the area under the ROC curve, pairs of measures (e.g., precision and recall), and also measures of the accuracy of class probability estimates (e.g., the Brier score). The package `ROCR` provides a good sample of these measures.

The selection of the evaluation metric for a given problem often depends on the goals of the user. This is a difficult decision that is often impaired by incomplete information such as the absence of information on the costs of misclassifying a class i case with class j (known as the misclassification costs).

In our case study we have no information on the misclassification costs, and thus we assume that it is equally serious to misclassify, for instance, an `E2A/PBX1` mutation as `NEG`, as it is to misclassify `ALL1/AF4` as `BCR/ABL`. Moreover, we have more than two classes, and generalizations of ROC analysis to multi-class problems are not so well established, not to mention recent drawbacks discovered in the use of the area under the ROC curve (Hand, 2009). In this context, we will resort to the use of the standard accuracy that is measured as

$$\overline{acc} = 1 - \frac{1}{N} \sum_{i=1}^{N} L_{0/1}(y_i, \hat{y}_i) \tag{5.1}$$

where N is the size of test sample, and $L_{0/1}()$ is a loss function defined as

$$L_{0/1}(y_i, \hat{y}_i) = \begin{cases} 0 & \text{if } y_i = \hat{y}_i \\ 1 & \text{otherwise} \end{cases} \tag{5.2}$$

5.4.3 The Experimental Procedure

The number of observations of the dataset we will use is rather small: 94 cases. In this context, the more adequate experimental methodology to obtain reliable estimates of the error rate is the Leave-One-Out Cross-Validation (LOOCV) method. LOOCV is a special case of the k-fold cross-validation experimental methodology that we have used before, namely, when k equals the number of observations. Briefly, LOOCV consists of obtaining N models, where N is the dataset size, and each model is obtained using $N-1$ cases and tested on the observation that was left out. In the book package you may find the function `loocv()` that implements this type of experiment. This function uses a process similar to the other functions we have described in previous chapters for experimental comparisons. Below is a small illustration of its use with the `iris` dataset:

```
> data(iris)
> rpart.loocv <- function(form,train,test,...) {
+   require(rpart,quietly=T)
+   m <- rpart(form,train,...)
+   p <- predict(m,test,type='class')
+   c(accuracy=ifelse(p == resp(form,test),100,0))
+ }
> exp <- loocv(learner('rpart.loocv',list()),
+              dataset(Species~.,iris),
+              loocvSettings(seed=1234,verbose=F))

> summary(exp)

== Summary of a Leave One Out Cross Validation  Experiment ==

 LOOCV experiment with verbose =  FALSE  and seed = 1234

* Dataset ::  iris
* Learner  ::  rpart.loocv  with parameters:

* Summary of Experiment Results:

         accuracy
avg      93.33333
std      25.02795
min       0.00000
max     100.00000
invalid   0.00000
```

The function `loocv()` takes the usual three arguments: the learner, the dataset, and the settings of the experiment. It returns an object of class `loocvRun` that we can use with the function `summary()` to obtain the results of the experiment.

The user-defined function (`rpart.loocv()` in the example above) should run the learner, use it for obtaining predictions for the test set, and return a vector with whatever evaluation statistics we wish to estimate by LOOCV. In this small illustration it simply calculates the accuracy of the learner. We should recall that in LOOCV the test set is formed by a single observation on each iteration of the experimental process so in this case we only need to check whether the prediction is equal to the true value.

5.4.4 The Modeling Techniques

As discussed before we will use three different datasets that differ in the predictors that are used. One will have all genes selected by an ANOVA test, while the other two will select 30 of these genes. All datasets will contain 94 cases of B-cell ALL. With the exception of the target variable, all information is numeric.

To handle this problem we have selected three different modeling techniques. Two of them have already been used before in this book. They are random forests and support vector machines (SVMs). They are recognized as some of the best off-the-shelf prediction methods. The third algorithm we will try on this problem is new. It is a method based on distances between observations, known as k-nearest neighbors.

The use of random forests is motivated by the fact that these models are particularly adequate to handle problems with a large number of features. This property derives from the algorithm used by these methods (see Section 5.4.4.1) that randomly selects subsets of the full set of features of a problem. Regarding the use of k-nearest neighbors, the motivation lies on the assumption that samples with the same mutation should have a similar gene "signature," that is, should have similar expression values on the genes we use to describe them. The validity of this assumption is strongly dependent on the genes selected to describe the samples. Namely, they should have good discrimination properties across the different mutations. As we will see in Section 5.4.4.2, k-nearest neighbors methods work by assessing similarities between cases, and thus they seem adequate for this assumption. Finally, the use of SVMs is justified with the goal of trying to explore nonlinear relationships that may eventually exist between gene expression and cytogenetic abnormalities.

SVMs were described in Section 3.4.2.2 (page 127). They are highly nonlinear models that can be used on both regression and classification problems. Once again, among the different implementations of SVMs that exist in R, we will use the `svm()` function of package `e1071`.

5.4.4.1 Random Forests

Random forests (Breiman, 2001) are an example of an ensemble model, that is, a model that is formed by a set of simpler models. In particular, random

forests consist of a set of decision trees, either classification or regression trees, depending on the problem being addressed. The user decides the number of trees in the ensemble. Each tree is learned using a bootstrap sample obtained by randomly drawing N cases with replacement from the original dataset, where N is the number of cases in that dataset. With each of these training sets, a different tree is obtained. Each node of these trees is chosen considering only a random subset of the predictors of the original problem. The size of these subsets should be much smaller than the number of predictors in the dataset. The trees are fully grown, that is, they are obtained without any post-pruning step. More details on how tree-based models are obtained appear in Section 2.6.2 (page 71).

The predictions of these ensembles are obtained by averaging over the predictions of each tree. For classification problems this consists of a voting mechanism. The class that gets more votes across all trees is the prediction of the ensemble. For regression, the values predicted by each tree are averaged to obtain the random forest prediction.

In R, random forests are implemented in the package `randomForest`. We have already seen several examples of the use of the functions provided by this package throughout the book, namely, for feature selection.

Further readings on random forests

The reference on Random Forests is the original work by Breiman (2001). Further information can also be obtained at the site http://stat-www.berkeley.edu/users/breiman/RandomForests/.

5.4.4.2 k-Nearest Neighbors

The k-nearest neighbors algorithm belongs to the class of so-called *lazy learners*. These types of techniques do not actually obtain a model from the training data. They simply store this dataset. Their main work happens at prediction time. Given a new test case, its prediction is obtained by searching for similar cases in the training data that was stored. The k most similar training cases are used to obtain the prediction for the given test case. In classification problems, this prediction is usually obtained by voting and thus an odd number for k is desirable. However, more elaborate voting mechanisms that take into account the distance of the test case to each of the k neighbors are also possible. For regression, instead of voting we have an average of the target variable values of the k neighbors.

This type of model is strongly dependent on the notion of similarity between cases. This notion is usually defined with the help of a metric over the input space defined by the predictor variables. This metric is a distance function that can calculate a number representing the "difference" between any two observations. There are many distance functions, but a rather frequent selection is the Euclidean distance function that is defined as

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{k=1}^{p} (X_{i,k} - X_{j,k})^2} \tag{5.3}$$

where p is the number of predictors, and $\mathbf{x}_i$ and $\mathbf{x}_j$ are two observations.

These methods are thus very sensitive to both the selected metric and also to the presence of irrelevant variables that may distort the notion of similarity. Moreover, the scale of the variables should be uniform; otherwise we might underestimate some of the differences in variables with lower average values.

The choice of the number of neighbors (k) is also an important parameter of these methods. Frequent values include the numbers in the set $\{1, 3, 5, 7, 11\}$, but obviously these are just heuristics. However, we can say that larger values of k should be avoided because there is the risk of using cases that are already far away from the test case. Obviously, this depends on the density of the training data. Too sparse datasets incur more of this risk. As with any learning model, the "ideal" parameter settings can be estimated through some experimental methodology.

In R, the package `class` (Venables and Ripley, 2002) includes the function `knn()` that implements this idea. Below is an illustrative example of its use on the `iris` dataset:

```
> library(class)
> data(iris)
> idx <- sample(1:nrow(iris), as.integer(0.7 * nrow(iris)))
> tr <- iris[idx, ]
> ts <- iris[-idx, ]
> preds <- knn(tr[, -5], ts[, -5], tr[, 5], k = 3)
> table(preds, ts[, 5])

preds        setosa versicolor virginica
  setosa         14          0         0
  versicolor      0         14         2
  virginica       0          1        14
```

As you see, the function `knn()` uses a nonstandard interface. The first argument is the training set with the exception of the target variable column. The second argument is the test set, again without the target. The third argument includes the target values of the training data. Finally, there are several other parameters controlling the method, among which the parameter `k` determines the number of neighbors. We can create a small function that enables the use of this method in a more standard formula-type interface:

```
> kNN <- function(form, train, test, norm = T, norm.stats = NULL,
+     ...) {
+     require(class, quietly = TRUE)
+     tgtCol <- which(colnames(train) == as.character(form[[2]]))
+     if (norm) {
```

```
+         if (is.null(norm.stats))
+             tmp <- scale(train[, -tgtCol], center = T, scale = T)
+         else tmp <- scale(train[, -tgtCol], center = norm.stats[[1]],
+             scale = norm.stats[[2]])
+         train[, -tgtCol] <- tmp
+         ms <- attr(tmp, "scaled:center")
+         ss <- attr(tmp, "scaled:scale")
+         test[, -tgtCol] <- scale(test[, -tgtCol], center = ms,
+             scale = ss)
+     }
+     knn(train[, -tgtCol], test[, -tgtCol], train[, tgtCol],
+         ...)
+ }
> preds.norm <- kNN(Species ~ ., tr, ts, k = 3)
> table(preds.norm, ts[, 5])

preds.norm   setosa versicolor virginica
  setosa         14          0         0
  versicolor      0         14         3
  virginica       0          1        13

> preds.notNorm <- kNN(Species ~ ., tr, ts, norm = F, k = 3)
> table(preds.notNorm, ts[, 5])

preds.notNorm setosa versicolor virginica
   setosa         14          0         0
   versicolor      0         14         2
   virginica       0          1        14
```

This function allows the user to indicate if the data should be normalized prior to the call to the `knn()` function. This is done through parameter `norm`. In the example above, you see two examples of its use. A third alternative is to provide the centrality and spread statistics as a list with two components in the argument `norm.stats`. If this is not done, the function will use the means as centrality estimates and the standard deviations as statistics of spread. In our experiments we will use this facility to call the function with medians and IQRs. The function `kNN()` is actually included in our book package so you do not need to type its code.

Further readings on k-nearest neighbors

The standard reference on this type of methods is the work by Cover and Hart (1967). Good overviews can be found in the works by Aha et al. (1991) and Aha (1997). Deeper analysis can be found in the PhD theses by Aha (1990) and Wettschereck (1994). A different, but related, perspective of lazy learning is the use of so-called local models (Nadaraya, 1964; Watson, 1964). Good references on this vast area are Atkeson et al. (1997) and Cleveland and Loader (1996).

5.4.5 Comparing the Models

This section describes the process we have used to compare the selected models using a LOOCV experimental methodology.

In Section 5.3, we have seen examples of several feature selection methods. We have used some basic filters to eliminate genes with low variance and also control probes. Next, we applied a method based on the conditioned distribution of the expression levels with respect to the target variable. This method was based on an ANOVA statistical test. Finally, from the results of this test we tried to further reduce the number of genes using random forests and clustering of the variables. With the exception of the first simple filters, all other methods depend somehow on the target variable values. We may question whether these filtering stages should be carried out before the experimental comparisons, or if we should integrate these steps into the processes being compared. If our goal is to obtain an unbiased estimate of the classification accuracy of our methodology on new samples, then we should include these filtering stages as part of the data mining processes being evaluated and compared. Not doing so would mean that the estimates we obtain are biased by the fact that the genes used to obtain the models were selected using information of the test set. In effect, if we use all datasets to decide which genes to use, then we are using information on this selection process that should be unknown as it is part of the test data. In this context, we will include part of the filtering stages within the user-defined functions that implement the models we will compare.

For each iteration of the LOOCV process, a feature selection process, is carried out, prior to the predictive model construction, using only the training data provided by the LOOCV routines. The initial simple filtering step will be carried out before the LOOCV comparison. The genes removed by this step would not change if we do it inside the LOOCV process. Control probes would always be removed, and the genes removed due to very low variance would most probably still be removed if a single sample is not given (which is what the LOOCV process does at each iteration).

We will now describe the user-defined functions that need to be supplied to the LOOCV routines for running the experiments. For each of the modeling techniques, we will evaluate several variants. These alternatives differ not only on several parameters of the techniques themselves, but also on the feature selection process that is used. The following list includes the information on these variants for each modeling technique:

```
> vars <- list()
> vars$randomForest <- list(ntree=c(500,750,100),
+                           mtry=c(5,15,30),
+                           fs.meth=list(list('all'),
+                                        list('rf',30),
+                                        list('varclus',30,50)))
> vars$svm <- list(cost=c(1,100,500),
+                  gamma=c(0.01,0.001,0.0001),
```

```
+                 fs.meth=list(list('all'),
+                              list('rf',30),
+                              list('varclus',30,50)))
> vars$knn <- list(k=c(3,5,7,11),
+                 norm=c(T,F),
+                 fs.meth=list(list('all'),
+                              list('rf',30),
+                              list('varclus',30,50)))
```

The list has three components, one for each of the algorithms being compared. For each model the list includes the parameters that should be used. For each of the parameters a set of values is given. The combinations of all these possible values will determine the different variants of the systems. Regarding random forests, we will consider three values for the parameter `ntree` that sets the number of trees in the ensemble, and three values for the `mtry` parameter that determines the size of the random subset of features to use when deciding the test for each tree node. The last parameter (`fs.meth`) provides the alternatives for the feature selection phase that we describe below. With respect to SVMs, we consider three different values for both the `cost` and `gamma` parameters. Finally, for the k-nearest neighbors, we try four values for k and two values for the parameter that determines if the predictors data is to be normalized or not.

As mentioned above, for each of the learners we consider three alternative feature sets (the parameter `fs.meth`). The first alternative (`list('all')`) uses all the features resulting from the ANOVA statistical test. The second (`list('rf',30)`) selects 30 genes from the set obtained with the ANOVA test, using the feature ranking obtained with a random forest. The final alternative select 30 genes using the variable clustering ensemble strategy that we described previously and then builds an ensemble using 50 models with 30 predictors randomly selected from the variable clusters. In order to implement the idea of the ensembles based on different variable sets generated by a clustering of the genes, we have created the following function:

```
> varsEnsembles <- function(tgt,train,test,
+                           varsSets,
+                           baseLearner,blPars,
+                           verb=F)
+ {
+   preds <- matrix(NA,ncol=length(varsSets),nrow=NROW(test))
+   for(v in seq(along=varsSets)) {
+     if (baseLearner=='knn')
+       preds[,v] <- knn(train[,-which(colnames(train)==tgt)],
+                        test[,-which(colnames(train)==tgt)],
+                        train[,tgt],blPars)
+     else {
+       m <- do.call(baseLearner,
+                    c(list(as.formula(paste(tgt,
+                                            paste(varsSets[[v]],
```

```
+                                                  collapse='+'),
+                                           sep='~')),
+                           train[,c(tgt,varsSets[[v]])]),
+                     blPars)
+                   )
+       if (baseLearner == 'randomForest')
+         preds[,v] <- do.call('predict',
+                              list(m,test[,c(tgt,varsSets[[v]])],
+                                   type='response'))
+       else
+         preds[,v] <- do.call('predict',
+                              list(m,test[,c(tgt,varsSets[[v]])]))
+     }
+   }
+   ps <- apply(preds,1,function(x)
+                  levels(factor(x))[which.max(table(factor(x)))])
+   ps <- factor(ps,
+                levels=1:nlevels(train[,tgt]),
+                labels=levels(train[,tgt]))
+   if (verb) structure(ps,ensemblePreds=preds) else ps
+ }
```

The first arguments of this function are the name of the target variable, the training set, and the test set. The next argument (`varsSets`) is a list containing the sets of variable names (the obtained clusters) from which we should sample a variable to generate the predictors of each member of the ensemble. Finally, we have two arguments (`baseLearner` and `blPars`) that provide the name of the function that implements the learner to be used on each member of the ensemble and respective list of learning arguments. The result of the function is the set of predictions of the ensemble for the given test set. These predictions are obtained by a voting mechanism among the members of the ensemble. The difference between the members of the ensemble lies only in the predictors that are used, which are determined by the `varsSets` parameters. These sets result from a variable clustering process, as mentioned in Section 5.3.4.

Given the similarity of the tasks to be carried out by each of the learners, we have created a single user-defined modeling function that will receive as one of the parameters the learner that is to be used. The function `genericModel()` that we present below implements this idea:

```
> genericModel <- function(form,train,test,
+                          learner,
+                          fs.meth,
+                          ...)
+   {
+     cat('=')
+     tgt <- as.character(form[[2]])
+     tgtCol <- which(colnames(train)==tgt)
```

```
+
+     # Anova filtering
+     f <- Anova(train[,tgt],p=0.01)
+     ff <- filterfun(f)
+     genes <- genefilter(t(train[,-tgtCol]),ff)
+     genes <- names(genes)[genes]
+     train <- train[,c(tgt,genes)]
+     test <- test[,c(tgt,genes)]
+     tgtCol <- 1
+
+     # Specific filtering
+     if (fs.meth[[1]]=='varclus') {
+       require(Hmisc,quietly=T)
+       v <- varclus(as.matrix(train[,-tgtCol]))
+       VSs <- lapply(1:fs.meth[[3]],function(x)
+                     getVarsSet(v$hclust,nvars=fs.meth[[2]]))
+       pred <- varsEnsembles(tgt,train,test,VSs,learner,list(...))
+
+     } else {
+       if (fs.meth[[1]]=='rf') {
+         require(randomForest,quietly=T)
+         rf <- randomForest(form,train,importance=T)
+         imp <- importance(rf)
+         imp <- imp[,ncol(imp)-1]
+         rf.genes <- names(imp)[order(imp,decreasing=T)[1:fs.meth[[2]]]]
+         train <- train[,c(tgt,rf.genes)]
+         test <- test[,c(tgt,rf.genes)]
+       }
+
+       if (learner == 'knn')
+         pred <- kNN(form,
+              train,
+              test,
+              norm.stats=list(rowMedians(t(as.matrix(train[,-tgtCol]))),
+                              rowIQRs(t(as.matrix(train[,-tgtCol])))),
+              ...)
+       else {
+         model <- do.call(learner,c(list(form,train),list(...)))
+         pred <- if (learner != 'randomForest') predict(model,test)
+                 else predict(model,test,type='response')
+       }
+
+     }
+
+     c(accuracy=ifelse(pred == resp(form,test),100,0))
+   }
```

This user-defined function will be called from within the LOOCV routines for each iteration of the process. The experiments with all these variants on

the microarray data will take a long time to complete.[4] In this context, we do not recommend that you run the following experiments unless you are aware of this temporal constraint. The objects resulting from this experiment are available at the book Web page so that you are able to proceed with the rest of the analysis without having to run all these experiments. The code to run the full experiments is the following:

```
> require(class,quietly=TRUE)
> require(randomForest,quietly=TRUE)
> require(e1071,quietly=TRUE)
> load('myALL.Rdata')
> es <- exprs(ALLb)
> # simple filtering
> ALLb <- nsFilter(ALLb,
+                  var.func=IQR,var.cutoff=IQR(as.vector(es))/5,
+                  feature.exclude="^AFFX")
> ALLb <- ALLb$eset
> # the dataset
> featureNames(ALLb) <- make.names(featureNames(ALLb))
> dt <- data.frame(t(exprs(ALLb)),Mut=ALLb$mol.bio)
> DSs <- list(dataset(Mut ~ .,dt,'ALL'))
> # The learners to evaluate
> TODO <- c('knn','svm','randomForest')
> for(td in TODO) {
+   assign(td,
+          experimentalComparison(
+               DSs,
+               c(
+                 do.call('variants',
+                         c(list('genericModel',learner=td),
+                           vars[[td]],
+                           varsRootName=td))
+                ),
+               loocvSettings(seed=1234,verbose=F)
+                             )
+          )
+   save(list=td,file=paste(td,'Rdata',sep='.'))
+ }
```

This code uses the function `experimentalComparison()` to test all variants using the LOOCV method. The code uses the function `variants()` to generate all `learner` objects from the variants provided by the components of list `vars` that we have seen above. Each of these variants will be evaluated by an LOOCV process. The results of the code are three `compExp` objects with the names `knn`, `svm`, and `randomForest`. Each of these objects contains the results of the variants of the respective learner. All of them are saved in a file with the same name as the object and extension "`.Rdata`". These are

[4] On my standard desktop computer it takes approximately 3 days.

the files that are available at the book Web site, so in case you have not run all these experiments, you can download them into your computer, and load them into R using the `load()` function (indicating the name of the respective file as argument):

```
> load("knn.Rdata")
> load("svm.Rdata")
> load("randomForest.Rdata")
```

The results of all variants of a learner are contained in the respective object. For instance, if you want to see which were the best SVM variants, you may issue

```
> rankSystems(svm, max = T)

$ALL
$ALL$accuracy
  system    score
1 svm.v2 86.17021
2 svm.v3 86.17021
3 svm.v5 86.17021
4 svm.v6 86.17021
5 svm.v9 86.17021
```

The function `rankSystems()` takes an object of class `compExp` and obtains the best performing variants for each of the statistics that were estimated in the experimental process. By default, the function assumes that "best" means smaller values. In case of statistics that are to be maximized, like accuracy, we can use the parameter `max` as we did above.[5]

In order to have an overall perspective of all trials, we can join the three objects:

```
> all.trials <- join(svm, knn, randomForest, by = "variants")
```

With the resulting `compExp` object we can check the best overall score of our trials:

```
> rankSystems(all.trials, top = 10, max = T)

$ALL
$ALL$accuracy
            system    score
1           knn.v2 88.29787
2           knn.v3 87.23404
3  randomForest.v4 87.23404
4  randomForest.v6 87.23404
```

[5] In case we measure several statistics, some that are to be minimized and others maximized, the parameter `max` accepts a vector of Boolean values.

```
5           svm.v2 86.17021
6           svm.v3 86.17021
7           svm.v5 86.17021
8           svm.v6 86.17021
9           svm.v9 86.17021
10         svm.v23 86.17021
```

The top score is obtained by a variant of the k-nearest neighbor method. Let us check its characteristics:

```
> getVariant("knn.v2", all.trials)

Learner::  "genericModel"

Parameter values
         learner  =  "knn"
         k  =  5
         norm  =  TRUE
         fs.meth  =  list("rf", 30)
```

This variant uses 30 genes filtered by a random forest, five neighbors and normalization of the gene expression values. It is also interesting to observe that among the top ten scores, only the last one ("svm.v23") does not use the 30 genes filtered with a random forest. This tenth best model uses all genes resulting from the ANOVA filtering. However, we should note that the accuracy scores among these top ten models are rather similar. In effect, given that we have 94 test cases, the accuracy of the best model means that it made 11 mistakes, while the model on the tenth position makes 13 errors.

It may be interesting to know which errors were made by the models, for instance, the best model. Confusion matrices (see page 120) provide this type of information. To obtain a confusion matrix we need to know what the actual predictions of the models are. Our user-defined function does not output the predicted classes, only the resulting accuracy. As a result, the `compExp` objects do not have this information. In case we need this sort of extra information, on top of the evaluation statistics measured on each iteration of the experimental process, we need to make the user-defined functions return it back to the experimental comparison routines. These are prepared to receive and store this extra information, as we have seen in Chapter 4. Let us imagine that we want to know the predictions of the best model on each iteration of the LOOCV process. The following code allows us to obtain such information. The code focuses on the best model but it should be easily adaptable to any other model.

```
> bestknn.loocv <- function(form,train,test,...) {
+   require(Biobase,quietly=T)
+   require(randomForest,quietly=T)
+   cat('=')
+   tgt <- as.character(form[[2]])
```

```
+   tgtCol <- which(colnames(train)==tgt)
+   # Anova filtering
+   f <- Anova(train[,tgt],p=0.01)
+   ff <- filterfun(f)
+   genes <- genefilter(t(train[,-tgtCol]),ff)
+   genes <- names(genes)[genes]
+   train <- train[,c(tgt,genes)]
+   test <- test[,c(tgt,genes)]
+   tgtCol <- 1
+   # Random Forest filtering
+   rf <- randomForest(form,train,importance=T)
+   imp <- importance(rf)
+   imp <- imp[,ncol(imp)-1]
+   rf.genes <- names(imp)[order(imp,decreasing=T)[1:30]]
+   train <- train[,c(tgt,rf.genes)]
+   test <- test[,c(tgt,rf.genes)]
+   # knn prediction
+   ps <- kNN(form,train,test,norm=T,
+             norm.stats=list(rowMedians(t(as.matrix(train[,-tgtCol]))),
+                             rowIQRs(t(as.matrix(train[,-tgtCol])))),
+             k=5,...)
+   structure(c(accuracy=ifelse(ps == resp(form,test),100,0)),
+             itInfo=list(ps)
+            )
+ }
> resTop <- loocv(learner('bestknn.loocv',pars=list()),
+                 dataset(Mut~.,dt),
+                 loocvSettings(seed=1234,verbose=F),
+                 itsInfo=T)
```

The `bestknn.loocv()` function is essentially a specialization of the function `genericModel()` we have seen before, but focused on 5-nearest neighbors with random forest filtering and normalization using medians and IQRs. Most of the code is the same as in the `genericModel()` function, the only exception being the result that is returned. This new function, instead of returning the vector with the accuracy of the model, returns a structure. We have seen before that structures are R objects with appended attributes. The `structure()` function allows us to create such "enriched" objects by attaching to them a set of attributes. In the case of our user-defined functions, if we want to return some extra information to the `loocv()` function, we should do it on an attribute named `itInfo`. In the function above we are using this attribute to return the actual predictions of the model. The `loocv()` function stores this information for each iteration of the experimental process. In order for this storage to take place, we need to call the `loocv()` function with the optional parameter `itsInfo=T`. This ensures that whatever is returned as an attribute with name `itInfo` by the user-defined function, it will be collected in a list and returned as an attribute named `itsInfo` of the result of the `loocv()`

function. In the end, we can inspect this information and in this case see what were the actual predictions of the best model on each iteration.

We can check the content of the attribute containing the wanted information as follows (we are only showing the first 4 predictions):

```
> attr(resTop, "itsInfo")[1:4]

[[1]]
[1] BCR/ABL
Levels: ALL1/AF4 BCR/ABL E2A/PBX1 NEG

[[2]]
[1] NEG
Levels: ALL1/AF4 BCR/ABL E2A/PBX1 NEG

[[3]]
[1] BCR/ABL
Levels: ALL1/AF4 BCR/ABL E2A/PBX1 NEG

[[4]]
[1] ALL1/AF4
Levels: ALL1/AF4 BCR/ABL E2A/PBX1 NEG
```

The function `attr()` allows us to obtain the value of any attribute of an R object. As you see, the `itsInfo` attribute contains the predictions of each iteration of the LOOCV process. Using this information together with the true value of the class of the dataset, we can obtain the confusion matrix:

```
> preds <- unlist(attr(resTop, "itsInfo"))
> table(preds, dt$Mut)

preds      ALL1/AF4 BCR/ABL E2A/PBX1 NEG
  ALL1/AF4       10       0        0   0
  BCR/ABL         0      33        0   4
  E2A/PBX1        0       0        3   1
  NEG             0       4        2  37
```

The confusion matrix can be used to inspect the type of errors that the model makes. For instance, we can observe that the model correctly predicts all cases with the `ALL1/AF4` mutation. Moreover, we can also observe that most of the errors of the model consist of predicting the class `NEG` for a case with some mutation. Nevertheless, the reverse also happens with five samples with no mutation, incorrectly predicted as having some abnormality.

5.5 Summary

The primary goal of this chapter was to introduce the reader to an important range of applications of data mining that receives a lot of attention from the R community: bioinformatics. In this context, we explored some of the tools of the project Bioconductor, which provides a large set of R packages specialized for this type of applications. As a concrete example, we addressed a bioinformatics predictive task: to forecast the type of genetic mutation associated with samples of patients with B-cell acute lymphoblastic leukemia. Several classification models were obtained based on information concerning the expression levels on a set of genes resulting from microarray experiments. In terms of data mining concepts, this chapter focused on the following main topics:

- Feature selection methods for problems with a very large number of predictors
- Classification methods
- Random forests
- k-Nearest neighbors
- SVMs
- Ensembles using different subsets of predictors
- Leave-one-out cross-validation experiments

With respect to R, we have learned a few new techniques, namely,

- How to handle microarray data
- Using ANOVA tests to compare means across groups of data
- Using random forests to select variables in classification problems
- Clustering the variables of a problem
- Obtaining ensembles with models learned using different predictors
- Obtaining k-nearest neighors models
- Estimating the accuracy of models using leave-one-out cross-validation.