CATS: Feature selection and Classification in R

Annika Jacobsen & Maurits Dijkstra

Using R

This tutorial will help you write an R script that can perform feature selection and classification in R (version 3.0 or newer).

First some R basics: if you have written an R script “myscript.R”, you can use the *source* function to run your script:

```
> source('myscript.R')
```

To get more information on a function you can use the help facility. Below you can see an example of how to get more information on *source*.

```
> help(source)
```

Note that the script you need to hand in for CATS, should contain functions to keep to code readable. In addition to the functions defined in R libraries you are able to create your own. Below you can find a basic function written in R, which sums two variables *a* and *b* and returns the sum *c*.

```
> myfunction <- function(a, b){
    c <- a + b
    return(c)
}
> myFunction(32, 10)
42
```

This tutorial requires the 'caret' R package, which is not available on the practical machines. To install it, type the following command in an R shell:

```
> install.packages('caret', dependencies=TRUE)
```

If you've never used R on the practical machines before it may ask you to select a mirror site and/or to create a private repository where it will install third-party packages. Selecting the default choices for both questions will work fine in this instance. The 'caret' package provides all the functionality related to feature selection, parameter tuning and cross-validation, but it depends on other R packages to build the actual classification models. Thus, if you are planning to use a specific machine learning method it may be useful to install the required packages beforehand as well.

```
> install.packages('gbm', dependencies=TRUE)
```

Now you are able to verify whether your R installation is working correctly.

```
> library('caret')
```

Loading the data

The required data files are provided on the blackboard page for this course. You should download them and place them somewhere in your home directory. For convenient access to the files you should then instruct R to change the working directory to the directory where you placed the files.

```
> setwd('/your/path/to/CATS/here/')
```

The first task is to load the preprocessed aCGH data and the associated clinical labels. If you inspect these files in an editor you are able to see that they are in tab separated format. In order to read these files you should use the *read.delim* function provided by R. If you are unsure about what arguments *read.delim* accepts have a look at the built-in manual:

```
>help(read.delim)
read.delim(file, header = TRUE, sep = "\t", quote = "\"", dec = ".",
fill = TRUE, comment.char = "", ...)
```

When you have successfully loaded the data take a look at it to confirm that it is in a format comparable to the output below. This will also allow you to examine the format of the various columns.

```
> head(mydata[,1:7])
  Chromosome  Start     End Nclone Array.129 Array.34 Array.67
1          1   2927   43870      3         0        0        0
2          1  85022  216735      4         0        0        0
3          1 370546  372295      4         0        0        0
4          1 471671  786483      5         0        0        0
5          1 792533  907406     13         0        0        0
6          1 912799 1266212     96         0        0        0
```

We can't directly feed the data to our machine learning library in the current format, because some of the columns store miscellaneous information about the different regions instead of feature data. Additionally, we need to transpose the matrix so the rows contain samples and

the columns contain features. You can use the three following functions *as.data.frame*, *t* and *subset*. Again, you can use the built-in help facility to get more information about them.

You should now apply the same techniques to load the data file containing the clinical outcome for each sample. This file should require substantially less processing to transform it in the right format.

Processing the data

After loading the data you should obtain two data frames, one containing the feature data per sample and one containing the associated clinical outcome. However, our machine library expects a single data frame containing both feature and outcome data. Luckily, merging the two data frames is easily accomplished through the *merge* function defined by the R standard library. This function, when given two data frames, will combine rows based on a shared value in a certain column. A small example illustrating the use of *merge* is given below.

```
> A
  firstname lastname
1    Albert Einstein
2    Edsger Dijkstra
3   Charles   Darwin
> B
  lastname            field
1 Dijkstra Computer Science
2   Darwin          Biology
3 Einstein          Physics
> merge(A, B, by="lastname")
  lastname firstname            field
1   Darwin   Charles          Biology
2 Dijkstra    Edsger Computer Science
3 Einstein    Albert          Physics
```

Note that the order of the corresponding rows is different in *A* and *B*, so just concatenating the columns of *A* and *B* would assign the wrong fields in *B* to the first and last names in *A*. Given our newfound knowledge of *merge* we are now able to combine our two data frames.

```
> Combo <- merge(Labels, Train, by="row.names")
> row.names(Combo) <- Combo$Row.names
> Combo$Row.names <- NULL
```

The last two lines are to work around a quirk in the implementation of *merge* and understanding them is not important for the assignment. If you store the sample name ("Array.xxx") in a different column then you should replace "row.names" with the column

name you chose. You should now inspect *Combo* to verify that the merge was performed correctly:

```
> head(Combo[,1:10])
            Subgroup 1 2 3 4  5  6  7 8  9
Array.10       HER2+ 0 0 0 0  0  0  0 0  0
Array.100        HR+ 0 0 0 0  0  0  0 0  0
Array.101        HR+ 0 0 1 1  1  1  1 1  1
Array.102 Triple Neg 0 0 0 0 -1 -1 -1 0 -1
Array.104 Triple Neg 0 0 0 0  0  0  0 0  0
Array.105      HER2+ 0 0 0 0  0  0  0 0 -1
```

Feature Selection

To get a better performing classifier and to reduce the risk of overfitting we want to decrease the number of features by only using those that distinguish between the clinical outcomes best. The importance of each feature on determining each of the three classes can be estimated with the *filterVarImp* function. Use the help facility to get more information on the function. The *filterVarImp* function takes the matrix of the prediction data and a vector of outcomes as arguments.

```
> head(rocVarImp)
      HER2.       HR. Triple.Neg
1 0.4848090 0.4887153  0.4887153
2 0.4707031 0.5047743  0.5047743
3 0.5273438 0.5477431  0.5477431
4 0.5869141 0.6037326  0.6037326
5 0.5410156 0.5711806  0.5711806
6 0.5000000 0.5434028  0.5434028
```

Using this function will give us give us the importance per class, but we are interested in the total importance over all the classes.

The *apply* function found in the R standard library can be used to combine the importance per class into a single importance value for every feature. We will look at how *apply* works in a small example. First we define a simple data set consisting of 5 rows and 2 columns.

```
> A
     [,1] [,2]
[1,]    8   13
[2,]    5    2
[3,]    6    4
```

```
[4,]    9    7
[5,]   11   15
```

Calling *apply* on *A* will call a user-specified function on each of A's rows, collecting the return values in a new array which is then returned. We will now look at its behavior in-depth when provided with a number of standard R functions such as *mean* and *sum*.

```
> apply(A, 1, mean)
[1] 10.5  3.5  5.0  8.0 13.0
> apply(A, 1, sum)
[1] 21  7 10 16 26
```

The second parameter controls which part of the array is provided to the chosen function per step. In this case we're interested in calculating the function over the rows of the matrix, so we give '1' as argument. If we want to apply the function to every element in the matrix rather than every row, that's also possible:

```
> increment <- function(x) {
     return(x+1)
}
> apply(A, 2, increment)
     [,1] [,2]
[1,]    9   14
[2,]    6    3
[3,]    7    5
[4,]   10    8
[5,]   12   16
```

This last example returns the matrix resulting from incrementing every element in *A* by 1.

Finally we sort the rows in the order of decreasing importance by using the function *order*. We need to do this because we only want to keep the *n* most important features. We then filter the feature columns of the training set based on this selection. Note that choosing the optimal value of *n* is not trivial and you should observe how different values impact the performance of your classifier. **NOTE: be very careful when experimenting with feature selection, as feature selection is also a form of training and should thus also be validated**

Training the classifier

The time has come to use the remaining part of the data set after feature selection to build a classifier. For this simple example we used a simple classification method called k-Nearest Neighbours (knn), but you may want to try out other more sophisticated classifiers such as

general boosted models, random forests or neural networks. The *caret* package also makes it very easy to try out and validate the performance of different models and/or parameter combinations. By using the functions *trainControl* and *train* you should now be able to train a classifier.

trainControl creates an object describing the training protocol. You can for instance choose to go with 10-fold cross-validation with 10 repeats. Note that this choice is arbitrary and a training protocol may exist which offers better performance.

train does the actual training of the classification model. Once this function completes it will show you the optimal values determined during the parameter tuning process as well as the estimated performance. In the case of the k-Nearest Neighbour classifier there is only one parameter, namely *k*. *train* returns an object (*modelFit* in this example) which you should inspect to see more details about the training process:

```
> modelFit
k-Nearest Neighbors

100 samples
 20 predictors
  3 classes: 'HER2+', 'HR+', 'Triple Neg'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 10 times)

Summary of sample sizes: 91, 91, 89, 90, 90, 90, ...

Resampling results across tuning parameters:

  k  Accuracy  Kappa  Accuracy SD  Kappa SD
  5  0.633     0.443  0.13         0.198
  7  0.604     0.398  0.127        0.194
  9  0.611     0.408  0.126        0.193

Accuracy was used to select the optimal model using  the largest
value.
The final value used for the model was k = 5.
```

If you want to implement your own training loop have a look at the *dataPartition* function. This function will partition your dataset while preserving the class stratification of the samples. As always, have a look at the built-in help pages or the caret documentation at http://caret.r-forge.r-project.org/ if you want to see a more advanced example or want more information about the various parameters that *train* and *trainControl* accept.

Making predictions

The object returned by *train* also contains a copy of the classification model with the best prediction performance during training. This can be used to make class predictions of samples where the clinical outcome is not known. First we have to read in the validation samples in the same way as we read in the training samples. Then we select the same *n* features from the validation samples, since these are the features our classifier expects. Finally, we can then use the *predict* function from R to generate a vector of class predictions.

Generating output

All that is now left is to write the class predictions to a text file so they can be submitted to the contest. It is important to note that contests are very strict about adhering to a specific output format and will disqualify an entry if it fails to do so! This includes 'invisible' details, such as whether spaces or tabs are used to separate the fields. You can find an example of the required format on blackboard (“example_lable.txt”). To create the correct output format first use the *data.frame* function to create a data frame from the sample names and the predictions and then use the *write.table* function to write it to disk. To see how these two functions work we have created a small example. First we define two vectors, then we create a data frame from the two vectors by using the *data.frame* function.

```
> x
 [1] 4 6 3 6 9
> y
 [1] 3 5 0 1 7

> Output <- data.frame(x, y)
> Output
   x y
1  4 3
2  6 5
3  3 0
4  6 1
5  9 7
```

The *write.table* function can be used to get the correct output format. Take a look at the example output file to give the correct arguments to the function. **NOTE: the example below does *not* write the correct format, you need to modify it so it does.**

```
> write.table(Output, "myPredictions.txt", sep="_",
              row.names=FALSE)
```

```
"x"_"y"
4_3
6_5
3_0
6_1
9_7
```

Where to go from here

The information above should be sufficient to get you started, but the quality of the predictions by the very basic classifier which has been described may be suboptimal. Possible areas of improvement you could investigate include (but are not limited to):

- The use of a more sophisticated classification method.
- Improvements to the feature selection process.
- The adoption of a more rigorous validation protocol.

Independent of which area you choose to improve, it is a good idea to read the documentation for the *caret* package at http://caret.r-forge.r-project.org/. In addition, you could take a look at other R packages for feature selection and classification (e.g. *CORElearn*, *RWeka*).