Nathalie Vialaneix
Année 2018/2019

M1 in Economics and Economics and Statistics
Applied multivariate Analysis - Big data analytics
Worksheet 4 - Parallel computing

This worksheet's aim is to introduce parallel programming in R by presenting some standard packages allowing this kind of computing. The first two exercises will present the package **foreach** , usable for standard parallel computing, and the last exercise will present the package **rmr2** for Map-Reduce parallel programming. This worksheet uses data set from the R package **mlbench**:

```
library(mlbench)
```

Warning: This worksheet has been compiled on Linux (xUbuntu 18.04 LTS, satanic edition). As some of the command lines are OS-dependant, the results for other OS are not given even though the command lines are provided. However, the results should be very similar.

Exercice 1 Parallel bagging with CART

This first exercise illustrates the use of the R package **foreach** for simple parallel programming with a `for` loop. This package is loaded with

```
library(foreach)
```

and requires the loading of a package supporting parallel computing which is **doMC** for Unix-like OS and **doParallel** for Windows© OS. The number of cores in your computer can be detected with the function `detectCores` from the package **parallel**. This package is automatically loaded when loading one of the two packages for parallel computing: **doMC** for Unix-like OS:

```
library(doMC)

## Loading required package:  iterators
## Loading required package:  parallel
```

or **doParallel** for Windows© OS:

```
## Not Run
library(doParallel)
```

1. Using the function `detectCores`, find out how many cores are available on your computer.
2. Unless a proper instruction is called to set R to use parallel computation, the `foreach` loop **will not be run in parallel**. To see how many cores are going to be used by the `foreach` function, you can use the function `getDoParWorkers` (package **foreach**). What is the initial number of cores used by the function `foreach`? This value can be increased to a number larger than 1 with an instruction that is specific to your OS. For Unix-like OS (using the **doMC** package), the number of cores is set with the function `registerDoMC`. On Windows© OS, the number of cores is set with the function `registerDoParallel` (package **doParallel**). Use these functions to set the number of cores to their default values: what is the default number of cores? Use these functions to set the number of cores to a larger value (that must not exceed the total number of available cores minus one).
3. The data used in this exercise will be the data used in the first exercise of the previous worksheet:

```r
data(Sonar)
```

Split the data into a training and a test sets with respective sizes 100 and 108.

4. Use a `for` loop to train $B = 1000$ classification trees from bootstrap samples obtained from the training data set. For each tree, find the predicted classes for the training and test sets (store these values in two matrices with 1000 columns and a number of rows equal to, respectively, 100 and 108). What is the computational time for this loop? Using these results and a majority vote scheme, find the bagging estimates for the class and compute the training and test misclassification rates. Use the package **rpart** for this question:

   ```r
   library(rpart)
   ```

   Remark: Here a simple misclassification rate for the training data set is required although a OOB misclassification rate would have been more suited.

5. Answer the same questions with a `foreach` loop (use `.combine=cbind` to obtain column-wise results containing the prediction for the training data set in the first rows and prediction for the test data set in the last rows. Comment the difference in computation times between the two approaches. The ellapsed time is approximately 2 times smaller with the `foreach` loop than with the `for` loop whereas the user time is comparable (but has been divided on several cores). The performances are similar to the ones obtained with the `for` function.

Exercice 2 Parallel random forest

This exercise uses the data `Ionosphere`, already presented in the first exercise of worksheet number 2:

```r
data(Ionosphere)
```

This exercise's aim is to present how to use random forest in parallel and to compare its execution with a sequencial approach. Hence, it requires the loading of the package **randomForest**:

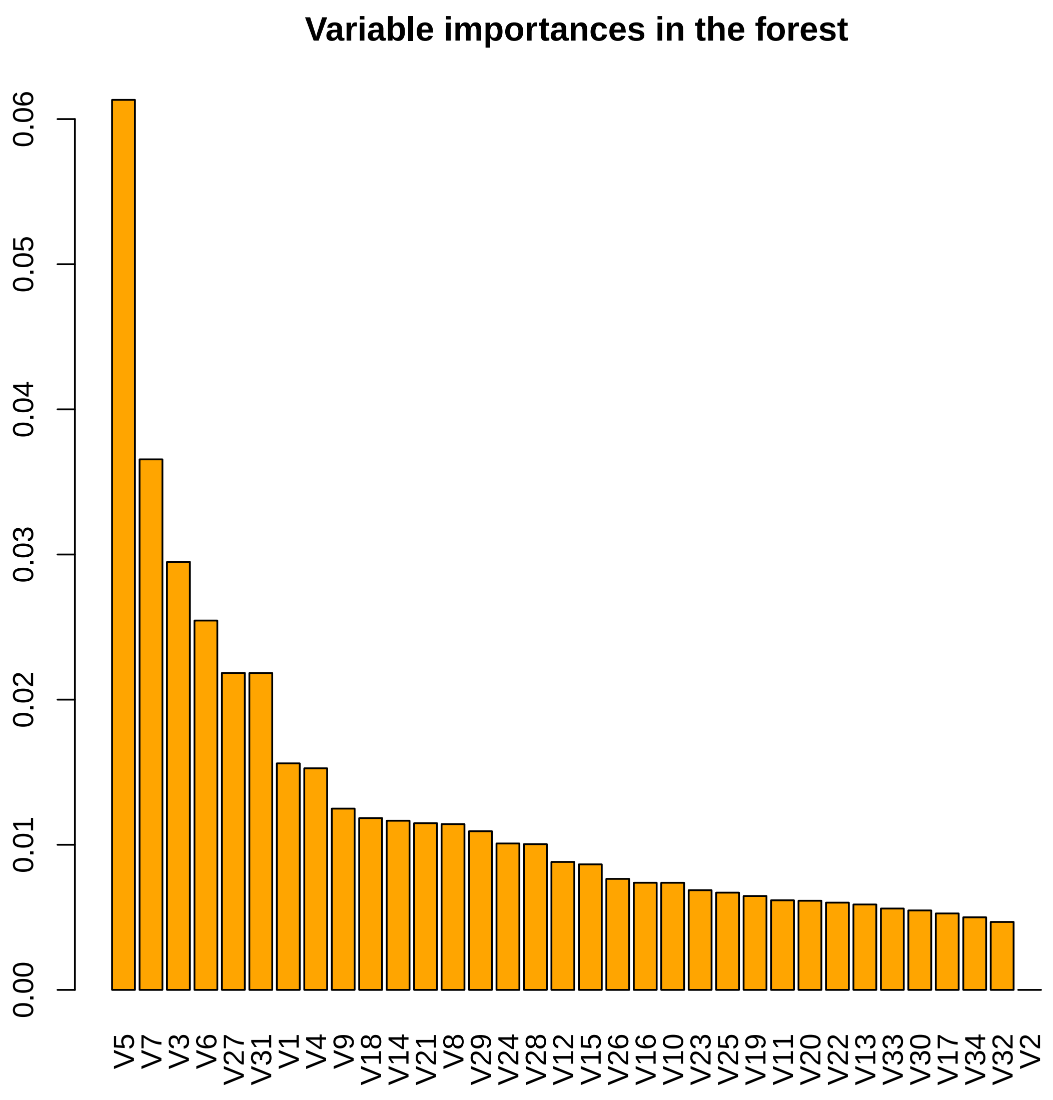
```r
library(randomForest)

## randomForest 4.6-14
## Type rfNews() to see new features/changes/bug fixes.
```

1. Briefly describe the data and their purpose and split them into a training and a test sets with respective sizes 201 and 150.

2. Train a random forest on the training data set, with 1000 trees in the forest and using the arguments `xtest` and `ytest` to obtain the test error. Also, ask the computation of the importance of the variables and keep the forest (option `keep.forest`). What is the computational time required to train such a forest?

3. Use a `foreach` loop with the option `.combine=combine` to train 10 times a random forest with 100 trees on several cores. This will result in a random forest with 1000 trees as in the previous question (use the same options for the test error, the importance and for keeping the forest). Compare the computational times required with this approach.

4. Use the combined forest obtained in the previous question to compute the OOB and test misclassification rates and the importances of the predictors. Make a graphic for the importances. Comment on those results.



Exercice 3 First Map/Reduce job

In this exercise, we will be using a larger data set that describes social environment of several thousands Americans. The purpose is to predict whether or not these people have an income that exceeds $50K/yr. We will be using these data to test our first Map/Reduce (MR) job that will simply consist in calculating the average age for people who have an incone smaller/lager than $50K/yr. The data set is downloaded from the UCI ML repository[1] with the following command lines:

```
dir.create("dataset")

## Warning in dir.create("dataset"):  'dataset' already exists

download.file("http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data",
              "dataset/adult_train.csv")
download.file("http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.test",
              "dataset/adult_test.csv")
```

These lines first create a directory called `dataset` and download two files (respectively the training and test data sets) in two CSV files called `adult_train.csv` and `adult_test.csv`.

[1] https://archive.ics.uci.edu/ml

Disclaimer: Usually, if the task is performed in a true Hadoop environment with multiple computers, the data are not loaded in R but accessed through HDFS via the package **rhdfs**. This prevents to work only with data sets that can be loaded in RAM. Here, we will be using the package **rmr2** "as if" Hadoop was installed but with a local backend, which means that the data set will be manipulated with R directly rather than with HDFS. This setting is set using the following command lines (that load the package **rmr2** and set a local backend).

```
library(rmr2)

## Warning:  S3 methods 'gorder.default', 'gorder.factor', 'gorder.data.frame',
'gorder.matrix', 'gorder.raw' were declared in NAMESPACE but not found
## Please review your hadoop settings.  See help(hadoop.settings)

rmr.options(backend="local")

## NULL
```

Finally, to help you write your first MR job, you can have a look at this tutorial https://github.com/RevolutionAnalytics/rmr2/blob/master/docs/tutorial.md.

1. Using the function `mapreduce` with the data set path as an input and the input format properly set using the function `make.input.format`, load the two files into the (virtual) HDFS. Try to print the result of this command: what do you obtain?
   Hint: Inspect the file with a simple text editor before performing this step to set up properly the options `quote`, `sep` and `skip` (see `help(read.table)` to check the meaning of these options).

2. Using the function `from.dfs` allows R to read the result of the `mapreduce` function. However, in general, this is not recommended (because the data may be much too large to be loaded in RAM). The result is expressed as a list with two components: `$key` and `$val`. Use this function to print the keys and combine it with `head` and `dim` to obtain the dimension of the data sets and to print the first lines.

3. Describe the map job that would produce a list of pairs (key, value) to obtain the average age for the two income statuses. Implement it within the function `mapreduce` in properly setting the option `map` by using the function `keyval` and use it on the training data set. Then, use the functions `from.dfs` and `head` on the output of the function `mapreduce` to print the first keys and first values obtained from the MR job.

4. Describe the reduce job that would produce the average age for the two income statuses. Modify your previous function to include this reduce job and run it on the training data set. This is done using the arguement `reduce` and identifying what the output keys and values should be. Finally, use the function `mapreduce` to print the final result.

5. Use an MR job on the training and test data sets to compute the number of males and females per country. Use a combination of the country and the gender as the indexing key (with the function `paste` and separator _ for instance).

6. Use the functions `strplit` and `unlist` on the keys to obtain, for each value, its country and its gender. Make a data frame with the first two columns being the country and the gender respectively and the last column being the corresponding frequency. Using the function `xtabs`, build a contingency table between country and gender from this matrix and perform a χ^2 test for the training and test sets.