TD Parcours **Data Science** - Machine learning

Groupe 1 - AVIT // BEELMEON // SOW

décembre 10, 2020

Contents

Dans ce travail pratique destiné à nous familiariser aux méthodes de Machine Learning, nous allons présenter les Random Forests, une technique qui a connu un gros essort depuis sa mise en place à la fois pour des problématiques de regression et de classification.

Après avoir brièvement étudié les données à disposition, nous allons travailler avec trois algorithmes de RandomForest de caret : rf, Rborist et RRF, et comparer leurs performances respectives.

Importation des données

Nous commençons par importer la librairie **caret** qui contient le jeu de donnée **BloodBrain** sur lequel nous allons étudier l'efficacité des différents algorithmes de RandomForest, ainsi que des fonctions et méthodes utiles. Nous importons de plus la librairie tidyverse, qui est une collection de plusieurs librairies dont readr, dplyr, tidyr, ggplot2 pour pouvoir traiter et visualiser avec la philosophie "tidy" notre jeu de données.

```
library(caret)
library(tidyverse)
```

L'importation du jeu de données BloodBrain à partir de caret se fait facilement avec la fonction utils::data().

```
data(BloodBrain)
```

BloodBrain contient un dataframe **bbbDescr** de 208 observations et 134 variables, ainsi qu'un vecteur réponse **logBBB** de taille 208. Il est issu des travaux de Mente et Lombardo des laboratoires de R&D du

géant américain Pfizer. Ces dernières dans leur article paru en 2005 dans le Journal of Computer-Aided Molecular Design ont développé des modèles qui prédisent le log ratio de la concentration de composés chimiques dans le cerveau et dans le sang.
On étudie ici le rôle de l'interface sang/encéphale, d'où le nom du jeu de données : Blood Brain Barrier (=BBB). Pour chaque composé chimique sont calculés 3 ensembles de descripteurs moléculaires : **MOE 2D**, **rule-of-five** et **Charge Polar Surface Area**. Le dataframe bbbDescr est donc constitué par les variables explicatives X (il s'agit des features) et le vecteur réponse logBBB est notre variable expliquée (notre target), que l'on souhaite tenter de prédire. Nous sommes donc dans le cadre d'un apprentissage supervisé.

Afin de mieux appréhender cette problématique et avant toute tentative de prédiction avec les RandomForest, nous allons étudier nos features et target.
Nous avons décidé par souci de praticité de lier le vecteur réponse à notre dataframe de variables explicatives. La majorité des jeu de données pour le machine learning adoptant a priori cette configuration.

```
mydata <- cbind(bbbDescr, data.frame(logBBB))
```

Étude des variables explicatives : Features Nous pouvons tout d'abord regarder la structure du jeu de données.

```
# str(mydata[,-135]) # -135 pour ne pas considérer la colonne target (logBBB)
#Comme il y a 134 variables, on n'affiche pas celles-ci dans le Rmd pour des raisons de clareté.
```

L'ensemble des variables paraissent continues (integer/numeric).

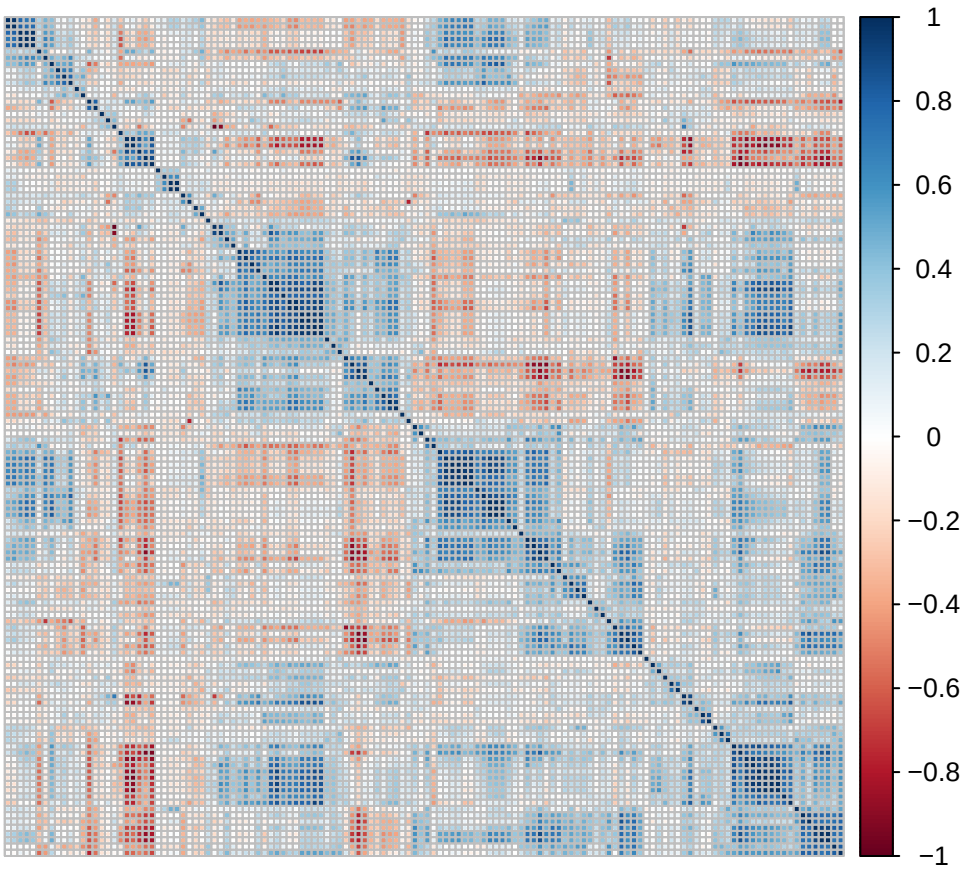
Nous cherchons aussi la présence de valeurs manquantes qui peuvent perturber les prédictions de nos modèles.

```
table(is.na(mydata[,-135]))
```

```
##
## FALSE
## 27872
```

Nous constatons l'absence de données manquantes, le jeu de données est parfaitement complet.

Nous pouvons aussi étudier les potentielles corrélations entre variables. L'étude des corrélations entre variables explicatives est très importante dans une démarche de modélisation et de prédiction. Si une multitude de variables sont corrélées cela induit une redondance dans l'information, ce qui perturbe certains algorithmes prédictifs de base très utilisés comme les régressions.
Nous utilisons ici la fonction corrplot du package du même nom.



Le heatmap ci-dessus révèle la présence d'une multitude de corrélations (positives et négatives) dans nos variables explicatives, ce qui était attendu compte tenu du nombre important de variables et du contexte biologique. Selon les méthodes que l'on souhaite utiliser, certains auteurs recommandent d'éliminer les variables beaucoup trop corrélées.
Dans notre cas, les Random Forest sont une famille de méthodes qui gèrent très efficacement la redondance de l'information, et nous pouvons nous permettre de garder toutes les variables (ce sont les algorithmes de Random Forest eux-mêmes qui vont gérer la redondance).

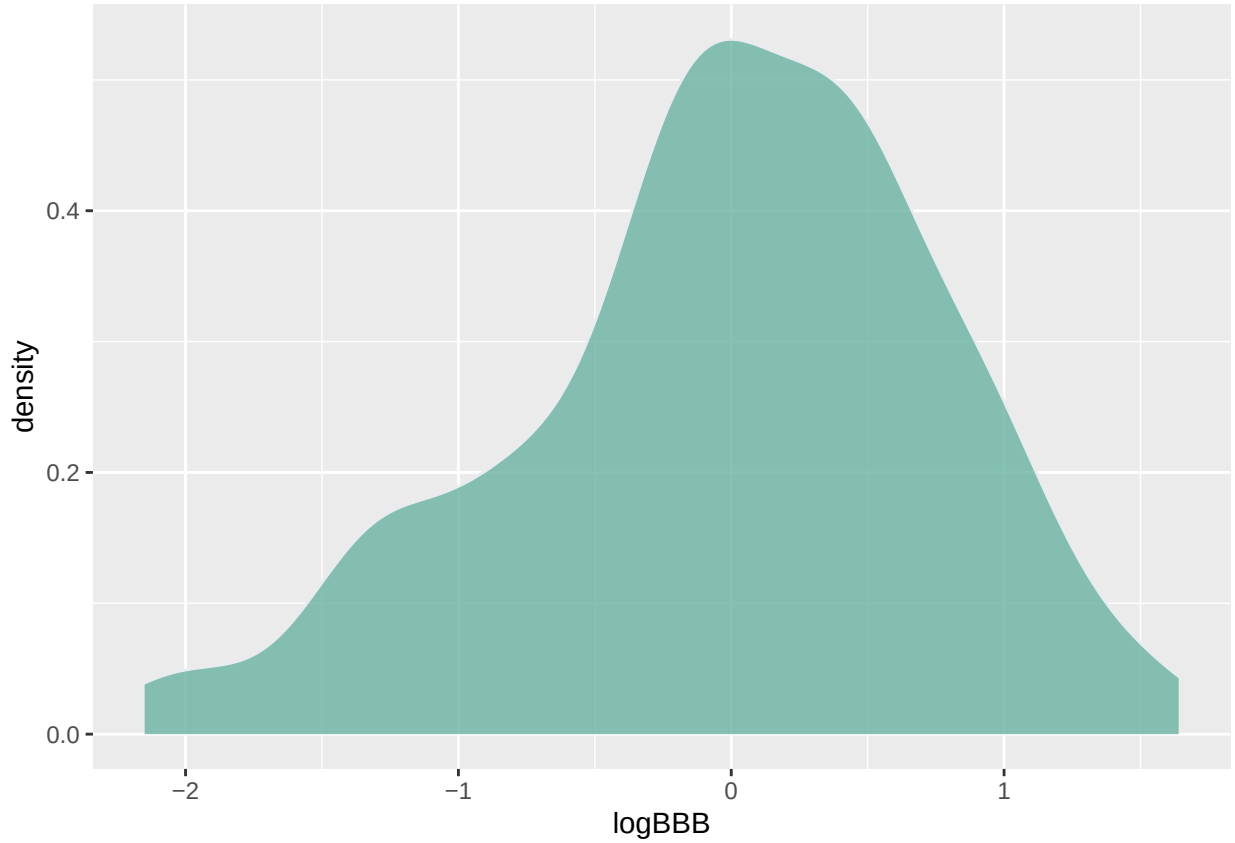
Étude de la variable réponse : Target

Nous pouvons d'abord sortir les première statistiques qui lui sont associées.

```
summary(mydata$logBBB)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.15000 -0.42250  0.02000 -0.01889  0.53000  1.64000
```

. . . Et la représenter graphiquement.



Nous pouvons observer que notre variable réponse suit une loi de densité proche de celle d'une loi normale.

```
table(is.na(mydata$logBBB))
```

```
##
## FALSE
##   208
```

Là encore, il n'y a pas de données manquantes.

Nous souhaitons donc effectuer une prédiction d'une valeur quantitative à partir de données quantitatives. Le jeu de données est complet (il n'y a pas de valeurs manquantes). Il y a beaucoup de variables prédictives (presque autant que d'observations !), et il y a une forte redondance de l'information entre celles-ci. L'utilisation de Random Forest pour notre prédiction de logBBB paraît tout à fait justifiée.

Famille de méthodes : Random Forest

Après notre brève présentation du jeu de données, nous allons maintenant appliquer les Random Forest à celui-ci, et étudier leur efficacité.

Description de la famille de méthodes

Les méthodes Random Forest découlent directement des arbres de décision.

Le principe d'un arbre de décision est plutôt simple :
Celui-ci est constitué de plusieurs noeuds. A chaque noeud, on sélectionne une des variables d'entrées, et selon la valeur de celle-ci, l'individu est envoyé vers un des noeuds suivants. A la fin de la ramification de l'arbre, une classe (s'il s'agit d'une classification) ou une valeur (s'il s'agit d'une régression) est associée à chaque "feuille" de l'arbre.
L'arbre est créé par apprentissage, c'est à dire qu'un jeu de données d'entraînement permet de construire un arbre efficace sur celui-ci, en prévision d'une utilisation sur un jeu de données à prédire.
Contrairement à la plupart des autres méthodes de Machine learning, les arbres de décision ont la vertue de ne pas être des boîtes noires : on sait exactement quels sont les critères utilisés pour la classification/régression.
Malheureusement, en pratique, les arbres de décision sont peu efficaces car peu robustes, fortement soumis au surapprentissage (c'est à dire qu'un arbre pourra donner l'impression de bien fonctionner sur le jeu de données d'entraînement, mais aura des performances médiocres sur le jeu de données à prédire).

Les Random Forest sont un développement visant à contrer ce problèmes :
Au lieu d'utiliser un seul arbre de décision, on génère de très nombreux arbres de décision, et la prédiction finale est une moyenne des prédictions de tous ces arbres. Pour que les différents arbres de la Forêt soient différents les un des autres (s'ils étaient tous semblables, cela reviendrait à ne pas faire de Random Forest mais juste un arbre de décision), on utilise un bootstrap (rééchantillonnage avec remise) différent pour chacun des arbres, et des contraintes sur les prédicteurs à utiliser sont ajoutées aléatoirement.
Cette stratégie est payante, et les Random Forest sont des méthodes de prédiction très efficaces et utilisées.
On peut en revanche regretter qu'elles perdent du coup l'avantage des arbres de décision de ne pas être des boîtes noires.

Dans notre approche de modélisation et de prédiction, nous utilisons comme indiqué précédemment le package caret. Ce dernier propose une multitude de familles de méthodes, et de nombreuses sous-méthodes dans chacune de ces familles de méthodes.
Nous allons comparer ici trois Random Forest proposées par caret : rf, Rborist et RRFglobal. Ceux-ci ne sont que des exemples : caret met à disposition un total de 238 modèles de RandomForest différents.
La différence entre ces nombreuses variations de Random Forest semble résider dans les paramètres optimisés lors de la création de la forêt.

Outils dans R

Mise en oeuvre sous R

On sépare le jeu de données en un jeu de données d'entraînement et un jeu de données de validation. Le premier va nous servir à entraîner nos modèles et générer les forêts aléatoires, le second nous permettra d'évaluer l'efficacité de nos régressions.

```
set.seed(007)
trainIndex <- createDataPartition(y=mydata$logBBB, times=1, p=0.8, list=F)
# Train
Train_set <- mydata[trainIndex,]
Test_set <- mydata[-trainIndex,]
```

Après avoir importé les librairies complémentaires de chacune des méthodes, on génère les 3 Random Forest différentes. Les modèles sont entraînés sur les données centrées et réduites, afin d'éviter que l'importance de certaines des variables ne soit surestimée par rapport à d'autres.
De plus, on ajoute system.time afin de pouvoir comparer les temps d'exécution des 3 fonctions.

```
library(randomForest)
library(Rborist)
library(RRF)
```

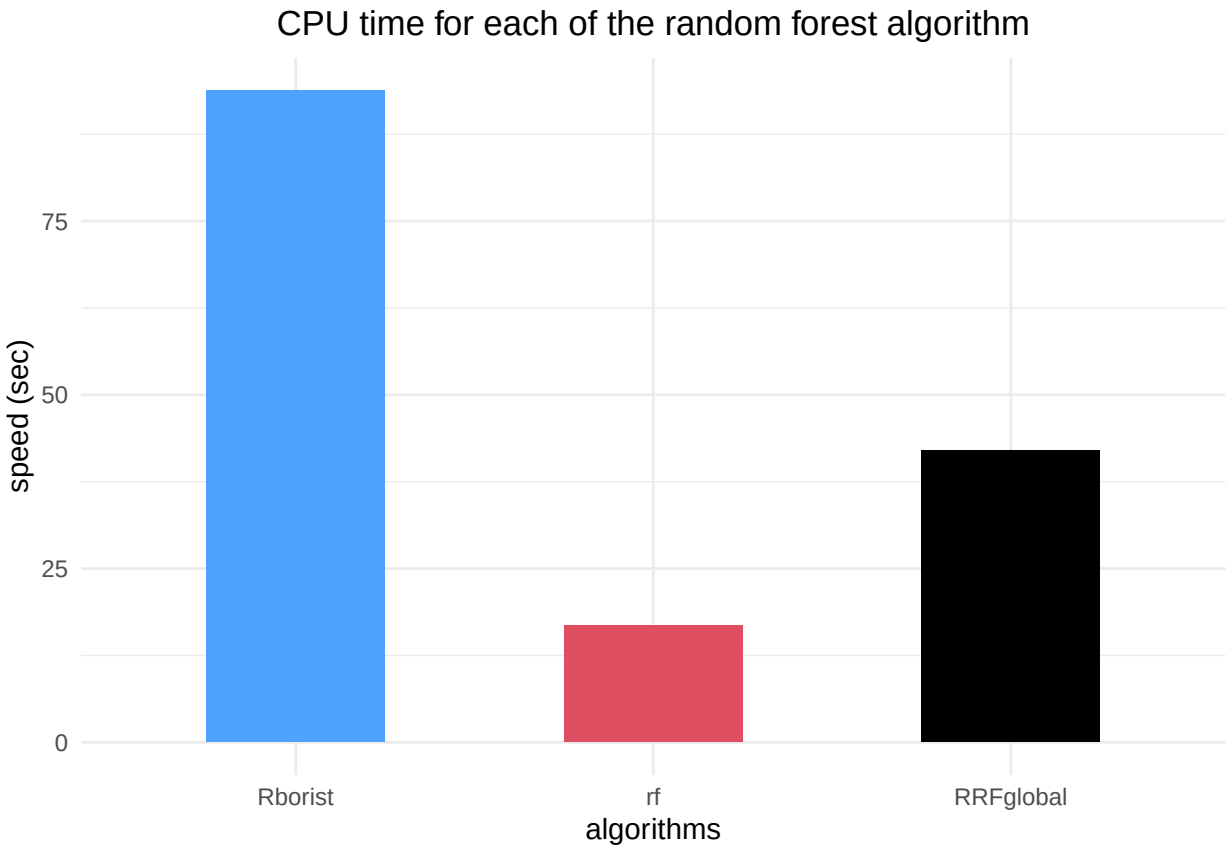
```r
set.seed(007)
fit_Control_rcv <- trainControl(method = "repeatedcv",number=5) #pour chacun des modèles l'entraînement

speed_rf <- system.time(
mod_rf <- train(
             logBBB ~.,
             Train_set,
             method="rf",
             trControl = fit_Control_rcv,
             importance=T,
             preProcess = c("center","scale") # Centrage réduction.
          )
)

speed_Rborist <- system.time(
mod_Rborist <- train(
             logBBB ~.,
             Train_set,
             method="Rborist",
             trControl = fit_Control_rcv,
             importance=T,
             preProcess = c("center","scale")
          )
)

speed_RRFglobal <- system.time(
mod_RRFglobal<- train(
             logBBB ~.,
             Train_set,
             method="RRFglobal",
             trControl = fit_Control_rcv,
             importance=T,
             preProcess = c("center","scale")
          )
)
```

On compare les temps de calcul pour chacune des méthodes, qui diffèrent fortement. Cependant, nous avons pu constater que sur nos différentes machines, les vitesses relatives n'étaient pas les mêmes : sur deux de nos ordinateurs, Rborist était (de loin) la méthode la plus lente, tandis qu'elle était la plus rapide sur le troisième.

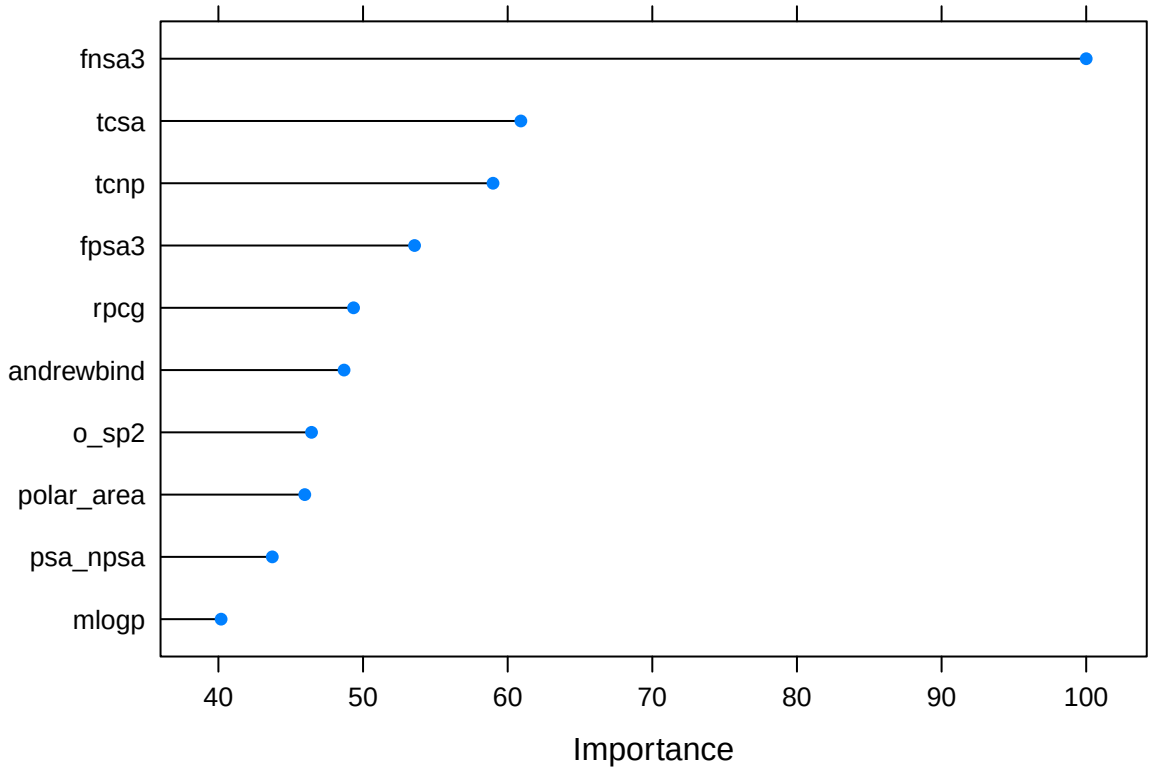


Nous pouvons également observer quelles sont les variables utilisées majoritairement par chacun des trois modèles prédictifs. On constate que ce ne sont pas les mêmes : cela découle de la forte corrélation entre les variables précédemment évoquée (d'ailleurs, si on relançait les algorithmes, ce ne serait sans doute pas les mêmes variables pour un algorithme donné, il y a une part d'aléa).

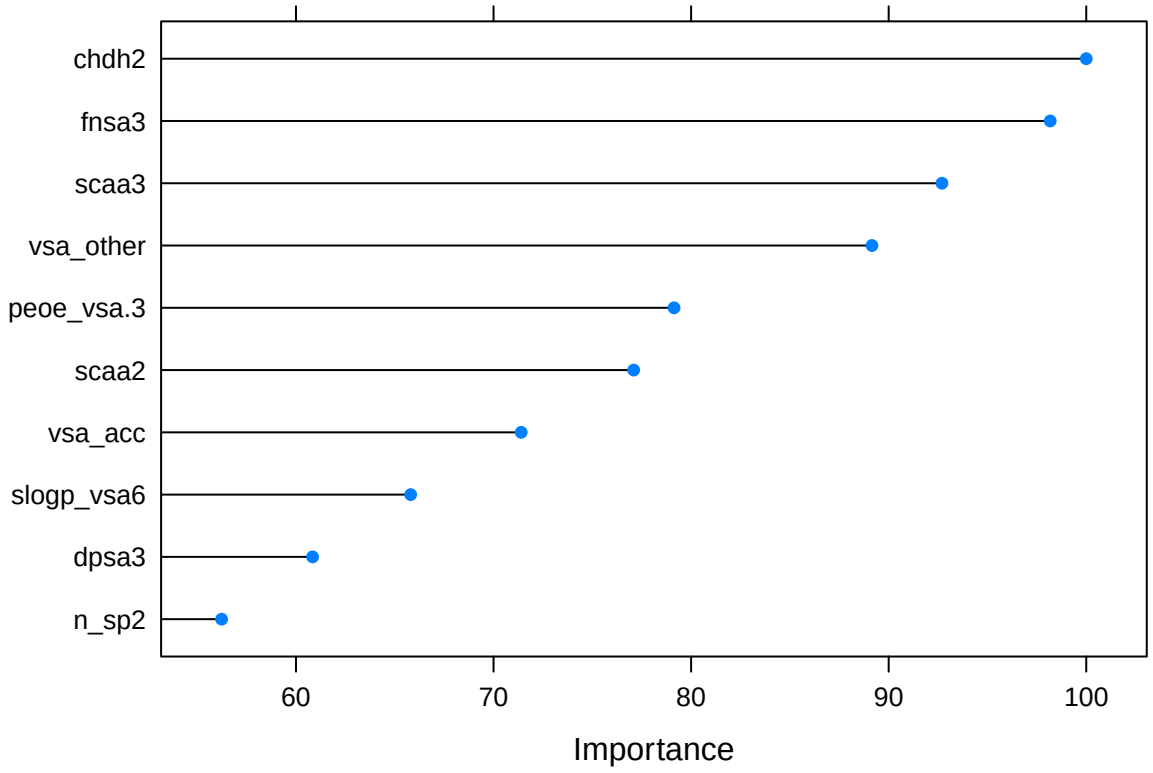
L'intérêt des méthodes de forêts aléatoires réside aussi dans la notion d'importance de variables. En effet, on peut, en regardant quelles sont les variables utilisées dans la décision de chaque arbre de la forêt aléatoire, déterminer quelles sont les variables le splus utilisées par la forêt.

Nous pouvons constater avec le classement (top 10) des variables les plus importantes que mêmes si les 3 méthodes ont des variables les plus importantes communes, il y a tout de même des différences. On peut aussi souligner que rf accorde une importance prépondérante à une variable en particulier (fnsa3).
Comme évoqué précédemment, ces différences sont dues au Feature selection, qui a notamment des effets sur l'ordre d'importance des variables explicatives.

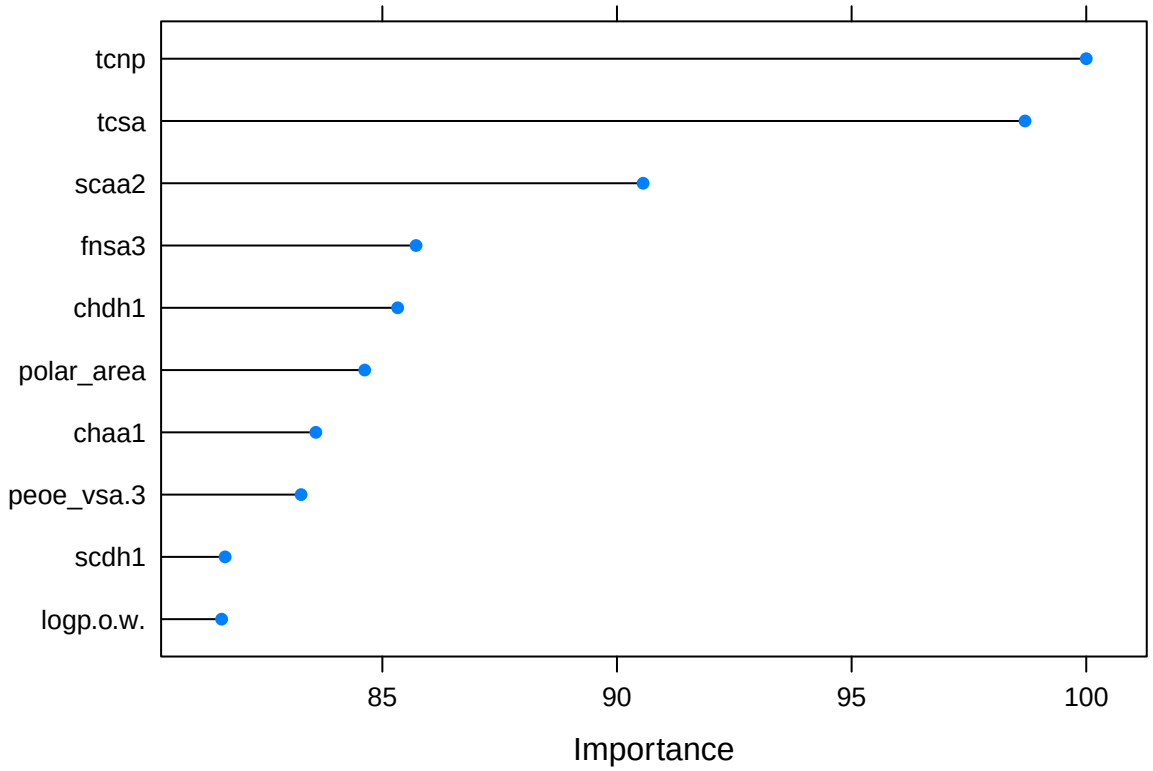
```
plot(varImp(mod_rf),10)
```




```
plot(varImp(mod_Rborist), 10)
```




```
plot(varImp(mod_RRFglobal), 10)
```



On constate que les 3 modèles ont des RMSE (écarts quadratiques moyens) semblables sur les jeux d'entraînements, RRFglobal étant légèrement moins bon (RMSE plus élevé).
Il est néanmoins bon de rappeler que la capacité d'un modèle à coller aux données d'entraînement n'est pas forcément représentative de sa capacité prédictive (problématique du surapprentissage).

```
max(mod_rf$results$RMSE)
```

```
## [1] 0.5249069
```

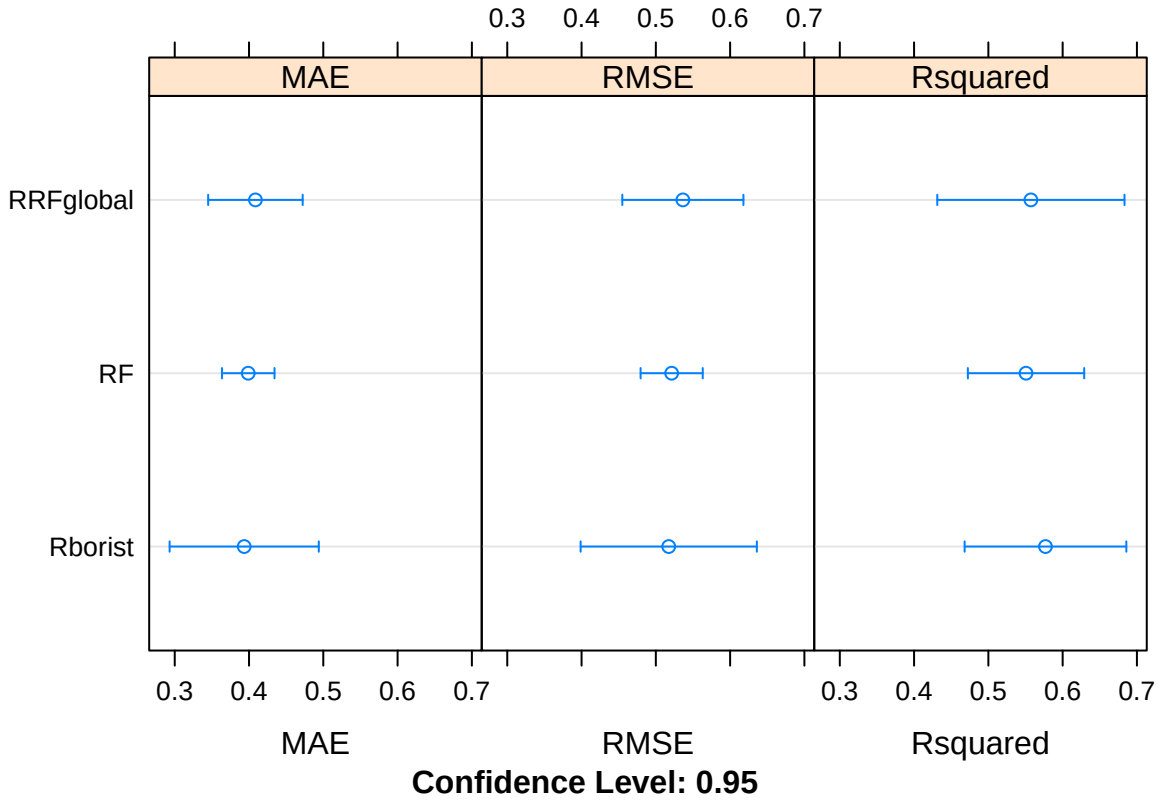
```
max(mod_Rborist$results$RMSE)
```

```
## [1] 0.5340995
```

```
max(mod_RRFglobal$results$RMSE)
```

```
## [1] 0.616283
```

Les critères de performance présentés ici sont le MAE, le RMSE et le Rsquared (qui sont trois mesures de l'erreur).
Caret fournit en effet une fonction caret::resamples qui permet de collecter et d'analyser les résultats d'entraînement de modèles. La visualisation peut par la suite se faire facilement avec un dotplot également disponible dans le package.



Nous constatons avec les trois critères présentés ci-dessus (le Mean Absolute Error - le Root Mean Square Error et le R^2) que les trois méthodes ont des performances très semblables. Ces trois critères sont les plus utilisés dans les problématiques de regression.

Au delà de ces critères fournis directement par caret::train(), il est nécessaire d'étudier la performance de nos modèles sur des données autres que celles sur lesquelles ils ont été entrainés. Ce qui est un écueil à éviter dans une démarche cohérente et rigoureuse de machine learning.

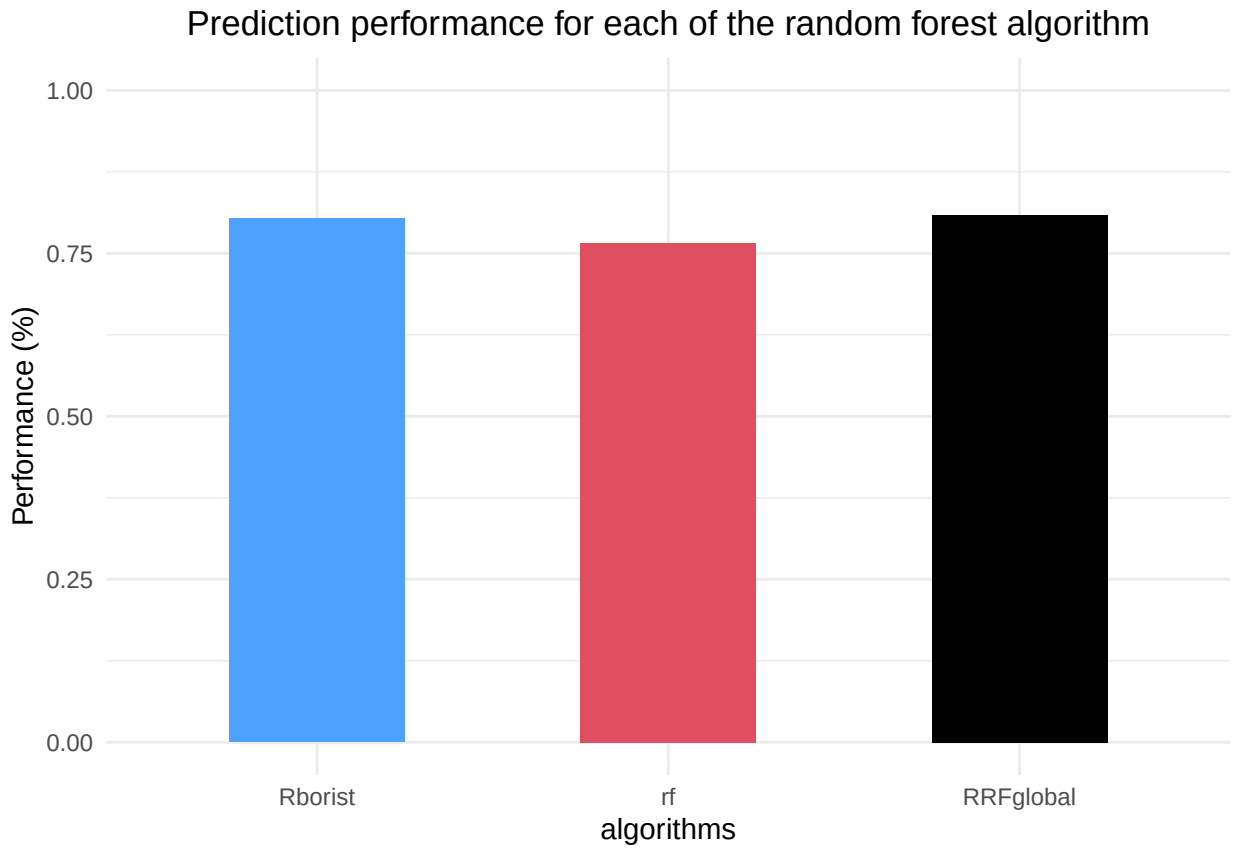
Afin de réaliser l'étude de la performance nous utilisons la fonction caret::predict() en lui fournissant le modèle sur lequel il va se baser et le Test_set (données de test).

```
pred_rf <- predict(mod_rf, newdata = Test_set)
pred_Rborist <- predict(mod_Rborist, newdata = Test_set)
pred_RFglobal <- predict(mod_RRFglobal, newdata = Test_set)
```

En régression, contrairement à une situation de classification, il n'est pas possible d'utiliser une matrice de confusion pour évaluer la performance de prédiction de nos modèles de Random Forest. Nous nous proposons ici d'utiliser une technique assez équivalente où l'on calcule la corrélation entre les valeurs prédites de logBBB et le logBBB dans le Test_set.

Plus la corrélation entre ces valeurs est forte, plus notre modèle sera performant.

Nous réprésentons graphiquement la performance de prédiction de chacune de nos méthodes.



Il ne semble pas y avoir de différence significative en termes de performance de prédiction entre nos trois méthodes.

Nos trois modèles fournissent des prédictions sur le jeu de test correlées à 75-80% avec la variable à prédire (il faudrait voir, en fonction du contexte, s'il s'agit d'un taux de corrélation acceptable ou non).

En guise de rappel, nous pouvons distinguer les trois méthodes que nous avons utilisées par les hyper-paramètres optimisés.

Pour rf caret n'optimise que mtry, pour Rborist predFixed et minNode sont optimisés et pour RRFglobal c'est mtry et coefReg qui sont optimisés.

```
mod_rf$modelInfo$parameters[]
```

```
##   parameter   class                         label
## 1      mtry numeric #Randomly Selected Predictors
```

```
mod_Rborist$modelInfo$parameters[]
```

```
##   parameter   class                         label
## 1 predFixed numeric #Randomly Selected Predictors
## 2   minNode numeric             Minimal Node Size
```

```
mod_RRFglobal$modelInfo$parameters[]
```

```
##   parameter   class                         label
## 1      mtry numeric #Randomly Selected Predictors
## 2   coefReg numeric          Regularization Value
```

Pour finir :
- Même si elles peuvent être considérées comme des boîtes noires, les méthodes de Random Forest ont connu une ascension très importante depuis leur initiation par Breiman et al. (2001) pour répondre d'abord aux insuffisances des arbres de décison puis à celles du bagging (la transition entre arbres de décision et random forests).
- il en existe plusieurs déclinaisons disponibles dans caret. Nous en avons présenté ici 3 : rf, Rborist et RRFglobal.
- au delà de leur efficacité démontrée en classification comme en regression elles nous ont paru intéressantes, permettant notamment de bien voir quelles étaient les variables les plus utilisées par le modèle de prédiction.

Session Info

```
## R version 4.0.2 (2020-06-22)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 18363)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=French_France.1252  LC_CTYPE=French_France.1252
## [3] LC_MONETARY=French_France.1252 LC_NUMERIC=C
## [5] LC_TIME=French_France.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] RRF_1.9.1           Rborist_0.2-3       randomForest_4.6-14
##  [4] forcats_0.5.0       stringr_1.4.0       dplyr_1.0.2
##  [7] purrr_0.3.4         readr_1.3.1         tidyr_1.1.2
## [10] tibble_3.0.3        tidyverse_1.3.0     caret_6.0-86
## [13] ggplot2_3.3.2       lattice_0.20-41
##
## loaded via a namespace (and not attached):
##  [1] httr_1.4.2           jsonlite_1.7.1       splines_4.0.2
##  [4] foreach_1.5.0        prodlim_2019.11.13   modelr_0.1.8
##  [7] assertthat_0.2.1     stats4_4.0.2         blob_1.2.1
## [10] cellranger_1.1.0     yaml_2.2.1           corrplot_0.84
## [13] ipred_0.9-9          pillar_1.4.6         backports_1.1.10
## [16] glue_1.4.2           pROC_1.16.2          digest_0.6.25
## [19] rvest_0.3.6          colorspace_1.4-1     recipes_0.1.13
## [22] htmltools_0.5.0      Matrix_1.2-18        plyr_1.8.6
## [25] timeDate_3043.102    pkgconfig_2.0.3      broom_0.7.0
## [28] haven_2.3.1          scales_1.1.1         gower_0.2.2
## [31] lava_1.6.8           farver_2.0.3         generics_0.0.2
## [34] ellipsis_0.3.1       withr_2.2.0          nnet_7.3-14
## [37] cli_2.0.2            survival_3.1-12      magrittr_1.5
## [40] crayon_1.3.4         readxl_1.3.1         evaluate_0.14
```

```
## [43] fs_1.5.0             fansi_0.4.1          nlme_3.1-148
## [46] MASS_7.3-53          xml2_1.3.2           class_7.3-17
## [49] tools_4.0.2          data.table_1.13.0    hms_0.5.3
## [52] lifecycle_0.2.0      munsell_0.5.0        reprex_0.3.0
## [55] compiler_4.0.2       rlang_0.4.7          grid_4.0.2
## [58] iterators_1.0.12     rstudioapi_0.11      labeling_0.3
## [61] rmarkdown_2.5        gtable_0.3.0         ModelMetrics_1.2.2.2
## [64] codetools_0.2-16     DBI_1.1.0            reshape2_1.4.4
## [67] R6_2.4.1             lubridate_1.7.9      knitr_1.30
## [70] stringi_1.5.3        Rcpp_1.0.5           vctrs_0.3.4
## [73] rpart_4.1-15         dbplyr_1.4.4         tidyselect_1.1.0
## [76] xfun_0.19
```

Références

1. R Core Team (2020). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.
2. Max Kuhn (2020). caret: Classification and Regression Training. R package version 6.0-86. https://CRAN.R-project.org/package=caret
3. https://topepo.github.io/caret/