# Objectif

Ce tutoriel la trame pour implementer un Random Forest et boosting sous R. La construction d’un arbre de décision est d’abord décrite, nous mesurons les performances en prédiction, puis nous voyons ce que peuvent apporter les méthodes ensemblistes. Différents aspects de ces méthodes seront mis en lumière : l’importance des variables, l’influence du paramétrage, l’impact des caractéristiques des arbres sous-jacents, etc.

Le langage R est utilisé avec les packages rpart, adabag et randomforest. Evaluer l’influence du paramétrage sur les performances sera notamment très intéressant.

# Données

Nous utilisons les données « [Image Segmentation Data Set](https://archive.ics.uci.edu/ml/datasets/Image%2BSegmentation) » du dépôt UCI Machine Learning. Elles décrivent 7 types d’images d’extérieur à partir des paramètres qui ont été extraites. Les observations ont été subdivisées en échantillons d’apprentissage (30 exemples par type d’image, soit 210 individus) et de test (300 par type).

Plutôt que de manipuler deux fichiers, nous d’abord lire les fichiers d’apprentissage et test

et les réunir dans un seul data frame

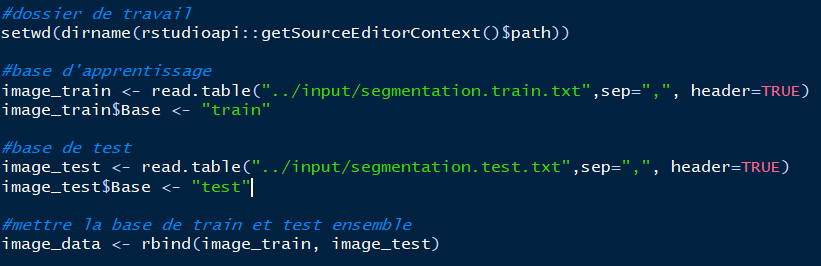
Une colonne supplémentaire « sample » est ajouté, indiquant leur d’appartenance (apprentissage [train] ou test [test]). REGION.TYPE est la variable cibl

# Analyse avec R

## Importation et préparation des données

Nous importons les fichiers « **segmentation.train.txt** » et « **segmentation.train.txt** »

avec les paramètres adéquats.



La commande **summary()** permet d’avoir un aperçu des caractéristiques des données et de s’assurer de leur intégrité.

REGION.TYPE REGION.CENTROID.COL REGION.CENTROID.ROW REGION.PIXEL.COUNT SHORT.LINE.DENSITY.5

BRICKFACE:330 Min. : 1.0 Min. : 11.0 Min. :9 Min. :0.00000

CEMENT :330 1st Qu.: 62.0 1st Qu.: 81.0 1st Qu.:9 1st Qu.:0.00000

FOLIAGE :330 Median :121.0 Median :122.0 Median :9 Median :0.00000

GRASS :330 Mean :124.9 Mean :123.4 Mean :9 Mean :0.01433

PATH :330 3rd Qu.:189.0 3rd Qu.:172.0 3rd Qu.:9 3rd Qu.:0.00000

SKY :330 Max. :254.0 Max. :251.0 Max. :9 Max. :0.33333

WINDOW :330

SHORT.LINE.DENSITY.2 VEDGE.MEAN VEDGE.SD HEDGE.MEAN HEDGE.SD

Min. :0.000000 Min. : 0.0000 Min. : 0.0000 Min. : 0.0000 Min. : 0.0000

1st Qu.:0.000000 1st Qu.: 0.7222 1st Qu.: 0.3556 1st Qu.: 0.7778 1st Qu.: 0.4216

Median :0.000000 Median : 1.2222 Median : 0.8333 Median : 1.4444 Median : 0.9630

Mean :0.004714 Mean : 1.8939 Mean : 5.7093 Mean : 2.4247 Mean : 8.2437

3rd Qu.:0.000000 3rd Qu.: 2.1667 3rd Qu.: 1.8064 3rd Qu.: 2.5556 3rd Qu.: 2.1833

Max. :0.222222 Max. :29.2222 Max. :991.7184 Max. :44.7222 Max. :1386.3292

INTENSITY.MEAN RAWRED.MEAN RAWBLUE.MEAN RAWGREEN.MEAN EXRED.MEAN EXBLUE.MEAN

Min. : 0.000 Min. : 0.00 Min. : 0.000 Min. : 0.000 Min. :-49.667 Min. :-12.444

1st Qu.: 7.296 1st Qu.: 7.00 1st Qu.: 9.556 1st Qu.: 6.028 1st Qu.:-18.556 1st Qu.: 4.139

Median : 21.593 Median : 19.56 Median : 27.667 Median : 20.333 Median :-10.889 Median : 19.667

Mean : 37.052 Mean : 32.82 Mean : 44.188 Mean : 34.146 Mean :-12.691 Mean : 21.409

3rd Qu.: 53.213 3rd Qu.: 47.33 3rd Qu.: 64.889 3rd Qu.: 46.500 3rd Qu.: -4.222 3rd Qu.: 35.778

Max. :143.444 Max. :137.11 Max. :150.889 Max. :142.556 Max. : 9.889 Max. : 82.000

EXGREEN.MEAN VALUE.MEAN SATURATION.MEAN HUE.MEAN Base

Min. :-33.889 Min. : 0.00 Min. :0.0000 Min. :-3.044 train: 210

1st Qu.:-16.778 1st Qu.: 11.56 1st Qu.:0.2842 1st Qu.:-2.188 test :2100

Median :-10.889 Median : 28.67 Median :0.3748 Median :-2.051

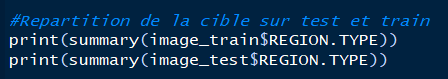
Mean : -8.718 Mean : 45.14 Mean :0.4269 Mean :-1.363

3rd Qu.: -3.222 3rd Qu.: 64.89 3rd Qu.:0.5401 3rd Qu.:-1.562

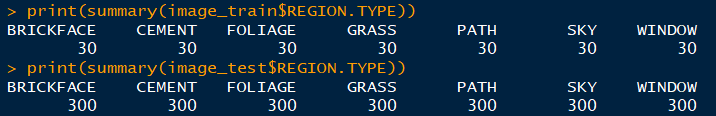
Max. : 24.667 Max. :150.89 Max. :1.0000 Max. : 2.912

On notera entres autres que les classes sont parfaitement équilibrées (REGION\_TYPE), nous disposons de 210 observations en apprentissage et 2100 en test.

Nous partitionnons les données en échantillons d’apprentissage et de test en utilisant la colonne « sample ».



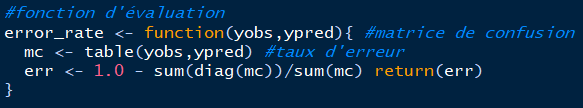
Nous avons respectivement les distributions de classes suivantes :



Nous sommes parés pour lancer les analyses.

## Fonction d’évaluation des performances

Le taux d’erreur sera utilisé pour évaluer la qualité de la prédiction. Nous écrivons une fois pour toutes une fonction à cet effet. Elle prend en entrée la variable cible observée et la prédiction d’un modèle.

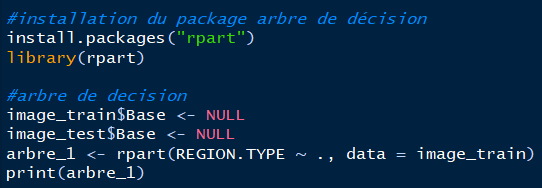


## Arbre de décision

Nous utilisons la librairie « [rpart](https://cran.r-project.org/web/packages/rpart/index.html) » pour construire les arbres de décision, parce qu’elle est très populaire, et surtout parce qu’elle est sous-jacente aux méthodes ensemblistes que nous verrons par la suite dans R. Les instructions de paramétrage sont réutilisables. Nous disposerons ainsi d’une vue cohérente des résultats.

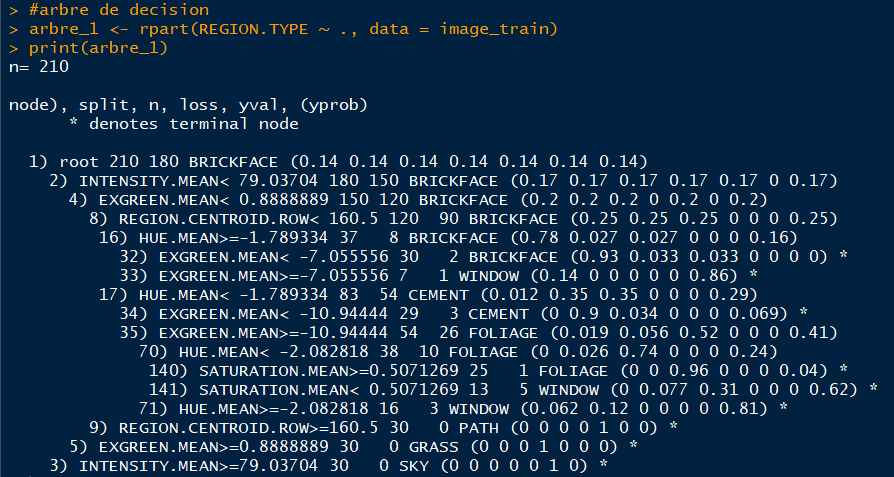
### 3.3.1 Arbre avec les paramètres par défaut

Nous construisons une première version des arbres avec les paramètres par défaut.



La commande ***install.packages*,** il faut l’utiliser juste une fois, pour installer la librairie

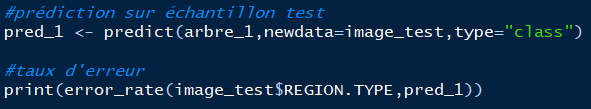
La commande ***rpart*** permet d’entraienr un arbre . Nous obtenons un arbre avec 9 feuilles :



Lisons attentivement l’arbre :

* + - le symbole « \* » indique les nœuds terminaux (les feuilles) de l’arborescence ;
    - il y a 9 feuilles, donc 9 règles ;
    - quelques variables seulement parmi les 19 disponibles ont été utilisées, certaines plusieurs fois (ex. EXGREEN\_MEAN) ;
    - détaillons la lecture du sommet n°34 : il comporte 29 observations (**n**), avec 3 contre- exemples (**loss**), la conclusion est CEMENT (**yval**), ce qui correspond à ≈ 90% ((29- 3)/29 = 0.8966) (**yprob**) des observations présentes sur le sommet.

Nous calculons la prédiction du modèle sur l’échantillon test, puis nous la confrontons avec la variable cible observée.

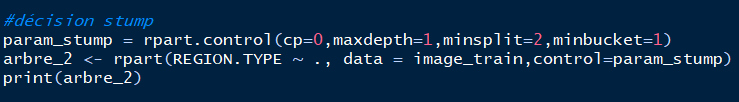


Le taux d’erreur est **12.86%**.

### Decision stump

Les « decision stump »3 sont des arbres limités à un niveau c.-à-d à une seule segmentation, avec deux feuilles lorsqu’ils (les arbres) sont binaires. C’est un non-sens dans le cadre de la construction d’un arbre prédictif unique, surtout lorsque le nombre de modalités K de la variable cible est strictement supérieur à 2 (K > 2). L’approche se justifie en revanche dans le contexte d’une technique ensembliste de type « boosting ». En effet, cette dernière réduit la composante « biais » de l’erreur. De fait, « Boosting de decision stump » est une méthode de référence reconnue dans la littérature. On se rend compte en effet que le modèle global correspond à un classifieur linéaire. Cette caractéristique apparaît clairement lorsque les prédicteurs sont tous binaires.

Dans cette section, il s’agit avant tout d’identifier les paramètres qui permettent de modifier le comportement de la procédure rpart().



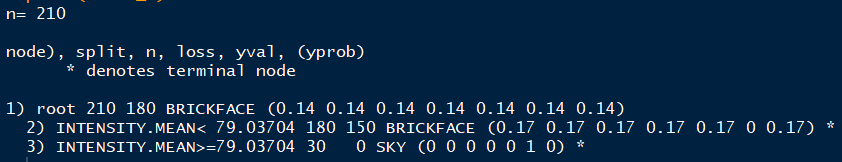
L’option control permet de spécifier les paramètres de l’algorithme :

* + - « cp » intervient en pré-élagage lors de la construction de l’arbre, une segmentation est acceptée uniquement si la réduction relative de l’indice de Gini est supérieure à

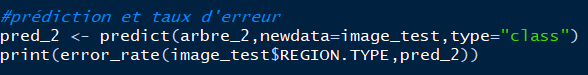
« cp ». En le mettant à zéro, nous désactivons son action.

* + - « minsplit » indique l’effectif minimum d’un somment pour tenter une segmentation.
    - « minbucket » est l’effectif d’admissibilité. Dans notre cas, nous acceptons les feuilles qui comportent au moins 1 individu. C’est le minimum que l’on puisse faire.
    - « maxdepth » indique la profondeur maximal de l’arbre, sachant que la racine est au niveau 0. Avec « maxdepth = 1 », nous définissons bien un « decision stump ».

Nous obtenons l’arbre suivant :



Qui s’avère catastrophique en prédiction…

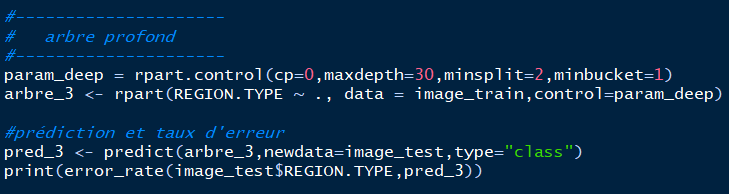


… avec un taux d’erreur de **71.43%**. C’était prévisible, seule la classe SKY est correctement reconnue.

### Arbre plus profond

Voyons ce qu’il en est maintenant si nous produisons un arbre très profond avec une profondeur maximale de 30 niveaux (qui correspond d’ailleurs à la valeur par défaut). Il ne faut pas que les paramètres de réduction de la pureté ou d’effectifs interfèrent. Nous les mettons au minimum.

Revoici la nouvelle séquence des traitements :



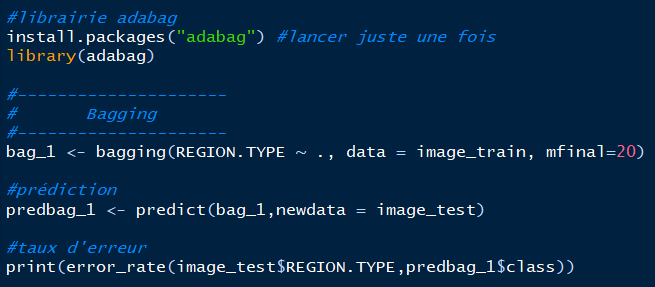
Nous obtenons un arbre avec 21 feuilles (essayez l’affichage de l’arbre), avec un taux d’erreur de **10.42%**. L’arbre profond est meilleur que le premier. Il n’y a pas de sur- apprentissage. Ce n’est pas très habituel. Cela laisse à penser que les classes sont peu bruitées. La limitation de la qualité de l’apprentissage provient du nombre d’observations dans l’échantillon d’apprentissage, facteur particulièrement déterminant s’agissant des performances des arbres de décision.

## Bagging

Nous utilisons le package « [adabag](https://cran.r-project.org/web/packages/adabag/adabag.pdf) » pour implémenter la méthode bagging sur nos données.

### Bagging avec 20 arbres (paramètres par défaut)

Nous commençons par un bagging de 20 arbres avec les paramètres par défaut.



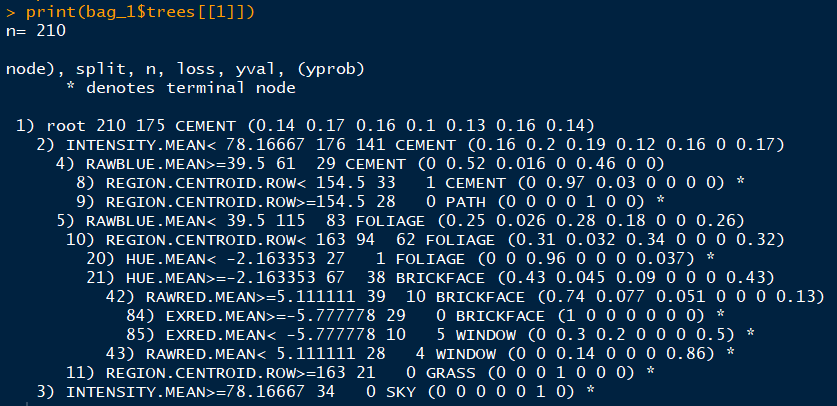
Nous effectuons la prédiction sur l’échantillon test. Nous noterons que l’objet prédiction est un type complexe comprenant les classes prédites par le modèle (**$class**). Nous l’opposons aux classes observées, le taux d’erreur est de **6.9%**.

### Accès aux arbres individuels

La méthode génère une collection d’arbres, ils sont accessibles avec le champ **$trees** de l’objet résultat. Accédons au premier arbre généré.



Nous avons :



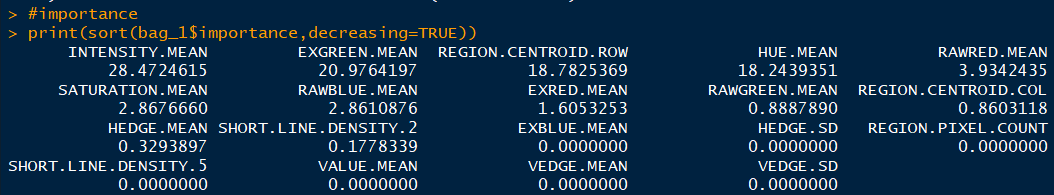
L’échantillon bootstrap est composé de 210 observations (n = 210 dans la sortie de R ci- dessus). Mais, de l’échantillon d’apprentissage initial, certains se répètent, d’autres sont absents. C’est pour cette raison que nous obtenons un arbre différent de celui élaboré avec rpart() sur l’échantillon d’apprentissage, pourtant basé sur les mêmes paramètres par défaut

### Importance des variables

Il est impossible d’analyser la multitude d’arbres pour évaluer l’influence des variables prédictives dans la modélisation. L’outil « importance des variables » permet de pallier cet inconvénient. Nous les affichons par ordre décroissant d’importance ici :



Nous avons :

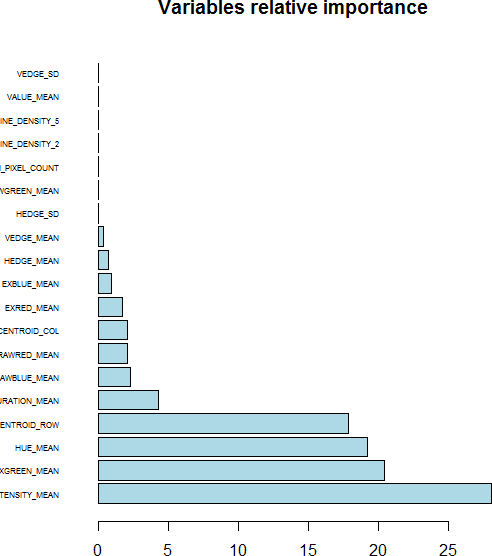


INTENSITY\_MEAN est la variable la plus intéressante dans le sens où elle a induit la somme de réduction d’impureté la plus élevée dans les arbres pour lesquels elle est apparue. HEDGE\_SD à VEDGE\_SD n’apparaissent dans aucun arbre. Leur influence est nulle (nous reviendrons sur ce commentaire plus loin, section [3.4.4](#_bookmark4)).

Une sortie graphique est disponible avec la commande **importanceplot()** :



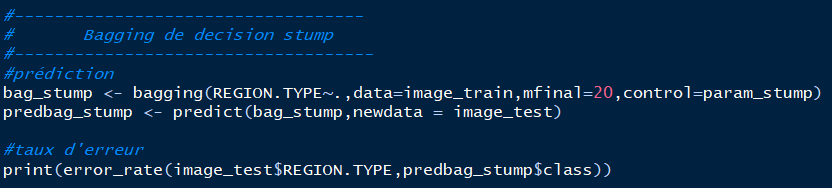
On note surtout un fort décalage entre la première, les 3 suivantes, puis les autres.



### Modifier les caractéristiques de l’arbre

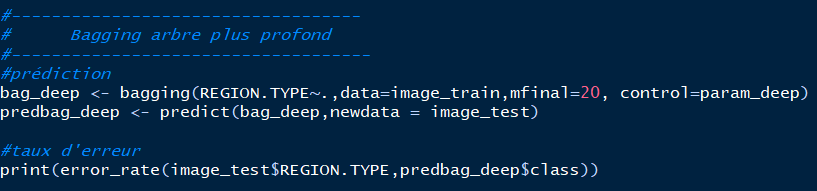
Le bagging n’agit que sur la composante variance de l’erreur, pas sur le biais. A priori, un bagging de decision stump ne devrait pas donner grand-chose ; en revanche, augmenter la taille des arbres individuels devrait améliorer le méta-modèle. Voyons ce qu’il en est.

**Bagging de decision stump**. Nous réutilisons les paramètres définis plus haut (section [3.3.2](#_bookmark1)).



Le taux d’erreur de **85.61%**, pire que l’arbre individuel.

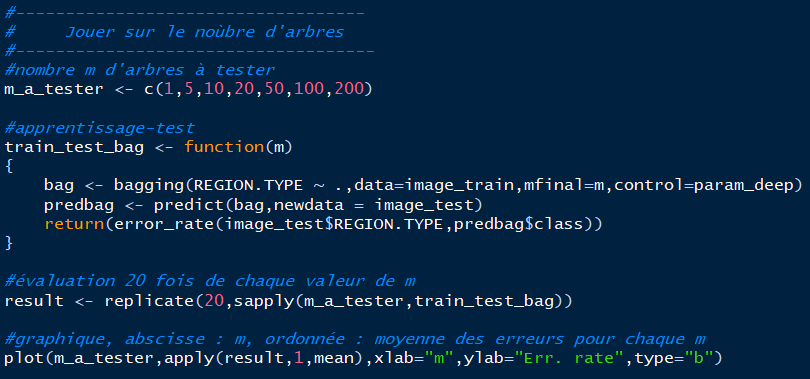
**Bagging avec un arbre plus profond**. On réduit le biais des arbres individuels en espérant que le mécanisme de vote va plus que compenser l’augmentation de la variance.

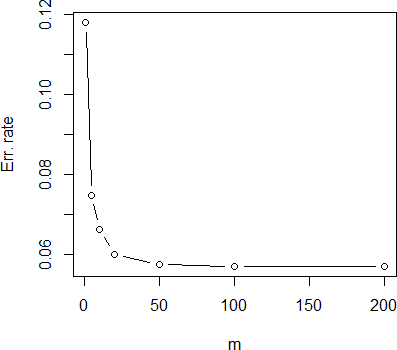


L’erreur est passée à **6.28%**, le meilleur résultat obtenu jusqu’à présent. Ce n’est qu’un exemple sur un seul fichier.

### Jouer sur le nombre d’arbres

Reste la question cruciale du nombre *m* d’arbres. Nous allons le faire évoluer [m = (1, 5, 10, 20, 50, 100, 200)] et mesurer le taux d’erreur sur l’échantillon test. Chaque valeur de *m* est évaluée **20** fois pour disposer d’une certaine stabilité dans les résultats. Nous calculons alors la moyenne des taux d’erreurs obtenus.





Le graphique relie le taux d’erreur avec le nombre de réplications. A partir de m = 50, les arbres additionnels n’améliorent pas les performances prédictives. Le taux d’erreur serait de **5.7%** pour m = 50. Nous devons prendre avec prudence cette valeur car nous avons utilisé l’échantillon test pour sélectionner le meilleur modèle. Il n’est plus vraiment impartial. Il serait plus judicieux d’utiliser une autre procédure pour la sélection de modèles (validation croisée par exemple)

## Random Forest (Forêts aléatoires)

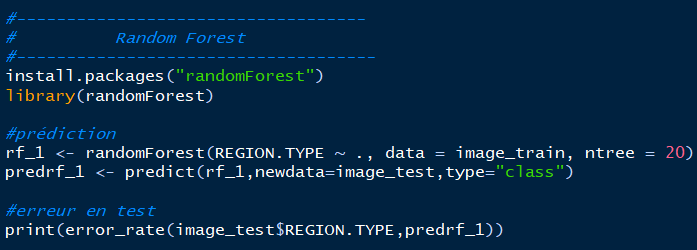
Par certains égards, Random Forest est une version améliorée du bagging, où le modèle sous-jacent sont forcément des arbres avec un processus de perturbation aléatoire pour les

« décorréler » (dans le bagging, le modèle sous-jacent peut être tout type de méthode bien que, dans les faits, les arbres sont quasiment systématiquement utilisés).

Nous utilisons la librairie « randomForest ». Un arbre de très grande taille est créée avec le paramétrage par défaut : lorsque *maxnodes* n’est pas spécifié, il n’y a pas de limitation sur la taille de l’arbre ; *nodesize* indique l’effectif minimal dans les feuilles (valeur par défaut 1).

### Random Forest avec 20 arbres

Nous construisons et évaluons un modèle avec 20 arbres.



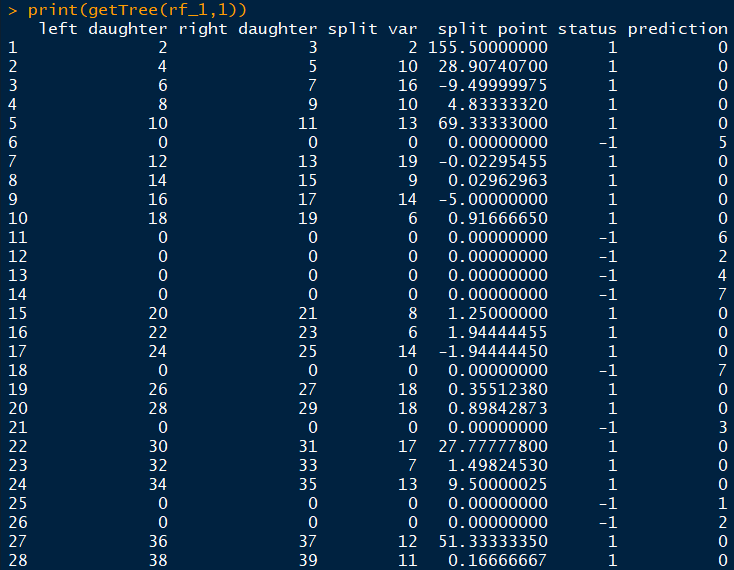
Le taux d’erreur est de **5.05%.** Nouveau record.

### Accès aux arbres

Nous avons accès aux arbres sous-jacents.



La présentation n’est pas vraiment intuitive :

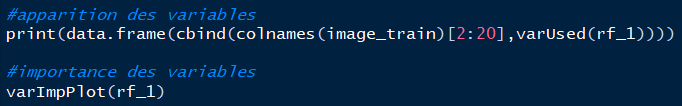


Les sommets sont numérotés, la variable de segmentation est identifiée par son numéro, un nœud intermédiaire correspond à un statut égal à 1, une feuille possède un statut égal à -1, le numéro de la classe est indiqué dans la colonne prédiction.

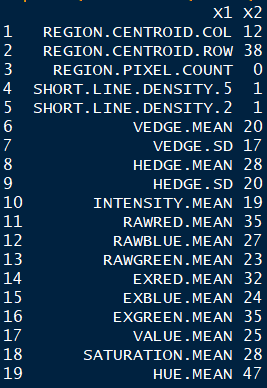
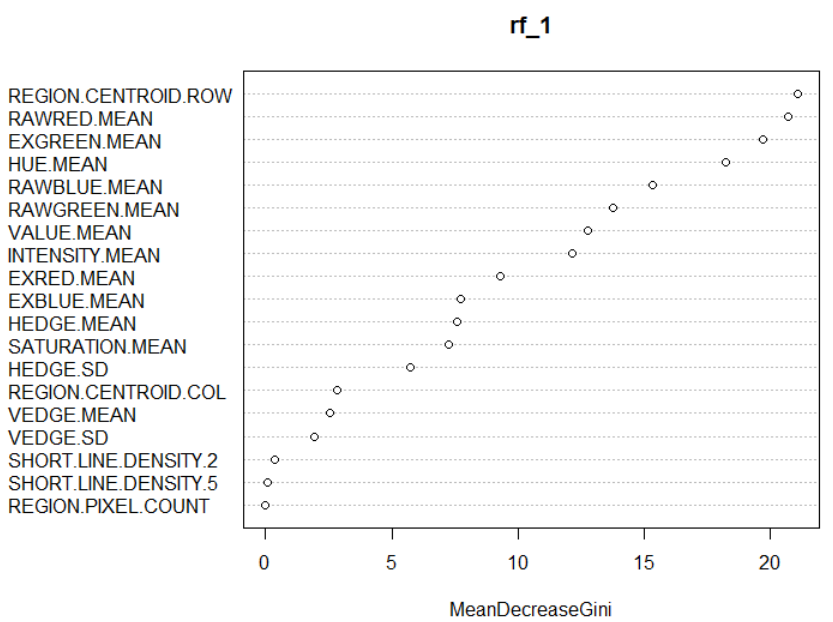
### Importance des variables

Le calcul de l’importance des variables est conforme au bagging. Le package

« randomForest » indique en plus le nombre d’apparition des variables, sachant qu’une variable peut être présente plusieurs fois dans un arbre.



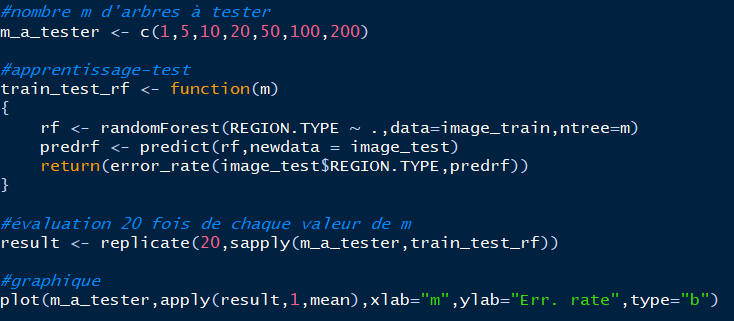
REGION\_PIXEL\_COUNT n’apparaît dans aucun arbre malgré le mécanisme de sélection aléatoire.

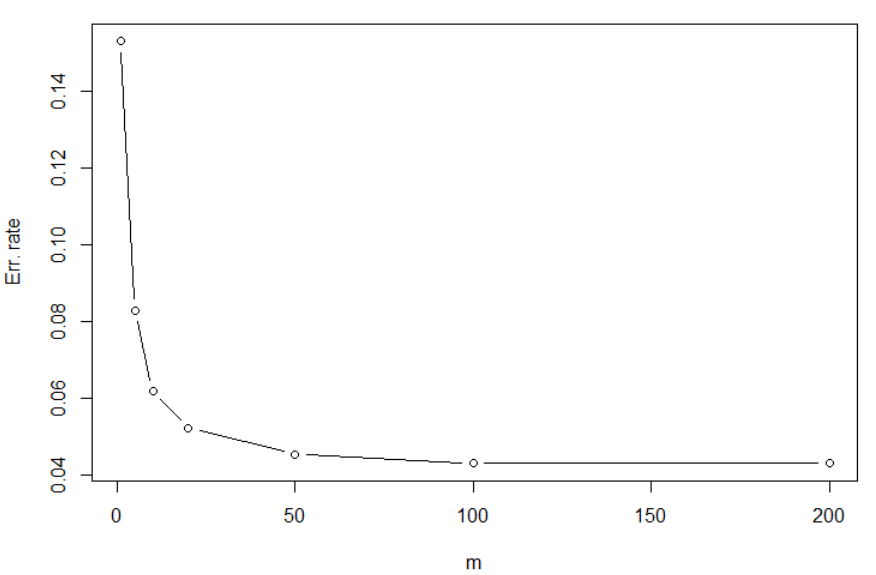
REGION.CENTROID.ROW semble la plus pertinente.

### Nombre d’arbres

Nous réitérons l‘expérimentation permettant d’identifier le bon nombre d’arbres. Par rapport à « adabag », randomForest se révèle particulièrement véloce. Les calculs sont rapidement menés à leur terme.



A partir de m = 100 arbres, l’erreur baisse faiblement, mais semble vouloir baisser quand même. Plus il y en a, mieux ça vaut, il n’y a pas de phénomène de sur apprentissage.



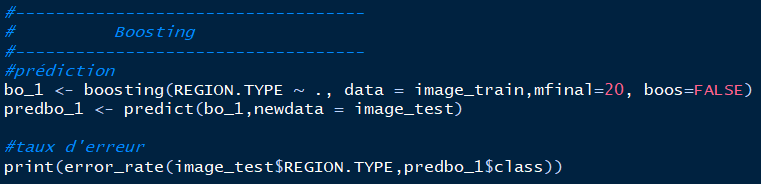
Avec m = 200, nous avons une erreur de **4.25%**.

## Boosting

Nous revenons au package « adabag » pour implémenter le boosting sous R. Nous ne serons pas dépaysés. La trame est identique aux deux analyses précédentes. L’importance des variables tient compte du « poids » de chaque arbre maintenant.

Les véritables enjeux tiennent au paramétrage des arbres sous-jacents et du nombre d’arbres. En effet, le boosting peut être sujet au sur apprentissage.

### Boosting avec 20 arbres (paramétrage par défaut)

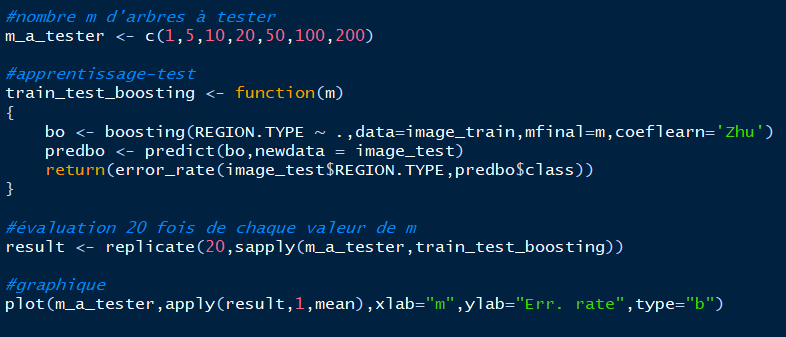


Le taux d’erreur est de **5.38%**. Meilleur que n’importe quel bagging que nous avions testé, moins bon (d’un souffle) en revanche que le random forest à 20 arbres (5.05%).

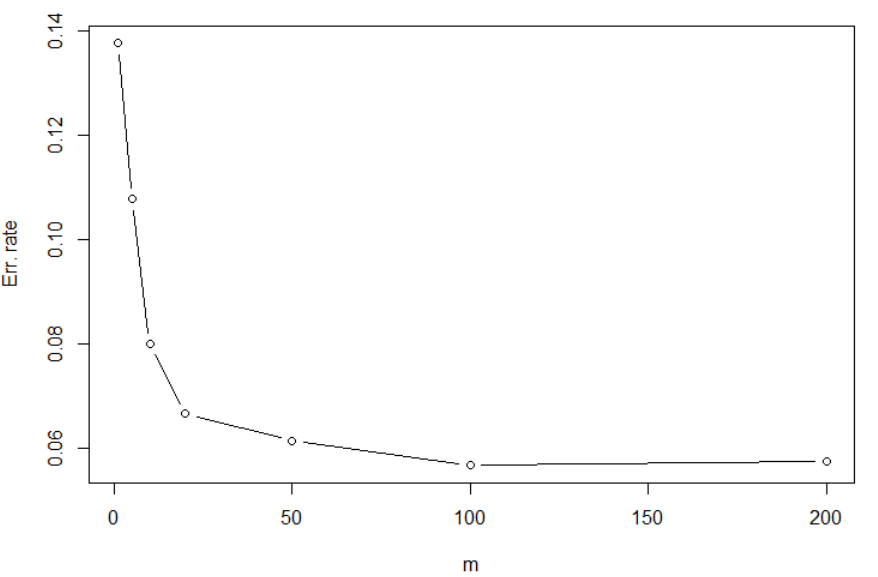
L’option « boos = FALSE » joue un rôle important. Il indique que nous utilisons la version originelle AdaBoost basée sur la pondération de tous les individus. .

### Nombre d’arbres

Nous réitérons l’expérimentation en changeant le nombre d’arbres.



Nous avons :



A partir de m = 50, la réduction est très faible. Il n’y pas de phénomène de sur apprentissage sur notre jeu de données lorsqu’on augmente « m » (jusqu’à m = 200 en tous les cas).

# Conclusion

Le premier objectif de ce tutoriel est d’apporter une touche concrète au support de cours consacré aux techniques ensemblistes de type bagging et boosting.J’en ai profité pour comparer les packages proposés par R (adabag, random forest). Random Forest et Boosting se partagent souvent les meilleures places dans les challenges. Il faut pas oublier que ces sont pas les seules technique ensemblistes qui existent, il y a plein d’autres qui sont très couramment utilisées dans l’industrie (XGBOOST, GBM, etc)

# Références

Package [‘’adabag](https://cran.r-project.org/web/packages/adabag/index.html)’’ pour R. Package ‘[’randomForest](https://cran.r-project.org/web/packages/randomForest/index.html)’’ pour R.