Université De Lille
Faculté des Sciences et Technologies
Département de Mathématiques

Classification supervisée

Méthodes d'ensemble (bagging et boosting)

Réalisé par :

- Abdelmonssif *Oufaska*
- Kamal *Samnouni*

Enseignant :

- *Prof.* Charlotte *Baey*

Année Universitaire :
2020/2021
15 Novembre 2020

Introduction

Lorsqu'il faut prendre une décision importante, il vaut souvent mieux recueillir plusieurs avis que de se fier à un seul. Utiliser des modèles ensemblistes pour prédire un comportement ou un prix, c'est un premier pas, dont les plus connues sont le bagging et le boosting.

Afin d'expliquer ce type de méthode, on va s'intéresser dans ce TP à la protéase du VIH, une enzyme qui possède un rôle majeur dans le cycle du virus du SIDA. Cette enzyme permet en effet le clivage de certaines protéines, ce qui a pour effet de libérer et de créer d'autres protéines néfastes à l'organisme et d'aider la propagation de la maladie. Pour mieux comprendre comment fonctionne la protéase du VIH, on étudiera une base de données contenant un ensemble d'octamères (i.e. une séquence de 8 acides aminés), dont certains ont été clivés par la protéase et d'autres non.

L'objectif de l'étude est de prédire, à partir de sa séquence d'acides aminés, si un octamère est clivé ou non par la protéase du VIH. A la fin, nous choisissons un modèle final, et nous testerons ce modèle final sur une nouvelle base. Nous verrons comment l'apport de ces modèles ensemblistes tient dans l'interprétation des résultats du modèle.
Les packages utilisés dans ce tutoriel sont les suivants :
—**caret**
—**rpart**
—**adabag**
—**randomForest**
—**xgboost**
—**ade4**

Table des matières

Protéase du VIH

Pré-traitement des données

Nous avons deux variables dans notre base de données : la première variable Octamer correspond à la séquence d'acides aminés composant l'octamère, et la deuxième variable Clived vaut 1 si l'octamère a été clivé par la protéase, et -1 sinon.
Nous allons effectuer ces changements pour notre bases de données, à partir de la séquence de 8 lettres, nous allons construire 8 variables qualitatives contenant chacune une des lettres de la séquence, puis nous allons recoder la variable cible Clived en un facteur valant 0 ou 1 : 1 si l'octamère a été clivé par la protéase, et 0 sinon
Voici un aperçu de l'analyse descriptive de notre nouvelle base de données :

Clived	Octamer1	Octamer2	Octamer3	Octamer4	Octamer5	Octamer6	Octamer7	Octamer8
0:1250	S :218	Q :149	V :250	L :208	L :186	E :186	V :140	T :156
1: 375	P :144	A :134	N :140	Y :141	V :136	A :161	A :130	Q :143
	A :135	R :133	L :119	S :125	A :133	V :158	Q :125	L :127
	T :114	G :127	T :111	E : 96	P :127	L :128	L :118	S :124
	L :110	K :123	A :103	K : 96	E :101	I :118	E :109	K :112
	K :106	L :119	E :101	A : 86	K : 95	T :105	M :100	E :107
	(Other):798	(Other):840	(Other):801	(Other):873	(Other):847	(Other):769	(Other):903	(Other):856

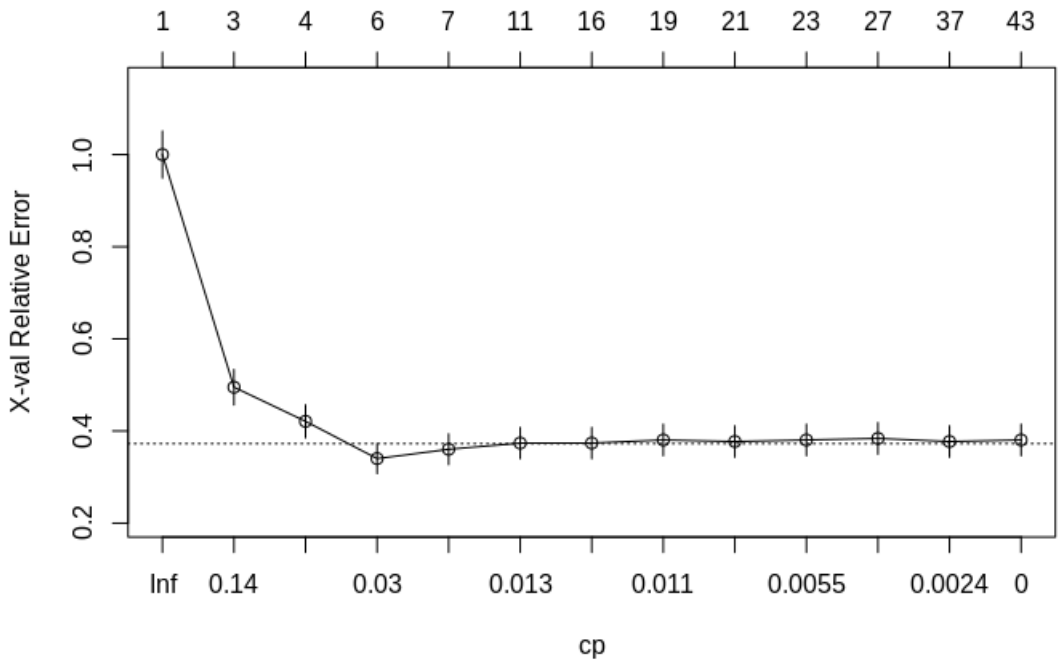
On peut noter que 1250 des octamères ont été clivés par la protéase ce qui correspond à 77% de nos observations. D'autre part 375 n'ont pas été clivés.
Afin d'évaluer les performances de notre modèle final, on propose de diviser notre base de données, une base d'apprentissage qui contient 80% des observations et une base test qui contient 20%. À partir de notre base d'apprentissage on va construire une famille de classifieurs par les méthodes de Bagging et Boosting. Après à l'aide de notre base test on calcule le taux d'erreur commis par le classifieur final de ces deux méthodes, dans le but de trouver les différents paramètres qui permettent de faire un compromis biais-variance pour les deux méthodes.
Pour bien visualiser les effets de ces deux méthodes sur un algorithme instable, on prend dans ce cas les arbres de décision.

Classifieur naïf

Au début on va construire un meilleur classifieur constant à partir de la base d'apprentissage. Cela revient à choisir la classe majoritaire de la base d'apprentissage. On a trouvé que son taux d'erreur sur la base de test est 24%, c'est une valeur un peu grande. Pour cette raison on doit chercher un meilleur classifieur que celui là.

Arbres de décision CART

Comme les arbres CART sont à la base de la plupart des méthodes d'ensemble, on commence par étudier les performances obtenues sur un arbre individuel.
A l'aide de la fonction rpart on fixe le paramètre de complexité à $cp = 0$ on obtient l'arbre maximal avec un taux d'erreur de 9.53% . Le graphique ci-dessous représente les erreurs trouver par la méthode de validation croisée en fonction du paramètre de complexité .



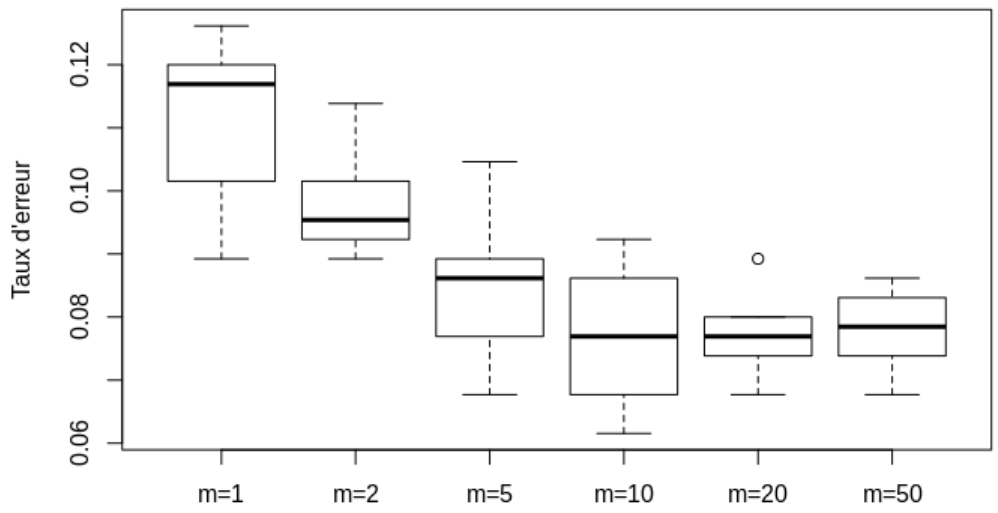
On remarque que la valeur de l'erreur se stabilise à une valeur de $cp = 0.013$ donc c'est la valeur qui réalise le compromis coût-complexité. Alors après l'élagage de l'arbre maximal selon le compromis complexité en utilisant la nouvelle valeur de cp dans la fonction précédente on obtient un arbre de 10 noeuds et 11 feuilles avec un taux d'erreur de 11.07%

Bagging

Le bagging permet de réduire la variance d'un classifieur par agrégation. Il est donc particulièrement adapté lorsque le classifieur de base a un faible biais et une forte variance. Nous allons tester (vérifier) cette propriété.
On construit un modèle de bagging à l'aide de la fonction bagging avec 20 arbres de type différent, on obtient les résultats suivants :

Type d'arbre	Par défaut	À un seul noeud	Profond
Taux d'erreur avec Bagging en %	5.84(v)	24(inv)	8.3(v)
Taux d'erreur sans Bagging en %	12	24	11.07

On remarque que le taux d'erreur du modèle de bagging avec 20 arbres de décision à 1 noeud reste inchangé après chaque compilation, car les arbres à un seul noeud construits sur les 20 échantillons Boostrap sont stable. En revanche le taux d'erreur de bagging pour les deux autres types d'arbres est changeable car à chaque compilation la fonction bagging permet de tirer avec remise 20 échantillons boostrap à partir de l'échantillon d'apprentissage.
On souhaite maintenant visualiser l'effet du nombre d'arbres (nombre des échantillons Boostrap) sur le taux d'erreur de modèle, on choisissant les arbres par défaut. En calculant le taux d'erreur moyen sur 10 répétitions, pour des valeurs de mfinal égales à 1, 2, 5, 10, 20 et 50, on a le graphique suivant

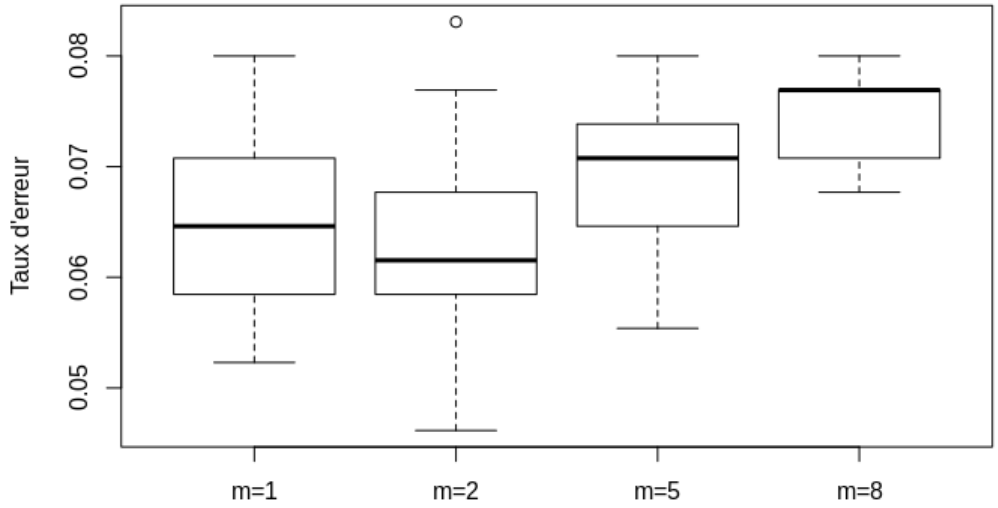


On remarque que plus que le nombre d'arbres est grand plus que le taux d'erreur diminue jusqu'à $m = 20$ après cette valeur le taux d'erreur augmente, on constate alors que la

meilleur valeur du nombre d'arbres qu'il faut utiliser pour minimiser le taux d'erreur est 20 arbres.

Les forêts aléatoires

Les forêts aléatoires sont un cas particulier d'algorithme de bagging appliqué aux arbres CART. Nous allons tester l'effet du nombre de variables explicatives et le nombre d'arbres utilisés dans réglages de l'algorithme.
Tout d'abord on va voir l'effet du nombre de variables explicatives. Le tableau ci-dessous représente l'évolution du taux d'erreur moyen calculé sur 10 forêts aléatoires pour chaque valeur du nombre de variables utilisés.



On peut remarque que le choix de $m = 2$ donne une valeur optimale de taux d'erreur de 6.33%. On veut savoir maintenant l'effet du nombre d'arbres on prenant la valeur optimale de mtry $m = 2$, sur le taux d'erreur en fonction du nombre d'arbres on prenant un exemple de 1, 2, 5, 10, 20, 50, 100, 200 ou 500. On a le tableau ci-dessous pour les valeurs de l'erreur :

nombre d'arbres	1	2	5	10	20	50	100	200	500
Taux d'erreur moyen en %	16.43	13.38	9.81	7.27	6.21	5.47	5.78	5.38	5.23

On remarque que la valeur de l'erreur diminue quand le nombre d'arbres augmente jusqu'à 50 on trouve le taux d'erreur 5.47%, après 50 on visualise une perturbation de l'erreur. On a trouvé le même résultat à chaque compilation. Donc pour trouver un meilleur taux d'erreur on doit retirer avec remise 50 échantillons Boostrap.

Boosting

On s'intéresse maintenant aux méthodes de boosting, qui contrairement aux méthodes de bagging, ne construisent pas des arbres de décision en parallèle, mais de façon séquentielle, chaque arbre se concentrant d'avantage sur les individus les moins bien classés par l'arbre précédent. Ces méthodes fonctionnent bien lorsque les classifieurs sont faibles (fort biais et donc fort variance)

Adaboost

Dans la version originelle de l'algorithme Adaboost, chaque observation est pondérée par un poids ω_i, et aucune source d'aléa n'est introduite contrairement aux méthodes de bagging. Les résultats de l'algorithme sont donc déterministes : si on lance plusieurs fois la méthode sur le même jeu de données, on obtient exactement les mêmes résultats.

On utilise la fonction boosting, on va construire un modèle basé sur l'algorithme Adaboost et différents types d'arbres, avec les deux option boos=TRUE pour pondérer chaque observation par un poids sur chaque itération et boos=TALSE pour fixer le même poids pour les observations à chaque itération. Puis on calculons le taux d'erreur. On propose de choisir 20 itérations de l'algorithme. Voici les résultats obtenus :

Type d'arbre	Par défaut	À un seul noeud	Profond
Taux d'erreur en (boos=T)%	4.61	6.46	4.30
Taux d'erreur en (boos=F)%	3.38	8.30	5.84

On remarque que avec boos=T le modèle d'Adaboost basé sur les arbres de décisions profonds est meilleur que les deux autres types, avec un taux d'erreur 4.30%, à chaque compilation on prend des nouveaux poids ce qui explique que le taux d'erreurs est changé, dans ce cas on trouve un taux d'erreur petit ce qui est compatible avec les résultats théoriques. En revanche, avec boos=F le résultat est inchangé après chaque compilation car les poids sont fixés et on utilise toutes les observations de la base d'apprentissage à chaque itération. Ainsi, la méthode Adaboost est meilleur que la méthode Bagging car le taux d'erreur est plus petit pour les trois types d'arbres de décision.

XGBoost

XGBoost n'est pas une méthode spécifique, mais une implémentation efficace des méthodes de gradient boosting. Sous R, cette librairie est disponible dans le package xgboost. Ce package fonctionne un peu différemment des précédents, et a sa propre syntaxe.

Tout d'abord on doit transformer notre base de données en une base adaptable à la fonction xgb.train qui prend les variables de type binaire {0,1}, donc on transforme chaque variable explicative de l'étude sous forme de variable indicatrice.

À l'aide de la fonction xgb.train qui applique un algorithme de gradient boosting adapté à la classification binaire, et avec nrounds=20 itérations, nous allons tester l'effet des paramètres de l'algorithme (maxdepth pour la taille des arbres, et eta pour le taux d'apprentissage) sur le taux d'erreur. On utilise l'option objective="binary :logistic", pour approcher le gradient de la fonction de perte à chaque itération par une fonction de perte logistique. On a le tableau suivant :

eta	0.75			0.5			0.3		
maxdepth	30	20	1	30	20	1	30	20	1
Erreur en %	5.53	5.53	11.07	4.92	5.23	12.30	5.84	5.84	17.23
À l'itération	9	9	8	8	8	9	11	11	7

On note que la quatrième ligne de notre tableau représente le nombre d'itérations qu'il faut prendre pour trouver le compromis entre l'erreur sur la base de test et l'erreur sur la base d'apprentissage.
L'avantage de cette méthode, le calcul rapide de l'erreur. On a remarqué que l'erreur sur la base d'apprentissage diminue à chaque itération pour tous les différents choix de eta ou maxdepth. En revanche, l'erreur sur la base test diminue jusqu'à une certaine itération on visualise des perturbations au niveau de l'erreur. Un choix d'une grande valeur (0.75) pour eta ou une petite valeur (0.3) donne un taux erreur un peu plus grand par rapport au choix de eta=0.5 cela d'une part et d'autre part la diminution de la taille des arbres provoque une augmentation de taux d'erreur. Donc on propose de prendre un taux d'apprentissage de 0.5 et utiliser les arbres de taille 30, pour trouver le meilleur taux d'erreur qui réalise un compromis entre l'erreur sur la base apprentissage et sur la base test.

Conclusion

D'après les résultats de notre étude on propose de choisir la méthode Adaboost basé sur les arbres de décisions profonds avec 20 itérations de l'algorithme pour prédire la classe

de nouvelles observations, car on a trouvé un taux d'erreur de 4.3% qui est très petit par rapport à les autres méthodes. Nous allons tester ce modèle final sur une nouvelle base de 746 observations. le tableau ci-dessous représente la matrice de confusion :

	Observés : 0	Observés : 1
Prédits : 0	328	16
Prédits : 1	12	390

À l'aide de notre méthode, on a prédit que 328 octamères n'ont pas été clivés et 390 octamères ont été clivés, En revanche notre modèle a prédit que 12 octamères ont été clivés et dans les observations ils sont pas clivés, pareil pour il a prédit que 16 octamères n'ont pas été clivés et dans les observations, ils sont clivés. Le taux d'erreur est 3.75%.

Annexe : Code R

```
1
2  #library
3  library(forestmodel)
4  library(ggplot2)
5  library(cowplot) # graphe combiner
6  library(forcats) # facteur
7  library(randomForest)
8  library(ade4)
9  library(rpart)
10 library(rpart.plot)
11 library(xgboost)
12 library(caret) #matrice de confusion
13 library(adabag)
14
15 Data
16 ```{r cars}
17 d <- read.table("~/Bureau/M thodes_Dapprentissag/bagging_boosting/1625Data.txt",sep
       = ',',header=T)
18
19 ```
20
21
22 ```{r cars}
23 head(d)
24 ```
25 ```{r cars}
26
27 ```
28 ```{r cars}
29 d$Octamer=as.character(d$Octamer)
30 Table_char=matrix(NA,ncol = 8,nrow = 1625)
31 for (i in 1:1625)
32 {
33   Table_char[i,]=unlist(strsplit(d[i,1],""))
34 }
35 c=cbind(d$Clived,Table_char)
36 ```
37
38
39 ```{r cars}
40 d=data.frame(c)
41 names(d)=c("Clived",paste0("Octamer",1:8))
42 ```
43
44 ```{r cars}
45 d$Octamer1=as.factor(d$Octamer1)
```

```
d$Octamer2=as.factor(d$Octamer2)
d$Octamer3=as.factor(d$Octamer3)
d$Octamer4=as.factor(d$Octamer4)
d$Octamer5=as.factor(d$Octamer5)
d$Octamer6=as.factor(d$Octamer6)
d$Octamer7=as.factor(d$Octamer7)
d$Octamer8=as.factor(d$Octamer8)
d$Clived=as.factor(d$Clived)
```

```{r cars}
d$Clived=factor(d$Clived,levels=c("-1","1"),labels = c("0","1"))
```

```{r cars}
summary(d)
```

```{r cars}
idtrain=sample(1:nrow(d),0.8*nrow(d))
dtrain=d[idtrain,]
dtest=d[-idtrain,]
```

```{r cars}
prop.table(table(dtest$Clived))
prop.table(table(dtest$Clived))
```

```{r cars}
#fonction qui calcul le taux d'erreur
tx_er=function(pred,vrais){
  mc=table(pred,vrais)
  1-sum(diag(mc))/sum(mc)
}


# Classifieur contant
table_clived=table(dtrain$Clived)
mcst=names(table_clived)[which.max(table_clived)]
te_cst=mean(dtest$Clived!=mcst)
cat("Taux d'erreur de classifieur constant est =",te_cst)
```

```{r cars}
#reglage par d faut
mcart=rpart(Clived~.,d=dtrain )
plotcp(mcart)
predcart=predict(mcart,newdata = dtest,type = "class")
te_cart=tx_er(predcart,dtest$Clived)
rpart.plot(mcart)
cat("Taux d'erreur de cart par defaut =",te_cart)
```

```{r cars}
# arbre maximal
mcartmax=rpart(Clived~.,d=dtrain,minbucket=1,maxdepth=30,cp=0)
plotcp(mcartmax)
rpart.plot(mcartmax)
```

```
predcartmax=predict(mcartmax,newdata=dtest,type = "class")
te_cartmax=tx_er(predcartmax,dtest$Clived)
cat("Taux d'erreur d'arbre maximal est =",te_cartmax)

head(confusionMatrix(dtest$Clived,predcartmax))# matrice de confusion
```
Si on lage selon le compromis cout-complexit
```{r cars}
mcart=prune(mcartmax,cp=0.013)
rpart.plot(mcart)
predcart=predict(mcart,newdata = dtest,type = "class")
te_cart=tx_er(predcart,dtest$Clived)
cat("Taux d'erreur d'arbre selon cout-complexite ",te_cart)
```
arbre   1 noeud
```{r cars}
mcartone=rpart(Clived~.,d=dtrain,minbucket=1,maxdepth=1,cp=0)
plotcp(mcartone)
rpart.plot(mcartone)
predcartone=predict(mcartone,newdata=dtest,type = "class")
te_cartone=tx_er(predcartone,dtest$Clived)
cat("Taux d'erreur d'arbre maximal est =",te_cartone)
```

Bagging

```{r cars}
# Par d faut
mbag <- bagging(Clived~.,data=dtrain,mfinal=20)
predbag20 <- predict(mbag,newdata=dtest,type="class")
te_bag20 <- tx_er(predbag20$class,dtest$Clived)
cat(" Taux d'erreur commise par le  classifieur final (par defaut): ",te_bag20)
```

```{r cars}
# Bagging d'arbres    1 noeud
mbagstump <- bagging(Clived~.,data=dtrain,mfinal=20,control=rpart.control(cp=0,
    maxdepth=1,minbucket=1))
predbagstump <- predict(mbagstump,newdata=dtest,type="class")
te_bagstump <- tx_er(predbagstump$class,dtest$Clived)
cat(" Taux d'erreur commise par le  classifieur final (un seul noeud) : ",te_bagstump
    )
```
```{r cars}
# Bagging d'arbres profonds
mbagdeep <- bagging(Clived~.,data=dtrain,mfinal=20,control=rpart.control(cp=0.013,
    maxdepth=30,minbucket=1))
predbagdeep <- predict(mbagdeep,newdata=dtest,type="class")
te_bagdeep <- tx_er(predbagdeep$class,dtest$Clived)
cat(" Taux d'erreur commise par le  classifieur final (arbres profonds) : ",te_
    bagdeep)
```
```
# Visualisation de l'effet du nombre d'arbres
```{r cars}
effetnbTreesBag <- function(m){
  err <- sapply(1:10, FUN = function(i){
    bag <- bagging(Clived ~ .,data=dtrain,mfinal=m)
    predbag <-predict(bag,newdata = dtest)
    return(tx_er(dtest$Clived,predbag$class))
```
```

```
  })
  return(err)
}

mval <- c(1,2,5,10,20,50)
err_fn_m_bag <- sapply(mval,FUN = function(m){effetnbTreesBag(m)})
dmBag <- as.data.frame(err_fn_m_bag)
names(dmBag) <- paste0("m=",mval)
boxplot(dmBag,ylab="Taux d'erreur")

plot(mval,apply(err_fn_m_bag,2,mean),type="b",xlab="Nombre d'arbres bagg s", ylab="
    Taux d'erreur")

```
RF


```{r cars}

mrf <- randomForest(Clived~.,data=dtrain,method="class",ntree=20,mtry=1)
predrf <- predict(mrf,newdata=dtest,type="class")
te_rf <- tx_er(predrf,dtest$Clived)
```
Effet du nombre de variables   chaque coupure
```{r cars}

effetmtry <- function(m){
  err <- sapply(1:10, FUN = function(i){
    rf <- randomForest(Clived ~ .,data=dtrain,ntree=20,mtry=m)
    predrf <-predict(rf,newdata = dtest)
    return(tx_er(dtest$Clived,predrf))
  })
  return(err)
}

mtryval <- c(1,2,5,8)
err_fn_mtry <- sapply(mtryval,FUN = function(m){effetmtry(m)})
dmBag <- as.data.frame(err_fn_mtry)
names(dmBag) <- paste0("m=",mtryval)
boxplot(dmBag,ylab="Taux d'erreur")

plot(mtryval,apply(err_fn_mtry,2,mean),type="b",xlab="mtry",ylab="Taux d'erreur")
```
Effet du nb d'arbres dans la for t
```{r cars}

effetnbTrees <- function(m){
  err <- sapply(1:10, FUN = function(i){
    rf <- randomForest(Clived ~ .,data=dtrain,mtry=2,ntree=m)
    predrf <-predict(rf,newdata = dtest)
    return(tx_er(dtest$Clived,predrf))
  })
  return(err)
}

mval <- c(1,2,5,10,20,50,100,200,500)
err_fn_m <- sapply(mval,FUN = function(m){effetnbTrees(m)})
dm <- as.data.frame(err_fn_m)
names(dm) <- paste0("m=",mval)
boxplot(dm,ylab="Taux d'erreur")

plot(mval,apply(err_fn_m,2,mean),type="b",xlab="Nombre d'arbres",ylab="Taux d'erreur"
```

```
    )
```

```
# Boosting
```{r cars}
#par d faut
mboost_def <- boosting(Clived~.,data=dtrain,mfinal=20,boos=F)

predboost_def <- predict(mboost_def,newdata=dtest,type="class")
te_boost_def <- tx_er(predboost_def$class,dtest$Clived)
cat ("Taux d'erreur comise par le classifieur final (par defaut) :",te_boost_def)

```{r cars}
# un seul noeud
mbooststump <- boosting(Clived~.,data=dtrain,mfinal=20,boos=F,control=rpart.control(
    cp=0,maxdepth=1,minbucket=1))
predbooststump <- predict(mbooststump,newdata=dtest,type="class")
te_booststump <- tx_er(predbooststump$class,dtest$Clived)
cat("Taux d'erreur comise par le classifieur final (un seul noeud) :",te_booststump)
```

```{r cars}
# arbre profond
mboostdeep <- boosting(Clived~.,data=dtrain,mfinal=20,boos=T,control=rpart.control(cp
    =0,maxdepth=30,minbucket=1))
predboostdeep <- predict(mboostdeep,newdata=dtest,type="class")
te_boostdeep <- tx_er(predboostdeep$class,dtest$Clived)
cat("Taux d'erreur comise par le classifieur final (arbre profond) :",te_boostdeep)

```
XGBoost

Transforme les donn es en dummy variables
```{r cars}

class <- as.numeric(d$Clived)
class <- class-1
ddummy <- cbind(class,acm.disjonctif(d[,-1]))

ddumtrain <- ddummy[idtrain,]
ddumtest <- ddummy[-idtrain,]

```

```{r cars}
dtrainXG <- xgb.DMatrix(as.matrix(ddumtrain[,-1]),label=as.matrix(ddumtrain[,1]))
dtestXG <- xgb.DMatrix(as.matrix(ddumtest[,-1]),label=as.matrix(ddumtest[,1]))
watchlist <- list(train=dtrainXG,test=dtestXG)

mxgb <- xgb.train(params = list(max_depth=1, eta = 0.3, objective="binary:logistic"),
                  data = dtrainXG,
                  nrounds = 20,
                  watchlist = watchlist)

```

Application au NewData
```{r cars}
NewData=read.table("~/Bureau/M thodes_Dapprentissag/bagging_boosting/746Data.txt",
    sep = ',',header=T)
NewData$Octamer=as.character(NewData$Octamer)
Table_char=matrix(NA,ncol = 8,nrow = 746)
```

```
for (i in 1:746)
{
  Table_char[i,]=unlist(strsplit(NewData[i,1],""))
}
u=cbind(NewData$Clived,Table_char)
NewData=data.frame(u)
names(NewData)=c("Clived",paste0("Octamer",1:8))
NewData$Octamer1=as.factor(NewData$Octamer1)
NewData$Octamer2=as.factor(NewData$Octamer2)
NewData$Octamer3=as.factor(NewData$Octamer3)
NewData$Octamer4=as.factor(NewData$Octamer4)
NewData$Octamer5=as.factor(NewData$Octamer5)
NewData$Octamer6=as.factor(NewData$Octamer6)
NewData$Octamer7=as.factor(NewData$Octamer7)
NewData$Octamer8=as.factor(NewData$Octamer8)
NewData$Clived=as.factor(NewData$Clived)
NewData$Clived=factor(NewData$Clived,levels=c("-1","1"),labels = c("0","1"))


```

```{r cars}
mboostdeep <- boosting(Clived~.,data=dtrain,mfinal=20,boos=T,control=rpart.control(cp
    =0,maxdepth=30,minbucket=1))
predboostdeep <- predict(mboostdeep,newdata=NewData,type="class")
te_boostdeep <- tx_er(predboostdeep$class,NewData$Clived)
cat("Taux d'erreur comise par le classifieur  :",te_boostdeep)
table(NewData$Clived,predboostdeep$class)
```