TP2 Apprentissage Statistique

Romane PERSCH & Maxime CALLOIX

15 novembre 2016

Prédicteur kNN et validation croisée

Validation croisée pour choisir k :

```
##  [1] 6 6 6 6 5 7 6 6 8 7
```

L'erreur la plus faible est obtenue pour k = 1, 2, 3, 4, 7 et 8 (6 erreurs de classification). Attention, cela dépend de la seed choisie.

Question 1. : La commande **apply(cvpred,2,function(x) sum(class!=x))** permet d'appliquer la fonction function(x) sum(class!=x) à la matrice cvpred **en colonne** (argument MARGIN égal à 2). Dans le cas présent, la matrice cvpred a 10 colonnes correspondant chacune à un k ($k = 1, \dots, 10$). Chaque colonne indique les prédictions obtenues sur l'ensemble de la base (c'est-à-dire sur les individus de chacun des 5 folds) avec le k considéré. La fonction function(x) sum(class!=x) appliquée à une colonne permet donc de compter le nombre de prédictions différentes de la réalité lorsque les prédictions sont obtenues avec le k considéré (les véritables valeurs sont contenues dans le vecteur class). En effet, un TRUE résultant du test class!=x est compté comme le nombre 1 par la fonction sum et un FALSE est compté comme 0. La commande renvoie donc finalement un vecteur de taille 10 indiquant le nombre d'erreurs obtenues pour chaque $k = 1, \dots, 10$.

Question 2. : Lorsqu'on ne met pas de seed (ce qui était le cas dans le TD initialement), on obtient des résultats différents à chaque fois que ce morceau de code est relancé. Ceci est dû au choix des folds qui a lieu aléatoirement avec la fonction sample : la fonction sample effectue une permutation aléatoire du vecteur rep(1:5,each=18) contenant 18 fois le chiffre 1, puis 18 fois le chiffre 2 etc. Ceci permet d'attribuer un numéro de folds entre 1 et 5 aléatoirement à chaque individu de la base train. En imaginant qu'on lance 100 fois ce morceau de code, une stratégie pour choisir k est de sommer les 100 erreurs obtenues pour chacun de k (autrement dit de sommer les 100 vecteurs issus de apply(cvpred,2,function(x) sum(class!=x))) et de choisir le k pour lequel l'erreur totale sur ces 100 essais est la plus petite.

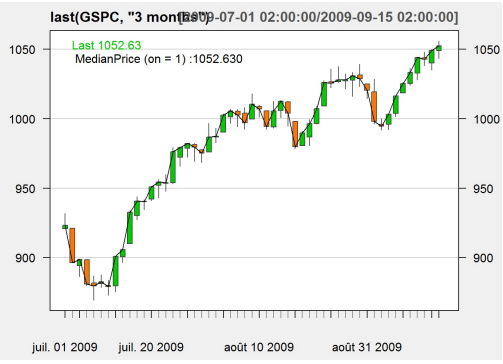
Predicting Stock Market Returns

Remarque : Lors du calcul de la fonction indicateur, il semble manquer un argument à la fonction Delt(v, k = x), permettant d'utiliser la valeur du Close pour calculer les V_i . Ici, le code calcule $V_i^j = \frac{\overline{P}_{i+j} - \overline{P}_i}{\overline{P}_i}$ au lieu de $V_i^j = \frac{\overline{P}_{i+j} - C_i}{C_i}$.

Question 3.1 : Ajouter les valeurs médianes de (C_i, H_i, L_i) au graphique des chandeliers japonais :

```
candleChart(last(GSPC, "3 months"), theme = "white", TA = NULL)
medPrice = function(p) apply(HLC(p), 1, median)
addMedPrice = newTA(FUN = medPrice, col = 1, legend = "MedianPrice")

get.current.chob<-function(){quantmod:::get.current.chob()}
candleChart(last(GSPC, "3 months"), theme = "white", TA = "addMedPrice(on=1)")
```



On peut remarquer que la valeur médiane de (C_i, H_i, L_i) est toujours égale à C_i , par définition du High et du Low. Elle est donc sans suprise toujours située à l'une des extrêmités des bâtons du graphique, chaque bâton indiquant par sa couleur s'il y a eu une hausse ou une baisse entre l'Open et le Close, et chacune de ses extrêmités correspondant à la valeur de l'Open et du Close. Ainsi, lorsqu'il y a eu une hausse au cours de la journée, le bâton est vert et la médiane (qui est égale à la valeur du Close) est située en haut du bâton (alors que l'Open correspond au bas du bâton). Au contraire, lorsqu'il y a eu une baisse au cours de la journée, le bâton est orange et la médiane est située en bas du bâton.

Question 3.1 (suite) : Lorsqu'on supprime l'argument 'on = 1' :

```
## Error: improper TA argument/call in chartSeries
```

On a l'erreur "Error: improper TA argument/call in chartSeries" car le package ne sait pas s'il faut tracer ou non la courbe indiquant la moyenne. Si on précise 'on = 0', il ne va pas tracer la courbe indiquant la moyenne.

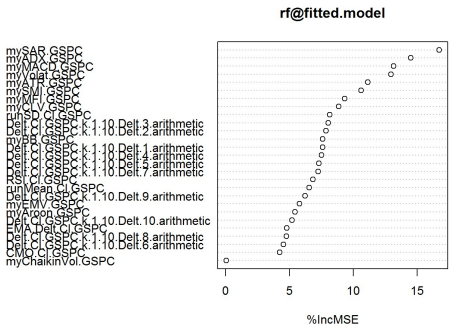
Et en indiquant l'argument 'on = 0', on obtient les chandeliers sans la courbe des valeurs moyennes.

Random Forest pour obtenir l'importance des variables :

Question 3.2 : L'option **training.per** permet de définir la base d'apprentissage : on spécifie ici qu'il s'agit de nos données xts (série temporelle multivariée) allant de la première date disponible jusqu'au 12 décembre 1999.

L'option **importance** permet d'indiquer au package randomForest qu'il doit aussi calculer l'importance de chaque prédicteur, pour que nous puissions l'exploiter ensuite.

```
#pourcentage d'augmentation de l'erreur quadratique due à la suppression d'une variable explicative
varImpPlot(rf@fitted.model, type = 1)
```



Question 3.3 : Le graphique indiquant le pourcentage d'augmentation de l'erreur quadratique due à la suppression d'une variable explicative, les 8 variables les plus pertinentes sont celles qui causent l'augmentation de l'erreur la plus forte en pourcentage. On conserve donc ici les 8 variables suivantes : mySAR, myADX, myMACD, myVolat, myATR, mySMI, myMFI et myCLV.

Question 3.4 : Le nouveau modèle est donc :

```
data.model = specifyModel(T.ind(GSPC) ~ mySAR(GSPC) + myADX(GSPC) + myMACD(GSPC)
                          + myVolat(GSPC) +myATR(GSPC) +  mySMI(GSPC) +  myMFI(GSPC)
                          + myCLV(GSPC))
set.seed(1234)
rf = buildModel(data.model,method="randomForest",
                training.per=c(start(GSPC),index(GSPC["1999-12-31"])), ntree=50, importance=T)
```

Question 3.5 : La fonction **na.omit** supprime les observations qui contiennent au moins un NA (une valeur manquante). Ici, en l'occurence, une seule observation a été supprimée. Il est important de le faire dans la base de test car on ne pourra pas faire de prédiction pour ces observations. Cependant, dans la base train, les valeurs manquantes posent moins problème car la commande na.omit est en général automatiquement effectuée lors de l'entraînement du modèle.

Signal

```
#Même traitement de la variable T.ind sur la base test, afin de pouvoir ensuite évaluer les résultats :
Tdata.eval[,1] = trading.signals(Tdata.eval[,1],0.1,-0.1)
names(Tdata.eval)[1] = "signal"
summary(Tdata.eval$signal)
```

Question 4. : Algorithme kNN pour prédire la variable signal

```
set.seed(1234)
#kNN avec k = 3
pred = knn(Tdata.train[,2:9],Tdata.eval[,2:9], Tdata.train[,1], k = 3)
# affichage du tableau de contingence
table(pred,Tdata.eval[,1])
```

```
##
## pred    s    h    b
##    s   90  344   59
##    h  284 1041  234
##    b   64  258   56
```

```
#Calcul du nombre total d'erreurs :
sum(Tdata.eval[,1]!=pred)
```

```
## [1] 1243
```

Validation croisée pour choisir k :

```
set.seed(1234)
# 5-fold cross-validation to select k
# from the set {1,...,10}
fold = sample(c(rep(1:5,each= dim(Tdata.train)[1] %/% 5), rep(5,dim(Tdata.train)[1] %% 5)) )
# creation des groupes B_v
cvpred = matrix(NA,nrow=dim(Tdata.train)[1],ncol=10) # initialisation de la matrice
# des prédicteurs
for (k in 1:10)
  for (v in 1:5)
    {
    sample1 = Tdata.train[which(fold!=v),2:9]
    sample2 = Tdata.train[which(fold==v),2:9]
    class1 = Tdata.train[which(fold!=v),1]
    cvpred[which(fold==v),k] = knn(sample1,sample2,class1,k=k)
  }
class = as.numeric(Tdata.train[,1]) # display misclassification rates for k=1:10
result = apply(cvpred,2,function(x) sum(class!=x)) # calcule l'erreur de classif.
result
```

```
##  [1] 1628 1785 1704 1782 1800 1828 1806 1857 1882 1917
```

On choisit donc, d'après la validation croisée, k = 1.

```
set.seed(1234)
#kNN avec k = 1
pred = knn(Tdata.train[,2:9],Tdata.eval[,2:9], Tdata.train[,1], k = 1)
# affichage du tableau de contingence
table(pred,Tdata.eval[,1])
```

```
##
## pred   s   h   b
##    s 102 409  52
##    h 259 957 241
##    b  77 277  56
```

```
prop.table(table(pred,Tdata.eval[,1]), 2)
```

```
##
## pred         s         h         b
##    s 0.2328767 0.2489349 0.1489971
##    h 0.5913242 0.5824711 0.6905444
##    b 0.1757991 0.1685940 0.1604585
```

```
#Calcul du nombre total d'erreurs :
sum(Tdata.eval[,1]!=pred)
```

```
## [1] 1315
```

On remarque pourtant qu'avec k = 1, on obtient finalement plus d'erreurs de prédiction sur la base test qu'avec k = 3.

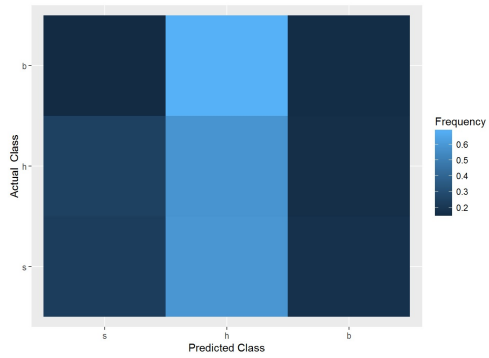
Le tableau de contigence nous indique que les classes sont très mal prédites. Les fréquences de prédiction sont proches des fréquences empriques de la base d'apprentissage : il semble presque que les valeurs sont prédites au hasard. Ce phénomène est probablement dû au fait que la classe "hold" est bien plus représentée que les deux autres (ce qui diminue la probabilité d'avoir des voisins autres que "hold").

```
library(ggplot2)

confusion <- as.data.frame(as.table(prop.table(table(pred,Tdata.eval[,1]), 2) ))

plot <- ggplot(confusion)

plot + geom_tile(aes(x=pred, y=Var2, fill=Freq)) + scale_x_discrete(name="Predicted Class") + scale_y_di
screte(name="Actual  Class") + scale_fill_gradient(breaks=seq(from=0, to=1, by=0.1)) + labs(fill="Freque
ncy")
```



Le graphique met bien en évidence ce phénomène : les classes b et s sont très mal prédites car hold est prédit en majorité pour ces 2 classes.

Question 5 : Arbre de décision

Si on laisse les paramètres par défaut de l'arbre de décision, on n'obtient qu'une racine. On peut donc éventuellement ajuster ces paramètres (voir Annexe), mais cela n'a pas conduit à une amélioration de la qualité des prédictions.

```
library(rpart)
rt.signal1 = rpart(signal ~ ., data = Tdata.train[,1:9], method = 'class')
rt.signal1
```

```
## n= 7542
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 7542 2136 h (0.1198621 0.7167860 0.1633519) *
```

Comme l'arbre ne contient qu'un seul noeud, on ne peut donc pas l'afficher sous forme de graphique :

```
par(lwd=2, bg="lemonchiffon3")
prettyTree(rt.signal1,col="navy",bg="lemonchiffon")
```

```
## Error in plot.rpart(t, uniform = uniform, branch = branch, margin = margin, : fit is not a tree, just
a root
```

```
pred1 = predict(rt.signal1,Tdata.eval[,2:9], type="class")
#Erreur de l'arbre de décision par défaut
print(sum(Tdata.eval[,1]!=pred1))
```

```
## [1] 787
```

```
#Répartition des prédictions
summary(pred1)
```

```
##    s    h    b
##    0 2430    0
```

Ce premier arbre, qui n'a qu'une racine, prédit donc la classe h pour toutes les dates. Le phénomène constaté avec k-NN de "sur-prédiction" de la classe h est donc ici amplifié. Néanmoins, le nombre d'erreurs de prédiction est 787, ce qui est bien inférieur aux erreurs de prédictions des deux modèles k-NN (1243 et 1315 pour le modèle issu de la validation croisée).

Conclusion

Pour conclure, aucun des modèles envisagés (k-NN et random forest) ne fait mieux en sens du nombre d'erreur de classification que le modèle nul consistant à prédire la classe la plus représentée, "hold".

Les résultats que nous avons obtenus sont loin d'être surprenants. Une première observation concerne la fonction de perte. On peut s'interroger sur la pertinence d'une perte binaire. En effet, les erreurs de prédictions ne sont pas toutes équivalentes. Prédire un "buy"" ou un "sell" par un "hold", ne constitue pas une perte (sur le plan financier) similaire à prédire un "buy" par un "sell" ou l'inverse. Dans la réalité ces valeurs sont ordonnées et un "buy" est plus éloigné d'un "sell" que d'un "hold". Or, les modèles envisagés ne sont pas ordonnés. De façon générale, il aurait aussi pu apparaître plus pertinent de comparer les modèles obtenus avec un critère d'erreur personnalisé, tenant justement compte de ces différences d'importance entre les erreurs de prédictions.

De plus, la période observée (1970-2009) est très importante dans un secteur qui a subit de profondes mutations. La dynamique des marchés financiers des années 1970, avec des échanges d'actions "à la voix", est fort différente de celle de l'époque actuelle où la majorité des échanges sont le fruit d'algorithmes. Cependant, nos modèles ne prennent pas en compte cette dynamique temporelle (une observation de 1970 a autant de poids qu'une observation de 1999).

Enfin, les théories financières (efficience faible des marchés) expliquent qu'il n'est normalement pas possible de trouver des stratégies optimales en utilisant l'analyse technique. Cette théorie n'est pas toujours vérifiée mais il apparaît difficile de trouver des stratégies dégageant des bénéfices importants (on considère un seuil de décision de 10 % !) sur l'indice le plus observé et échangé du monde.

Annnexe

Pour modifier l'arbre de décision, on peut jouer sur 2 types de paramètres :

- Les critères d'arrêt "classiques" : minsplit (nombre d'observations minimum que doit contenir un noeud avant d'entreprendre un découpage supplémentaire), minbucket (nombre d'observations minimum que doit contenir une feuille), maxdepth (profondeur maximum)
- Le critère "pré-élagage"" ou autrement dit le paramètre cp : si le découpage suivant fait décroître l'erreur relative (c'est-à-dire relativement à l'erreur obtenue avec un arbre réduit à une seule feuille dans laquelle la décision correspond à la classe majoritaire) de moins d'un facteur de cp, alors il n'est pas effectué et l'expansion s'arrête. Notons que cette erreur relative est évaluée sur la partie de l'échantillon servant à construire l'arbre. Elle diminue donc mécaniquement à chaque découpage.

On constate ici que les critères d'arrêt "classiques" ne sont pas ceux qui expliquent l'arrêt de l'arbre à la racine, puisque la profondeur maximale n'est évidemment pas atteinte, ni le nombre d'observations minimum dans un noeud. C'est le critère de "pré-élagage" ou *complexity parameter* (cp) qui joue ici puisqu'il est par défaut fixé à 0.01 et on constate dans le summary que l'erreur relative n'est diminuée que de 0.00983 (< 0.01) si un découpage suivant a lieu.

On peut donc diminuer le paramètre cp pour obtenir un arbre plus complexe.

Une approche possible est de choisir cp par validation croisée :

```
set.seed(1234)
# 5-fold cross-validation to select cp
# from the set {1,...,10}
fold = sample(c(rep(1:5,each= dim(Tdata.train)[1] %/% 5), rep(5,dim(Tdata.train)[1] %% 5)) )
# creation des groupes B_v
params_to_test = seq(0,0.01,0.001)
cvpred = matrix(NA,nrow=dim(Tdata.train)[1],ncol=length(params_to_test)) # initialisation de la matrice
# des prédicteurs
for (k in 1:length(params_to_test))
  for (v in 1:5)
    {
    sample1 = Tdata.train[which(fold!=v),1:9]
    sample2 = Tdata.train[which(fold==v),2:9]
    class1 = Tdata.train[which(fold!=v),1]
    params_tree = rpart.control(cp = params_to_test[k])
    rt.signal = rpart(signal ~ ., data = sample1, method = 'class', control = params_tree)
    cvpred[which(fold==v),k] = predict(rt.signal,sample2, type="class")
  }
class = as.numeric(Tdata.train[,1]) # display misclassification rates for k=1:10
result = apply(cvpred,2,function(x) sum(class!=x)) # calcule l'erreur de classif.
result
```

```
##  [1] 2056 2004 2021 2056 2071 2088 2090 2097 2098 2098 2117
```

Parmis les 11 valeurs de cp testées ici (0.000, 0.001, 0.002, 0.003, 0.004, 0.005, 0.006, 0.007, 0.008, 0.009, 0.010), 0.001 est la valeur qui fournit les meilleures prédictions.

Néanmoins, en prenant une valeur aussi faible, il y a un fort risque de sur-apprentissage. On peut donc choisir 0.001, mais éventuellement essayer d'ajouter une étape d'élagage (ou *pruning*) ensuite.

```
set.seed(1234)
params_tree = rpart.control(cp = 0.001)
rt.signal2 = rpart(signal ~ ., data = Tdata.train[,1:9], method = 'class', control = params_tree)
rt.signal2
```

On constate effectivement que l'arbre obtenu a de nombreuses ramifications.

Comparons avec l'arbre de décision construit précédemment :

```
pred1 = predict(rt.signal1,Tdata.eval[,2:9], type="class")
print("Erreur de l'arbre de décision par défaut :")
```

```
## [1] "Erreur de l'arbre de décision par défaut :"
```

```
print(sum(Tdata.eval[,1]!=pred1))
```

```
## [1] 787
```

```
summary(pred1)
```

```
##    s    h    b
##    0 2430    0
```

```
pred2 = predict(rt.signal2,Tdata.eval[,2:9], type="class")
print("Erreur de l'arbre de décision avec cp = 0.001:")
```

```
## [1] "Erreur de l'arbre de décision avec cp = 0.001:"
```

```
print(sum(Tdata.eval[,1]!=pred2))
```

```
## [1] 1225
```

```
summary(pred2)
```

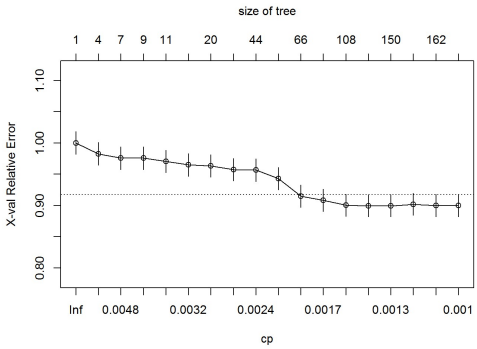
```
##    s    h    b
##  473 1398  559
```

Le second arbre s'avère moins performant que le premier (1225 erreurs mais moins d'erreurs que les modèle k-NN). Il reste trop complexe, ce qui entraîne un phénomène de surapprentissage.

Ajoutons donc une étape d'élagage en choisissant ici le critère d'élagage (paramètre cp) à partir des valeurs de cp (autrement dit des baisses d'erreurs à chaque découpage supplémentaire) constatées lors de la construction de l'arbre. Ceci peut être fait à l'aide des fonctions plotcp et printcp.

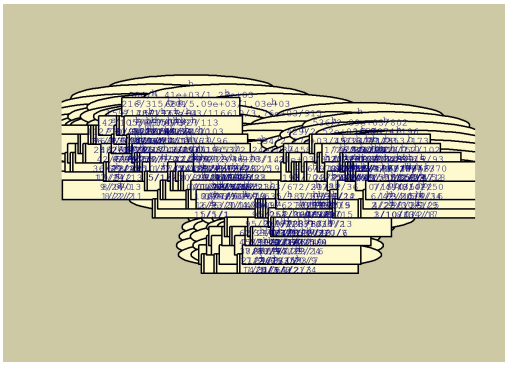
La courbe renvoyée par plotcp indique le taux de mauvaises classifications relativement au score d'origine (dans un arbre réduit à une seule feuille dans laquelle la décision correspond à la classe majoritaire), estimé par la validation croisée. Contrairement à l'erreur relative, qui est calculée sur l'échantillon d'entraînement, l'erreur estimée par validation croisée ne décroît pas mécaniquement. On choisit alors le paramètre d'élagage égal à la valeur de cp pour laquelle on a constaté l'erreur de validation croisée la plus faible lors de la construction de l'arbre.

```
set.seed(1234)
plotcp(rt.signal2)
```




```
pfit<- prune(rt.signal2, cp= rt.signal2$cptable[which.min(rt.signal2$cptable[,"xerror"]),"CP"])
```

L'arbre finalement obtenu reste très complexe et impossible à afficher clairement :



Comparons les erreurs de prédiction des 2 arbres de décisions finalement considérés (l'arbre par défaut et celui avec cp = 0.001 et élagage) :

```
pred1 = predict(rt.signal1,Tdata.eval[,2:9], type="class")
print("Erreur de l'arbre de décision par défaut :")
```

```
## [1] "Erreur de l'arbre de décision par défaut :"
```

```
print(sum(Tdata.eval[,1]!=pred1))
```

```
## [1] 787
```

```
summary(pred1)
```

```
##    s    h    b
##    0 2430    0
```

```
pred3 = predict(pfit,Tdata.eval[,2:9], type="class")
print("Erreur de l'arbre de décision avec cp = 0.001 puis élagage:")
```

```
## [1] "Erreur de l'arbre de décision avec cp = 0.001 puis élagage:"
```

```
print(sum(Tdata.eval[,1]!=pred3))
```

```
## [1] 1051
```

```
summary(pred3)
```

```
##    s    h    b
##  419 1570  441
```

Le nombre d'erreurs a diminué (1051 erreurs) mais reste supérieur au modèle nul. On constate ici le même problème qu'avec le modèle k-NN : la classe "hold" est prédite très souvent, à tel point que les observations dont la véritable classe est "buy" ou "sell" sont en majorité prédites comme appartenant à la classe "hold.