TP2 - kNN, Decision Trees and stock market returns

Apprentissage statistique
Chargé de TD : Christophe Denis

Pierre ALLAIN, Benoît CHOFFIN
22 novembre 2016

Question 1. La commande `apply(cvpred,2,function(x) sum(class!=x))` utilisée à la page 2 permet d'évaluer l'erreur de classification de l'algorithme de k-plus proches voisins utilisé. Plus précisément, on applique alors sur les colonnes de `cvpred` (d'où le `"2"` en argument) la fonction définie en interne et qui somme le nombre d'éléments pour lesquels la vraie classe n'est pas celle prédite. La classe prédite par le prédicteur des k-plus proches voisins pour chaque observation du jeu de données d'entraînement (`train`) est tout d'abord stockée dans les colonnes de `cvpred` (chacune de celles-ci correspond à un k des k-plus proches voisins), afin d'évaluer son impact sur l'erreur de prédiction ; les vraies valeurs sont quant à elles dans `class`. Ceci renvoie ce type d'output :

```
[1] 5 7 4 6 4 6 6 5 3 4
```

Chaque élément correspond donc au nombre d'erreurs de prédiction pour chaque k.

Question 2. Si l'on relance deux fois cette commande, les résultats ne sont pas nécessairement les mêmes, comme en atteste l'exemple suivant :

```
[1] 5 6 4 4 3 4 4 6 4 4
```

```
[1] 4 8 5 8 7 6 6 5 4 5
```

Ceci est en partie dû au fait que lors de l'étape de construction des 5 *folds* pour la validation croisée, le partitionnement se fait de manière aléatoire, avec la fonction `sample` de R. Dès lors, les groupes sur lesquels on évalue l'erreur de prédiction changent lorsqu'on relance une deuxième fois le code donné dans l'énoncé. Donc, la prédiction avec le prédicteur

des k-plus proches voisins peut elle aussi changer : cela peut par exemple arriver dans le cas où, d'une fois sur l'autre, une partie des k-plus proches voisins d'une observation d'un échantillon de test se retrouvent dans ce même échantillon et ne contribuent donc pas à la prédiction de la classe de cette observation.

Une autre raison pour laquelle les résultats ne sont pas identiques d'une fois sur l'autre est la suivante : dans le cadre de l'algorithme des k-plus proches voisins, lorsqu'il y a égalité entre les k voisins du point à prédire, la décision de la classe à prédire est décidée par un tirage au sort entre classes concurrentes. Il est donc possible d'obtenir des classes prédites différentes (et donc, une erreur différente) si l'on relance deux fois cet algorithme.

En combinant 100 résultats différents, il est alors possible d'étudier la distribution du nombre d'erreurs en fonction de k. Un boxplot des erreurs de prédiction en fonction de k pourrait permettre de visualiser efficacement cette distribution en fonction de k et ainsi de choisir le k optimal. Une méthode plus simple consisterait à utiliser la médiane ou la moyenne du nombre de prédictions pour chaque k sur ces 100 essais et de choisir alors le k dont la statistique retenue minimise le nombre d'erreurs de prédiction. Toutefois, il conviendrait alors de tenir également compte de l'écart-type du nombre d'erreurs pour chaque k, car une volatilité trop importante serait nuisible.

Question 3.

(0) La première commande renvoie une erreur car on fait appel à une fonction (`"candleChart"`) sans avoir chargé le package correspondant. Il faut donc lancer la commande suivante au préalable : `library(quantmod)`.

(1) Voici le code utilisé pour ajouter aux chandeliers japonais la courbe des valeurs médianes de (C_i, H_i, L_i) :

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

La figure suivante en est le résultat :

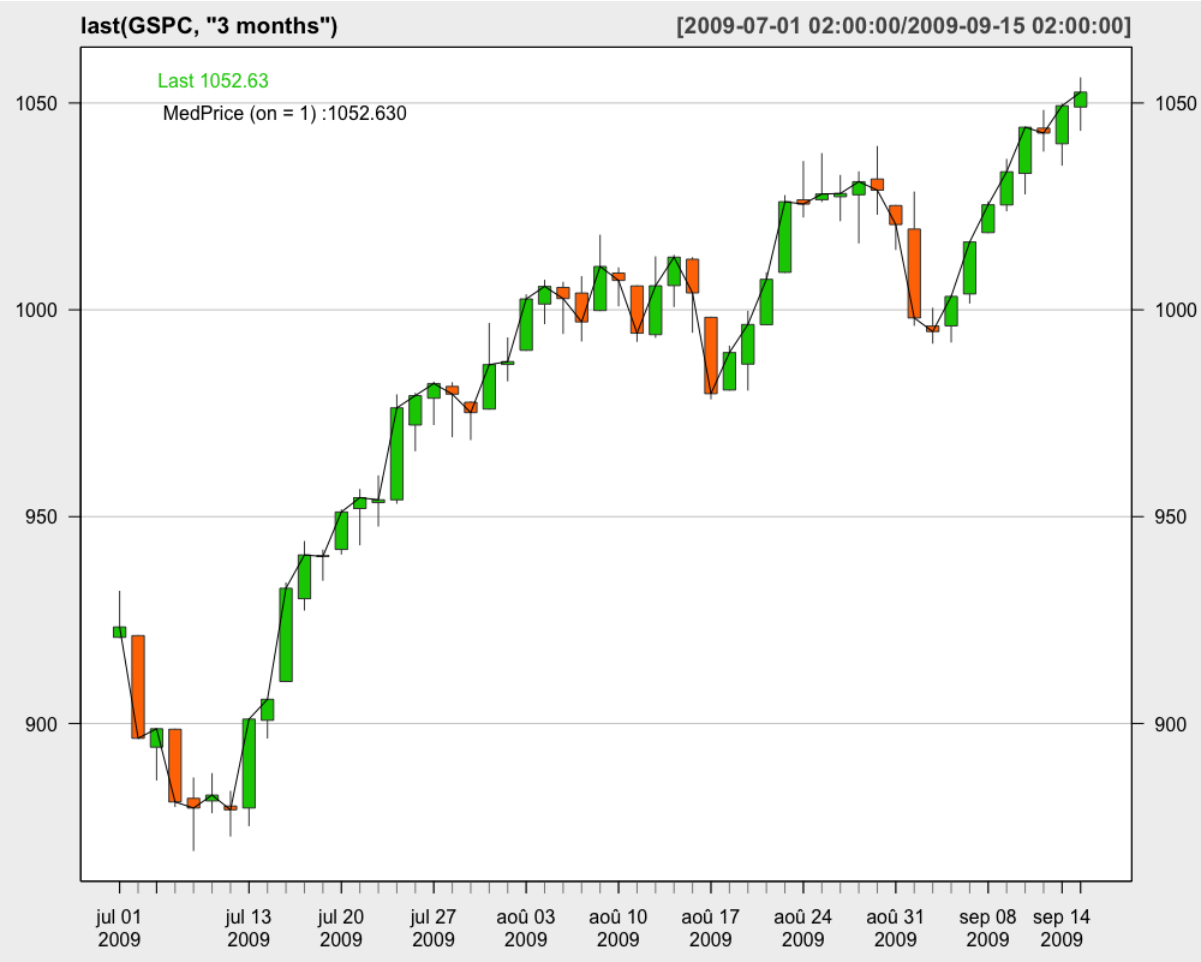

FIGURE 1. *Chandeliers japonais et courbe des valeurs médianes de* (C_i, H_i, L_i)

Si l'on enlève l'argument `on=1` de la fonction `addAvgPrice`, la fonction `avgPrice` est alors représentée non pas avec les chandeliers japonais, mais sur un graphique en-dessous du graphique principal :

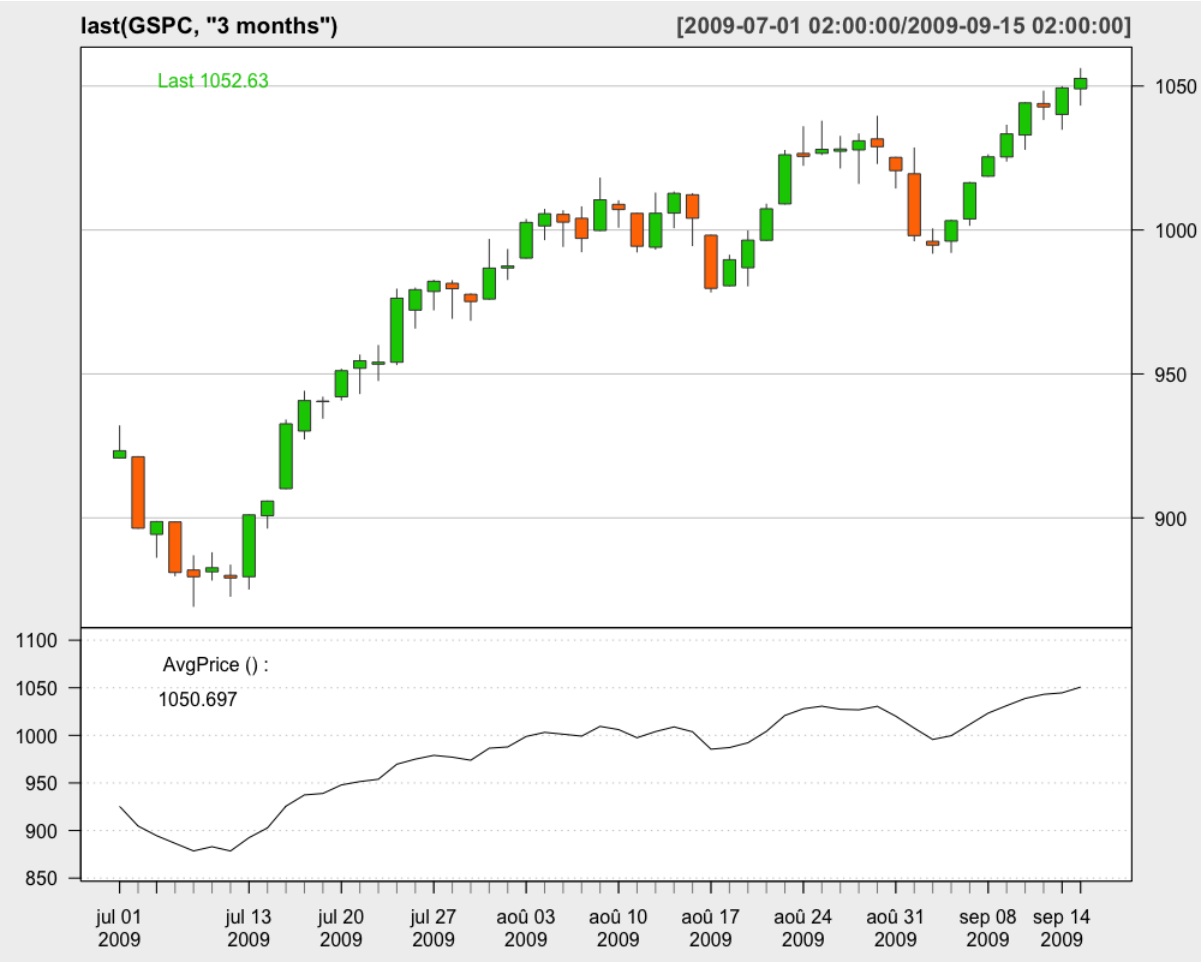

FIGURE 2. *Chandeliers japonais et courbe des valeurs moyennes de* (C_i, H_i, L_i)

(2) L'option `training.per` sert à spécifier les dates utilisées pour l'apprentissage du modèle de séries temporelles. L'option `importance`, quant à elle, provient de la fonction `randomForest` et indique que l'importance des variables doit être calculée (si on a mis `importance=TRUE`).

(3) La figure suivante représente l'importance des variables du modèle. Plus précisément, l'axe des abscisses correspond au pourcentage d'augmentation de l'erreur quadratique moyenne (MSE) lorsque cette variable est retirée du modèle. Une valeur élevée signifie donc que cette variable joue un rôle majeur dans la prédiction finale.

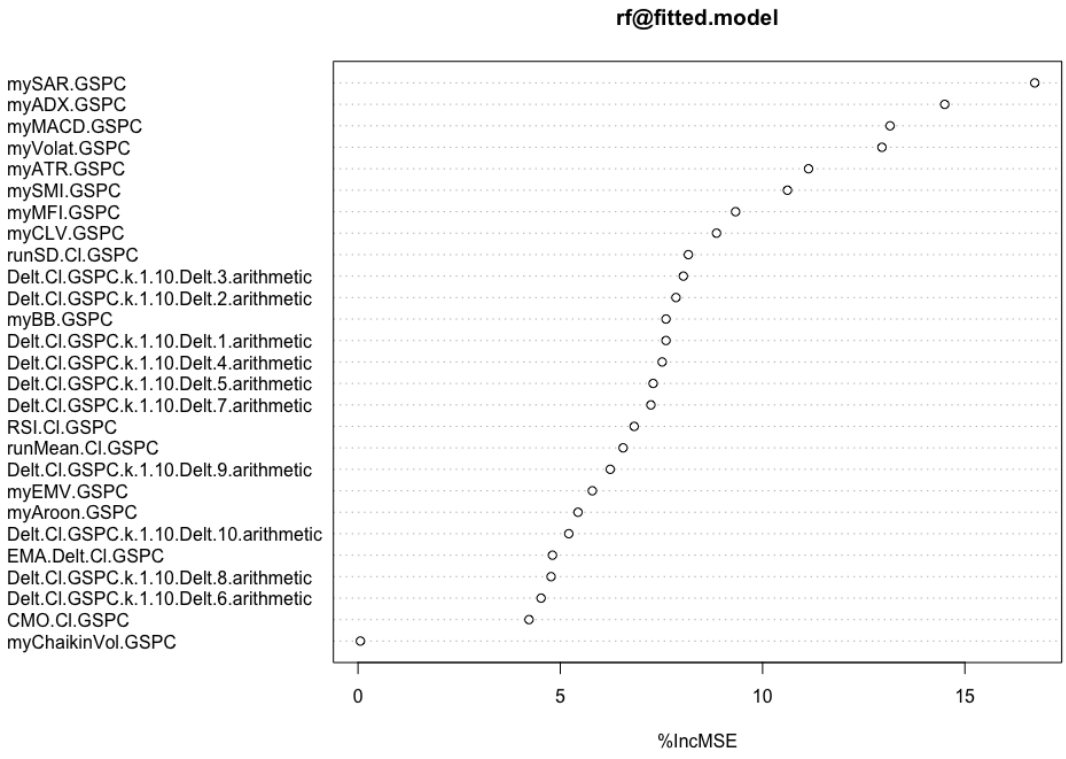

FIGURE 3. *Représentation de l'importance des variables du modèle*

D'après ce graphique, les 8 variables les plus pertinentes dans ce modèle sont (par ordre décroissant d'importance) : `mySAR.GSPC`, `myADX.GSPC`, `myMACD.GSPC`, `myVolat.GSPC`, `myATR.GSPC`, `mySMI.GSPC`, `myMFI.GSPC` et `myCLV.GSPC`.

(4) Le code suivant a servi à créer le nouveau modèle `data.model` pour expliquer la variable `T.ind(GSPC)` à partir des 8 variables les plus pertinentes précédentes :

```
>data.model = specifyModel(T.ind(GSPC) ~ mySAR(GSPC) + myADX(GSPC) +
>                          myMACD(GSPC) + myVolat(GSPC) + myATR(GSPC) +
>                          mySMI(GSPC) + myMFI(GSPC) + myCLV(GSPC))
```

(5) La fonction `na.omit` utilisée dans le code de l'énoncé sert à retirer toutes les observations du *dataset* (auquel on applique cette fonction) ayant au moins une valeur manquante (peu importe la variable). Si on a un `NA` à un endroit dans la base de test, alors on ne pourra pas prédire la variable dépendante pour cette observation car la règle de décision ne pourra pas s'appliquer. En revanche, c'est moins problématique pour la base d'entraînement puisque l'algorithme peut alors ne pas tenir compte de ce `NA` dans l'étape d'apprentissage.

Question 4. Nous allons d'abord utiliser l'algorithme des kNN pour prédire la variable `signal` sur l'échantillon de test. Il nous faut d'abord choisir le paramètre k : le nombre de

voisins à prendre en compte pour la prédiction. En nous inspirant de ce qui est présenté en introduction du TD, nous avons choisi d'utiliser la méthode de validation croisée des *V-folds* pour déterminer k. Cette méthode consiste à diviser la base d'entraînement en sous-échantillons, qui serviront chacun leur tour de base de test ; le reste représentant la base d'entraînement amputée d'un sous-échantillon. Après avoir déterminé l'erreur de prédiction pour chaque sous-échantillon, on calcule finalement la somme de ces erreurs qui correspond à la valeur de k testée. On retient le nombre de voisins qui minimise l'erreur de prédiction.

Dans notre cas, nous choisissons de diviser la base d'entraînement en 9 sous-groupes. C'est une quantité raisonnable, qui permet une exécution rapide de l'algorithme et qui a surtout l'avantage de diviser le nombre d'entrées de notre base `Tdata.train`. On détermine la valeur de k la mieux adaptée selon la validation croisée parmi les entiers allant de 1 à 10.

On obtient alors les résultats suivants :

```
[1] 1580 1777 1713 1739 1773 1791 1815 1841 1829 1870
```

La valeur $k = 1$ semble être le meilleur choix quant au nombre de voisins à prendre en compte. Comme cela a été précisé dans la question 2, les résultats changent légèrement à chaque nouvelle exécution de l'algorithme à cause du partitionnement aléatoire engendré par la fonction `sample`. Après avoir relancé plusieurs fois la validation croisée, on retient notre choix de départ pour la valeur de k.

Après avoir choisi la valeur de k à l'aide de la validation croisée, nous lançons l'algorithme sur la véritable base de test. La matrice de confusion regroupe les résultats de la prédiction. Chaque ligne de la matrice représente le nombre d'occurrences d'une classe estimée, tandis que chaque colonne représente le nombre d'occurrences d'une classe réelle :

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

On observe une grande quantité de signaux mal classés. On peut déterminer l'erreur globale de prédiction en divisant le nombre de signaux mal classés par le nombre total de signaux :

$$\text{Taux de mal classés} = \frac{409 + 52 + 259 + 241 + 77 + 277}{2430} = 54\%$$

Le taux d'erreur s'avère très élevé : plus de la moitié des observations sont mal classées. Nous nous retrouvons dans un cas typique de sur-apprentissage : en choisissant la valeur de k la plus faible possible, notre prédiction devient très sensible au bruit de notre base d'entraînement. L'algorithme des kNN n'arrive pas à prédire correctement la variable `signal` à partir de l'apprentissage sur notre base `Tdata.train`. Bien qu'il minimise l'erreur de prédiction dans la validation croisée, le choix $k = 1$ n'est sans doute pas le plus judicieux.

```
>#Prédiction de signal avec le knn
># On peut utiliser une validation croisée comme au début du td
>
>library(class)
># creation des groupes : 9 semble raisonnable
>fold = sample(rep(1:9,each=nrow(Tdata.train)/9))
>#(9 divise le nombre d'entrées de la base d'entraînement,
>on aurait aussi pu prendre 6)
>
>cvpred = matrix(NA,nrow=nrow(Tdata.train),ncol=10) # initialisation de la matrice
># des prédicteurs
>
>for (k in 1:10)
>  for (v in 1:9)
>  {
>    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$signal)
>
># display misclassification rates for k=1:10
>apply(cvpred,2,function(x) sum(class!=x)) # calcule >l'erreur de classif.
>
>## Application de l'algorithme kNN avec le k choisi
>pred = knn(Tdata.train[,2:9],Tdata.eval[,2:9], Tdata.train$signal , k = 1)
># display the confusion matrix
>table(pred,Tdata.eval$signal)
>(sum(Tdata.eval$signal!=pred))/nrow(Tdata.eval)
```

Question 5.

On utilise une deuxième méthode de classification – l'arbre de décision – dont on va comparer les résultats à ceux obtenus avec la méthode des kNN. À la différence de l'algorithme des k-plus proches voisins, la partition générée par un arbre de décision est fondée, outre les variables explicatives, sur les étiquettes observées dans la base d'entraînement. La construction a lieu en trois étapes : une première étape d'initialisation (on fixe un nœud racine), une deuxième dite d'expansion (on créer des nœuds fils pour chaque nœud qui ne vérifie pas la condition d'arrêt) et enfin l'élagage (supprime une branche de l'arbre si cette opération ne détériore pas significativement le risque estimé).

On utilise dans notre cas la fonction `rpart` pour créer un arbre de décision sur notre base d'entraînement. On choisit l'option `method=class` car il s'agit ici d'un arbre de classification. On utilise également un paramètre de contrôle pour modifier la taille de l'arbre créé : le paramètre de complexité. Modifier ce paramètre permet de jouer sur le critère d'arrêt et ainsi obtenir un nombre de nœuds raisonnable au vu de notre étude. Nous avons fixé `cp=0.0035` pour obtenir une profondeur d'arbre convenable. On obtient alors l'arbre suivant :

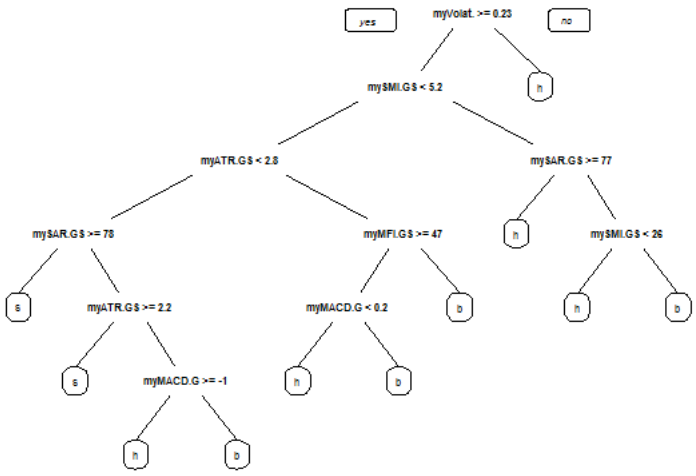

FIGURE 4. *Représentation de l'arbre de décision.*

On représente la matrice de confusion associée à l'arbre :

```
pred.tab    s    h    b
       s    0    0    0
       h  396 1601  291
       b   42   42   58
```

On peut voir que le signal de vente "sell" n'est jamais prédit sur l'ensemble de la base de test. Tous les signaux "sell" sont donc mal classés, avec une grande partie d'entre eux (396 sur 438) prédits en tant que signaux "hold". La modalité "hold" présente au contraire un grand nombre de signaux bien classés (1601). La modalité "buy" est quant à elle caractérisée par davantage de signaux mal classés que bien prédits : 58 prédictions justes sur un total de 349.

Enfin on calcule l'erreur de prédiction sur la base de test pour pouvoir le comparer avec celui obtenu avec les kNN. On obtient un taux d'erreur de 31.72 %. Bien que ce score reste décevant, il est nettement meilleur que celui obtenu à partir du modèle précédent – on obtenait un taux d'erreur de 54%. On peut donc affirmer que la classification renvoyée par l'arbre de décision est meilleure que celle du modèle des kNN quant à la prédiction du signal de trading. Le nombre de voisins constituait le paramètre fondamental du modèle précédent. Ici, il s'agit du paramètre de complexité "cp" ; il serait judicieux de l'optimiser pour obtenir de meilleurs prédictions.