Modèles prédictifs de machine learning appliqués à la finance et utilisation du calcul parallèle

Université Paris 1 Panthéon-Sorbonne, Master 1 Econométrie-Statistiques (MoSEF)

Ouriane Aïssou, Achraf Khallou, Etienne Mellier

Avril 2018

Résumé

Cet article présente quelques possibilités d'application des modèles d'apprentissage statistique à la prévision financière. L'objectif est double. Le premier est d'effectuer des prévisions sur l'évolution positive ou négative de l'indice français du CAC 40 en fonction de plusieurs autres indices boursiers à l'aide des techniques de forêt aléatoire et des k plus proches voisins. Le second est de montrer l'utilité dans ces conditions du calcul parallèle qui permet d'optimiser la puissance de calcul de la machine lors du lancement des algorithmes. Cette technique sera présentée en première partie.

Table des matières

1 Méthode du calcul parallèle

Il est parfois nécessaire d'optimiser le temps d'exécution des calculs d'un programme R lorsque le traitement des calculs paraît trop lent pour l'utilisateur. Pour cela il peut être judicieux d'utiliser le calcul parallèle.

1.1 Principe du calcul parallèle : intéret, terminologie et fonctionnement

Le calcul parallèle a pour objectif d'effectuer des calculs informatiques plus rapidement tout en économisant de l'énergie grâce à l'exploitation simultanée de plusieurs unités de calcul. Contrairement au calcul séquentiel qui exécute une seule série d'instructions sur une unité de calcul, le calcul parallèle fonctionne en trois étapes :

- Il sépare un calcul informatique en plusieurs séries d'instructions indépendantes
- Il calcule parallèlement (en même temps) les séries d'instructions sur plusieurs unités de calcul
- Il réunit les résultats des calculs et les renvoie.

Pour utiliser le calcul parallèle, il faut donc avoir accès à plusieurs unités de calcul. On peut retrouver ces unités de calculs sur CPU (Central Processit Unit) ou GPU (Graphical Processing Unit), que l'on appelle aussi processeur. On se concentrera ici plus particulièrement sur le calcul parallèle sur CPU, bien plus étudié que sur GPU.

On divise les composantes d'un ordinateur en deux catégories : les composantes physiques et les composantes logicielles. Les composantes physiques impliquées dans le calcul parallèle sont l'espace de stockage et les processeurs.

Concernant l'espace de stockage, il faut exécuter des calculs sur des données générant par la suite d'autres données, qui doivent nécessairement être stockées. Habituellement, l'espace de stockage peut se situer dans :

- le disque dur : accès plus lent que la mémoire vive, grande capacité de stockage, stockage permanent
- la mémoire vive (RAM : Random Access Memory) : accès rapide plus que le disque, plus faible capacité, stockage temporaire
- la mémoire cache : très rapide d'accès, petite capacité de stockage temporaire, située dans le processeur

Quand un processeur exécute un programme, il doit nécessairement garder des données en mémoire de manière temporaire. Il se sert d'abord de sa mémoire cache, et ensuite de la mémoire vive si la cache ne suffit pas. Mais si le programme à exécuter demande plus d'espace de stockage que celle de la mémoire vive, il peut arriver que l'exécution soit arrétée, ne réponde plus ou bien que le programme se serve de l'espace sur le disque dur, ralentissant ainsi les calculs.

Le processeur a pour rôle de lire et exécuter les instructions d'un programme. Il est généralement divisé en plusieurs unités de calcul que l'on appelle cœurs (on parle de processeurs multi-coeurs). Ainsi, les processeurs multi-coeurs permettent de faire du calcul parallèle en utilisant un seul ordinateur, qui exploite alors plusieurs cœurs. Les cœurs exécutent chacun plusieurs séries d'instructions de manière séquentielle, autrement dit les unes après les autres. Mais un cœur (physique) peut en fait exécuter plus d'une série d'instruction : celui-ci peut-être divisé en plusieurs cœurs dits logiques.

1.2 Exemple d'application : utilisation du package *parallel*

L'objectif de cet exemple est de montrer que le calcul parallèle est plus rapide que le calcul séquentiel.

On commence tout d'abord par charger le package parallel, basé sur l'utilisation des fonctions de la famille apply, et on demande grâce à R le nombre de coeurs présents. Ici on détecte 2 coeurs physiques divisés en 4 coeurs logiques. On crée ensuite une grappe en laissant au moins un coeur libre de telle sorte que l'ordinateur ne soit pas trop ralenti.

```
library(parallel)
coeur <- detectCores()
coeur
```

```
## [1] 4
```

```
cl <- makeCluster(coeur - 1)
```

Pour rester simple, on souhaite transformer des nombres en leur carré, leur racine carrée et leur exponentielle. On crée pour cela une liste de nombre allant de 1 à 1 million pour chaque calcul.

Calcul séquentiel

```
system.time(
nopar <- sapply(as.character(1:1000000),
function(exponent){
x <-as.numeric(exponent)
c(carré = x^2, racine = x^(1/2),
exponentiel = 2^(x))
}
))
```

```
##    user  system elapsed
##  16.699   0.405  17.445
```

On utilise ici la fonction sapply qui applique les fonctions carré, racine carrée et exponentielle à chaque éléments de la liste.

Calcul parallèle

```
system.time(
par <- parSapply(cl, as.character(1:1000000),
function(exponent){
x <-as.numeric(exponent)
c(carré = x^2, racine = x^(1/2),
exponentiel = 2^(x))
}
))
```

```
##    user  system elapsed
##   8.879   0.692  13.231
```

On fait maintenant appel à la fonction parSapply provenant du package parallel. C'est la version parallèle de la fonction sapply. Elle prend en compte la grappe créée précédemment afin de distribuer les calculs sur les coeurs que l'on a choisi d'utiliser.

On observe alors que le calcul parallèle est beaucoup plus rapide que le calcul séquentiel.

2 Analyse Statistique

2.1 Introduction

Dans cette partie nous construisons un modèle de prévisions du CAC 40 que nous allons tester. Pour cela, nous disposons d'une base de données contenant plusieurs agrégats macroéconomiques. L'objectif est de construire un modèle d'apprentissage statistique [1] qui va prévoir si le CAC va augmenter ou non.

2.2 Traitement et analyse de la base de données

On importe la table.

```
chemin <- "/Users/Ouriane/Desktop/data_challenge/database.csv"
brut <- read.csv(chemin, header = TRUE, sep=";", dec=",", na.strings = "#N/A")
```

Descriptif des variables

Code variable	Descriptif
CAC.Index	Indice CAC 40
SPX.Index	Indice Standards and poor's
UKX.Index	Indice Footsie
DAX.Index	Indice du DAX
INDU.Index	Indice DOW Jones
EUR.curncy	Change euro
EURJPY.curncy	EUR/YEN
EURGBP	EUR/GBP
GSEACII.Index	Inflaiton euro
GSUSCII.Index	Inflation US
GSJPCII.Index	Inflation Japon
GOLDS.Comdty	Prix de l'or
CL1.COMB.Comdty	Prix du baril de pétrole
EUR003M.Comdty	Taux euribor
US0003M.Index	Taux libor USD
BP0003M.Index	Taux libor GPD
JY0003.Index	Taux libor JAP

On peut avoir une vue d'ensemble de la table avec la fonction *str()*.

```
str(brut)
```

```
## 'data.frame':    4696 obs. of  19 variables :
##  $ Date           : Factor w/ 4696 levels "01/01/2001","01/01/2002",.. : 1733 1274 1119 964 809 653 
##  $ CAC.Index      : num  5277 5274 5254 5188 5170 ...
##  $ SPX.Index      : num  2257 2262 2223 2199 2201 ...
##  $ UKX.Index      : num  8134 8124 8078 8016 8004 ...
##  $ DAX.Index      : num  12418 12347 12356 12245 12114 ...
##  $ NKY.Index      : num  166 163 163 162 163 ...
##  $ INDU.Index     : num  20422 20568 20209 19999 20079 ...
##  $ EUR.curncy     : num  1.23 1.23 1.23 1.24 1.24 ...
##  $ EURJPY.curncy  : num  131 131 131 132 132 ...
##  $ EURGBP.curncy  : num  0.887 0.889 0.891 0.893 0.893 ...
```

1. "Machine learning" en anglais.

```
##  $ GSEACII.Index  : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ GSUSCII.Index  : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ GSJPCII.Index  : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ GOLDS.Comdty   : num  1073 1074 1073 1069 1077 ...
##  $ CL1.COMB.Comdty : num  49.8 50.4 48.8 49.3 50.5 ...
##  $ EUR003M.Index  : num  -0.327 -0.327 -0.327 -0.327 -0.327 -0.327 -0.327 -0.327 -0.327 -0.328 ...
##  $ US0003M.Index  : num  2.11 2.09 2.07 2.06 2.05 ...
##  $ BP0003M.Index  : num  0.602 0.601 0.6 0.601 0.596 ...
##  $ JY0003.Index   : num  -0.0547 -0.0563 -0.0503 -0.0537 -0.0537 ...
```

On crée une fonction nommée *nettoyage_kontest* qui va permettre de traiter la base de manière automatisée. Egalement, les données étant indicées par le temps, on charge la librairie *xts* qui permet de déclarer une base de données en series temporelles.

```
library(xts)
```

```
brut <- read.csv("/Users/Ouriane/Desktop/data_challenge/database.csv",
                 header = TRUE, sep=";", dec=",", na.strings = "#N/A")
format_date <- "%d/%m/%Y"
seuil_mensualite <- 90
```

On transforme la base en données xts, c'est-à-dire en données temporelles.

```
data_xts <- xts(brut[,-1], order.by=as.Date(brut$Date,format_date))
```

```
## Warning in strptime(x, format, tz = "GMT") : unknown timezone 'zone/tz/
## 2018c.1.0/zoneinfo/Europe/Paris'
```

```
n <- ncol(brut)
nbl <- nrow(data_xts)
nbc <- ncol(data_xts)
```

On analyse les données manquantes en créant un vecteur qui va indiquer la part de donénes manquantes.

```
donnee_na <- rep(0, nbc)
names(donnee_na) <- names(data_xts)
donnee_na <- apply(data_xts, 2, function(x) sum(is.na(x))/length(x)*100)
donnee_na
```

```
##       CAC.Index       SPX.Index       UKX.Index       DAX.Index
##        1.980409        3.577513        3.130324        2.597956
##       NKY.Index      INDU.Index      EUR.curncy   EURJPY.curncy
##        5.898637        3.577513        0.000000        0.000000
##   EURGBP.curncy   GSEACII.Index   GSUSCII.Index   GSJPCII.Index
##        0.000000       93.568995       93.568995       95.634583
##    GOLDS.Comdty CL1.COMB.Comdty   EUR003M.Index   US0003M.Index
##        0.000000        3.726576        1.916525        3.130324
##   BP0003M.Index    JY0003.Index
##        3.130324        3.130324
```

On cherche les premières valeurs pour lesquelles il y a au moins deux non valeurs manquantes de suite.

```
premier_na <- rep(0, nbc)
names(premier_na) <- names(data_xts)
premier_na <- apply(data_xts, 2, function(x) which.min(is.na(x)))
premier_na
```

```
##       CAC.Index       SPX.Index       UKX.Index       DAX.Index
##               1               1               1               1
```

```
##       NKY.Index      INDU.Index      EUR.curncy   EURJPY.curncy
##               1               1               1               1
##   EURGBP.curncy   GSEACII.Index   GSUSCII.Index   GSJPCII.Index
##               1             711             733            1951
##    GOLDS.Comdty CL1.COMB.Comdty   EUR003M.Index   US0003M.Index
##               1               1               1               1
##   BP0003M.Index    JY0003.Index
##               1               1
```

```
if (sum(premier_na > 1) > 0) {
  # Supprime les premieres observations et verifie frequence na
  data_xts_2 <- na.trim.default(data_xts, sides = "left")
  start(data_xts_2)
    }else{
    data_xts_2 <- data_xts
}
```

```
## [1] "2007-10-01"
```

```
donnee_na_2 <- apply(data_xts_2, 2, function(x) sum(is.na(x))/length(x)*100)
# Modifie la frequence des taux d'inflations en journalière
  if (sum(donnee_na_2 > seuil_mensualite)) {
    #Donne l'indice dans la base des series à modifier
    serie_mensuel <- which(donnee_na_2 > seuil_mensualite)
    #Effectue les modifications en prenant la valeur precedante
    data_xts_2[, serie_mensuel] <- na.locf(data_xts_2[, serie_mensuel])
  }
```

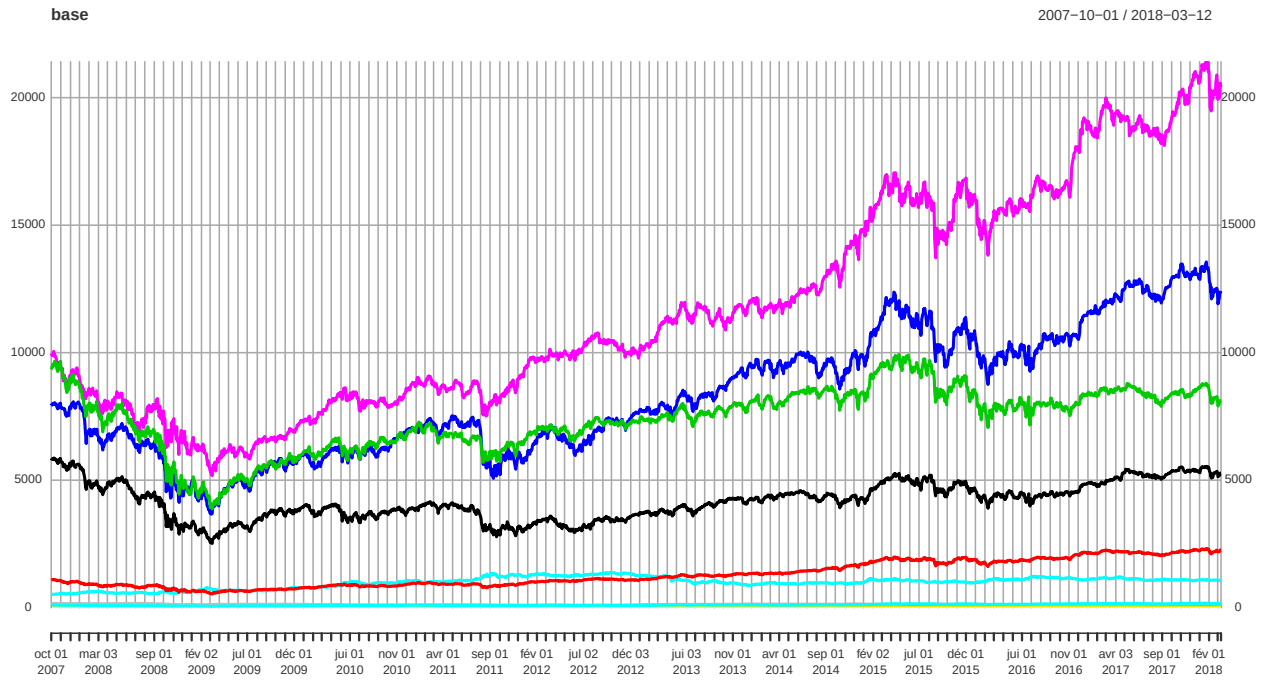
On interpole le reste des valeurs manquantes.

```
base <- na.approx(data_xts_2)
```

Notre base est donc nettoyée et utilisable.

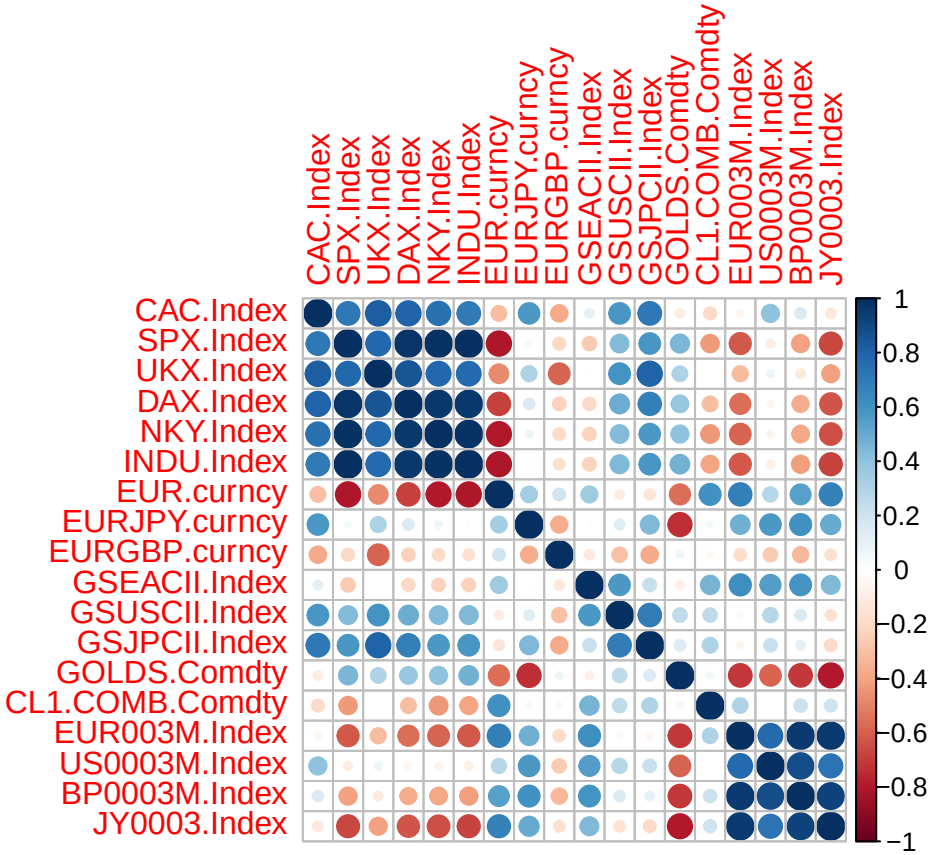
2.3 Analyse exploratoire

```
plot(base)
```




```
library(corrplot)
```

```
corr <- cor(base)
corrplot(corr)
```



On remarque que le CAC est très corrélé avec le SPX, le UKX, le DAX et le NKY. On choisis donc ces

variables pour les analyser d'un peu plus près.

```
base_2 <- base[,c(1:5)]
```

2.4 Modélisation

Notre but est de prévoir si le CAC 40 va évoluer à la baisse ou à la hausse. Pour cela nous allons utiliser deux méthodes d'apprentissage statistique : la méthode des k plus proches voisins et celle de la foret aléatoire. L'apprentissage statistique consiste à créer un algorithme qui va s'entrainer sur la majeure partie de la base pour ensuite etre testé sur l'autre partie. Dans notre conxtexte, nous allons créer une variable binaire qui va déclarer en période t-1 si le CAC 40 va baisser ou augmenter en période t. L'algorithme devient un peu devin, mais en réalité, cette configuration est tout à fait volontaire, puisque nous voulons que l'algorithme capte une tendance par le bias de l'entrainement. Ensuite, nous allons tester notre alogrithme pour savoir s'il arrive à prédire l'évolution à la hausse ou à la baisse de l'indice français.

Nous créeons une variable nommée *bool_cac* qui déclare en t-1 de la prédiction si le CAC va augmenter (TRUE) ou baisser (FALSE) pour la période t.

```
temp <- as.logical(base_2[1,] > 50)
# Differencie
base_3 <- diff.xts(base_2[, temp])

# Creation du booléen
Bool_CAC <- lag.xts(base_3$CAC.Index, k= -1)
Bool_CAC <- Bool_CAC > 0
Bool_CAC <- Bool_CAC[-1]

# Modifie la base
base_3 <- na.trim(base_3)
base_2 <- base_2[-1, ]
base_2[, temp] <- base_3

base_2$CAC.Index.1 <- Bool_CAC

base_2 <- na.trim(base_2)
```

2.4.1 Selection des horizons d'entrainement et de test

On sélectionne la base d'entrainement du modèle ainsi que sa période de test.

```
periode_train <- "2008/2017-10-12"
periode_test <- "2017-10-13/2018-03-12"
```

2.4.2 Méthode de la foret aléatoire

```
library(randomForest)
```

Ici on applique la méthode de la foret aléatoire.

```
base_train <- as.data.frame(base_2[periode_train, ])
base_test <- as.data.frame(base_2[periode_test, ])

base_train$CAC.Index.1 <- as.factor(base_train$CAC.Index.1)
```

```
base_test$CAC.Index.1 <- as.factor(base_test$CAC.Index.1)

system.time(
fit <- randomForest(CAC.Index.1 ~., data=base_train, ntree = 1500,
                        mtry = 3, importance = TRUE)
)
```

```
##    user  system elapsed
##   8.752   0.264   9.105
```

```
prev <- predict(fit, base_test)
```

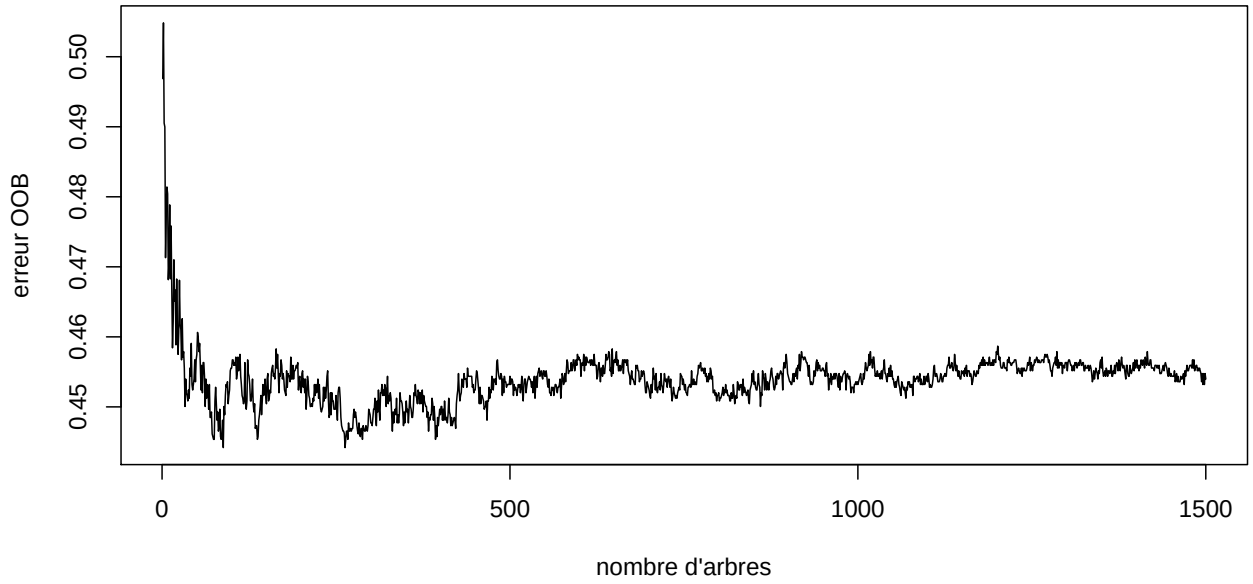
On imprime la matrice de confusion.

```
print((t <- table(prev, base_test$CAC.Index.1)))
```

```
##
## prev  0  1
##    0 28 17
##    1 24 37
```

On imprime le taux d'erreur pendant la période d'entrainement.

```
plot(fit$err.rate[, 1], type = "l", xlab = "nombre d'arbres", ylab = "erreur OOB")
```



On imprime également le taux de prédition du modèle lors de la phase de test cette fois ci.

```
print(sum(diag(t)) / sum(t))
```

```
## [1] 0.6132075
```

Nous voyons que le modèle de forêt aléatoire prédit avec un taux précision d'environ 60%.

On imprime également le taux d'erreur "Out of the bag".

Application du calcul parallèle

```
library(foreach)
library(doParallel)
```

Nous appliquons cette fois ci la meme méthode que précedemment à ceci près que nous appliquons la parralélisation du calcul. Nous séparons le calcul des 1500 arbes sur trois coeurs. Donc chaque coeur aura 500 arbres de décisions à calculer chacun.

```
ncoeurs <- detectCores()
ncoeurs
```

```
## [1] 4
```

```
kontest <- makeCluster(ncoeurs -1)

registerDoParallel(kontest)
system.time(
rf <- foreach(ntree=rep(500, 3), .combine= combine, .packages='randomForest') %dopar% {
      randomForest(CAC.Index.1~.,
                data=base_train,
                ntree=ntree, importance=TRUE, keep.forest=TRUE)
})
```

```
##    user  system elapsed
##   1.088   0.751   5.941
```

```
stopCluster(kontest)
```

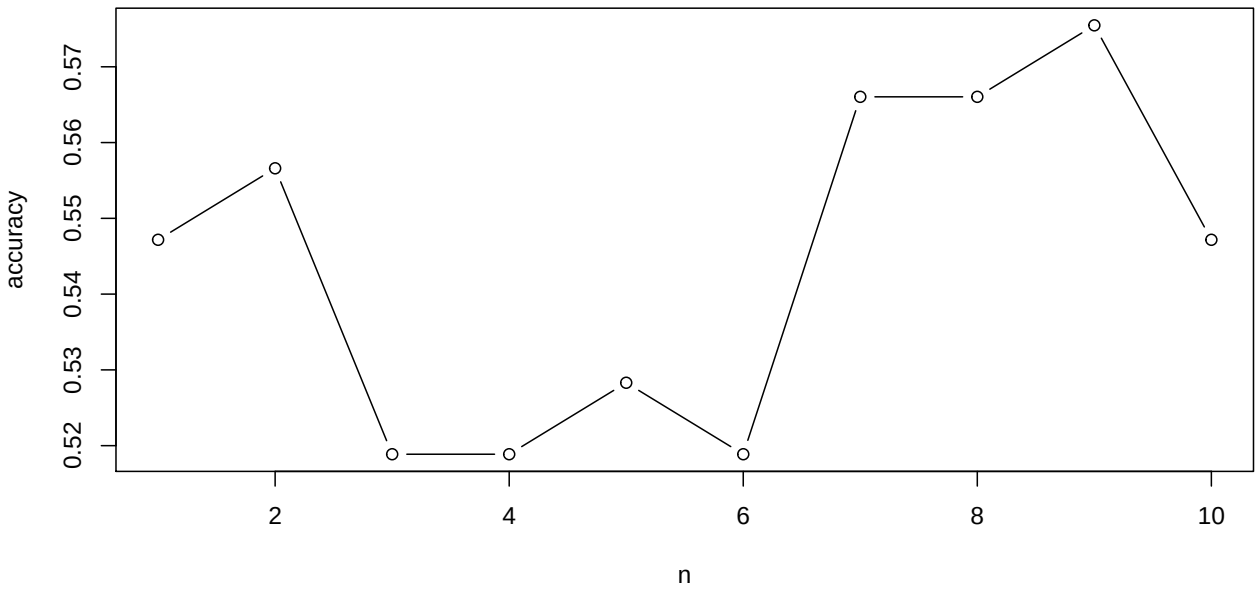
2.4.3 Méthode des k plus proches voisins

```
library(class)
library(dplyr)
library(lubridate)
```

```
indice_train <- base[periode_train, ]
indice_test <- base[periode_test, ]

lt <- nrow(indice_train)
ct <- ncol(indice_train)
lp <- nrow(indice_test)
cp <- ncol(indice_test)

perf_CAC <- na.omit(diff(base[, 1])/lag(base[, 1])*100)
bool_CAC <- perf_CAC > 0

bool_train_lag <- lag.xts(bool_CAC[periode_train], k = -1)
bool_test_lag <- lag.xts(bool_CAC[periode_test], k = -1)

n <- 1:10
accuracy <- rep(0, 10)

prediction <- rep(0, lp)
resultat <- matrix(0, nrow = lp, ncol = 10)

for (i in n) {
  prediction <- knn(indice_train[-lt, ], indice_test, bool_train_lag[-lt], k=i)
  resultat[, i] <- prediction
  table(prediction, bool_test_lag)
  accuracy[i] <- mean(prediction[-lp] == as.factor(bool_test_lag[-lp]))
```

```
}
```

```
plot(n, accuracy, type = 'b')
```



Nous voyons que le taux de prédiction maximal du modèle des k plus proches voisins est de 57% ce qui est moins bien que notre modèle foret aléatoire.