Projet Machine Learning: Reconnaissance des chiffres manuscrits

Yamina BOUBEKEUR et Amina GHOUL

23/10/2019

Introduction:

Ce travail a été réalisé en binôme dans le cadre de l'UE *Machine Learning* au cours de la deuxième année de Master *Data Science*.

Le but de ce projet est d'étudier la capacité de plusieurs machines d'apprentissage à construire un classificateur de chiffres manuscrit.

Le jeu de données contient des chiffres manuscrits écrits sur des enveloppes pour spécifier le code postal (dans ce TP, on traite uniquement le cas des chiffres 3 et 4), Chaque digit est décrit par une image de $16 * 16$ pixels en niveaux de gris normalisés. La valeur pour chaque pixel est une valeur flottante dans $[-1, 1]$.

Plan:

Le plan de notre travail se décompose de la manière suivante:

- Traitement des données
- Analyse exploratoire des données
- Entrainement de diffèrents algorithmes de machine learning
- Comparaison des algorithmes

Traitement des données:

- On charge les données zip.train et zip.test qui viennent du package ElemStatLearn :

Les données sont dans deux fichiers zippés et chaque ligne est composée de l'identité des chiffres (0-9) suivi des 256 valeurs de niveaux de gris.

```
library(ElemStatLearn)
```

```
data(zip.train)
data(zip.test)
```

- On a deux data frames : **train** (données d'entrainement) et **test** (données de test).
  - La colone *X1* correspond au *label* compris entre 0 et 9 notée par la suite Y.
  - Les colonnes X_2 à X_{257} correspondent au *256 valeurs* associées au label de la colonne X1 qui contiennent une valeur dans l'intervalle $[-1, 1]$ indiquant la luminosité ou l'obscurité de chaque cellule.

Remarque: Ces valeurs sont déjà normalisées, par conséquent, on peut mieux comparer les variables entres elles.

```
df_train = data.frame(zip.train)
```

```
df_test = data.frame(zip.test)
```

- Quelle est la taille des deux tableaux de données?

```
dim(df_train)
```

```
## [1] 7291  257
```

```
dim(df_test)
```

```
## [1] 2007  257
```

On vérifie bien que le data set **train** a pour dimension **7291 lignes** et **257 colonnes**, et le data set **test** a pour dimension **2007 lignes** et **257 colonnes**.

- On considère à présent un nouveau jeu de données qui contient que les *labels* 3 et 4:

```
data_train<-df_train[df_train$X1==c("3","4"),]
```

```
data_test<-df_test[df_test$X1==c("3","4"),]
```

- On supprime les deux dataframes entiers de l'environnement

```
rm(df_train)
rm(df_test)
```

- Est ce qu'il y a des valeurs manquantes dans le **train** et le **test**?

```
sum(is.na(data_train))
```

```
## [1] 0
```

```
sum(is.na(data_test))
```

```
## [1] 0
```

Remarques: il n'y a aucune valeur manquante dans nos jeux de données

- Conversion des "étiquettes" *X1* en facteur:

```
data_train[, 1] <- as.factor(data_train[, 1])
data_test[, 1] <- as.factor(data_test[, 1])
head(sapply(data_train[1,], class))
```

```
##        X1        X2        X3        X4        X5        X6
##  "factor" "numeric" "numeric" "numeric" "numeric" "numeric"
```

Remarque: Toutes les autres colonnes contiennent des *numériques*

- Modification des noms des colonnes:

```
colnames(data_train) <- c("Y", paste("X.", 1:256, sep = ""))
```

```
colnames(data_test) <- c("Y", paste("X.", 1:256, sep = ""))
```

Analyse exploratoire des données:

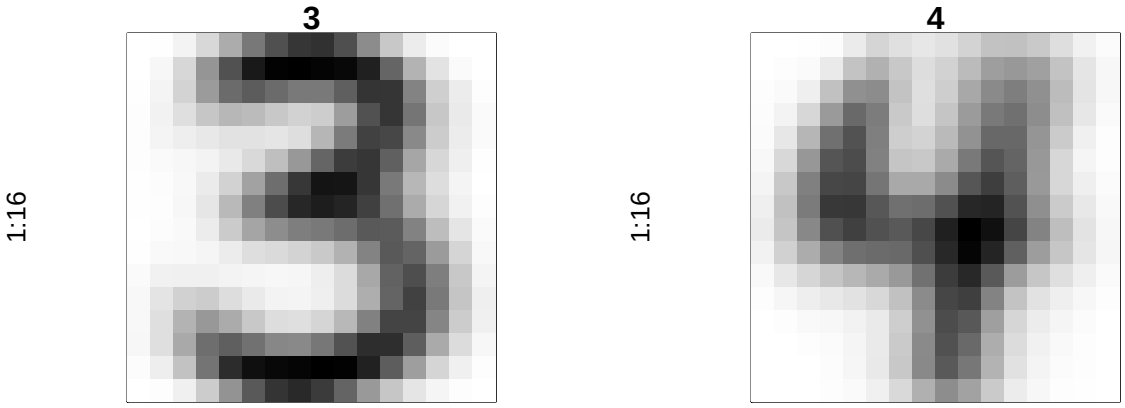
- Affichage des chiffres 3 et 4:

```
library(RColorBrewer)
COLORS <- c("white", "black")
CUSTOM_COLORS <- colorRampPalette(colors = COLORS)
CUSTOM_COLORS_PLOT <- colorRampPalette(brewer.pal(10, "Set3"))
par(mfrow = c(2,2), pty = "s", mar = c(1, 1, 1, 1), xaxt = "n", yaxt = "n")
images_digits <- array(dim = c(10, 16 * 16))
for (digit in 3:4) {
    print(digit)
    images_digits[digit + 1, ] <- apply(data_train[data_train[, 1] ==
        digit, -1], 2, sum)
    images_digits[digit + 1, ] <- images_digits[digit + 1, ]/max(images_digits[digit +
```

```
        1, ]) * 255
    z <- array(images_digits[digit + 1, ], dim = c(16, 16))
    z <- z[, 16:1]  ##right side up
    image(1:16, 1:16, z, main = digit, col = CUSTOM_COLORS(256))
}
```

```
## [1] 3
```

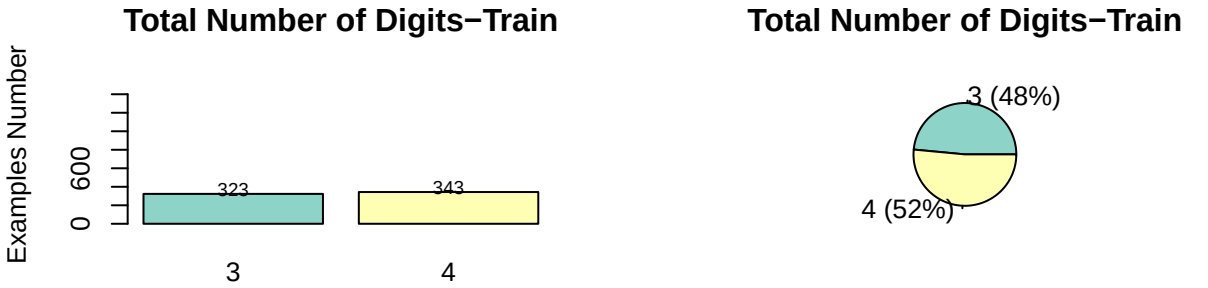
```
## [1] 4
```



- Quelle est la proportion de chaque chiffre dans le **Train** ?

```
resTable <- table(data_train$Y)
par(mfrow = c(2, 2))
par(mar = c(5, 4, 4, 2) + 0.1)
plot <- plot(data_train$Y, col = CUSTOM_COLORS_PLOT(10), main = "Total Number of Digits-Train",
    ylim = c(0, 1500), ylab = "Examples Number")
text(x = plot, y = resTable + 50, labels = resTable, cex = 0.75)


percentage <- round(resTable/sum(resTable) * 100)
labels <- paste0(row.names(resTable), " (", percentage, "%) ")  # add percents to labels
pie(resTable, labels = labels, col = CUSTOM_COLORS_PLOT(10), main = "Total Number of Digits-Train")
```



On remarque que les proportions des chiffres 3 et 4 dans le **train** sont très proches. Les algorithmes seront alors entrainés sur un échantillon d'entrainement bien équilibré.

Entrainement de diffèrents algorithmes de machine learning:

K-Nearest Neighbors (KNN):

Pour prédire une nouvelle instance, KNN calcule la distance euclidienne entre la nouvelle instance et toutes les instances de l'ensemble de la formation. Ensuite, l'algorithme recherche les K premières instances les plus proches (les plus similaires) et génère la classe avec la fréquence la plus élevée comme prédiction.

- La question est de savoir comment choisir K? La validation croisée peut être utilisée pour choisir la meilleure valeur pour K qui donne la plus grande précision. On utilise le package **CARET** pour utliliser le *KNN* et choisir *K*.

```
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
library(lattice)
library(ggplot2)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

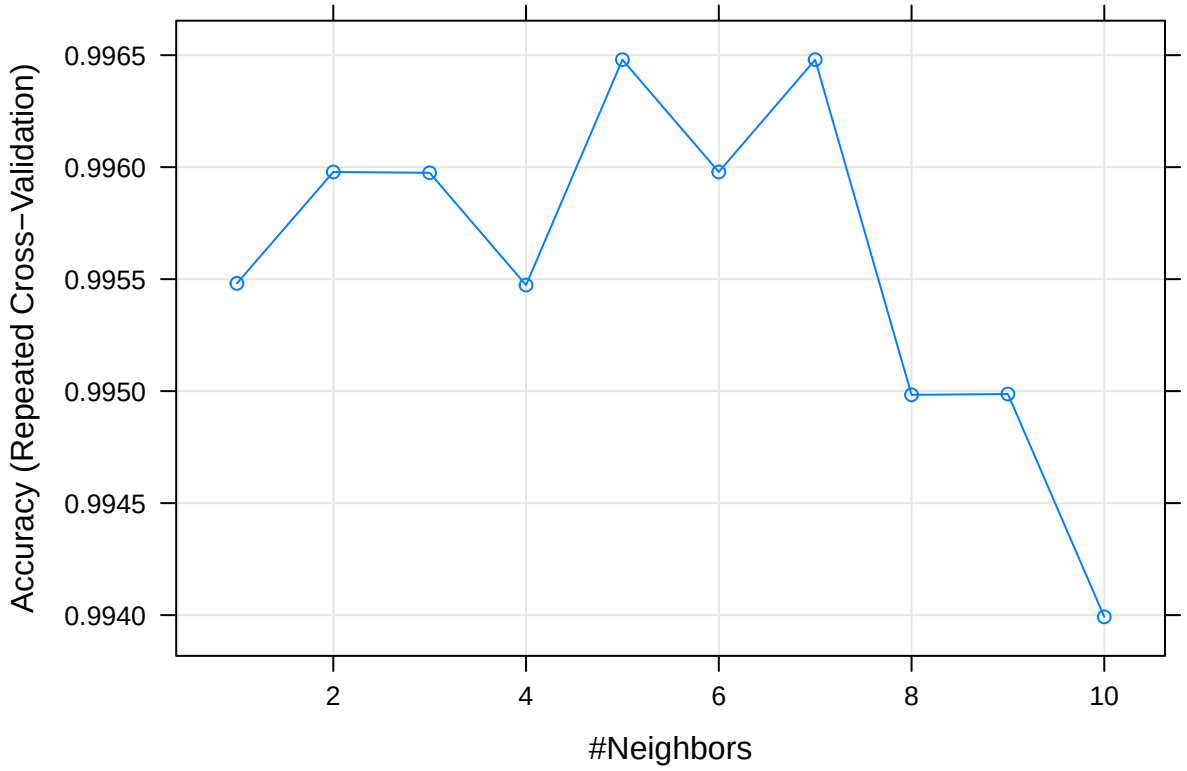
```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
set.seed(15)
#on répète 3 fois une Cross-Validation en 5-folds
control <- trainControl(method="repeatedcv", number=5, repeats=3)
fit.knn <- train(Y ~ .,
             method     = "knn",
             tuneGrid   = expand.grid(k = 1:10),
             trControl  = control,
             metric     = "Accuracy",
             data       = data_train)
```

- Affichage de la courbe de l'accuracy en fonction des k:

```
par(mfrow=c(2,2))
plot(fit.knn)
```



La valeur de k qui maximise cette courbe est $k = 7$

- Matrice de confusion **KNN**:

```
library(class)
set.seed(15)
pred_knn = knn(data_train,data_test,cl=data_train$Y, k =7)
MC_knn <- table(`Predicted Class`=pred_knn,`Actual Class` =data_test$Y)  #matrice de confusion
print(MC_knn)
```

```
##                Actual Class
## Predicted Class   3   4
##               3  87   0
##               4   2 108
```

On remarque que y a deux chiffres du **test** qui ont été mal prédit, en effet, le chiffre 3 a été deux fois prédit 4.

- Fonction **Accuracy**:

```
 accuracy <- function(x){sum(diag(x)/(sum(rowSums(x)))) * 100}
```

```
accuracies <- rep(0,8)  #liste qui contient les accuracy des algorithmes
algorithmes <- rep('algo',8)  #liste qui contient les noms des algorithmes
```

- Accuracy **KNN**:

```
acc_knn<-accuracy(MC_knn)
print(acc_knn)
```

```
## [1] 98.98477
```

```
accuracies[1]<-acc_knn
algorithmes[1]<-'KNN'
```

On remarque que l'accuracy sur le **test** est d'environ 99%

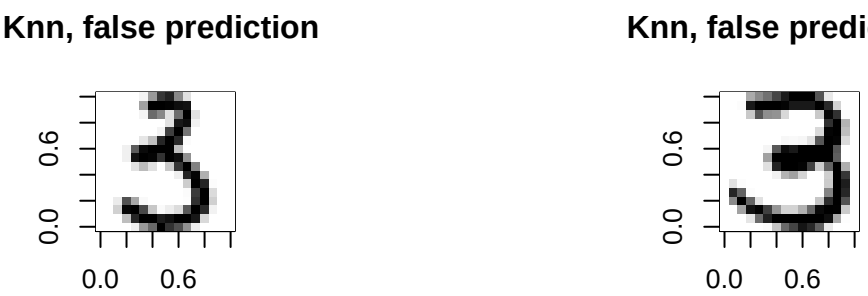
- Affichage des chiffres mal prédit par le modèle:

```
knn_false <- which(pred_knn != data_test$Y & data_test$Y == 3)
head(knn_false)
```

```
## [1]  31 107
```

Les deux chiffres mal prédits se trouvent aux lignes 31 et 107.

```
par(mfrow = c(2, 2), pty ='s')
BNW <- c("white", "black")
CUSTOM_BNW <- colorRampPalette(colors = BNW)
rotate <- function(x) t(apply(x, 2, rev))
for (i in 1:2){
  m = rotate(matrix(unlist(data_train[knn_false[i],-1]),ncol = 16,byrow = T))
  image(m,col=CUSTOM_BNW(255), main = "Knn, false prediction ")
}
```



Ces deux chiffres ont été prédit 4 alors qu'ils ont pour valeur réelle 3.

Linear Discriminant Analysis (LDA):

Cette méthode se base sur deux hypothèses, la première la normalité de chaque classe et la deuxième l'homoscédacité, c'est-à-dire toutes les classes ont la même matrice de variance covariance.

- Entrainement sur le *train*:

```
set.seed(15)
fit.lda <- train(Y ~ ., method = "lda", data = data_train)
```

- Prédiction sur le **test**:

```
pred_lda = predict(fit.lda, newdata = data_test)
```

- Accuracy **LDA**:

```
MC_lda<- table(`Predicted Class`=pred_lda,`Actual Class`=data_test$Y)
acc_lda <-accuracy(MC_lda)
print(acc_lda)
```

```
## [1] 97.96954
```

```
accuracies[2]<-acc_lda
algorithmes[2]<-'LDA'
```

On a une accuracy sur le **test** d'environ 98%

Regression Logistique:

- Entrainement sur le *train*:

```
set.seed(15)
fit.glm <- train(Y~., data=data_train, method="glm", trControl=control, metric="accuracy")
```

On a une accuracy sur le **train** d'environ 96%

- Prédiction sur le **test**:

```
pred_glm = predict(fit.glm, newdata = data_test)
```

- Accuracy **Regression logistique**:

```
MC_glm <- table(`Predicted Class`=pred_glm,`Actual Class`=data_test$Y)
acc_glm <- accuracy(MC_glm)
print(acc_glm)
```

```
## [1] 94.92386
```

```
accuracies[3]<-acc_glm
algorithmes[3]<-'reg log'
```

On a une accuracy sur le **test** d'environ 95%

Naîve Bayes:

La classification naïve bayésienne est un type de classification bayésienne probabiliste simple basée sur le *théorème de Bayes* avec une forte indépendance (dite naïve) des hypothèses.

- Entrainement sur le *train*:

```
library(e1071)
set.seed(15)
fit.NB <- naiveBayes(Y ~ ., data = data_train,metric="accuracy")
```

- Prédiction sur le **test**:

```
pred_NB = predict(fit.NB, newdata = data_test)
```

- Accuracy **Naïve Bayes** :

```
MC_NB <- table(`Predicted Class`=pred_NB,`Actual Class`=data_test$Y)
acc_BN <-accuracy(MC_NB)
print(acc_BN)
```

```
## [1] 96.4467
```

```
accuracies[4]<-acc_BN
algorithmes[4]<-'Bayes'
```

On a une accuracy sur le **test** d'environ 96%

Arbre de décision:

Cette méthode se base sur la représentation des choix sous la forme graphique d'un arbre avec les différentes décisions de classification placées dans les feuilles.

- Entrainement sur le **train**:

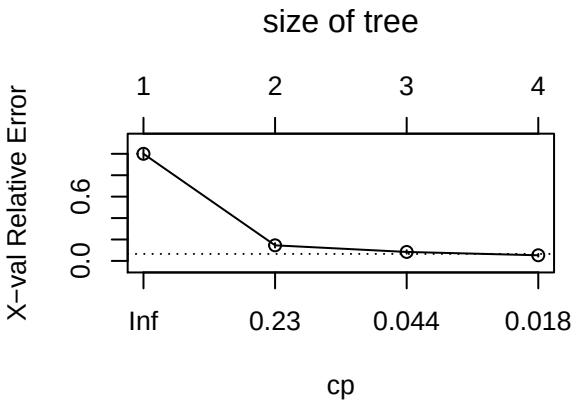
```
library(rpart)
set.seed(15)
model.rpart <- rpart(Y ~ ., method = "class", data = data_train)
```

-Estimation de l'arbre:

Notons par $|T|$ la taille de l'arbre (le nombre de noeuds terminaux). On pose $R_\alpha(t) = R(T) + \alpha|T|$

- **CP:** représente les valeurs du paramètre de complexité α
- **x-val relative error:** représente l'erreur relative $R(T)$
- On trace le paramètre de complexité α en fonction de l'erreur relative $R(T)$:

```
par(mfrow=c(2,2))
plotcp(model.rpart)
```



- On cherche la valeur du paramètre de complexité α qui minimise la courbe ci-dessus:

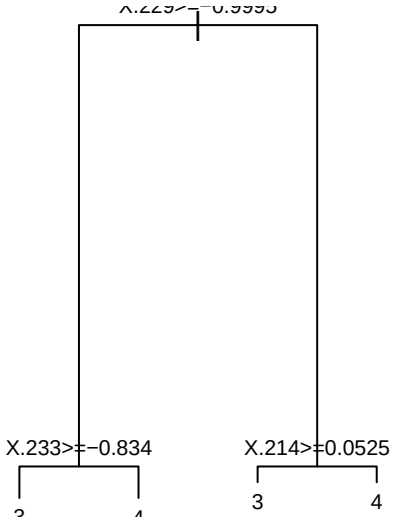
```
cpmin<-model.rpart$cptable[which.min(model.rpart$cptable[,4]),1]
print(cpmin)
```

```
## [1] 0.01
```

On trouve $\alpha = 0.01$

- Affichage de l'arbre de décison:

```
set.seed(15)
model.tree.opt <- prune(model.rpart, cp=cpmin)
par(mfrow=c(1,2))
plot(model.tree.opt)
text(model.tree.opt, cex = 0.7)
```



- Prédiction sur le **test**:

```
pred.rpart <- predict(model.tree.opt, newdata = data_test, type = "class")
```

- Accuracy **Arbre de décision** :

```
MC_rpart <-table(`Actual Class` = data_test$Y, `Predicted Class` = pred.rpart)
acc_tree <-accuracy(MC_rpart)
print(acc_tree)
```

```
## [1] 93.90863
```

```
accuracies[5]<-acc_tree
algorithmes[5]<-'tree'
```

On a une accuracy sur le **test** d'environ 94%

Bagging:

Le bagging est une technique utilisée pour améliorer la classification notamment celle des arbres de décision.il a pour but de réduire la variance de l'estimateur, en d'autres termes de corriger l'instabilité des arbres de décision.

- Entrainement sur les **train**:

```
library(ipred)
set.seed(15)
fit.bag <- bagging(Y~., data=data_train)
```

- Prédiction sur le **test**:

```
pred_bag = predict(fit.bag, newdata = data_test)
```

- Accuracy **Bagging**:

```
MC_bag <- table(`Predicted Class`=pred_NB,`Actual Class`=data_test$Y)
acc_bag <- accuracy(MC_bag)
print(acc_bag)
```

```
## [1] 96.4467
```

```
accuracies[6]<-acc_bag
algorithmes[6]<-'bagging'
```

On a une accuracy sur le **test** d'environ 96%

On remarque que l'accuracy du bagging est supérieure à celle des arbres de décision.

Forêt aléatoire (Random forest):

L'algorithme des forêts d'arbres décisionnels effectue un apprentissage sur de multiples arbres de décision entraînés sur des sous-ensembles de données légèrement différents.

- Entrainement sur les **train**:

```
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
set.seed(15)
fit.RF <- randomForest(Y~., data=data_train)
```

- Prédiction sur le **test**:

```
pred_RF = predict(fit.RF, newdata = data_test)
```

- Accuracy **Random forest**:

```
MC_RF <- table(`Predicted Class`=pred_RF,`Actual Class`=data_test$Y)
acc_RF <-accuracy(MC_RF)
print(acc_RF)
```

```
## [1] 98.98477
```

```
accuracies[7]<-acc_RF
algorithmes[7]<-'R.forest'
```

On a une accuracy sur le **test** d'environ 99%

On remarque que l'accuracy de Random forest s'est nettement améliorée par rapport à celle des arbres de décision

Boosting

Boosting regroupe de nombreux algorithmes qui s'appuient sur des ensembles de classifieurs binaires : dans notre cas il optimise les performances des arbres de décision.

- Entrainement sur le **train**:

```
library(adabag)
```

```
## Loading required package: foreach
```

```
## Loading required package: doParallel
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
##
## Attaching package: 'adabag'
```

```
## The following object is masked from 'package:ipred':
##
##     bagging
```

```
fit_boost <-boosting(Y ~ ., data = data_train, boos = FALSE, mfinal = 100)
```

- Prediction sur le **test**:

```
pred_boost <-predict(fit_boost, newdata = data_test)
```

- Accuracy **Boosting**:

```r
acc_boost<-1 - pred_boost$error
acc_boost*100
```

```
## [1] 98.47716
```

```r
accuracies[8]<-acc_boost*100
algorithmes[8]<-'boosting'
```

On remarque que l'accuracy sur le **test** est d'environ 98%

Nous regroupons les prédictions des algorithmes précédents dans une matrice, nous codons 1 les prédictions 3 , 0 les 4.

```r
pred.all<-data.frame(KNN=as.numeric(pred_knn==3),lda=as.numeric(pred_lda==3),
                     Bayes=as.numeric(pred_NB==3),bagging=as.numeric(pred_bag==3),
                     tree=as.numeric(pred.rpart==3),randomforest=as.numeric(pred_RF==3),
                     reglog=as.numeric(pred_glm==3))
print(head(pred.all))
```

```
##   KNN lda Bayes bagging tree randomforest reglog
## 1   1   1     1       1    1            1      1
## 2   0   0     0       0    0            0      0
## 3   0   0     0       0    0            0      0
## 4   0   0     0       0    0            0      0
## 5   1   1     1       1    1            1      1
## 6   1   0     1       1    1            0      0
```

Pour le premier, il y a unanimité des décisions pour 3, par contre pour le 6eme individu, KNN, Bayes, Bagging et tree sont d'accord et prédisent 3, contrairement à LDA, Random Forest et Regression logistique. Etc.

```r
#corrélation entre les prédictions
cor(pred.all)
```

```
##                     KNN       lda     Bayes   bagging      tree
## KNN          1.0000000 0.9382445 0.9486490 0.9486490 0.8778170
## lda          0.9382445 1.0000000 0.8869644 0.9486490 0.8778170
## Bayes        0.9486490 0.8869644 1.0000000 0.8973103 0.8263885
## bagging      0.9486490 0.9486490 0.8973103 1.0000000 0.9287904
## tree         0.8778170 0.8778170 0.8263885 0.9287904 1.0000000
## randomforest 0.9794148 0.9588297 0.9280875 0.9486490 0.8778170
## reglog       0.8769701 0.9385905 0.8461783 0.9077280 0.8365368
##              randomforest    reglog
## KNN             0.9794148 0.8769701
## lda             0.9588297 0.9385905
## Bayes           0.9280875 0.8461783
## bagging         0.9486490 0.9077280
## tree            0.8778170 0.8365368
## randomforest    1.0000000 0.8975102
## reglog          0.8975102 1.0000000
```

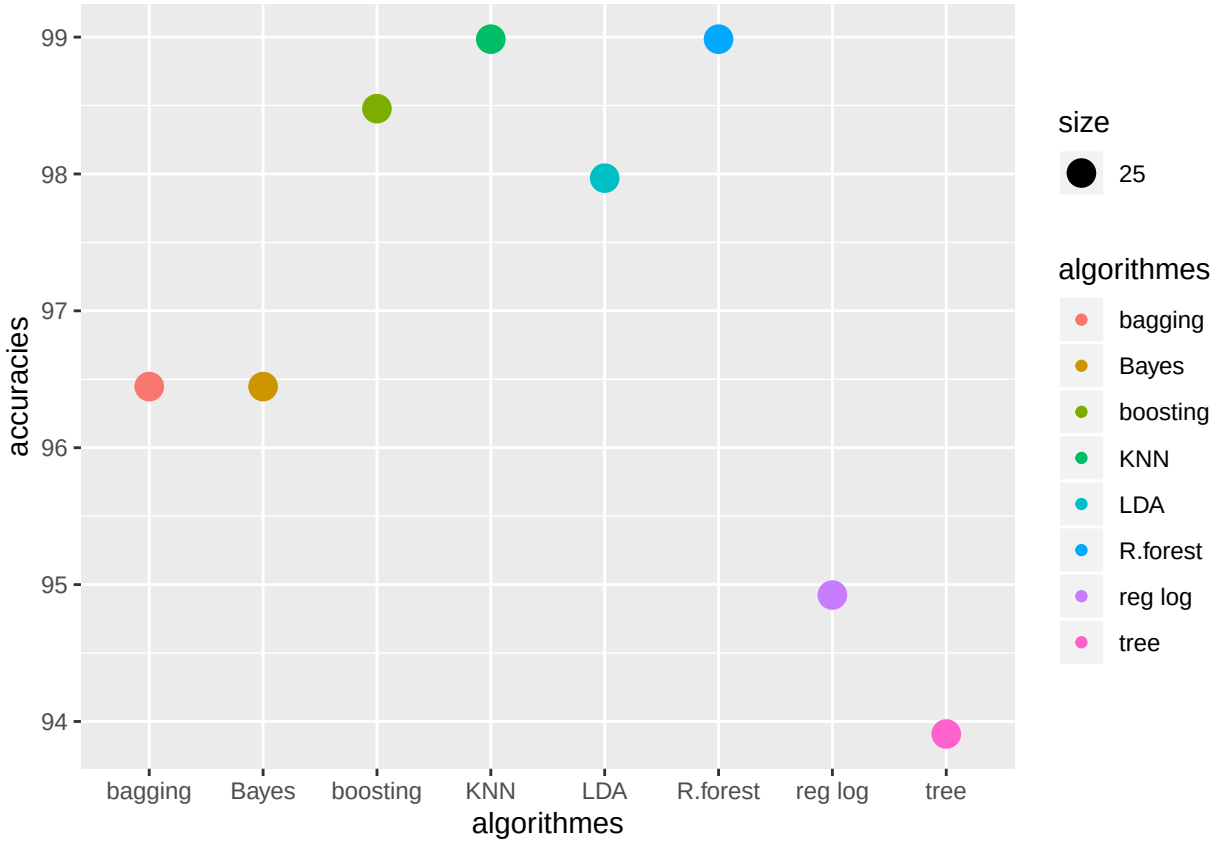
On remarque qu'il y a de fortes corrélations entre les prédictions pour les algorithmes précédent (les corrélations sont toutes > 0.80). On peut alors se dire qu'il n'y aura pas d'amélioration significative si on combine ces algorithmes en faisant du stacking.

Comparaison des algorithmes:

- Graphe de toutes les accuracy:

On représente les accuracy de chaque algorithme en pourcentage.

```
df_algo <- data.frame(accuracies,algorithmes)
ggplot(df_algo, aes(x=algorithmes, y=accuracies, group=algorithmes)) +
  geom_point(aes( color=algorithmes,size=25))
```



Conclusion:

D'après le graphe ci-dessous on conclut que les meilleurs algorithmes sont: le KNN, Random forest car ils ont les meilleurs accuracy sur le test.