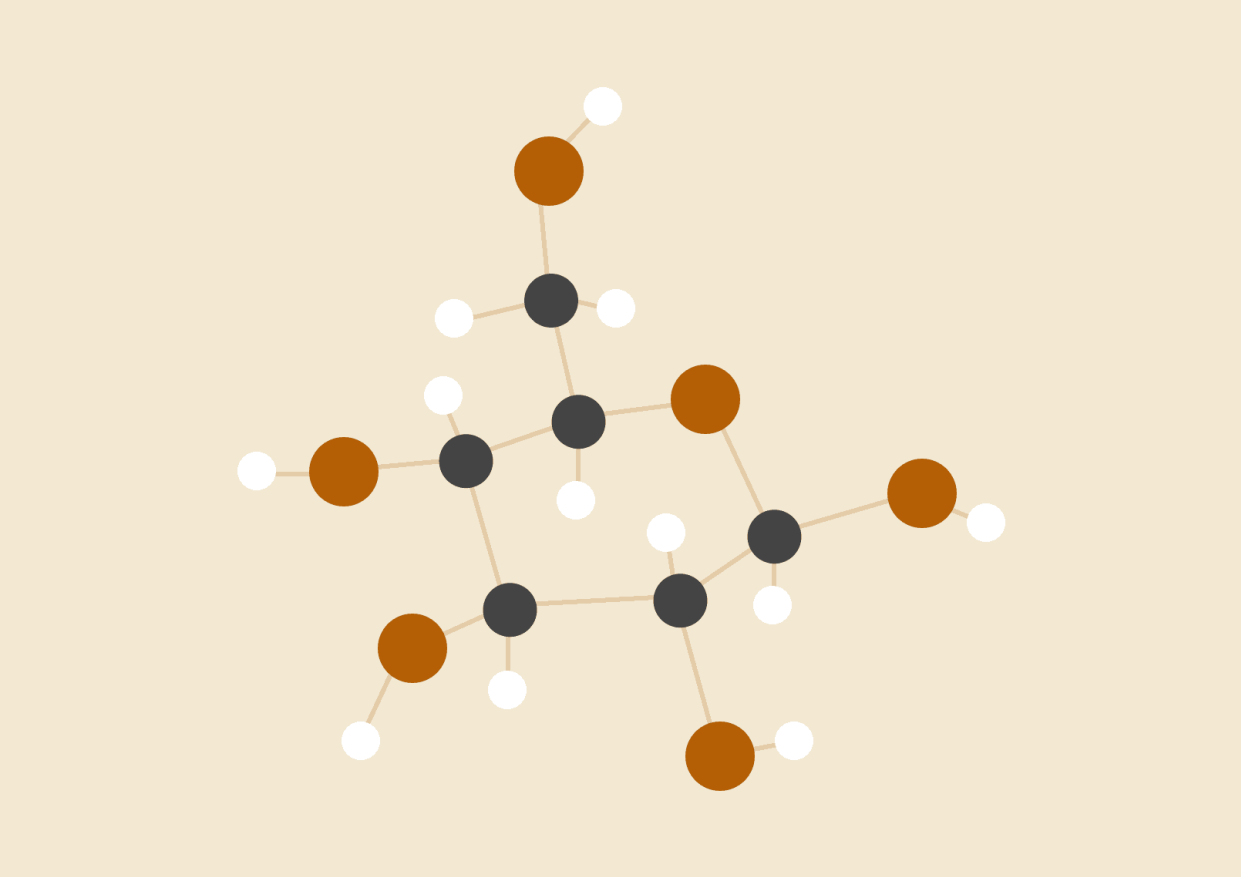
RAPPORT TP2

Cours d'apprentissage statistique de M. Dalalyan



Cordoba Muñoz Santiago
Ureña Stéphane
Watteau Rodolphe

13/11/2018
MS–DS

INTRODUCTION :

Ce document a vocation à répondre aux différentes questions relatives au TP2.

Ce TP s'intéresse aux techniques de datamining à des fins de détections des fraudes. Il s'agira dans ce document de faire une analyse de données afin de repérer les outliers d'une série de données et de faire des prédictions sur le caractère frauduleux ou non de données.

The Available Data :

Nous disposons d'une table de 401 146 lignes (chaque ligne correspond à une transaction) composées des informations suivantes : l'ID du vendeur, le numéro du produit vendu, la quantité d'unités vendues, la somme d'argent vendu, l'état de contrôle de la vente ('ok' , 'fraud', 'unkn').

Question 1

Les commandes 'table' et 'barplot' servent respectivement à construire un tableau de contingence entre les facteurs que l'on entre en input et construire un graphique en barre verticale ou horizontal selon l'option choisie.

```
# Loader le dataset
library(DMwR)
data(sales)

# Apercu des données
summary(sales)

#autre méthode d'observation
library(Hmisc)
describe(sales)

#nombre de products and salespeople
c(nlevels(sales$ID), nlevels(sales$Prod))

#check sur les non valeurs
length(which(is.na(sales$Quant)& is.na(sales$Val)))

#proportion de frauds
table(sales$Insp)/nrow(sales)*100

#(a) Affichage du nombre de rapports par vendeurs
données_graph1=table(sales$Insp,sales$ID)
barplot(données_graph1, width = 1, space = NULL)
# (b) Affichage du nombre de rapports par produits
données_graph2=table(sales$Insp,sales$Prod)
barplot(données_graph2, width = 1, space = NULL)
```

Illustration 1: Code qui load les données et construit les deux graphiques barplot ci-dessus

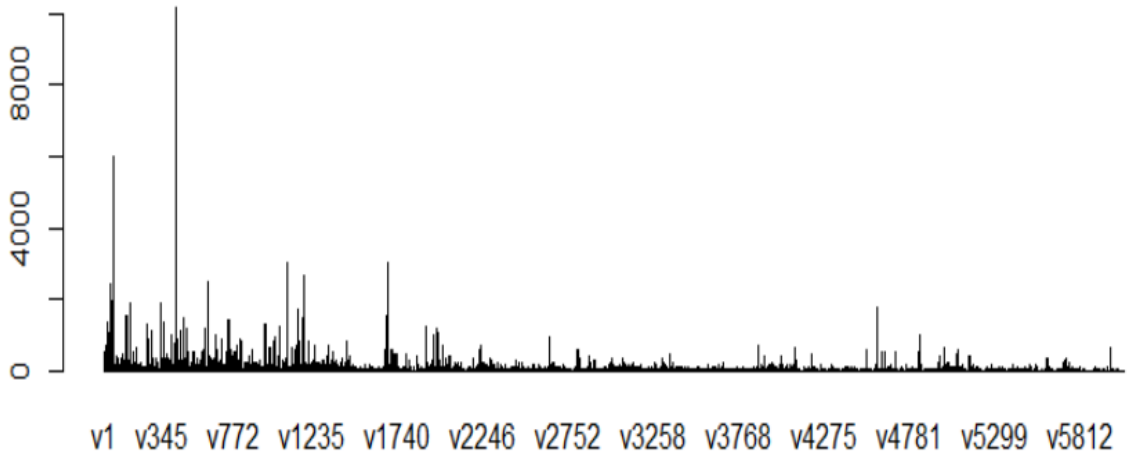


Illustration 2 : Diagrammes en bâtons représentant le nombre de rapports par vendeurs.

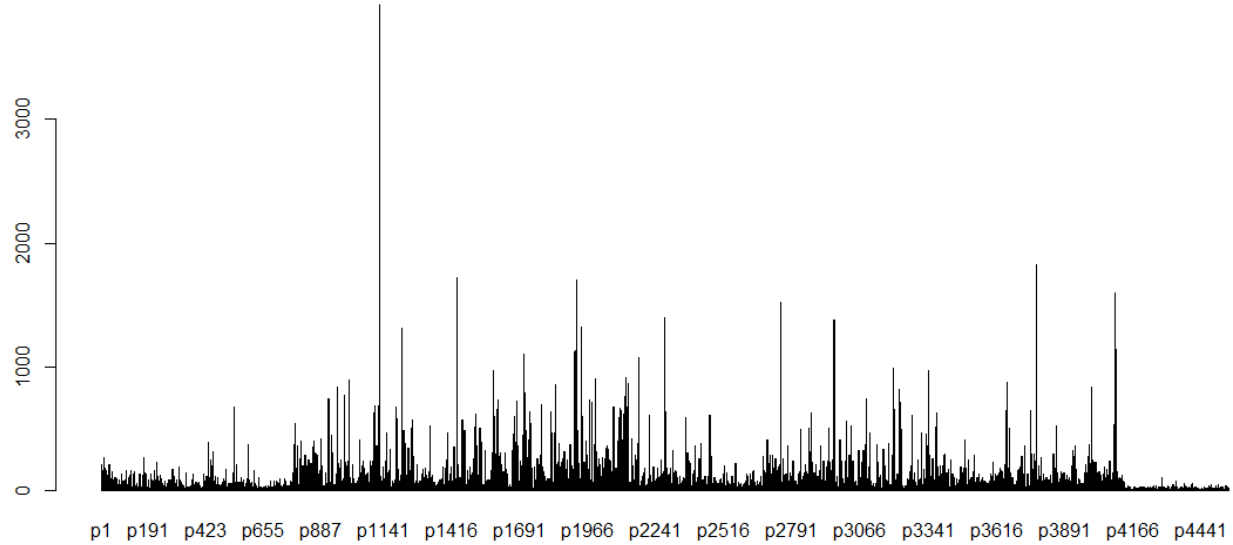


Illustration 3 : Diagrammes en bâtons représentant le nombre de rapports par produits.

En regardant ce graphique nous notons qu'il y a une grande variabilité dans le nombre de transactions d'un produit à l'autre ainsi que d'un vendeur à l'autre.

Source : https://www.rdocumentation.org/packages/base/versions/3.5.1/topics/table

https://www.rdocumentation.org/packages/graphics/versions/3.5.1/topics/barplot

Question 2 :

Maintenant nous allons raisonner en termes de prix unitaires sur chaque transaction. Afin de visualiser les outliers de notre série, nous faisons un graphique en boîte grâce à la commande Boxplot de R qui permet de construire ce genre d'objet. Le résultat est le suivant :

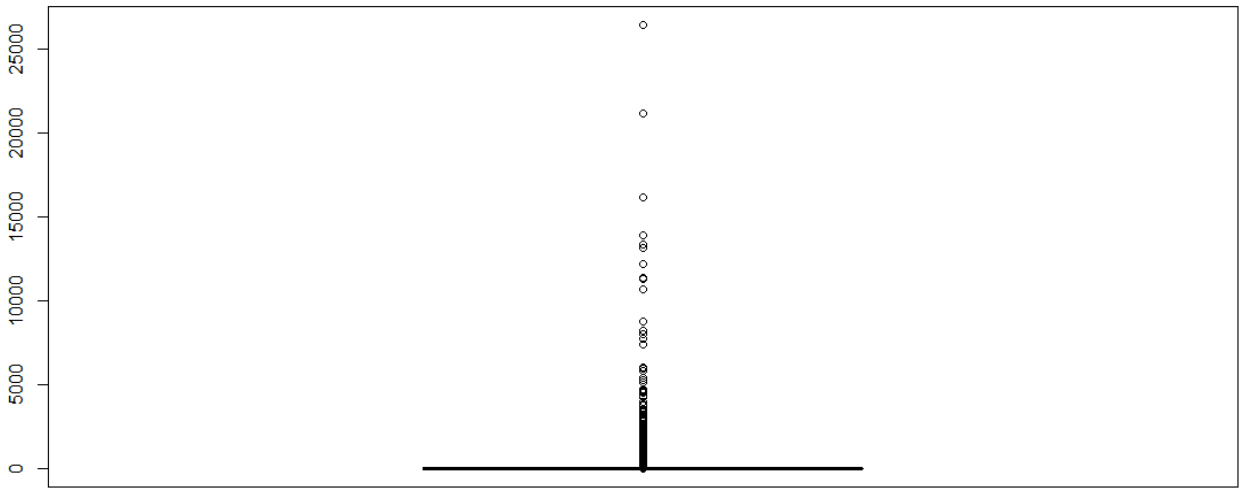


Illustration 4 : Boxplot qui permet de visualiser les variations des prix unitaires des produits sur l'ensemble des données disponibles.

Source : https://www.statmethods.net/graphs/boxplot.html

The boxplot rule for outlier detection :

Question 3 :

Maintenant nous allons nous intéresser de plus près aux transactions présentant des valeurs de prix unitaires anormales au regard des prix qu'ils affichent sur l'ensemble des transactions pour ce même produit.

Tout d'abord, avec la ligne de code suivante nous calculons le nombre de transaction (pour une série de produits numérotés dans le vecteur x) dont le prix unitaire se situe en dehors de l'intervalle issu de la formule suivante :

« Q3 + 1.5 x IQR (Q1 - 1.5 x IQR) »

where Q1 is the first quartile, Q3 the third quartile, and IQR = (Q3 - Q1) the inter-quartile range.

Ce nombre donne une première information sur le nombre de transactions présentant des valeurs anormales.

```
> x = c( 5,6,87,7,89,5,14,521,20,46,89,7,89,4,54,5) # échantillons de produits
> n = sum(boxplot.stats(x)$out) # calcul du nombre d'outliers
> n # affichage
[1] 521
```

Pour ce vecteur x, on dénombre 521 outiliers.

Question 3.1

La commande '*tapply*' permet d'appliquer une fonction à un ensemble d'objets, ici en l'occurrence on calcule le nombre d'outliers par produits compilant la ligne de codes suivantes.

```
out.uprice = tapply(Uprice,Prod, function(x) sum(boxplot.stats(x)$out))
out.uprice
```

Source : https://www.rdocumentation.org/packages/base/versions/3.5.1/topics/tapply

Question 3.2 :

Il suffit de faire la somme sur le vecteur précédent composé du nombre d'outliers par produits pour avoir le nombre total d'outliers. On observe qu'ils sont au nombre de 29 446.

```
#2)nombre total d'outlier => 29446
sum(out.uprice)
```

Question 3.3 :

Pour obtenir le pourcentage de transactions signalées comme anormale, il suffit d'utiliser le résultat obtenu ci-dessus divisé par le nombre total de transactions.

```
percentage.Abnormal=tapply(Uprice, Prod, function(x) sum(boxplot.stats(x)$out!=0)/length(Uprice))
sum(percentage.Abnormal)*100
```

Nous obtenons 7,34047% de transactions signalées anormales.

Question 3.4 :

On s'intéresse maintenant aux 10 produits présentant le plus de transactions signalées frauduleuses :

```
#4) 10 produits ayant le plus de transactions douteuses
table.total=table(Prod,sales$Insp)
table.fraud=table.total[,3]
table.fraud.tri=sort(table.fraud, decreasing = T)
dix_produits_fraud=table.fraud.tri[1:10]
dix_produits_fraud
```

Le résultat obtenu est le suivant avec en premier ligne le numéro du produit et en seconde ligne le nombre de transactions signalées frauduleuses :

```
p2456 p1000  p226  p314  p545 p2459 p3774 p1125 p2516  p318
   29    12     9     9     9     9     9     8     8     7
```

Supervised Techniques :

Question 4.1 :

La méthode ‘XGBoost’ permet de procéder à une classification en utilisant une méthode de Boosting avec l’algorithme XGBoost (eXtreme Gradient Boosting).

Ci- dessous le programme permettant d’effectuer l’algorithme xgBoost :

```r
#XGBoost, Logit and decison trees
# Programme XGBoost qui pour un nbr de tour allant de 1 à 10 :
# effectue des prédictions sur la nature de la transaction (ok ou fraud) par XGBoost
# et affiche les taux d'erreurs sur test et train
install.packages("drat", repos="https://cran.rstudio.com")
drat:::addRepo("dmlc")
install.packages("xgboost", repos="http://dmlc.ml/drat/", type = "source")
install.packages("xgboost")

require(xgboost)

#Patie TRAIN
sparse_matrix_train <- data.matrix(sales.train)
data_train = sparse_matrix_train[,1:3]
label_train = sparse_matrix_train[,4]-1
#partie TEST
sparse_matrix_test <- data.matrix(sales.test)
data_test = sparse_matrix_test[,1:3]
label_test = sparse_matrix_test[,4]-1

#Algorithme XGBOOST
err<-list()
rg<-list()
for (i in 1:10) {
  #training:
  bst=xgboost(data = data_train,label = label_train ,
              max.depth = 2, eta = 1, nthread = 2,
              nrounds =i, objective = "binary:logistic")
  #Prédictions with test :
  pred_xgb <- predict(bst, data_test)

  #with regression (binary result)
  prediction_xgb <- as.numeric(pred_xgb > 0.5)

  #add to list every error recorded
  err[[i]] <- mean(as.numeric(pred_xgb > 0.5) != label_test)
  print(paste("test-error=", err[[i]]))
  rg[[i]]<-i
}
```

Nous obtenons les erreurs suivantes à chaque round allant de 1 à 10 :

```
[1]     train-error:0.168367
[1] "test-error= 0.1312178387650 09"
[1]     train-error:0.168367
[2]     train-error:0.175719
[1] "test-error= 0.0793310463121784"
[1]     train-error:0.168367
[2]     train-error:0.175719
[3]     train-error:0.147689
[1] "test-error= 0.0698970840480274"
[1]     train-error:0.168367
[2]     train-error:0.175719
[3]     train-error:0.147689
[4]     train-error:0.146126
[1] "test-error= 0.0885506003430532"
[1]     train-error:0.168367
[2]     train-error:0.175719
[3]     train-error:0.147689
[4]     train-error:0.146126
[5]     train-error:0.141715
[1] "test-error= 0.0960548885077187"
```

```
[1]     train-error:0.168367
[2]     train-error:0.175719
[3]     train-error:0.147689
[4]     train-error:0.146126
[5]     train-error:0.141715
[6]     train-error:0.133627
[1] "test-error= 0.0939108061749571"
[1]     train-error:0.168367
[2]     train-error:0.175719
[3]     train-error:0.147689
[4]     train-error:0.146126
[5]     train-error:0.141715
[6]     train-error:0.133627
[7]     train-error:0.132157
[1] "test-error= 0.0958404802744425"
[1]     train-error:0.168367
[2]     train-error:0.175719
[3]     train-error:0.147689
[4]     train-error:0.146126
[5]     train-error:0.141715
[6]     train-error:0.133627
[7]     train-error:0.132157
[8]     train-error:0.128297
[1] "test-error= 0.0889794168096055"
[1]     train-error:0.168367
[2]     train-error:0.175719
[3]     train-error:0.147689
[4]     train-error:0.146126
[5]     train-error:0.141715
[6]     train-error:0.133627
[7]     train-error:0.132157
[8]     train-error:0.128297
[9]     train-error:0.123978
[1] "test-error= 0.0838336192109777"
[1]     train-error:0.168367
[2]     train-error:0.175719
[3]     train-error:0.147689
[4]     train-error:0.146126
[5]     train-error:0.141715
[6]     train-error:0.133627
[7]     train-error:0.132157
[8]     train-error:0.128297
[9]     train-error:0.123978
[10]    train-error:0.115982
[1] "test-error= 0.0769725557461406"
```

On note que l'algorithme tourne bien sur chaque itération en incrémentant le nombre de train-error à chaque round.

Source :

https://xgboost.readthedocs.io/en/latest/R-package/xgboostPresentation.html

https://en.wikipedia.org/wiki/Xgboost

https://github.com/dmlc/xgboost

Question 4.2 :

Maintenant, nous allons déterminer le round optimal c'est-à-dire celui qui enregistre la plus faible erreur du l'échantillon testé.

Voici le code permettant de le déterminer :

```
> a<-0
> for (i in 1:10) {
+   if (err[[i]]==min(unlist(err))){
+     a<-i
+   }
+ }
> print("question 4.2 : erreur minimale pour le xgboost:")
[1] "question 4.2 : erreur minimale pour le xgboost:"
> print(min(unlist(err)))
[1] 0.05553173
> print("le round de l'erreur minimale du xgboost est : ")
[1] "le round de l'erreur minimale du xgboost est : "
> print(a)
[1] 10
```

On constate donc que le xgboost enregistre 5,5% d'erreurs de prédictions pour un round de 10.

Cette erreur est très faible ce qui prouve la performance prédictive de l'algorithme xgboost.

Questions 4.3 :

Nous procédons à des prédictions issues d'un modèle à arbres de décisions et calculons notre erreur de prédiction afin de comparer avec le xgBoost . Ci-dessous le code que nous exécutons pour obtenir nos résultats sur ces mêmes données :

```
#binary tree
library(rpart)
tree <- rpart(Insp ~ ID + Prod + Uprice,method="class", data=sales.train)
t_pred = predict(tree,sales.test,type="class")
t = sales.test['Insp']
confMat <- table(sales.test$Insp,t_pred)
accuracy <- sum(diag(confMat))/sum(confMat)
print(accuracy)
error=1-accuracy
print(error)
```

Ci-dessous les résultats obtenus :

```
print(accuracy)
] 0.9341767
error=1-accuracy
print(error)
] 0.06582333
```

Nous obtenons une précison de 93,4% et ainsi une erreur de prédiction de 6,58% sur le modèle d'arbres de décisions. Nous concluons que le xgboost à un pouvoir prédictif plus puissant que les arbres de décisions.

Enfin nous allons procéder au même exercice en passant via un modèle logistique 'logit '.

Le code ci-dessous permet d'exécuter l'algorithme du modèle logistique :

```
#logit algorithm
model <- glm(Insp ~.,family=binomial(link='logit'),data=sales.train)
summary(model)
#no response
```

Malheureusement la puissance de calcul de nos machines ne parvient pas à aboutir à un résultat avec ce modèle logistique. Nous concluons que compte tenu de l'important volume de données de nos échantillons le modèle n'arrive pas à aboutir en un temps raisonnable, ainsi en comparaison de ces 2 modèles le modèle xgboost est le plus performant et sera retenu.

Source :

https://www.rdocumentation.org/packages/DMwR/versions/0.4.1/topics/kNN

https://www.rdocumentation.org/packages/rpart/versions/4.1-13/topics/rpart

http://perso.ens-lyon.fr/lise.vaudor/realiser-une-regression-logistique-avec-r/

The Class Imbalance Problem :

Question 5 :

Dans cette question nous allons procéder au même exercice de comparaison des modèles en modifiant le dataset d'entrainement initial grâce à la commande 'SMOTE'.

Nous appliquons pour cela la ligne de code suivante afin de modifier notre échantillon :

```
sales.train.smote <- SMOTE(Insp ~ ., sales.train, perc.under = 500)
```

Ensuite nous procédons à l'exécution successive des 3 modèles présentés dans la question 4.3 et comparons les résultats obtenus.

- Pour le Xgboost :

```
[1]	train-error:0.155038
[1] "test-error= 0.101200686106346"
[1]	train-error:0.155038
[2]	train-error:0.144403
[1] "test-error= 0.135077186963979"
[1]	train-error:0.155038
[2]	train-error:0.144403
[3]	train-error:0.134592
[1] "test-error= 0.107847341337907"
[1]	train-error:0.155038
[2]	train-error:0.144403
[3]	train-error:0.134592
[4]	train-error:0.121207
[1] "test-error= 0.0887650085763293"
[1]	train-error:0.155038
[2]	train-error:0.144403
[3]	train-error:0.134592
[4]	train-error:0.121207
[5]	train-error:0.118089
[1] "test-error= 0.0917667238421955"
[1]	train-error:0.155038
[2]	train-error:0.144403
[3]	train-error:0.134592
[4]	train-error:0.121207
[5]	train-error:0.118089
[6]	train-error:0.119006
[1] "test-error= 0.0836192109777015"
[1]	train-error:0.155038
[2]	train-error:0.144403
[3]	train-error:0.134592
[4]	train-error:0.121207
[5]	train-error:0.118089
[6]	train-error:0.119006
[7]	train-error:0.118823
[1] "test-error= 0.0834048027444254"
[1]	train-error:0.155038
[2]	train-error:0.144403
[3]	train-error:0.134592
[4]	train-error:0.121207
[5]	train-error:0.118089
[6]	train-error:0.119006
[7]	train-error:0.118823
[8]	train-error:0.115247
[1] "test-error= 0.092409948542024"
[1]	train-error:0.155038
[2]	train-error:0.144403
[3]	train-error:0.134592
[4]	train-error:0.121207
[5]	train-error:0.118089
[6]	train-error:0.119006
```

```
[7]	train-error:0.118823
[8]	train-error:0.115247
[9]	train-error:0.109563
[1] "test-error= 0.0849056603773585"
[1]	train-error:0.155038
[2]	train-error:0.144403
[3]	train-error:0.134592
[4]	train-error:0.121207
[5]	train-error:0.118089
[6]	train-error:0.119006
[7]	train-error:0.118823
[8]	train-error:0.115247
[9]	train-error:0.109563
[10]	train-error:0.111946
[1] "test-error= 0.0906946826758148"
```

Nous constatons que les erreurs sont nettement plus importantes dans ce cadre d'étude.

```
> print("question 4: erreur minimale pour le xgboost:")
[1] "question 4: erreur minimale pour le xgboost:"
> print(min(unlist(err_smote)))
[1] 0.0834048
> print("le range de l'erreur minimale du xgboost est : ")
[1] "le range de l'erreur minimale du xgboost est : "
> print(b)
[1] 7
```

Nous constatons que l'erreur la plus faible de xgboost est 8,3% pour 7 round, cette valeur est donc supérieur lorsque l'on modifie l'échantillon d'apprentissage avec la commande SMOTE.

- Intéressons au modèle d'arbre de décision :

```
> library(rpart)
> tree_smote <- rpart(Insp ~ ID + Prod + Uprice,method="class", data=sales.train.smote)
> t_pred_smote = predict(tree_smote,sales.test,type="class")
> t_smote = sales.test['Insp']
> confMat_smote <- table(sales.test$Insp,t_pred_smote)
> accuracy_smote <- sum(diag(confMat_smote))/sum(confMat_smote)
> print(accuracy_smote)
[1] 0.8745712
> error_smote=1-accuracy_smote
> print(error_smote)
[1] 0.1254288
```

Nous obtenons une précision de 87,4% et ainsi une erreur de prédiction de 12,54% sur le modèle d'arbres de décisions. La précision de ce modèle a elle aussi été impactée par la modification liminaire. Nous concluons que le xgBoost à un pouvoir prédictif toujours plus puissant que les arbres de décisions.

- Modèle logistique : logit

```
#logit algorithm

model_smote <- glm(Insp ~.,family=binomial(link='logit'),data=sales.train.smote)
summary(model)
```

Nous nous confrontons au même problème que précécemment avec un algorithme qui s'éxécute trop lentement. Nous ne parvenons pas à sortir des résultats avec ce modèle logistique.

Nous pouvons conclure qu'en modifiant l'échantillon d'apprentissage avec le commande 'SMOTE', nous dégradons la précision de nos modèles, néanmoins nous conservons la conclusion sur la domination de l'algorithme xgBoost pour les prédictions.

Source : https://www.rdocumentation.org/packages/DMwR/versions/0.4.1/topics/SMOTE

RÉFÉRENCES :

Question 1 :

https://www.rdocumentation.org/packages/base/versions/3.5.1/topics/table

https://www.rdocumentation.org/packages/graphics/versions/3.5.1/topics/barplot

Question 2 :

https://www.statmethods.net/graphs/boxplot.html

Question 3.1

https://www.rdocumentation.org/packages/base/versions/3.5.1/topics/tapply

Question 4.1 :

https://xgboost.readthedocs.io/en/latest/R-package/xgboostPresentation.html

https://en.wikipedia.org/wiki/Xgboost

https://github.com/dmlc/xgboost

Question 4.3 :

https://www.rdocumentation.org/packages/DMwR/versions/0.4.1/topics/kNN

https://www.rdocumentation.org/packages/rpart/versions/4.1-13/topics/rpart

http://perso.ens-lyon.fr/lise.vaudor/realiser-une-regression-logistique-avec-r/