Scénario: discrimination de nuages gaussiens

Résumé

Les méthodes de discrimination sont comparées sur un jeu de données fictif obtenu par la simulation d'observations (x_i, y_i) séparées en 2 classes et réalisation de 4 gaussiennes bidimensionnelles. L'objectif est de mettre en évidence le rôle des paramètres de complexité de différentes méthodes (régression logistique, k-nn, réseaux de neurones, arbre de décision, bagging, svm) et de comparer les formes spécifiques des frontières estimées par chacune d'elle.

1 Introduction

Les données très rudimentaires ont été obtenues par la simulation de 4 gaussiennes bidimensionnelles ; 3 gaussiennes sont associées à une classe la 4ème à une autre classes. L'objectif est d"apprendre" ces données très particulières afin de discriminer les deux classes. Les données étant simplement dans $\mathbb{R}2$, il est facile de prévoir la classe de chaque point du plan et ainsi de visualiser la frontière entre les prévisions des deux classes. L'intérêt est de représenter ainsi facilement le rôle jouer par les paramètres de complexité de chaque méthode et de comparer les formes des frontières obtenues et donc la plus ou moins bonne adéquation d'une méthode à la spécificité de ces données simulées.

2 Les données

2.1 Données issues de Matlab

Les données servent d'exemple dans la *toolbox* Matlab de *classification*. Ce pourrait être l'occasion de tester le transfert de données entre Matlab et R à l'aide de la librairie `R.matlab`. Charger le fichier `clouds.mat` du répertoire `data` de Wikistat.

```
# Lire les données d'un format matlab
library(R.utils)
library(R.matlab)
nuage=readMat("clouds.mat")
# composants de l'archive matlab
attributes(nuage)
nuage$distribution.parameters
nuage1=nuage$targets
nuage2=nuage$patterns
# création du data frame R
cloud=data.frame("classe"=as.factor(t(nuage1)),
   "X"=t(nuage2)[,1],"Y"=t(nuage2)[,2])
summary(cloud)
# nuage de points de "clouds"
plot(cloud[,2:3],col=as.integer(cloud[,1]),
   pch=16,cex=.2)
```

2.2 Données en format texte

Sans passer par le format Matlab, charger directement les données au format texte du fichier `clouds.dat` avant de les lire.

```
# lire
cloud=read.table("clouds.dat")
cloud[,1]=as.factor(cloud[,1])
# vérifier
summary(cloud)
```

```
##  classe         X                  Y
##  0:2500   Min.   :-3.2919   Min.   :-3.6399
##  1:2500   1st Qu.:-0.2157   1st Qu.:-0.2177
##           Median : 0.1676   Median : 0.3539
##           Mean   : 0.4924   Mean   : 0.5224
##           3rd Qu.: 1.7283   3rd Qu.: 1.4423
##           Max.   : 3.4806   Max.   : 4.7569
```

```
# nuage de points de "clouds"
```

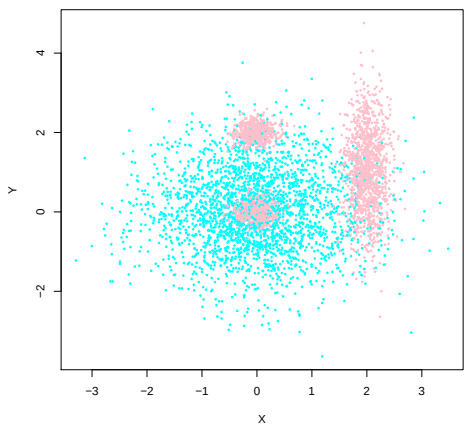

FIGURE 1 – Ensemble d'apprentissage composé du mélange de deux gaussiennes bidimensionnelles

```
plot(cloud[,2:3],col=c("pink", "cyan")
    [as.integer(cloud[,1])],pch=16,cex=.5)
```

Remarquer comment les gaussiennes se répartissent en 2 classes dont une, composée du mélange de 3 gaussiennes, produit une classe qui n'est pas connexe.

2.3 Points du plan

Construction des données de "test" ou plutôt de tous les points du plan (X, Y) et dont la prévision "dessinera" les frontières des classes.

```
testy=rep(seq(-5,5,length.out = 100),100)
testx=sort(rep(seq(-5,5,length.out = 100),100))
test=data.frame("X"=testx,"Y"=testy)
summary(test)
```

```
##        X                Y
##  Min.   :-5.0   Min.   :-5.0
##  1st Qu.:-2.5   1st Qu.:-2.5
##  Median : 0.0   Median : 0.0
##  Mean   : 0.0   Mean   : 0.0
##  3rd Qu.: 2.5   3rd Qu.: 2.5
##  Max.   : 5.0   Max.   : 5.0
```

3 Méthodes de classification

L'objectif est donc d'apprendre la discrimination entre les deux classes des nuages de points. L'utilisation de chaque méthode suit le même déroulement :

1. Estimation du modèle pour une complexité donnée,
2. prévision des points du plan,
3. représentation des classes prévues dans le plan
4. même chose pour différentes valeurs du ou des paramètres de complexité,
5. optimisation de la complexité
6. estimation et conservation de la prévision "optimale".

3.1 Régression logistique

Une méthode linéaire n'est évidemment pas adaptée à la forme particulière des données. D'autre part, comme seule deux variables sont concernées, la complexité ne peut être optimisée.

```
# estimation du modèle
mod.logit=glm(classe~.,data=cloud,family=binomial)
# prévision des points du plan
pred.logit=predict(mod.logit,newdata=test)>0.5
# représentation des prévisions des classes
plot(test, col = c("pink", "cyan")
    [as.numeric(pred.logit)+1], pch = 19,cex=.5)
```

Un modèle linéaire est de toute évidence inadapté.

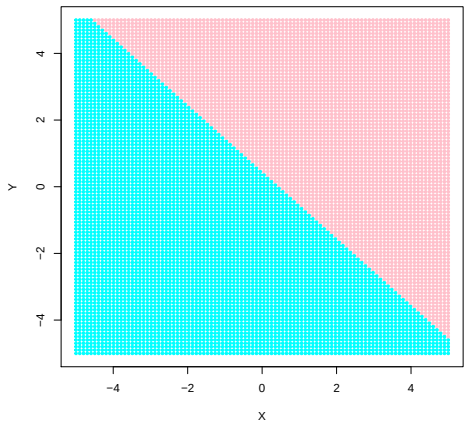

FIGURE 2 – Discrimination par la régression logistique

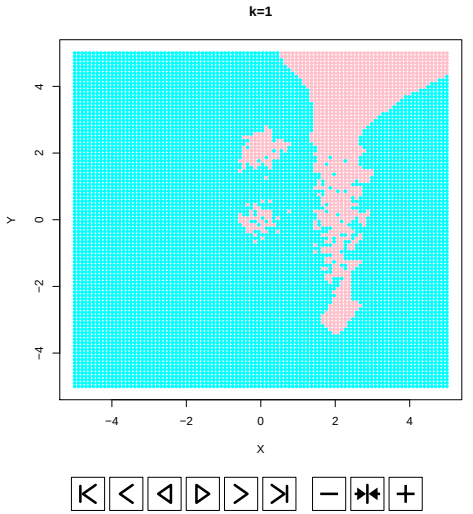

FIGURE 3 – Frontière des plus proches voisins pour différentes valeurs de k

3.2 Analyse discriminante par $k - nn$

Prévision et tracés pour plusieurs valeurs de k avant optimisation.

```
library(class)
# k "petit"
prev.knn = knn(cloud[,2:3], test, cloud[,1],k=1)
plot(test, col = c("pink", "cyan")
     [as.numeric(prev.knn)],pch = 19,cex=.5,
     main="k=1")
```

```
# k "grand"
prev.knn = knn(cloud[,2:3], test, cloud[,1],k=60)
plot(test, col = c("pink", "cyan")
    [as.numeric(prev.knn)],pch = 19,cex=.5,
    main="k=60")
```

```
# choix k "optimal"
prev.knn = knn(cloud[,2:3], test, cloud[,1],k=30)
plot(test, col = c("pink", "cyan")
    [as.numeric(prev.knn)],pch = 19,cex=.5,
    main="k=30")
```

Détermination de k "optimal" par validation croisé.

```
# Optimisation de k
library(e1071)
plot(tune.knn(cloud[,2:3],cloud[,1],
    k=seq(2,50, by=2)))
```

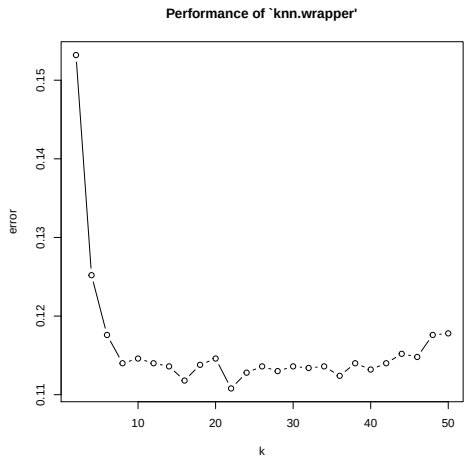

FIGURE 4 – Estimation de l erreur par validation croisée fonction de k

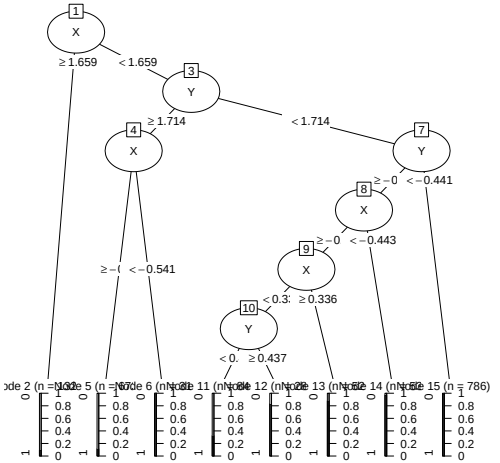

FIGURE 5 – Arbre binaire de discrimination pour une forte pénalité

3.3 Arbre binaire de discrimination

L'optimisation de la complexité (paramètre `cp`) est opérée de façon directe comme suggérée dans la librairie `rpart`.

```
library(rpart)
# forte pénalité
mod.tree=rpart(classe~.,data=cloud,
   parms=list(split="information"),cp=0.01)
# summary(mod.tree)
# en commentaire car trop bavarde
# Tracé de l'arbre
library(partykit) # si java est bien installé

## Loading required package: grid

plot(as.party(mod.tree))
```

```
# sinon
# plot(mod.tree)
# text(mod.tree)
# ou dans un fichier postscript
post(mod.tree)
```

Prévisions.

```
pred.tree=predict(mod.tree,newdata=test,
   type="class")
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.tree)], pch = 19,cex=.5,
   main="CP=0.01")
```

```
# pénalité nulle
mod.tree=rpart(classe~.,data=cloud,
                parms=list(split="information"),cp=0)
```

```
pred.tree=predict(mod.tree,newdata=test,
                  type="class")
plot(test, col = c("pink", "cyan")
     [as.numeric(pred.tree)], pch = 19,cex=.5,
     main="CP=0")
```

```
# "optimisation" de cp par validation croisée
xmat = xpred.rpart(mod.tree)
# Comparaison des valeurs prédites et observées
xerr=as.integer(cloud[,"classe"])!= xmat
# Calcul et affichage des estimations des taux d'erreur
# apply(xerr, 2, sum)/nrow(xerr)
# recherche du minimum
cpopt=which.min(apply(xerr, 2, sum)/nrow(xerr))
opt.tree=prune(mod.tree,cp=cpopt)
pred.tree=predict(mod.tree,newdata=test,
                  type="class")
plot(test, col = c("pink", "cyan")
     [as.numeric(pred.tree)], pch = 19,cex=.5,
     main="CP=opt")
```

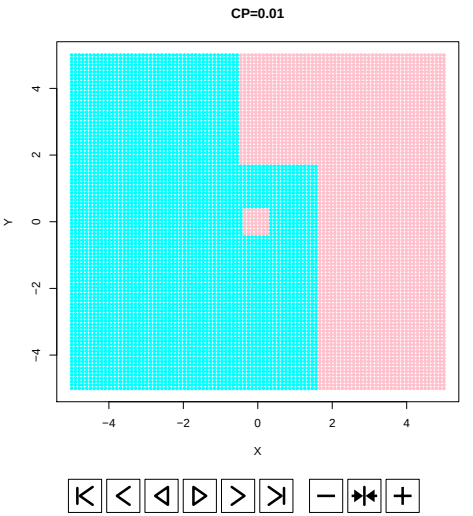

FIGURE 6 – Frontières pour des arbres avec différentes valeurs de pénalité CP

3.4 Réseaux de neurones

Au moins deux paramètres de complexité peuvent être considérés : le nombre de neurones et le pénalisation de type *ridge* (`decay`).

```
library(nnet)
# sans contrainte
mod.rn=nnet(classe~.,data=cloud,size=10,decay=0)
```

```
## # weights:  41
## initial  value 3640.162566
## iter  10 value 2603.735259
## iter  20 value 2095.837570
## iter  30 value 1652.633019
## iter  40 value 1318.581355
## iter  50 value 1285.298942
## iter  60 value 1269.670687
## iter  70 value 1265.296842
## iter  80 value 1261.210558
## iter  90 value 1258.222649
## iter 100 value 1249.829894
## final  value 1249.829894
## stopped after 100 iterations
```

```
pred.rn=predict(mod.rn,newdata=test,type="class")
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.rn)+1], pch = 19,cex=.5,
   main="size=10 et decay=0")
```

```
# choix "optimal"
mod.rn=nnet(classe~.,data=cloud,
   size=4,decay=1,maxit=200)

## # weights:  17
## initial  value 3455.224381
## iter  10 value 2609.106717
## iter  20 value 2526.776874
## iter  30 value 2272.683600
## iter  40 value 2234.781537
## iter  50 value 2226.569265
## iter  60 value 2224.590249
## iter  70 value 2224.266563
## final  value 2224.266501
## converged

pred.rn=predict(mod.rn,newdata=test,type="class")
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.rn)+1], pch = 19,cex=.5,
   main="size=4 et decay=1")
```

```
# forte pénalisation
mod.rn=nnet(classe~.,data=cloud,size=10,decay=5)

## # weights:  41
## initial  value 4403.424085
## iter  10 value 2624.185833
## iter  20 value 2593.349497
## iter  30 value 2579.682656
## iter  40 value 2576.916270
## iter  50 value 2575.959879
## iter  60 value 2575.098353
## iter  70 value 2575.064771
## final  value 2575.064171
## converged
```

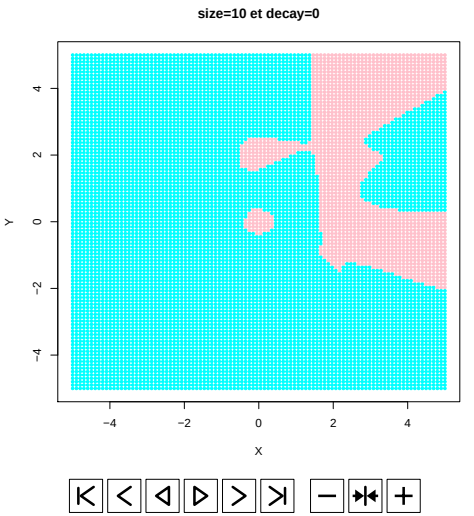

FIGURE 7 – Réseaux de neurones pour quelques valeurs de la taille et de la pénalisation

```
pred.rn=predict(mod.rn,newdata=test,type="class")
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.rn)+1], pch = 19,cex=.5,
   main="size=10 et decay=5")
```

```
# "optimisation" à exécuter avec un peu de temps
# devant soi... (retirer les ##)
##plot(tune.nnet(classe~.,data=cloud,
##    size=c(2,3,4),decay=c(1,2,3),maxit=200))
```

3.5 Agrégation de modèles

En dimension 2, les algorithmes d'agrégation ont bien moins d'intérêt d'autant que celui des forêts aléatoires ne se distingue pas du *bagging*. Seul ce

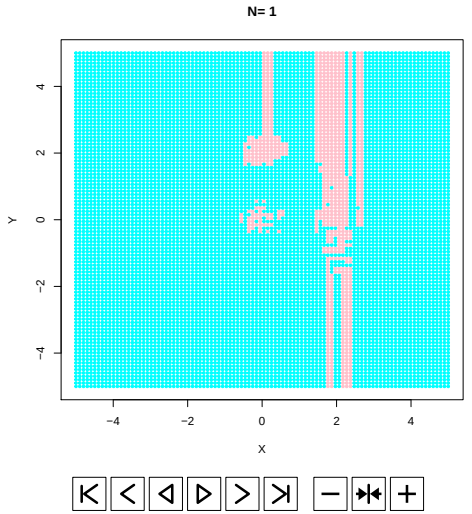

FIGURE 8 – Bogging d'arbres en faisant croître le nombre d'arbres

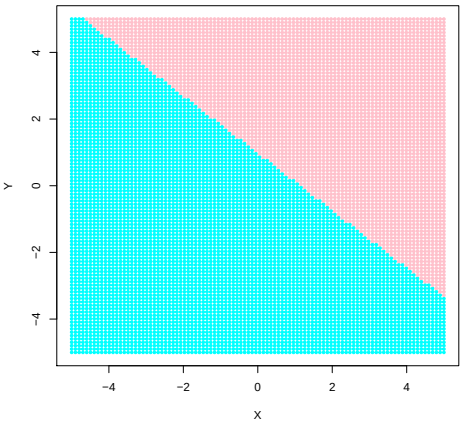

FIGURE 9 – SVM linéaire

dernier est testé. Observer l'évolution des frontières avec l'accroissement du nombre d'arbres dans l'agrégation.

```
library(ipred)
for (i in c(1,2,3,4,10,100)) {
mod.bag=bagging(classe~.,data=cloud,nbag=i)
pred.bag=predict(mod.bag,newdata=test,type="class")
plot(test, col = c("pink", "cyan")
    [as.numeric(pred.bag)], pch = 19,cex=.5,
    main=paste("N=",as.character(i)))
    }
```

3.6 Machines à vecteurs supports

Deux noyaux sont testés, celui linéaire simplement pour mémoire alors que celui gaussien (`radial`) fait intervenir deux paramètres : la pénalisation (`cost`) de mauvais classement et la "largeur" (`gamma`) du noyau gaussien. Deux valeurs relativement extrêmes sont testées avant de les optimiser.

Noyau linéaire

```
# noyau linéaire
mod.svm=svm(classe~.,data=cloud,kernel="linear")
pred.svm=predict(mod.svm,newdata=test)
plot(test, col = c("pink", "cyan")
    [as.numeric(pred.svm)], pch = 19,cex=.5)
```

Noyau gaussien

```
# noyau gaussien
mod.svm=svm(classe~.,data=cloud,
   kernel="radial",cost=5, gamma=1)
pred.svm=predict(mod.svm,newdata=test)
plot(test, col = c("pink", "cyan")
```

```
   [as.numeric(pred.svm)], pch = 19,cex=.5,
   main="cost=5 et gamma=1")
```

```
mod.svm=svm(classe~.,data=cloud,
   kernel="radial",cost=5, gamma=0.1)
pred.svm=predict(mod.svm,newdata=test)
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.svm)], pch = 19,cex=.5,
   main="cost=5 et gamma=0.1")
```

```
mod.svm=svm(classe~.,data=cloud,kernel="radial",
   cost=1, gamma=1)
pred.svm=predict(mod.svm,newdata=test)
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.svm)], pch = 19,cex=.5,
   main="cost=1 et gamma=1")
```

```
mod.svm=svm(classe~.,data=cloud,
   kernel="radial",cost=1, gamma=0.1)
pred.svm=predict(mod.svm,newdata=test)
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.svm)], pch = 19,cex=.5,
   main="cost=1 et gamma=0.1")
```

```
# optimisation
# A exécuter avec un peu de temps devant soi...
## plot(tune.svm(classe~.,data=cloud,
##   kernel="radial",cost=c(2,3,4,5,6),
#  gamma=c(1,0.7,0.5,0.3,0.1)))
mod.svm=svm(classe~.,data=cloud,
   kernel="radial",cost=4,gamma=.8)
pred.svm=predict(mod.svm,newdata=test)
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.svm)], pch = 19,cex=.5,
   main="cost=4 etgamma=.8")
```

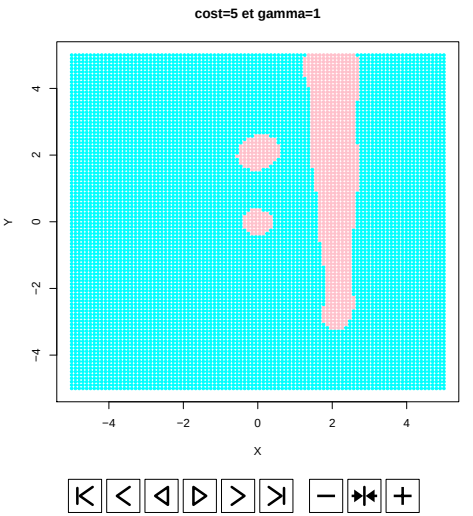

FIGURE 10 – SVM gaussien en faisant varier la pénalisation et le coefficient du noyau gaussien

4 Synthèse

A l'exception des modèles linéaires triviaux, les graphiques obtenus pour chacune des derniers "meilleurs" modèles pour chacune des méthodes sont représentés simultanément pour faciliter la comparaison.

```
par(mfcol=c(2,3))
plot(cloud[,2:3], col = as.integer(cloud[,1]),
   pch = 19,cex=.1,main="train")
plot(test, col = c("pink", "cyan")
   [as.numeric(prev.knn)],pch = 19,cex=.5,
        main="k-nn")
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.tree)],pch = 19,cex=.5,
        main="tree")
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.rn)+1],pch = 19,cex=.5,
        main="rn")
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.bag)],pch = 19,cex=.5,
        main="bag")
plot(test, col = c("pink", "cyan")
   [as.numeric(pred.svm)], pch = 19,cex=.5,
        main="svm")
```

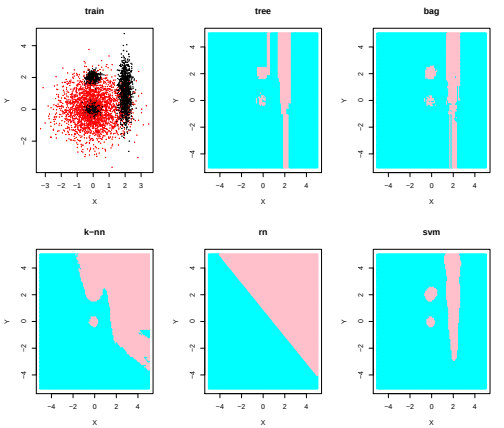

FIGURE 11 – Comparaison des frontières optimisées pour chaque algorithme.

Quelle méthode semble la mieux adaptée à des données ? Pour quelle raison ? Est-ce généralisable à d'autres données ?