Scénario: Diagnostic cytologique

Résumé

Comparaison sur le même jeu de données (Wisconsin Breast Cancer Database) des qualités de prévision de plusieurs modèles. Il s'agit de prévoir la gravité d'un cancer par régression logistique, analyse discriminante, arbre de discrimination, agrégation de modèles, SVM. Les distributions des estimations des erreurs de prévision et les courbes ROC sont comparées à la suite du tirage itératif d'un ensemble d'échantillons tests.

1 Données, objectif

Les données sont issues d'une base dévolue à la comparaison des techniques de modélisation et d'apprentissage :

```
http://www.ics.uci.edu/~mlearn/MLRepository.html
```

Elles proviennent initialement de l'hopital de l'Université du Wisconsin à Madison et ont été collectées par Wolberg et Mangasarian [1]. Elles ont été complétées puis intégrées à la bibliothèque `mlbench` de R.

Sur ces données concernant le cancer du sein, il s'agit de prévoir la nature maligne ou bénigne de la tumeur à partir des variables biologiques. Les données sont plus précisément décrites dans le support de cours. On se propose de tester différentes méthodes : régression logistique, analyse discriminante, réseau de neurones, arbre de décision, agrégation d'arbres, SVM. L'objectif final, à ne pas perdre de vue, est la comparaison de ces méthodes afin de déterminer la plus efficace pour répondre au problème de prévision. Ceci passe par la mise en place d'un protocole très strict afin de s'assurer d'un minimum d'objectivité pour cette comparaison.

Penser à conserver dans des fichiers les commandes successives utilisées ainsi que les principaux résultats tableaux et graphes.

1. Multisurface method of pattern separation for medical diagnosis applied to breast cytology. In Proceedings of the National Academy of Sciences, 87, 1990, 9193-9196.

Toutes les opérations sont réalisées dans R avec l'appui de bibliothèques complémentaires éventuellement à télécharger : `mlbench, MASS, boot, class, e1071, rpart, nnet, ipred, gbm, randomForest`

2 Extraction des échantillons apprentissage et test

2.1 Rappel : protocole de comparaison

La démarche mise en œuvre enchaîne les étapes suivantes :

i. Après une éventuelle première étape descriptive uni ou multidimensionnelle visant à repérer les incohérences, les variables non significatives, les individus non concernés…et à étudier les structures des données, procéder à un tirage aléatoire d'un échantillon *test* qui ne sera utilisé que lors de la *dernière étape*. Remarque : cette étape exploratoire n'est pas nécessaire sur ces données publiques déjà pré traitées.

ii. Sur la partie restante qui sera découpée en échantillon d'*apprentissage (des paramètres du modèle) et échantillon de* validation (pour estimation sans biais du taux de mauvais classés), optimiser les choix afférents à chacune des méthodes de discrimination :
   - variables et méthode pour l'analyse discriminante,
   - variables et interactions à prendre en compte dans la régression logistique,
   - nombre de nœuds dans l'arbre de classification,
   - architecture (nombre de couches, de neurones par couche, fonctions de transferts, nombre de cycles…) du perceptron.
   - algorithme d'agrégation
   - noyau et complexité des SVM

   Remarques :
   - En cas d'échantillon petit face au nombre des variables il est recommandé d'itérer la procédure de découpage par validation croisée, c'est le cas de ces données, afin d'estimer les erreurs de classement avec des variances plus faibles.

iii. Comparaison finale des qualités de prévision sur la base du taux de mal classés pour le seul échantillon test qui est resté à l'écart de tout effort ou acharnement pour l'optimisation des modèles.

Attention, ne pas "tricher" en modifiant le modèle obtenu lors de l'étape précédente afin d'améliorer le résultat sur l'échantillon test !

2.2 Prise en compte des données

Ces données, souvent utilisées comme benchmark pour comparer des méthodes, sont accessibles dans une bibliothèque de R.

```
# Traitement des données breastcancer
# liste des variables :
# Cl.thickness Cell.size Cell.shape
# Marg.adhesion Epith.c.size  Bare.nuclei
# Bl.cromatin Normal.nucleoli Mitoses
# Chargement de la librairie contenant des données
library(mlbench)
data(BreastCancer) # Déclaration des données
summary(BreastCancer) # résumé des données
```

Remarquer le type des variables, la présence d'une colonne d'identificateurs inutiles pour la suite ainsi que celle de quelques données manquantes risquant de poser des problèmes pour certaines techniques de modélisation. Celles-ci étant peu nombreuses sur une seule variable, les observations correspondantes sont simplement supprimées.

```
# retirer la colonne des identificateurs
data=BreastCancer[,-1]
# retirer les 16 valeurs manquantes
data=data[!is.na(data[,"Bare.nuclei"]),]
summary(data)
```

2.3 Extraction des échantillons

Le programme ci-dessous réalise l'extraction du sous-ensemble des données d'apprentissage et de test. Attention, chaque étudiant ou binôme tire un échantillon différent ; il est donc "normal" de na pas obtenir les mêmes modèles pour chaque méthode.

Utiliser trois chiffres au hasard, en remplacement de "111" ci-dessous, comme initialisation du générateur de nombres aléatoires.

```
set.seed(111) # initialisation du générateur
# Extraction des échantillons
test.ratio=.2   # part de l'échantillon test
npop=nrow(data) # nombre de lignes dans les données
nvar=ncol(data) # nombre de colonnes
# taille de l'échantillon test
ntest=ceiling(npop*test.ratio)
# indices de l'échantillon test
testi=sample(1:npop,ntest)
# indices de l'échantillon d'apprentissage
appri=setdiff(1:npop,testi)
```

Construction des échantillons avec les variables explicatiues qualitatives.

```
# construction de l'échantillon d'apprentissage
datapq=data[appri,]
# construction de l'échantillon test
datestq=data[testi,]
summary(datapq) # vérifications
summary(datestq)
```

Les données (variables explicatives) se présentent sous la forme de variables ordinales. On verra que ceci posera des problèmes à certaines méthodes lors de l'estimation du modèle : 10, modalités pour 10 variables cela fait par exemple (1+9)*9 paramètres à estimer en régression logistique. On se propose de "tricher" en considérant les variables ordinales comme quantitatives entières. C'est à dire qu'elles sont associées à des "scores" évalués par le biologiste sur une échelle. Le modèle utilisé s'en trouve moins "flexible", il est défini par moins de paramètres et peut conduire à une meilleure efficacité en prévision.

Ceci nécessite de transformer les données pour en changer le type. A l'exception de la 10ème qui est la variable binaire à prévoir, les colonnes de type facteur sont transformées en un vecteur d'entiers puis en une matrice et enfin à nouveau en un "data frame". Les noms des variables sont conservées.

```
datar=data.frame(matrix(as.integer
   (as.matrix(data[,1:9])),ncol=9,dimnames=
   dimnames(data[,1:9])),Class=data[,10])
```

Il suffit ensuite de recréer les mêmes échantillons d'apprentissage et de test :

```
 # construction de l'échantillon d'apprentissage
datapr=datar[appri,]
# construction de l'échantillon test
datestr=datar[testi,]
summary(datapr) # vérifications
summary(datestr)
```

3 Régression logistique

Cette section a pour objectif d'appliquer les procédures. Différentes stratégies de choix de modèle sont utilisées. Les erreurs de prévision sont estimées sur l'échantillon test.

3.1 Sur les variables "qualitatives"

La première exécution (modèle complet) soulève quelques soucis de convergence :

```
library(MASS) # chargement de la bibliothèque
pr.glm = glm(Class~.,family=binomial,data=datapq)
anova(pr.glm,test="Chisq")
```

Le logiciel affiche des "warning" dus à un ajustement exact des probabilités entraînant un problème de convergence de l'algorithme qui trouve aussi des p-values très grandes sur certaines variables apparaissant comme inutiles. Un choix de modèle s'impose dans ce cadre.

3.2 Choix de modèle automatique

La stratégie descendante "à la main" peut être réalisée mais voici la version automatique minimisant le critère d'Akaïke (AIC). Ce critère étant différent de celui utilisé par SAS (test de Wald ou du rapport de vraisemblance), les démarches peuvent conduire à des modèles différents.

Procédure ascendante à partir du modèle constant.

```
# Procédure ''forward''
pr.glm = glm(Class~1,family=binomial,data=datapq)
pr.step <- step(pr.glm, direction="forward",
scope=list(lower=~1,upper=~Cl.thickness+Cell.size+
   Cell.shape+Marg.adhesion+Epith.c.size+Bare.nuclei+
   Bl.cromatin+Normal.nucleoli+Mitoses), trace = TRUE)
anova(pr.step,test="Chisq")
```

Procédure ascendante et descendante

```
pr.glm = glm(Class~1,family=binomial,data=datapq)
pr.step=step(pr.glm,direction="both",
   scope=list(lower=~1,upper=~Cl.thickness+Cell.size+
   Cell.shape+Marg.adhesion+Epith.c.size+Bare.nuclei+
   Bl.cromatin+Normal.nucleoli+Mitoses), trace = TRUE)
anova(pr.step,test="Chisq")
```

Comparer les modèles, leur critère d'Akaïke. Il y a toujours une série de "warnings".

3.3 Variables "quantitatives"

Procedure automatique descendante à partir du modèle complet. Attention, `datapq` est remplacé par *datapr* dans les programmes précédents. Tester les différentes procédures de sélection avec ces données.

```
pr.glm = glm(Class~.,family=binomial,data=datapr)
anova(pr.glm,test="Chisq")
pr.step <- step(pr.glm, trace = TRUE)
anova(pr.step,test="Chisq")
```

3.4 Comparaison des qualités de prévision des modèles

Pour chacun des modèles obtenus, il est intéressant d'estimer l'erreur par resubstitution ou erreur apparente et l'erreur par validation croisée.

```
# Modele qualitatif complet
pr1.glm=glm(Class~.,family=binomial,data=datapq)
# matrice de confusion
table(pr1.glm$fitted.values>0.5,
   datapq$Class=="malignant")
```

```
# Modele quantitatif complet
pr2.glm=glm(Class~.,family=binomial,data=datapr)
# matrice de confusion
table(pr2.glm$fitted.values>0.5,
   datapr$Class=="malignant")
```

Commentaire sur ces erreurs. Faire la même chose avec les modèles sélectionnés comme par exemple :

```
pr3.glm = glm(Class~Cl.thickness+Cell.shape+
   Marg.adhesion+Bare.nuclei+Normal.nucleoli+Mitoses,
   family=binomial,data=datapr)
anova(pr3.glm,test="Chisq")
table(pr3.glm$fitted.values>0.5,
   datapr$Class=="malignant")
```

L'estimation des erreurs par validation croisée est calculée en utilisant une fonction proposée dans la bibliothèque `boot`. Cette fonction peut rencontrer des difficultés à cause de la répartition très particulière des données qui posent des problèmes de convergence à l'algorithme de la fonction glm. Des ajustements dans le choix des variables sont éventuellement nécessaires afin d'aboutir à des résultats raisonnables. Attention ces ajustements peuvent dépendre de l'échantillon initialement tiré.

```
library(boot) # chargement d'une autre bibliothèque
# Fonction calculant le taux d'erreur
# Cette fonction est utilisée par la procédure
# de validation croisée cv.glm
cout = function(r, pi=0)
    mean(abs(as.integer(r)-pi)>0.5)
# Application de cette fonction
# pour l'erreur d'ajustement :
1-cout(datapq$Class,pr1.glm$fitted.values)
1-cout(datapr$Class,pr2.glm$fitted.values)
```

Enfin, évaluation de l'erreur par validation croisée en 10 groupes pour le modèle restreint :

```
cv.glm(datapr, pr3.glm, cout,K=10)$delta[1]
```

Comparer les erreurs estimées par validation croisée et celles par resubstitution pour chacun des modèles "calculables" puis les erreurs entre les modèles estimées par validation croisée. Conclusion.

3.5 Prévision de l'échantillon test

A la fin de ces procédures, appliquer le meilleur modèle trouvé, au sens de la minimisation de l'erreur estimée par validation croisée, à l'échantillon test. Calculer l'erreur sur cet échantillon. Attention, ce n'est pas nécessairement celui retenu ci-dessous.

```
# prevision
predr.glm=predict(pr3.glm,newdata=datestr)
# matrice de confusion
table(predr.glm>0.5,datestr[,"Class"])
```

Noter cette erreur pour les comparaisons à venir.

4 Analyse discriminante

4.1 Introduction

L'objectif est de comparer les trois méthodes d'analyses discriminantes disponibles dans R : `lda` paramétrique linéaire (homoscédasticité), `lqa` paramétrique quadratique (hétéroscédasticité) sous hypothèse gaussienne et celle non-paramétrique des k plus proches voisins.

Attention, ces techniques n'acceptent par principe que des variables explicatives ou prédictives quantitatives. Néanmoins, une variable qualitative à deux modalités, par exemple le sexe, peut être considérée comme quantitative sous la forme d'une fonction indicatrice prenant ses valeurs dans $\{0, 1\}$ et, de façon plus "abusive", une variable ordinale est considérée comme "réelle". Dans ce denier cas, il ne faut pas tenter d'interpréter les fonctions de discrimination, juste considérer des erreurs de prévision.

4.2 Estimations

La bibliothèque standard de R (MASS) pour l'analyse discriminante ne propose pas de procédure automatique de choix de variable contrairement à la

procédure stepdisc de SAS.

```
library(MASS) # chargement des librairies
library(class) # pour kNN
# analyse discriminante linéaire
fitq.disl=lda(Class~.,data=datapq)
# analyse discriminante quadratique
fitq.disq=qda(Class~.,data=datapq)
fitq.knn=knn(datapq[,-10],datestq[,-10],
   datapq$Class,k=10) # k plus proches voisins
```

R signale un problème quand il cherche à inverser la matrice de covariance associée à la classe "benign". L'origine de ce problème reste mystérieux dans la mesure où la même approche de SAS (voir ci-dessous) estime cette matrice.

Noter le manque d'homogénéité des commandes de R issues de librairies différentes. L'indice de colonne négatif (-10) permet de retirer la colonne contenant la variable à prédire de type facteur. Celle-ci est mentionnée en troisième paramètre pour les données d'apprentissage.

4.3 Erreurs d'ajustement

Calculer les erreurs sur la prévision de l'échantillon d'apprentissage (erreur dite apparente, d'ajustement ou de resubstitution)

```
# erreur d'apprentissage
table(datapq[,"Class"],predict(fitq.disl,
   datapq)$class)
table(datapq[,"Class"],knn(datapq[,-10],
   datapq[,-10],datapq$Class,k=5))
```

4.4 Optimisation de k par validation croisée

Le choix du nombre de voisins k peut être optimisé par validation croisée mais la procédure proposée par la bibliothèque `class` est celle *leave-one-out* donc trop coûteuse en temps pour des gros fichiers. Il serait simple de la programmer mais une autre bibliothèque (`e1071`) propose également une batterie de fonctions de validation croisée pour de nombreuses techniques de discrimination.

```
library(e1071)
plot(tune.knn(datapq[,-10],datapq$Class,k=2:20))
```

Remarquer que chaque exécution de la commande précédente donne des résultats différents donc très instables. Choisir une valeur "raisonnable" de k et l'utiliser pour prédire l'échantillon test.

4.5 Taux d'erreur sur l'échantillon test

Les commandes suivantes calculent les tables de confusion pour chacune des méthodes d'analyse discriminante.

```
fitq.knn<-knn(datapq[,-10],datestq[,-10],
   datapq$Class,k=5)
# réestimer le modèle de discrimination linéaire
fitq.disl=lda(Class~.,data=datapq)
table(datestq[,"Class"],predict(fitq.disl,
   datestq)$class)
table(fitq.knn,datestq$Class)
```

Noter ces taux d'erreur en vue d'une comparaison.

5 Estimation d'un arbre

5.1 Introduction

Deux librairies, `tree` et `rpart`, proposent les techniques CART avec des algorithmes analogues à ceux développés dans Splus mais moins de fonctionnalités ; la librairie `rpart` fournissant des graphes plus explicites, des options plus détaillées et une procédure d'élagage plus performante est préférée. Cette fonction intègre une procédure de validation croisée pour évaluer le paramètre de pénalisation de la complexité.

5.2 Estimation

Dans le cas d'une discrimination, le critère par défaut est l'indice de concentration de Gini ; il est possible de préciser le critère d'entropie ainsi que des poids sur les observations, une matrice de coûts ainsi que des probabilités a priori (`?rpart` pour plus de détails).

D'autres paramètres peuvent être modifiés : `cp` désigne la complexité minimale pour la construction de l'arbre maximal, le nombre minimal d'observation par nœud, le nombre de validations croisées (par défaut 10)... (`?rpart.control` pour plus de détails). En fait la documentation des fonctions concernées est un peu "laconique" et il est difficile d'en comprendre tous les détails sans suivre leur évolution par les forums d'utilisateurs. C'est souvent un des travers des logiciels en accès libre mais la communauté est très "réactive" pour répondre aux questions à condition qu'elles ne soient pas trop "naïves". Il est en effet important de consulter les archives pour ne pas reposer des problèmes déjà résolus et abondamment commentés.

```
library(rpart)  # Chargement de la librairie
```

La première estimation favorise un arbre très détaillé c'est-à-dire avec un faible coefficient de pénalisation de la complexité de l'arbre et donc du nombre de feuilles important. Le critère d'hétérogénéité choisi est l'entropie.

```
fitq.tree=rpart(Class~.,data=datapq,
    parms=list(split='information'),cp=0.001)
summary(fitq.tree)  # description de l'arbre
print(fitq.tree)
plot(fitq.tree)  # Tracé de l'arbre
text(fitq.tree)  # Ajout des légendes des noeuds
```

Le contenu de l'arbre n'est pas très explicite ou alors il faudrait aller lire dans le "summary" les informations complémentaires. Un arbre plus détaillé est construit par l'intermédiaire d'un fichier postscript. Ce fichier est créé dan le répertoire de lancement de R. Différentes options sont disponibles permettant de gérer le titre et autres aspects du graphique : `?post`.

```
post(fitq.tree)
```

5.3 Elagage

Il est probable que l'arbre présente trop de feuilles pour une bonne prévision. Il est donc nécessaire d'en réduire le nombre par élagage. C'est un travail délicat d'autant que la documentation n'est pas très explicite et surtout les arbres des objets très instables.

Par échantillon de validation

Cette option, systématiquement mise en place dans SAS Enterprise Miner, n'est pas explicitement prévue dans la bibliothèque `rpart`, elle suppose des échantillons de taille importante. Elles est simulée sur ces données à titre pédagogique.

Une première étape réalise l'extraction d'un échantillon de validation comme il a été fait pour l'échantillon test :

```
set.seed(4) # autre initialisation du générateur
# Extraction des échantillons
# part de l'échantillon de validation
valid.ratio=.2
# nombre de lignes dans les données restantes
npop=nrow(datapq)
# taille de l'échantillon de validation
nvalid=ceiling(npop*valid.ratio)
# indices de l'échantillon de validation
validi=sample(1:npop,nvalid)
# indices complémentaires de l'échantillon de validation
appri=setdiff(1:npop,validi)
# construction de l'échantillon d'apprentissage restant
datap2q=data[appri,]
# construction de l'échantillon de validation
davalq=data[validi,]
summary(datap2q) # vérifications
summary(davalq)
```

Les tailles des échantillons ainsi obtenus sont relativement réduite d'où un manque de robustesse très probable de cette approche.

La deuxième étape estime le modèle sur l'échantillon d'apprentissage puis l'erreur de prévision sur l'échantillon de validation pour différentes valeurs du coefficient de pénalisation.

```
cpi=1
for(i in 1:20) {
tree_i=rpart(Class~.,data=datap2q,
   parms=list(split='information'),cp=cpi)
pred_i=predict(tree_i,newdata=davalq,type="class")
tab_i=table(pred_i,davalq$Class)
```

```
cat("cp=",cpi, " err=",(tab_i[1,2]+tab_i[2,1])/nvalid,"\n")
cpi=cpi*0.7
}
```

Exécuter plusieurs fois ces deux étapes pour différentes valeurs de l'initialisation du générateur de nombres aléatoires. Remarquer le comportement plutôt erratique du procédé causé par des échantillons trop petits. La validation croisée tente d'améliorer cette démarche en "moyennant" l'estimation de l'erreur sur plusieurs échantillons.

Par validation croisée "intégrée"

Une première façon consiste à utiliser les fonctionnalités intégrées à la fonction `rpart` qui permet de tracer la décroissance de l'estimation par validation croisée de l'erreur relative en fonction du coefficient de complexité, c'est-à-dire plus ou moins aussi aussi en fonction de la taille de l'arbre ou nombre de feuilles. Attention, cette relation entre complexité et nombre de feuilles n'est pas directe car l'erreur est calculée sur des arbres estimés à partir d'échantillons aléatoires ($k-1$ morceaux) différents et sont donc différents les uns des autres avec pas nécessairement le même nombre de feuilles, ils partagent juste le même paramètre de complexité au sein de la même famille de modèles emboîtés.

```
# après réestimation de l'arbre sur
# l'échantillon d'apprentissage initial
fitq.tree=rpart(Class~.,data=datapq,
   parms=list(split='information'),cp=0.001)
# les estimations des erreurs sont contenues dans la table
printcp(fitq.tree)
# et tracées dans le graphe :
plotcp(fitq.tree)
```

Remarquer que plusieurs exécutions donnent des résultats différents. Il faut alors choisir une valeur du coefficient de pénalisation de la complexité pour élaguer l'arbre. Attention, ce choix dépend de l'échantillon tiré. Il peut être différent de celui choisi dans les commandes ci-dessous. L'usage recommandé est de prendre la première valeur à gauche sous la ligne horizontale pointillée.

```
fitq2.tree=prune(fitq.tree,cp=0.048)
plot(fitq2.tree)
text(fitq2.tree,use.n=TRUE)
```

Élagage par validation croisée externe

Comme la procédure intégrée de validation croisée est relativement "frustre", une autre fonction est proposé permettant de mieux contrôler la décroissance de la complexité. La commande suivante calcule les prédictions obtenues pas 10-fold validation croisée pour chaque arbre élagué suivant les valeurs du coefficients de complexité donné. La séquence de ces valeurs doit être adaptée à l'exemple traité.

```
xmat = xpred.rpart(fitq.tree,xval=10,
    cp=seq(0.1,0.001,length=10))
# Comparaison de la valeur prédite avec la valeur observée
xerr=as.integer(datapq$Class)!= xmat
# Calcul et affichage des estimations des taux d'erreur
apply(xerr, 2, sum)/nrow(xerr)
```

Faire plusieurs exécutions en modifiant aussi la séquence de paramètres... Les choix sont-ils confirmés ?

Une fois le choix définitif arrêté :

Élagage puis affichage de l'arbre

```
fitq3.tree=prune(fitq.tree,cp=0.023)
post(fitq3.tree)
```

Remarquer qu'elles sont les variables extraites par l'arbre, retrouve-t-on les mêmes que celles sélectionnées par la régression logistique.

5.4 Prévision de l'échantillon test

Une fois l'arbre élagué, les commandes ci-dessous calculent la prévision de la variable `Class` pour l'échantillon test et croisent la variable prédite avec la variable observée afin de construire la matrice de confusion et donc d'estimer un taux d'erreur.

```
predq.tree=predict(fitq.tree,datestq,type="class")
table(predq.tree,datestq$Class)
```

Noter le taux d'erreur. Attention, ne plus modifier le modèle pour tenter de diminuer l'erreur sur l'échantillon test, cela conduirait à un biais et un sur-ajustement.

6 Réseaux de neurones

6.1 Introduction

Il s'agit d'estimer un modèle de type perceptron multicouche avec en entrée les variables qualitatives ou quantitatives et en sortie la variable binaire à prévoir. Des fonctions R pour l'apprentissage d'un perceptron élémentaire ont été réalisées par différents auteurs et sont accessibles sur le réseau. La librairie `nnet` de (Ripley, 1999), est limitée aux perceptrons à une couche. C'est théoriquement suffisant pour approcher d'aussi près que l'on veut toute fonction à condition d'insérer suffisamment de neurones.

La variable à expliquer étant binomiale, la fonction de transfert du neurone de sortie est choisie sigmoïdale (choix par défaut) comme toutes celles de la couche cachée. Sinon, il faudrait la déclarer linéaire. Le paramètre important à déterminer est le nombre de neurones sur la couche cachée parallèlement aux conditions d'apprentissage (temps ou nombre de boucles). Une alternative à la détermination du nombre de neurones est celle du `decay` qui est un paramètre de régularisation analogue à celui utilisé en régression ridge. Il pénalise la norme du vecteurs des paramètres et contraint ainsi la flexibilité du modèle. Très approximativement il est d'usage de considérer, qu'en moyenne, il faut une taille d'échantillon d'apprentissage 10 fois supérieure au nombre de poids c'est-à-dire au nombre de paramètres à estimer. On remarque qu'ici la taille de l'échantillon d'apprentissage (546) est modeste pour une application raisonnable du perceptron. Seuls des nombres restreints de neurones peuvent être considérés et sur une seule couche cachée.

Attention, le nombre de paramètres associés à des variables qualitatives dépend du nombre de modalités. Ceci est important à considérer suivant que le réseau est construit sue les variables explicatives oridanles ou quantitatives. Le nombre de neurones ne peut être le même dans les deux cas.

6.2 Estimation

```
library(MASS)
library(nnet)
# apprentissage
fitq.nnet=nnet(Class~.,data=datapq,size=3,decay=1)
summary(fitq.nnet)
```

La commande donne la "trace" de l'exécution avec le comportement de la convergence mais le détail des poids de chaque entrée de chaque neurone ne constituent pas des résultats très explicites ! Contrôler le nombre de poids estimés.

Voici l'erreur apparente ou par resubstituation mesurant la qualité de l'ajustement.

```
#matrice de confusion
table(fitq.nnet$fitted.values>0.5,datapq$Class)
```

Compte-tenu de la taille restreinte de l'échantillon, une approche de type "validation croisée" est nécessaire afin de tenter d'optimiser les choix en présence : nombre de neurones, "decay" et éventuellement le nombre max d'itérations.

6.3 nombre de neurones

De base, il n'y a pas de fonction dans la librairie `nnet` permettant d'optimiser le nombre de neurones de la couche cachée par validation croisée. Voici, à titre pédagogique, le texte d'une fonction de k-validation croisée adaptée aux réseaux de neurones.

i. Ouvrir l'éditeur pour la création de la fonction : `fix(CVnn, editor="emacs"))` avec l'éditeur emacs ou un autre (kwrite, kate...). L'éditeur par défaut étant vi.

ii. Rentrer le texte de la fonction. Les espaces ne sont pas importants, la mise en page est automatique.

```
function(formula, data, size, niter = 1,
   nplis = 10, decay = 0, maxit = 100)
{
   n = nrow(data)
   tmc <- 0
   un <- rep(1, n)
   ri <- sample(nplis, n, replace = T)
   cat(" k= ")
   for(i in sort(unique(ri))) {
    cat(" ", i, sep = "")
    for(rep in 1:niter) {
     learn <- nnet(formula, data[ri != i,  ],
```

```
      size = size, trace = F, decay = decay,
      maxit = maxit)
      tmc = tmc + sum(un[(data$Class[ri == i]
      == "malignant") !=(predict(learn,
      data[ri == i,  ]) > 0.5)])
        }
    }
    cat("\n", "Taux de mal classes")
    tmc/(niter * length(unique(ri)) * n)
}
```

iii. Sauver le texte, quitter l'éditeur

iv. Si des messages d'erreur apparaissent, `CVnn=edit(editor="emacs")` permet de relancer l'éditeur pour les corriger.

Le paramètre `niter` permet de répliquer l'apprentissage du réseau et de moyenner les résultats afin de réduire la variance de l'estimation de l'erreur. Il est par défaut de 1 mais peut-être augmenté à condition de se montrer plus patient. R, langage interprété comme matlab, n'est pas particulièrement véloce lorsque plusieurs boucles sont imbriquées. Attention, cette fonction dépend des données utilisées mais elle pourrait être facilement généralisées en ajoutant des paramètres.

Tester la fonction ainsi obtenue et l'exécuter pour différentes valeurs de `size` (3, 4, 5) et `decay` (0, 1, 2).

Attention, l'initialisation de l'apprentissage d'un réseau de neurone comme celle de l'estimation de l'erreur par validation croisée sont aléatoires. Chaque exécution donne donc des résultats différents. À ce niveau, il serait intéressant de construire un plan d'expérience à deux facteurs (ici, les paramètres de taille et decay) de chacun trois niveaux. Plusieurs réalisations pour chaque combinaison des niveaux suivies d'un test classique d'anova permettraient de se faire une idée plus juste de l'influence de ces facteurs sur l'erreur.

```
CVnn(Class~.,data=datapq,size=3, decay=1)
...
# exécuter pour différents couples
```

Noter la taille et le "`decay`" optimaux. Il faudrait aussi faire varier le nombre total d'itérations. Cela risque de prendre un peu de temps !

6.4 Comparaison avec les variables quantitatives

Refaire les traitements en remplaçant les variables ordinales par leur transformation en valeurs entières. Il suffit de remplacer `datapq` par `datapr` dans les programmes précédents. Vérifier l'impact de ce choix sur le nombre de paramètres ou "poids" estimés dans le modèle. Comparer les taux d'erreur estimés par validation croisée, retenir le meilleur modèle et l'appliquer à l'échantillon test.

6.5 Test

Ré-estimer le réseau avec les paramètres optimaux puis estimer le taux d'erreur sur l'échantillon test. Noter le taux d'erreur.

```
fitr.nnet<-nnet(Class~.,data=datapr,size=3,decay=1)
# code ''true'' la prévision de ''Coui''
pred.test<-predict(fitr.nnet,datestr)>0.5
table(pred.test,datestr$Class=="malignant")
```

7 Agrégation de modèles

7.1 Introduction

Des sections précédentes ont permis d'expérimenter les techniques maintenant classiques de construction d'un modèle de prévision assorties de leur problème récurrent liés à l'optimisation de la complexité du modèle. Cette section aborde d'autres stratégies récemment développées dont l'objectif est de s'affranchir de ce problème de choix, par des méthodes se montrant pas ou moins sensibles au sur-apprentissage ; c'est le cas des algorithmes d'agrégation de modèles.

Sur le plan logiciel, R montre dans cette situation tout son intérêt. La plupart des techniques récentes sont en effet expérimentées avec cet outil et le plus souvent mises à disposition de la communauté scientifique sous la forme d'une librairie afin d'en assurer la "promotion". Pour les techniques d'agrégation de modèles, nous pouvons utiliser les librairies `gbm` et `randomForest` respectivement réalisées par Greg Ridgeway et Leo Breiman. Ce n'est pas systématique, ainsi J. Friedman a retiré l'accès libre à ses fonctions (MART) et créé son entreprise (Salford).

Objectifs

i. tester le bagging et le choix des ensembles de variables ainsi que le nombre d'échantillons considérés,

ii. étudier l'influence des paramètres (profondeur d'arbre, nombre d'itérations, shrinkage) sur la qualité de la prévision par boosting ;

iii. même chose pour les forêts aléatoires (nb de variables mtry, nodesize).

iv. Expérimenter les critères de Breiman qui lui permettent de mesurer l'influence des variables au sein d'une famille agrégée de modèles. Les références bibliographiques sont accessibles sur le site de l'auteur :

`www.stat.Berkeley.edu/users/breiman`

Méthodologie

Sur le plan méthodologique, il serait intéressant, comme pour les réseaux de neurones, de monter un plan d'expérience pour évaluer la sensibilité des techniques aux valeurs des paramètres contrôlant l'exécution des algorithmes (nombre d'itérations, profondeur d'arbre, nombre de variables extraites à chaque décision pour la construction d'un nœud...

Remarque : les arbres ne font pas la différence entre variables quantiatives et qualitatives ordinales : il n'est donc pas nécessaire d'envisager deux traitements comme pour certaines méthodes.

7.2 Bagging

En utilisant la fonction `sample` de R, il est très facile d'écrire un algorithme de bagging. Il existe aussi une librairie qui propose des exécutions plus efficaces. Par défaut, l'algorithme construit une famille d'abres complets (cp=0) et donc de faible biais mais de grande variance. L'erreur out-of-bag permet de contrôler le nombre d'arbres ; un nombre raisonnable semble suffire ici.

Estimations

L'utilisation est immédiate :

```
library(ipred)
bagging(Class~., nbag=50, data=datapq,coob=TRUE)
bagging(Class~., nbag=25, data=datapq,coob=TRUE)
```

Le résultat est alétoire et chaque exécution donne des résultats différents. Néanmoins, ceux-ci semblent relativement stables et peu sensibles au nombre d'arbre. Enfin, il est possible de modifier certaines paramètres de rpart pour juger de leur influence : ci-dessous, l'élagage de l'arbre.

```
bagging(Class~., nbag=50,control=
   rpart.control(cp=0.1), data=datapq,coob=TRUE)
```

Cela nécessite une optimisation du choix du paramètre. Remarquer néanmoins que le nombre d'arbres (nbag) n'est pas un paramètre "sensible" et qu'il suffit de se contenter d'arbres "entiers".

Prévision de l'échantillon test

En utilisant l'erreur oob comme une estimation par validation croisée, retenir la meilleure combinaison de paramètres qui n'est pas nécessairement celle ci-dessous puis réestimer le modèle avant de construire la matrice de confusion pour l'échantillon test.

```
fit.bag=bagging(Class~., nbag=50, data=datapq)
pred.test=predict(fit.bag,newdata=datestq,type="class")
table(pred.test,datestq$Class)
```

7.3 Boosting

Deux librairies proposent des versions relativement sophistiquées des algorithmes de boosting dans R. La librairie `boost` propose 4 approches : adaboost, bagboost et deux logitboost. Développées pour une problématique particulière : l'analyse des données d'expression génomique, elle n'est peut-être pas complètement adaptée aux données étudiées ; elles se limitent à des prédicteurs quantitatifs et peut fournir des résultats étranges. La librairie `gbm` lui est préférée ; elle offre aussi plusieurs versions dépendant de la fonction coût choisie. Cependant, le chargement de cette librairie peut poser des problèmes en fonction de l'environnement système.

La variable à prévoir doit être codée numériquement (0,1) pour cette implémentation. Le nombre d'itérations, ou nombre d'arbres, est paramétré ainsi qu'un coefficient de "shrinkage" contrôlant le taux ou pas d'apprentissage. Attention, par défaut, ce paramètre a une valeur très faible (0.001) et il faut un nombre important d'itérations pour atteindre une estimation raisonnable. La

qualité est visualisée par un graphe représentant l'évolution de l'erreur d'apprentissage ; d'autre part, une procédure de validation croisée est incorporée qui fournit le nombre optimal d'itérations à considérer.

Apprentissage sans shrinkage

```
library(gbm)
fit.boost=gbm(as.numeric(Class)-1~., data=datapq,
distribution="adaboost",n.trees=500, cv.folds=10,
n.minobsinnode = 5,shrinkage=1,verbose=FALSE)
best.iter=gbm.perf(fit.boost,method="cv")
# nombre optimal d'itérations par validation croisée
print(best.iter)
plot(fit.boost$cv.error)
```

Le premier graphe représente l'erreur d'apprentissage tandis que le deuxième celle estimée par validation croisée. Ces erreurs présentent des comportements étranges.

Shrinkage

Les exécutions suivantes permettent d'évaluer l'impact du coefficient de shrinkage :

```
fit1.boost=gbm(as.numeric(Class)-1~., data=datapq,
distribution="adaboost",n.trees=500, cv.folds=10,
n.minobsinnode = 5,shrinkage=0.01,verbose=FALSE)
best1.iter=gbm.perf(fit1.boost,method="cv")
print(best1.iter)
plot(fit1.boost$cv.error)

fit2.boost=gbm(as.numeric(Class)-1~., data=datapq,
distribution="adaboost",n.trees=500, cv.folds=10,
n.minobsinnode = 5,shrinkage=0.05,verbose=FALSE)
best2.iter=gbm.perf(fit2.boost,method="cv")
print(best2.iter)
plot(fit2.boost$cv.error)
```

```
fit3.boost=gbm(as.numeric(Class)-1~., data=datapq,
distribution="adaboost",n.trees=500, cv.folds=10,
n.minobsinnode = 5,shrinkage=0.1,verbose=FALSE)
best3.iter=gbm.perf(fit3.boost,method="cv")
print(best3.iter)
plot(fit3.boost$cv.error)
```

Comparer les valeurs des erreurs par validation croisée associées au nombre optimal d'itération pour chaque valeur du taux d'apprentissage :

```
fit1.boost$cv.error[best.iter]
fit2.boost$cv.error[best2.iter]
fit3.boost$cv.error[best2.iter]
```

et déterminer le "bon" choix.

Echantillon test

On peut s'assurer de l'absence d'un phénomènre de surapprentissage critique en calculant puis traçant l'évolution de l'erreur sur l'échantillon test :

```
test=numeric()
for (i in 10:1000){
pred.test=predict(fit2.boost,newdata=datestq,n.trees=i)
taux=table(as.factor(sign(pred.test)),datestq$Class)
test=c(test,(taux[1,2]+taux[2,1])/137)}

plot(10:1000,test,type="l")
# nb optimal par validation croisée
abline(v=best1.iter)
```

La prévision de l'échantillon test et de la matrice de confusion associée sont obtenus par les commandes :

```
pred.test=predict(fit2.boost,newdata=datestq,
   n.trees=best2.iter)
table(as.factor(sign(pred.test)),datestq$Class)
```

7.4 Forêt aléatoire

Le programme est disponible dans la librairie `randomForest`. Il est écrit en fortran, donc efficace en terme de rapidité d'exécution, mais facile à utiliser grâce à une interface avec R. Les paramètres et résultats sont explicités dans l'aide en ligne.

Estimation

```
# charger la librairie
library(randomForest)
# aide en ligne
?randomForest
# Avec les seules variables quantitatives :
fit=randomForest(Class~.,data=datapq,
            xtest=datestq[,-10],
            ytest=datestq[,"Class"],do.trace=20,
            importance=TRUE,norm.vote=FALSE)
print(fit)
```

Observer l'évolution des erreurs (oob, test) en fonction du nombre d'arbres.

Il n'y a pas de modèle a interpréter mais une liste de coefficients d'importance associés à chaque variable :

```
print(round(fit$importance, 2))
```

Comparer avec les variables mises en évidence par la régfression logistique ou un arbre de décision.

7.5 Echantillon test

Estimer avec les valeurs "optimales" des paramètres :

```
fit.rf=randomForest(Class~.,data=datapq,
            xtest=datestq[,-10],
            ytest=datestq[,"Class"],do.trace=20,
            mtry=2,ntree=400,norm.vote=FALSE)
```

Matrice de confusion pour l'échantillon test :

```
table(fit.rf$test$predicted,  datestq$Class)
```

Quelle stratégie d'agrégation de modèles vous semble fournir le meilleur résultat de prévision ? Est-elle plus efficace que les modèles classiques expérimentés auparavant ?

8 Séparateurs à Vaste Marge (SVM)

8.1 Introduction

Cette séance propose d'aborder une nouvelle famille d'algorithmes : les SVM ou (*Support Vector Machines* traduit par Séparateurs à Vaste Marge ou machine à vecteurs support) dont le principe fondateur est d'intégrer l'optimisation de la complexité d'un modèle à son estimation. Une bibliothèque de R, réalisée par Chang, Chih-Chung et Lin Chih-Jen, est destinée à cette approche ; elle est intégrée au package `e1071`.

Au delà du principe fondateur de recherche de parcimonie (nombre de supports) incorporé à l'estimation, il n'en reste pas moins que cette approche laisse, en pratique, un certain nombre de choix et réglages à l'utilisateur. Il est donc important d'en tester l'influence sur la qualité des résultats.

i. choix du paramètre de régularisation ou pondération d'ajustement,
ii. choix du noyau,
iii. le cas échéant, choix du paramètre associé au noyau : largeur d'un noyau gaussien, degré d'un noyau polynomial...

Notons la même remarque qu'avec les techniques précédentes sur l'intérêt à mettre en œuvre une approche "plan d'expérience" voire même "surface de réponse" afin d'optimiser le choix des paramètres.

Les exécutions et tests sont réalisés à partir des fichiers de données (`datapr, datestr`) des séances précédentes. En effet, seules des variables explicatives quantitatives peuvent être prises en compte avec les noyaux et options de base. La prise en compte de variables explicatives qualitatives nécessitent l'utilisation d'un noyau spécifique.

Options par défaut

Une première exécution en utilisant simplement les options par défaut de l'algorithme conduit aux résultats suivants :

```
fit.svm=svm(Class~., data=datapr)
```

```
summary(fit.svm) # affichage des options
fit.pred=predict(fit.svm,data=datapr)
table(fit.pred,datapr$Class) # erreur d'apprentissage
```

Les résultats semblent corrects. Peut-on les améliorer en ajustant les paramètres et surtout en estimant l'erreur par validation croisée ?

Choix d'options par validation croisée

Malgré les assurances théoriques concernant ce type d'algorithme, les résultats dépendant fortement du choix des paramètres. Nous nous limiterons d'abord au noyau gaussien (choix par défaut) ; la fonction `tune.svm` permet de tester facilement plusieurs situations en estimant la qualité de prévision par validation croisée sur une grille. Le temps d'exécution est un peu long... en effet, contrairement à beaucoup d'algorithmes de modélisation, la complexité de l'algorithme de résolution des SVM croît très sensiblement avec le nombre d'observations mais moins avec le nombre de variables.

```
obj = tune.svm(Class~., data = datapr,
   gamma = 2^(-8:-5), cost = 2^(-4:2))
summary(obj)
plot(obj)
```

Modifier éventuellement les bornes de la grille. Noter le couple optimal de paramètres (difficile car cela change à chaque exécution !).

Autres noyaux

Les choix précédents utilisaient le noyau gaussien par défaut. Voici la syntaxe pour utilisation d'un noyau polynomial. Tester également d'autres valeurs des paramètres.

```
fit.svm=svm(Class~., data=datapr,
   kernel="polynomial",gamma=0.02,cost=1,degree=3)
```

Même chose pour un noyau sigmoïdal :

```
fit.svm=svm(Class~., data=datapr,
   kernel="sigmoid",gamma=0.02,cost=1)
```

Le changement de noyau a-t-il un effet sensible ?

8.2 Echantillon test

Exécuter l'estimation en utilisant le noyau associé avec les valeurs optimales des paramètres, puis prévoir l'échantillon test.

```
fit.svm=svm(Class~., data=datapr,gamma=0.015,cost=0.5)
fit.pred=predict(fit.svm,newdata=datestr)
# erreur sur échantilon test
table(fit.pred,datestr$Class)
```

Comparer avec les autres méthodes.