TP ozone : réseaux neuronaux

Résumé

Prévision du pic d'ozone par réseaux de neurones.

1 Introduction

Il s'agit d'estimer un modèle de type "perceptron" avec en entrée les variables qualitatives ou quantitatives et en sortie la variable 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.

Comme pour les arbres, la variable à expliquer est soit quantitative soit qualitative ; la fonction de transfert du neurone de sortie d'un réseau doit être adaptée en conséquence. Elle est choisie linéaire dans le cas quantitatif et sigmoïdale (choix par défaut) comme toutes celles de la couche cachée dans le cas qualitatif. 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 (832) est modeste pour une application raisonnable du perceptron. Seuls des nombres restreints de neurones peuvent être considérés et sur une seule couche cachée.

2 Cas de la régression

```
library(MASS)
library(nnet)
```

```
# apprentissage
nnet.reg=nnet(O3obs~.,data=datappr,size=5,decay=1,
   linout=TRUE,maxit=500)
summary(nnet.reg)
```

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.

L'optimisation des paramètres nécessite encore le passage par la validation croisée. Il n'y a pas de fonction dans la librairie `nnet` permettant permettant de le faire mais la fonction `tune.nnet` de la librairie `e1071` est adaptée à cette démarche.

```
library(e1071)
plot(tune.nnet(O3obs~.,data=datappr,size=c(2,3,4),
   decay=c(1,2,3),maxit=200,linout=TRUE))
plot(tune.nnet(O3obs~.,data=datappr,size=4:5,decay=1:10)
```

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 ! Noter également que chaque exécution donne des résultats différents... il n'est donc pas très utile d'y passer beaucoup de temps, surtout que les temps d'exécution sont assez long !

Ré-estimer le modèle supposé optimal avant de tracer le graphe des résidus.

```
nnet.reg=nnet(O3obs~.,data=datappr,size=3,decay=2,
   linout=TRUE,maxit=200)
# calcul et graphe des résidus
fit.nnetr=predict(nnet.reg,data=datappr)
res.nnetr=fit.nnetr-datappr[,"O3obs"]
plot.res(fit.nnetr,res.nnetr)
```

3 Cas de la discrimination

```
# apprentissage
nnet.dis=nnet(DepSeuil~.,data=datappq,size=5,decay=1)
summary(nnet.reg)
```

Calculer l'erreur apparente ou par re-substitution mesurant la qualité de l'ajustement.

```
#matrice de confusion
table(nnet.dis$fitted.values>0.5,datappq$DepSeuil)
```

La validation croisée est toujours nécessaire afin de tenter d'optimiser les choix en présence : nombre de neurones, *decay* et éventuellement le nombre max d'itérations. Néanmoins, la fonction `tune.nnet` pose des problèmes dans le cas de la discrimination. Voici donc, à titre pédagogique, le script d'une fonction de k-validation croisée adaptée aux réseaux de neurones.

1. Ouvrir l'éditeur pour la création de la fonction : `fix(CVnn, editor="emacs"))` avec l'éditeur emacs ou un autre sous Linux (kwrite, kate...) ; L'éditeur par défaut étant vi. Utiliser l'éditeur par défaut sous Windows.
2. 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$DepSeuil[ri == i]
         == "TRUE") != (predict(learn, data[ri == i,
         ]) > 0.5)])
       }
   }
   cat("\n", "Taux de mal classes")
   tmc/(niter * length(unique(ri)) * n)
}
```

3. Sauver le texte, quitter l'éditeur
4. 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 (code TRUE des modalités) mais elle pourrait être facilement généralisée et adaptées à tout type de modélisation.

Tester la fonction ainsi obtenue et l'exécuter pour différentes valeurs de `size` comme (5, 6, 7) et `decay` (0, 1, 2). Il semble plus simple de fixer une valeur un peu grande du nombre de neurones (7) et de ne faire varier que le paramètre de decay entr 0 et 5 avec plusieurs exétcutions pour chaque valeur de ce paramètre. En effet, 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(DepSeuil~.,data=datappq,size=7, decay=0)
...
# exécuter pour différentes valeur du decay
```

Noter la taille et le *decay* optimaux et ré-estimer le modèle pour ces valeurs.

4 Prévision de l'échantillon test

Différentes prévisions sont considérées assorties des erreurs estimées sur l'échantillon test. Prévision quantitative de la concentration, prévision de dépassement à partir de la prévision quantitative et directement la prévision de dépassement à partir de l'arbre de décision.

```
# Calcul des prévisions
```

```
pred.nnetr=predict(nnet.reg,newdata=datestr)
pred.nnetq=predict(nnet.dis,newdata=datestq)
# Erreur quadratique moyenne de prévision
sum((pred.nnetr-datestr[,"O3obs"])^2)/nrow(datestr)
# Matrice de confusion pour la prévision du
# dépassement de seuil (régression)
table(pred.nnetr>150,datestr[,"O3obs"]>150)
# Même chose pour la discrimination
table(pred.nnetq>0.5,datestq[,"DepSeuil"])
```

Noter les taux d'erreur.

5 Comparaison des courbes ROC

```
library(ROCR)
rocnnetr=pred.nnetr/300
prednnetr=prediction(rocnnetr,datestq$DepSeuil)
perfnnetr=performance(prednnetr,"tpr","fpr")
rocnnetq=pred.nnetq
prednnetq=prediction(rocnnetq,datestq$DepSeuil)
perfnnetq=performance(prednnetq,"tpr","fpr")
# tracer les courbes ROC en les superposant
# pour mieux comparer
plot(perflogit,col=1)
plot(perfnnetr,col=2,add=TRUE)
plot(perfnnetq,col=3,add=TRUE)
```

Une méthode de prévision d'occurence de dépassement du pic de pollution est-elle globalement meilleure ?