TP3 - Willot - Collas

Problème de classification

Chargement des données

Nous commençons par charger les données du problème de classification. Ce problème de classification ne possède que 2 classes. 43% des individus appratiennent à la classe 1. 57% appartiennent à la classe 2. Ce jeu de données contient 200 individus, chacun est décrit par 36 variables.

```
full_data <- read.table('../tp3_a18_clas_app.txt')
```

Nombre de variables	37.000
Nombre de donnes	200.000
Donnes classe 1	0.425
Donnes classe 2	0.575

Ensuite nous découpons l'ensemble des données en 2 ensembles: un pour l'entrainement et un pour le test. Nous répartissons les données entre les 2 ensembles de manière aléatoire afin que nos 2 ensembles aient les mêmes proportions des étiquettes que dans l'ensemble initial. Nous fixons le générateur de nombres aléatoires afin de conserver le même ensemble de test au cours des différentes executions de ce notebook.

```
set.seed(123)
ratio_training <- 3/4
train <- sample(1:nrow(full_data), floor(nrow(full_data)*ratio_training))
data_train <- full_data[train,]
data_test <- full_data[-train,]
```

Méthode d'évaluation

Nous évaluons chaque modèle sur l'ensemble d'entrainement à l'aide d'une validation croisée (avec 10 parties). Nous calculons l'erreur moyenne et l'écart type. Nous appliquons la "One-standard-error rule" pour choisir notre modèle final. Enfin, nous estimons l'erreur de notre modèle sur l'ensemble de test. L'erreur de classification est définie par la probabilité d'erreur.

Validation croisée

Nous avons implémenté une fonction qui calcule l'erreur de classification et l'écart type par validation croisée pour l'ensemble des modèles que nous utiliserons par la suite.

Analyses discriminantes (linéaire, quadratique, et classifieur bayésien naïf).

Nous commençons par utiliser les analyses discriminantes linéaire et quadratique, ainsi que le classifieur bayésien naif.

```
set.seed(123)
#validation croisée sur l'ensemble d'entrainement
res <- data.frame("Erreur"=rep(0,3), "Ecart type"=rep(0,3))
row.names(res) <- c("ADL", "ADQ", "Bayésien naif")
res[1,] <- CV_eval('adl', data_train) #analyse discriminantes linéaire
```

```r
res[2,] <- CV_eval('adq', data_train) #analyse discriminantes quadratique
res[3,] <- CV_eval('bayesien_naif', data_train)
```

Nous obtenons les erreurs suivantes:

	Erreur	Ecart.type
ADL	0.2600000	0.1595237
ADQ	0.3666667	0.0980824
Baysien naif	0.1266667	0.1056344

Régression logistique

Nous testons la régression logistique bien que ce modèle soit similaire à l'analyse discriminante linéaire (qui ne diffère que par l'évaluation des poids).

```r
set.seed(123)
#validation croisée sur l'ensemble d'entrainement avec une régression logistique
data_train_glm <- data_train
data_train_glm$y <- data_train_glm$y - 1
res <- data.frame("Erreur"=0, "Ecart type"=0)
row.names(res) <- "Regression logistique"
res[1,] <- suppressWarnings(CV_eval('reg_logistique', data_train_glm))
```

Comme attendu, nous obtenons une erreur très proche de ADL:

	Erreur	Ecart.type
Regression logistique	0.2533333	0.1010425

K plus proches voisins

Nous testons les K plus proches voisins. Pour cela nous devons chercher le meilleur K, c'est à dire le K pour lequel notre erreur de validation croisée est minimale. Nous évaluons les valeurs de K entre 1 et 100.

```r
set.seed(123)
row.names(res) <- "KNN"
res[1,1] <- 1
for (k in 1:100){
  hyperparametres <- list(K=k)
  temp <- CV_eval('knn', data_train, hyperparametres)
  if(temp[1]<res[1]){
    k_min <- k
    res[1,1] <- temp[1]
    res[1,2] <- temp[2]
  }
}
```

	Erreur	Ecart.type
KNN	0.2266667	0.1007949

Nous obtenons l'erreur minimale de 23% pour K=5.

Analyse discriminante régularisée

Dans cette partie nous utilisons une analyse discriminante régularisée implémentée dans le package **klaR**. Cette implémentation permet d'obtenir des modèles régularisés de l'analyse disciminante linéaire, de l'analyse discriminante quadratique et du classifieur bayésien naïf. L'impléméntation du package **klaR** cherche les hyperparamètres γ et λ à l'aide d'une validation croisée sur l'ensemble d'entrainement. Les hyperparamètres optimaux trouvés sont $\gamma = 0.86$ et $\lambda = 0.04$. D'après la documentation du package klaR (page 58), cela correspond à des matrices de variances-covariances dont les termes de chaque diagonale sont quasiement égaux et les éléments en dehors de la diagonale sont proches de 0. Ce classifieur correspond à un classifieur bayésien naïf régularisé (par les termes des diagonales de matrices de variances-covariances égaux). L'erreur trouvée est proche de celle trouvée du classifieur bayésien naïf.

```
set.seed(123)
model <- rda(formula=as.factor(y)~., data=data_train, crossval=TRUE, fold=10)
model$regularization
```

```
##     gamma    lambda
## 0.8580221 0.0405724
```

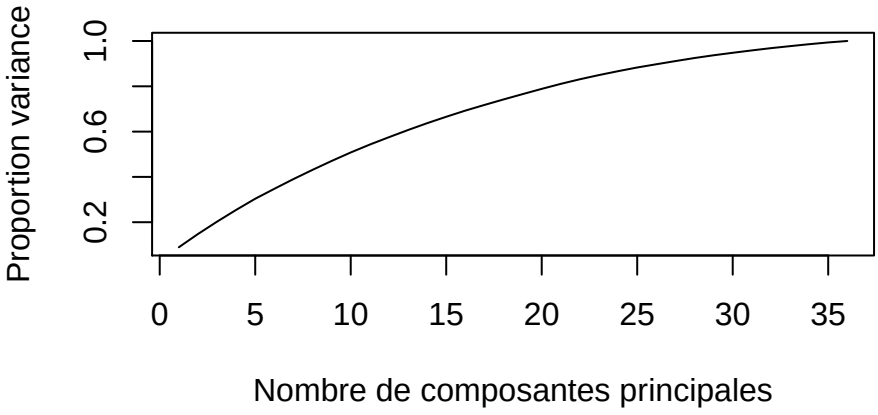
```
model$error.rate[2]
```

```
##  crossval
## 0.1125913
```

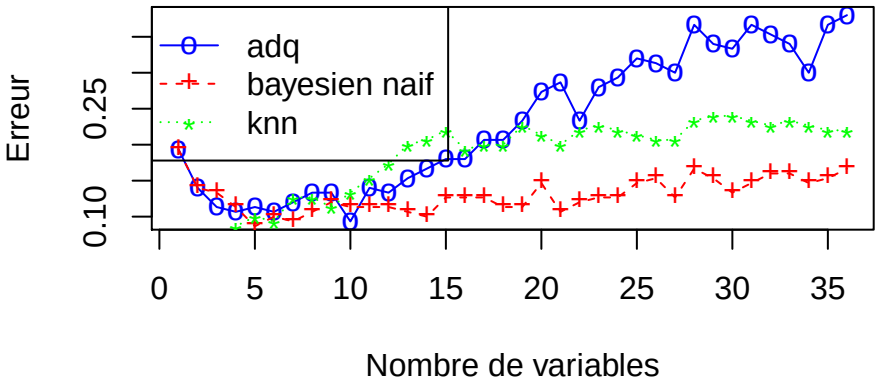
Selection de variables: Analyse en composante principale

Une deuxième méthode pour régulariser nos modèles est de les entrainer avec un sous ensemble de variables. En réduisant la dimensionnalité en entrée, nous réduisons fortement le nombre de paramètres (réduction quadratique pour ADL et ADQ) tout en essayant de conserver un maximum de l'information initiale. Pour cela nous utilisons une analyse en composantes principales.

```
set.seed(123)
pca <- princomp(formula=~.-y, data=data_train)
Z <- pca$scores
lambda <- pca$sdev^2
plot(cumsum(lambda)/sum(lambda),type="l",xlab="Nombre de composantes principales", ylab="Proportion varian
```



Ensuite, evaluons l'erreur par validation croisée en utilisant des sous ensembles des composantes principales. Nous choisissons d'abord la première composante principale puis ajoutons la deuxième, etc... Nous traçons l'erreur en fonction du nombre de composantes principales utilisées:



Nous observons que les 5 premières composantes principales ne contiennent que 20% de la variance totale des données. Cependant, conserver seulement ces 5 premières composantes permet de réduire fortement l'erreur de l'analyse discriminante quadratique (de 36% à 10%). Pour les K plus proches voisins, nous cherchons pour chaque sous ensemble de composantes principales le K minimisant l'erreur de validation croisée. L'erreur passe de 23% à moins de 10% en ne conservant que les 4 premières composantes principales. Le classifieur bayésien naif profite moins de la baisse du nombre de variables du fait de son faible nombre de paramètres même avec beaucoup de variables.

Sélection de variables: Sélection rétrograde pas à pas

Enfin, une autre méthode pour régulariser nos modèles est de selectionner parmis les modèles formés de sous-ensembles de prédicteurs, un modèle plus simple que le modèle complet, et dont l'indicateur de performance possède une bonne valeur. L'indicateur que nous avont choisi ici est le BIC (Bayesian Information Criterion ou Critère d'Information Bayésien) et le but est de trouver le sous-ensemble de prédicteurs qui réalise son minimum. Le calcul de ces sous-ensembles se fait ici avec la méthode rétrograde, car énumérer tous les sous-ensembles de 36 prédicteurs est trop coûteux (2^36). Celle-ci n'évalue que 1+36*37/2 = 667 modèles en retirant à chaque étape le prédicteur le moins significatif. Pour obtenir des résultats plus stables, ce calcul de sous-ensemble performant de prédicteur est effectué au sein d'une validation croisée, et ensuite le modèle désiré (par exemple l'Analyse Discriminante Linéaire) est appris avec le meilleur sous-ensemble de prédicteur pour cette étape de la validation croisée. La fonction CV_eval2 réalise cette méthodologie pour un ensemble de models passé en argument pour un ensemble d'apprentissage et retourne l'erreur moyenne et l'écart type.

```
set.seed(123)
#validation croisée sur l'ensemble d'entrainement
res <- CV_eval2(c('adl','adq','bayesien_naif'), data_train)
colnames(res) <- c("Erreur", "Ecart type")
row.names(res) <- c("ADL", "ADQ", "Bayesien naif")
```

Nous obtenons les erreurs suivantes:

	Erreur	Ecart type
ADL	0.1466667	0.1268718
ADQ	0.1666667	0.1086718
Bayesien naif	0.1266667	0.0995439

Modèle retenu

Nous choisissons le modèle Bayesien naif avec le meilleur sous-ensemble de prédicteur comme définit dans la partie sélection rétrograde pas à pas. En effet elle possède la plus petite erreur moyenne que l'on ait trouvé, à égalité avec le modèle Bayesien naif avec tous les prédicteurs, mais cette dernière possède un écart-type plus grand, ce qui signifie que le modèle est plus variable que l'autre. Il est donc préférable de choisir le modèle avec la plus petite variance, le modèle avec le sous-ensemble de prédicteurs. Nous évaluons l'erreur sur l'ensemble de test (réservé à cet effet au départ): cette erreur nous donne une estimation non biaisée.

```
sub <- regsubsets(y~.,data=data_train,method='backward');
a <- summary(sub);
coefs <- coef(sub, which.min(a$bic))
colnames <- names(coefs)
colnames <- colnames[colnames != "(Intercept)"]
formula <- as.formula(paste("y ~ ", paste(colnames, collapse = " + ")))
data_train[,37] <- as.factor(data_train[,37])
data_test[,37] <- as.factor(data_test[,37])
class.mod <- naiveBayes(formula, data_train)
pred <- predict(class.mod, data_test[,1:36])
error <- sum(data_test$y!=pred) / length(pred)
error
```

```
## [1] 0.22
```

Problème de régression

Chargement des données

Nous commençons par charger les données du problème de régression. Ce problème contient 200 individus (comme le problème de classification). Chaque individu est decrit par 50 variables quantitatives. La variable nommée "y" est la variable à prédire.

```
rm(list=ls())
full_data <- read.table('../tp3_a18_reg_app.txt')
nrow(full_data)
```

```
## [1] 200
```

Ensuite nous découpons l'ensemble des données en 2 ensembles: un pour l'entrainement et un pour le test. Nous fixons le générateur de nombres aléatoires afin de conserver le même ensemble de test au cours des différentes executions de ce notebook.

```
set.seed(123)
ratio_training <- 3/4
train <- sample(1:nrow(full_data), floor(nrow(full_data)*ratio_training))
data_train <- full_data[train,]
data_test <- full_data[-train,]
```

Méthode d'évaluation

Pour ce problème l'erreur de régression est définie par l'espérance de l'erreur quadratique. Nous estimons cette erreur et son écart type pour chaque modèle sur l'ensemble d'entrainement à l'aide d'une validation croisée (avec 10 parties). Ensuite, nous appliquons la "One-standard-error rule" pour choisir notre modèle final. Enfin, nous estimons l'erreur de notre modèle sur l'ensemble de test.

Régression linéaire

```
reg <- lm(formula = y~., data=data_train)
res <- resid(reg)
```

Analyse des résidus

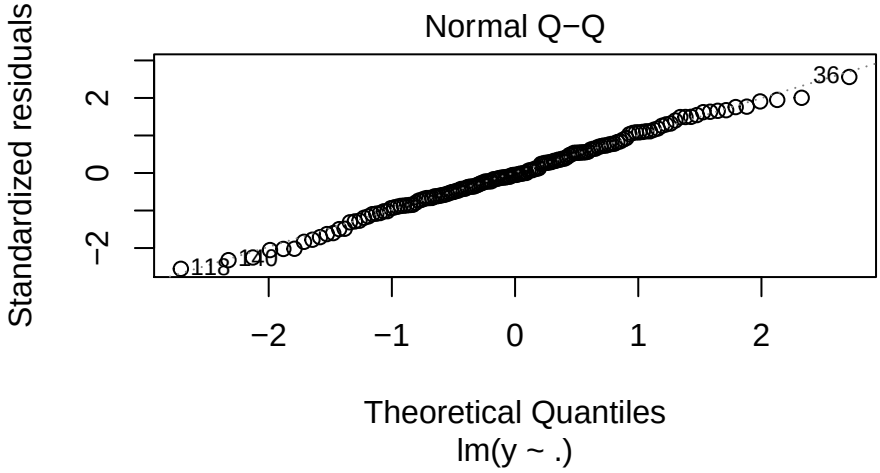
Nous commençons par vérifier les hypothèses sur les résidus de la régression linéaire sur l'ensemble d'entrainement. Tout d'abord nous testons la normalité des résidus à l'aide d'un test de Shapiro sur les résidus.

```
shapiro.test(res)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  res
## W = 0.9943, p-value = 0.824
```

La p-value obtenue est très élevée, 0.824, les résidus sont donc probablement gaussiens. Nous pouvons confirmer ce résultat à l'aide d'un diagramme Q-Q des quantiles des résidus standardisés par rapport à ceux d'une loi normale.

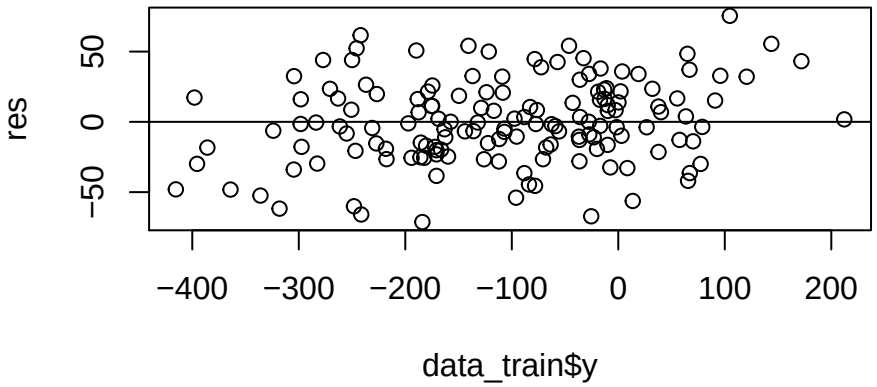
```
plot(reg, which=2)
```



La quasi totalité des points sont situés sur la première diagonale, ce qui confirme que les résidus sont gaussiens.

De plus, nous vérifions l'homoscédasticité des résidus. Pour cela nous représentons graphiquement les résidus en fonction des valeurs prédites.

```
res <- resid(reg)
plot(data_train$y, res)
abline(0,0)
```



Nous observons que la variance est la même quelque soit y. L'hypothèse d'homoscédasticité est donc vérifiée.

Analyse de la stabilité

Ensuite, nous cherchons si certaines observations ont une forte influence sur les coefficients de la régression linéaire. Deux types de points ont cet effet: ceux à effet de levier et ceux à fort résidu. Pour cela nous calculons la distance de Cook qui mesure l'effet de la suppression d'un point. Les points qui ont une distance de Cook supérieure à 1 sont considérés comme atypique.

```
sum(cooks.distance(reg)>1)
```

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

Les données ont toutes des distances de Cook inférieures à 1, il n'y a donc a priori pas de point atypique.

Calcul de l'erreur par validation croisée

Nous calculons l'erreur par validation croisée

```
source('CV.R')
```

```
set.seed(123)
#validation croisée sur l'ensemble d'entrainement
res <- data.frame("Erreur"=rep(0,1), "Ecart type"=rep(0,1))
row.names(res) <- c("Regression linéaire")
res[1,] <- CV_eval('reg_lineaire', data_train)
```

Nous obtenons l'erreur suivante:

	Erreur	Ecart.type
Regression linaire	2351.408	922.5063

Régression linéaire régularisées (ridge et lasso)

Etant donné le grand nombre de variables (50) pour le faible nombre de d'individus (200), nous cherchons à pénaliser les variables (faire tendre leurs coefficients vers 0) qui ont peu d'effet sur la valeur à prédire. Malgré que la méthode des moindres carrés est optimale sur l'ensemble d'entrainement pour l'erreur quadratique moyenne, nous espérons une diminution de l'erreur de généralisation (en abaissant la variance de notre modèle) en ajoutant des termes de pénalisation. Pour cela nous utilisons les modèles ridge et lasso qui ajoutent des termes de penalisation dans le problème de minimisation des moindres carrés.

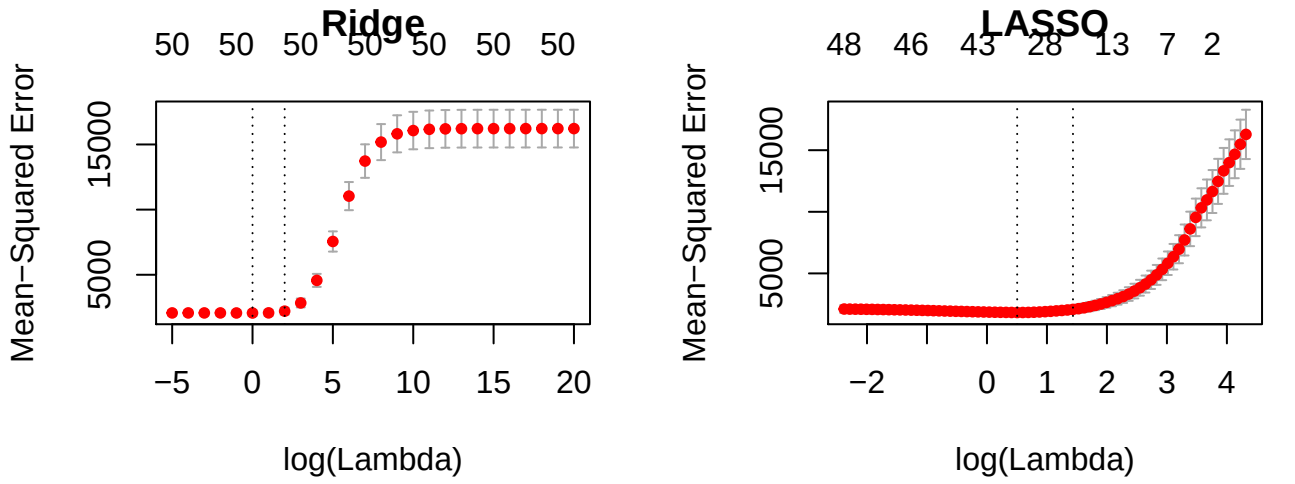
```
xapp <- model.matrix(y~., data_train)[,2:51]
yapp <- data_train$y
```

Ridge et LASSO

Nous commençons par chercher l'hyperparamètre lambda du modèle ridge. Nous le cherchons à l'aide d'une validation croisée sur l'ensemble d'entrainement. Le modèle ridge ainsi que la validation croisée sont implémentés dans le package **glmnet**. Ensuite nous cherchons l'hyperparamètre lambda du modèle lasso. Nous procédons de la même manière que pour ridge. Du fait du terme de pénalisation, ridge et lasso nécessitent de centrer et réduire les variables (paramètre standardize dans glmnet).

```
res <- data.frame("Erreur"=rep(0,2), "Ecart type"=rep(0,2))
row.names(res) <- c("Ridge", "LASSO")
cv.out <- cv.glmnet(xapp, yapp, alpha=0, lambda=exp(-5:20), type.measure='mse', nfolds=10, standardize=TRU
hyperparametres <- list(alpha=0, lambda = cv.out$lambda.min)
res[1,] <- CV_eval('ridge_lasso', data_train, hyperparametres)
par(mfrow=c(1,2))
```

```
plot(cv.out, main="Ridge")
cv.out <- cv.glmnet(xapp, yapp, alpha=1, lambda=NULL, type.measure='mse', nfolds=10, standardize=TRUE)
hyperparametres <- list(alpha=1, lambda=cv.out$lambda.min)
res[2,] <- CV_eval('ridge_lasso', data_train, hyperparametres)
plot(cv.out, main='LASSO')
```



Nous obtenons les erreurs minimales suivantes:

	Erreur	Ecart.type
Ridge	2116.148	1085.6905
LASSO	1790.634	635.4609

L'erreur en utilisant la pénalisation Ridge est très proche de l'erreur de la régression linéaire simple. Cependant, la pénalisation LASSO donne un résultat bien meilleur que le résultat simple des moindres carrés. L'erreur moyenne est inférieure tout comme l'écart type. Par ailleurs 17 coefficients sur 50 sont nuls en utilisant la pénalisation LASSO!

```
glmnet.fit <- glmnet(xapp, yapp, lambda=cv.out$lambda.min, alpha=1, standardize=TRUE)
sum(glmnet.fit$beta==0)
```

```
## [1] 17
```

Modèle retenu

Nous choisissons la méthode LASSO qui donne la plus faible erreur. Nous récupérons le meilleur lambda déterminé par validation croisée puis apprenons sur les données d'entrainement. Nous évaluons l'erreur sur l'ensemble de test (réservé à cet effet au départ): cette erreur nous donne une estimation non biaisée.

```
lambda <- cv.out$lambda.min
reg <- glmnet(xapp, yapp, lambda=lambda, alpha=1, standardize=TRUE)
xtst <- model.matrix(y~., data_test)[,2:51]
pred <- predict(reg, newx=xtst)
erreur <- sum((pred-data_test$y)^2)/length(pred)
erreur
```

```
## [1] 1328.118
```

```
#rm(list=ls())
#load(file="env_classifieur.RData")
#load(file="env_regresseur.RData")
#save(class.mod, reg, file="env.RData")
```