

Projet STA203 : étude d'un jeu de données musical

Hamou Claude, Ray Loïc

08/05/2020

Dans ce projet nous allons étudier un jeu de données musical. Ce jeu de données contient 191 variable quantitatives et 1 variables qualitative qui représente le genre de musique.

Nous allons implémenter 3 méthodes permettant de prédire la variable de genre à partir des autres variables. Il s'agira des méthodes de régression logistique, des k plus proches voisins, et de régression ridge. Commençons donc par importer le jeu de données.

```
data=read.table("Music.txt", header = TRUE, sep=";")
#head(data)
#summary(data) #trop long
#str(data)
ncol(data) #nombre de variable
```

```
## [1] 192
```

```
nrow(data) #nombre d'observation
```

```
## [1] 6447
```

Partie 1: Régression logistique

Question 1

Il y a trop de variables pour pouvoir extraire des informations en regardant les données brutes. Mais on peut toutefois faire des analyses univariée et bivariée sur quelques variables.

Regardons tout d'abord le nombre d'individus de chaque classe.

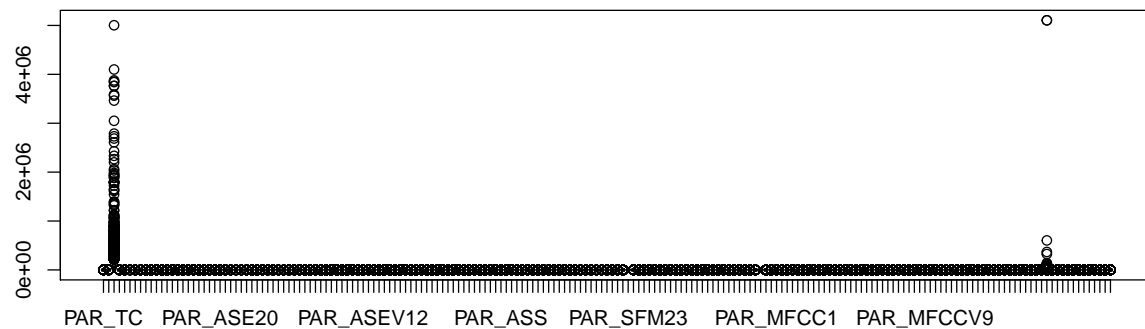
```
summary(data[192]) #nombre de d'individu de chaque groupes
```

```
##          GENRE
## Classical:3444
## Jazz      :3003
```

On peut déjà remarquer que le jeu de données est relativement équilibré, ce qui nous permet de faire une étude qui ne soit pas trop biaisée.

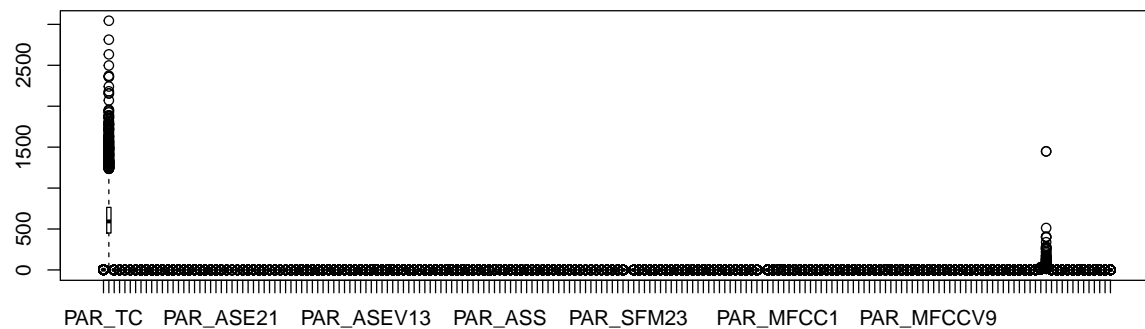
Faisons une analyse univariée, sans considérer la variable qualitative 192.

```
boxplot(data[-192])
```



Cela ne se voit pas bien sur le graphique mais les variables 3 **PAR_SC_V**, et 179 **PAR_PEAK_RMS10FR_VAR** prennent des valeurs bien plus élevées que les autres variables. On peut notamment remarquer que ces deux variables ont des variances élevées (du fait de la répartition des points au dessus de leur boîte) Affichons un nouveau boxplot sans prendre en compte ces variables.

```
boxplot(data[-c(3,179,192)])
```



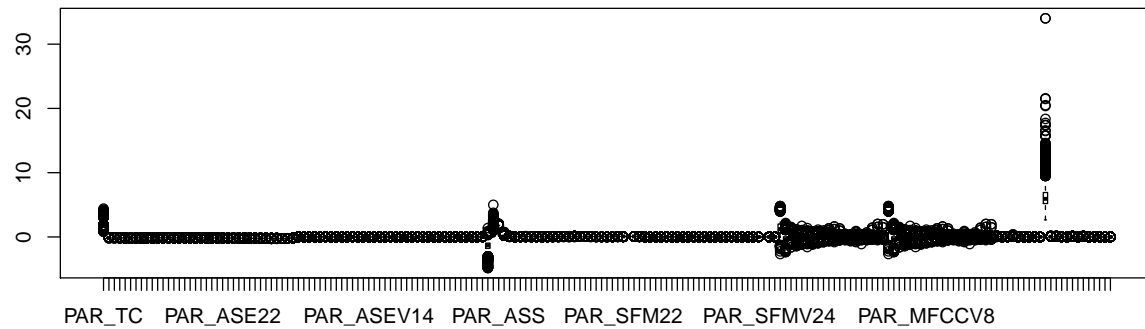
```
#boxplot((data[c(-3,-179,-192)])[2:10])
```

Nous voyons ici que les variables et 2 **PAR_SC** et 178 **PAR_PEAK_RMS10FR_MEAN** prennent elles aussi des valeurs bien supérieures aux autres variables, avec là encore une grande variance entre les valeurs. Par ailleurs cela n'est pas étonnant car ces variables et les variables précédentes sont reliées. En effet les variables que nous avons ici représentent des moyennes, et les variables précédentes représentaient la variance associée.

Faisons un dernier boxplot sans ces variables.

**ON POURRAIT FAIRE UN TRUC MIEUX EN ECRIVANT UNE FONCTION QUI REN-
VOIE LE NOMS DE TOUTES LES VARIABLES QUI PRENNENT DES VALEURS SU-
PERIEURES A UNE CONSTANT GENRE 1**

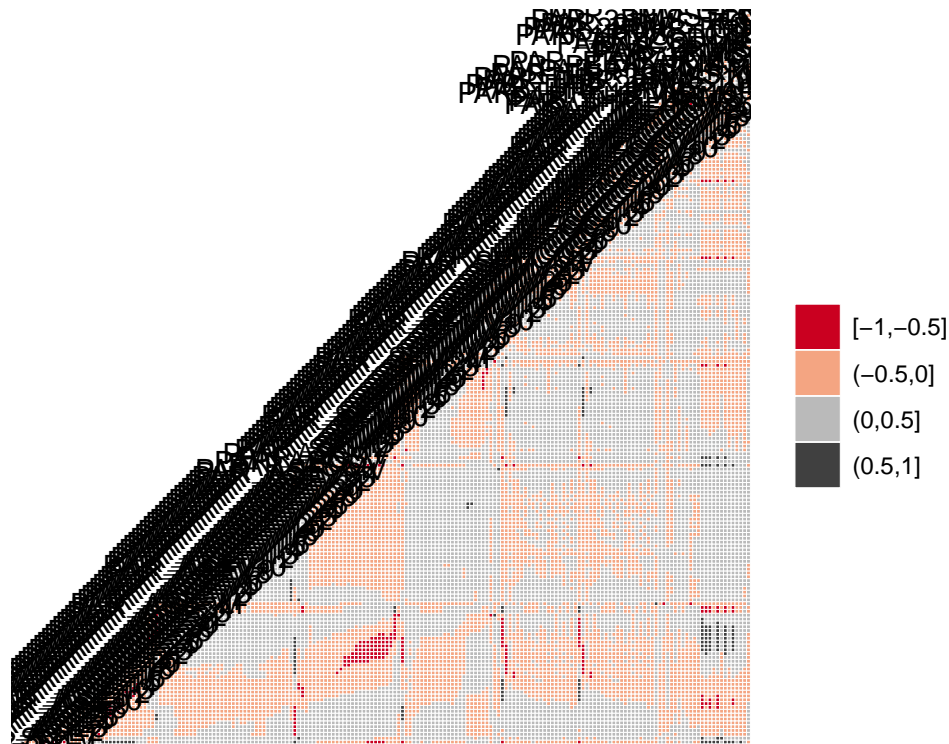
```
boxplot(data[-c(2,3,178,179,192)])
```



```
#boxplot((data[-c(2,3,178,179,192)])[175:180])
```

Intéressons nous maintenant aux corrélations entre les variables, en faisant une étude bivariable du jeu de données. Pour cela calculons et affichons la matrice de corrélations.

```
matrice_data=data.matrix(data)
correlation_data=cor(matrice_data)
ggcorr(matrice_data,nbreaks = 4, palette = "RdGy")
```



```
#corrplot(correlation_data, tl.pos='n')
```

Comme cela était attendu, le graphique est quasiment illisible. Mais on parvient tout de même à discerner des zones de forte covariance. Implementons une fonction qui affiche les variables dont la covariance est comprise entre 2 bornes, afin de retirer des informations plus pertinentes de la matrice de corrélation.

```

print_corr_borne= function(mat_cor,seuil_min,seuil_max){
  l=nrow(mat_cor)
  found=FALSE
  for(i in 1:l){
    for(j in 1:i){
      if(mat_cor[i,j]>seuil_min && mat_cor[i,j]<seuil_max){
        #Affiche le nom des variables correspondantes
        found=TRUE
        print(names(data)[c(i,j)])
      }
    }
  }
  if(!found){
    print("Il n'y a aucune covariance n'est comprise entre ces bornes")
  }
}

```

Nous pouvons alors afficher les variables très corrélées, dont la covariance se trouve dans]0.99;1[

```
print_corr_borne(correlation_data,0.99,1)
```

```

## [1] "PAR_ASE34" "PAR_ASE33"
## [1] "PAR_ASEV34" "PAR_ASEV33"
## [1] "PAR_MFCCV1" "PAR_MFCC1"
## [1] "PAR_MFCCV5" "PAR_MFCC5"
## [1] "PAR_MFCCV12" "PAR_MFCC12"
## [1] "PAR_MFCCV13" "PAR_MFCC13"
## [1] "PAR_MFCCV14" "PAR_MFCC14"
## [1] "PAR_MFCCV19" "PAR_MFCC19"
## [1] "PAR_ZCD_10FR_MEAN" "PAR_ZCD"

```

Ainsi que les variables très anti-corrélées, dont la covariance se trouve dans] -1;0.99[

```
print_corr_borne(correlation_data,-1,-0.99)
```

```
## [1] "Il n'y a aucune covariance n'est comprise entre ces bornes"
```

On remarque donc que les variables très corrélées sont de type *MFCCV* et *MFCC* ainsi que des variables *ASE*.

Considérons les variables 128 à 147 et 148 à 167. En regardant le jeu de données et son descriptif, il semblerait que ces deux groupes de variables soient égaux. Pour le confirmer on écrit un script qui renvoie le nombre de différence entre ces 2 groupes.

```

#Egalite 128:147 et 148:167
dif=0
for(i in 128:147){
  dif=sum(data[i]!=data[i+20])
}
dif

```

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

Comme indiqué dans le descriptif du dataset, les colonnes 128 à 147 et 148 à 167 ont les mêmes valeurs. On ne considérera donc pas les colonnes 148 à 167 dans la suite.

Les données **PAR_ASE_M**, **PAR_ASE_MV**, **PAR_SFM_M** et **PAR_SFM_MV** représentent les moyennes des variables 4 à 37, 39 à 72, 78 à 101, et 103 à 126. Pour réduire le nombre de variable il peut être préférable dans un premier temps de pas considéré les colonnes 4 à 37, 39 à 72, 78 à 101, et 103 à 126

comme les variables **PAR_ASE_M**, **PAR_ASE_MV**, **PAR_SFM_M** et **PAR_SFM_MV** en sont des agrégats.

On réalise les opérations de nettoyage précédemment expliquées et on note X le nouveau data frame de données que nous allons utiliser dans la suite. Et Y le vecteur contenant la variable qualitative **GENRE** en binaire, avec $Classical = 0$ et $Jazz = 1$.

```
#Colonnes que nous n'utiliserons pas dans la suite
del=c(148:167,
      4:37,
      39:72,
      78:101,
      103:126)

#
X=data[,-c(del,192)]
#log des variables PAR_SC_V et PAR_ASC_V
X["PAR_SC_V"]=log(data["PAR_SC_V"])
X["PAR_ASC_V"]=log(data["PAR_ASC_V"])

#
GENRE=data[,192]
Y=1*(GENRE=="Jazz")
```

Nous cherchons à déterminer un modèle logistique permettant de d'estimer les valeurs de la variable **Y**. Cette variable prend deux valeurs **0** (s'il s'agit du genre classique) et **1** (s'il s'agit du genre jazz). C'est donc la variable binaire **Y** que nous cherchons à expliquer. Les valeurs des covariables x_i , qui représentent les autres paramètres, sont différentes pour chaque segment de morceau musical, on modélise l'expérience comme la réalisation de $n = 6447$ variables aléatoires indépendantes de Bernoulli Z_i , de paramètre $\pi(x_i) = \mathbb{P}(Z_i = 1|x_i)$. Dans notre cas, on a donc $Z_i \sim \mathcal{B}(1, \pi(x_i))$. Ici, une régression linéaire ne serait pas adaptée (à cause de la contrainte sur π). En revanche, on peut définir la fonction de lien *logit* telle que

$$\text{logit}(\pi(x_i)) = \log\left(\frac{\pi(x_i)}{1 - \pi(x_i)}\right)$$

. Notre modèle est alors tel que

$$\text{logit}(\pi(x_i)) = x_i \theta$$

, régresseur linéaire des covariables.

Question 2

On utilise le code proposé pour générer des data-frames de *training* permettant de fitter le modèle et des data-frames de *test*.

```
set.seed(103)
n=nrow(data)
train=sample(c(TRUE,FALSE),n,rep=TRUE,prob=c(2/3,1/3))
X_training=X[train,]
X_test=X[!train,]

GENRE_training=GENRE[train]
GENRE_test=GENRE[!train]

Y_training=Y[train]
Y_test=Y[!train]
```

Question 3

Mod0

```
indices=c("PAR_TC","PAR_SC", "PAR_SC_V", "PAR_ASE_M", "PAR_ASE_MV", "PAR_SFM_M", "PAR_SFM_MV")
Mod0=glm(Y_training~.,family=binomial,data=X_training[indices])
summary(Mod0)

##
## Call:
## glm(formula = Y_training ~ ., family = binomial, data = X_training[indices])
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -2.7431  -0.9390  -0.5053   0.9487   2.6548
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  4.102e+01  4.563e+00   8.989  <2e-16 ***
## PAR_TC       1.136e-01  1.035e-01   1.098  0.2723
## PAR_SC       4.523e-04  2.106e-04   2.148  0.0317 *
## PAR_SC_V     8.266e-02  5.034e-02   1.642  0.1006
## PAR_ASE_M    3.211e+02  2.875e+01  11.172  <2e-16 ***
## PAR_ASE_MV   7.823e+03  5.404e+02  14.476  <2e-16 ***
## PAR_SFM_M    1.202e+02  5.721e+00  21.008  <2e-16 ***
## PAR_SFM_MV  -3.762e+02  3.966e+02  -0.949  0.3428
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##      Null deviance: 5914.0  on 4277  degrees of freedom
## Residual deviance: 4917.7  on 4270  degrees of freedom
## AIC: 4933.7
##
## Number of Fisher Scoring iterations: 4
#par(mfrow=c(2,2))
#plot(Mod0)
Mod0.predict_test=predict(Mod0, newdata=X_test[indices], type="response")
```

ModT

```
ModT=glm(Y_training~.,family=binomial,data=X_training)

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
summary(ModT)

##
## Call:
## glm(formula = Y_training ~ ., family = binomial, data = X_training)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -3.8685  -0.6061  -0.1862   0.5031   3.7982
```

```

##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)    2.055e+01  1.515e+01  1.356 0.175021
## PAR_TC         2.526e-02  1.457e-01  0.173 0.862348
## PAR_SC         8.970e-03  1.622e-03  5.530 3.20e-08 ***
## PAR_SC_V       4.327e-01  1.240e-01  3.491 0.000481 ***
## PAR_ASE_M      3.287e+02  8.815e+01  3.729 0.000193 ***
## PAR_ASE_MV     3.481e+03  8.209e+02  4.240 2.24e-05 ***
## PAR_ASC       -6.657e+00  4.294e-01 -15.503 < 2e-16 ***
## PAR_ASC_V     -7.947e-01  1.471e-01 -5.401 6.61e-08 ***
## PAR_ASS       -3.464e+00  5.902e-01 -5.869 4.38e-09 ***
## PAR_ASS_V      8.605e+00  1.317e+00  6.536 6.30e-11 ***
## PAR_SFM_M      6.371e+01  9.455e+00  6.739 1.60e-11 ***
## PAR_SFM_MV     2.071e+03  5.763e+02  3.593 0.000327 ***
## PAR_MFCC1      5.178e-01  1.213e-01  4.269 1.96e-05 ***
## PAR_MFCC2     -6.355e-01  1.529e-01 -4.156 3.24e-05 ***
## PAR_MFCC3     -1.734e+00  1.958e-01 -8.856 < 2e-16 ***
## PAR_MFCC4      5.294e-01  2.183e-01  2.425 0.015302 *
## PAR_MFCC5     -1.128e+00  2.390e-01 -4.722 2.34e-06 ***
## PAR_MFCC6     -4.167e-01  2.323e-01 -1.794 0.072843 .
## PAR_MFCC7     -1.008e+00  2.595e-01 -3.883 0.000103 ***
## PAR_MFCC8      2.092e+00  3.092e-01  6.765 1.33e-11 ***
## PAR_MFCC9     -4.639e+00  3.476e-01 -13.345 < 2e-16 ***
## PAR_MFCC10     3.022e+00  4.075e-01  7.416 1.21e-13 ***
## PAR_MFCC11    -3.011e+00  4.876e-01 -6.175 6.60e-10 ***
## PAR_MFCC12     2.994e+00  5.537e-01  5.408 6.38e-08 ***
## PAR_MFCC13    -4.544e+00  6.494e-01 -6.998 2.60e-12 ***
## PAR_MFCC14    -5.789e-01  7.003e-01 -0.827 0.408486
## PAR_MFCC15    -3.805e+00  7.890e-01 -4.822 1.42e-06 ***
## PAR_MFCC16    -1.004e+00  9.154e-01 -1.097 0.272645
## PAR_MFCC17    -1.094e+00  1.013e+00 -1.080 0.280343
## PAR_MFCC18     6.916e-01  1.204e+00  0.574 0.565719
## PAR_MFCC19    -5.525e-01  1.301e+00 -0.425 0.670930
## PAR_MFCC20     1.164e+00  1.184e+00  0.983 0.325734
## PAR_THR_1RMS_TOT -1.518e+00  9.632e+00 -0.158 0.874786
## PAR_THR_2RMS_TOT -9.202e+01  2.164e+01 -4.253 2.11e-05 ***
## PAR_THR_3RMS_TOT -1.267e+02  5.253e+01 -2.412 0.015884 *
## PAR_THR_1RMS_10FR_MEAN 6.203e+01  9.215e+00  6.731 1.68e-11 ***
## PAR_THR_1RMS_10FR_VAR -1.756e+02  1.008e+02 -1.742 0.081444 .
## PAR_THR_2RMS_10FR_MEAN -1.012e+02  2.094e+01 -4.833 1.34e-06 ***
## PAR_THR_2RMS_10FR_VAR  4.255e+03  1.383e+03  3.076 0.002098 **
## PAR_THR_3RMS_10FR_MEAN 7.201e+02  6.639e+01 10.847 < 2e-16 ***
## PAR_THR_3RMS_10FR_VAR -2.162e+04  7.725e+03 -2.799 0.005127 **
## PAR_PEAK_RMS_TOT -3.648e-03  5.777e-02 -0.063 0.949653
## PAR_PEAK_RMS10FR_MEAN -2.539e-02  1.072e-02 -2.369 0.017826 *
## PAR_PEAK_RMS10FR_VAR  4.831e-05  2.563e-05  1.884 0.059502 .
## PAR_ZCD       -6.201e+01  4.339e+01 -1.429 0.153005
## PAR_1RMS_TCD   3.488e+01  4.451e+01  0.783 0.433355
## PAR_2RMS_TCD   1.126e+02  1.179e+02  0.955 0.339472
## PAR_3RMS_TCD   3.203e+02  1.837e+02  1.743 0.081294 .
## PAR_ZCD_10FR_MEAN  6.242e+01  3.623e+01  1.723 0.084946 .
## PAR_ZCD_10FR_VAR -5.405e+02  2.342e+02 -2.307 0.021029 *
## PAR_1RMS_TCD_10FR_MEAN -1.853e+01  4.147e+01 -0.447 0.654971

```

```
## PAR_1RMS_TCD_10FR_VAR -4.451e+02 2.224e+02 -2.002 0.045322 *
## PAR_2RMS_TCD_10FR_MEAN 1.764e+02 1.043e+02 1.691 0.090830 .
## PAR_2RMS_TCD_10FR_VAR -4.792e+03 9.872e+02 -4.854 1.21e-06 ***
## PAR_3RMS_TCD_10FR_MEAN -2.755e+02 1.676e+02 -1.643 0.100334
## PAR_3RMS_TCD_10FR_VAR 1.715e+04 3.266e+03 5.250 1.52e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 5914.0 on 4277 degrees of freedom
## Residual deviance: 3284.1 on 4222 degrees of freedom
## AIC: 3396.1
##
## Number of Fisher Scoring iterations: 9
#par(mfrow=c(2,2))
#plot(ModT)
ModT.predict_test=predict(ModT, newdata=X_test, type="response")
```

Mod1

```
#Pr(>|t|) de ModT
ModT.p_val=coef(summary(ModT))[,4]
#On cherche les indices de p-value < 5%
indice_sign_5=names(which(ModT.p_val<0.05))
#Prediction de Mod1
Mod1=glm(Y_training~.,family=binomial,data=X_training[indice_sign_5])
summary(Mod1)

##
## Call:
## glm(formula = Y_training ~ ., family = binomial, data = X_training[indice_sign_5])
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -3.5826  -0.6244  -0.2359   0.5251   3.1858
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)    3.481e+01  1.205e+01  2.890 0.003857 **
## PAR_SC         1.289e-02  8.605e-04 14.975 < 2e-16 ***
## PAR_SC_V       3.948e-01  1.084e-01  3.643 0.000270 ***
## PAR_ASE_M      4.062e+02  7.052e+01  5.760 8.42e-09 ***
## PAR_ASE_MV     4.365e+03  7.800e+02  5.596 2.19e-08 ***
## PAR_ASC       -6.160e+00  3.901e-01 -15.792 < 2e-16 ***
## PAR_ASC_V     -8.288e-01  1.340e-01 -6.187 6.15e-10 ***
## PAR_ASS       -2.155e+00  4.489e-01 -4.800 1.59e-06 ***
## PAR_ASS_V      6.319e+00  1.169e+00  5.404 6.52e-08 ***
## PAR_SFM_M      6.488e+01  8.734e+00  7.429 1.10e-13 ***
## PAR_SFM_MV     1.994e+03  5.520e+02  3.613 0.000303 ***
## PAR_MFCC1      3.950e-01  1.095e-01  3.608 0.000309 ***
## PAR_MFCC2     -4.136e-01  1.345e-01 -3.075 0.002102 **
## PAR_MFCC3     -1.689e+00  1.831e-01 -9.221 < 2e-16 ***
```



```
## PAR_MFCC4          6.126e-01  2.062e-01  2.971 0.002972 **
## PAR_MFCC5          -1.317e+00  2.285e-01 -5.763 8.25e-09 ***
## PAR_MFCC7          -1.446e+00  2.489e-01 -5.810 6.25e-09 ***
## PAR_MFCC8           2.510e+00  2.974e-01  8.440 < 2e-16 ***
## PAR_MFCC9          -4.689e+00  3.354e-01 -13.983 < 2e-16 ***
## PAR_MFCC10         2.953e+00  3.808e-01  7.755 8.87e-15 ***
## PAR_MFCC11         -2.824e+00  4.556e-01 -6.199 5.68e-10 ***
## PAR_MFCC12         3.078e+00  5.171e-01  5.954 2.62e-09 ***
## PAR_MFCC13         -4.483e+00  6.113e-01 -7.334 2.24e-13 ***
## PAR_MFCC15         -3.223e+00  6.055e-01 -5.323 1.02e-07 ***
## PAR_THR_2RMS_TOT   -5.354e+01  1.648e+01 -3.249 0.001159 **
## PAR_THR_3RMS_TOT   -2.390e+02  2.670e+01 -8.950 < 2e-16 ***
## PAR_THR_1RMS_10FR_MEAN 5.020e+01  6.044e+00  8.306 < 2e-16 ***
## PAR_THR_2RMS_10FR_MEAN -9.507e+01  1.833e+01 -5.186 2.15e-07 ***
## PAR_THR_2RMS_10FR_VAR  4.544e+03  1.274e+03  3.566 0.000362 ***
## PAR_THR_3RMS_10FR_MEAN 7.059e+02  5.651e+01 12.493 < 2e-16 ***
## PAR_THR_3RMS_10FR_VAR -2.242e+04  7.209e+03 -3.111 0.001868 **
## PAR_PEAK_RMS10FR_MEAN 2.061e-03  3.184e-03  0.647 0.517553
## PAR_ZCD_10FR_VAR    -1.161e+03  1.715e+02 -6.768 1.31e-11 ***
## PAR_1RMS_TCD_10FR_VAR  5.905e+01  2.140e+02  0.276 0.782662
## PAR_2RMS_TCD_10FR_VAR -2.306e+03  9.078e+02 -2.540 0.011092 *
## PAR_3RMS_TCD_10FR_VAR  1.249e+04  2.640e+03  4.729 2.26e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 5914.0 on 4277 degrees of freedom
## Residual deviance: 3409.6 on 4242 degrees of freedom
## AIC: 3481.6
##
## Number of Fisher Scoring iterations: 6
```

```
#par(mfrow=c(2,2))
#plot(Mod1)
Mod1.predict_test=predict(Mod1, newdata=X_test[indice_sign_5], type="response")
```

Mod2

```
#Pr(>|t|) de ModT
ModT.p_val=coef(summary(ModT))[,4]
#On cherche les indices de p-value < 20%
indice_sign_20=names(which(ModT.p_val<0.2))[c(-1)] #car 1er element est l'intercept
#Prediction de Mod1
Mod2=glm(Y_training~.,family=binomial,data=X_training[indice_sign_20])

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

summary(Mod2)

##
## Call:
## glm(formula = Y_training ~ ., family = binomial, data = X_training[indice_sign_20])
##
## Deviance Residuals:
```

```

##      Min      1Q   Median      3Q      Max
## -3.7649 -0.6085 -0.1940  0.4972  3.7311
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)    2.401e+01  1.345e+01  1.785 0.074272 .
## PAR_SC         9.022e-03  1.585e-03  5.694 1.24e-08 ***
## PAR_SC_V       4.344e-01  1.221e-01  3.558 0.000374 ***
## PAR_ASE_M      3.489e+02  7.763e+01  4.494 6.98e-06 ***
## PAR_ASE_MV     3.567e+03  8.030e+02  4.441 8.94e-06 ***
## PAR_ASC       -6.457e+00  4.093e-01 -15.776 < 2e-16 ***
## PAR_ASC_V     -7.935e-01  1.455e-01 -5.455 4.90e-08 ***
## PAR_ASS       -3.358e+00  5.661e-01 -5.931 3.01e-09 ***
## PAR_ASS_V      8.326e+00  1.260e+00  6.607 3.93e-11 ***
## PAR_SFM_M      6.108e+01  9.208e+00  6.633 3.29e-11 ***
## PAR_SFM_MV     2.082e+03  5.702e+02  3.651 0.000261 ***
## PAR_MFCC1      5.082e-01  1.143e-01  4.447 8.70e-06 ***
## PAR_MFCC2     -5.476e-01  1.482e-01 -3.695 0.000220 ***
## PAR_MFCC3     -1.734e+00  1.878e-01 -9.231 < 2e-16 ***
## PAR_MFCC4      5.477e-01  2.155e-01  2.541 0.011041 *
## PAR_MFCC5     -1.144e+00  2.334e-01 -4.903 9.44e-07 ***
## PAR_MFCC6     -3.633e-01  2.291e-01 -1.586 0.112722
## PAR_MFCC7     -1.037e+00  2.560e-01 -4.049 5.15e-05 ***
## PAR_MFCC8      2.146e+00  3.061e-01  7.013 2.34e-12 ***
## PAR_MFCC9     -4.612e+00  3.421e-01 -13.481 < 2e-16 ***
## PAR_MFCC10     2.997e+00  3.951e-01  7.586 3.30e-14 ***
## PAR_MFCC11    -2.959e+00  4.661e-01 -6.348 2.18e-10 ***
## PAR_MFCC12     3.042e+00  5.290e-01  5.751 8.89e-09 ***
## PAR_MFCC13    -4.321e+00  6.233e-01 -6.931 4.17e-12 ***
## PAR_MFCC15    -2.952e+00  6.064e-01 -4.868 1.13e-06 ***
## PAR_THR_2RMS_TOT -9.695e+01  1.764e+01 -5.496 3.90e-08 ***
## PAR_THR_3RMS_TOT -1.513e+02  3.314e+01 -4.566 4.97e-06 ***
## PAR_THR_1RMS_10FR_MEAN 6.395e+01  7.037e+00  9.088 < 2e-16 ***
## PAR_THR_1RMS_10FR_VAR -1.601e+02  8.163e+01 -1.961 0.049842 *
## PAR_THR_2RMS_10FR_MEAN -9.636e+01  1.924e+01 -5.008 5.49e-07 ***
## PAR_THR_2RMS_10FR_VAR  4.447e+03  1.359e+03  3.272 0.001070 **
## PAR_THR_3RMS_10FR_MEAN 7.266e+02  5.947e+01 12.218 < 2e-16 ***
## PAR_THR_3RMS_10FR_VAR -2.107e+04  7.633e+03 -2.760 0.005781 **
## PAR_PEAK_RMS10FR_MEAN -2.567e-02  9.425e-03 -2.724 0.006452 **
## PAR_PEAK_RMS10FR_VAR  4.914e-05  2.365e-05  2.078 0.037703 *
## PAR_ZCD       -2.914e+01  3.928e+01 -0.742 0.458109
## PAR_3RMS_TCD   5.706e+02  9.435e+01  6.047 1.47e-09 ***
## PAR_ZCD_10FR_MEAN  4.170e+01  3.120e+01  1.337 0.181365
## PAR_ZCD_10FR_VAR  -6.965e+02  2.159e+02 -3.225 0.001258 **
## PAR_1RMS_TCD_10FR_VAR -3.158e+02  2.226e+02 -1.418 0.156080
## PAR_2RMS_TCD_10FR_MEAN 2.741e+02  3.607e+01  7.599 2.98e-14 ***
## PAR_2RMS_TCD_10FR_VAR -4.984e+03  9.656e+02 -5.161 2.46e-07 ***
## PAR_3RMS_TCD_10FR_MEAN -5.005e+02  1.068e+02 -4.687 2.77e-06 ***
## PAR_3RMS_TCD_10FR_VAR  1.751e+04  3.232e+03  5.417 6.05e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##

```

```
## Null deviance: 5914.0 on 4277 degrees of freedom
## Residual deviance: 3297.2 on 4234 degrees of freedom
## AIC: 3385.2
##
## Number of Fisher Scoring iterations: 8

#par(mfrow=c(2,2))
#plot(Mod2)
Mod2.predict_test=predict(Mod2, newdata=X_test[indice_sign_20], type="response")
```

ModAIC

```
library(MASS)
ModAIC=stepAIC(ModT, direction = "both", trace = FALSE)

summary(ModAIC)

##
## Call:
## glm(formula = Y_training ~ PAR_SC + PAR_SC_V + PAR_ASE_M + PAR_ASE_MV +
## PAR_ASC + PAR_ASC_V + PAR_ASS + PAR_ASS_V + PAR_SFM_M + PAR_SFM_MV +
## PAR_MFCC1 + PAR_MFCC2 + PAR_MFCC3 + PAR_MFCC4 + PAR_MFCC5 +
## PAR_MFCC6 + PAR_MFCC7 + PAR_MFCC8 + PAR_MFCC9 + PAR_MFCC10 +
## PAR_MFCC11 + PAR_MFCC12 + PAR_MFCC13 + PAR_MFCC15 + PAR_MFCC18 +
## PAR_THR_2RMS_TOT + PAR_THR_3RMS_TOT + PAR_THR_1RMS_1OFR_MEAN +
## PAR_THR_1RMS_1OFR_VAR + PAR_THR_2RMS_1OFR_MEAN + PAR_THR_2RMS_1OFR_VAR +
## PAR_THR_3RMS_1OFR_MEAN + PAR_THR_3RMS_1OFR_VAR + PAR_PEAK_RMS1OFR_MEAN +
## PAR_PEAK_RMS1OFR_VAR + PAR_2RMS_TCD + PAR_3RMS_TCD + PAR_ZCD_1OFR_VAR +
## PAR_1RMS_TCD_1OFR_VAR + PAR_2RMS_TCD_1OFR_MEAN + PAR_2RMS_TCD_1OFR_VAR +
## PAR_3RMS_TCD_1OFR_MEAN + PAR_3RMS_TCD_1OFR_VAR, family = binomial,
## data = X_training)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -3.8062  -0.6061  -0.1900   0.5041   3.7919
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)    2.115e+01  1.302e+01  1.624 0.104289
## PAR_SC          9.307e-03  9.480e-04  9.818 < 2e-16 ***
## PAR_SC_V        4.257e-01  1.150e-01  3.703 0.000213 ***
## PAR_ASE_M       3.307e+02  7.601e+01  4.351 1.36e-05 ***
## PAR_ASE_MV     3.448e+03  8.049e+02  4.284 1.84e-05 ***
## PAR_ASC       -6.486e+00  4.091e-01 -15.852 < 2e-16 ***
## PAR_ASC_V     -7.842e-01  1.394e-01 -5.627 1.83e-08 ***
## PAR_ASS      -3.350e+00  4.754e-01 -7.047 1.83e-12 ***
## PAR_ASS_V      8.326e+00  1.252e+00  6.652 2.89e-11 ***
## PAR_SFM_M      6.225e+01  8.977e+00  6.934 4.08e-12 ***
## PAR_SFM_MV     2.095e+03  5.649e+02  3.709 0.000208 ***
## PAR_MFCC1      5.168e-01  1.137e-01  4.545 5.50e-06 ***
## PAR_MFCC2     -5.730e-01  1.415e-01 -4.051 5.11e-05 ***
## PAR_MFCC3     -1.721e+00  1.872e-01 -9.191 < 2e-16 ***
## PAR_MFCC4      5.493e-01  2.155e-01  2.548 0.010827 *
## PAR_MFCC5     -1.126e+00  2.347e-01 -4.798 1.60e-06 ***
```

```
## PAR_MFCC6          -3.417e-01  2.291e-01  -1.492  0.135799
## PAR_MFCC7          -1.007e+00  2.565e-01  -3.927  8.60e-05 ***
## PAR_MFCC8           2.128e+00  3.072e-01   6.925  4.34e-12 ***
## PAR_MFCC9          -4.674e+00  3.428e-01 -13.634  < 2e-16 ***
## PAR_MFCC10         3.036e+00  3.953e-01   7.679  1.60e-14 ***
## PAR_MFCC11         -3.087e+00  4.694e-01  -6.575  4.85e-11 ***
## PAR_MFCC12         3.091e+00  5.298e-01   5.835  5.37e-09 ***
## PAR_MFCC13         -4.382e+00  6.236e-01  -7.027  2.10e-12 ***
## PAR_MFCC15         -3.222e+00  6.330e-01  -5.090  3.59e-07 ***
## PAR_MFCC18         1.249e+00  8.025e-01   1.556  0.119610
## PAR_THR_2RMS_TOT   -1.019e+02  1.790e+01  -5.693  1.25e-08 ***
## PAR_THR_3RMS_TOT   -1.222e+02  3.571e+01  -3.421  0.000625 ***
## PAR_THR_1RMS_10FR_MEAN  6.364e+01  7.053e+00   9.023  < 2e-16 ***
## PAR_THR_1RMS_10FR_VAR -1.752e+02  9.111e+01  -1.923  0.054472 .
## PAR_THR_2RMS_10FR_MEAN -9.724e+01  1.929e+01  -5.040  4.65e-07 ***
## PAR_THR_2RMS_10FR_VAR  4.057e+03  1.350e+03   3.006  0.002644 **
## PAR_THR_3RMS_10FR_MEAN  7.239e+02  5.966e+01  12.135  < 2e-16 ***
## PAR_THR_3RMS_10FR_VAR -2.152e+04  7.623e+03  -2.822  0.004766 **
## PAR_PEAK_RMS10FR_MEAN -2.654e-02  9.765e-03  -2.718  0.006566 **
## PAR_PEAK_RMS10FR_VAR  5.172e-05  2.526e-05   2.047  0.040655 *
## PAR_2RMS_TCD       1.666e+02  7.799e+01   2.136  0.032675 *
## PAR_3RMS_TCD       2.715e+02  1.644e+02   1.651  0.098695 .
## PAR_ZCD_10FR_VAR   -6.008e+02  2.285e+02  -2.629  0.008554 **
## PAR_1RMS_TCD_10FR_VAR -3.738e+02  2.193e+02  -1.705  0.088240 .
## PAR_2RMS_TCD_10FR_MEAN  1.609e+02  6.545e+01   2.458  0.013968 *
## PAR_2RMS_TCD_10FR_VAR -5.037e+03  9.555e+02  -5.271  1.35e-07 ***
## PAR_3RMS_TCD_10FR_MEAN -2.703e+02  1.526e+02  -1.771  0.076564 .
## PAR_3RMS_TCD_10FR_VAR  1.758e+04  3.174e+03   5.538  3.06e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 5914.0  on 4277  degrees of freedom
## Residual deviance: 3292.3  on 4234  degrees of freedom
## AIC: 3380.3
##
## Number of Fisher Scoring iterations: 8

#par(mfrow=c(2,2))
#plot(ModAIC)
ModAIC.predict_test=predict(ModAIC, newdata=X_test, type="response")
```

Question: 4

Dans un premier temps, traçons la courbe ROC du modèle **ModT** ainsi que les courbes des règles aléatoires et parfaites. Pour cela on récupère la liste des prédictions du modèle sur les données de *training*, grâce à la fonction **predict**. Et on les stocke dans la variable **ModT.predict_training**. Pour rappel, on a généré les prédictions sur les données *test* dans la partie précédente.

Ensuite on utilise la fonction **prediction** du package **ROCR** pour comparer nos prédictions aux valeurs réelles de **Y_test**.

```
ModT.predict_training=predict(ModT,type="response") #prediction de ModT sur les données training
library(ROCR)
```

```
## Warning: package 'ROCR' was built under R version 3.6.3
## Loading required package: gplots
## Warning: package 'gplots' was built under R version 3.6.3
##
## Attaching package: 'gplots'
## The following object is masked from 'package:stats':
##
##      lowess
```

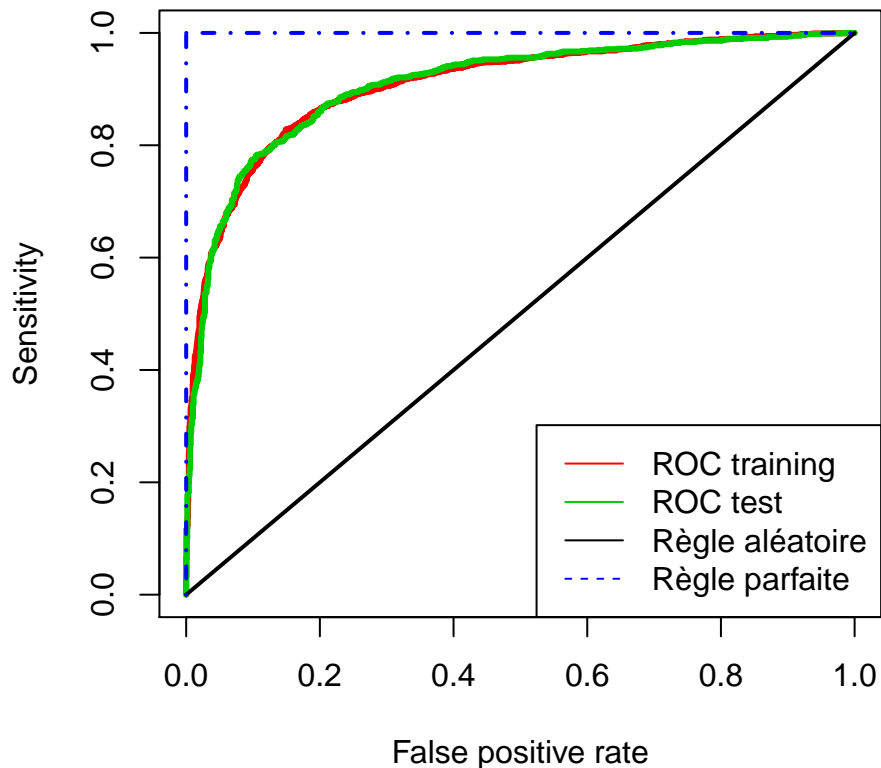
```
p_training=prediction(ModT.predict_training,Y_training)
p_test=prediction(ModT.predict_test,Y_test)
```

On peut alors utiliser la fonction **performance** avec les attributs *sens* et *fpr* pour calculer la courbe de ROC des données training et test, et afficher le tout dans un graphique. On ajoute à ce graphique la règle aléatoire (première bissectrice) et la règle parfaite qui prédit 100% des vrais positifs et 0% des faux positif (segment $(0,0) - (0,1)$ et $(1,0) - (1,1)$)

```
#plot ROC des données training
plot(performance(p_training,"sens","fpr"),col=2,lwd=3,add=FALSE)
#plot ROC des données test
plot(performance(p_test,"sens","fpr"),col=3,lwd=3,add=TRUE)
# règle aléatoire
lines(c(0,1),c(0,1),col=1,lty=1,lwd=2)
# règle parfaite
segments(c(0,0),c(0,1),c(0,1),c(1,1),col=4,lty=4,lwd=2)

title("ROC sur les données de training \n et de test pour ModT")
legend("bottomright",
      legend=c("ROC training",
               "ROC test", "Règle aléatoire",
               "Règle parfaite"),
      col=c(2,3,1,4), lty=c(1,1,1,2))
```

ROC sur les données de training et de test pour ModT



On remarque que les courbes ROC des données *training* et *test* se superposent. Donc le modèle se comporte de la même manière sur les données de *test* que sur les données de *training*. **//// EST CE QUE C'EST VRAI CA ?????? ///**

Traçons maintenant les courbes ROC de tous les modèles de la question 3, pour des prédictions sur les données de test. Cela va nous permettre de comparer les modèles entre eux. On procède de la même manière que précédemment pour générer les courbes ROC. Et on stock la sortie de la fonction **prediction** dans les variables **.ROC_pred_test**. Pour une raison de clarté on appelle aussi la fonction sur ModT, même si on l'a déjà fait au début de cette question.

```
Mod0.ROC_pred_test=prediction(Mod0.predict_test,Y_test)
ModT.ROC_pred_test=prediction(ModT.predict_test,Y_test)
Mod1.ROC_pred_test=prediction(Mod1.predict_test,Y_test)
Mod2.ROC_pred_test=prediction(Mod2.predict_test,Y_test)
ModAIC.ROC_pred_test=prediction(ModAIC.predict_test,Y_test)
```

Pour pouvoir comparer les modèles entre eux il est plus pertinent de comparer les aires sous les courbes ROC. Calculons donc ces aires pour nos cinq modèles. Pour ce faire on utilise la fonction **performance** avec l'attribut *auc*. On stock la sortie de cette fonction dans les variables **.perf_AUC**. La valeur de l'aire sous la courbe ROC se trouve alors dans l'attribut *y.values* de ces variables (qui est une liste). On arrondit cette valeur à 3 décimales, et on la stock dans les variables **.AUC**.

```
Mod0.perf_AUC = performance(Mod0.ROC_pred_test, "auc")
ModT.perf_AUC = performance(ModT.ROC_pred_test, "auc")
Mod1.perf_AUC = performance(Mod1.ROC_pred_test, "auc")
```

```
Mod2.perf_AUC = performance(Mod2.ROC_pred_test, "auc")
ModAIC.perf_AUC = performance(ModAIC.ROC_pred_test, "auc")
```

#Recuperation valeur aire

```
Mod0.AUC=round(Mod0.perf_AUC@y.values[[1]],3)
ModT.AUC=round(ModT.perf_AUC@y.values[[1]],3)
Mod1.AUC=round(Mod1.perf_AUC@y.values[[1]],3)
Mod2.AUC=round(Mod2.perf_AUC@y.values[[1]],3)
ModAIC.AUC=round(ModAIC.perf_AUC@y.values[[1]],3)
```

On peut alors tracer un graphique contenant les ROC de nos cinq modèles. On affiche la valeur de l'aire sous la courbe de ROC de chaque modèle dans la légende.

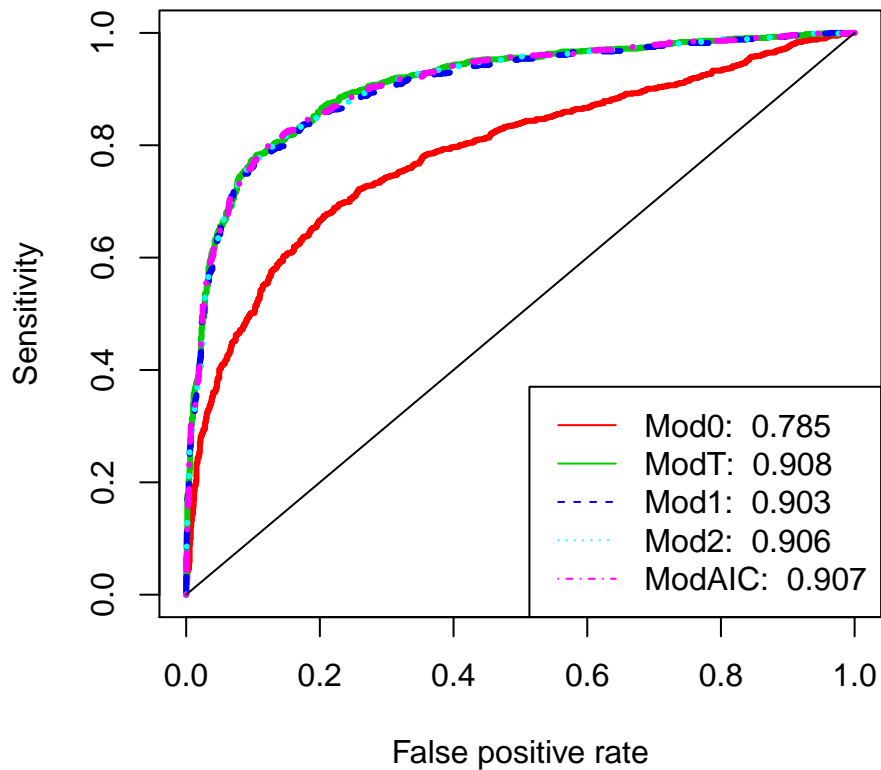
#Plot tous les ROC

```
plot(performance(Mod0.ROC_pred_test,"sens","fpr"),col=2,lwd=3,add=FALSE)
plot(performance(ModT.ROC_pred_test,"sens","fpr"),col=3,lwd=3,lty=1,add=TRUE)
plot(performance(Mod1.ROC_pred_test,"sens","fpr"),col=4,lwd=3,lty=2,add=TRUE)
plot(performance(Mod2.ROC_pred_test,"sens","fpr"),col=5,lwd=3,lty=3,add=TRUE)
plot(performance(ModAIC.ROC_pred_test,"sens","fpr"),col=6,lwd=3,lty=4,add=TRUE)
lines(c(0,1),c(0,1),col=1,lty=1,lwd=1)
```

#Titre et legende avec valeur aire

```
title("ROC sur les données de test")
legend("bottomright",
      legend=c(paste("Mod0: ", Mod0.AUC),
                paste("ModT: ", ModT.AUC),
                paste("Mod1: ", Mod1.AUC),
                paste("Mod2: ", Mod2.AUC),
                paste("ModAIC: ", ModAIC.AUC)),
      col=c(2,3,4,5,6), lty=c(1,1,2,3,4))
```

ROC sur les données de test



Partie 2: K plus proches voisins (K-NN)

Question 1: Principe du K-NN

La méthode des k plus proches voisins (knn ou k nearest neighbours) est une méthode simple à utiliser dans des cas de classification ou régression. Il s'agit d'une méthode non paramétrique dans laquelle le modèle mémorise les observations de l'ensemble d'apprentissage et s'en sert pour les observations des données de test. Elle consiste à fixer un nombre k de voisins des nouvelles données d'entrée, de sélectionner les k plus proches (en fonction d'une certaine distance, la distance euclidienne par exemple) et de conserver la classe correspondant à celle majoritairement représentée parmi les différents voisins retenus.

Pour choisir le meilleur k, il faut tester différentes valeurs et retenir celle qui minimise le taux d'erreur de l'ensemble de test.

Question 2: Implémentation K-NN

En premier lieu on effectue la méthode pour

$$K = 1$$

voisin.

```
library(class)
pred_knn_test <- knn(X_training, X_test, Y_training, k=1) # tout d'abord on teste avec k=1
summary(pred_knn_test)
```



```
##      0      1
## 1278  891
err_test_1=sum(pred_knn_test!=Y_test)/length(Y_test)
```

On fait maintenant une boucle qui permet de calculer cette méthode pour

$$k$$

allant de 1 à 200 voisins et on retient à chaque fois les erreurs d'apprentissage et d'entraînement.

```
err_test = rep(NA,length=150)
err_train = rep(NA, length=150)
for (k in 1:200){
  mod_train <-knn(X_training,X_training,Y_training,k=k)
  pred_knn <-knn(X_training,X_test, Y_training, k=k)
  err_train[k] = mean(mod_train!=Y_training)
  err_test[k] = mean(pred_knn!=Y_test)}

save(mod_train,file = "mod_train_sta203.RData")
save(pred_knn,file = "pred_knn_sta203.RData")
save(err_test,file = "err_test_sta203.RData")
save(err_train,file = "err_train_sta203.RData")
```

Pour ne pas avoir à faire le calcul lors de la compilation du rapport, nous avons enregistré les données. On affiche maintenant la courbe des deux erreurs en fonction du nombre de voisins retenus.

```
load("mod_train_sta203.RData")
load("pred_knn_sta203.RData")
load("err_test_sta203.RData")
load("err_train_sta203.RData")
K = which.min(err_test)
K # c'est toujours le cas où k=1 qui minimise le taux d'erreur de test
```

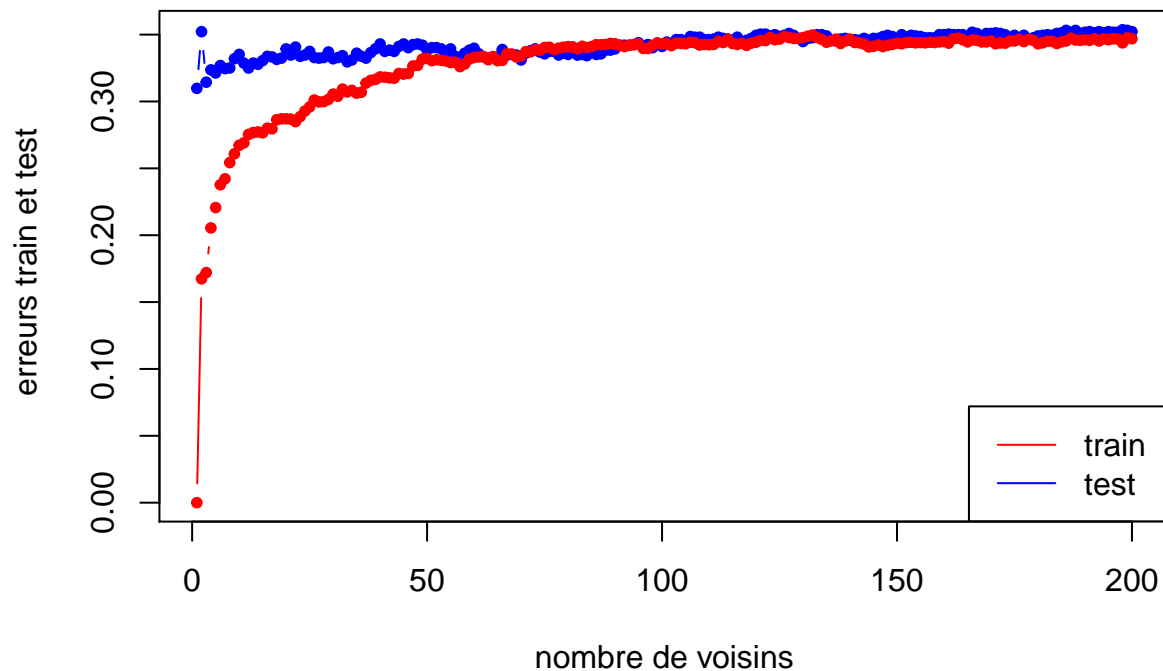
```
## [1] 1
```

```
err_test[(-1)]
```

```
##      [1] 0.3522361 0.3144306 0.3236515 0.3213462 0.3268787 0.3245735 0.3250346
##      [8] 0.3319502 0.3351775 0.3287229 0.3250346 0.3287229 0.3278008 0.3305671
##     [15] 0.3337944 0.3333333 0.3314892 0.3324112 0.3393269 0.3347165 0.3407100
##     [22] 0.3337944 0.3347165 0.3374827 0.3328723 0.3324112 0.3328723 0.3370217
##     [29] 0.3319502 0.3328723 0.3342554 0.3296450 0.3310281 0.3360996 0.3337944
##     [36] 0.3324112 0.3365606 0.3393269 0.3430152 0.3374827 0.3384048 0.3374827
##     [43] 0.3411710 0.3430152 0.3402490 0.3425542 0.3430152 0.3420931 0.3384048
##     [50] 0.3402490 0.3402490 0.3393269 0.3365606 0.3393269 0.3337944 0.3328723
##     [57] 0.3360996 0.3384048 0.3397879 0.3356385 0.3333333 0.3324112 0.3333333
##     [64] 0.3333333 0.338658 0.3347165 0.3351775 0.3347165 0.3310281 0.3360996
##     [71] 0.3379438 0.3393269 0.3370217 0.3356385 0.3374827 0.3374827 0.3347165
##     [78] 0.3360996 0.3351775 0.3407100 0.3347165 0.3351775 0.3342554 0.3374827
##     [85] 0.3351775 0.3356385 0.3384048 0.3384048 0.3393269 0.3420931 0.3420931
##     [92] 0.3420931 0.3425542 0.3439373 0.3416321 0.3425542 0.3402490 0.3430152
##     [99] 0.3411710 0.3443983 0.3430152 0.3462425 0.3462425 0.3453204 0.3485477
##    [106] 0.3480867 0.3480867 0.3462425 0.3462425 0.3467036 0.3480867 0.3471646
##    [113] 0.3467036 0.3457815 0.3480867 0.3448594 0.3457815 0.3471646 0.3494698
##    [120] 0.3503919 0.3503919 0.3490088 0.3499308 0.3485477 0.3494698 0.3508529
##    [127] 0.3494698 0.3471646 0.3448594 0.3471646 0.3476256 0.3499308 0.3499308
##    [134] 0.3490088 0.3457815 0.3453204 0.3467036 0.3471646 0.3457815 0.3462425
```

```
## [141] 0.3457815 0.3471646 0.3457815 0.3471646 0.3471646 0.3485477 0.3494698
## [148] 0.3485477 0.3467036 0.3499308 0.3480867 0.3494698 0.3490088 0.3485477
## [155] 0.3476256 0.3476256 0.3476256 0.3499308 0.3499308 0.3503919 0.3499308
## [162] 0.3499308 0.3499308 0.3480867 0.3517750 0.3508529 0.3480867 0.3508529
## [169] 0.3508529 0.3513140 0.3508529 0.3499308 0.3467036 0.3494698 0.3467036
## [176] 0.3494698 0.3467036 0.3476256 0.3490088 0.3494698 0.3499308 0.3494698
## [183] 0.3508529 0.3513140 0.3531581 0.3517750 0.3531581 0.3508529 0.3517750
## [190] 0.3522361 0.3513140 0.3522361 0.3513140 0.3522361 0.3517750 0.3517750
## [197] 0.3536192 0.3531581 0.3522361
```

```
vec_k = 1:200
plot(vec_k, err_test, type="b", col="blue", xlab="nombre de voisins", ylab="erreurs train et test", pch=15,
     ylim=range(c(err_test, err_train)))
lines(vec_k, err_train, type="b", col="red", pch=20)
legend("bottomright", lty=1, col=c("red", "blue"), legend = c("train ", "test "))
```



Question 3 : Quelle conclusion en tirer ?

En regardant uniquement l'erreur d'apprentissage, c'est le modèle le plus complexe qui est désigné, c'est-à-dire celui de

$$k = 1$$

. Il s'agit du phénomène de sur-apprentissage. On veut vérifier que ce soit est adapté en regardant l'erreur de test (courbe bleue). Celle-ci pointe aussi le même modèle mais cette vérification confirme qu'ici le sur-apprentissage est le meilleur choix de méthode.

Partie 3: Régression Ridge

Question 1: Interêt de la régression ridge

Dans le cadre du modèle binomial qui est le nôtre, la régression ridge est un type de régression logistique pénalisée, qui correspond donc à l'optimisation d'un critère. L'ajustement des paramètres de pénalisation permet de choisir un modèle plus performant qu'une "simple" régression logistique.

Question 2

Les deux cas extrêmes représentent les valeurs extrêmes pour

$$\lambda$$

qui permet la pénalisation. Pour un

$$\lambda$$

élevé comme

$$10^{10}$$

, le critère est fortement pénalisé, on se rapproche du cas i.i.d. Pour un

$$\lambda$$

faible comme

$$10^{-2}$$

on est dans le cas inverse où le critère est peu pénalisé, ce qui correspond presque à une régression logistique habituelle.

```
library(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 3.6.3
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 3.0-2
```

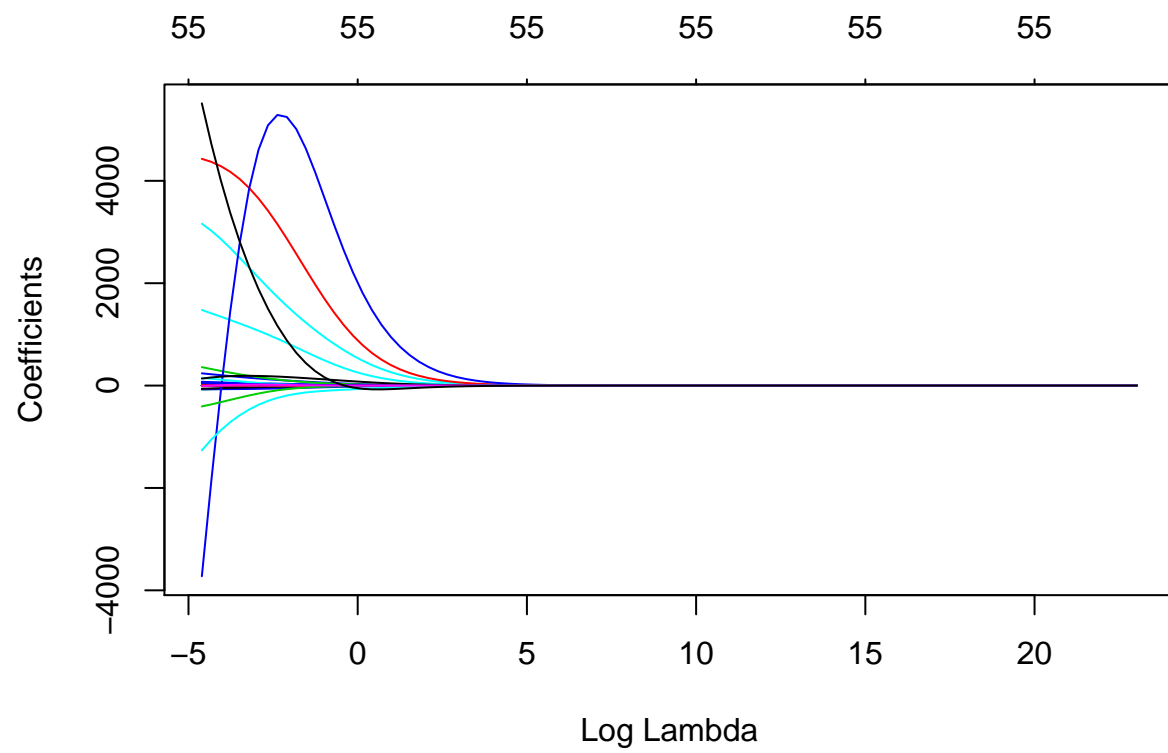
```
les_lambda=10^seq(10,-2,length=100)
```

```
#model_matrix=model.matrix(GENRE~.,data)[,-1]
```

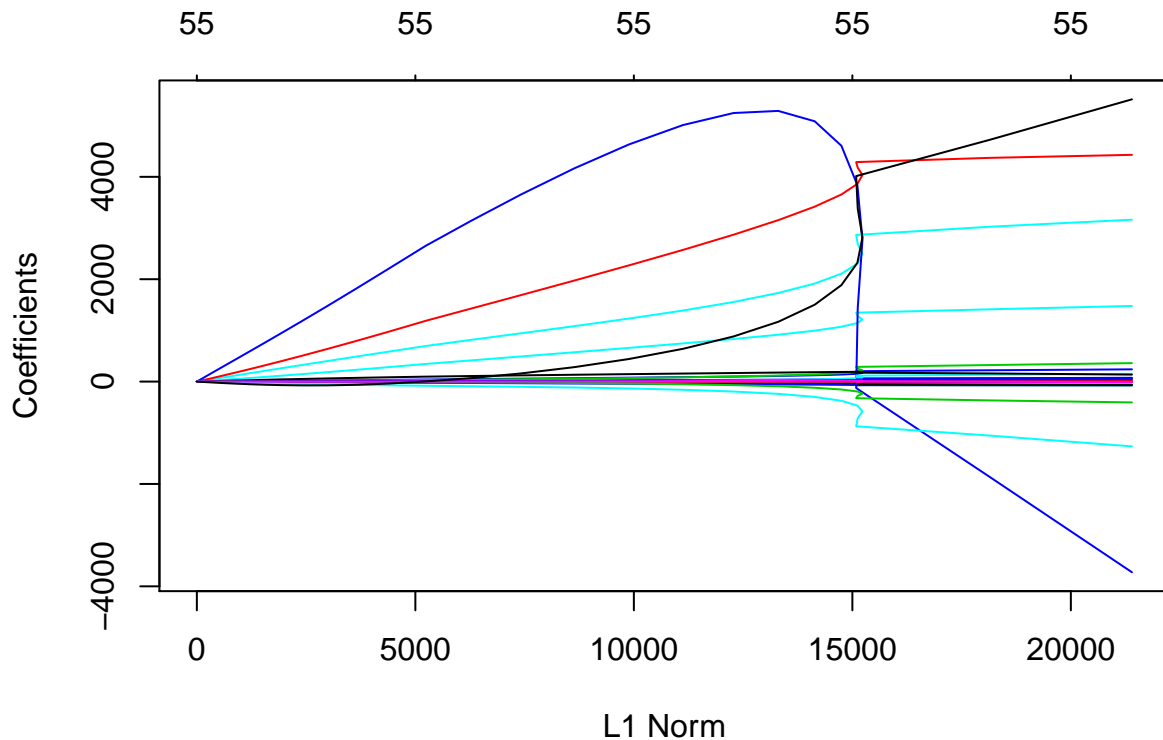
```
model_matrix=model.matrix(Y~.,X)[,-1]
```

```
ridge=glmnet(model_matrix[train,],Y_training,family = c("binomial"),alpha=0,lambda=les_lambda) #alpha=0
```

```
plot(ridge,xvar="lambda") #coef vs log(lambda)
```



```
plot(ridge) # norme L1
```



```
ridge_predict=predict(ridge, newx = model_matrix[!train,], s = 10^5)
```

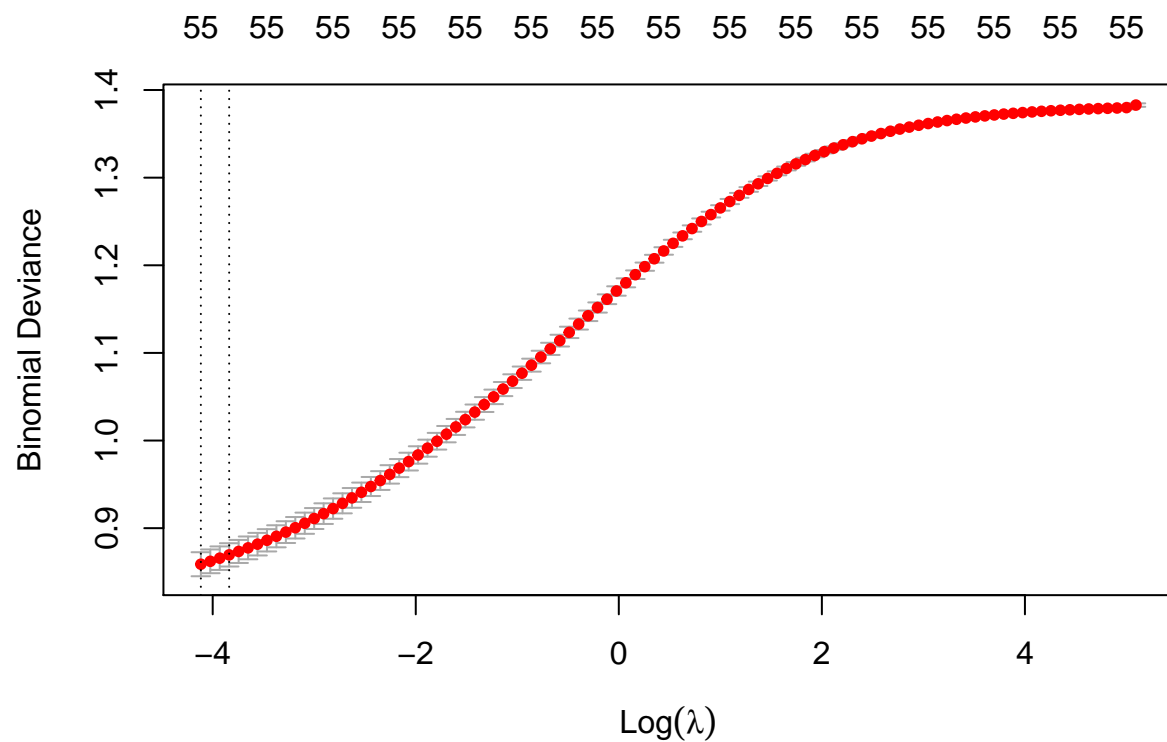
Question 3

La régression ridge implique d'optimiser un hyperparamètre **lambda**. La fonction **cv.glmnet** permet de faire cette optimisation automatiquement, en appliquant un algorithme de validation croisée afin de comparer un modèle ridge pour différentes valeurs de lambda. Cette valeur optimale sera le **lambda** qui minimise la courbe affichée lorsque l'on appelle la fonction plot sur la sortie de **cv.glmnet**.

```
set.seed(314)
(cv=cv.glmnet(model_matrix[train,],Y_training,family = c("binomial"),alpha=0,nfolds = 10))#alpha=0=ridge

##
## Call: cv.glmnet(x = model_matrix[train, ], y = Y_training, nfolds = 10,      family = c("binomial")
##
## Measure: Binomial Deviance
##
##      Lambda Measure      SE Nonzero
## min 0.01630  0.8588 0.01371      55
## 1se 0.02155  0.8695 0.01329      55

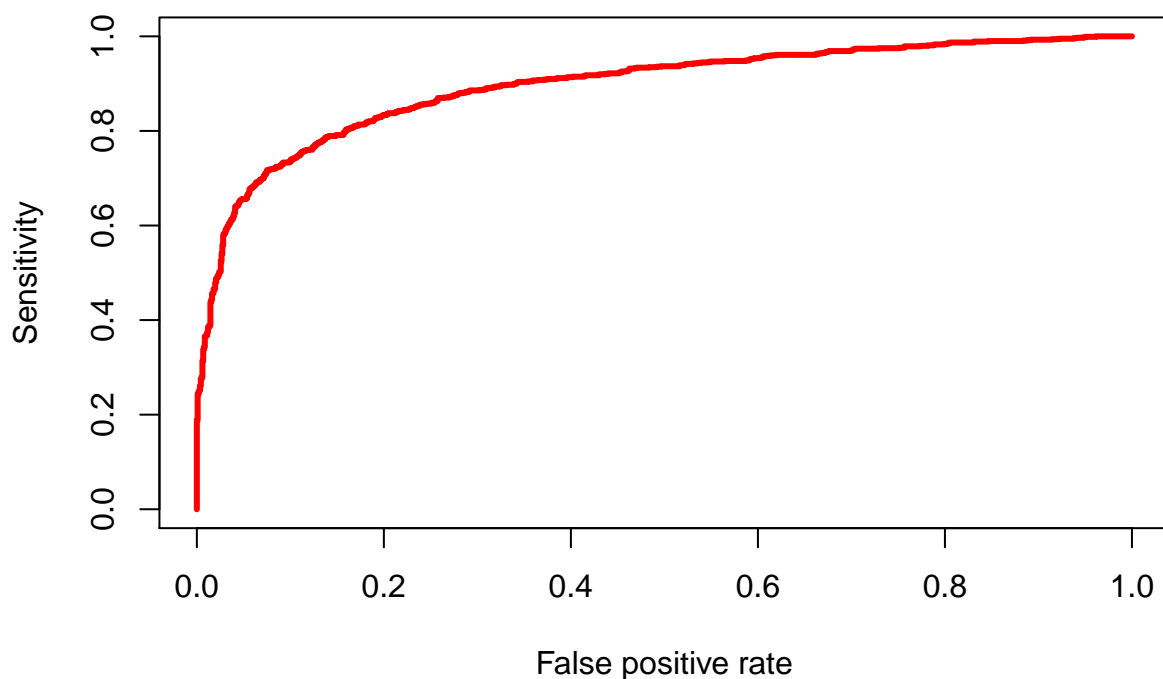
plot(cv)
```



```
(best_lambda=cv$lambda.min) #meilleur lambda

## [1] 0.016298
ridge_pred_best1=predict(ridge,s=cv$lambda.min ,newx=model_matrix[!train,])
mean((ridge_pred_best1-Y_test)^2)

## [1] 3.254702
plot(performance(prediction(ridge_pred_best1,Y_test),"sens","fpr"),col=2,lwd=3,lty=1,add=FALSE)
```



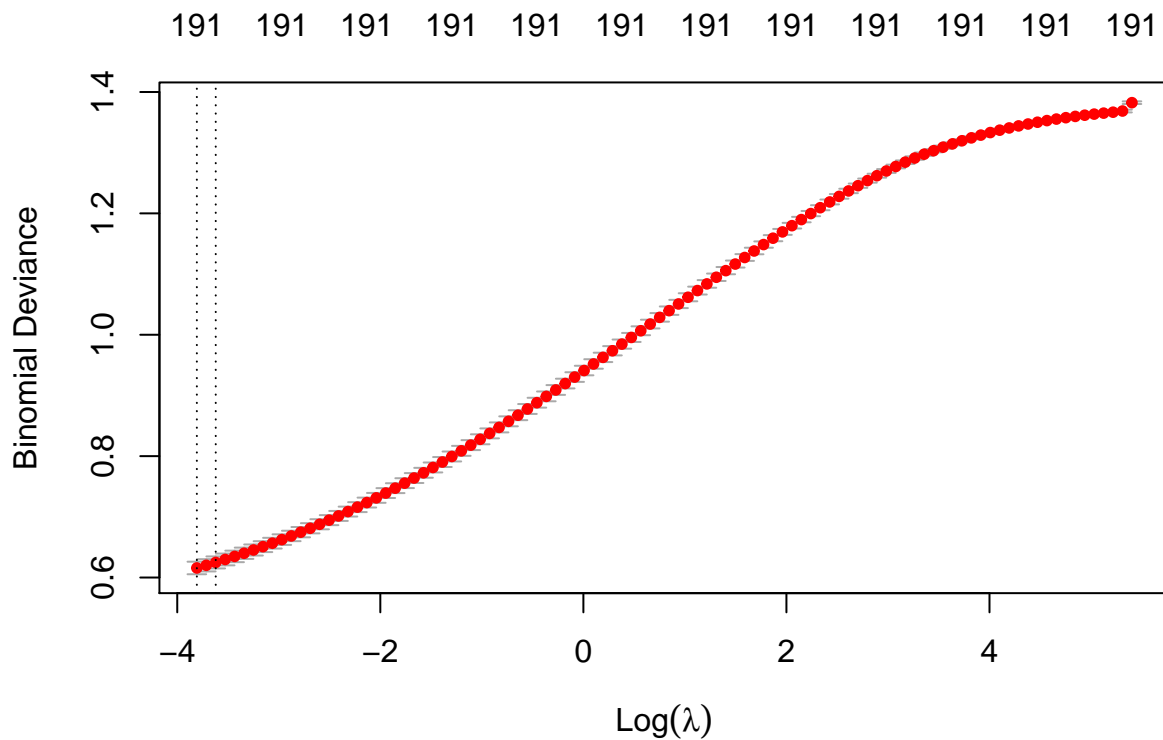
Question 4

On reprend la question 3 en utilisant toutes les variables du data-frame **data** (excepté la variable GENRE).

```
set.seed(4658)
model_matrix=model.matrix(Y~.,data[-c(192)])[, -1]
(cv_all=cv.glmnet(model_matrix[train,],Y_training,family = c("binomial"),alpha=0,nfolds = 10))#alpha=0=

##
## Call: cv.glmnet(x = model_matrix[train, ], y = Y_training, nfolds = 10,      family = c("binomial")
##
## Measure: Binomial Deviance
##
##      Lambda Measure      SE Nonzero
## min 0.02221  0.6158 0.010346      191
## 1se 0.02676  0.6249 0.009881      191

plot(cv_all)
```



```
(best_lambda=cv_all$lambda.min) #meilleur lambda
```

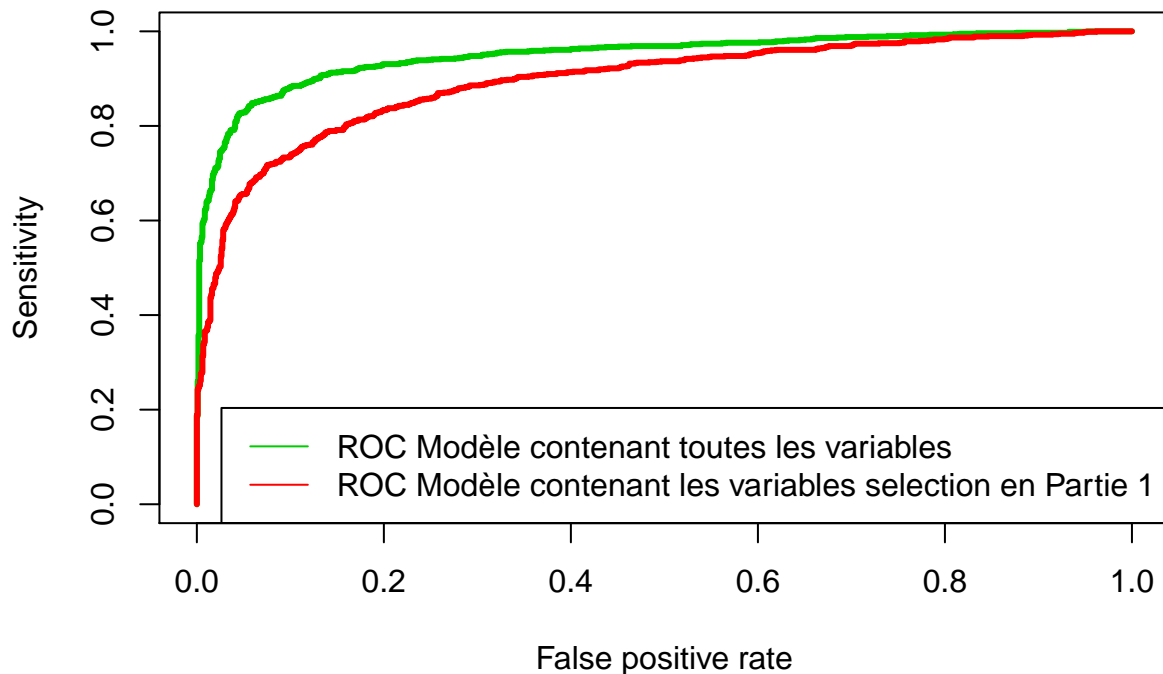
```
## [1] 0.02221237
```

```
ridge_all_pred_best1=predict(cv_all,s=cv_all$lambda.min ,newx=model_matrix[!train,])
mean((ridge_all_pred_best1-Y_test)^2)
```

```
## [1] 5.569992
```

```
plot(performance(prediction(ridge_all_pred_best1,Y_test),"sens","fpr"),col=3,lwd=3,lty=1,add=FALSE)
plot(performance(prediction(ridge_pred_best1,Y_test),"sens","fpr"),col=2,lwd=3,lty=1,add=TRUE)
title("ROC sur les modèles ridge")
legend("bottomright",
      legend=c("ROC Modèle contenant toutes les variables",
               "ROC Modèle contenant les variables selection en Partie 1"),
      col=c(3,2), lty=c(1,1))
```


ROC sur les modèles ridge



Conclusion

On aurait pu aussi essayer d'autres méthodes de classification comme celle des forêts aléatoires ou des arbres CART.

Bonus

Implémentons un arbre de décision et une forêt aléatoire.

Arbre de décision

Commençons par un arbre décision élagué.

```
library(rpart)
library(rpart.plot)
```

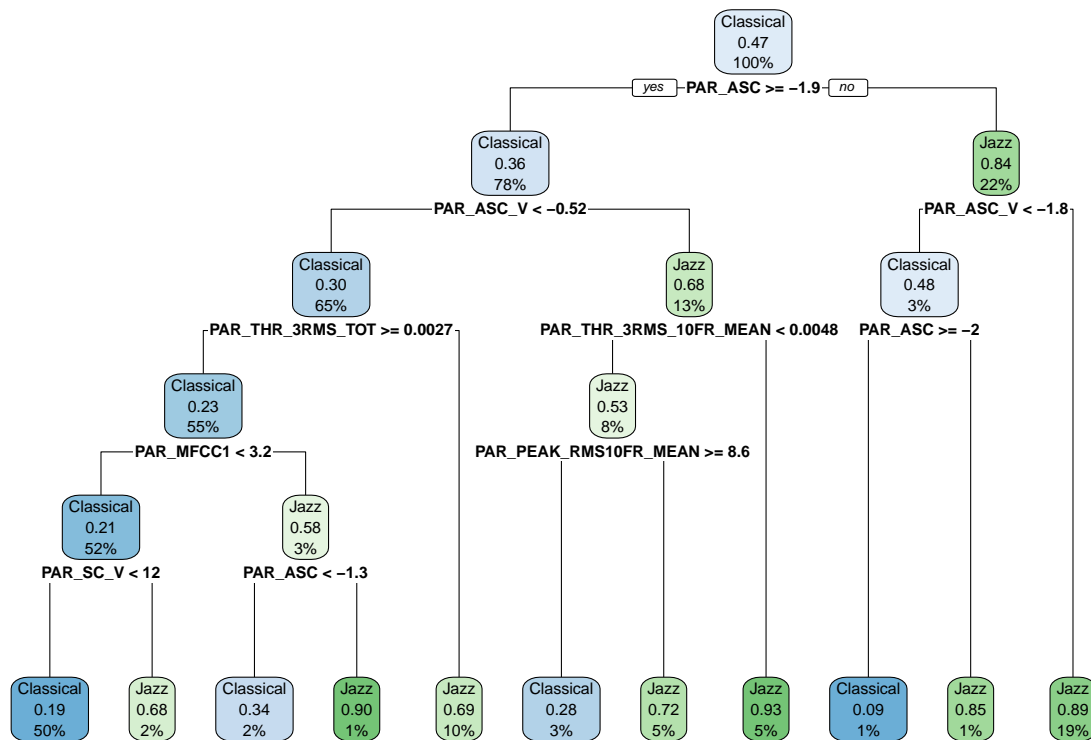
```
## Warning: package 'rpart.plot' was built under R version 3.6.3
```

```
arbre = rpart(GENRE_training~., data=X_training)
arbre$cptable
```

```
##           CP nsplit rel error   xerror   xstd
## 1 0.32103689     0 1.0000000 1.0000000 0.01627115
## 2 0.10119641     1 0.6789631 0.6894317 0.01525050
## 3 0.07826520     2 0.5777667 0.6181456 0.01479290
## 4 0.01620140     3 0.4995015 0.5229312 0.01402718
```

```
## 5 0.01271186      5 0.4670987 0.4925224 0.01374121
## 6 0.01196411      7 0.4416750 0.4715852 0.01353153
## 7 0.01171486      8 0.4297109 0.4685942 0.01350068
## 8 0.01000000     10 0.4062812 0.4591226 0.01340152
```

```
arbre = prune(arbre, cp = arbre$cptable[which.min(arbre$cptable[, "xerror"]), "CP"])
rpart.plot(arbre)
```



La prédiction sur les données de test nous donne:

```
arbre_predict = predict(arbre, X_test, type="class")
table(arbre_predict, GENRE_test)
```

```
##          GENRE_test
## arbre_predict Classical Jazz
##   Classical      970  283
##   Jazz          202  714
```

Forêt aléatoire

Implémentons maintenant une forêt aléatoire de 50 arbres.

```
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 3.6.3
## randomForest 4.6-14
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##      margin

foret=randomForest(GENRE_training~.,data=X_training, ntree = 50,
                    mtry = 2, na.action = na.roughfix)
foret_predict = predict(foret, X_test, type="class")
table(foret_predict, GENRE_test)

##              GENRE_test
## foret_predict Classical Jazz
##      Classical      1092  159
##      Jazz           80   838
```

Erreur de test

Regradons l'erreur moyenne sur les données de prédictions de ces deux méthodes.

```
mean(arbre_predict!=GENRE_test)
```

```
## [1] 0.2236053
```

```
mean(foret_predict!=GENRE_test)
```

```
## [1] 0.110189
```

COMPARER AUX AUTRES MODELES