PROJET METHODES AVANCEES EN APPRENTISSAGE SUPERVISE ET NON SUPERVISE

BENREGUIG Yanis
CARRIE Chloe
REGAZZETTI Léa

M2 SISE
Promotion 2021-2022

Table of Contents

Introduction

Dans le cadre du cours de Méthodes avancées en apprentissage supervisé et non-supervisé, nous avons réalisé un projet qui consistait à appliquer de manière concrète les méthodes de classification vues durant ce cours. Lors de la première partie de ce projet, nous avons eu l'occasion de travailler sur une méthode supervisée qui est le one-class svm et une méthode non supervisée qui est l'Analyse en Composantes Principales à noyaux.

Lors de la seconde partie, nous avons, tout d'abord, fait une étape de recherche de données. Une fois que nous avons trouvé un jeu de données qui nous semblait praticable, nous avons fait une étape de pré-traitement des données. Une fois ceci fait, nous avons appliqué à notre jeu de données les trois méthodes demandées, qui sont le modèle linéaire pénalisé par une fonction de régularisation elasticnet, un réseau de neurones avec une couche cachée et un support vector-machine.

Partie 1 : TP4 : Détection de nouveauté par One-class SVM et Kernel PCA

Dans cette partie, il s'agit de répondre aux questions posées dans le sujet du TP4.

1. Présentation de l'étude cas

L'étude de cas porte sur des données de médecine, plus précisément sur des tumeurs de cancer du sein, classées malignes ou bénignes à partir de différents variables quantitatives. Par ailleurs, on se trouve dans le cas d'un jeu de données déséquilibré, avec plus de cellules bénignes que malignes. Il s'agit ici d'aborder la problématique de la détection de nouveauté.

2. Lecture et description des données

Question 1 : Informations générales

En consultant le fichier, on apprend que la base de données est composée de 11 variables observées sur 699 individus, qui étaient à l'origine divisés en plusieurs groupes en fonction de la date de report faite par le docteur en charge de l'étude clinique. La variable cible est *"Class"*, codée 2 pour bénigne et 4 pour maligne.

Question 2 : Chargement des données

On remarque que les données sont séparées par une virgule et que les données manquantes sont indiquées par des "?", il faut donc l'indiquer dans la commande **read.table** permettant d'importer les données. Cela se fait grâce aux arguments *sep* et *na.strings* respectivement. De plus, les noms de colonnes ne sont pas présents dans le jeu de données, nous pouvons alors les obtenir à partir du fichier d'informations.

```
D = read.table("breast-cancer-wisconsin.data", sep = ",", na.strings = "?")
colnames(D) = c("Sample_Code_Number", "Clump_Thickness",
"Uniformity_Cell_Size", "Uniformity_Cell_Shape", "Marginal_Adhesion",
"Single_Epithelial_Cell_Size", "Bare_Nuclei", "Bland_Chromatin",
"Normal_Nucleoli", "Mitoses", "Class")
```

Nous avons en sortie une variable *D*, qui est un dataframe contenant les données, avec 699 observations et 11 variables, ainsi que les noms de colonnes. De plus, les "?" sont détectés comme des valeurs manquantes (NA).

Question 3 : Inspection des données

```
class(D)

## [1] "data.frame"
```

La première commande, **class**, nous donne le type de l'objet passé en paramètre, c'est-à-dire de la variable *D* que nous avons créée à la question précédente. Il s'agit bien d'un dataframe.

```
str(D)

## 'data.frame':    699 obs. of  11 variables:
##  $ Sample_Code_Number         : int  1000025 1002945 1015425 1016277
1017023 1017122 1018099 1018561 1033078 1033078 ...
##  $ Clump_Thickness            : int  5 5 3 6 4 8 1 2 2 4 ...
##  $ Uniformity_Cell_Size       : int  1 4 1 8 1 10 1 1 1 2 ...
##  $ Uniformity_Cell_Shape      : int  1 4 1 8 1 10 1 2 1 1 ...
##  $ Marginal_Adhesion          : int  1 5 1 1 3 8 1 1 1 1 ...
##  $ Single_Epithelial_Cell_Size: int  2 7 2 3 2 7 2 2 2 2 ...
##  $ Bare_Nuclei                : int  1 10 2 4 1 10 10 1 1 1 ...
##  $ Bland_Chromatin            : int  3 3 3 3 3 9 3 3 1 2 ...
##  $ Normal_Nucleoli            : int  1 2 1 7 1 7 1 1 1 1 ...
##  $ Mitoses                    : int  1 1 1 1 1 1 1 1 5 1 ...
##  $ Class                      : int  2 2 2 2 2 4 2 2 2 2 ...
```

La deuxième commande, **str**, permet de visualiser de manière succincte la structure de l'objet passé en argument, à savoir ici du dataframe. Ainsi, on retrouve le type de l'objet, mais également le nombre d'individus et de variables de notre jeu de données, à savoir 699 observations et 11 variables. On retrouve aussi la liste des différentes variables, que nous avons choisi de renommer pour une facilité de lecture, avec leur type, et les premières valeurs prises par chaque variable.

```
kable(head(D)) # on utilise kable pour le formatage de la sortie avec une table, plutôt que la sortie de base
```

Sample_Code_Number	Clump_Thickness	Uniformity_Cell_Size	Uniformity_Cell_Shape	Marginal Adhesion	Single_Epithelial Cell_Size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses	Class
1000025	5	1	1	1	2	1	3	1	1	2
1002945	5	4	4	5	7	10	3	2	1	2
1015425	3	1	1	1	2	2	3	1	1	2
1016277	6	8	8	1	3	4	3	7	1	2
1017023	4	1	1	3	2	1	3	1	1	2
1017122	8	10	10	8	7	10	9	7	1	4

La troisième commande, **head**, permet de visualiser les premières lignes de notre jeu de données.

```
kable(summary(D), digits = 3)
```

Sample_Code_Number	Clump_Thickness	Uniformity_Cell_Size	Uniformity_Cell_Shape	Marginal_Adhesion	Single_Epithelial_Cell_Size	Bare_Nuclei	Bland_Chromatin	Normal_Nucleoli	Mitoses	Class
Min. : 61634	Min. : 1.000	Min. : 1.000	Min. : 1.000	Min. : 1.000	Min. : 1.000	Min. : 1.000	Min. : 1.000	Min. : 1.000	Min. : 1.000	Min. :2.00
1st Qu.: 870688	1st Qu.: 2.000	1st Qu.: 1.000	1st Qu.: 1.000	1st Qu.: 1.000	1st Qu.: 2.000	1st Qu.: 1.000	1st Qu.: 2.000	1st Qu.: 1.000	1st Qu.: 1.000	1st Qu.: 2.00
Median : 1171710	Median : 4.000	Median : 1.000	Median : 1.000	Median : 1.000	Median : 2.000	Median : 1.000	Median : 3.000	Median : 1.000	Median : 1.000	Median :2.00
Mean : 1071704	Mean : 4.418	Mean : 3.134	Mean : 3.207	Mean : 2.807	Mean : 3.216	Mean : 3.545	Mean : 3.438	Mean : 2.867	Mean : 1.589	Mean :2.69
3rd Qu.: 1238298	3rd Qu.: 6.000	3rd Qu.: 5.000	3rd Qu.: 5.000	3rd Qu.: 4.000	3rd Qu.: 4.000	3rd Qu.: 6.000	3rd Qu.: 5.000	3rd Qu.: 4.000	3rd Qu.: 1.000	3rd Qu.: 4.00
Max. :13454352	Max. :10.000	Max. :10.000	Max. :10.000	Max. :10.000	Max. :10.000	Max. :10.000	Max. :10.000	Max. :10.000	Max. :10.000	Max. :4.00
NA	NA	NA	NA	NA	NA	NA’s :16	NA	NA	NA	NA

Enfin, la fonction **summary**, produit des statistiques descriptives des différentes variables. Etant donné que toutes les variables ont été détectées comme des variables numériques, pour chacune d’entre elles, on va retrouver la valeur minimale, le premier quartile, la médiane, la moyenne, le troisième quartile et la valeur maximale. On peut d’ores et déjà remarquer que la variable *Bare_Nuclei* comporte des valeurs manquante.

Une autre remarque que nous avons pu faire sur le jeu de données concerne la variable *Sample_Code_Number*. Il s'agit des identifiants des patients, certes cette variable n'est pas pertinente dans des modèles de machine learning, cependant elle nous apprend quelque chose que nous avons jugé assez intéressant pour en parler.

```
length(unique(D$Sample_Code_Number))
```

```
## [1] 645
```

En effet, on peut voir que seuls 645 individus sur les 699 au total, possèdent un identifiant unique. Nous avons donc des observations avec les mêmes identifiants.

```
kable(D %>% dplyr::select(Sample_Code_Number) %>%
group_by(Sample_Code_Number) %>% summarise(n=length(Sample_Code_Number))
%>%arrange(desc(n)) %>% filter(n > 1), align = 'c')
```

Sample_Code_Number	n
1182404	6
1276091	5
1198641	3
320675	2
385103	2
411453	2
466906	2
493452	2
560680	2
654546	2
695091	2
704097	2
733639	2
734111	2
769612	2
798429	2
822829	2
897471	2
1017023	2
1033078	2
1061990	2
1070935	2
1100524	2
1105524	2

Sample_Code_Number	n
1114570	2
1115293	2
1116116	2
1116192	2
1143978	2
1158247	2
1168736	2
1171710	2
1173347	2
1174057	2
1212422	2
1218860	2
1238777	2
1240603	2
1277792	2
1293439	2
1299596	2
1299924	2
1320077	2
1321942	2
1339781	2
1354840	2

On peut supposer que certains patients ont été observés plusieurs fois à différents moments. Ceci ne fait pas partie du TD mais nous nous sommes demandés si cette redondance d'information n'entraînerait pas un biais dans nos analyses futures.

3. Séparation des données en “train” et “test”

Question 4 : Valeurs manquantes

Afin d’identifier les valeurs manquantes, nous pouvons utiliser la fonction **complete.cases**, qui permet d’obtenir les individus sans valeurs manquantes dans le dataframe. Ainsi, en utilisant son contraire, avec le *!*, on obtient la strate des individus avec des valeurs manquantes.

```
kable(D[!complete.cases(D), ])
```

	Sample_Code_Number	Clump_Thickness	Uniformity_Cell_Size	Uniformity_Cell_Shape	Marginal_Adhesion	Single_Epithelial_Cell_Size	Bare_Nuclei	Bland_Chromatin	Normal_Nucleoli	Mitoses	Class
24	1057013	8	4	5	1	2	NA	7	3	1	4
41	1096800	6	6	6	9	6	NA	7	8	1	2
140	1183246	1	1	1	1	1	NA	2	1	1	2
146	1184840	1	1	3	1	2	NA	2	1	1	2
159	1193683	1	1	2	1	3	NA	1	1	1	2
165	1197510	5	1	1	1	2	NA	3	1	1	2
236	1241232	3	1	4	1	2	NA	3	1	1	2
250	169356	3	1	1	1	2	NA	3	1	1	2
276	432809	3	1	3	1	2	NA	2	1	1	2
293	563649	8	8	8	1	2	NA	6	10	1	4
295	606140	1	1	1	1	2	NA	2	1	1	2
298	61634	5	4	3	1	2	NA	2	3	1	2
316	704168	4	6	5	6	7	NA	4	9	1	2
322	733639	3	1	1	1	2	NA	3	1	1	2
412	1238464	1	1	1	1	1	NA	2	1	1	2
618	1057067	1	1	1	1	1	NA	1	1	1	2

Nous pouvons remarquer qu’il y a 16 individus comportant des valeurs manquantes dans le jeu de données, tous pour la variable *Bare_Nuclei*. Ces individus portent les identifiants suivants : 21, 41, 140, 146, 159, 165, 236, 250, 276, 293, 295, 298, 316, 322, 412 et 618.

Question 5 : Gestion des valeurs manquantes

Dans cette question, on nous demande de modifier *D* de sorte à ce qu'il ne possède que des données complètes. Plusieurs méthodes sont alors possibles, soit utiliser uniquement les données complètes, soit imputer les données manquantes. Etant donné que seulement 16 observations sont manquantes, nous avons décidé de supprimer les observations présentant des valeurs manquantes. Pour cela, il suffit de ne garder que les individus sans valeurs manquantes grâce à la fonction **complete.cases**.

```
D = D[complete.cases(D),]
nrow(D)

## [1] 683

sum(is.na(D))

## [1] 0
```

Le jeu de données comporte alors 683 observations et plus aucune donnée manquante.

Question 6 : Séparation variables explicatives et variable cible

Les variables explicatives sont les variables 2 à 10, en effet l'identifiant est exclu car il n'apporte pas d'information dans les analyses. La variable cible est placée en dernière position du dataframe.

```
X = D[,2:10]
ncol(X)

## [1] 9

y = D[,11]
```

Il y a donc 9 variables explicatives, permettant de prédire la variable cible, à savoir le type de la tumeur.

Question 7 : Recodage de la variable cible

La variable cible étant actuellement identifiée comme une variable numérique, prenant seulement les valeurs 2 et 4, nous allons la recoder.

```
y = dummy_cols(y)[,3]
```

Pour cela, nous avons utiliser la fonction **dummy_cols** qui permet de créer un codage disjonctif complet de la variable passée en paramètre. Puis, comme la modalité bénigne doit être recodée en 0 et la modalité maligne en 1, il s'agit de sélectionner la troisième colonne du résultat fourni par la fonction.

Question 8 : Indices des observations benignes et malignes

Rappelons que les tumeurs bénignes sont codées en 0 et les malignes en 1. Il faut alors sélectionner les observations qui correspondent à ces critères avec la fonction **which**.

```
benin = which(y==0)
malin = which(y==1)
```

Nous pouvons alors voir que 444 observations sont bénignes et 239 sont malignes.

Question 9 : Constitution des échantillons d'apprentissage et de test

L'ensemble d'entraînement contient les 200 premières observations bénignes. Les autres observations bénignes ainsi que les observations malignes constituent l'ensemble de test.

```
train_set = benin[1:200]
test_set = c(benin[201:length(benin)], malin)
```

Dans la variable *train_set* se trouve les 200 observations correspondantes à l'échantillon d'apprentissage. La variable *test_set* quand à elle contient les 483 indices des observations de l'échantillon de test.

4. One-Class SVM

Le One-Class SVM pour la détection de nouveautés est une méthode d'apprentissage supervisé dont l'objectif est de déterminer la classe d'appartenance d'un objet, en apprenant à partir d'un ensemble d'entraînement contenant uniquement des objets d'une seule classe, bien qu'il existe d'autres classes possibles. Pour cela, on définit un hyperplan qui va rassembler une grande majorité des données d'un côté et peu voire pas de données de l'autre. Ainsi, le plan est utiliser comme frontière, et on va chercher à maximiser la marge entre cette frontière et l'origine de l'espace.

Question 10 : Chargement de la libraire

Afin d'installer la librairie nécessaire, nous l'installons avec la commande **install.packages**, si elle n'a pas déjà été installée sur la machine, puis la chargeons avec la fonction **library**.

```
#install.packages("e1071")
library(e1071)
```

Question 11 : Estimation du modèle

La fonction **svm** permet d'entraîner un modèle de one-class SVM. Pour cela, il faut le spécifier dans l'argument *type*. De plus, on utilisera un noyau gaussien (radial), avec l'argument *kernel*, de paramètre *gamma* = 1/2. Il n'y a pas besoin de centrer-réduire les données vu qu'elles sont toutes sur la même échelle de 1-10.

```
oc_svm_fit = e1071::svm(x = X[train_set,], y = y[train_set], type = "one-
classification", kernel = "radial", gamma = 1/2, scale = FALSE)
```

On obtient ainsi un objet de type svm grâce auquel on pourra faire des prédictions.

Question 12 : Prédictions

Grâce à la commande **predict**, nous pouvons prédire les scores des observations de test. Pour cela, nous spécifions l'objet de type svm que nous avons crée précédemment, les données de test, et l'argument *decision.values* à TRUE afin que les valeurs de décision soient calculées et retournées.

```
oc_svm_pred_test = predict(oc_svm_fit,X[test_set,],decision.values = TRUE)
```

Question 13 : Exécution de commandes

```
attr(oc_svm_pred_test ,"decision.values")
```

```
##                 /
## 381 -2.251634e-04
## 383 -3.189874e+00
## 384  5.239372e-01
## 385  5.239372e-01
## 386 -5.619410e+00
## 388 -4.957322e+00
## 389  2.738621e-01
## 390 -3.286926e+00
699 -5.703733e+00
```

```
oc_svm_score_test=-as.numeric(attr(oc_svm_pred_test ,"decision.values"))
```

La première commande permet d'accéder à l'attribut *decision.values* de l'objet *oc_svm_pred_test* des prédictions. Par soucis de clareté, nous avons affiché seulement les premieres valeurs. Ensuite, on définit les scores comme l'opposé des valeurs de décision.

5. Courbe ROC

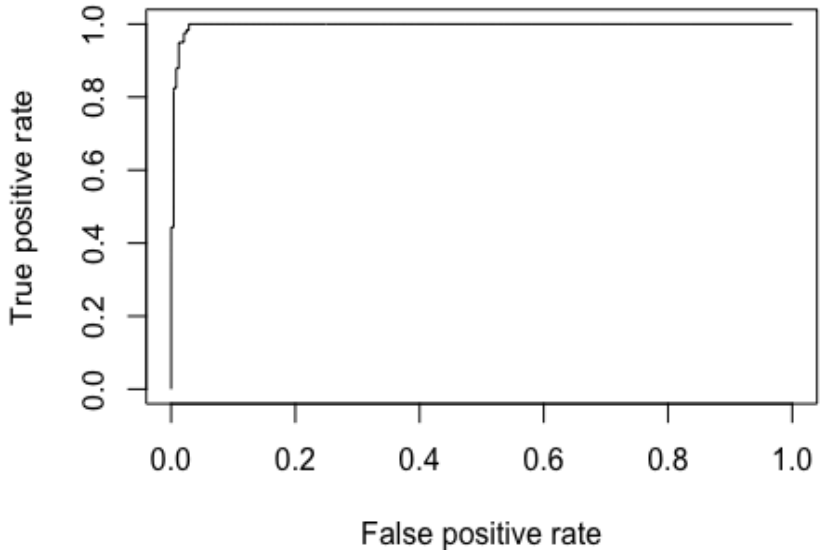
Question 14 : Chargement de la librairie ROCR

La commande **install.packages** permet d'installer la librairie, si celle-ci ne l'est pas déjà, puis la fonction **library** la charge.

```
#install.packages("ROCR")
library(ROCR)
```

Question 15 : Exécution de commandes

```
pred_oc_svm=prediction(oc_svm_score_test ,y[test_set])
oc_svm_roc = performance(pred_oc_svm, measure = "tpr", x.measure = "fpr")
plot(oc_svm_roc)
```



Dans un premier temps, la commande **prediction**, qui prend en argument les prédictions puis les vraies valeurs de la variable cible, renvoie un objet de type *prediction*, qui contient notamment le nombre de faux positifs, de vrais positifs, de vrais négatifs ainsi que de faux négatifs. Ensuite, la commande **performance**, permet d'effectuer des mesures de performances. Dans notre cas, on souhaite calculer le taux de vrais positifs et le taux de faux positifs à l'aide de l'objet de type *prediction* que nous avons crée précédemment. Puis, nous affichons le taux de vrais positifs en fonction du taux de faux positifs, qui est donc la courbe ROC.

Question 16 : Commentaire sur les performances du modèle

Parmi les outils d'évaluation d'un modèle, il y a la courbe ROC. Celle-ci comporte de nombreux avantages, dont notamment sa visualisation graphique. De ce fait, nous pouvons voir grâce à la courbe ROC que le modèle est performant. En effet, le point idéal est le cas

(0,1), et on s'en rapproche. Une autre manière de connaître la performance du modèle est de calculer l'indicateur numérique associé à la courbe ROC, qui est l'aire sous la courbe (AUC). L'AUC indique la probabilité pour que la fonction de score du modèle place un positif devant un négatif. Dans le meilleur des cas, l'AUC vaut 1.

```
performance(pred_oc_svm, measure = "auc")@y.values

## [[1]]
## [1] 0.9959188
```

Dans notre cas, l'AUC est très proche de 1, notre modèle est donc performant.

6. Kernel PCA

Le kernel PCA est une méthode d'apprentissage non supervisé fondée sur l'Analyse en Composantes Principales à noyaux. Tout d'abord, il s'agit de projeter les données d'apprentissage dans un espace F grâce au calcul de la matrice à noyau K. Puis de transformer cette matrice afin ensuite de la diagonaliser pour pouvoir représenter les données d'apprentissage dans un espace de plus petite dimension. Enfin, on projette les données de test dans cet espace réduit.

Question 17 : Exécution de commandes

```
#install.packages("kernlab")
library(kernlab)
kernel=rbfdot(sigma=1/8)
Ktrain=kernelMatrix(kernel, as.matrix(X[train_set,]))
```

Tout d'abord, nous chargeons la librairie **kernlab** nécessaire à l'utilisation des commandes suivantes. La commande **rbfdot** est une fonction de génération de noyau, et plus précisemment d'un noyau gaussien rbf de paramètre *sigma* = 1/8. On obtient ainsi en sortie une fonction qui peut être utilisée en tant qu'argument du noyau d'une autre fonction. Cette fonction à noyau est ainsi utilisée comme paramètre de la commande **kernelMatrix** avec la matrice des données d'entraînement. En sortie, on obtient une matrice carrée de taille 200*200 (la taille de l'échantillon d'apprentissage) des produits scalaires calculés avec la fonction à noyau précédemment créée. C'est à dire la matrice à noyau des données d'entraînement.

Question 18 : Exécution de commandes

```
k2=apply(Ktrain,1,sum)
k3=apply(Ktrain,2,sum)
k4=sum(Ktrain)

n = length(train_set)
KtrainCent=matrix(0,ncol=n,nrow=n)

for (i in 1:n)
{
  for (j in 1:n)
  {
    KtrainCent[i,j]=Ktrain[i,j]-1/n*k2[i]-1/n*k3[j]+1/n^2*k4
  }
}
```

Dans un premier temps, la variable *k2* correspond à la somme en ligne de la matrice Ktrain. *k3* est la somme en colonne de la matrice Ktrain. *k4* est la somme des éléments de la matrice Ktrain. Dans l'équation (1), *k2* correspond à la somme sur s allant de 1 à n (n étant la taille du vecteur train_set, à savoir la taille des données d'apprentissage) de K(xi, xs). *k3*

correspond à la somme sur r allant de 1 à n de K(xr, xj). Enfin, *k4* correspond au calcul de la double somme. Par la suite, on instancie *KtrainCent*, une matrice carrée de taille 200*200, avec des 0. Puis on parcourt les lignes et les colonnes pour transformer Ktrain en KtrainCent en appliquant la formule (1).

Question 19 : Décomposition spectrale

```
eigen_KtrainCent = eigen(KtrainCent)
```

La fonction **eigen** calcule les valeurs propres et les vecteurs propres associés de la matrice passée en paramètre de la fonction. Ces calculs permettront de représenter dans un espace de dimension réduit les données d'entraînement.

Question 20 : Sélection des 80 axes principaux

```
s = 80
A = eigen_KtrainCent$vectors[,1:s] %*%
diag(1/sqrt(eigen_KtrainCent$values[1:s]))
```

Tout d'abord, on instance une variable *s* à 80, qui va permettre de sélectionner les 80 premiers axes principaux. Les coefficients associés à ces axes sont ensuite contenus dans la variable *A*. Ils sont calculés comme le produit matriciel des 80 premiers vecteurs propres avec une matrice diagonale, créée avec la commande **diag**, composée d'une forme normalisée (1/racine(vp)) des valeurs propres.

Question 21 : Exécution de commandes

```
K = kernelMatrix(kernel, as.matrix(X))
```

De manière analogue à la question 17, dans la commande **kernelMatrix** nous utilisons la fonction à noyau créée précedemment, et cette fois-ci toutes les données de notre dataset. Nous obtenons en sortie une matrice carrée d'ordre 683, la matrice à noyau *K*.

Question 22 : Termes composants l'équation (4)

```
p1 = diag(K[test_set,test_set])
p2 = apply(K[test_set,train_set], 1, sum)
p3 = sum(K[train_set, train_set])
```

On initialise les 3 termes composants l'équation (4) dans les variables *p1*, *p2* et *p3* respectivement.

Question 23 : vecteur de la quantité (4) pour toute observation de test

```
ps = p1 - (2/n)*p2 + (1/(n^2))*p3
```

A partir des variables définies précédemment, on peut déterminer le vecteur *ps* qui pour toute observation des données de test donne la quantité (4).

Question 24 : termes composants l'équation (5)

```
f1 = K[test_set,train_set]
f2 = apply(K[train_set, train_set], 1, sum)
f3 = p2
f4 = p3
```

Les termes composants l'équation (5) sont assignés aux variables *f1*, *f2*, *f3* et *f4* respectivment. Le troisième terme est égal au terme *p2* déjà calculé. Le quatrième terme est égal au terme *p3* déjà calculé.

Question 25 : matrice de la quantité (5) pour toute observation de test

```
n2 = length(test_set)
fl = matrix(0, ncol = s, nrow = n2)

for (m in 1:s) #m pour les axes retenus
{
  cpt = 0 #le compteur pour boucler
  for (i in 1:n2) #n2 pour le nombre de données tests
  {
    cpt = cpt + 1
    tmp = 0 #variable temporaire permettant de calculer la somme au final
    for (j in 1:n) #pour boucler sur les n lignes
    {
      tmp = tmp + (A[j,m] * (f1[cpt,j] - (1/n)*f2[j] - (1/n)*f3[cpt] +
(1/n^2)*f4))
    }
    fl[cpt,m] = tmp
  }
}
```

Dans un premier temps, on instancie *fl* la matrice qui va contenir les quantités (5) pour toute observation de test, ainsi que *n2* le nombre de données tests. Puis, à partir des variables créées précédemment, on remplit la matrice avec la formule données en (5).

Question 26 : score défini en (3)

```
kpca_score_test = ps - apply(fl^2, 1, sum)
```

Le score en (3) est défini comme la quantité calculée en (4), contenue dans la variable *ps*, moins la somme sur les axes retenus de la quantité calculée en (5) au carré.

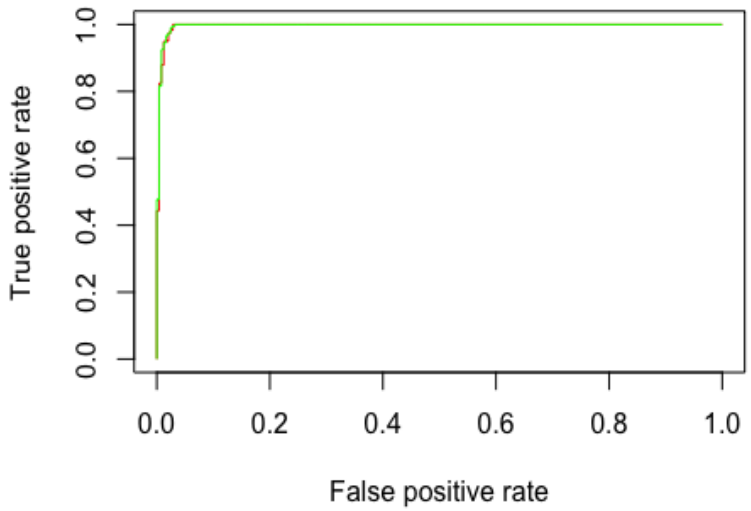
Question 27 : courbe ROC

De la même manière que pour le SVM, on calcule les prédictions et les mesures de perfomance cette fois-ci pour le kernel PCA.

```
pred_kpca=prediction(kpca_score_test ,y[test_set])
kpca_roc = performance(pred_kpca, measure = "tpr", x.measure = "fpr")
```

Puis nous affichons les deux courbes sur le même graphique afin de les comparer. Nous avons modifié les couleurs pour une meilleure lisibilité.

```
plot(oc_svm_roc, col='red')
plot(kpca_roc, add=T,col="green")
```



Au premier abord, les deux méthodes semblent aussi performantes l'une que l'autre, avec peut-être une légère supériorité pour le kernel PCA. Nous pouvons vérifier cela en calculant l'aire sous la courbe pour cette méthode. Pour rappel, nous avions déjà calculé l'AUC pour le SVM et nous avions obtenu 0.9959188.

```
performance(pred_kpca, measure = "auc")@y.values

## [[1]]
## [1] 0.9962789
```

En effet, l'AUC est légèrement plus élevé pour le kernel PCA. Ainsi, pour cette étude, nous préconiserions cette méthode pour répondre à la problématique, en sachant toutefois que le SVM donne également de très bons résultats.

Partie 2 : Etude de cas

Dans cette partie, il s'agit de mettre en oeuvre et de comparer plusieurs méthodes d'apprentissage supervisé sur un jeu de données de notre choix.

1. Import et présentation du jeu de données

Le jeu de données provient du domaine de la santé et correspond à des mesures physiques réalisées sur des individus ainsi que des performances liés à l'exercice physique. Il s'agit de prédire le niveau de performance des individus en fonction des différentes variables numériques.

Tout d'abord, nous importons le jeu de données avec la commande **read.csv**.

```
D = read.csv("bodyPerformance.csv",header=TRUE,sep=',')
```

Puis nous en affichons sa structure.

```
str(D)
```

```
## 'data.frame':    13393 obs. of  12 variables:
##  $ age                    : num  27 25 31 32 28 36 42 33 54 28 ...
##  $ gender                 : chr  "M" "M" "M" "M" ...
##  $ height_cm              : num  172 165 180 174 174 ...
##  $ weight_kg              : num  75.2 55.8 78 71.1 67.7 ...
##  $ body.fat_.             : num  21.3 15.7 20.1 18.4 17.1 22 32.2 36.9
27.6 14.4 ...
##  $ diastolic              : num  80 77 92 76 70 64 72 84 85 81 ...
##  $ systolic               : num  130 126 152 147 127 119 135 137 165 156
...
##  $ gripForce              : num  54.9 36.4 44.8 41.4 43.5 23.8 22.7 45.9
40.4 57.9 ...
##  $ sit.and.bend.forward_cm: num  18.4 16.3 12 15.2 27.1 21 0.8 12.3 18.6
12.1 ...
##  $ sit.ups.counts         : num  60 53 49 53 45 27 18 42 34 55 ...
##  $ broad.jump_cm          : num  217 229 181 219 217 153 146 234 148 213
...
##  $ class                  : chr  "C" "A" "C" "B" ...
```

Le jeu de données est composée de 12 variables observées sur 13393 individus. Quasiment toutes les variables sont quantitatives, sauf le sexe (*gender*), mais que nous recoderons par la suite, et la variable cible.

Les 12 variables qui composent le jeu de données sont les suivantes : * *age*, l'âge exprimé en années * *gender*, le genre codé M ou F * *height_cm*, la taille en centimètres * *weight_kg*, le poids en kilogrammes * *body.fat_.*, la graisse corporelle exprimée en pourcentage * *diastolic*, la pression artérielle diastolique exprimée en millimètres de mercure (mmHg) * *systolic*, la pression artérielle systolique exprimée en millimètres de mercure (mmHg) * *gripForce*, le résultat d'un test de force musculaire * *sit.and.bend.forward_cm*, la mesure en centimètres lorsque l'on s'assoit et que l'on se penche en avant * *sit.ups.counts*, le nombre maximal de redressements assis en 2 minutes * *broad.jump_cm*, le saut en longueur exprimé en centimètres * *class*, la performance.

Nous pouvons également regarder les statistiques descriptives du jeu de données afin de mieux en prendre connaissance.

```
summary(D)

##       age           gender            height_cm       weight_kg
##  Min.   :21.00   Length:13393       Min.   :125.0   Min.   : 26.30
##  1st Qu.:25.00   Class :character   1st Qu.:162.4   1st Qu.: 58.20
##  Median :32.00   Mode  :character   Median :169.2   Median : 67.40
##  Mean   :36.78                      Mean   :168.6   Mean   : 67.45
##  3rd Qu.:48.00                      3rd Qu.:174.8   3rd Qu.: 75.30
##  Max.   :64.00                      Max.   :193.8   Max.   :138.10
##    body.fat_.      diastolic        systolic       gripForce
##  Min.   : 3.00   Min.   :  0.0   Min.   :  0.0   Min.   : 0.00
##  1st Qu.:18.00   1st Qu.: 71.0   1st Qu.:120.0   1st Qu.:27.50
##  Median :22.80   Median : 79.0   Median :130.0   Median :37.90
##  Mean   :23.24   Mean   : 78.8   Mean   :130.2   Mean   :36.96
##  3rd Qu.:28.00   3rd Qu.: 86.0   3rd Qu.:141.0   3rd Qu.:45.20
##  Max.   :78.40   Max.   :156.2   Max.   :201.0   Max.   :70.50
##  sit.and.bend.forward_cm sit.ups.counts  broad.jump_cm      class
##  Min.   :-25.00          Min.   : 0.00   Min.   :  0.0   Length:13393
##  1st Qu.: 10.90          1st Qu.:30.00   1st Qu.:162.0   Class :character
##  Median : 16.20          Median :41.00   Median :193.0   Mode  :character
##  Mean   : 15.21          Mean   :39.77   Mean   :190.1
##  3rd Qu.: 20.70          3rd Qu.:50.00   3rd Qu.:221.0
##  Max.   :213.00          Max.   :80.00   Max.   :303.0
```

Puis nous en affichons les premières lgines.

```
head(D)
```

```
##   age gender height_cm weight_kg body.fat_. diastolic systolic gripForce
## 1  27      M     172.3     75.24       21.3        80      130      54.9
## 2  25      M     165.0     55.80       15.7        77      126      36.4
## 3  31      M     179.6     78.00       20.1        92      152      44.8
## 4  32      M     174.5     71.10       18.4        76      147      41.4
## 5  28      M     173.8     67.70       17.1        70      127      43.5
## 6  36      F     165.4     55.40       22.0        64      119      23.8
##   sit.and.bend.forward_cm sit.ups.counts broad.jump_cm class
## 1                    18.4            60          217     C
## 2                    16.3            53          229     A
## 3                    12.0            49          181     C
## 4                    15.2            53          219     B
## 5                    27.1            45          217     B
## 6                    21.0            27          153     B
```

Voyons maintenant la répartition des différentes modalités de la variable cible.

```
prop.table(table(D$class))
```

```
##
##         A         B         C         D
## 0.2499813 0.2499067 0.2500560 0.2500560
```

On peut voir que les modalités sont réparties quasiment de la même manière dans le jeu de données. Il ne sera pas question de ré-échantillonage ou de jeu de données déséquilibré.

2. Pré-traitement des données

Commençons par vérifier la présence de données manquantes dans le dataframe.

```
ok <- complete.cases(D) #Récupération de booléens pour savoir si une
observation est une valeur manquante ou non
print(sum(!ok))
```

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

Aucune valeur manquante n'est à déplorer. Nous pouvons passer au recodage de certaines variables.

Pour la variable *gender*, codée “M” ou “F”, nous la recodons en 0, 1.

```
#install.packages("dplyr")
library(dplyr)
D$gender = recode(D$gender, F = 0, M = 1)
```

Ensuite, nous séparons le jeu de données en données d’apprentissage et de test. Nous avons choisi 30% pour le test et 70% pour le train.

```
set.seed(123)
id = sample(nrow(D),nrow(D)*0.7)

training_data = D[id,]
testing_data = D[-id,]
```

Nous vérifions que l’équilibre des classes est toujours présent.

```
print(prop.table(table(training_data$class)))

##
##         A         B         C         D
## 0.2459733 0.2540800 0.2507733 0.2491733

print(prop.table(table(testing_data$class)))

##
##         A         B         C         D
## 0.2593330 0.2401692 0.2483823 0.2521155
```

En fonction des librairies utilisées par la suite, nous aurons besoin soit des variables réunies, soit des variables explicatives séparées de la variable cible.

```
X_train = training_data[,-12]
X_test = testing_data[,-12]

y_train = training_data[,12]
y_test = testing_data[,12]
```

Par ailleurs, les variables explicatives étant exprimées dans des unités différentes, nous allons les standardiser. Nous utilisons uniquement les moyennes et écarts-types calculés sur l'échantillon d'apprentissage pour normaliser les deux échantillons.

```
mean_train = sapply(X_train,mean)
sd_train <- sapply(X_train,sd)

X_train_scale = data.frame(scale(X_train, center = mean_train, scale =
sd_train))
X_test_scale = data.frame(scale(X_test, center = mean_train, scale =
sd_train))

training_data_scale = cbind(X_train_scale, class = y_train)
testing_data_scale = cbind(X_test_scale, class = y_test)
```

3. Validation croisée permettant d'évaluer les performances de différentes méthodes

3.1. Modèle linéaire pénalisé par une fonction de régularisation elasticnet

Elastic net est une méthode de régression servant à régulariser des modèles linéaires ou logistiques.

La librarie utilisée ici est la librairie glmnet, utilisée pour de l'ajustement des modèles Lasso ou Elastic Net pour de la régression.

Installation et chargement de la librairie glmnet

```
#install.packages('glmnet')
library(glmnet)
```

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

```
##
## Attaching package: 'Matrix'
```

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

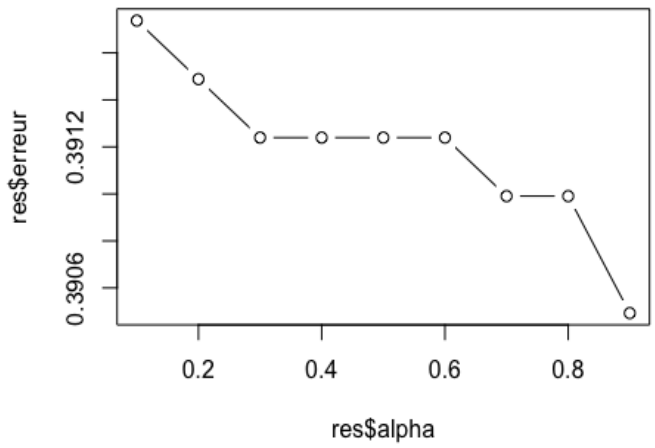
```
## Loaded glmnet 4.1-3
```

Nous allons tester les différentes valeurs du paramètre alpha comprises entre 0.1 et 0.9

```
set.seed(123)
#liste des hyperparamètres
param_alpha <- list(alpha = seq(from=0.1, to=0.9, by = 0.1))
#Nous transformons maintenant la variable param_alpha en grid
param_alpha=expand.grid(param_alpha)

#nombre de ligne de param_alpha
n=nrow(param_alpha)
#Nous initialisons des vecteurs de 0 pour lambda et pour l'erreur
lambda = rep(0,n)
erreur=rep(0,n)

# recherche des meilleurs paramètres
for (i in 1:n){
  cv.en =
cv.glmnet(x=as.matrix(X_train),y=as.matrix(y_train),family="multinomial",type
.multinomial="grouped",nfolds=10,alpha=param_alpha$alpha[i], standardize = T)
  #Plus petit lambda
  #lambda min
  lambda[i] = cv.en$lambda.min

  #prédiction
  pred.en =
predict(cv.en,as.matrix(X_test),s=c(cv.en$lambda.min),type="class")
  conf=table(y_test,pred.en)
  e=1-sum(diag(conf))/sum(conf)
  erreur[i]=e
}

res = cbind(cbind(param_alpha,lambda),erreur)
print(res)

##   alpha       lambda    erreur
## 1   0.1 0.0002980219 0.3917372
## 2   0.2 0.0001635393 0.3914883
## 3   0.3 0.0001736006 0.3912394
## 4   0.4 0.0001721173 0.3912394
## 5   0.5 0.0001820231 0.3912394
## 6   0.6 0.0002005199 0.3912394
## 7   0.7 0.0002272076 0.3909905
## 8   0.8 0.0002181900 0.3909905
## 9   0.9 0.0002336095 0.3904928
```

Nous affichons ensuite un graphique avec les erreurs calculées et les alpha.

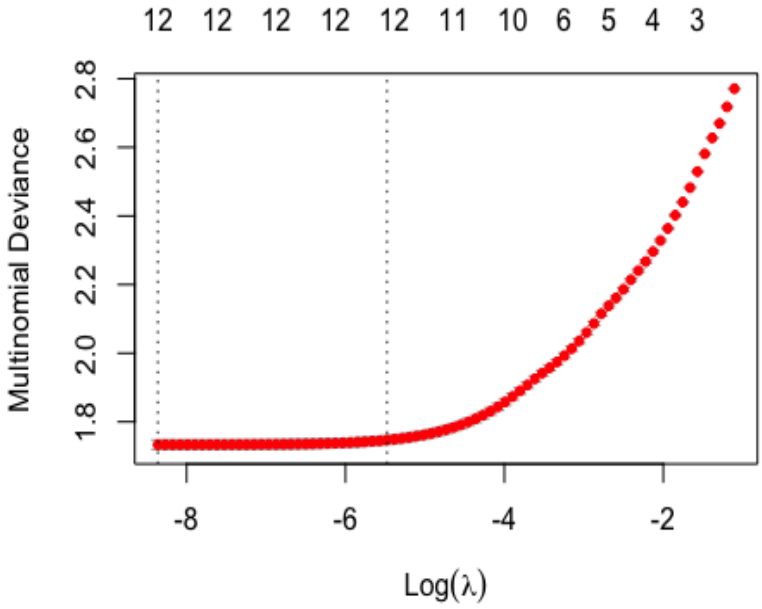
```
plot(res$alpha,res$erreur,type="b")
```



La meilleure valeur de alpha estimée en validation croisée est de 0.9.

Validation croisée avec alpha minimisant l'erreur :

```
cv.en2 <-
cv.glmnet(x=as.matrix(X_train),y=as.matrix(y_train),family="multinomial",type
.multinomial="grouped",
nfolds=10,alpha=res[res$erreur==min(res$erreur),'alpha'], standardize = T)
plot(cv.en2)
```



```
pred.en = predict(cv.en2,as.matrix(X_test),s=c(min(res$lambda)),type="class")
print(table(y_test,pred.en))
```

```
##       pred.en
## y_test   A   B   C   D
##      A 758 264  20   0
##      B 230 433 273  29
##      C  85 219 495 199
##      D  13  53 184 763
```

```
erreur_elasticnet = (sum(y_test != pred.en)/length(y_test))
print(erreur_elasticnet)
```

```
## [1] 0.3904928
```

Le modèle de régression elasticnet donne un taux d'erreur de prédiction de 39%. Nous avons tenter d'optimiser alpha et lambda afin d'obtenir le taux d'erreur le plus faible en validation croisée.

3.2. Réseau de neurones avec une couche cachée

Un réseau de neurones est une succession de neurones caractérisé par le nombre de couche dans le réseau, le nombre de neurones dans chaque couche, les fonctions d'activations utilisées, le type de tâche à réaliser (régression, classification). Le premier neurone reçoit les données "brutes". Ensuite, chaque neurone réalise une combinaison linéaire des entrées qu'il a reçu, à laquelle un biais est ajouté. La valeur de sortie du neurone est calculée grâce à une fonction dite fonction d'activation. Puis cette valeur est transmise au neurone de la couche suivante, etc jusqu'à la couche de sortie. Lors de la phase d'entraînement d'un réseau de neurones, l'algorithme va déterminer les coefficients synaptiques permettant de minimiser l'erreur sur les données d'apprentissage.

Première approche : librairie nnet

Dans un premier temps, nous allons utiliser la librairie **nnet**. Elle est utilisée pour les perceptrons avec une seule couche cachée.

Essayons avec 2 neurones (*size = 2*) dans l'unique couche cachée (*skip = F*).

```
set.seed(123)
nn_nnet = nnet(as.factor(class) ~ ., data = training_data_scale, skip = F,
size = 2)
```

```
## # weights:  36
## initial  value 14304.401086
## iter  10 value 9456.731460
## iter  20 value 8871.620843
## iter  30 value 8582.368936
## iter  40 value 8295.229203
```

```
## iter  50 value 8048.431415
## iter  60 value 7880.425896
## iter  70 value 7806.946047
## iter  80 value 7788.301699
## iter  90 value 7784.091731
## iter 100 value 7740.327863
## final  value 7740.327863
## stopped after 100 iterations
```

On calcule ensuite les prédictions permettant l'évaluation du modèle.

```
pred_nn_nnet = predict(nn_nnet,newdata=X_test_scale,type="class")
```

Nous calculons la matrice de confusion :

```
mc_nn_nnet = table(testing_data_scale$class,pred_nn_nnet)
print(mc_nn_nnet)

##    pred_nn_nnet
##       A   B   C   D
##   A 875 162   5   0
##   B 292 475 191   7
##   C 108 253 482 155
##   D  16  52 223 722
```

Puis le taux d'erreur :

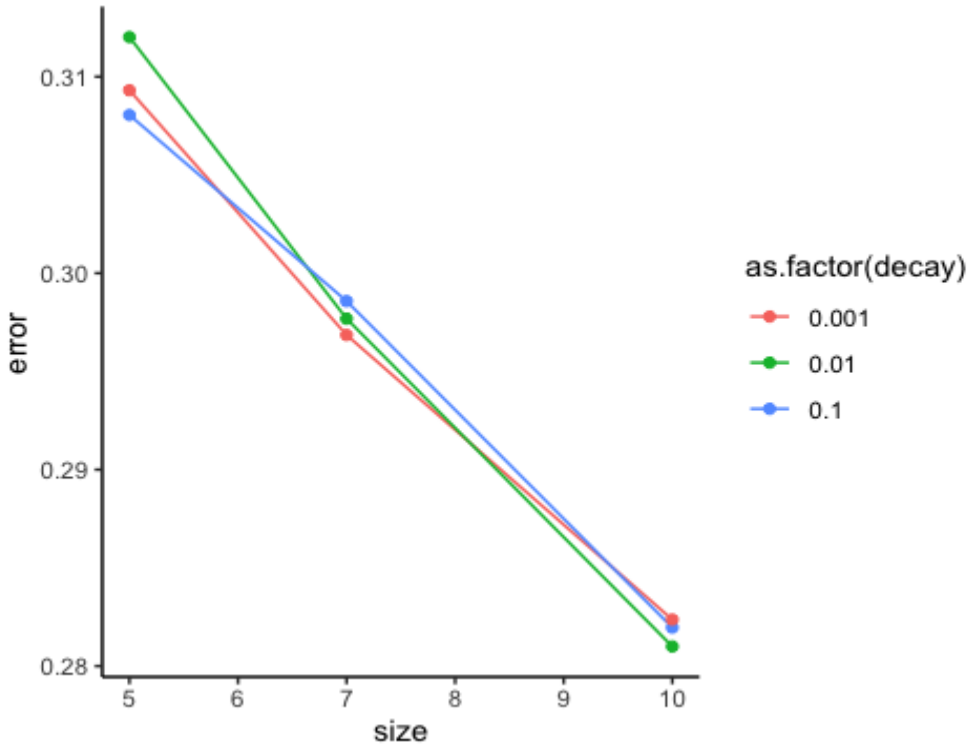
```
err_nn_nnet = 1-sum(diag(mc_nn_nnet))/sum(mc_nn_nnet)
print(err_nn_nnet)

## [1] 0.3643604
```

Avec les paramètres par défaut pour notre première tentative, le taux d'erreur est assez élevé. Nous allons tenter de "jouer" avec les paramètres pour construire un modèle adapté aux données d'apprentissage mais également performant sur la prédiction des nouvelles observations. Les paramètres sur lesquels nous pouvons travailler sont *size*, qui correspond au nombre de neurones dans la couche cachée et *decay*, le pas d'apprentissage. Pour faire cela, nous allons utiliser la fonction **tune** du package **e1071**.

```
#install.packages("e1071")
library(e1071)
set.seed(123)
nn_nnet_tune = tune.nnet(as.factor(class) ~ ., data = training_data_scale,
skip = F, size = c(5,7,10), decay = c (0.1, 0.01, 0.001))
```

Nous pouvons maintenant regarder les résultats avec les différents paramètres utilisés.

```
ggplot(nn_nnet_tune$performances, aes(x=size, y=error, group =
as.factor(decay))) + geom_line(aes(color=as.factor(decay))) +
geom_point(aes(color=as.factor(decay))) + theme_classic()
```



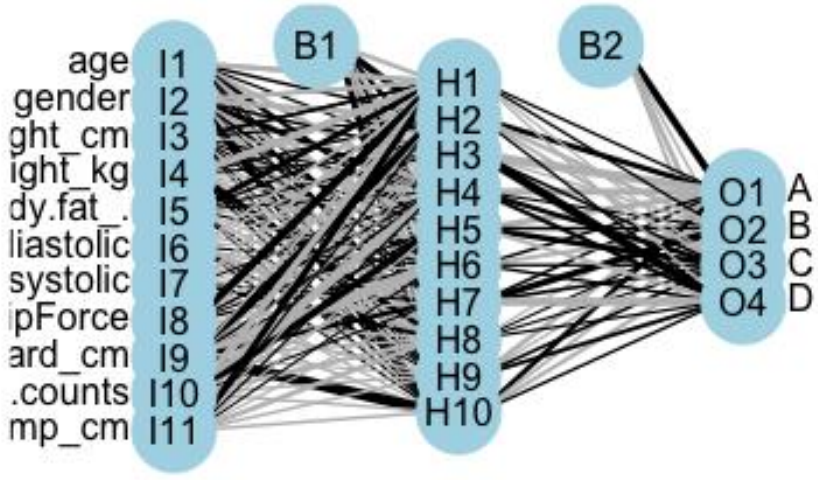
Vis à vis de chaque ensemble de paramètres utilisés, on peut mettre en évidence que plus le nombre de neurones dans la couche cachée est grand, plus le taux d'erreur baisse, en revanche pour le pas d'apprentissage il est plus difficile de démarquer les différentes valeurs testées les unes des autres.

Nous affichons alors les paramètres du meilleur modèle estimé en validation croisée, que nous pouvons également lire sur le graphique précédent.

```
nn_nnet_tune

##
## Parameter tuning of 'nnet':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  size decay
##    10  0.01
##
## - best performance: 0.2809999
```

Il semblerait que le modèle avec 10 neurones dans la couche cachée et un pas d'apprentissage de 0.01 soit le meilleur. Nous allons tout d'abord ré-estimé le modèle supposé optimal pour pouvoir en avoir une sortie graphique.

```
nn_nnet_best = nnet(as.factor(class) ~ ., data = training_data_scale, skip =
F, size = 10, decay = 0.01 )

## # weights:  164
## initial  value 17417.400439
## iter  10 value 8189.106532
## iter  20 value 7390.643719
## iter  30 value 6763.485267
## iter  40 value 6488.163075
## iter  50 value 6363.580943
## iter  60 value 6229.396102
## iter  70 value 6137.741510
## iter  80 value 6054.087866
## iter  90 value 6001.417381
## iter 100 value 5974.785992
## final  value 5974.785992
## stopped after 100 iterations



#install.packages("gamlss.add")
library(gamlss.add)
plot(nn_nnet_best)
```



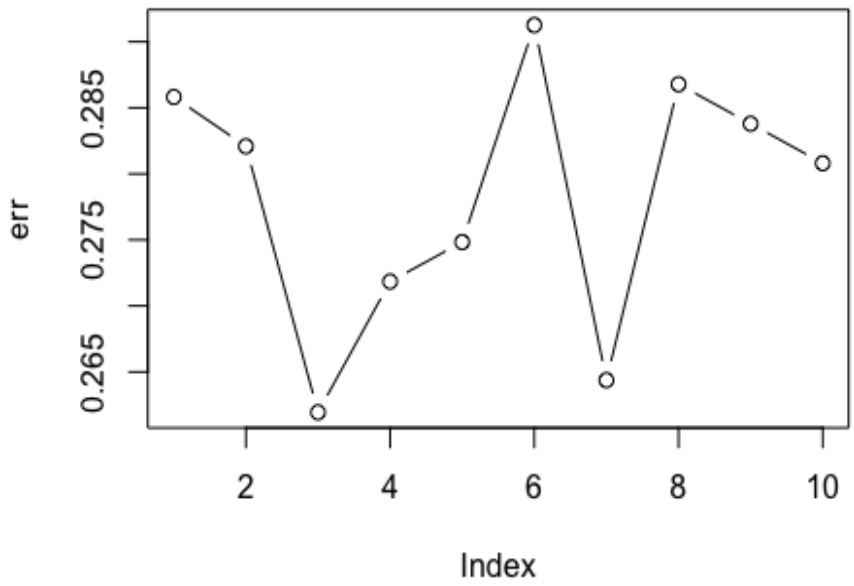
Egalement , c'est celui que nous allons retenir pour la validation croisée.

```
set.seed(123)
nb_folds = 10
n = nrow(D)
err = rep(0,nb_folds)
folds_obs = sample(rep(1:nb_folds,length.out=n))
for (k in 1:nb_folds)
{
  print(paste("===== Fold :",k))

  test = which(folds_obs==k)
  test_cv = D[test,]

  train = setdiff(1:n,test)
  train_cv = D[train,]

  X_train_cv = train_cv[,-12]
  X_test_cv = test_cv[,-12]

  y_train_cv = train_cv[,12]
  y_test_cv = test_cv[,12]

  mean_train_cv = sapply(X_train_cv,mean)
  sd_train_cv <- sapply(X_train_cv,sd)

  X_train_cv_scale = data.frame(scale(X_train_cv, center = mean_train_cv,
scale = sd_train_cv))
  X_test_cv_scale = data.frame(scale(X_test_cv, center = mean_train_cv, scale
= sd_train_cv))

  training_cv_scale = cbind(X_train_cv_scale, class = y_train_cv)
  testing_cv_scale = cbind(X_test_cv_scale, class = y_test_cv)

  cv_nnet=nnet(as.factor(class) ~ ., data = training_cv_scale, skip = F, size
= 10, decay = 0.01)

  pred_cv_nnet = predict(cv_nnet,newdata=testing_cv_scale,type="class")

  cm_cv = table(testing_cv_scale$class,pred_cv_nnet)

  e=1-sum(diag(cm_cv))/sum(cm_cv)

  err[k]=e
}
```

Ici, nous n'avons pas afficher la sortie, celle-ci étant relativement longue.

Nous pouvons également afficher le taux d'erreur pour chaque fold.

```
plot(err,type="b")
```



Et pour terminer, la validation croisée nous permet d'évaluer les performances de notre modèle.

```
erreur_nn = mean(err)
print(erreur_nn)

## [1] 0.2796979
```

Ainsi, en moyenne nous avons un taux d'erreur de 0.27.

Deuxième approche : librairie neuralnet

Pour cette deuxième approche, nous utilisons la librairie **neuralnet**. Celle-ci permet de réaliser tout types de réseaux de neurones. Nous nous limitons à une seule couche cachée pour commencer.

```
library(neuralnet)
library(fastDummies)
```

Pour l'utilisation de la fonction **neuralnet**, nous devons tout d'abord créer une variable contenant explicitement la formule passée en paramètres de la fonction *neuralnet*, celle-ci ne pouvant gérer les formules abrégées (avec le point).

```
nom_var_exp = names(training_data_scale)[1:11]
nom_clas_cib = names(training_data_scale)[12]
var_exp=paste(nom_var_exp,collapse = "+")
clas_cib=paste(nom_clas_cib,collapse = "+")
mod=paste(clas_cib,"~",var_exp,sep="")
mod=as.formula(mod)
```

Par ailleurs, nous devons également transformer la variable cible en facteur.

```
training_data_scale$class = as.factor(training_data_scale$class)
```

Ensuite, nous lançons l'apprentissage. De nombreux paramètres sont à prendre en compte : * *hidden*, pour le nombre de neurones dans la couche caché * *threshold*, le critère d'arrêt des dérivées partielles de la fonction d'erreur, par défaut à 0.01 * *algorithm*, le choix de l'algorithme, par défaut il s'agit de la rétropropagation résiliente avec retour de poids * *err.fct*, la fonction pour le calcul de l'erreur (*ce* pour entropie croisée et *sse* pour la somme des erreurs au carré) * *linear.output*, afin de réaliser ou non une regression linéaire. Par défaut à *TRUE*, mais nous le fixons à *FALSE* puisque nous sommes dans un problème de classification.

```
set.seed(123)
nn_neuralnet = neuralnet(formula = mod, data = training_data_scale, hidden =
5, err.fct = "sse", linear.output = FALSE, threshold = 0.1)
```

On calcule ensuite les probabilités d'affectation sur l'échantillon test.

```
prob_nn_neuralnet = compute(nn_neuralnet, X_test_scale)
```

Il nous faut ensuite déterminer la classe pour laquelle la probabilité d'affectation est la plus élevée pour chaque nouvelle observation. Et nous retrouvons la vraie classe de la même manière.

```
pred_clas_nn_neuralnet = apply(prob_nn_neuralnet$net.result,1,which.max)
true_clas_nn_neuralnet = as.numeric(as.factor(testing_data_scale$class))
```

Nous calculons la matrice de confusion :

```
mc_nn_neuralnet = table(true_clas_nn_neuralnet,pred_clas_nn_neuralnet)
print(mc_nn_neuralnet)

##                       pred_clas_nn_neuralnet
## true_clas_nn_neuralnet   1   2   3   4
##                      1 899 143   0   0
##                      2 242 615  93  15
##                      3  86 254 602  56
##                      4  14  96 131 772
```

Puis le taux d'erreur :

```
err_nn_neuralnet = 1-sum(diag(mc_nn_neuralnet))/sum(mc_nn_neuralnet)
print(err_nn_neuralnet)

## [1] 0.2812344
```

A nouveau, avec la librairie **neuralnet** cette fois-ci, notre première tentative avec les paramètres par défaut donne un taux d'erreur assez élevé.

Nous allons essayer de tester avec des combinaisons de paramètres différentes pour améliorer notre modèle. Il faut dans un premier temps définir la grille de recherche pour les paramètres. Etant donné le nombre de paramètres relativement important dans la fonction **neuralnet**, nous avons sélectionné ceux qui nous paraissent les plus importants, à savoir le nombre de neurones dans la couche cachée et le critère d'arrêt.

```
tune.grid.neuralnet = expand.grid(hidden = c(2,5,7,10), threshold =
c(0.1,0.01))
```

Puis nous effectuons la recherche des paramètres optimaux. Cependant, nous n'avons pas pu du fait de la non convergence de l'algorithme malgré différents essais.

3.3. SVM

Présentation de la méthode :

La méthode SVM signifie Machine à vecteurs de support ou séparateur à vaste marge en français. Ce sont des algorithmes d'apprentissage supervisé qui permettent de résoudre des problèmes de classification, regression ou de détection d'anomalies. Les SVM sont des généralisations des classifieurs linéaires. Le principe des SVM est simple, ils ont pour but de séparer les données en classes à l'aide d'une frontière aussi simple que possible, de telle façon que la distance entre les différents groupes de données et la frontière qui les sépare soit maximale.

Présentation de la librairie utilisée :

La librairie utilisée est la librairie "e1071". Elle permet l'analyse de différentes méthodes comme le clustering flou, les SVM, le calcul du plus court chemin, les k-plus proche voisins… Pour les SVM, cette librairie nous permet d'entraîner notre modèle avec la fonction svm(), de rechercher les paramètres optimaux avec la fonction tune.svm() et de faire la validation croisée avec svm() aussi.

```
#Chargement du package
library(e1071)
```

Après avoir chargé la librairie, la première étape était de savoir quel est le noyau le plus adapté pour nos données pour le modèle SVM. Pour cela, nous avons entraîné un modèle différent pour chaque noyau sur nos données et nous avons comparé les résultats.

```
#Recherche du noyau idéal
set.seed(123)
training_data_scale$class = as.factor(training_data_scale$class)

SVM_lineaire = svm(class~.,
data=training_data_scale,kernel="linear",scale=F,type="C-classification")
SVM_poly = svm(class~.,
data=training_data_scale,kernel="polynomial",scale=F,type="C-classification")
SVM_radial = svm(class~., data=training_data_scale,
kernel="radial",scale=F,type="C-classification")
SVM_sig = svm(class~., data=training_data_scale,
kernel="sigmoid",scale=F,type="C-classification")

summary(SVM_lineaire) #7161 vecteurs de supports

##
## Call:
## svm(formula = class ~ ., data = training_data_scale, kernel = "linear",
##     type = "C-classification", scale = F)
##
##
## Parameters:
```

```
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1
##
## Number of Support Vectors:  7161
##
##  ( 2267 1134 2382 1378 )
##
##
## Number of Classes:  4
##
## Levels:
##  A B C D
```

```
summary(SVM_poly)     #7267 vecteurs de supports
```

```
##
## Call:
## svm(formula = class ~ ., data = training_data_scale, kernel =
"polynomial",
##     type = "C-classification", scale = F)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  polynomial
##        cost:  1
##      degree:  3
##      coef.0:  0
##
## Number of Support Vectors:  7267
##
##  ( 2227 1176 2360 1504 )
##
##
## Number of Classes:  4
##
## Levels:
##  A B C D
```

```
summary(SVM_radial)   #6946 vecteurs de supports
```

```
##
## Call:
## svm(formula = class ~ ., data = training_data_scale, kernel = "radial",
##     type = "C-classification", scale = F)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
```

```
##        cost:  1
##
## Number of Support Vectors:  6946
##
##  ( 2157 1065 2347 1377 )
##
##
## Number of Classes:  4
##
## Levels:
##  A B C D

summary(SVM_sig)      #5766 vecteurs de supports

##
## Call:
## svm(formula = class ~ ., data = training_data_scale, kernel = "sigmoid",
##     type = "C-classification", scale = F)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  sigmoid
##        cost:  1
##      coef.0:  0
##
## Number of Support Vectors:  5766
##
##  ( 1828 873 1942 1123 )
##
##
## Number of Classes:  4
##
## Levels:
##  A B C D
```

La fonction svm() permet d'entraîner notre modèle sur nos données d'apprentissage. Les paramètres de cette fonction sont nombreux mais nous avons uniquement utilisé ceux décrits ci-dessous: **formula** : la formule à estimer avec la variable cible et les variables explicatives. **data** : le jeu de données utilisé. **kernel** : le noyau utilisé. **scale** : si on standardise les variables ou non. **type** : le type de SVM utilisé (classification ou régression).

Ici, nous réalisons une classification pour prédire la variable "class" en fonction de toutes les autres. Nous utilisons donc le type "C-classification" et les données d'entrainement.

En regardant les résultats de nos quatre modèles, nous pouvons voir qu'ils estiment un nombre de vecteurs supports différents.Les « vecteurs de support » sont les points les plus proches de la frontière. Le modèle linéaire estime le plus grand nombre de vecteurs de supports et le modèle sigmoïd le moins. Le nombre moyen de points supports est d'environ 6500 sur les 13 393 données ce qui est quand-même assez important.

Pour trouver le meilleur noyau pour nos données, nous avons appliqué chaque modèle sur nos données test pour estimer l'erreur de prédiction et les comparer.

```
# NOYAU LINEAIRE

# prédiction sur l'ensemble test
predict_lineaire=predict(SVM_lineaire,newdata=testing_data_scale,type="class"
)

# Matrice de confusion
matrice_confusion_lineaire = table(predict_lineaire, y_test) ;
matrice_confusion_lineaire
```

```
##                  y_test
## predict_lineaire   A   B   C   D
##                A 784 235  90  13
##                B 245 480 260  66
##                C  13 223 490 196
##                D   0  27 158 738
```

```
#Taux d'erreur
err_lineaire = 1-
sum(diag(matrice_confusion_lineaire))/sum(matrice_confusion_lineaire) ;
err_lineaire
```

```
## [1] 0.3797909
```

```
# Avec le noyau linéaire et les paramètres initiaux, nous avons un taux
d'erreur de 37,97%.



# NOYAU POLYNOMIAL

# prédiction sur l'ensemble test
predict_poly=predict(SVM_poly,newdata=testing_data_scale,type="class")

# Matrice de confusion
matrice_confusion_poly = table(predict_poly, y_test) ; matrice_confusion_poly
```

```
##             y_test
## predict_poly   A   B   C   D
##            A 754 211  79  10
##            B 266 456 245  58
##            C  22 280 634 259
##            D   0  18  40 686
```

```
#Taux d'erreur
err_poly = 1-sum(diag(matrice_confusion_poly))/sum(matrice_confusion_poly) ;
err_poly
```

```
## [1] 0.3703335

# Avec le noyau polynomial et les paramètres initiaux, nous avons un taux
d'erreur de 37.03%.



# NOYAU RADIAL

# prédiction sur l'ensemble test
predict_radial=predict(SVM_radial,newdata=testing_data_scale,type="class")

# Matrice de confusion
matrice_confusion_radial = table(predict_radial, y_test) ;
matrice_confusion_radial

##                y_test
## predict_radial   A   B   C   D
##              A 831 243  75   9
##              B 204 535 234  63
##              C   6 164 632 172
##              D   1  23  57 769

#Taux d'erreur
err_radial = 1-
sum(diag(matrice_confusion_radial))/sum(matrice_confusion_radial) ;
err_radial

## [1] 0.3113489

# Avec le noyau radial et les paramètres initiaux, nous avons un taux
d'erreur de 31.13%.



# NOYAU SIGMOID

# prédiction sur l'ensemble test
predict_sig=predict(SVM_sig,newdata=testing_data_scale,type="class")

# Matrice de confusion
matrice_confusion_sig = table(predict_sig, y_test) ; matrice_confusion_sig

##             y_test
## predict_sig   A   B   C   D
##           A 570 337 222 114
##           B 315 200 104  20
##           C  39 159 312 265
##           D 118 269 360 614
```

```
#Taux d'erreur
err_sig = 1-sum(diag(matrice_confusion_sig))/sum(matrice_confusion_sig) ;
err_sig

## [1] 0.5778995

# Avec le noyau sigmoid et les paramètres initiaux, nous avons un taux
d'erreur de 57.78%.
```

Concernant les paramètres de la fonction "predict", nous pouvons voir : **object** : le modèle d'apprentissage, ici, les différents modèles précédents avec les différents noyaux. **newdata** : les données pour la prédiction, ici, les données de test. **type** : le type de prédiction, ici, les classes d'appartenance avec les différentes classes "A","B","C" ou "D".

Nous avons utilisé les paramètres par défaut pour chaque noyau. Nous pouvons voir que le taux d'erreur pour le noyau linéaire est d'environ 37%, pour le noyau polynomial 37%, pour le noyau radial 31% et pour le noyau sigmoïd 57%. Donc, pour nos données, le noyau le plus adapté pour les SVM est le noyau radial. Le noyau radial est défini ainsi : exp(-gamma*|u-v|^2)).

Le noyau qui nous donne des meilleures prédictions est le noyau radial, cependant, avec les paramètres par défaut, nous obtenons un taux d'erreur de prédiction de 31% ce qui est important. Nous devons donc tester différentes valeurs pour chaque paramètre de la fonction afin de trouver les meilleurs paramètres. Pour les SVM, les paramètres les plus importants sont C et Gamma. **C** correspond au paramètre de régularisation soit la taille de la marge du SVM. Les points situés à l'intérieur de cette marge ne sont classés dans aucune classe. La force de la régularisation est inversement proportionnelle à C donc quand la valeur de C augmente, la marge diminue et quand elle diminue, la marge augmente. **gamma** contrôle la distance de l'influence d'un seul point d'entraînement dans le cas du noyau radial. De faibles valeurs de gamma indiquent un grand rayon de similarité qui se traduit par le regroupement de plus de points. Pour des valeurs élevées de gamma, les points doivent être très proches les uns des autres afin d'être considérés dans le même groupe (ou classe). Par conséquent, les modèles avec des valeurs gamma très élevées ont tendance à se surajuster.

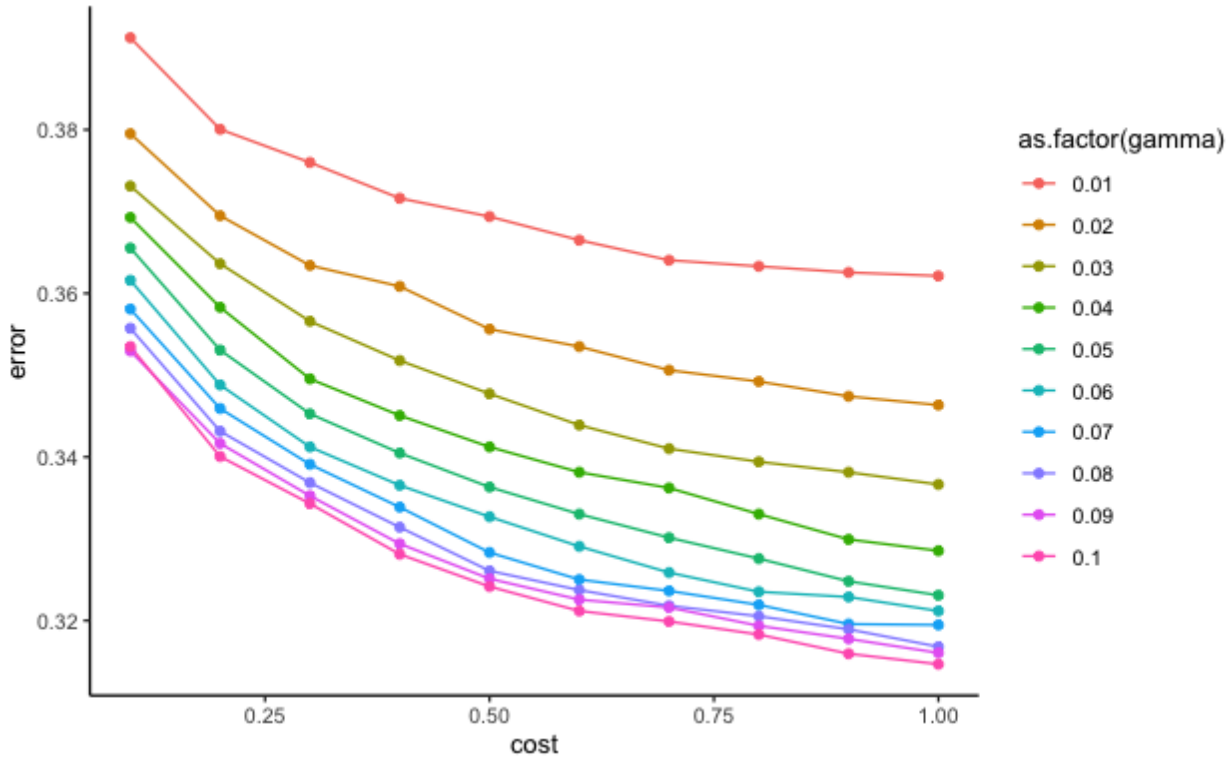
Pour cela, nous utilisons la fonction tune.svm(). Les différents paramètres de cette fonction que nous utilisons et qui n'ont pas été présentés précédemment sont : **gamma** : paramètre gamma présenté précédemment. **cost** : paramètre C présenté précédemment.

```
set.seed(123)
param_opti = tune.svm(class~., data=training_data_scale, kernel='radial',
scale=F, type = "C-classification", gamma = seq(0.01, 0.1, by = 0.01), cost =
seq(0.1,1, by = 0.1))

param_opti$best.model

##
## Call:
## best.svm(x = class ~ ., data = training_data_scale, gamma = seq(0.01,
##     0.1, by = 0.01), cost = seq(0.1, 1, by = 0.1), kernel = "radial",
```

```
##     scale = F, type = "C-classification")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  6921

param_opti$best.parameters

##    gamma cost
## 16   0.1    1
```

Nous constatons que les meilleurs paramètres pour ce modèle sont gamma = 0.1 et cost =1.

```
ggplot(param_opti$performances, aes(x=cost, y=error, group =
as.factor(gamma))) + geom_line(aes(color=as.factor(gamma))) +
geom_point(aes(color=as.factor(gamma))) + theme_classic()
```



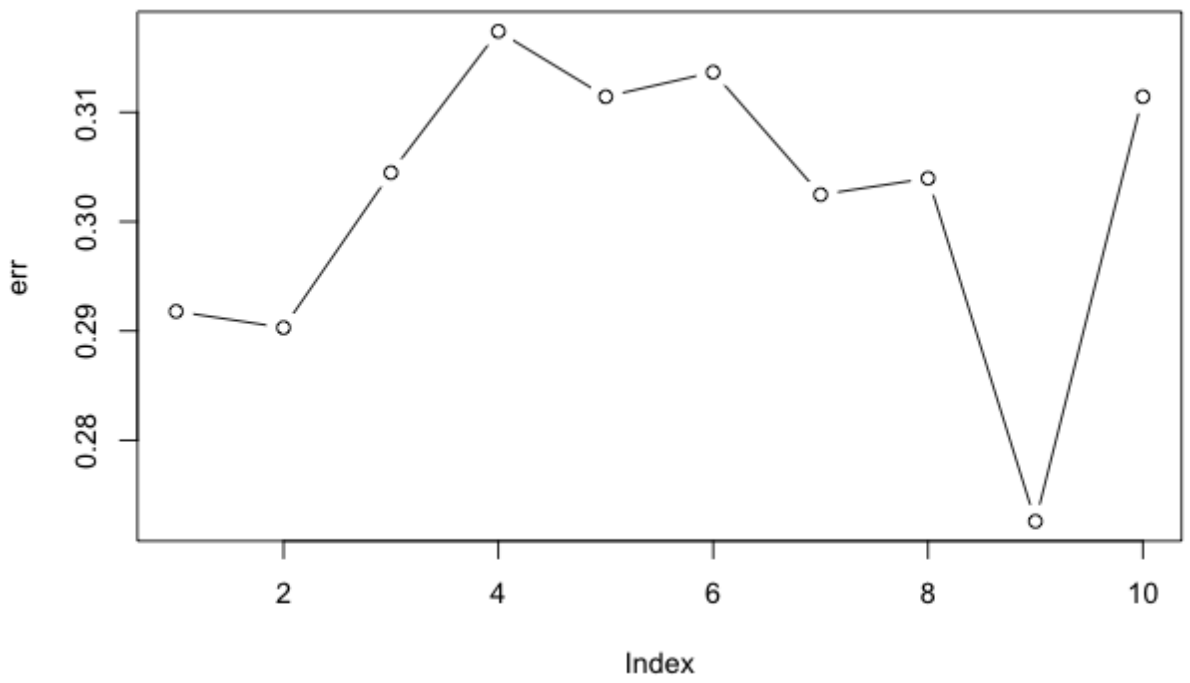
Sur le graphique, nous pouvons voir que plus gamma et cost augmentent et plus le taux d'erreur diminue.

Pour finir, nous avons réalisé une validation croisée afin d'obtenir le score moyen de performance de notre modèle.

```
set.seed(123)
nb_folds = 10
n = nrow(D)
err = rep(0,nb_folds)
folds_obs = sample(rep(1:nb_folds,length.out=n))
for (k in 1:nb_folds)
{
  print(paste("===== Fold :",k))

  test = which(folds_obs==k)
  test_cv = D[test,]

  train = setdiff(1:n,test)
  train_cv = D[train,]

  X_train_cv = train_cv[,-12]
  X_test_cv = test_cv[,-12]

  y_train_cv = train_cv[,12]
  y_test_cv = test_cv[,12]

  mean_train_cv = sapply(X_train_cv,mean)
  sd_train_cv <- sapply(X_train_cv,sd)

  X_train_cv_scale = data.frame(scale(X_train_cv, center = mean_train_cv,
scale = sd_train_cv))
  X_test_cv_scale = data.frame(scale(X_test_cv, center = mean_train_cv, scale
= sd_train_cv))

  training_cv_scale = cbind(X_train_cv_scale, class = y_train_cv)
  testing_cv_scale = cbind(X_test_cv_scale, class = y_test_cv)

  cv_svm = svm(as.factor(class)~., data=training_cv_scale,
kernel="radial",scale=F,type="C-classification",
               gamma = 0.1, cost =1)

  pred_cv_nnet = predict(cv_svm,newdata=testing_cv_scale,type="class")

  cm_cv = table(testing_cv_scale$class,pred_cv_nnet)

  e=1-sum(diag(cm_cv))/sum(cm_cv)

  err[k]=e
}
```

Nous pouvons réaliser un graphique qui montre l'évolution de l'erreur.

```
plot(err,type="b")
```



Pour connaître la performance réelle de notre modèle, il faut faire la moyenne de l'erreur des 10 validations croisées. En effet, ce taux d'erreur est plus robuste.

```
erreur_svm = mean(err)
print(erreur_svm)

## [1] 0.3019502
```

Nous pouvons donc voir que notre modèle d'SVM avec un noyau radial, gamma = 0.1 et cost = 1 donne un taux d'erreur de prédiction d'environ 30%. Avec les paramètres par défaut d'un noyau radial, nous avions un taux d'erreur de 30% donc nous pouvons en déduire que l'optimisation des paramètres n'a pas fait réellement baisser le taux d'erreur. Ce taux d'erreur est quand même assez important. Pour aller plus loin, on aurait pu tester plus de valeurs pour les paramètres gamma et cost. Cependant, le temps d'éxécution de la fonction tune() était très important et plus on augmente les paramètres à tester et plus c'est long.

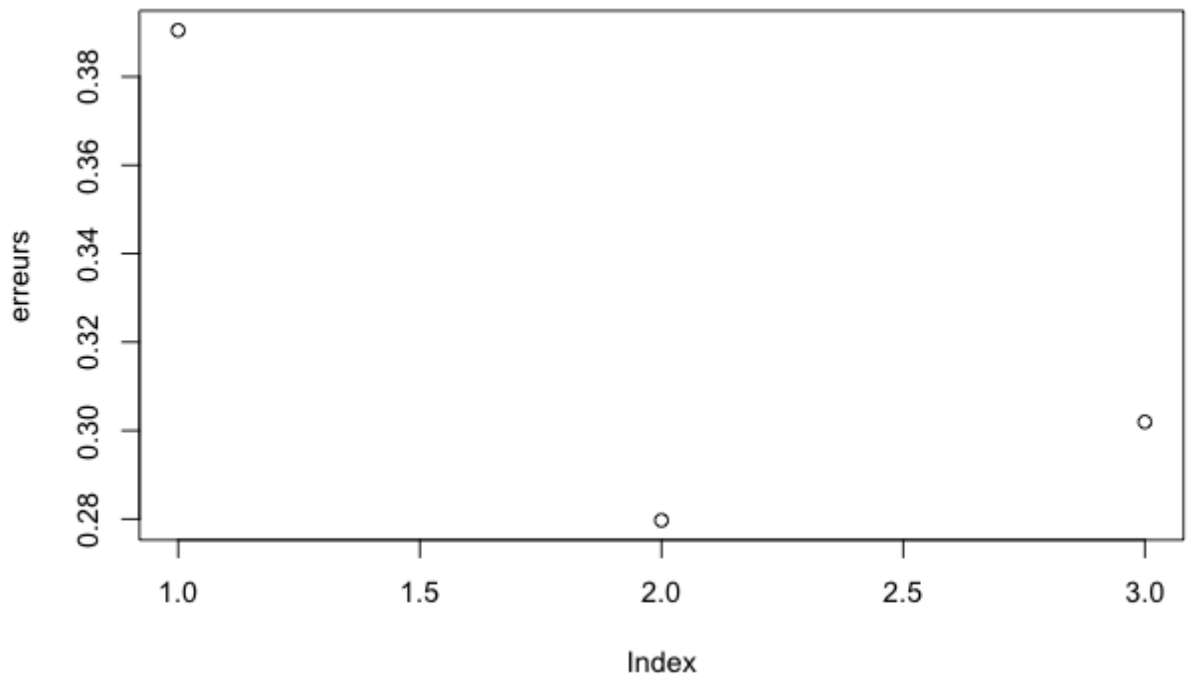
4. Comparaison des meilleurs modèles pour chaque type de méthode

Ainsi, pour chaque type de modèle, nous avons identifié le meilleur modèle en réalisant un tuning des hyperparamètres. Puis nous avons évalué le taux d'erreur de ce meilleur modèle en validation croisée.

```
erreurs = c(erreur_elasticnet, erreur_nn, erreur_svm)
```

Nous pouvons alors affiché les résultats obtenus pour chaque modèle.

```
plot(erreurs)
```



Le premier modèle, qui correspond au modèle linéaire pénalisé par une fonction de régularisation elasticnet est celui qui donne le moins bon résultat. Pour ce modèle, nous avons tenté d'optimiser en même temps alpha et lambda pour obtenir le plus petit taux d'erreur en validation croisée. Or, plus on augmente le nombre de paramètres à optimiser, plus le processus devient spécifique aux données, et on risque le problème de surapprentissage. Concernant le réseau de neurones et le SVM, le taux d'erreur est plus faible, et quasi comparable entre les deux méthodes. Il pourrait donc s'agir des modèles à privilégier pour répondre à ce problème de classification de la performance des individus.

Conclusion

Ce projet nous a permis de mettre en pratique les méthodes de classification vues en cours. De plus, nous avons pu réaliser un projet de A à Z c'est-à-dire le choix du jeu de données, les pré-traitements nécessaires, l'application des différentes méthodes de classification et la comparaison des modèles afin de choisir celui qui correspondait au mieux à nos données. En termes de perspectives, avec plus de temps, nous aurions pu tester plus de paramètres lors de la fonction tune() afin de trouver les meilleurs. Nous aurions également pu explorer des méthodes sortant des techniques vues en cours. Le temps d'exécution plus ou moins long en fonction de la méthode exécutée pour la recherche des hyperparamètres était également une contrainte.