I - Introduction

Expliquer les principes utilisés, notamment faire plusieurs jeux d'erreur de X-Validation et d'en regarder les boxplot pour voir leur variance/moyenne.

Nous avons dans ce rapport testé plusieurs méthodes différentes mais certaines d'entre elles se sont revelées peu adaptées et nous avons choisis de ne pas les intégrer.

C'est notamment le cas des méthodes additives généralisées, les méthodes arborescentes ou les méthodes basées sur l'algorithme EM qui nous ont donné de mauvais résultats quand nous les avons essayées.

Nous ne feront pas preuve d'exhaustivité notamment à cause du fait que ce rapport fait uniquement 12 pages et qu'on a donc du se concentrer sur les méthodes qui nous ont semblé les plus intéressantes.

Nous allons utiliser les librairies suivantes tout au long du rapport :

```
options(warn=-1)
library(klaR)
```

```
## Loading required package: MASS
```

```
library(randomForest)
```

```
## randomForest 4.6-12
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
library(MASS)
library(mclust)
```

```
## Package 'mclust' version 5.4
## Type 'citation("mclust")' for citing this R package in publications.
```

```
library(e1071)
```

II - Expressions faciales

1) Notes sur les données

La première chose qu'il faudra noter sur ces données est que les prédicteurs sont les 4200 pixels de l'image et que le nombre de prédicteur est donc très grand.

Pour réduire cela nous allons supprimer tous les pixels noirs qui sont présents aux mêmes endroits dans les coins de chacuns de ces photos :

```
data <- read.table('expressions/expressions_train.txt')
p <- length(data) - 1
N <- dim(data)[1]
pixelsNoirs <- which(colMeans(data[,-(p+1)]) == rep(0,p-1))
data <- data[,-pixelsNoirs]
cat("On passe de  p = ", p , " à p =", (length(data) - 1), "\nN = ", N)
```

```
## On passe de  p =  4200  à p = 3660
## N =  108
```

On se retrouve donc avec une valeur de p toujours considérable et ce pour un nombre relativement petit d'échantillons.

On devra donc s'attendre à ce que n'importe quelle valeur avec un peu trop de paramètres se retrouve à faire très facilement de l'overfitting.

On s'attendra aussi dans de telles condition à avoir un taux d'erreur assez elevé.

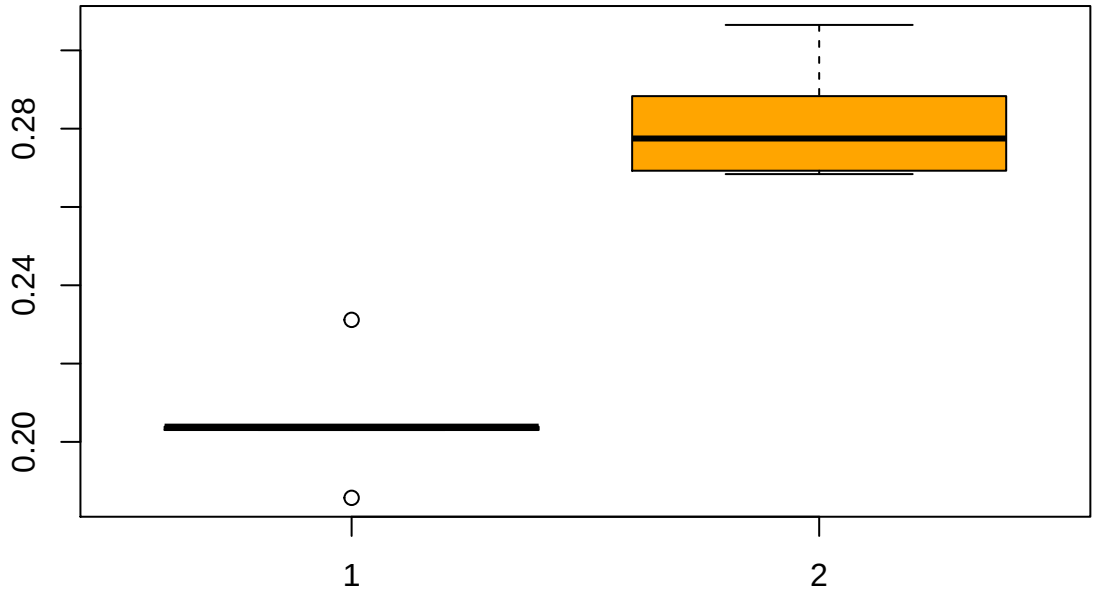
2) Méthodes classiques

2.a LDA/QDA/RDA

Nous avons voulu essayé de sélectionner le meilleur compromis entre LDA/QDA/Naive Bayes en produisant une matrice des erreurs avec une méthode RDA pour différentes valeurs de gamma/lambda.

Seulement en raison du très grand nombre de prédicteurs les calculs de QDA et RDA prennent un temps de calcul considérable et consomment tellement de mémoire qu'ils forcent la fermeture de R sur nos PC, nous avons donc décidé de tester les méthodes LDA et Naive Bayes qui possèdent de plus beaucoup moins de paramètres et sont ainsi plus aptes à donner de bons résultat.

```
load(file="expressions/discriminant.RData")
boxplot(lda.cv, naive.cv, col=c("red","orange"))
```



On observe que la lda (en rouge) est plus efficace que la méthode de Bayes naïve (en orange). A noter qu'on obtient un boxplot car on a effectué 5 différentes X-validation, chacune divisant d'une manière différente l'ensemble d'apprentissage en 5 partitions aléatoires.

Nous avons fait cela car on observait des variations d'un calcul de X-validation à l'autre selon la façon de partitionner l'ensemble d'apprentissage. Cela s'explique par la taille du jeu de donnée d'environ 100 échantillons : le choix des ~20 élements de chaque groupe peu facilement donner une valeur différente si on

les choisit différement. Nous ferons donc plusieurs X-Validations dans lesquelles les partitions seront faites de manière aléatoire et nous moyenneront les erreurs.

2.b FDA et autres méthodes

Etant donné le très grand nombre de prédicteurs contenu dans les données que l'on veut analyser, on va essayer de réduire ceux-ci à un nombre plus réduit de features par l'analyse discriminante factorielle FDA qu'on a vu en cours.

Pour se faire nous allons appeler une fonction R **lda** sur l'ensemble d'apprentissage choisi (ie **les K-1 partitions d'apprentissages** pour une cross-validation) pour ensuite utiliser la matrice **lda$scaling** et s'en servir pour multiplier la matrice des données :

```
app <- data[sample(1:N,2*N/3),]
lda <- lda(y ~ ., data=app)
p <- length(data) - 1
proj <- data.frame( t(t(lda$scaling) %*% t(data[,-(p+1)])), y=data[,(p+1)])
```

On obtient une matrice non plus de *3660 par 107* mais de *5 par 107*.

On va essayer d'appliquer une RDA sur cet ensemble de données qui associent aux features projetées la classification des différents échantillons. Nous allons tenter d'en tuner les hyperparamètres **lambda** et **gamma** pour voir si l'on peut obtenir de meilleurs résultats :

```
load("expressions/tuneRDA.RData")
expression.proj.rda.cv
```

```
##             0       0.2       0.4       0.6       0.8         1
## 0   0.3144589 0.2443290 0.2240693 0.2129870 0.2149784 0.2112554
## 0.2 0.2462338 0.2260606 0.2202597 0.2148918 0.2148918 0.2112554
## 0.4 0.2243290 0.2222511 0.2148918 0.2167965 0.2112554 0.2112554
## 0.6 0.2225108 0.2206061 0.2168831 0.2149784 0.2131602 0.2112554
## 0.8 0.2244156 0.2187879 0.2150649 0.2148918 0.2131602 0.2112554
## 1   0.2260606 0.2222511 0.2168831 0.2149784 0.2131602 0.2112554
```

On obtient les résultats pour les différentes gamma (sur les lignes) et lambda (sur les colonnes). Le nombre de paramètre dépendant du nombre de prédicteurs qui est passé de 3660 à 5 nous avons réussi à lancer un script rda pour chaque valeur qui, contrairement à ce qu'on a vait voulu faire avant la projection, fonctionnera bien et même très rapidement. Nous avons même pu moyenner les résultat de 5 différentes partitions de l'ensemble d'apprentissage donné sur 5 cross-validation différentes étant donné qu'on avait encore des variations. On observe que tant que lambda = 1 on obtient les même résultats, toujours assez satisfaisants et qu'on peut donc se contenter d'effectuer une LDA (pour laquelle lambda=1 et gamma=0) sur la projection.

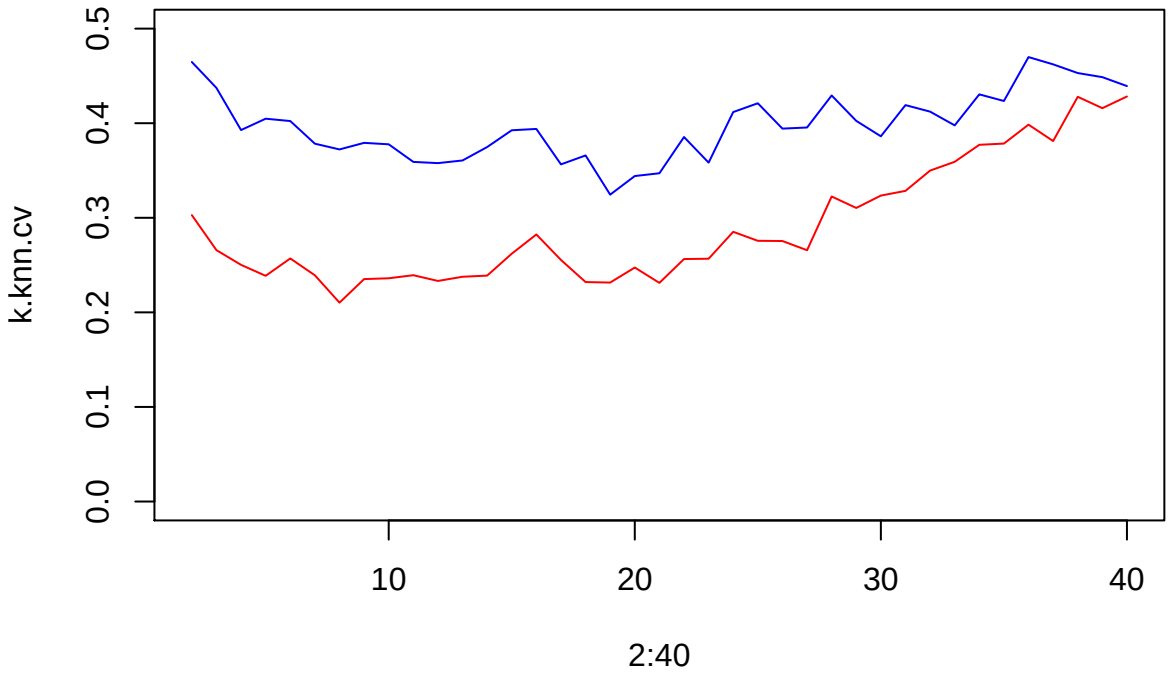
Dans les faits on observe que l'on retrouve les même résultats qu'avec la lda sur l'ensemble des prédicteurs de départ. On suppose – sans être sur – que cela est du au fait que la LDA de R utilise déjà cet espace de feature.

2.c KNN

Etant donné qu'il est très facile de faire de l'overfitting sur ces données, on a voulu essayer une méthode non-paramétrique qui peut "fit" les données de manière très raisonnable en raison de sa simplicité : la méthode des K-Nearest-Neighboors (knn).

On a essayé d'appliqué les KNN sur l'ensemble de projection **et** sur l'ensemble de départ. Affichons les résultats qu'on a obtenu pour 5 réalisations de X-Validation de l'erreur différentes pour le KNN sur la projection (en rouge) et sur l'ensemble de départ (en bleu) pour des nombres de voisins allant de 2 à 40.

```r
load(file = "expressions/knnFit.RData")
plot(2:40,k.knn.cv, col="blue", type="l", ylim=c(0,0.5))
lines(2:40, y=k.proj.knn.cv, col='red')
```

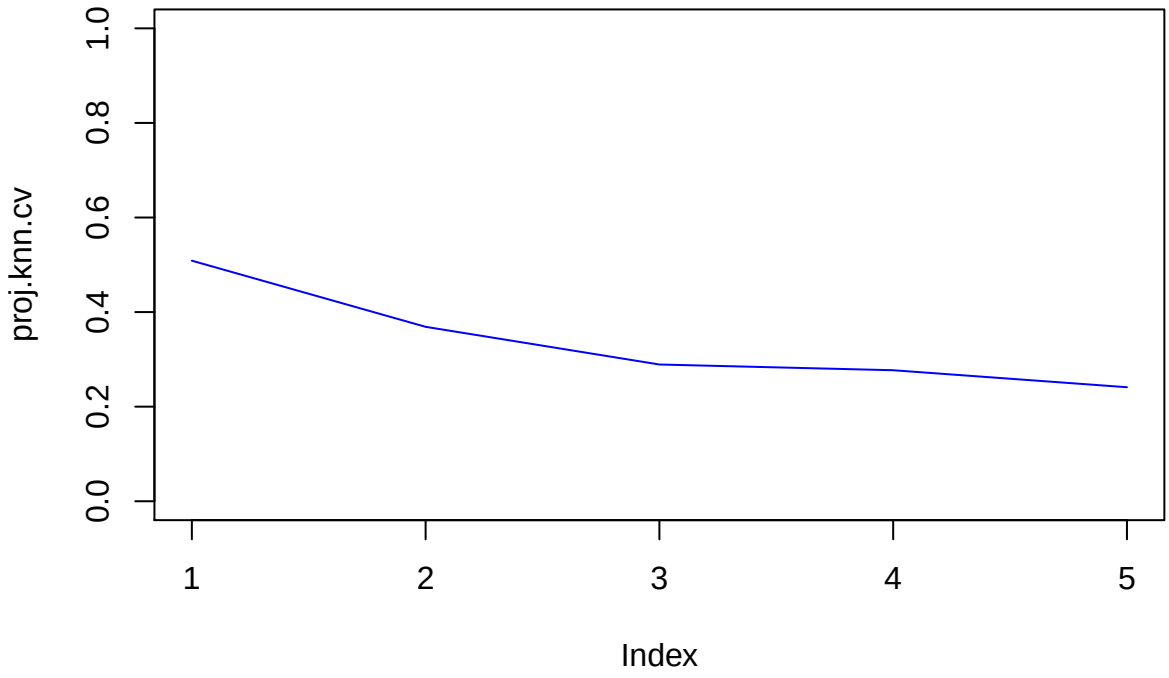


On observe que la méthode KNN est bien plus efficace sur l'ensemble de projection (surement à cause de la *"malédiction des grandes dimensions"*) et que le nombre optimal de voisins à considérer semble être autour de 8.

On s'est demandé, en raison du fait que KNN semble mieux marcher dans de plus petites dimensions si on ne pouvait pas encore plus réduire l'ensemble de projection, par défaut à 5 feature, en selectionnant les 1, 2, 3 ou 4 premières colonnes de la matrice lda$scaling (celles correspondant aux valeurs propres les plus grandes selon la documentation).

Voici l'évolution des erreurs en fonction du nombre de colonnes selectionnées :

```r
plot(proj.knn.cv, type="l", col="blue", ylim=c(0,1))
```



Il semble que KNN marche sensiblement mieux avec 5 features.

3) Méthodes avancées

Nous avons essayé différentes méthodes avancées relevant des chapitres sur les *splines* et *generalized additive models*, ainsi que les classification par *mixtures models* mais celles-ci faisant augmenter le nombre de paramètres et nous n'avons ainsi eu aucun résultat satifaisant et nous ne nous attarderons donc pas dessus. On expliquera cependant le travail effectué pour essayer de fit le modèle avec des *Support Vector Machine*

3.a SVM

Les méthodes SVM n'étant pas incluses par défaut dans R pour la classification multi-classes, nous avons créé deux méthodes pour classifier par SVM avec plusieurs classes selon les principe **ONE VS ONE** et **ONE VS ALL** :

```
oneallSVMpred <- function (app, test, kernel, degree=1, gamma=1, coef0=0, cost=1, nu=1)
{
  levels <- levels(app$y)
  K <- length(levels)
  N <- (dim(test)[1])
  votes <- matrix(0,N,6)

  for(class in 1:K)
  {
      currentEmotion <- levels(app$y)[class]
```

```
      otherEmotions <- levels(app$y)[-class]
      app.classes <- app
      app.classes[app$y != currentEmotion,length(app)] <- sample(otherEmotions, 1)
      svmfit <- svm(y~., data=app.classes, method = "C-classification", kernel=kernel,
                    degree=degree,
                    gamme=gamma, coef0=coef0, cost=cost, nu=nu)
      prediction <- predict(svmfit, newdata = test)
      currentVote <- which(prediction == currentEmotion)
      votes[currentVote,class] <- votes[currentVote,class] + 1
  }
  majorite <- levels[apply(votes,1,which.max)]
  correspondingRow <- row.names(test)
  res <- data.frame(majorite)
  rownames(res) <- row.names(test)
  return(res)
}
```

et pour one VS one

```
oneoneSVMpred <- function (app, test, kernel, degree=1, gamma=1, coef0=0, cost=1, nu=1)
{
  levels <- levels(app$y)
  K <- length(levels)
  N <- (dim(test)[1])
  votes <- matrix(0,N,6)

  for(left in 1:(K-1))
  {
    for(right in min(left+1,K):K)
    {
      # cat("\nleft is ", left, " and right is", right)

      emotionLeft <- levels(app$y)[left]
      emotionRight <- levels(app$y)[right]
      app.classes <- app[app$y==emotionLeft | app$y==emotionRight,]
      svmfit <- svm(y~., data=app.classes, method = "C-classification", kernel=kernel,
                    degree=degree,
                    gamme=gamma, coef0=coef0, cost=cost, nu=nu)
      prediction <- predict(svmfit, newdata = test)
      leftVotes <- which(prediction == emotionLeft)
      rightVotes <- which(prediction == emotionRight)
      votes[leftVotes,left] <- votes[leftVotes,left] + 1
      votes[rightVotes,right] <- votes[rightVotes,right] + 1
      # print(votes)
    }
  }
  majorite <- levels[apply(votes,1,which.max)]
  correspondingRow <- row.names(test)
  res <- data.frame(majorite)
  rownames(res) <- row.names(test)
  return(res)
}
```

Ces méthodes ont un fonctionnement qui est plutôt évident mais, programmées de façon naïves, on peut se douter qu'elles n'auront pas les meilleurs performances. On a cependant du faire avec pour ce TP. Observons

maintenant les résultats qu'on a pu avoir pour différents noyaux avec différents hyperparamètres.

3.d Tuning SVM

Voici, les résultat de ces tuning effectuées respecivement avec FDA (c'est à dire sur un espace de feature restreint) ou sans et en système de vote **one VS one** ou **one VS all** pour le kernel radial :

```
load(file=("expressions/tuningSVM.RData"))
print(fda.oo.radial.cost.gamma) # FDA/ONEvONE
```

```
##           0.001      0.01       0.1         1        10
## 0.1   0.8419913 0.8419913 0.8813853 0.8419913 0.8419913
## 1     0.8419913 0.3865801 0.2961039 0.5653680 0.7969697
## 10    0.3233766 0.2779221 0.3510823 0.5458874 0.8160173
## 100   0.3510823 0.3142857 0.4839827 0.5458874 0.8160173
## 1000  0.3047619 0.3688312 0.4839827 0.5458874 0.8160173
## 10000 0.3229437 0.3870130 0.4839827 0.5458874 0.8160173
```

```
print(fda.oa.radial.cost.gamma) # FDA/ONEvALL
```

```
##           0.001      0.01       0.1         1        10
## 0.1   0.8419913 0.8419913 0.8419913 0.8419913 0.8419913
## 1     0.8419913 0.8324675 0.3774892 0.7025974 0.8419913
## 10    0.8233766 0.2865801 0.3948052 0.6562771 0.8419913
## 100   0.2965368 0.4324675 0.4341991 0.6562771 0.8419913
## 1000  0.3051948 0.4229437 0.4341991 0.6562771 0.8419913
## 10000 0.4229437 0.3796537 0.4341991 0.6562771 0.8419913
```

```
print(oo.radial.cost.gamma) #NoFDA/ONEvONE
```

```
##         0.001      0.01       0.1
## 0.1 0.8424242 0.8424242 0.8424242
## 1   0.8424242 0.8424242 0.8424242
## 10  0.8519481 0.8424242 0.8424242
## 100 0.8519481 0.8424242 0.8424242
```

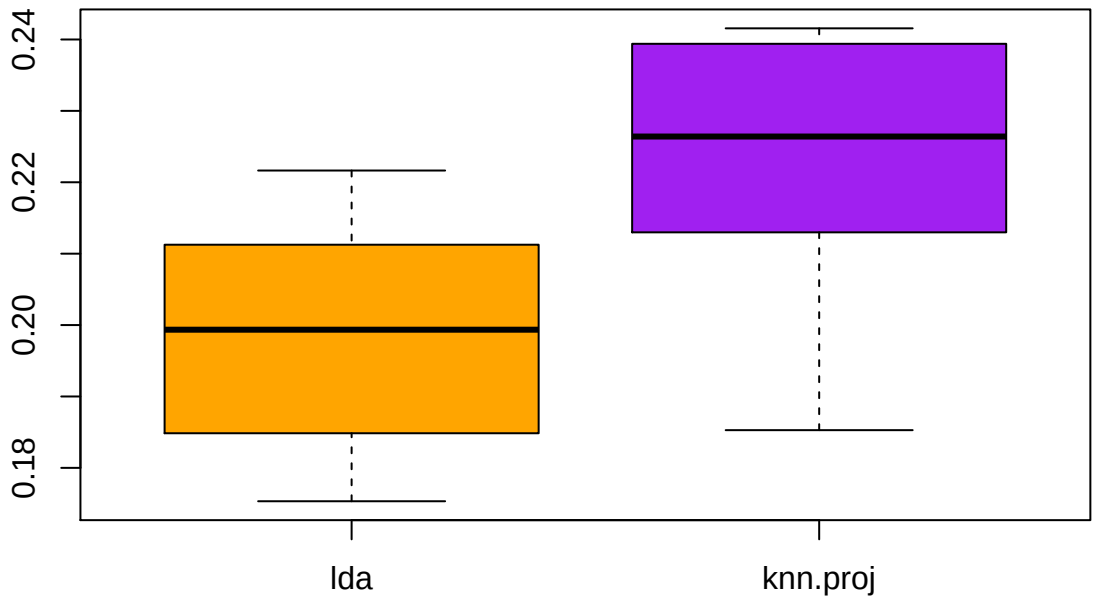
On remarque que dans chaque cas on a obtenu une performance moindres. Nous n'écrirons pas les résultats ici mais nous avons eu des résultats aussi mauvais avec les autres noyaux.

4) Conclusion

Pour ces données même si très peu de méthode proposent des taux d'erreurs faibles deux semblent sortir du lot : la **LDA** , et la méthode **KNN sur un espace réduit de feature**.

Voici donc leur résultat pour 10 réalisations du calcul de l'erreur de cross-validation :

```
load(file="expressions/res.RData")
boxplot(res, col=c("orange", "purple"))
```



Nous en concluront que lda est la meilleure méthode avec un taux d'erreur légerement inférieur à 20% .

Il est à noter que ce taux d'erreur reste très grand, mais nous sommes finalement parvenus à la conclusion que cela était propre à la façon dont étaient présentées les données.

Il aurait fallu, pour avoir de meilleurs résultats, soit posséder un nombre d'échantillons plus conséquent, soit utiliser des méthodes spécifiques au traitement d'image qui permettent d'y analyser des régularités et des formes et qui sortent du cadre de l'UV SY19.

III - Parole

```
data <- read.table('parole/parole_train.txt')
p <- length(data) - 1
N <- dim(data)[1]
K <- length(levels(data$y))
cat(p, " prédicteurs, ", N, " échantillons et ", K, " classes différentes.")
```

```
## 256  prédicteurs,  2250  échantillons et  5  classes différentes.
```

Il y a toujours un nombre assez conséquent de prédicteurs, mais beaucoup moins que pour les images et surtout il y a beaucoup plus d'échantillons.

On s'attendra donc à ce que les modèles avec beaucoup de paramètre marchent peu, mais dans une mesure moindre que pour les données précedentes.

1) Méthodes classiques

1.a RDA avec et sans FDA

Voici les erreurs de cross-validation pour différentes valeurs de gamma (ligne) et lambda (colonne) avec une RDA sur tous les prédicteurs

```
load(file = "parole/tuneRDA.RData")
parole.rda.cv
```

```
##             0        0.2        0.4        0.6        0.8          1
## 0   0.4653333 0.15688889 0.12533333 0.11333333 0.09911111 0.08311111
## 0.2 0.1075556 0.10488889 0.09955556 0.09022222 0.08044444 0.07600000
## 0.4 0.1000000 0.09511111 0.08533333 0.08044444 0.07155556 0.07422222
## 0.6 0.0920000 0.08800000 0.08222222 0.07733333 0.07200000 0.07333333
## 0.8 0.0880000 0.08400000 0.08044444 0.07777778 0.07600000 0.07866667
## 1   0.1320000 0.13155556 0.13111111 0.13200000 0.13422222 0.13511111
```

Dans laquelle le minimum est à de 7.16% pour gamma=0.4 et lambda=0.8.

Et en faisant une projection des x sur un ensemble restreint de feature – selon la méthode expliquée plus haut – les valeurs de la RDA pour ces différents hyper-paramètres deviennent :

```
parole.proj.rda.cv
```

```
##              0        0.2        0.4        0.6        0.8          1
## 0   0.08711111 0.08711111 0.08488889 0.08266667 0.08400000 0.08311111
## 0.2 0.08755556 0.08444444 0.08444444 0.08355556 0.08311111 0.08355556
## 0.4 0.08488889 0.08444444 0.08444444 0.08355556 0.08444444 0.08355556
## 0.6 0.08400000 0.08444444 0.08444444 0.08311111 0.08400000 0.08400000
## 0.8 0.08355556 0.08355556 0.08400000 0.08400000 0.08444444 0.08444444
## 1   0.08400000 0.08488889 0.08488889 0.08577778 0.08444444 0.08488889
```

On remarque immédiatement que la RDA sans sélection de features par FDA est plus efficace !

2) SVM

2.a Vue d'ensemble

Pour les méthodes de type *support vector machines* il existe quatre types de noyaux qui sont contenus dans la fonction "svm" de R : linéaire, polynomial, radial et sigmoïde. Sachant qu'on a immédiatement remarqué que les résultats de la SVM avec un noyau sigmoïdal étaient très mauvais nous approfondiront le travail qui a été fait pour trouver de bons résultat avec un **noyau polynomial** et un **noyau radial**.

En effet le noyau linéaire peu – avec les bons hyperparamètres – se rapprocher d'un noyau polynomial.

2.b Polynomiale

Il est enfait assez compliquer de tuner exhaustivement les paramètres de la SVM polynomiale pour que celle-ci soit optimale étant donné qu'il existe 4 hyperparamètres à évaluer : gamma, coef0, degree et cost.

On va donc pour trouver la SVM polynomiale optimale faire preuve d'astuce et produire des matrices des erreurs sur les combinaisons d'hyperparamètres pris deux à deux en commençant par ceux qui nous paraissent le plus importants **le coût** et **le degré du polynome**.

A noter qu'il était compliqué de tuner les paramètres avec une méthode déjà existante étant donné que nous avons utilisé notre propre fonction "oneoneSVMpred".

On a donc décidé d'évaluer les meilleures valeurs de ces hyper-paramètres en calculant différentes erreur de X-validation et il nous a ainsi fallu être parcimonieux sur le nombre de valeurs à tester pour chaque hyperparamètre.

En effet étant donné que l'on vérifie le taux d'erreur sur les 5 partitions de la cross-validation, on a besoin de faire 5 * Nombre_degré_hyperparam1 * Nombre_degré_hyperparam2 itérations de oneoneSVMpred qui elle même fait appel (5+4+3+2+1)=**15** fois à la méthode svm.

On doit au final faire **125 fois le produit des degrés à tester** appels à la fonction SVM !

On a donc obtenus les résultats suivants avec la méthode **one vs one** :

```
load("parole/polyTuning.RData")
polynomial.cost.degree_1
```

```
##               1         2          3         4
## 0.01 0.17333333 0.8008889 0.86755556 0.8093333
## 0.1  0.07955556 0.5533333 0.23422222 0.5271111
## 1    0.07422222 0.3911111 0.12755556 0.3991111
## 10   0.08311111 0.3462222 0.09422222 0.3506667
## 100  0.09555556 0.3488889 0.09022222 0.3462222
```

Avec en abscisse **le degré** et en ordonnée **la valeur de cost**, pour des valeurs de *gamma* et *coef0* laissées par défaut.

On note qu'on obtient une valeur intéressante de **7.4%** pour **degré = 1** et **cost = 1**.

On peut aussi tester de la même manière les méthodes **one vs all** :

```
polynomial.oa.cost.degree_1
```

```
##               1         2          3         4
## 0.01 0.83733333 0.8386667 0.83777778 0.8377778
## 0.1  0.09111111 0.8404444 0.70577778 0.8400000
## 1    0.08044444 0.1275556 0.19155556 0.4066667
## 10   0.09777778 0.1097778 0.09555556 0.1551111
## 100  0.11777778 0.1231111 0.10888889 0.1213333
```

On obtient des meilleurs résultats pour les degrés autre que 1 que en **one vs one**. Et pour des degrés plus élevé on regarde ce que donnent des valeurs de cost aussi plus grandes (selon la façon dont semblent évoluer les erreurs dans la matrices précédente), ce qui nous donne :

```
polynomial.oa.cost.degree_2
```

```
##               3         4         5         6
## 100   0.1146667 0.1280000 0.1440000 0.2533333
## 1000  0.1120000 0.1248889 0.1422222 0.1635556
## 10000 0.1120000 0.1235556 0.1435556 0.1484444
## 1e+05 0.1120000 0.1235556 0.1435556 0.1657778
## 1e+06 0.1120000 0.1235556 0.1435556 0.1657778
```

```
polynomial.oa.cost.degree_3
```

```
##               3         5         7         9
## 1e+09 0.1035556 0.1271111 0.2182222 0.3231111
## 1e+12 0.1035556 0.1271111 0.2182222 0.3231111
## 1e+15 0.1035556 0.1271111 0.2182222 0.3231111
## 1e+18 0.1035556 0.1271111 0.2182222 0.3231111
```

Nous en concluront deux choses :

- On obtient des meilleures valeurs avec la méthode **one vs one**.

- Quelque soit la valeur de **cost** que l'on choisit le degré le plus efficace est toujours 1.
- Les valeurs de **cost** optimales pour un noyau polynomial de degré 1 sont autour de 1.

On a aussi essayé de voir si on peut encore être plus précis en regardant les différentes valeurs de p autour de la valeur par défaut

```
1/p
```

```
## [1] 0.00390625
```

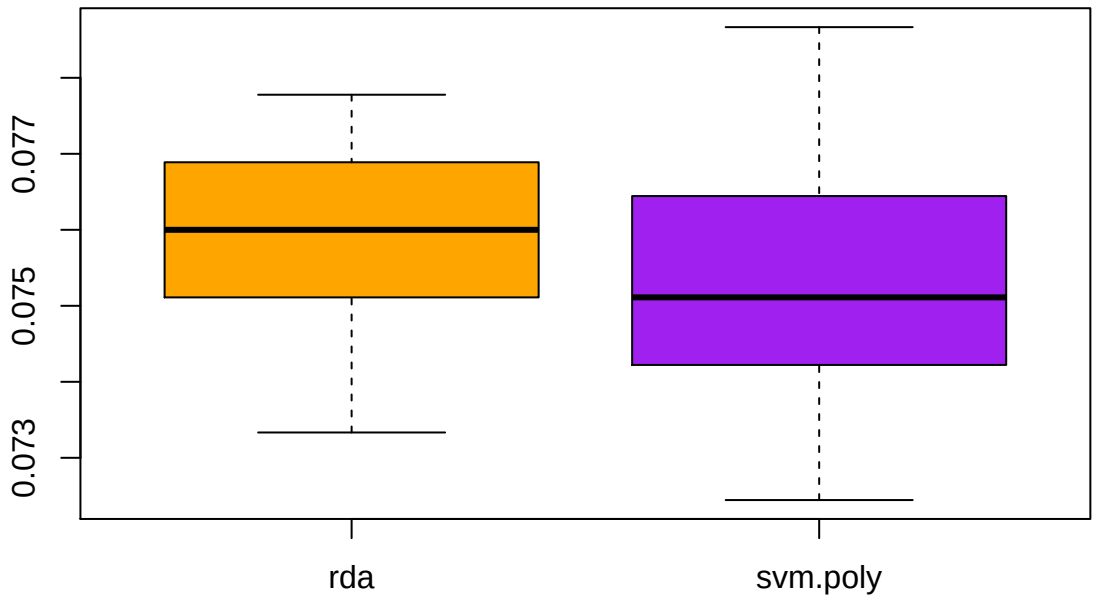
Mais on a obtenu les meilleurs valeurs proches de la valeur par défaut et on a donc déicdé de ne pas en changer. ### 2.c Radiale Nous avons été moins exhaustifs sur la recherche du polynôme avec un kernel radial étant donné que non seulement il n'y avait que deux paramètres à évaluer mais qu'on s'est vite rendu compte que les SVM avec ce kernel présentaient des performances elles aussi moins intéressantes que le noyau polynomial.

3) Conclusion

On obtient essentiellement deux méthodes avec des résultats intéressant : la **svm polynomiale** et la **rda** avec les hyper-paramètres bien choisis.

Comme pour le jeu de données précedent on va essayer de trancher entre les deux en faisant pour chacune des méthodes 10 réalisations du calcul de l'erreur de X-validation et d'observer le boxplot des résultats :

```
load(file="parole/res.RData")
boxplot(res, col=c("orange", "purple"))
```



On va donc pencher pour la méthode **SVM** qui semble donner constamment de meilleurs résultats. # IV -

Characters

```
data <- read.table('characters/characters_train.txt')
p <- length(data)- 1
N <- dim(data)[1]
K <- length(levels(data$y))
cat(p, " prédicteurs, ", N, " échantillons et 26 classes différentes.")
```

```
## 16  prédicteurs,  10000  échantillons et 26 classes différentes.
```

Il y a cette fois beaucoup d'échantillon pour un nombre raisonnable de prédicteur. On s'attendra donc à pouvoir plus facilement utiliser les méthodes d'apprentissage avec beaucoup de paramètre. Cependant cela sera contre-balancé par le fait qu'il y a beaucoup de classes diférentes.

1) Méthodes classiques

On va essayer différentes méthodes avec et sans projection étant donné qu'il existe tout de même 16 prédicteurs et que cela fait beaucoup de calcul à effectuer avec autant d'échantillons

```
load(file="characters/tuneRDA.RData")
parole.rda.cv
```

```
##           0   0.33   0.66      1
## 0    0.1248 0.1596 0.2066 0.3008
## 0.33 0.2090 0.2294 0.2593 0.3193
## 0.66 0.2899 0.2958 0.3088 0.3450
## 1    0.4500 0.4431 0.4364 0.4380
```

```
parole.proj.rda.cv
```

```
##          0    0.2    0.4    0.6    0.8      1
## 0   0.2083 0.2191 0.2362 0.2609 0.2953 0.3598
## 0.2 0.2239 0.2369 0.2530 0.2735 0.3028 0.3602
## 0.4 0.2481 0.2589 0.2726 0.2903 0.3151 0.3632
## 0.6 0.2753 0.2848 0.2959 0.3079 0.3294 0.3660
## 0.8 0.3147 0.3198 0.3248 0.3349 0.3498 0.3684
## 1   0.3068 0.3047 0.3023 0.3000 0.2997 0.2981
```

On remarque que la rda fonctionne bien mieux quand on ne projette pas les prédicteurs sur un ensemble de feature mais qu'on reste avec les prédicteurs de départ. Cela peut se comprendre car ceux-ci ne sont pas si nombreux que ça. On remarque aussi que les valeurs optimales de **gamma** et **lambda** sont 0 et 0 ce qui correspond à une QDA.

2) SVM

Pour la SVM on notera que les méthodes sur données prennent énormement de temps – probablement à cause du nombre d'échantillons – et donc qu'on va essayer de les tester un nombre minimum de fois.

On voudra aussi les essayer sur une projection de feature et sur les prédicteurs de départ.

2.a Radiale

Regardons l'erreur avec la méthode de vote **one vs one**

```
load(file="characters/tuningRadial.RData")
characters.radial.svm.err_1
```

```
##     0.0058  0.058   0.58
## 0.1 0.4524 0.1978 0.5041
## 1   0.1955 0.0668 0.0785
## 10  0.1069 0.0396 0.0752
```

On a trouvé des valeurs très intéressantes pour gamma=0.058 et cost=10. On a essayé d'explorer des valeurs de cost plus grandes :

```
characters.radial.svm.err_2
```

```
##       0.058
## 10   0.0393
## 100  0.0408
## 1000 0.0411
```

On en conclura que les valeurs optimales se trouvent vers **gamma=0.058** et **cost=10** avec un taux d'erreur autour de **4%**. **A NOTER** On a aussi essayé les méthodes **one vs all** et celles-ci, malgré le fait qu'il y ait plus de classes, fonctionnent bien moins.

2.b Autres kernel

On a trouvé des valeurs d'erreurs un peu plus elevées pour d'autres kernel et en raison non seulement de la qualité des résultats avec un kernel radial et du temps de calcul nécessaire pour tuner les valeurs nous n'avons pas été exhaustifs.

3) Conclusion

Pour ces données là la conclusion sera plus radicale : la **svm avec kernel radial** promet sans conteste de bien meilleures performances et c'est donc la méthode que nous allons choisir. Nous ne sommes pas surpris que ce soient les données avec laquelle la méthode SVM marche le mieux car celle-ci utilise finalement beaucoup de paramètres pour comparer chaque classes deux à deux en **one vs one** (il y a 26+27/2 =351 svm à faire), mais ce fait est compensé par le nombre d'échantillons et le faible nombre de prédicteurs. La méthode, ainsi très adaptatives s'adaptent donc bien aux nombreuses données et donne un résultat intéressant.

On aurait pu pour cette méthode, plus se pencher sur d'autre méthodes qui ont beaucoup de paramètres, notamment les méthodes G.A.M. ou des utilisations de l'algorithme EM basées sur les G.M.M..

Nous serons finalement plutôt satisfait de ce résultat.