Compte-rendu SY19 TP6

Jean-Baptiste Douet | Louis Martignoni

Introduction

Dans ce TP, nous disposons de trois jeux de données. Le premier représente des expressions de visages, le second des lettres de l'alphabet et le troisième des sons. Le but est alors de trouver les meilleurs classifieurs pour ces trois jeux de données en utilisant des méthodes d'apprentissage supervisé.

Expressions

Ce premier dataset sur les expressions du visage contient les niveaux de gris des images de taille 60 x 70. La classification doit se faire parmi 6 labels: joie, surprise, tristesse, dégoût, colère, peur.

Analyse

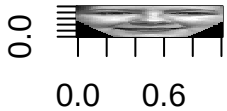
C'est un jeu de données particulier avec beaucoup plus de paramètres (4200) que d'individus dans le dataset, on n'a que 108 images en tout et pour tout. Cela rend donc la tâche plus complexe et le preprocessing important. Un autre aspect des données dont il va être difficile de tenir compte est la géométrie de l'image et donc la place de chaque pixel l'un par rapport à l'autre. En effet les algorithmes de machine learning classiques prennent des vecteurs (et non des matrices ou tenseurs) en entrée, on ne s'attend donc pas, en utilisant ces méthodes classiques, à d'excellents résultats. En ce qui concerne l'équilibre entre chaque classe, le nombre d'individus par classe est équilibré à l'exception de 2 classes (joy et sadness) qui ont plus d'individus que les autres. Il faudra donc voir si cette différence du nombre d'individus est significative et induit le classifieur en erreur ou non.

```
## y_expressions
##    anger  disgust     fear      joy  sadness surprise
##       17       15       16       21       23       16
```

Preprocessing

Elimmination des pixels noirs

Tout d'abord, il a fallu essayer de voir s'il était possible de réduire le nombre de dimensions du dataset.



Comme on le voit sur l'image ci dessous, il y a des zones noires sur les angles inférieurs.

Ces parties noires sont présentes sur toutes les images au même endroit et résultent donc en colonne égale à zéro dans le dataset. Puisqu'on ne tient pas compte de la géométrie de l'image, ces zones noires ne permettent pas de différencier une image par rapport à une autre (puisqu'elles sont similaires dans toutes les images), on peut donc éliminer leurs colonnes respectives dans le dataset.

```
data_preprocessed <- data_expressions[, !apply(data_expressions == 0, 2, all)]
dim(data_preprocessed)
```

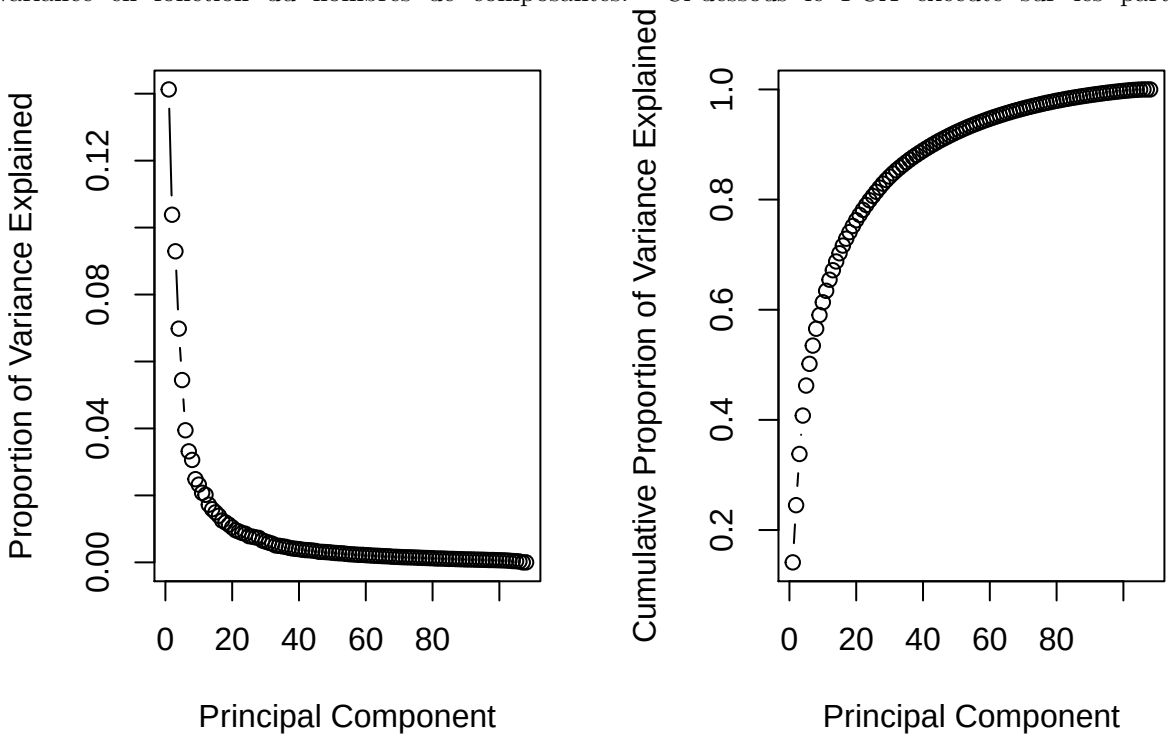
```
## [1]  108 3661
```

Sélection des parties expressives du visage

Une autre manière de réduire la dimension du dataset est de sélectionner les parties qui expriment l'expression du visage. Nous sommes parties de l'hypothèse que les parties les plus expressives sont celles autour des yeux et de la bouche. C'est une hypothèse forte que l'on a voulu tester et puisque les visages ne bougent pas (pas de rotation, faibles translations), il suffit de sélectionner une zone de pixels. Ci-dessous, un exemple du pca après la sélection des parties expressives du visage.

PCA

La dernière technique pour réduire le nombre de dimensions a été le PCA. En effet avec ou sans les sélections précédentes, le nombre de features dépassait toujours le millier. Dans chaque cas le choix du nombres de composantes principales s'est fait en traçant les courbes de propotions de la variance en fonction du nombres de composantes. Ci-dessous le PCA exécuté sur les parties de



Classification

Afin d'obtenir le meilleur résultat de classification nous avons testé plusieurs classifieurs sur différents types de vecteurs en entrée. En effet, tous les algorithmes ne réagissent pas de la même manière aux étapes de preprocessing présentées. Différentes combinaison parmi ces étapes ont donc été testées en fonction des propriétés des classifieurs.

Analyse

Voici les résultats (la précision obtenue) pour les differentes méthodes testées:

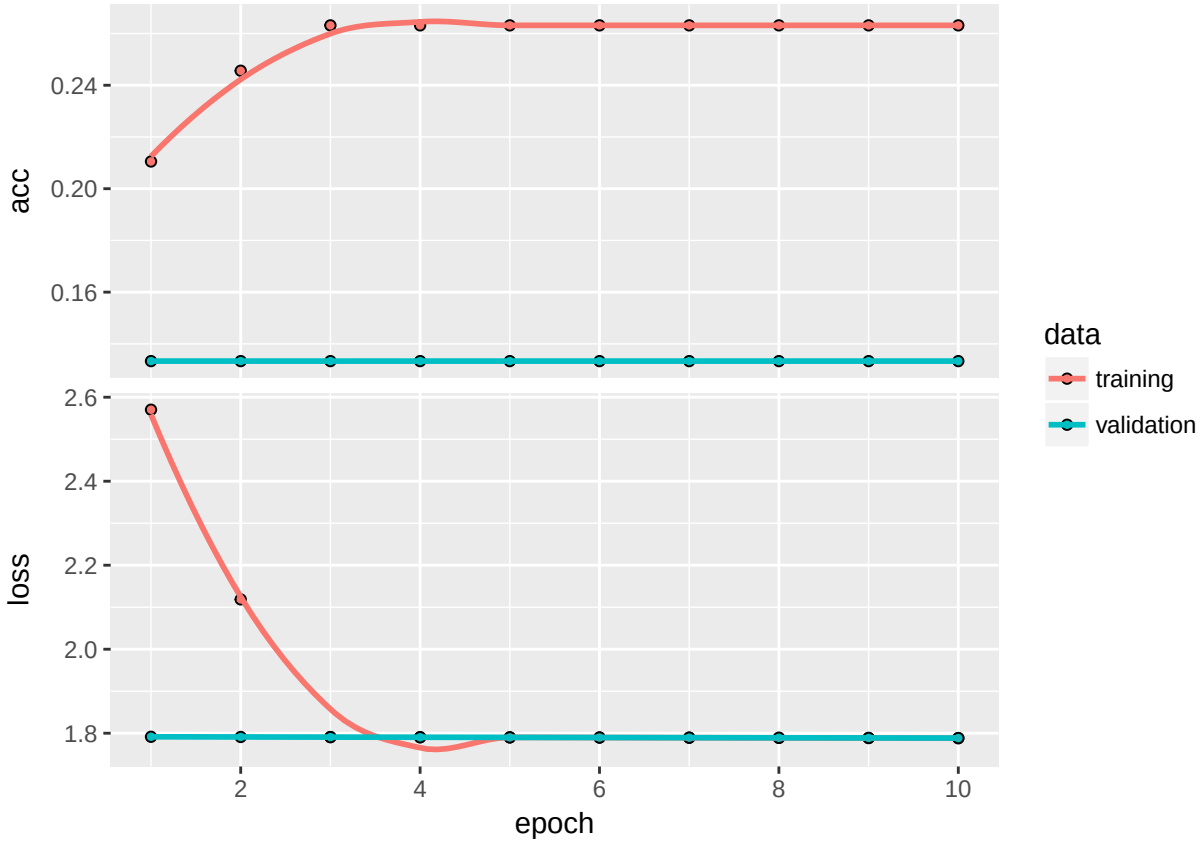
- SVM: 0.926
- RandomForest: 0.918
- LDA: 0.917
- RDA: 0.903
- Naive-Bayes: 0.88
- QDA: 0.66
- SVM + PCA: 0.66
- Naive-Bayes + PCA: 0.57
- LDA + PCA: 0.56
- QDA + PCA: 0.63

Autres méthodes

Afin d'exploiter la géométrie des images, il existe des algorithmes basés sur des réseaux de neurones à convolution appelés CNN. Ce sont ces algorithmes qui, pendant les dernières années, ont permis d'améliorer nettement les résultats de classification d'images par rapport aux algorithmes de machine learning classique. A l'aide de la librairie Keras, nous avons essayé de les exploiter de deux manières différentes.

1) Construire un réseau CNN

```
plot(history)
```



2) Transfer learning

Le transfer learning exploite un réseau de neurones profond déjà pré-entrainé afin de

La première chose

Character

Analyse

Nous commençons dans un premier temps par analyser notre jeu de données. Grâce à la commande 'summary', nous pouvons voir que chaque élément est décrit par 16 variables quantitatives et que nos individus vont logiquement être divisés en 26 classes représentant l'alphabet. On note également que les variables semblent issues d'une loi centrée autour de 0. Enfin les individus sont repartis assez Ã©quitablement dans les differentes classes.

```
table(character_data$Y)
```

```
##
##   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P   Q   R
## 399 345 367 397 401 418 390 347 372 366 382 381 397 367 396 420 412 360
##   S   T   U   V   W   X   Y   Z
## 366 367 404 392 383 414 395 362
```

Approche

Pour trouver le meilleur classifieur pour ces données, nous allons appliquer plusieurs méthodes étudiées en cours. Nous pourrons alors comparer l'efficacité de ces méthodes en comparant l'erreur de chaque classifieur fournie par validation croisée. Nous allons détailler certaines méthodes ci-dessous et l'ensemble des méthodes appliquées avec leurs résultats seront présentées ultérieurement dans un tableau récapitulatif.

Application

Dans un premier temps, nous allons appliquer simplement plusieurs mÃ©thodes conjointement avec la validation croisÃ©e pour obtenir des rÃ©sultats comparables et significatifs. Nous commenÃ§ons donc par diviser nos donnÃ©es en deux parties, un ensemble train comportant les deux tiers des donnÃ©es et un ensemble de test comportant le reste.

Une fois la séparation faite, nous pouvons alors commencer à tester diffÃ©rents modÃ¨les pour avoir une première idÃ©e des performances de chacun. Pour cela, on entraine notre modèle sur la partie 'train' et nous prÃ©disons nos classes sur la partie 'test'. Cependant, pour comparer les différentes mÃ©thodes il est plus judicieux d'avoir recours Ã la validation croisée. Nous mettons alors en place ce système qui nous permettra d'avoir des rÃ©sultats plus significatifs pour comparer nos classifieurs. Nous divisons alors nos donnÃ©es en dix parties de mÃªme taille, Ã chaque itération de notre boucle nous entraÃ®nons notre modèle sur la partie des donnÃ©es de 'train' et ensuite nous pouvons prédire les donnÃ©es de test. A chaque itÃ©ration un nouvel ensemble de 'test' et donc de 'train' sont utilisÃ©s. On fait alors une moyenne des rÃ©sultats que nous avons obtenu pour obtenir une erreur stable et significative qu'on utilisera pour comparer nos differentes mÃ©thodes et ensuite choisir celle qui nous donne la plus petite erreur.

Après avoir testé nos differents modèles, nous avons également mis en place une mÃ©thode de réduction de la dimension (ACP). Les données étant déjà centrées autour de 0 et avec des valeurs assez proches, nous n'avons ni besoin de normaliser ces données ni de les redimensionner avant d'appliquer l'ACP.

Pour ce jeu de données, c'est le modèle du random forest qui a donné le meilleur résultat.

```
n_folds <- 10
folds_i <- sample(rep(1:n_folds, length.out = n))
CV<-rep(0,10)
for (k in 1:n_folds) {
  test_i <- which(folds_i == k)
  train_xy <- character[-test_i, ]
```

```
  test_xy <- character[test_i, ]
  rf <- randomForest(Y ~ ., data = train_xy)
  pred_rf<-predict(rf, newdata = test_xy, type = "response")
  prop.table(table(test_xy$Y,pred_rf))
  cm= as.matrix(table(test_xy$Y,pred_rf))
  CV[k]<- sum(diag(cm)) / sum(cm)
}
CVerror= sum(CV)/length(CV)
```

En effet, pour ce jeu de données les méthodes de résolution linéaire sont moins efficaces. C'est pour cette raison que le LDA a des performances moindres comparé au random forest qui est moins dépendant des variables. De même on peut remarquer que le QDA nous donne de meilleures performances que le LDA. Ceci est probablement expliqué par le fait que les variables n'ont pas les mÃªmes matrices de variance-covariance. Dans ce cas, les méthodes linéaires comme le LDA sont beaucoup moins efficaces pour differencier les differentes classes. Dans ce cas de figure, les méthodes quadratiques comme le QDA ainsi que d'autres méthodes moins dépendantes de cette caractéristique comme le random forest nous donne de meilleurs résultats.

Résultats

Voici les résultats de précision obtenus pour les differentes méthodes testées:

- RandomForest: 0.9343
- SVM: 0.913
- Naive-Bayes: 0.69
- LDA: 0.70
- QDA: 0.88
- RDA: 0.87
- SVM + PCA: 0.71
- Naive-Bayes + PCA: 0.64
- LDA + PCA: 0.65
- QDA + PCA: 0.68

Comme prévu, les méthodes simples et linéaires sont celles qui nous donnent les classifieurs les moins précis. Naive-Bayes est ici peu performant ce qui pourrait s'expliquer par une trop grande corrélation entre plusieurs prÃ©dicteurs.
Nous avons alors ensuite ajouté une partie de traitement des données avec l'ACP. En effet, notamment pour amÃ©liorer les résultats de modèles comme Naive-Bayes nous avons appliquÃ© l'ACP pour réduire les dimensions de notre jeu de donnÃ©es. Cependant, quel que soit le modèle auquel on a appliquÃ© l'ACP, le résultat de notre classifieur devenait moins bon. Ceci s'explique par le fonctionnement de ce traitement. En effet, l'ACP prend en compte seulement les coordonnÃ©es des points de nos données. L'ACP peut donc supprimer des informations qui sont pourtant importantes pour classifier nos donnÃ©es. En fonction de notre jeu de donnÃ©es ce processus de construction peut mener Ã de mauvaises composantes n'expliquant pas bien nos classes et donc rÃ©sultant en des classifieurs moins performants. C'est pour cette raison que cette phase de traitement n'a pas Ã©tÃ© maintenue pour notre classifieur final.

Conclusion

Après le test de nos différents modèles et mÃªme de l'ajout d'une phase de traitement des donnÃ©es, les rÃ©sultats obtenus par validation croisÃ©e nous permettent de choisir le meilleur classifieur pour ce jeu de donnÃ©es. Ainsi c'est le randomForest qui nous donne le meilleur rÃ©sultat avec une erreur de seulement 6.6% ce qui est assez satisfaisant.

Paroles

Analyse

Comme pour le jeu de donnÃ©es prÃ©cÃ©dent, nous commenÃ§ons par analyser notre jeu de donnÃ©es avec la commande 'summary'. Nous avons donc prÃ¨s de 2500 individus caractÃ©risÃ©s par 256 variables. Ces individus sont rÃ©partis dans cinq classes.

```
table(parole$y)
```

```
##
##  aa  ao dcl  iy  sh
## 365 500 370 588 427
```

Nous pouvons voir que les diffÃ©rentes classes ne contiennent pas le mÃªme nombre d'individus mais les diffÃ©rences sont relativement faibles (entre 16 et 26%). On note Ã©galement que les variables semblent issues d'une loi centrÃ©e autour de 0.

Approche

Nous allons aborder ce problÃ¨me de la mÃªme maniÃ¨re que le prÃ©cÃ©dent. Nous allons comparer diffÃ©rents modÃ¨les en comparant l'erreur issue de la validation croisÃ©e et pourrons ainsi choisir notre meilleur classifieur pour ce jeu de donnÃ©es.

Application

De la même manière que pour le jeu de données précédent, nous allons créer notre boucle de validation croisée de dix itérations et à l'intérieur de celles ci nous allons entrainer et tester nos modèles.
De même, nous avons également mis en place une mÃ©thode de réduction de la dimension (ACP). Les données étant déjà centrées autour de 0 et avec des valeurs assez proches, nous n'avons ni besoin de normaliser ces données ni de les redimensionner avant d'appliquer l'ACP.

```
n_folds <- 10
folds_i <- sample(rep(1:n_folds, length.out = n))
CV<-rep(0,10)
for (k in 1:n_folds) {
  test_i <- which(folds_i == k)
  train_xy <- parole[-test_i, ]
  test_xy <- parole[test_i, ]
  svm_train <- svm(y ~ ., data = train_xy)
  pred_svm<-predict(svm_train, newdata = test_xy, type = "response")
  prop.table(table(test_xy$y,pred_svm))
  cm= as.matrix(table(test_xy$y,pred_svm))
  CV[k]<- sum(diag(cm)) / sum(cm)
}
CVerror= sum(CV)/length(CV)
CV
```

```
##  [1] 0.9066667 0.9111111 0.9111111 0.9244444 0.9466667 0.9288889 0.9244444
##  [8] 0.9244444 0.9377778 0.9066667
```

```
CVerror
```

```
## [1] 0.9222222
```

Ici, c'est le SVM qui nous donne le meilleur résultat. ### Analyse Voici les résultats de précision obtenus pour les differentes méthodes testées:

- SVM: 0.926
- RandomForest: 0.918
- LDA: 0.917
- RDA: 0.903
- Naive-Bayes: 0.88
- QDA: 0.66
- SVM + PCA: 0.66
- Naive-Bayes + PCA: 0.57
- LDA + PCA: 0.56
- QDA + PCA: 0.63

On remarque ici que quatre modèles ont des performances très similaires Ã savoir SVM, RandomForest, LDA ainsi que RDA. D'après les résultats, c'est le SVM qui est le meilleur classifieur pour ce jeu de données. Cependant, on voit que le LDA a un score très proche or il est souvent plus simple d'interpréter les résultats avec le LDA qu'avec le SVM. Pour ce TP, nous allons tout de même garder le meilleur score pour notre classifieur puisqu'on cherche le classifieur le plus précis possible. On peut remarquer qu'ici aussi en ajoutant une phase de traitement des donnÃ©es avec rÃ©duction des dimensions grÃ¢ce Ã l'ACP, les performances des classifieurs sont diminuÃ©es. Encore une fois, ceci s'explique probablement par de la perte d'informations lors de la crÃ©ation des composantes.

Conclusion

Après avoir testé différents modèles pour ce jeu de données et mis en place une phase de traitement des données, les résultats obtenus par validation croisée nous permettent de choisir le meilleur classifieur. Ainsi c'est le SVM qui nous donne le meilleur résultat avec une erreur de seulement 7.4% ce qui est assez satisfaisant.