Introduction au Machine Learning : Apprentissage supervisé

Guillaume Metzler

2021-2022

Abstract

Ce premier TP a pour but de vous faire coder quelques algorithmes simples par vous même avant de vous introduire les fonctions implémentées sous R qui vous permettront de résoudre diverses tâches d'apprentissage supervisé. Nous commencerons par :

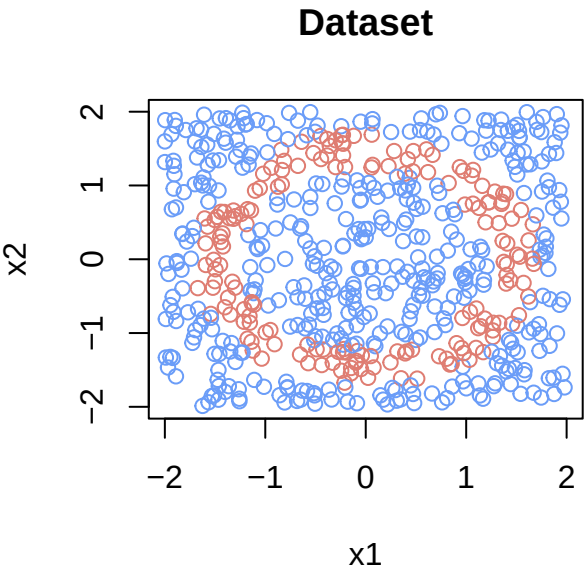
- Travailler sur l'algorithme du plus proche voisin en écrivant soit même le processus
- Nous verrons ensuite comme faire de la validation à l'aide du logiciel R pour tuner la valeur de k de l'algorithme
- Nous aborderons ensuite le problème de classification des SVM linéaires et non linéaires (se référer aux slides) avec R et nous tunerons les paramètres pour faire de la validation-croisée des hyper-paramètres.

Algorithme des plus proches voisins

Une implémentation brute et la version R

Nous commençerons par regarder comment implémenter son propre algroithme des plus proches voisins. Pour cela on va simuler notre propre jeu de données sur lequel nous allons travailler.

- Générer un jeu de données X en 2D dans l'espace $[-2, 2]^2$ selon une loi uniforme, on prendra $n = 600$
- Calculer la distance des points à l'origine et attribuer l'étiquette $Y = 1$ si la distance à l'origine est comprise entre 1.5 et 3 et -1 sinon
- Représenter votre jeu de données, il devrait ressembler à la figure ci-dessous



On va maintenant implémenter la fonction nous permettant de créer notre propre algorithme du plus proche voisin "my_knn", il prendra au total quatre arguments : la valeur de k, votre jeu d'entraînement X_{train} ainsi

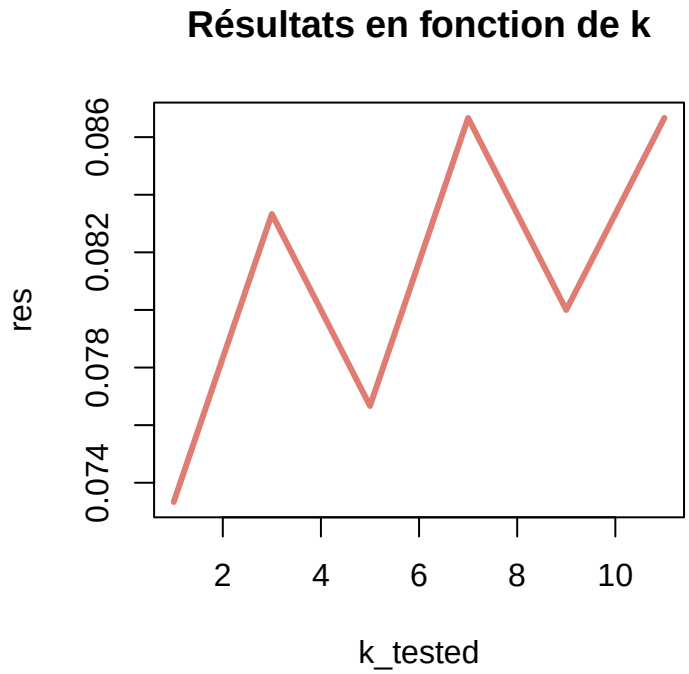
que les étiquettes Y et les données que vous souhaitez classer X_{test}. Vous êtes libres sur la façon dont vous implémenter votre algorithme (boucle ou autres process) mais vous devrez faire figurer les étapes suivantes

- Calculer la distance entre les objets de X_{test} aux objets X_{train}.
- Pour chaque exemple, on ordonnera les distances dans l'ordre croissant à l'aide de la fonction "sort" —> on s'intéresse surtout aux indices des exemples pour récupérer les étiquettes associées !
- On attribue ensuite les étiquettes aux données tests en fonction des k plus proches voisins.

On va maintenant regarder les performances de l'algorithme, on va donc créer une fonction qui va prendre comme paramètres les données labels de nos données d'entraînement et les labels des données tests pour calculer le taux d'erreur. Ecrire une telle fonction.

Une fois que la fonction est écrite :

- générez un jeu de données tests, de taille $n = 300$, de la même façon que votre jeu d'entraînement
- testez votre modèle pour différentes valeurs de k, par exemple pour $k = 1, 3, 5, 7, 9$ et $k = 11$ et représentez graphiquement les résultats afin d'obtenir un graphe qui ressemble à celui présenté ci-dessous :



- Comparez vos résultats avec la fonction k-NN de la librarie "class" R, en utilisant les instructions ci-dessous (on s'assurera de retrouver la même chose en terme de résultats)

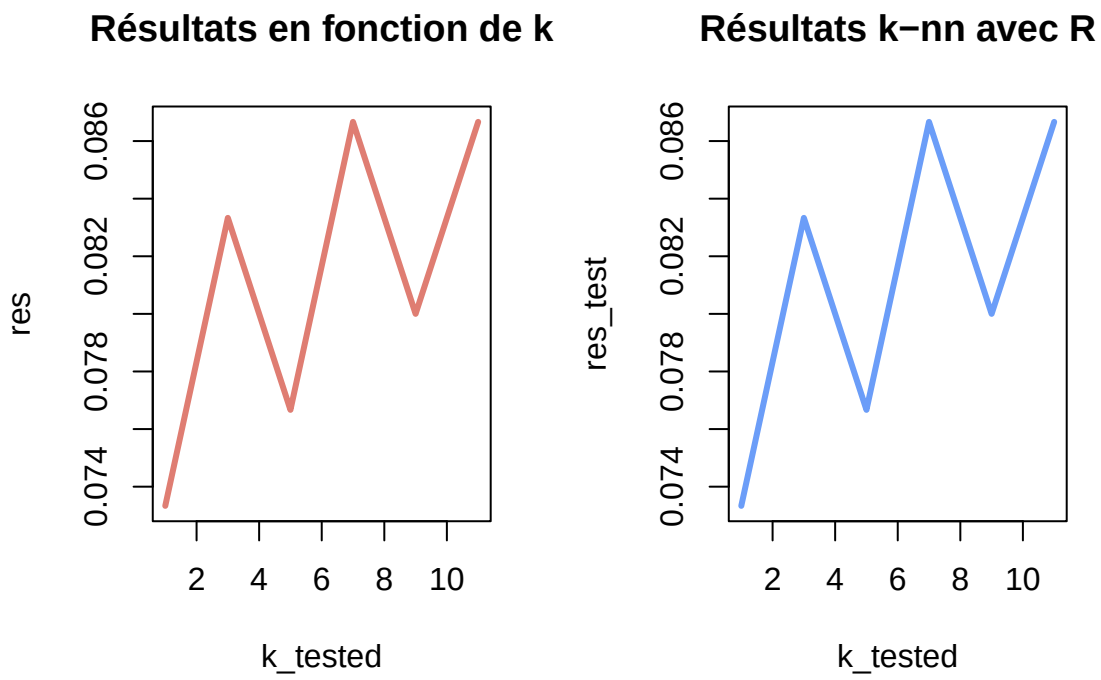
```
library(class)

res_test = NULL
i=1

for (j in seq(1,11,2)){
  pred <- knn(X,X_test,Y,k=j)
  res_test[i] = mean(pred != Y_test)
  i=i+1
}

par(mfrow=c(1,2))
```

```
plot(k_tested, res, type ='l', lwd= 3, col =  "#DF7D72", main = "Résultats en fonction de k")
plot(k_tested, res_test, type ='l', lwd= 3, col =  "#6B9DF8", main = "Résultats k-nn avec R")
```



Cette première étape a permis de se familiariser avec la manipulation d'un ensemble d'apprentissage et de test avec une fonction implémentée soit même et une fonction prête à l'emploi sous R. On va maintenant se concentrer uniquement sur l'usage de fonctions disponibles avec R et aborder le problème d'un point de vue Machine Learning.

Un vrai apprentissage de modèles : validation croisée

Pour le moment, nous avons simplement tester différentes valeurs de k, mais dans la pratique, il faut que l'algorithme se serve de l'ensemble d'apprentissage pour qu'ils apprennent de lui-même quelle est la valeur de k la plus intéressante. Cela peut se faire à l'aide d'un ensemble de validation par validation croisée.

Pour cela, utilisera le package "caret" disponible sous R ainsi qu'un jeu de données "Smarket" disponible par le biais de la librairie "ISLR". Avec ce jeu de données, on cherche à prédire si un produit est rentable ("Direction") à l'aide d'un ensemble de descripteur

```
library(ISLR)
library(caret)
```

```
## Le chargement a nécessité le package : ggplot2
```

```
## Le chargement a nécessité le package : lattice
```

```
data(Smarket)
Var = c("Lag1" ,"Lag2", "Lag3", "Lag4", "Lag5", "Volume", "Today","Direction")
```

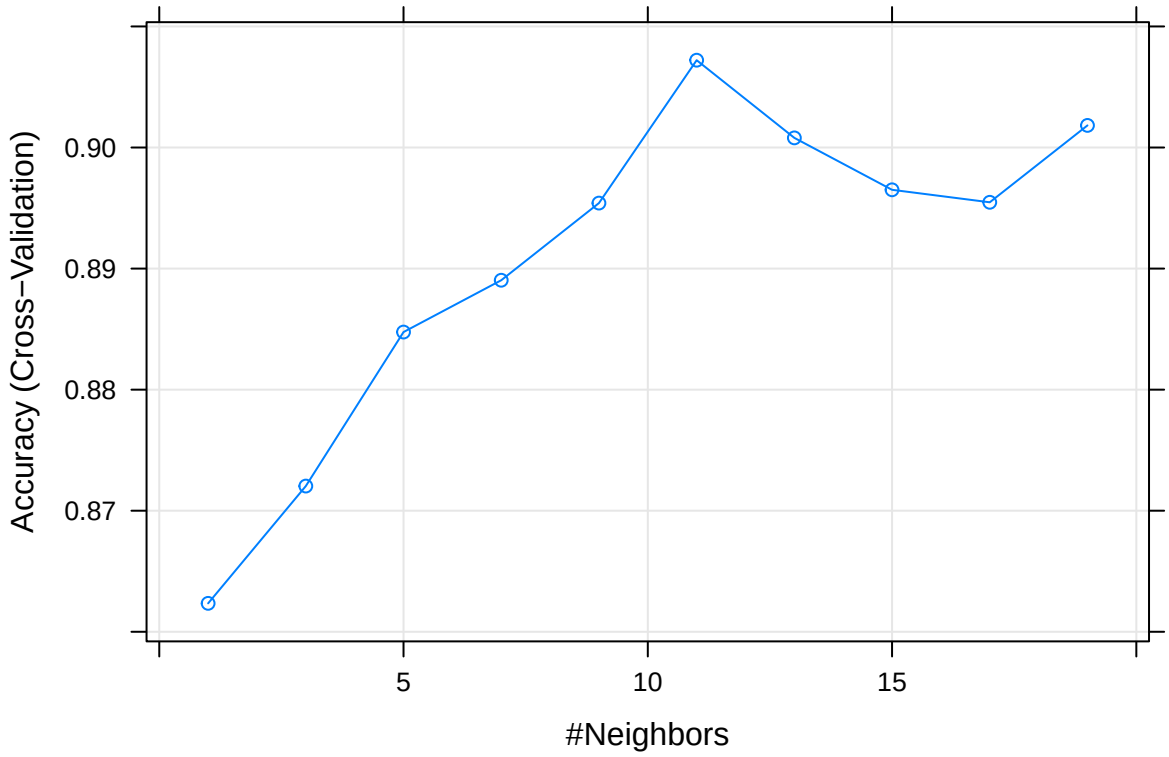
Pour cette étude on écartera la variable "Year", elle ne sera pas prise en compte pour l'étude du modèle.

- Etudiez le range de valeurs des différentes variables et, si nécessaire, normalisez les données par la méthode de votre choix.
- A l'aide de la fonction "createDataPartition", séparez votre jeu de données en deux : 75% des données pour l'entraînement et 25% pour le test
- Etudiez le fonctionnement de la fonction "trainControl" afin de définir une "10 fold cross-validation"

- A l'aide de la fonction "expand.grid" définir la grille de recherche des valeurs de k pour l'algorithme, on prendra les valeurs impaires de 1 à 19
- Le modèle s'apprend à l'aide de la fonction "train" comme illustré ci-dessous. On peut ensuite regarder ce que donne le modèle à l'aide de la fonction "plot". Quelle remarque pouvez-vous faire concernant la valeur de k ?

```
# Apprentissage du modèle
model <- train( Direction~ . ,data=train,
                trControl=train_control,
                method="knn",
                tuneGrid = mygrid
                )

plot(model)
```



On peut avoir le détail des résultats de la cross-validation à l'aide de la fonction summary. Par défaut, le modèle optimal retenu en phase de test sera celui maximisant l'accuracy en validation croisée.

- Evaluer votre modèle sur votre jeu de données test en calculant l'accuracy à l'aide de la fonction "predict"
- Donner la matrice de confusion
- Comparez l'erreur en cross-validation et l'erreur en test pour les différentes valeurs de k de votre algorithme. Est-ce que la valeur retenue par le processus de cross-validation vous semble pertinente ?

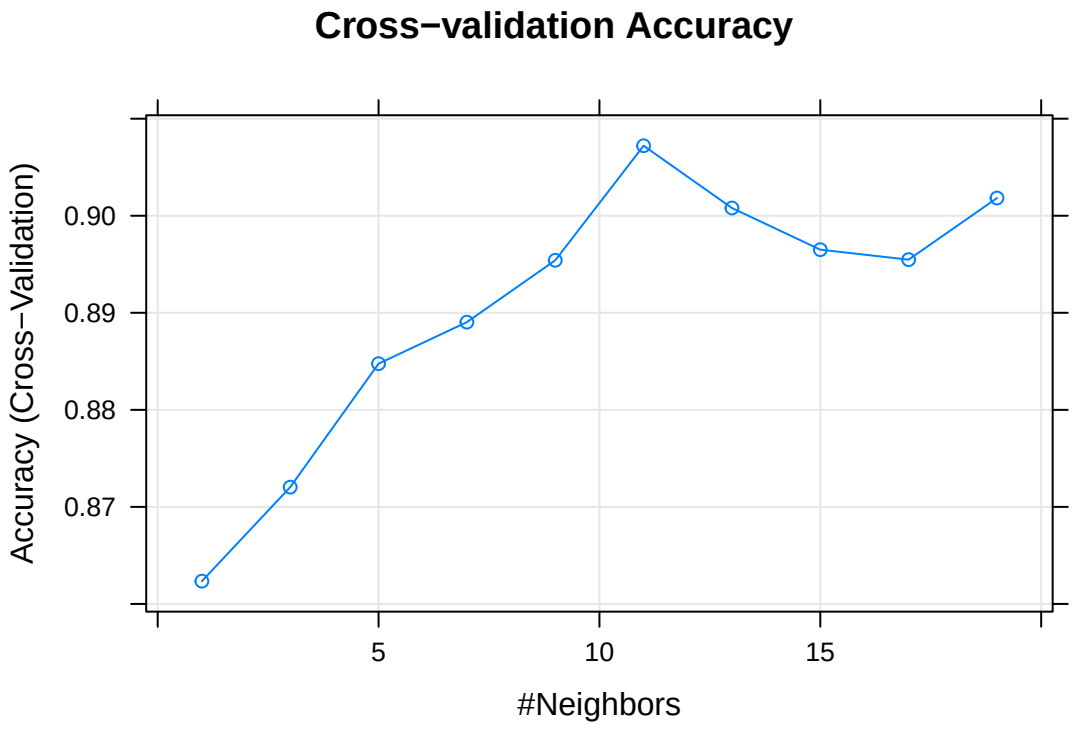
```
### Vérification en test
table_test = NULL
i=1

for (j in seq(1,19,2)){
  res <- knn(train[,-which(colnames(test)=="Direction")],
```

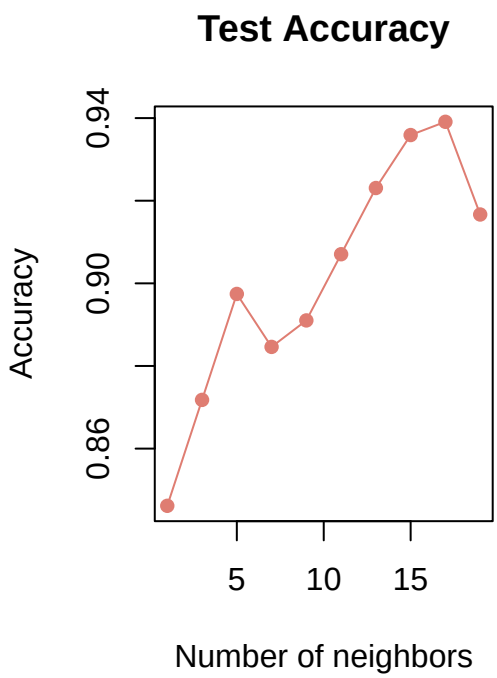
```
              test[,-which(colnames(test)=="Direction")],
              train[,"Direction"],
              k=j)
  table_test[i] = mean(res == test[,"Direction"])
  i=i+1
}

par(mfrow = c(1,2))

plot(model, main = "Cross-validation Accuracy")
```




```
plot(seq(1,19,2), table_test, xlab = "Number of neighbors" , ylab="Accuracy", pch=16,
     col =  "#DF7D72", type='o', main = "Test Accuracy" )
```



Autour des SVM linéaires et non linéaires

Un premier modèle simple pour appréhender les modèles

Le code ci-dessous vous permettra de générer des graphiques vous permettant de visualiser les résultats de votre algorithmes. Compléter ce code à l'aide des informations données ci-dessous :

- créer un jeu de données en deux dimensions de taille de 100, distribuées selon une loi normale centrée réduite (notée x)
- à l'aide d'un vecteur y attribuer le label 1 à la moitié des données et le label -1 à l'autre moitié
- Augmenter de 4 la valeur de chaque composante des données dont le label est $+1$

```
# On fixe la graîne, cela vous permettra d'avoir des résultats identiques
set.seed(10111)

# librairie
library(e1071)

# Grille pour graphe
make.grid = function(x, n = 200) {
  grange = apply(x, 2, range)
  x1 = seq(from = grange[1,1], to = grange[2,1], length = n)
  x2 = seq(from = grange[1,2], to = grange[2,2], length = n)
  expand.grid(X1 = x1, X2 = x2)
}

#Jeu de données
n_ech = 100
x = cbind(rnorm(100), rnorm(100))
y = (2*rbinom(100,1,0.5)-1)
x[y == 1,] = x[y == 1,] + 4
```

```
# Génération de la grille adaptée à la taille des donneés
xgrid = make.grid(x)
```

On va maintenant s'attaquer à l'apprentissage du modèle. Pour cela regarder comment fonctionne la fonction "svm" et apprenait un modèle de SVM linéaire qui va chercher à prédire y en fonction de x avec un paramètre C (cost) que vous fixerez à 1 dans un premier temps et répondez aux questions suivantes :

- Faites variez C (cost) dans l'intervalle $[0, 10]$ par exemple.
- Qu'observez-vous ? Quelle est l'influence de ce paramètre ?
- Vous avez abservé que certains points sont encadrés, que peut-on dire la position de ces points par rapport au séparateur (ligne noire) ? Que représentent ces points ?

```
# Apprentissage du modèle et évaluation des labels
dat = data.frame(x, y = as.factor(y))
svmfit = svm(y~., data = dat, cost = 10, scale = FALSE, kernel = "linear"  )
print(svmfit)
plot(svmfit,dat)
ygrid = predict(svmfit, xgrid)

# Représentation graphique
beta = drop(t(svmfit$coefs)%*%x[svmfit$index,])
beta0 = svmfit$rho

plot(xgrid, col = ifelse( (as.numeric(ygrid)-1)==1 ,"#DF7D72","#6B9DF8"), pch = 20, cex = .2,
     main = "Un SVM linéaire")
points(x, col = ifelse( y==1 ,"#DF7D72","#6B9DF8"), pch = 19)
points(x[svmfit$index,], pch = 5, cex = 2)
abline(beta0 / beta[2], -beta[1] / beta[2])
abline((beta0 - 1) / beta[2], -beta[1] / beta[2], lty = 2)
abline((beta0 + 1) / beta[2], -beta[1] / beta[2], lty = 2)
```

Ce cas là est presque trop simple dans le sens où nos données sont parfaitement séparables. Reprendez le bout de code permettant de générer les données de façon à ce que le jeu de données ne soit pas linéairement séparable et refaite quelques tests sur la valeur du paramètre C (cost) en affichant aussi la matrice de contingence des résultats de classification.

Vers une modèle non linéaire

En repartant du code précédant et en modifiant le jeu de données comme indiqué ci-dessous, que pouvez vous dire concernant ce type de modèle ?

```
# Génération des données
n_ech = 400
x = matrix(rnorm(n_ech), n_ech/2, 2)
y = ifelse(x[,1]^2+x[,2]^2<1, 1, -1)
```

En fait les modèles linéaires ne sont que très rarement utiles en pratiques lorsqu'ils sont employés seuls (en les combinant on peut créer des modèles non linéaires puissants ! Mais cela sort du cadre de ce TP). Une autre approche consiste à utiliser des méthodes à noyaux, cela permet de potentiellement projeter ses données dans un espace de dimension infinie dans lequel un séparateur linéaire est capable de correctement séparer les données.

- Regardez les options de la fonction "svm" afin de créer un modèle gaussien.
- Combien d'hyper-paramètres devons-nous renseigner pour ce type de modèle ? Essayez d'expliquer l'influence de ces paramètres.

- Jouez avec le paramètre spécifique au noyau gaussien pour observer son effet.

On pourra également comparer les différences de résultats sur le jeu de données “cats” disponible dans la librairie “MASS”.

Apprentissage d’un modèle

On se propose maintenant d’apprendre le meilleur modèle non linéaire gaussien qui permet de maximiser les performances en classification et les comparer à un potentiel meilleur modèle linéaire. Pour cela, on utilisera un jeu de données “ESL.mixture.rda” disponible dans les ressources de ce TP. On se passera cette fois-ci d’une représentation graphique, sauf si vous souhaitez en faire une. On rappelle les différentes étapes

- Commencez par séparer votre jeu de données en un ensemble train/test avec 75%/25% des données.
- Effectuez une 4-CV pour apprendre le meilleur SVM linéaire en tunant l’hyper-paramètres associé
- Faite de même avec un SVM non linéaire
- Testez les deux modèles sur votre jeu de données test et comparez les résultats.