

# Big Data Mining

## Apprentissage Supervisé

### Master 2 BIBD et Master 2 SISE

Guillaume Metzler

guillaume.metzler@univ-lyon2.fr



**INSTITUT  
de la  
communication**



Université de Lyon, Lyon 2, ERIC EA3083, Lyon, France

Automne 2020

# Introduction

# Qu'est-ce que l'apprentissage supervisé ?

Considérons  $\mathcal{D} = \mathcal{X} \times \mathcal{Y}$  la distribution inconnue de nos données. L'espace  $\mathcal{X}$  est appelé **input space** ou encore **feature space**, en général  $\mathcal{X} \subset \mathbb{R}^d$ . L'espace  $\mathcal{Y}$  est l'**espace des étiquettes** ou encore l'**output space** :

- $\mathcal{Y} = \mathbb{R}$  : régression
- $\mathcal{Y} = \{0, 1\}$  : classification binaire
- $\mathcal{Y} = \{1, \dots, C\}$  : classification multi-classe
- $\mathcal{Y} = \{0, 1\}^C$  : classification multi-label

**Objectif** : trouver un algorithme  $\mathcal{A}$ , générant une hypothèse  $h$  capable d'effectuer la tâche souhaitée.

**Problème** : en pratique  $\mathcal{D}$  est inconnue

# En pratique

On dispose uniquement d'un échantillon fini  $S = \{\mathbf{x}_i, y_i\}_{i=1}^m \sim \mathcal{D}$ . Dans la suite, on supposera que nos données  $\mathbf{x}_i$  sont numériques et on notera :

$$X = (\mathbf{x}_1, \dots, \mathbf{x}_m) = \begin{pmatrix} \mathbf{x}_{11} & \cdots & \mathbf{x}_{1d} \\ \vdots & \ddots & \vdots \\ \mathbf{x}_{m1} & \cdots & \mathbf{x}_{md} \end{pmatrix}$$

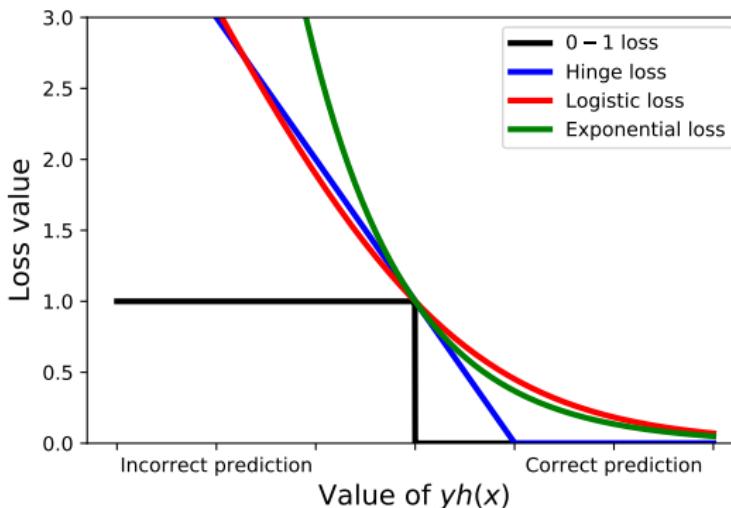
On souhaite donc trouver une hypothèse  $h$  qui soit performante sur notre échantillon  $S$  mais aussi sur tout nouvel exemple  $(\mathbf{x}, y)$ .

**Comment apprendre une telle hypothèse ?**

# En pratique

## Deux éléments importants :

- définir un algorithme  $\mathcal{A}$  (SVM,  $k$ -NN, Boosting, ...)
- définir la quantité que l'on souhaite minimiser, i.e. définir **une fonction de coût ou loss function**  $\ell : (\mathbf{x}, y) \mapsto \ell(h(\mathbf{x}), y)$



# En pratique

Notre algorithme  $\mathcal{A}$  va donc chercher une hypothèse optimale  $h^*$  telle que :

$$h^* = \arg \min_h \sum_{i=1}^m \ell(h(\mathbf{x}), y).$$

## Remarque

Comme on dispose d'un échantillon fini, on peut trouver un algorithme qui va apprendre parfaitement nos données !

→ **Problème** : il ne sera pas capable de généraliser sur de nouvelles données

## Comment se prévenir d'un tel phénomène ?

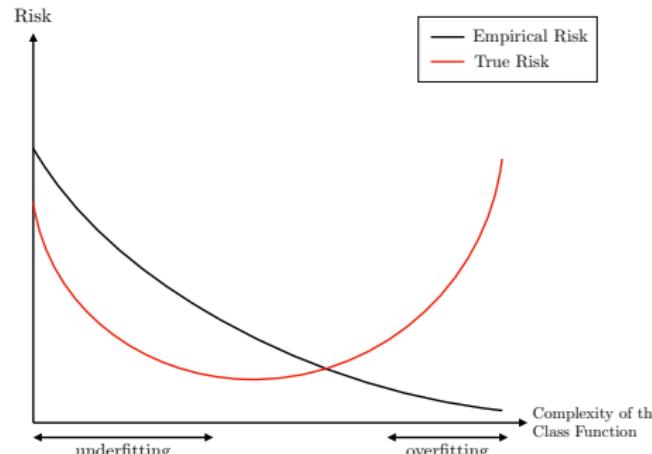
Il faut restreindre le spectre des modèles que l'on s'autorise à apprendre, i.e. diminuer la richesse ou la complexité de notre espace d'hypothèses  $\mathcal{H}$ .

# Ajout d'un terme de Régularisation

→ Utilisations de termes de régularisations dans notre problème d'optimisation.

$$h^* = \arg \min_h \sum_{i=1}^m \ell(h(\mathbf{x}_i), y_i) + \lambda \|H\|,$$

où  $\lambda$  : paramètre de régularisation, contrôle la complexité de  $\mathcal{H}$ .



# Validation croisée

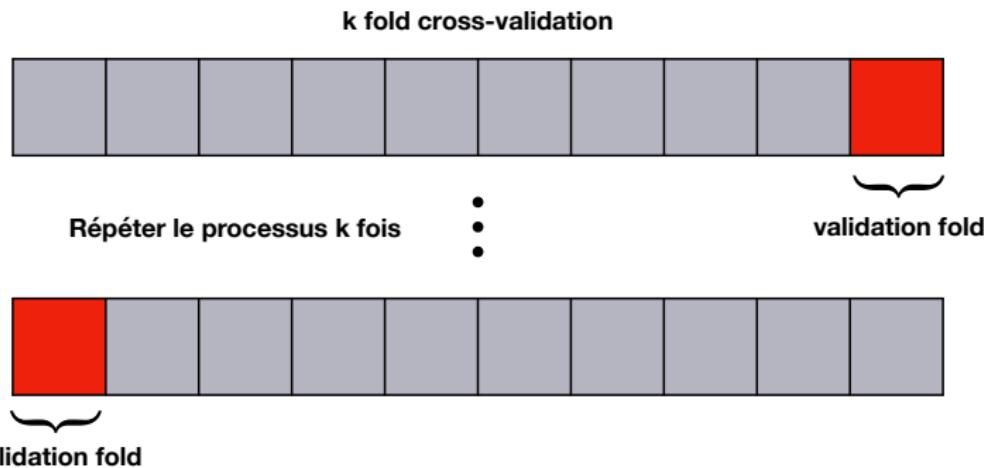
On peut également éviter le risque de **sur-apprentissage** (overfitting) en séparant notre ensemble d'apprentissage en deux (ou plusieurs) échantillons distincts pour valider notre modèle → **création d'un ensemble de validation.**

Cet échantillon va permettre de vérifier les capacités qu'à notre modèle à généraliser.

- **Validation standard** : on sépare notre échantillon d'apprentissage en deux ensembles *train/valid* ( $2/3 - 1/3$ )
- **k-fold cross validation** : partition de l'échantillon en  $k$  ensembles.  $k - 1$  servent à apprendre et 1 sert à la validation
- **Leave and One Out** : on apprend sur tous les exemples sauf 1 qui sert à valider le modèle

# k-fold cross validation

On effectue une série de  $k$  expériences en utilisant une validation classique, mais on change l'apprentissage et la validation à chaque run.

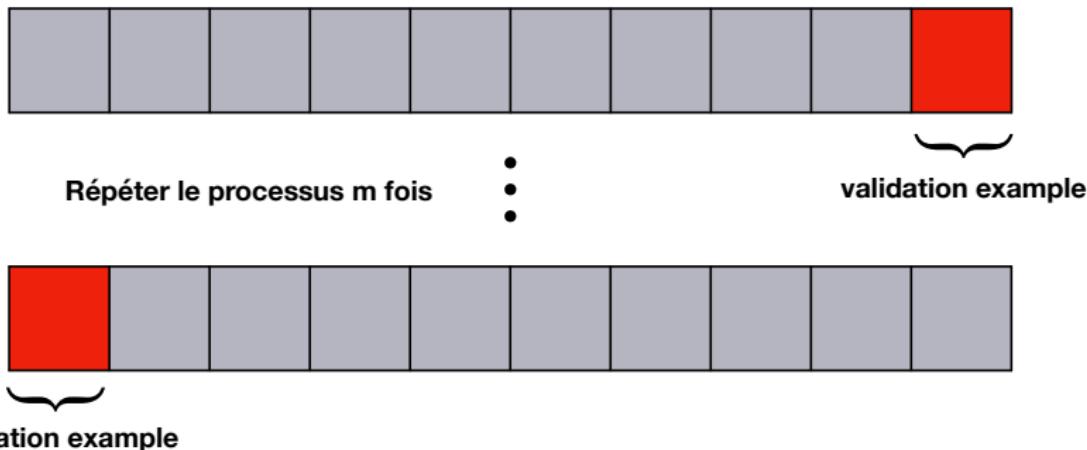


On retient le modèle ayant les meilleurs résultats en moyenne sur les  $k$ -folds qui ont servi à la validation

# Leave and One Out

Le principe reste le même mais on utilise cette fois-ci un seul exemple pour valider à chaque run, on doit effectuer un plus grand nombre d'apprentissage ( $m$  au lieu de  $k$ ).

**Leave and One Out**



On retient le modèle ayant les meilleurs résultats en moyenne sur les  $m$ -folds qui ont servi à la validation

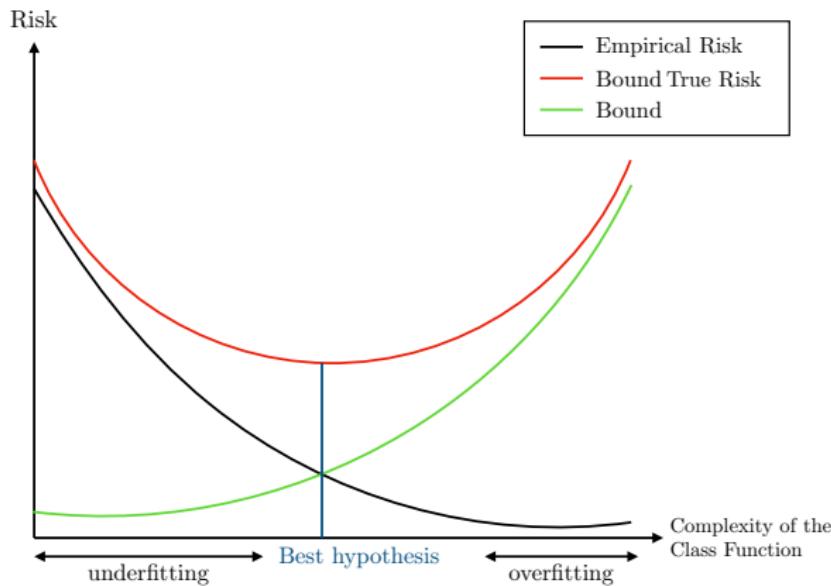
# Quelques remarques

- Pour éviter le problème de sur-apprentissage on utilise à la fois un (ou plusieurs) terme de régularisation dans notre problème d'optimisation
- On combine cela à de la k-fold cross validation (on prendra communément  $k = 10$ ) afin de s'assurer que le modèle généralise bien
- **Attention :** ces approches sont empiriques ! Pour bien faire, il faudrait étudier les garanties en généralisation

$$\left| \mathbb{E}_{(\mathbf{x},y) \sim \mathcal{D}} \ell(h(\mathbf{x}), y) - \mathbb{E}_{(\mathbf{x},y) \sim S} \ell(h(\mathbf{x}), y) \right| \leq \mathcal{O}(\sqrt{1/m}, \lambda^{-1})$$

- Il faut parfois faire la distinction entre le problème d'optimisation et le critère de performance que vous souhaitez optimiser.

# Bornes en généralisation

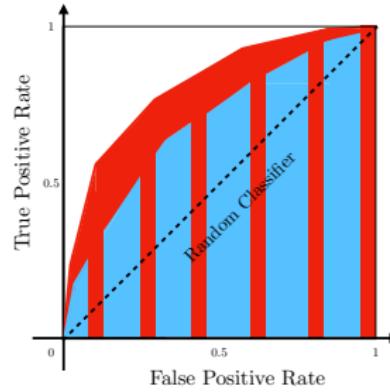
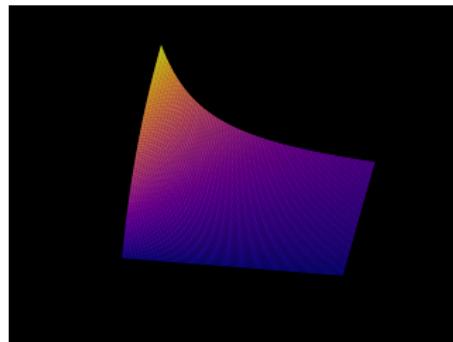


**Pour aller plus loin** (Valiant, 1984; Bousquet and Elisseeff, 2002; Bousquet et al., 2004; Mohri et al., 2012)

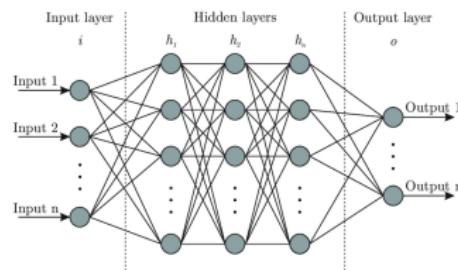
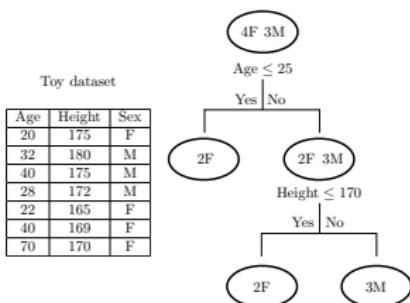
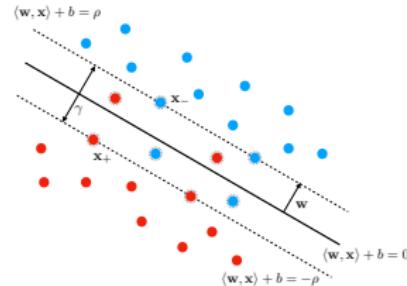
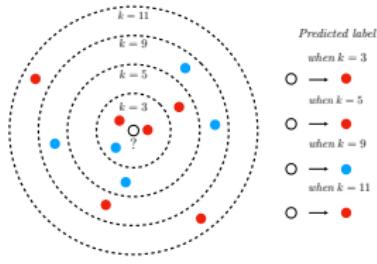
# De nombreux critères de performances

La littérature est pleine de mesures de performances plus adaptées dans certains contextes que dans d'autres :

- **Classification** : Accuracy
- **Recherche d'information** : F-mesure
- **Ranking** : AUC ROC ; P@k ; Recall@k
- **Imbalanced Learning** : Rappel ; Précision ; F-mesure ; G-mesure
- **Regression** : Erreur de reconstruction (square loss)



# De nombreux algorithmes

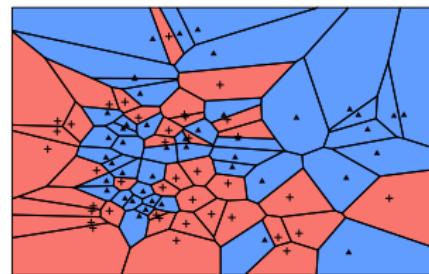
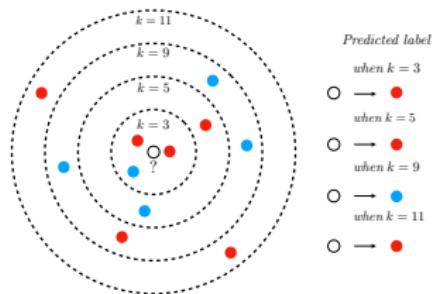


Régression Logistique - Analyse Discriminante - Forêts aléatoires - ...

# Algorithme des plus proches voisins

# Rappels

Il s'agit d'un algorithme non paramétrique pour effectuer des tâches de classification (mais aussi de régression) (Cover and Hart, 1967).



## Exemples $k$ -NN

## Régions de Voronoï, $k = 1$

Il s'agit, en outre, d'un algorithme de classification basé sur la règle de Bayes, i.e. une donnée est prédite comme appartenant à la classe  $c^*$  si

$$c^* = \arg \max_{c \in \mathcal{Y}} p_k(y = c \mid \mathbf{x}).$$

# Rappels

## Règle de classification

La classe assignée à un nouvel exemple  $x$  va dépendre de son positionnement dans l'espace des données par rapport aux données de l'ensemble d'entraînement. Plus précisément, on lui assigne la même étiquette que l'étiquette majoritaire dans son  $k$ -voisinage

$$c^* = \arg \max_{c \in \mathcal{Y}} \frac{k_c}{k},$$

ou  $k_c$  désigne le nombre de voisins appartenant à la classe  $c$ .

- **Apprentissage** : nécessite de garder en mémoire tous les exemples d'entraînement (mémoire en  $\mathcal{O}(m)$ )
- **Prédiction** : nécessite de calculer la distance à tous les exemples d'apprentissages (complexité  $\mathcal{O}(md)$ ) puis d'ordonner les plus proches voisins

# Rappels

Cet algorithme repose donc sur la notion de **distance**. On utilise couramment la distance euclidienne  $\|\mathbf{x} - \mathbf{x}'\|_2 = \sqrt{\sum_{j=1}^d (x_j - x'_j)^2}$ , mais nous pourrions utiliser n'importe quelle norme  $\ell^p$ . Mieux encore ! On pourrait définir sa propre distance → **Metric Learning**

## Remarque

Bien que très simple d'utilisation, cette méthode présente rapidement quelques limites :

- en grande dimension, tous les exemples seront rapidement éloignés (l'espace est très rapidement vide)
- un temps de calcul qui augmente linéairement avec le nombre d'exemples

# Quelques solutions

Il existe des méthodes de pré-traitement qui permettent de réduire le temps de calcul de cet algorithme, en supprimant des données en ou en créant des groupes ou encore en réduisant la dimension.

## Méthodes

- Condensed Nearest Neighbor

---

**Input:** Echantillon d'apprentissage  $S$

**Output:** Un échantillon  $S'$  plus petit que  $S$

**begin**

    Séparer  $S$  en deux ensembles aléatoires  $S_1$  et  $S_2$

**while**  $S_1$  et  $S_2$  sont modifiés **do**

        Retirer de  $S_1$  tous les exemples mal classifiés à l'aide de  $S_2$  et  
        d'un 1-NN

        Retirer de  $S_2$  tous les exemples mal classifiés à l'aide de  $S_1$  et  
        d'un 1-NN

$S' = S_1 \cup S_2;$

**return**  $S'$

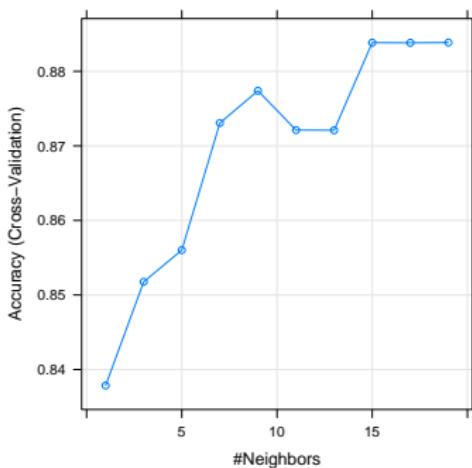
- 
- Clustering : *k-means* , *kd-tree*
  - ACP ou Apprentissage de représentations

# Mise en Pratique

On va mettre en pratique la cross-validation sur l'algorithme  $k$ -NN à l'aide du package "*Caret*" et en précisant le range des valeurs de  $k$  à tester.

```
1 # Chargement des librairies
2 library(class)
3 library(ISLR)
4 library(caret)
5
6 # Chargement du jeu de données
7 data(Smarket)
8
9 # Séparation des données en train / test
10 index_train <- createDataPartition(y=Smarket$Direction, p = 0.75, list=FALSE)
11 train <- Smarket[index_train,]
12 test <- Smarket[-index_train,]
13
14 # Mise en place de la stratégie de tuning
15 train_control <- trainControl(method="cv", number=10)
16 mygrid <- expand.grid(k=seq(1,19,2))
17
18 # Apprentissage / validation du modèle
19 model <- train( Direction ~ . , data=train, trControl=train_control, method="knn", tuneGrid
20   = mygrid)
21 print(model)
22 plot(model)
```

# Mise en pratique

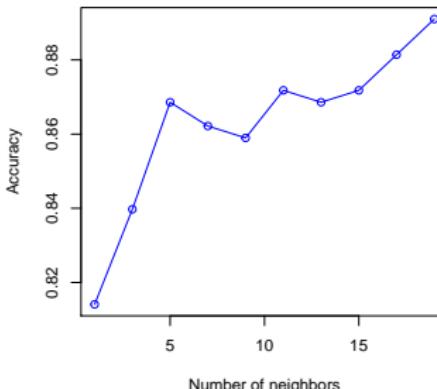


Avec  $k=19$ , les performances en tests sont égales à 0.891

```
1 # Prediction sur l'ensemble test et calcul de l'accuracy
2 knn_prediction <- predict(model,newdata = test )
3 mean(knn_prediction == test$Direction)
4
5 # Si on souhaite tracer la table de confusion
6 confusionMatrix(knn_prediction, test$Direction)
```

# Mise en pratique

```
1 table_test = NULL
2 i=1
3
4 for (j in seq(1,19,2)){
5   res <- knn(train[,-which(colnames(test)== "Direction")],test[,-which(colnames(test)== "Direction")],train[, "Direction"],k=j)
6   table_test[i] = mean(res == test[, "Direction"])
7   i=i+1
8 }
9
10 plot(seq(1,19,2), table_test, xlab = "Number of neighbors" , ylab="Accuracy", pch=1, col
     = "blue", type='o')
```

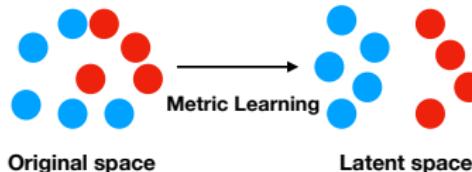


# Pour aller plus loin

L'algorithme  $k$ -NN est un algorithme simple mais qui sert à dans de nombreuses méthodes d'échantillonnages (Fernández et al., 2018) :

- **oversampling** : SMOTE et ses variantes comme Adasyn ou BorderSMOTE
- **undersampling** : Tomek Link - Edited Nearest Neighbor

C'est un algorithme qui également utilisé dans les algorithmes d'apprentissage de métriques (*Metric Learning*), voir exemple de l'algorithme *Large Margin Nearest Neighbor* (Weinberger and Saul, 2009)



# Arbres de Décision et Forêt Aléatoires

# A propos

Les arbres de décisions ont été introduits dans les années 80 (Breiman et al., 1984; Quinlan, 1986) et constituent un puissant algorithme de data mining mais également de classification/régression.

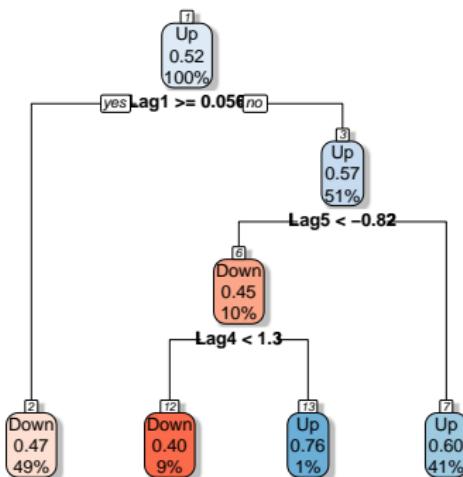
L'objectif de cet algorithme (CART) est de séparer les données en créant des groupes les plus **homogènes** possibles. C'est d'ailleurs cette notion **d'homogénéité** qui va servir de critère d'apprentissage des arbres.

## Quelques avantages

- capables de gérer des données aussi bien numériques que catégoriques
- appropriés sur de très grands jeux de données
- facilement interprétables de par leur construction

# Principe

- A chaque séparation on utilise l'information d'une seule variable pour séparer nos données
- On choisit la variable et la valeur de cette variable qui fournit la meilleure séparation au sens d'un critère défini
- On répète le process jusqu'à obtenir des feuilles *pures*



# Principe

En pratique, on ne construit pas les arbres jusqu'à l'obtention de feuilles pures pour éviter de sur-apprendre nos données. Pour cela, on va jouer sur plusieurs paramètres :

- la profondeur maximal de l'arbre
- le nombre minimum d'exemples dans une feuille
- le nombre minimum d'exemples pour effectuer une séparation
- le gain minimum à chaque séparation

Ce sont autant de paramètres que l'on va devoir valider lors de l'apprentissage d'un modèle !

**Bon ... et quel critère employer pour effectuer nos différentes séparations ?**

# Critères de séparations

- **L'entropie** (mesure du désordre)  $S$ , utilisé dans l'algorithme C4.5

$$S = - \sum_{j=1}^C \frac{m_j}{m} \log \left( \frac{m_j}{m} \right).$$

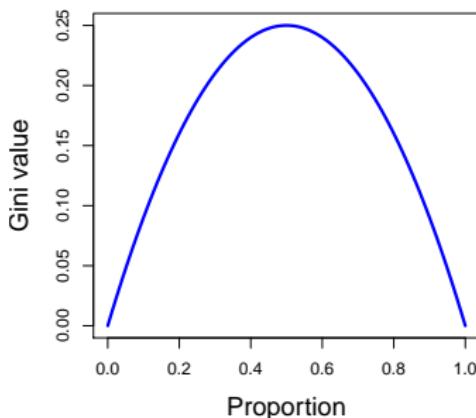
- **L'indice de Gini**  $D$ , utilisé dans l'algorithme **CART**

$$D = \sum_{j=1}^C \frac{m_j}{m} \left( 1 - \frac{m_j}{m} \right).$$

- La **variance**  $V$  dans le cas de la régression

$$V = \frac{1}{m} \sum_{i=1}^m (x_i - \bar{x})^2.$$

# Critères de séparations



On choisit ensuite le couple  $\theta^* = (\text{variable}, \text{valeur})$  tel que

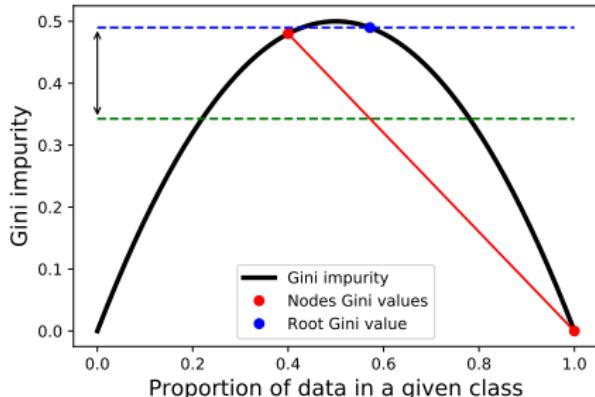
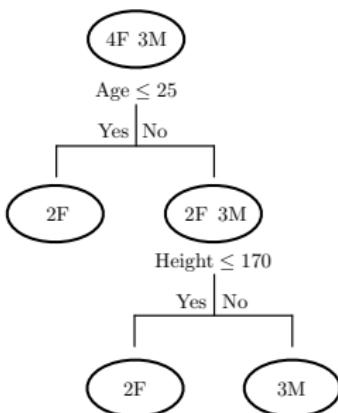
$$\theta^* = \arg \max_{\theta} \frac{m_{\text{left}}}{m} \Gamma_{\text{left}}(\theta) + \frac{m_{\text{right}}}{m} \Gamma_{\text{right}}(\theta),$$

où  $\Gamma$  est l'un des critère précédemment mentionné.

# Exemple

Toy dataset

Age	Height	Sex
20	175	F
32	180	M
40	175	M
28	172	M
22	165	F
40	169	F
70	170	F



**Exercice :** Calculer le gain d'information de la première séparation en prenant comme critère l'indice de Gini où le gain est défini par

$$\Gamma_{\text{node}} = \frac{m_{\text{left}}}{m} \Gamma_{\text{left}} + \frac{m_{\text{right}}}{m} \Gamma_{\text{right}}.$$

# Prédiction

Quelle valeur affecter à chaque feuille ?

## Règles courantes

Les règles couramment utilisées sont

- en classification : attribuer l'étiquette de la classe majoritairement présente dans la feuille
- en régression : attribuer à la feuille la valeur moyenne de la feature à déterminer

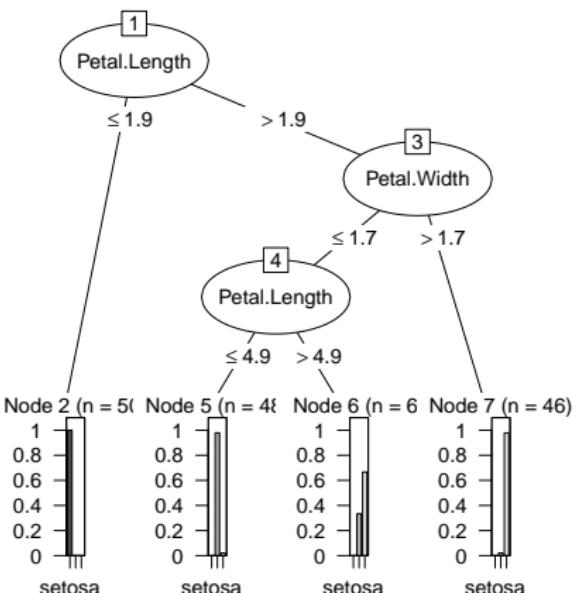
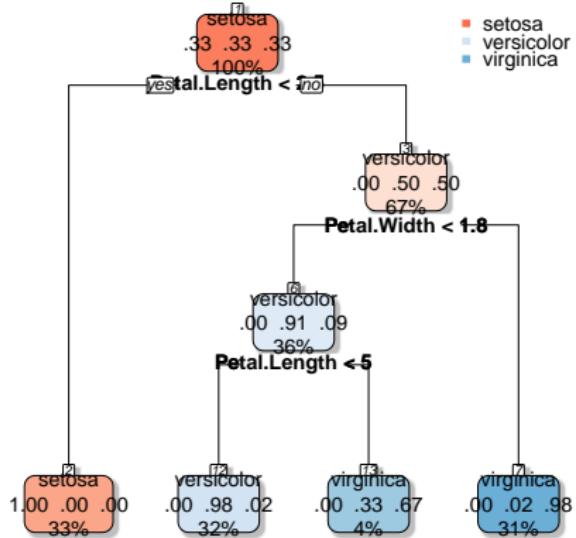
On peut aussi faire des règles plus complexes, cela peut arriver lorsque toutes les données n'ont pas le même poids.

# Mise en pratique

Plusieurs librairies sont possibles sous R: C50 et rpart

```
1 # Jeu de données
2 data(iris)
3
4 # A) Utilisation de l'entropie de Shannon
5
6 # Librairie C50 : uniquement pour la classification
7 library(C50)
8 list_control = C5.0Control(minCases=5)
9 # Apprentissage du modèle
10 treeModel <- C5.0(Species ~ ., data=iris, rules =FALSE)
11 # Affichage des résultats
12 summary(treeModel)
13 # Affichage graphique
14 plot(treeModel)
15
16 # B) Utilisation l'indice de Gini (classification) ou la variance (régression)
17
18 # Librairie rpart : classification et régression
19 library(rpart)
20 library(rpart.plot)
21 # Contrôle des paramètres de l'arbre
22 ctrl = rpart.control(maxdepth=5,
23                      minsplit = 20,
24                      minbucket = 5)
25 # Apprentissage du modèle
26 tree <- rpart(Species ~ ., data=iris, control = ctrl)
27 # Visualisation de l'arbre
28 rpart.plot(tree, box.palette="RdBu", shadow.col="gray", nn=TRUE)
```

# Mise en pratique



# Mise en pratique

Faire de la cross-validation avec *rpart* avec le package *caret*

```
1 library(caret)
2 library(rpart)
3
4 data(iris)
5
6 index_train <- createDataPartition(y=iris$Species, p = 0.75, list=FALSE)
7 train <- iris[index_train,]
8 test <- iris[-index_train,]
9
10 train_control <- trainControl(method="cv", number=10)
11 mygrid <- expand.grid(cp=seq(0,1,0.01))
12 model <- train( Species~ . ,data=train, trControl=train_control, method="rpart", tuneGrid
13   = mygrid)
14 model
15 plot(model)
16
17 # Prediction sur l'ensemble test et calcul de l'accuracy
18 tree_prediction <- predict(model,newdata = test )
19 mean(tree_prediction == test$Species)
20
21 # Si on souhaite tracer la table de confusion
22 confusionMatrix(tree_prediction, test$Species )
```

On peut aussi le faire manuellement à l'aide de boucles *for* voire *foreach*

# Quelques remarques

Les arbres de décisions (classification ou régression)

- sont des modèles simples à mettre en œuvre
- sont facilement interprétables, on identifie facilement les raisons pour lesquelles une donnée est prédite dans telle ou telle classe
- peuvent être facilement parallélisables dans leur construction
- capables de créer des modèles non linéaires à l'aide de simples "hyperplans"

Mais ils présentent quelques inconvénients

- une forte instabilité : une perturbation de l'échantillon d'apprentissage peut changer la structure de l'arbre
- un risque de sur-apprentissage très important (modèle à trop faible variance)

# Combinaisons de modèles

Comme souvent en Machine Learning, il est d'usage de combiner plusieurs modèles ensembles pour en augmenter leurs performances et surtout leur stabilité !

C'est ce que l'on va faire avec les arbres de décisions, en créant plusieurs arbres à partir du plusieurs échantillonnages de notre ensemble d'apprentissage.

→ C'est le principe du bagging ou bootstrap aggregating

# Bagging

## Principe

Si on souhaite construire un ensemble de  $T$  modèles on doit alors:

- tirer  $T$  échantillons de taille  $m$  (ou moins) avec remise
- apprendre un modèle à l'aide de chaque échantillon
- déterminer l'étiquette (ou la valeur) d'une nouvelle donnée à l'aide de la combinaison des différents modèles : le choix de la règle de combinaison est libre ! Le plus souvent on pratique le *vote de majorité* (petit clin d'oeil à la théorie PAC-Bayes).

**Objectif :** créer des modèles suffisamment divers afin de créer un modèle plus puissant en les combinant. Plus on a de diversité, plus on gagne en stabilité.

→ **on peut même faire de l'échantillonnage sur les variables !**

# Bagging

Un exemple d'application du bagging sur des arbres de décisions à l'aide de la librairie adabag.

```
1 library(adabag)
2 library(rpart.plot)
3 data(iris)
4
5 index <- sample(1:nrow(iris), size = round(nrow(iris)*2/3))
6 model_bag <- bagging(Species~, data = iris[index,], mfinal =5)
7 pred <- predict.bagging(model_bag, newdata = iris[-index,])
8
9 model_bag$trees    # donne l'accès aux arbres
10 rpart.plot(model_bag$trees[[1]]) # plot d'un des arbres
11 model_bag$trees    # explique l'importance de chaque variable dans les performances
12 model_bag$votes    # donne les détails du vote
13
14 pred$confusion    # pour avoir la matrice de confusion
```

Mais comme énoncé avant, on peut aussi faire de l'échantillonnage sur les variables, c'est le principe même des forêts aléatoires (Breiman, 2001).

# Forêts aléatoires

Un petit exemple en pratique, très semblable à un arbre de décision.

```
1 library(randomForest)
2
3 # mtry indique combien de variables sont testées à chaque noeud d'un arbre.
4 # ntree indique le nombre d'arbres
5 # on peut ensuite définir des options similaires à celles d'un arbre : profondeur -
  nombre d'exemples, ...
6 model_rf <- randomForest(Species~, data = iris, ntree = 20, mtry = 2)
7
8 model_rf$confusion # matrice de confusion sur les données non utilisées pour construire
  un arbre
9 model_rf$oob_times # nombre de fois où un exemple est utilisé pour calculer l'erreur OOB
```

L'avantage est que cela permet également de déterminer quels sont les variables les plus discriminantes dans la classification des exemples (s'il y en a !).

```
1 library(randomForest)
2 varImpPlot(model_rf)
3 # ou
4 model_rf$importance
```

# Pour finir

- Même si les forêts aléatoires sont des algorithmes intéressants par la création de plusieurs arbres divers, ces derniers sont construits indépendamment les uns des autres et cela peut avoir un impact sur les performances.
- Il existe un autre algorithme très efficace dans la pratique qui utilise le principe de combinaison de modèles. Ce dernier repose sur l'utilisation du *Boosting* et des modèles à base d'arbres → **Gradient Tree Boosting**

Ils sont plus **rapides** et plus **efficaces** !

Mais nous verrons cela au cours d'un TD

# References I

- Bousquet, O., Boucheron, S., and Lugosi, G. (2004). *Introduction to Statistical Learning Theory*, pages 169–207. Springer Berlin Heidelberg.
- Bousquet, O. and Elisseeff, A. (2002). Stability and generalization. *Journal of Machine Learning Research*, 2:499–526.
- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.
- Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and regression trees*. The Wadsworth statistics/probability series. Wadsworth and Brooks/Cole Advanced Books and Software, Monterey, CA.
- Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27.
- Fernández, A., García, S., Herrera, F., and Chawla, N. V. (2018). Smote for learning from imbalanced data: Progress and challenges, marking the 15-year anniversary. *Journal of Artificial Intelligence Research*, 61:863–905.
- Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2012). *Foundations of Machine Learning*. The MIT Press.
- Quinlan, J. R. (1986). Induction of decision trees. *Mach. Learn.*, 1(1):81–106.
- Valiant, L. G. (1984). A theory of the learnable. *Communication of the ACM*, 27(11):1134–1142.
- Weinberger, K. Q. and Saul, L. K. (2009). Distance metric learning for large margin nearest neighbor classification. *Journal of Machine Learning Research*, 10:207–244.