

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 :

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

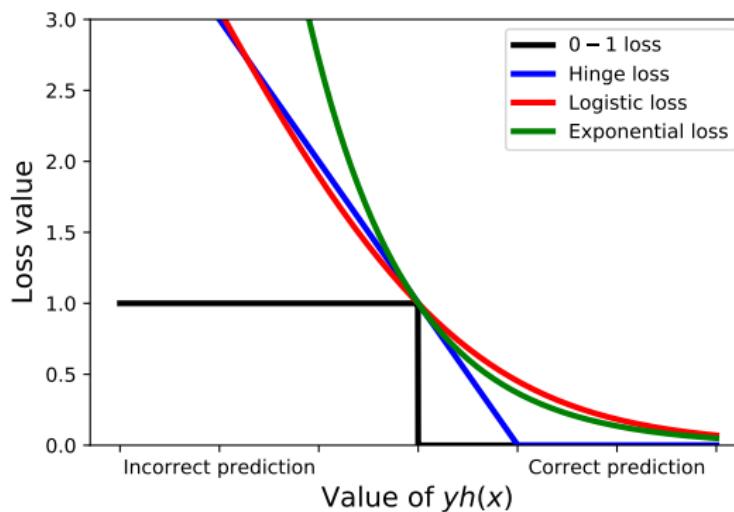
On souhaite donc trouver une hypothèse h qui soit performante sur notre échantillon S mais aussi sur tout nouvel exemple (\mathbf{x}, y) .

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 :

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

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 ?

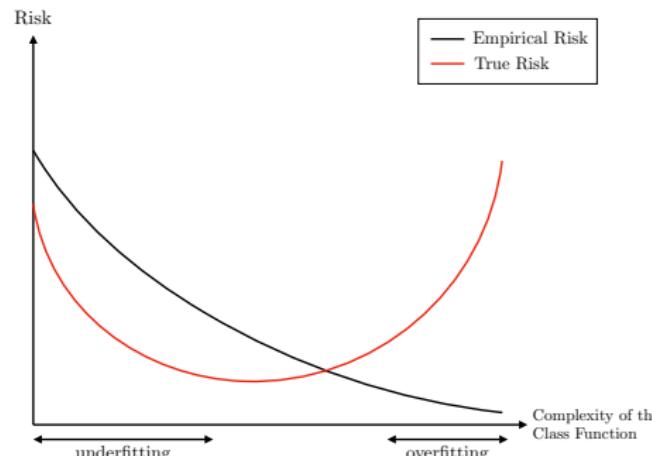
Il faut restreindre le spectre des modèles que l'on s'autorise à apprendre, i.e. diminuer la richesse ou la complexité de notre espace d'hypothèses \mathcal{H} .

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 λ : paramètre de régularisation, contrôle la complexité de \mathcal{H} .



Validation croisée

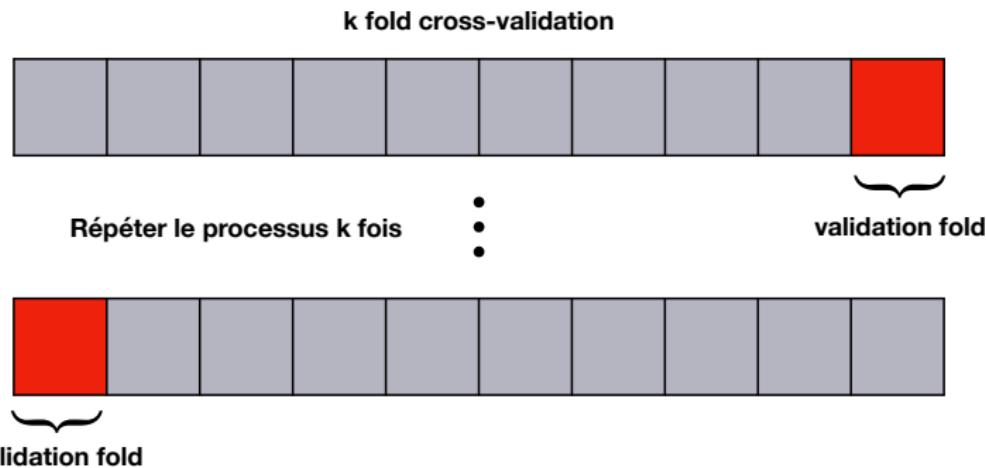
On peut également éviter le risque de **sur-apprentissage** (overfitting) en séparant notre ensemble d'apprentissage en deux (ou plusieurs) échantillons distincts pour valider notre modèle → **création d'un ensemble de validation.**

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.

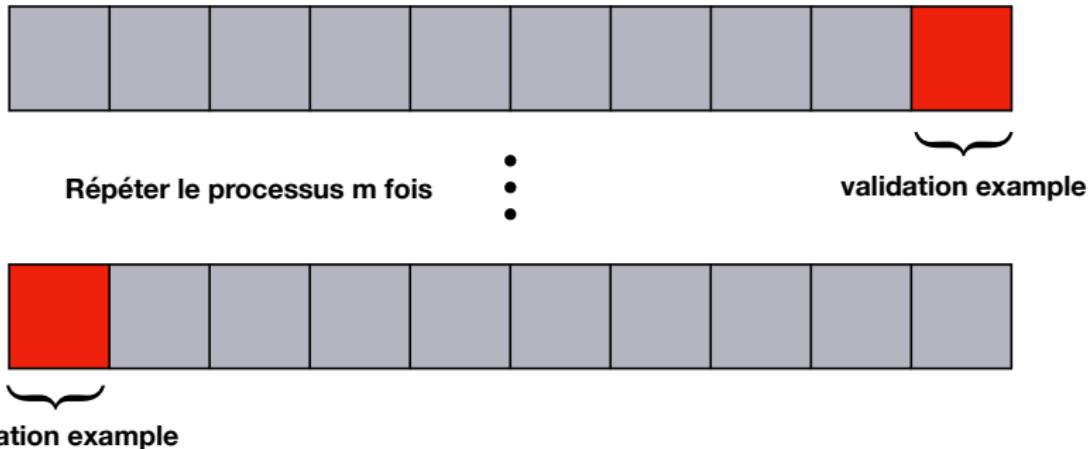


On retient le modèle ayant les meilleurs résultats en moyenne sur les k -folds qui ont servi à la validation

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

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

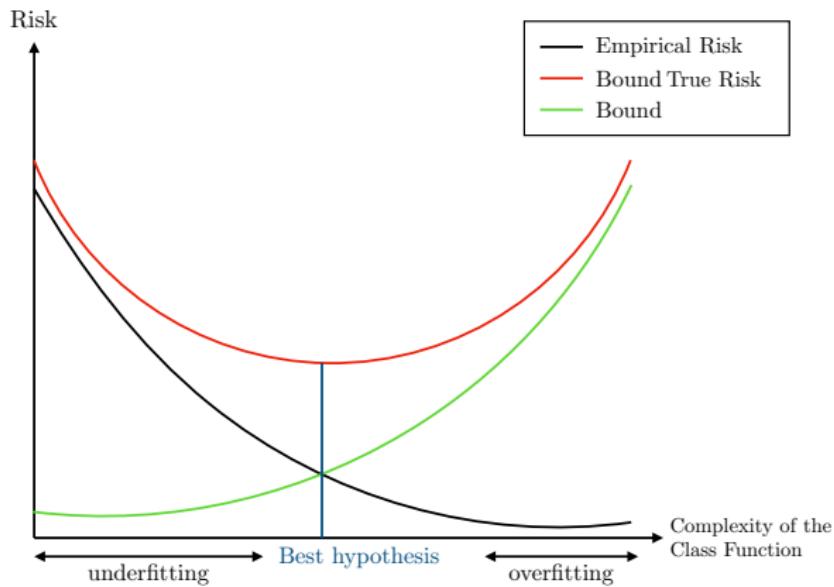
- Il faut parfois faire la distinction entre le problème d'optimisation et le critère de performance que vous souhaitez optimiser.

Bornes en généralisation

Les bornes en généralisation reposent sur :

- des inégalités de concentration, comme l'inégalité de Azuma-Hoeffding (nous verrons cela au cours d'un TD)
- des propriétés de la loss : convexité, caractère lipschitz ou α -elliptique,
...
- la complexité de notre espace d'hypothèses \mathcal{H} qui peut se traduire par le choix de l'hyper-paramètre dans notre terme de régularisation ou des mesures de complexité : **VC-dim** ou **Complexité de Rademacher**
- une mesure de divergence entre une distribution *a priori* et *a posteriori* de nos hypothèses → **KL-divergence**

Bornes en généralisation

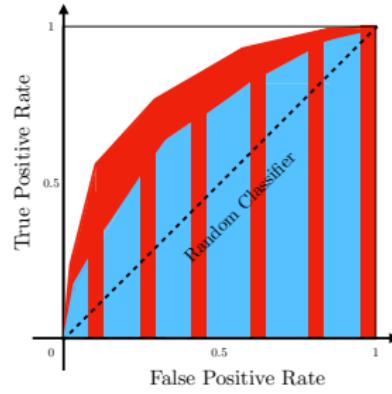
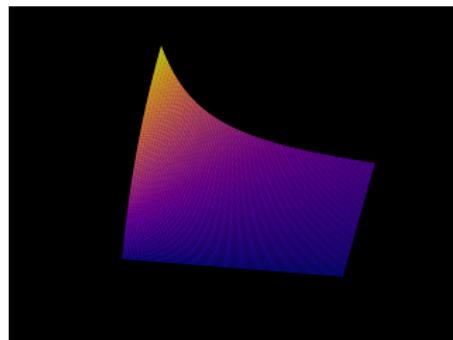


Pour aller plus loin (Valiant, 1984; Bousquet and Elisseeff, 2002; Bousquet et al., 2004; Mohri et al., 2012)

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 critères de performances

		Réalité		
			Positive	Négative
Prediction				
Positive			TP = Vrai positif	FP = Faux positif
Negative			FN = Faux négatif	TN = Vrai négatif

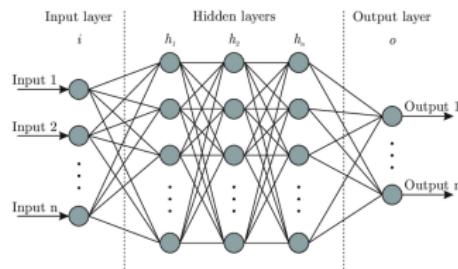
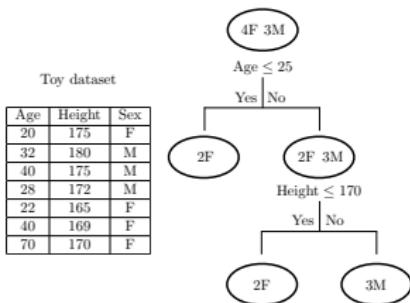
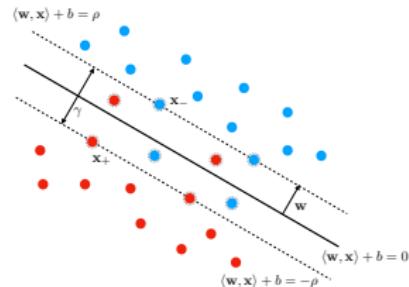
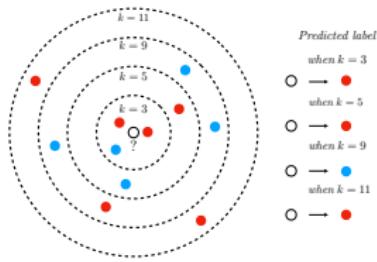
- **Classification :** Accuracy = $\frac{TP + TN}{FP + FN + TP + TN}$, c'est aussi le taux de bonne classification.
- **Recherche d'information :** F-mesure elle dépend d'un paramètre β qui va permettre d'accorder plus d'importance au *rappel* ou à la *précision* de votre modèle.

$$\text{Rappel} = \frac{TP}{TP + FN} \quad \text{et} \quad \text{Precision} = \frac{TP}{TP + FP}.$$

On définit alors la F-mesure par

$$F_\beta = \frac{(1 + \beta^2) \times (\text{Precision} \times \text{Rappel})}{\beta^2 \text{Precision} + \text{Rappel}} = \frac{(1 + \beta^2)TP}{\beta^2 \text{Precision} + TP}$$

De nombreux algorithmes

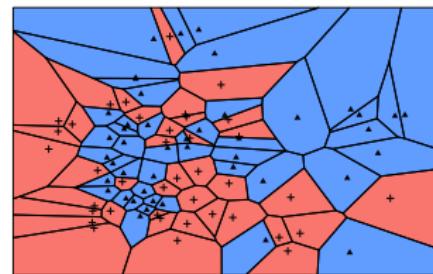
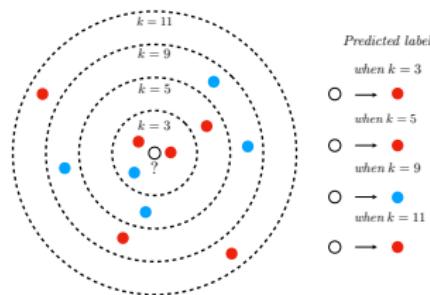


Régression Logistique - Analyse Discriminante - Forêts aléatoires - ...

Algorithme des plus proches voisins

k -NN

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

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

k -NN

Règle de classification

La classe assignée à un nouvel exemple \mathbf{x}_i 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

k -NN

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 ℓ^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'

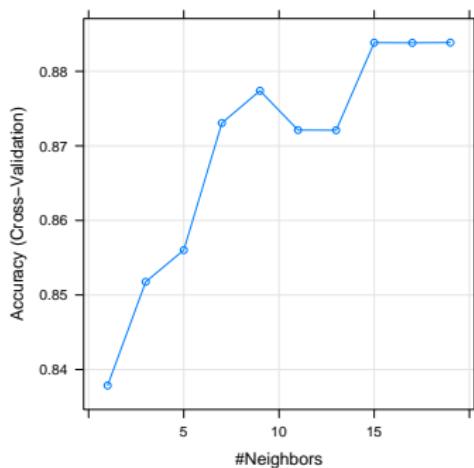
-
- Clustering : *k-means* , *kd-tree*
 - ACP ou Apprentissage de représentations

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 donnees
7 data(Smarket)
8
9 # Separation des donnees 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 strategie de tuning
15 train_control <- trainControl(method="cv", number=10)
16 mygrid <- expand.grid(k=seq(1,19,2))
17
18 # Apprentissage / validation du modele
19 model <- train( Direction~ . ,data=train, trControl=train_control, method="knn", tuneGrid
20   = mygrid)
21 print(model)
22 plot(model)
```

Mise en pratique

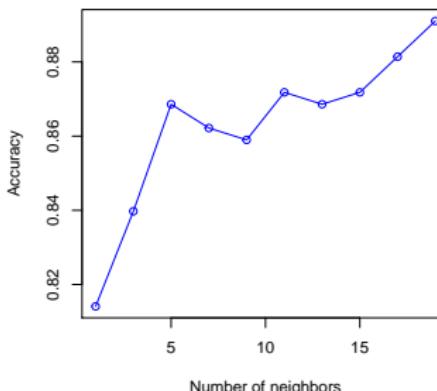


Avec $k=19$, les performances en tests sont égales à 0.891

```
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')
```

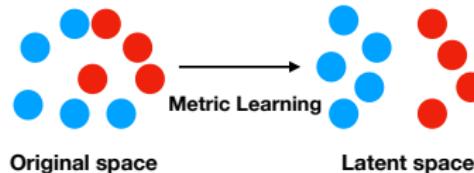


Pour aller plus loin

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)



Apprentissage de Métrique

Idée : étendre la notion de distance à des notions autres que l'usage des normes ℓ_p . Fondée sur la notion de distance de *Mahalanobis* :

$$d_{\Sigma}^2(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i - \mathbf{x}_j)^T \Sigma^{-1} (\mathbf{x}_i - \mathbf{x}_j),$$

où Σ est la matrice de variance-covariance des données, elles donc **PSD** : semi-définie positive, ce qui montre d_{Σ} définie bien une distance.

On souhaite maintenant apprendre une matrice \mathbf{M} qui pourrait répondre à une tâche particulière en l'utilisant dans une notion de distance $d_{\mathbf{M}}$

$$d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i - \mathbf{x}_j)^T \mathbf{M} (\mathbf{x}_i - \mathbf{x}_j),$$

où la matrice $\mathbf{M} \in \mathbb{R}^{d \times d}$ est une matrice *PSD*.

Apprentissage de Métrique

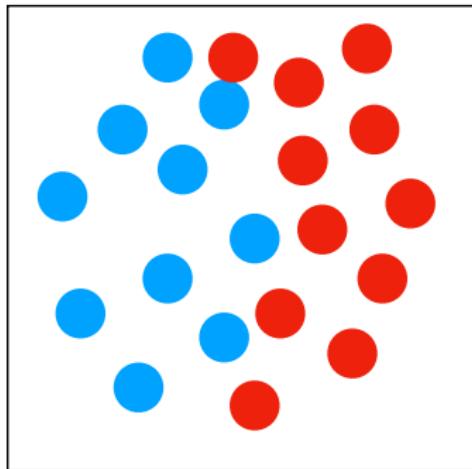
Comme \mathbf{M} est PSD, on peut également la décomposer en $\mathbf{M} = \mathbf{L}^T \mathbf{L}$, ce qui nous donne :

$$\begin{aligned} d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) &= (\mathbf{x}_i - \mathbf{x}_j)^T \mathbf{M} (\mathbf{x}_i - \mathbf{x}_j), \\ &= (\mathbf{x}_i - \mathbf{x}_j)^T \mathbf{L}^T \mathbf{L} (\mathbf{x}_i - \mathbf{x}_j), \\ &= (\mathbf{L}\mathbf{x}_i - \mathbf{L}\mathbf{x}_j)^T (\mathbf{L}\mathbf{x}_i - \mathbf{L}\mathbf{x}_j), \end{aligned}$$

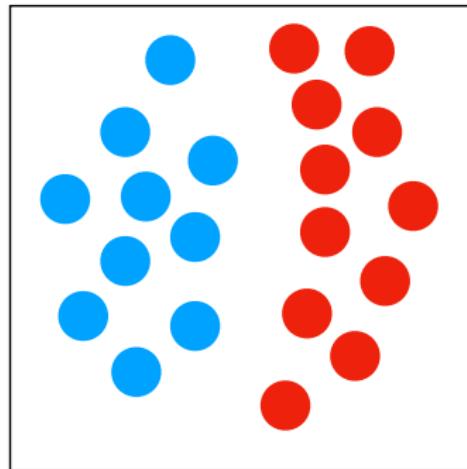
- \mathbf{L} : matrice de projection → norme euclidienne dans un nouvel espace de représentation des données
- $\mathbf{L} \in \mathbb{R}^{k \times d}$: potentiellement de rang inférieur à d → réduction de dimension
- apprendre \mathbf{M} pour des tâches particulières

Apprentissage de Métrique

Apprentissage de représentation pour une tâche de classification



Espace d'origine



Espace après projection

Apprentissage de Métrique

Large Margin Nearest Neighbor (LMNN) (Weinberger and Saul, 2009).

Principe : apprendre une représentation, i.e. \mathbf{M} pour améliorer les performances de k -NN en utilisant des paires d'exemples *similaires ou différentes*

$$\mathcal{S} = \{(\mathbf{x}_i, \mathbf{x}_j) \mid y_i = y_j\} \text{ et } \mathcal{D} = \{(\mathbf{x}_i, \mathbf{x}_j) \mid y_i \neq y_j\}$$

- **Contrainte forte :**

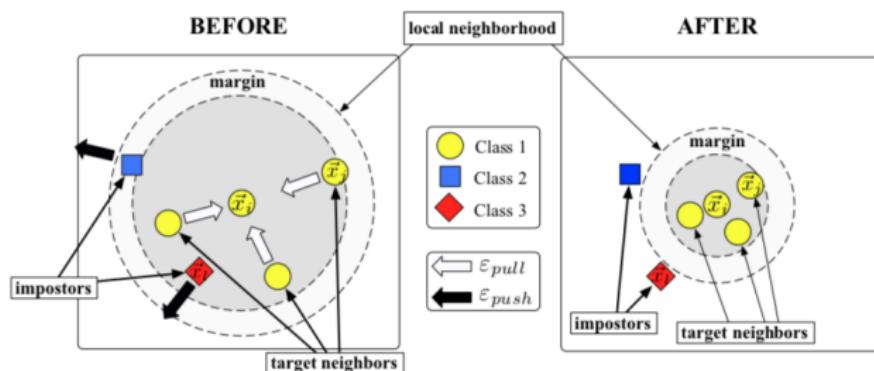
$$d_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_j) \leq \alpha, \forall (\mathbf{x}_i, \mathbf{x}_j) \in \mathcal{S}$$

- **Contrainte relatives**

$$d_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_j) \leq d_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_k) - m, \forall (\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k) \in \mathcal{R}$$

Apprentissage de Métrique

Large Margin Nearest Neighbor (LMNN) (Weinberger and Saul, 2009).



$$\min_{\mathbf{M}, \xi} \quad (1 - \mu) \sum_{(i,j) \in \mathcal{S}} (\mathbf{x}_i - \mathbf{x}_j)^T \mathbf{M} (\mathbf{x}_i - \mathbf{x}_j) + \mu \sum_{(i,j) \in \mathcal{S}, (i,k)} (1 - y_{ik}) \xi_{ijk}$$

$$s.t. \quad (\mathbf{x}_i - \mathbf{x}_j)^T \mathbf{M} (\mathbf{x}_i - \mathbf{x}_k) - (\mathbf{x}_i - \mathbf{x}_j)^T \mathbf{M} (\mathbf{x}_i - \mathbf{x}_j) \geq m - \xi_{ijk}$$

$$\xi_{ijk} \geq 0$$

$$\mathbf{M} \text{ PSD}$$

Apprentissage de Métrique

Comparaison entre 3–NN et d'autres algorithmes de classification, en terme de F -mesure.

DATASETS	SIZE	DIM	IR	3NN	LMNN	ITML	IML	IMBML	MLFP (OURS)
BALANCE	625	4	1.2	0.880 ± 0.018	0.874 ± 0.019	0.931 ± 0.032	0.886 ± 0.029	0.960 ± 0.019	0.874 ± 0.003
AUTOMPG	392	7	1.7	0.780 ± 0.054	0.792 ± 0.031	0.801 ± 0.018	0.785 ± 0.021	0.790 ± 0.044	0.805 ± 0.021
IONOSPHERE	351	34,	1.8	0.745 ± 0.015	0.803 ± 0.049	0.831 ± 0.054	0.823 ± 0.044	0.786 ± 0.053	0.923 ± 0.026
PIMA	768	8	1.9	0.601 ± 0.042	0.591 ± 0.037	0.583 ± 0.022	0.591 ± 0.037	0.575 ± 0.026	0.635 ± 0.032
WINE	178	13	2	0.968 ± 0.016	0.992 ± 0.016	0.992 ± 0.016	0.992 ± 0.016	0.992 ± 0.016	0.961 ± 0.041
GLASS	214	9	2.1	0.735 ± 0.049	0.710 ± 0.064	0.759 ± 0.051	0.710 ± 0.064	0.716 ± 0.043	0.747 ± 0.034
GERMAN	1000	23	2.3	0.407 ± 0.049	0.358 ± 0.029	0.430 ± 0.073	0.352 ± 0.029	0.388 ± 0.043	0.511 ± 0.006
VEHICLE	846	18	3.3	0.850 ± 0.045	0.928 ± 0.024	0.931 ± 0.019	0.933 ± 0.026	0.937 ± 0.014	0.859 ± 0.037
HAYES	132	4	3.4	0.581 ± 0.210	0.824 ± 0.089	0.829 ± 0.071	0.824 ± 0.089	0.908 ± 0.083	0.930 ± 0.109
SEGMENTATION	2310	19	6	0.882 ± 0.031	0.888 ± 0.011	0.866 ± 0.029	0.895 ± 0.020	0.909 ± 0.028	0.882 ± 0.024
ABALONE8	4177	10	6.4	0.223 ± 0.025	0.220 ± 0.040	0.213 ± 0.025	0.228 ± 0.021	0.200 ± 0.023	0.336 ± 0.018
YEAST3	1484	8	8.1	0.719 ± 0.028	0.734 ± 0.020	0.742 ± 0.034	0.717 ± 0.032	0.723 ± 0.023	0.725 ± 0.022
PAGEBLOCKS	5473	10	8.8	0.855 ± 0.027	0.844 ± 0.027	0.850 ± 0.023	0.842 ± 0.027	0.865 ± 0.021	0.860 ± 0.022
SATIMAGE	6435	36	9.3	0.688 ± 0.034	0.707 ± 0.038	0.710 ± 0.024	0.710 ± 0.039	0.731 ± 0.030	0.697 ± 0.030
LIBRAS	360	90	14	0.694 ± 0.188	0.725 ± 0.105	0.722 ± 0.204	0.690 ± 0.120	0.729 ± 0.157	0.694 ± 0.066
REDWINEQUALITY4	1599	11	29.2	0.062 ± 0.075	0.057 ± 0.114	0.027 ± 0.053	0.000 ± 0.000	0.031 ± 0.062	0.083 ± 0.039
YEAST6	1484	8	41.4	0.560 ± 0.205	0.578 ± 0.246	0.523 ± 0.205	0.629 ± 0.244	0.606 ± 0.148	0.527 ± 0.152
ABALONE17	4177	10	71	0.000 ± 0.000	0.000 ± 0.000	0.029 ± 0.057	0.000 ± 0.000	0.073 ± 0.000	0.053 ± 0.033
ABALONE20	4177	10	159.7	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000	0.044 ± 0.089	0.000 ± 0.093	0.078 ± 0.029
MEAN				0.591	0.612	0.619	0.613	0.627	0.643

Apprentissage de Métrique

Quelques régularisations intéressantes

- Utiliser la norme $L_{2,1}$: $\|\mathbf{M}\|_{2,1} = \sum_{j=1}^d \|\mathbf{M}_j\|_2$
 Tends à annuler des colonnes entières → sélection de variables
 (Ying et al., 2009)
 Problème convexe mais non lisse à cause de la norme ℓ_1 , on peut cependant utiliser des méthodes de **gradient proximal** pour résoudre ce type de problème
- Utiliser la norme trace, nucléaire : $\|\mathbf{M}\|_\sigma = \sum_{j=1}^d \sigma_j(\mathbf{M})$
 Favorise les matrices de rang faible → réduction de la dimension (McFee and Lanckriet, 2010)
 Problème à nouveau convexe mais non lisse, utilisation d'algorithmes de type *Frank-Wolfe* (Jaggi, 2013).

Apprentissage de Métrique

On peut l'utiliser pour faire de la

- réduction de dimension
- sélection de variable (en jouant sur le terme de régularisation)

Des applications en

- Computer Vision
- Classification (Imbalanced ou non)

Méthode qui peut être étendue à des versions non linéaires : *Kernel Metric Learning* ou encore *Deep Metric Learning*.

Vous pouvez consulter plusieurs présentations/tutoriels sur le Metric Learning sur le site de **Aurélien Bellet**.

Pas de package sous  mais vous trouverez votre bonheur sous Python.

Support Vector Machines

A propos des SVM

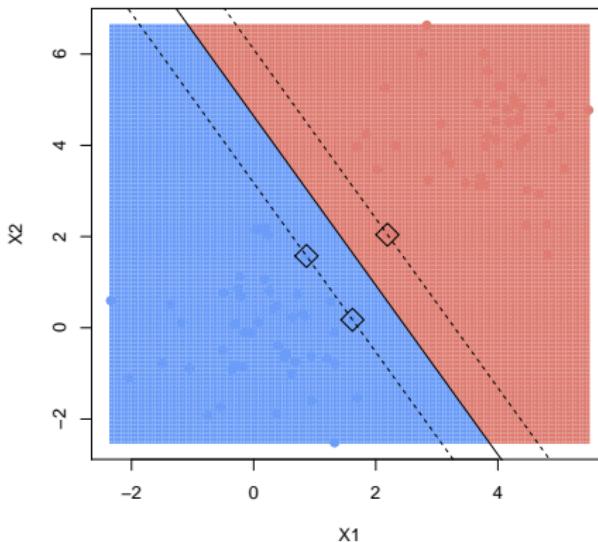
Certainement l'un des algorithmes les plus répandus en apprentissage machine (Vapnik and Cortes, 1995) pour effectuer des tâches de classification binaires.

Idée : apprendre une hypothèse h dont le but est de prédire l'étiquette d'une donnée. Elle se présente sous la forme d'un hyperplan (affine) qui va séparer l'espace en deux : $\{-1, +1\}$:

$$h(\mathbf{x}) = \text{sign}[\langle \mathbf{w}, \mathbf{x} \rangle + b] = \begin{cases} -1 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b < 0, \\ +1 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b > 0. \end{cases}$$

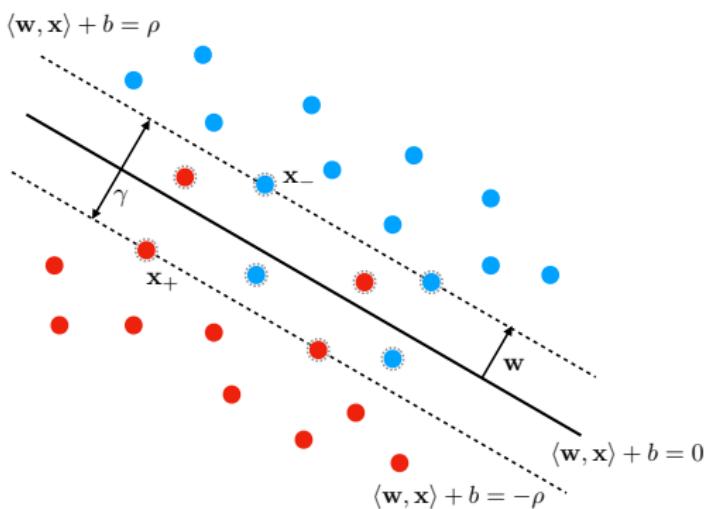
Problème : plusieurs plans peuvent être solutions et séparer parfaitement nos données.

A propos des SVM



→ une solution consiste à prendre le SVM qui présente la plus grande marge (Boser et al., 1992), c'est celui qui généralisera le mieux !

Marge et définition d'un SVM



La marge γ est la distance entre les deux hyperplans d'équations

$$\langle \mathbf{w}, \mathbf{x} \rangle + b = \pm \rho.$$

On va maintenant voir comment la marge γ est liée à notre hyperplan défini par \mathbf{w} .

Marge et définition d'un SVM

On utilisera la décomposition suivante $\mathbf{x} = \mathbf{x}^w + \mathbf{x}^{w^\perp}$ et on prend deux points \mathbf{x}_+ et \mathbf{x}_- se trouvant sur la marge

$$\begin{aligned} h(\mathbf{x}_+) - h(\mathbf{x}_-) &= 2, \\ \langle \mathbf{w}, \mathbf{x}_+ \rangle + b - (\langle \mathbf{w}, \mathbf{x}_- \rangle + b) &= 2, \\ \underbrace{\langle \mathbf{w}, \mathbf{x}_+^w \rangle}_{=0} + \underbrace{\langle \mathbf{w}, \mathbf{x}_+^{w^\perp} \rangle}_{=0} + b - (\underbrace{\langle \mathbf{w}, \mathbf{x}_-^w \rangle}_{=0} + \underbrace{\langle \mathbf{w}, \mathbf{x}_-^{w^\perp} \rangle}_{=0} + b) &= 2, \\ \langle \mathbf{w}, \mathbf{x}_+^w \rangle - \langle \mathbf{w}, \mathbf{x}_-^w \rangle &= 2, \\ 2\langle \mathbf{w}, \mathbf{x}_+^w \rangle &= 2. \end{aligned}$$

En prenant la norme de chaque côté :

$$\gamma = 2\|\mathbf{x}_+^w\|_2 = \frac{2}{\|\mathbf{w}\|_2}.$$

Marge et définition d'un SVM

Maximiser la marge \Leftrightarrow minimiser $\|\mathbf{w}\|$

On cherche donc l'hyperplan séparateur qui vérifie le problème d'optimisation suivant :

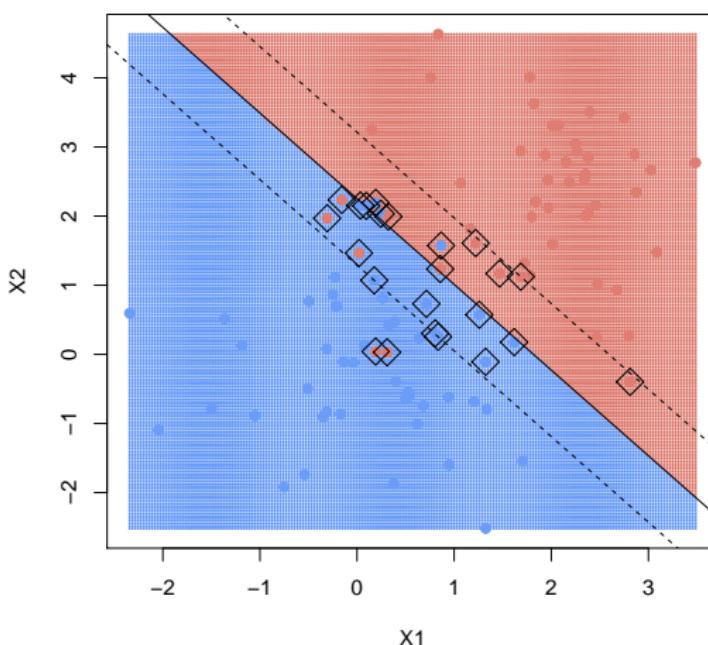
$$\begin{aligned} \min_{(\mathbf{w}, b) \in \mathbb{R}^{d+1}} \quad & \frac{1}{2} \|\mathbf{w}\|^2 \\ \text{s.t.} \quad & y_i (\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1, \quad \text{for all } i = 1, \dots, m. \end{aligned}$$

→ Formulation connue sous le nom de **Hard Margin SVM**

Problème : les problèmes ne sont que très rarement linéairement séparables ...

Limite du Hard Margin SVM

Un SVM en 2D



... il faut donc savoir relâcher les contraintes de notre classifieur ...

Soft Margin SVM

On commence par relâcher nos contraintes en autorisant des données d'une certaine étiquette à se trouver du mauvais côté de l'hyperplan

→ **introduction de variables slacks** ξ_i que l'on va inclure dans notre problème d'optimisation :

$$\begin{aligned} \min_{\xi \in \mathbb{R}^m, (\mathbf{w}, b) \in \mathbb{R}^{d+1}} \quad & \frac{1}{2} \|\mathbf{w}\|_2^2 + \frac{C}{m} \sum_{i=1}^m \xi_i \\ \text{s.t.} \quad & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 - \xi_i, \quad \forall i = 1, \dots, m, \\ & \xi_i \geq 0, \quad \forall i = 1, \dots, m. \end{aligned}$$

Une autre écriture :

$$\min_{(\mathbf{w}, b) \in \mathbb{R}^{d+1}} \frac{1}{2} \|\mathbf{w}\|_2^2 + \frac{C}{m} \sum_{i=1}^m [1 - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle) + b]_+$$

C : paramètre de régularisation entre erreurs et complexité du modèle.

Soft Margin SVM

Quelques remarques

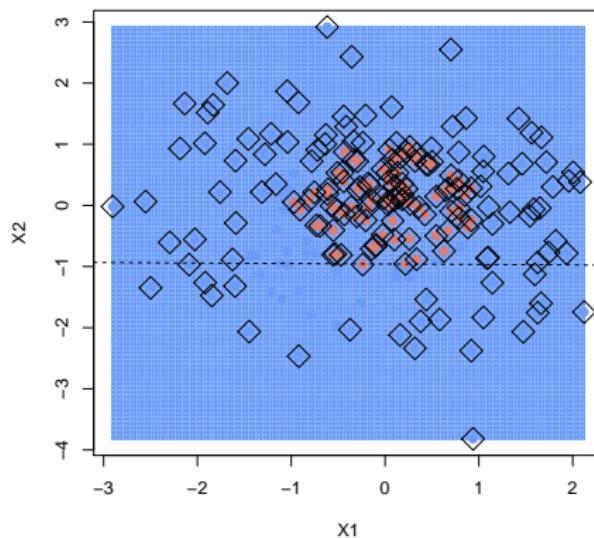
- Seuls les vecteurs du support participent à la construction de la solution (conséquence des conditions de KKT + lemme de Farkas)
- Problème difficile à optimiser car beaucoup de contraintes.
- Une difficulté croissante lorsque la dimension du jeu de données est élevée avec une complexité en $\mathcal{O}(d^3)$ (Chapelle, 2007).
- On passe le plus souvent, lorsque la taille du jeu de données reste raisonnable, par la formulation duale qui se présente sous la forme (*exercice : essayer de le montrer*)

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & -\frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j \mathbf{x}_i \mathbf{x}_j + \sum_{i=1}^m \alpha_i, \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq \frac{C}{m} \quad \forall i = 1, \dots, m, \\ & \sum_{i=1}^m y_i \alpha_i = 0. \end{aligned}$$

Nécessité des modèles non linéaires

Certaines tâches ne peuvent se résoudre avec de simples modèles linéaires !

Un SVM en 2D



→ Astuce du noyaux

Méthodes à noyaux

Qu'est-ce qu'un noyaux

Une application \mathbf{K} est une application de $\mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ qui est (i) symétrique et (ii) semi-définie positive

- (i) $\forall (\mathbf{x}, \mathbf{x}') \in \mathbb{R}^d \times \mathbb{R}^d$, nous avons: $K(\mathbf{x}, \mathbf{x}') = K(\mathbf{x}', \mathbf{x})$,
- (ii) $\forall (\mathbf{x}_i, \mathbf{x}_j) \in \mathbb{R}^d \times \mathbb{R}^d$ et $\forall \mathbf{c} \in \mathbb{R}^d$, nous avons :
 $c^T \mathbf{K} c = \sum_{i=1}^m \sum_{j=1}^m c_i c_j K(\mathbf{x}_i, \mathbf{x}_j) \geq 0$.

Le cas classique $\mathbf{K} = \mathbf{I}_d$: on retrouve ce que l'on appelle le *noyau linéaire* (*cf* cas précédent).

L'idée est donc d'introduire ce type d'applications dans notre précédent problème d'optimisation

Méthodes à noyaux

Formulation d'un problème d'optimisation avec une méthode à noyau

$$\begin{aligned} \max_{\alpha} \quad & -\frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) + \sum_{i=1}^m \alpha_i, \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq \frac{C}{m}, \quad \text{for all } i = 1, \dots, m, \\ & \sum_{i=1}^m \alpha_i = 0, \end{aligned}$$

où $\mathbf{K} = K(\mathbf{x}_i, \mathbf{x}_j)$ est donc une matrice à noyau, *i.e.* notre application noyau évaluée en les différents couples de points de notre jeu de données.

En quoi cela va nous permettre de résoudre notre problème précédent ? Utiliser un noyau revient à projeter nos données dans un espace de dimension potentiellement infini ! (Mercer, 1909).

Méthodes à noyaux

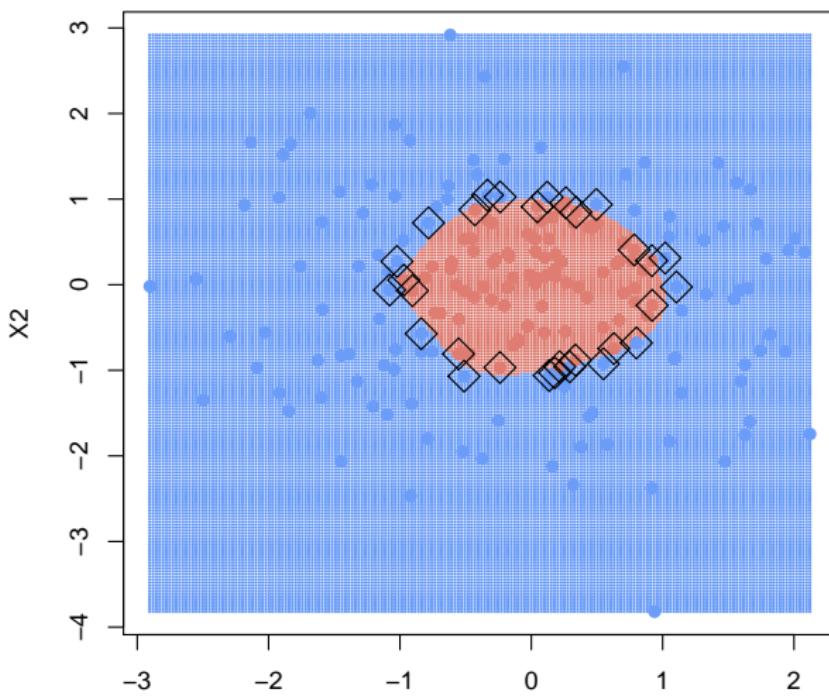
Quelques exemples de noyaux

- **Noyau linéaire** : $K(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle$, il s'agit du produit scalaire standard
- **Noyau gaussien** : $K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\sigma^2}\right)$, où σ est un hyper-paramètre à tunner. Il contrôle l'importance que va avoir la similarité entre deux points. Plus ce paramètre est grand, moins on accorde d'importance à la similarité entre deux exemples et plus le rôle de chaque exemple sera uniforme dans l'apprentissage du modèle (moins de risque d'overfitter).
- **Noyau polynomial** : $K(\mathbf{x}, \mathbf{x}') = (1 + \langle \mathbf{x}, \mathbf{x}' \rangle)^p$, p est le degré du polynôme.

Vous pouvez trouver une liste plus complète de noyaux utilisés en *ML* chez Genton (2002).

A l'aide d'un noyau gaussien

Un SVM en 2D



Processus d'apprentissage

On va cette fois-ci effectuer la procédure d'apprentissage par cross-validation de nos hyper-paramètres et tester notre modèle sur un ensemble séparé, appelé ensemble test.

```
1 # Librairies et jeu de données
2
3 library(class)
4 library(ISLR)
5 library(caret)
6
7 load(file = "~/Desktop/Cours/Lyon2/ESL.mixture.rda")
8 attach(ESL.mixture)
9
10 # Préparation des jeux de données
11
12 index_train <- createDataPartition(y=y, p=0.75, list=FALSE)
13 train <- cbind(x[index_train,], y[index_train])
14 test <- cbind(x[-index_train,], y[-index_train])
15 colnames(train) = c("X", "Y", "y")
16 colnames(test) = c("X", "Y", "y")
```

Processus d'apprentissage

On regarde maintenant comment mettre en place la cross-validation en tunant deux hyper-paramètres de notre modèle : *régularisation* et *paramètre du noyau gaussien*.

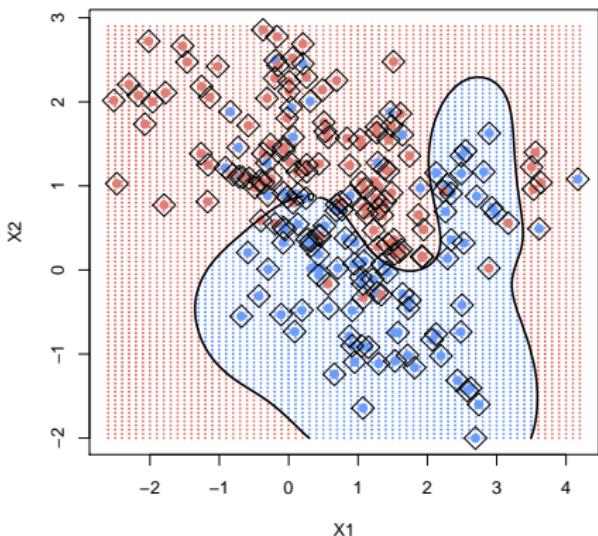
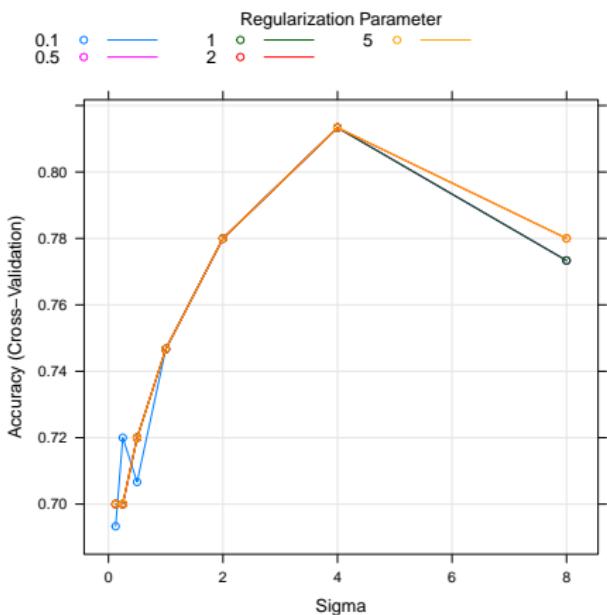
```
1 # Apprentissage par Cross-Validation
2
3 train_control <- trainControl (method = "cv", number =5)
4 mygrid <- expand.grid(lambda = c(0.1, 0.5, 1, 2, 5), qval = 1, sigma = 2^(-3:3))
5 model <- train(as.factor(y)~., data = train, trControl = train_control,
6                 method = "dwdRadial", tuneGrid = mygrid)
7
8 model
9 plot(model)
10
11
12 # Test et matrice de confusion
13
14 svm_prediction <- predict(model, newdata = test)
15 mean(svm_prediction == test[,3])
16
17 confusionMatrix(svm_prediction, as.factor(test[,3]))
```

Visualisation des résultats

Regardons comment visualiser ces résultats

```
1 xgrid = expand.grid (X1 = px1, X2 = px2)
2 ygrid = predict(model, xgrid)
3 param <- model$results[which.max(model$results[, "Accuracy"]), c("lambda", "sigma")]
4
5 dat = data.frame(y = as.factor(y), x)
6 fit = svm(factor(y)~., data = dat, scale = FALSE, kernel = "radial", cost = param[1],
    gamma = param[1])
7
8 func = predict(fit, xgrid, decision.values = TRUE)
9 func = attributes(func)$decision
10
11 plot(xgrid, col = ifelse(as.numeric(y.grid) - 1) == 1, "#DF7D72", "#6B9DF8"), pch = 20,
    cex = 0.2)
12 points(x, col = ifelse(y == 1, "#DF7D72", "#6B9DF8"), pch = 19)
13 contour(px1, px2, matrix(func, 69, 99), level = 0, add = TRUE, lwd = 2)
```

Visualisation des résultats



Prédiction

- **Cas du SVM linéaire**

Dans le cas présent, il suffit de regarder le signe de la quantité

$$\langle \mathbf{w}, \mathbf{x} \rangle + b = \langle \tilde{\mathbf{w}}, \mathbf{x} \rangle$$

qui traduit une distance relative à l'hyperplan séparateur.

- **Cas du SVM non linéaire**

Le label d'une nouvelle donnée \mathbf{x} est donné par le signe de la fonction h

$$h(\mathbf{x}) = \sum_{i=1}^m \alpha_i K(\mathbf{x}, \mathbf{x}_i).$$

Remarque : si le modèle a peu de points supports, on peut drastiquement réduire le nombre d'éléments dans la somme ! Cela revient à considérer uniquement les **points supports**, i.e. ceux dont α_i est non nul.

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.

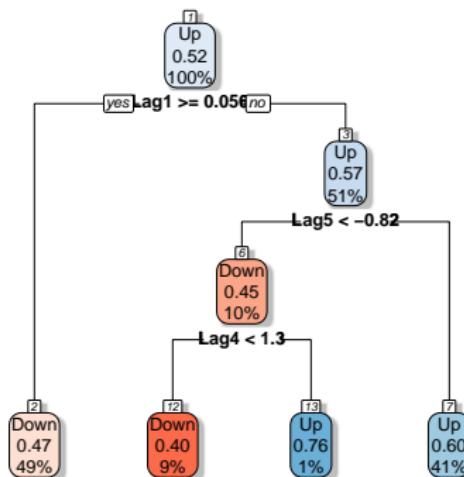
L'objectif de cet algorithme (CART) est de séparer les données en créant des groupes les plus **homogènes** possibles. C'est d'ailleurs cette notion **d'homogénéité** qui va servir de critère d'apprentissage des arbres.

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

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

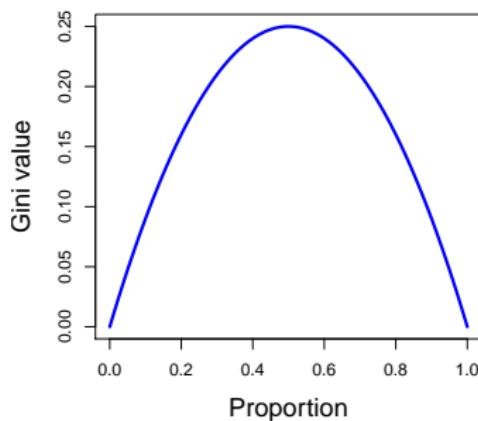
- **L'indice de Gini** D , utilisé dans l'algorithme **CART**

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

- 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

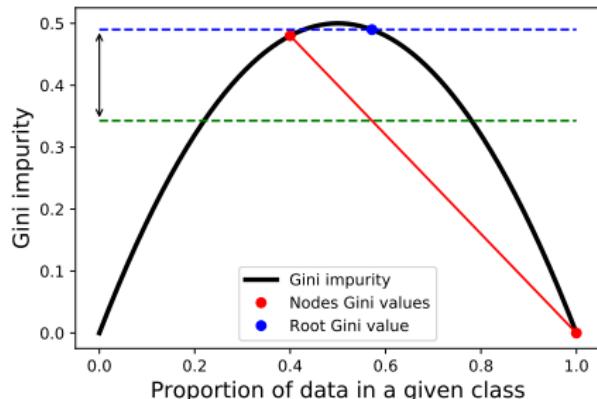
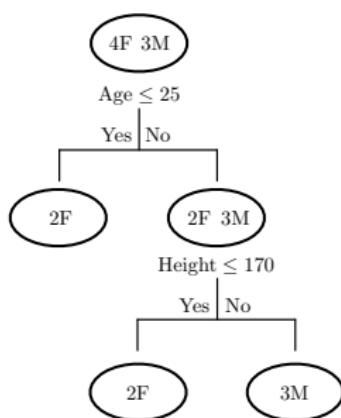
$$\theta^* = \arg \max_{\theta} \frac{m_{\text{left}}}{m} \Gamma_{\text{left}}(\theta) + \frac{m_{\text{right}}}{m} \Gamma_{\text{right}}(\theta),$$

où Γ 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

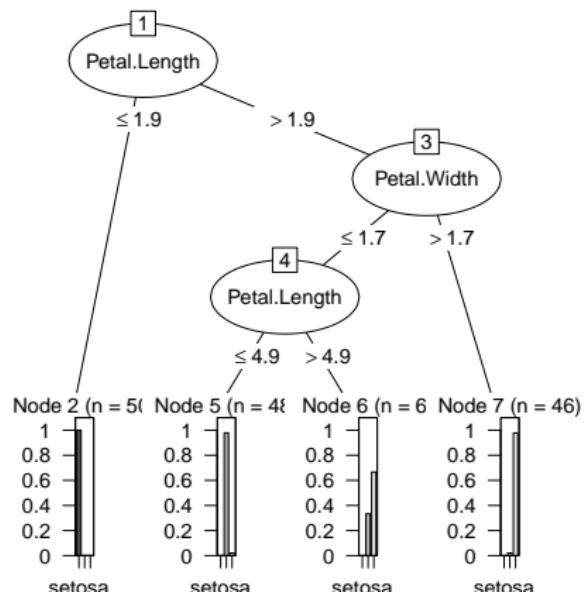
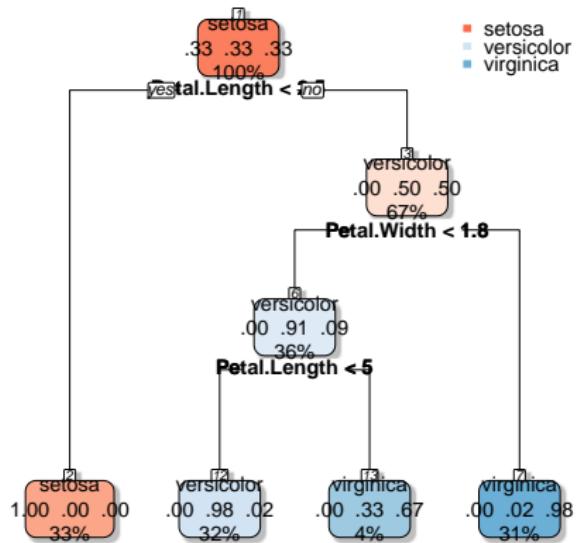
Plusieurs librairies sont possibles sous R: C50 et rpart

```

1 # Jeu de donnees
2 data(iris)
3
4 # A) Utilisation de l'entropie de Shannon
5
6 # Librairie C50 : uniquement pour de 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 (regression)
17
18 # Librairie rpart : classification et regression
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$confusion # matrice de confusion sur les données non utilisées pour construire
  un arbre
9 model_tf$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

- Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152. ACM.
- 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.
- Chapelle, O. (2007). Training a support vector machine in the primal. *Neural Computation*, 19:1155–1178.
- 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.

References II

- Genton, M. G. (2002). Classes of kernels for machine learning: A statistics perspective. *Journal of Machine Learning Research*, 2:299–312.
- Jaggi, M. (2013). Revisiting frank-wolfe: Projection-free sparse convex optimization. In *Proceedings of the 30th international conference on machine learning*, number CONF, pages 427–435.
- McFee, B. and Lanckriet, G. R. (2010). Metric learning to rank. In *ICML*.
- Mercer, J. (1909). Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 209(441-458):415–446.
- 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.
- Vapnik, V. and Cortes, C. (1995). Support-vector networks. *Machine Learning*, 20:273–297.
- 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.
- Ying, Y., Huang, K., and Campbell, C. (2009). Sparse metric learning via smooth optimization. In *Advances in neural information processing systems*, pages 2214–2222.