Fouille
de
Données Massives

Master 2 Informatique - SISE

Guillaume Metzler

Université Lumière Lyon 2
Laboratoire ERIC, UR 3083, Lyon

guillaume.metzler@univ-lyon2.fr

Automne 2022

Plan du cours I

Une séance de CM

Elle pour objectif de vous présenter le contexte du projet dans au cours duquel vous serez amenés à travailler des **données réelles** et avec une quantité **importante de données** (10 mois de transactions). Mais aussi

- présentation de la problématique Imbalanced
- quelques mesures de performances associées
- stratégies avant apprentissage
- quelques algorithmes adaptés

Plan du cours II

4 séances sur Machine

- une première séance de remise en route sur l'apprentissage machine (déjà effectuée)
- une deuxième séance plus théorique sur la façon d'établir des bornes en généralisation (on active un peu la théorie)
- une troisième séance qui va graviter autour des SVM (Séparateurs à Vaste Marge)
- une quatrième séance sur le boosting
- une cinquième séance sur les approches deep

La dernière séance sera consacrée à votre examen ainsi qu'à un moment d'échange autour du projet qui vous sera confié.

Plan du cours III

En terme d'évaluations, vous aurez donc un projet (à réaliser par groupe de deux personnes) ainsi qu'un examen sur table.

Un poids plus important sera accordé au projet. L'examen sur table comportera deux parties :

- une partie *cours* : vous serez interrogés sur des éléments vus en cours ou encore pendant les séances sur Machine
- une partie *mise en situation* : un ou plusieurs contextes d'études vous seront donnés. Il faudra, à partir de vos connaissances, proposer une approche permettant de traiter le problème et décrire votre protocole expérimental.

Introduction

(Big) Data Mining I

Le data mining, ou fouille de données, a pour objectif d'extraire de la connaissance à partir d'un ensemble de données (structurées) ou non.

Cette extraction a pour but d'acquérir des connaissances sur des sujets particuliers définis par l'utilisateur. Elle peut permettre la compréhension d'un phénomène ou encore la réalisation de certaines tâches (comme de la classification - régression).

L'extraction de connaissances se fait au moyen d'outils statistiques ou informatiques qui pourront servir d'outils d'aide à la décision.

(Big) Data Mining II

Historiquement, l'analyse de données se faisait sur une quantité de données très faibles mais aussi présentant un faible nombres de variables (features). Les outils informatiques actuels n'étant pas à disposition des scientifiques de l'époque.

Avec le développement constant de la capacité de stockage des données de la part des entreprises, on commence à se rendre compte de l'importance (marketing, économique) que peuvent représenter les données.

Les sociétés s'y intéressent d'avantage et n'hésite pas à accroître des moyens de **stockage** et d'**analyse de ces données**, on peut alors parler d'émergence du **data mining**.

(Big) Data Mining III

Les capacités de stockage ne cessent d'augmenter (loi de Moore) et on commence à étudier une variété plus importante de données, comme les données génomiques dont le coût d'acquisition est très important mais qui ont la particularité de présenter un très grand nombres de variables $\rightarrow$ statistiques en grande.

Ce n'est que depuis peu de temps que nous sommes maintenant capables d'analyser des données en quantités importantes mais aussi avec un grand nombre de variables : graphes avec l'émergence des réseaux sociaux.

Ces avancées poussent au développement à de nouveaux outils d'analyses et de traitements de plus en plus sophistiqués.

(Big) Data Mining IV

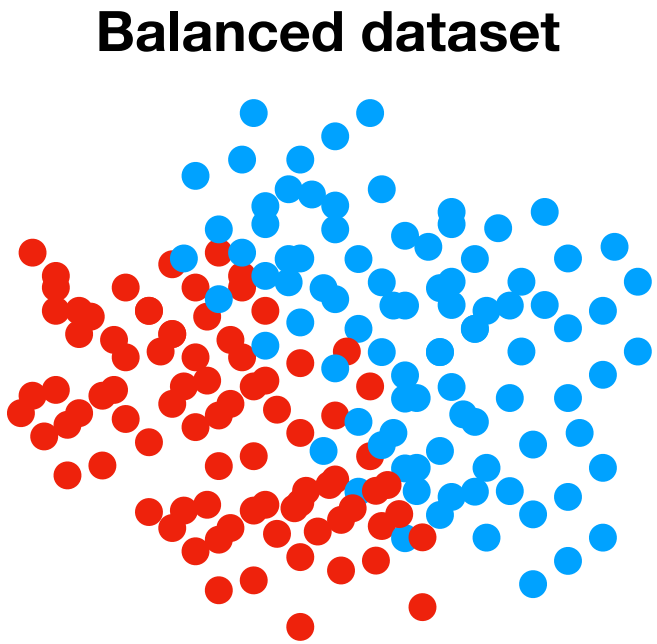
Le data mining se compose de 4 étapes principales

- Collecte et intégration des données
- Pré-traitement des données
- Analyse des données : régression, classification, segmentation
- Validation du protocole expérimentale et test

Imbalanced Learning

Contexte habituelle I

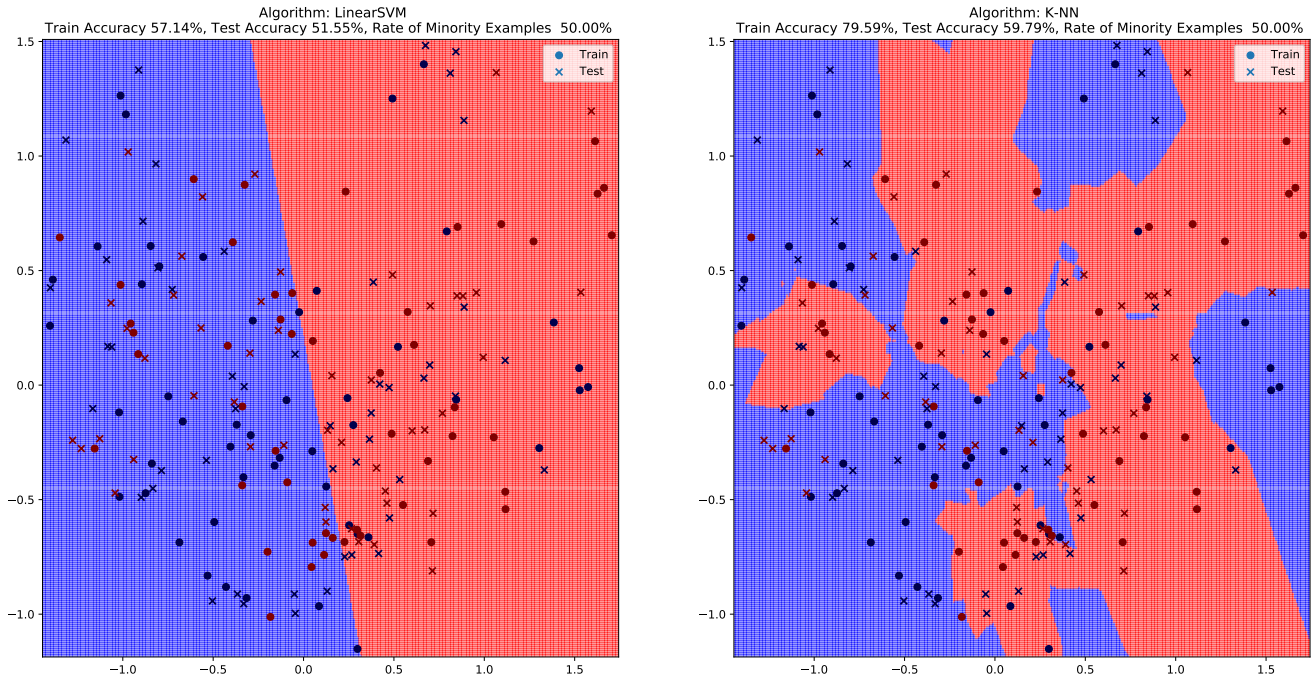
La plupart des algorithmes/méthodes employées en Apprentissage Machine suppose que toutes les classes sont représentées de façon quasi égales.



En général vous avez 50% dans chaque classe.

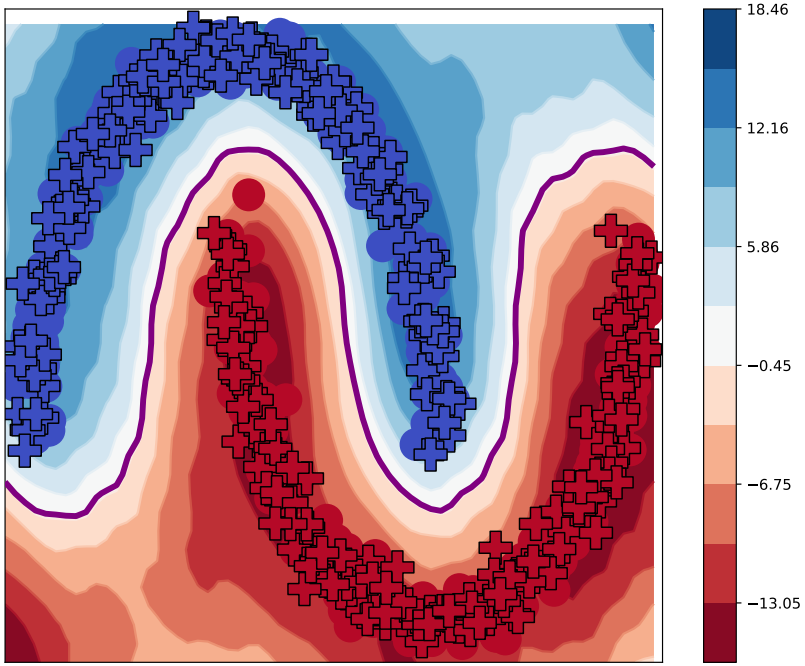
Contexte habituelle II

Des algorithmes standards, qu'ils soient paramétriques ou non, comme les SVM ou encore les k- plus proches voisins permettent d'obtenir des performances raisonnables sur de telles données.



Contexte habituelle III

Des algorithmes encore plus complexe comme les SVM non linéaires, *i.e.* avec un noyau gaussien sont donc également capables de très bien résoudre nos problèmes de classifications, même si le jeu de données est complexe.



Contexte habituelle IV

L'ensemble de ces algorithmes cherchent à minimiser leur taux d'erreur sur l'ensemble d'apprentissage (sauf pour le k-plus proche voisin)

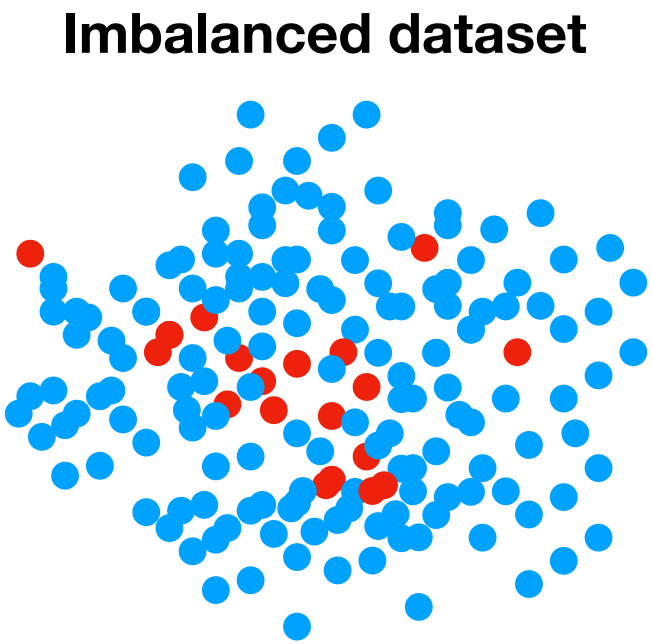
$$\mathcal{R}_S = \frac{1}{m} \sum_{i=1}^{m} \mathbb{1}_{\{h(\mathbf{x}_i) \neq y_i\}}.$$

Ou plus précisément des substitues du taux d'erreur, *i.e.* des bornes supérieures, via l'utilisation de fonctions de loss

$$\mathcal{R}_S^{\ell} = \frac{1}{m} \sum_{i=1}^{m} \ell(h(\mathbf{x}_i), y_i).$$

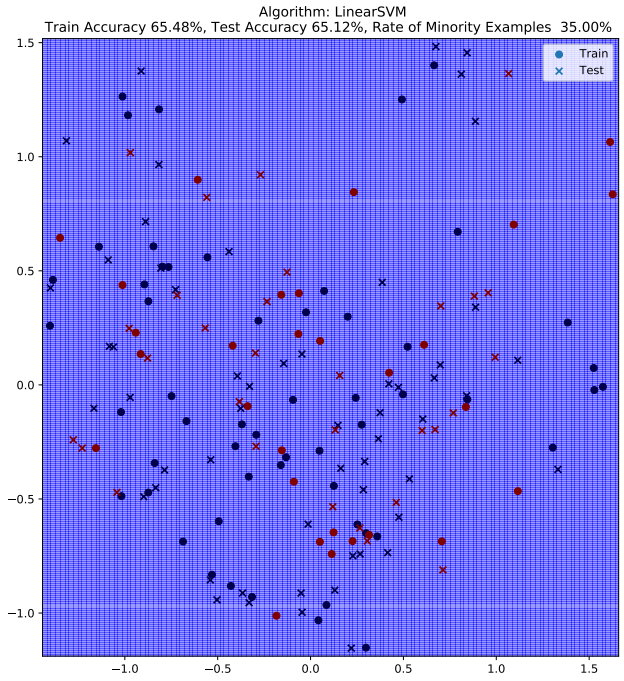
Classe minoritaire I

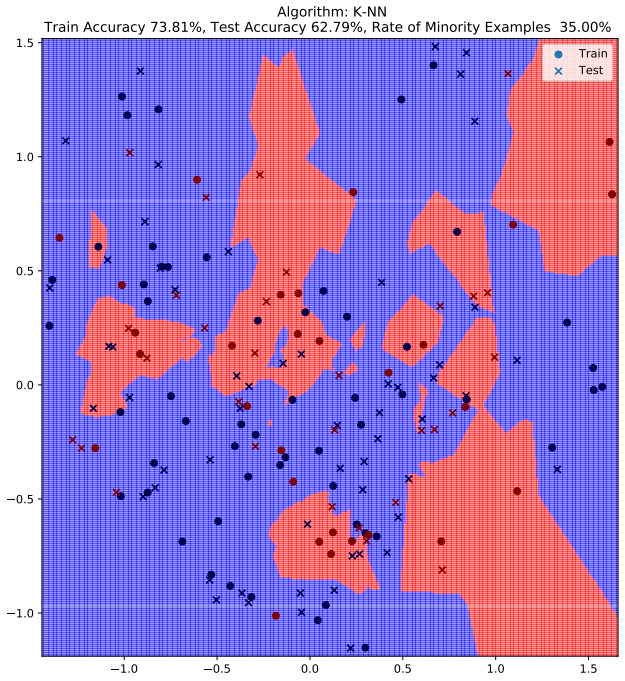
Qu'est-ce qui se passe lorsqu'une classe est sous représentée par rapport aux autres classes ? Lorsque le nombre d'exemples d'une classe devient très faibles par rapport à une ou plusieurs classes



Classe minoritaire II

Comment se comportent nos algorithmes ? Que peut-on dire des surfaces des décisions et surtout ... pourquoi ?





Classe minoritaire III

- Le SVM linéaire a tendance à prédire toutes les données comment étant négatives, ce qui a une conséquence direct sur les performances de l'algorithme. Le taux d'erreur est égal à la représentation de la classe minoritaire
  Nous avons une performance proche de 65%.
- La surface de décision d'un plus proche voisin a tendance à se déséquilibrer également, on réduit les chances de trouver des exemples de la classe minoritaire.
- Pourquoi ? Le problème peut venir de l'algorithme ou du critère que l'on cherche à optimiser.

Classe minoritaire IV

Généralement dans les problèmes déséquilibrés, ce qui nous intéresse, c'est de retrouver les exemples de la classe minoritaire.
Cette dernière peut représenter des *fraudes*, *anomalies* ou des *exemples atypiques* qui peuvent représenter une menace.

Exemple : fraudes bancaires - fraudes fiscales - intrusion dans des systèmes - anomalie sur une chaîne de montage - anomalie dans des analyses médicales - anomalie dans des images médicales - détection d'anomalies pour appareils de mesure comme des capteurs, ...

Classe minoritaire V

Au fait, est-ce que l'on peut parler indifféremment d'anomalie ou de de fraudes ? Quid des points communs/différences ?

Mesures de Performances

Mesures de Performances I

- Chercher à minimiser le taux d'erreur n'est pas une bonne solution dans ce contexte, car il tend à favoriser la classe majoritaire.
- On préfère évaluer le modèle avec des mesures plus adaptées comme la précision, le rappel ou encore la F-mesure, en utilisant **notre matrice de confusion**

	$h(\mathbf{x}) = 1$	$h(\mathbf{x}) = -1$
$y = 1$	Vrai positif	Faux négatif
$y = -1$	Faux positif	Vrai négatif

Table – Matrice de confusion d'un classifieur binaire

Mesures de Performances II

- Précision : on évalue la justesse du modèle dans ses prédictions positives

$$\frac{TP}{TP + FP}$$

- Rappel : évalue la capacité du modèle à retrouver des exemples de la classe positive ou minoritaire

$$\frac{TP}{TP + FN}$$

Mesures de Performances III

- F-mesure : c'est une moyenne harmonique entre les deux précédentes mesures, elle dépend d'un paramètre β qui gère le trade-off entre Rappel et Précision (on prend souvent $\beta = 1$)

$$\frac{1+\beta^2}{\frac{1}{Rappel} + \frac{\beta^2}{Précision}} = \frac{(1+\beta^2)TP}{(1+\beta^2)TP + \beta^2 FN + FP}.$$

où un *TP* est une fraude prédite fraude par votre modèle, un *FN* est une fraude non identifiée comme tel par votre modèle, un *FP* est une transaction non frauduleuse mais identifiée comme frauduleuse par votre modèle et enfin un TN est une transaction non frauduleuse.

Mesures de Performances IV

• Vous pouvez aussi chercher à maximiser l'aire sous la courbe ROC (AUC ROC). Sa définition est à peine plus complexe, il s'agit de l'aide sous courbe (TPR, FPR) où

$$TPR = \frac{TP}{TP + FN} \quad \text{et} \quad FPR = \frac{FP}{FP + TN}$$

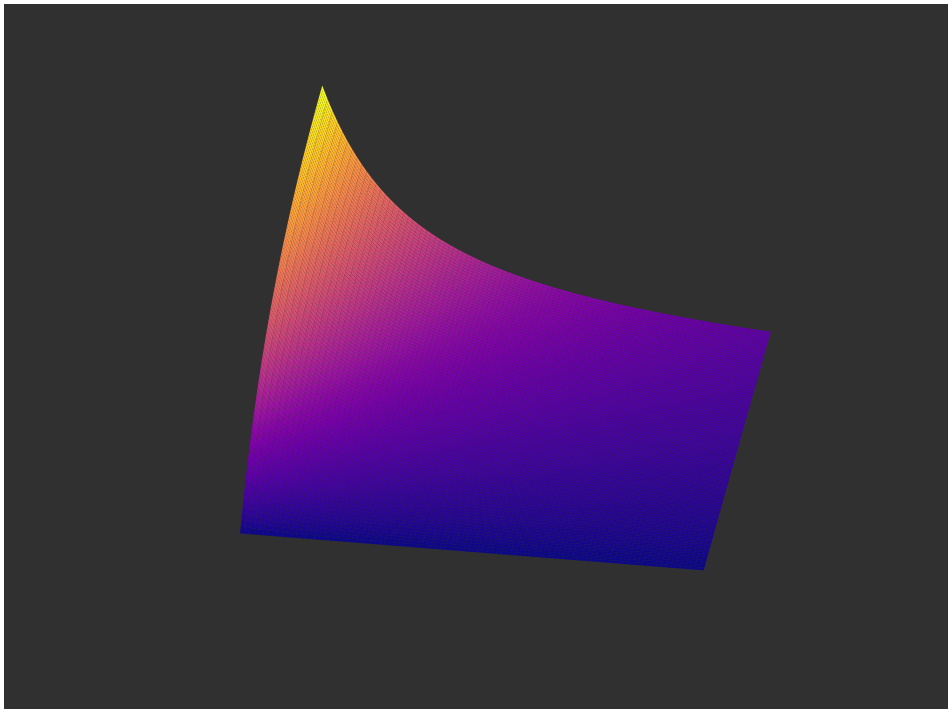
• Soit un modèle f renvoyant un score, on peut aussi définir l'AUC ROC par la relation :

$$\frac{1}{PN} \sum_{i=1}^{P} \sum_{j=1}^{N} \mathbb{1}_{\{f(\mathbf{x}_i^+) \geq f(\mathbf{x}_i^-)\}}.$$

Cette dernière mesure offre plus d'informations sur la qualité du modèle, plus adaptée aussi aux problématiques de ranking.

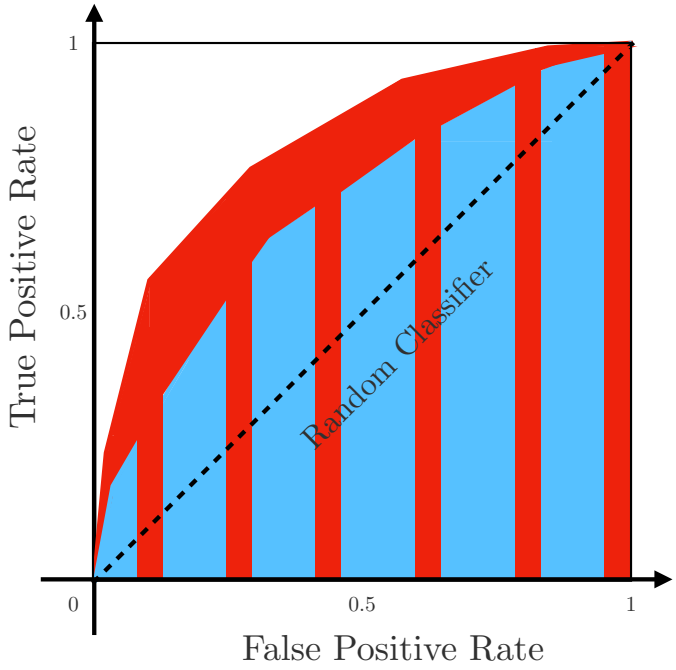
Mesures de Performances V

F-Mesure



$$F_\beta = \frac{(1+\beta^2)(P - FN)}{(1+\beta^2)P - FN + FP}$$

AUC

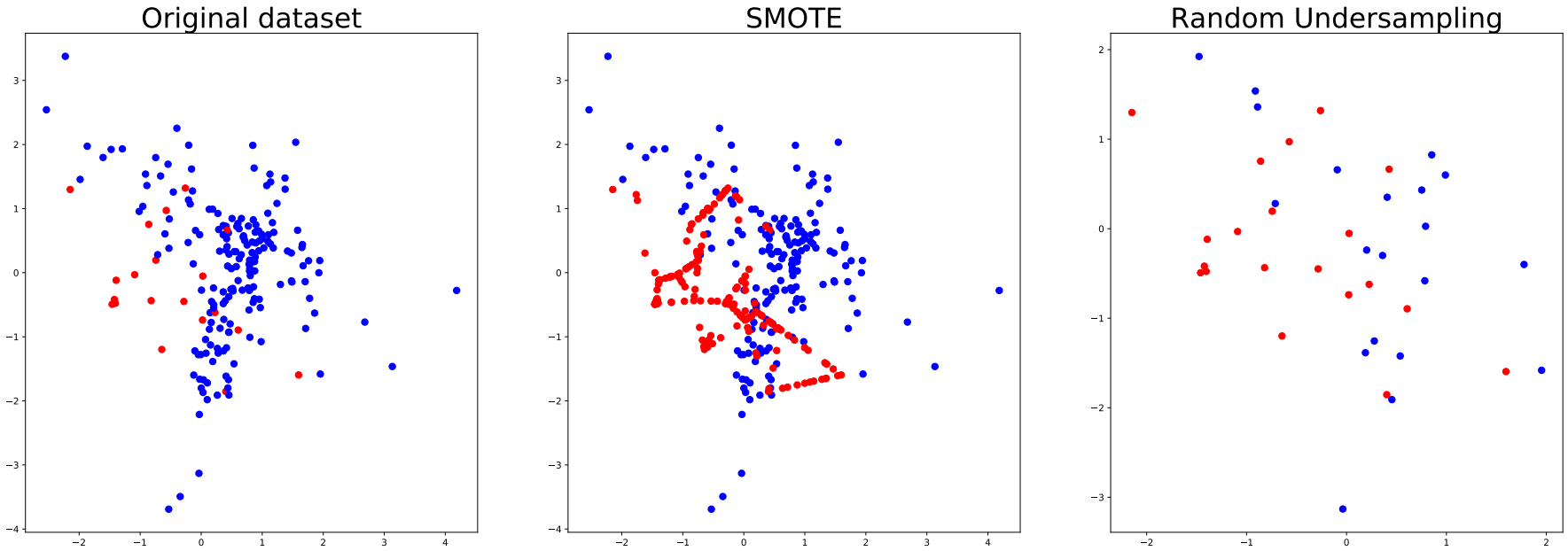


$$\mathbb{P}[f(x_+) > f(x_-)]$$

Stratégies échantillonnage

Apprentissage dans un contexte déséquilibré

Utiliser des stratégies d'échantillonnages



- Oversampling : Random - SMOTE - BorderSMOTE, ...
- Undersampling : Random - Tomek Link - ENN, ...

Oversampling I

Aléatoire

C'est certainement la plus simple des approches à employer. Il ne s'agit pas de générer des points aléatoirement mais simplement de dupliquer des points existants de la classe minoritaire.

Ainsi, on va demande à notre algorithme de se concentrer d'avantage dans les zones de l'espace où ces points ont été répliqués.

Malheureusement, en procédant ainsi, on n'ajoute pas d'information dans nos données et on ne contrôle pas vraiment ce qu'il se passe ... le processus est totalement aléatoire.
Voyons comment on peut **ajouter de l'information**.

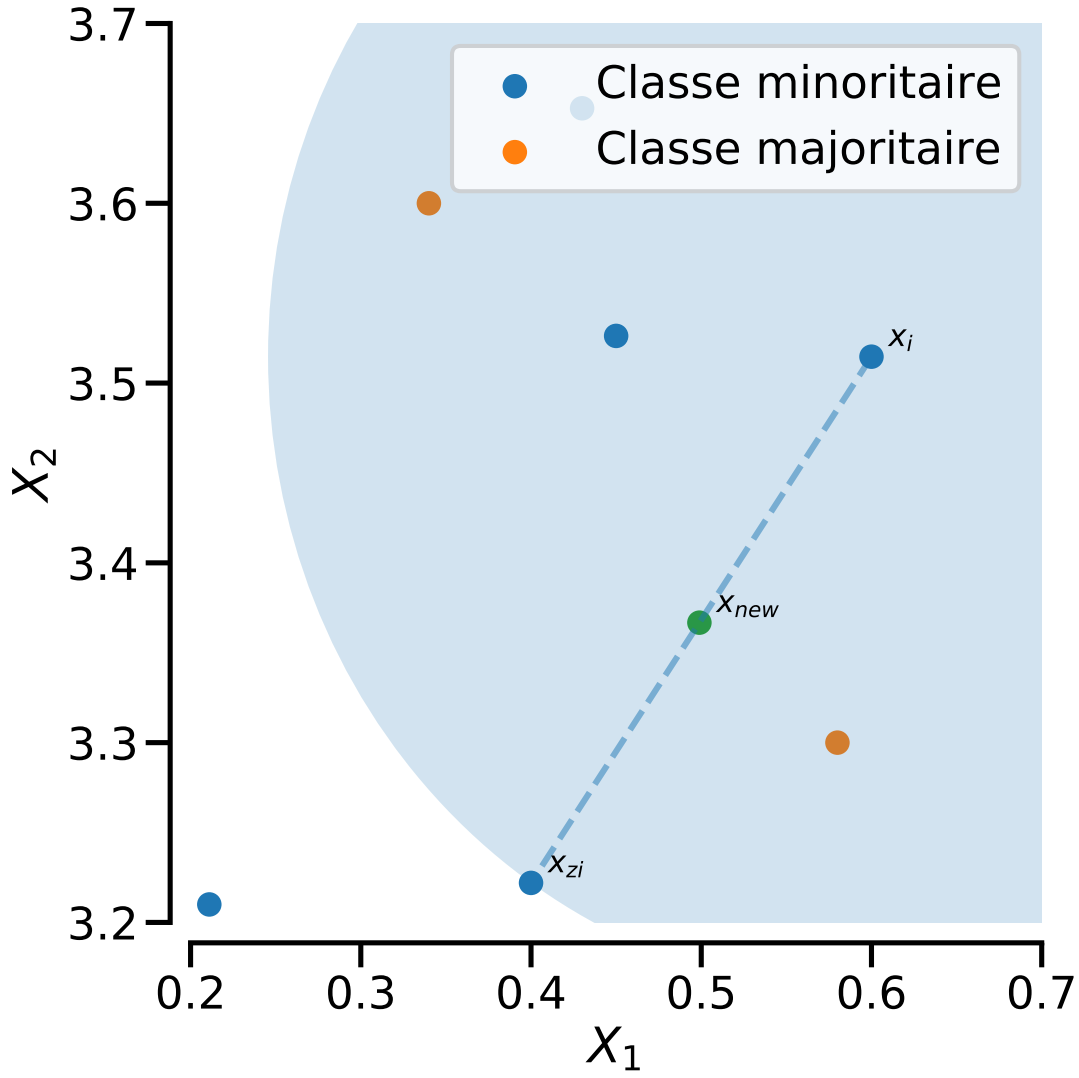
Oversampling II

SMOTE

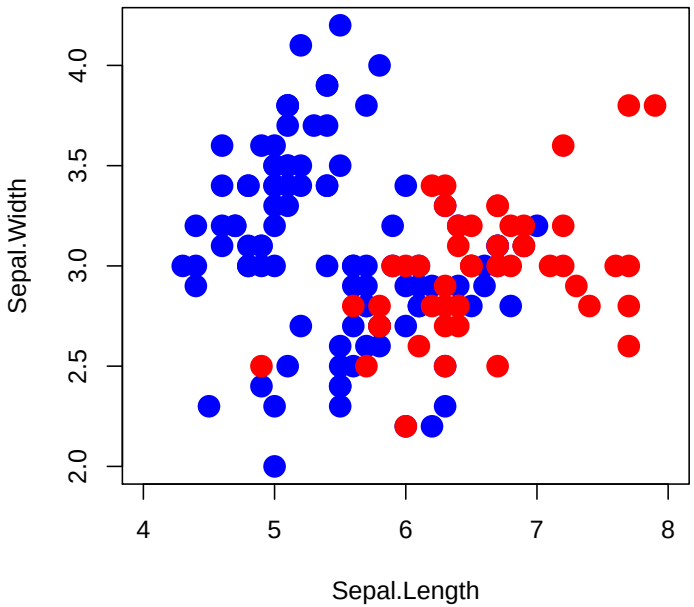
Algorithm 1: Algorithme SMOTE [Chawla et al., 2002]

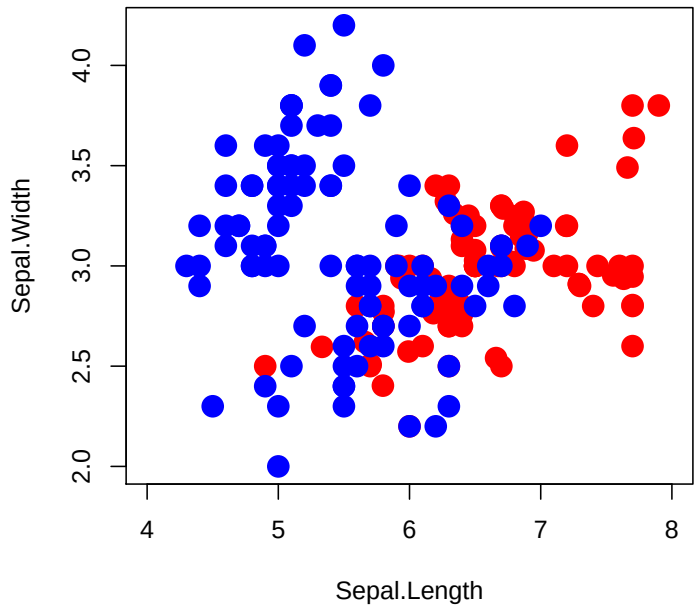
Input: Echantillon d'apprentissage S de taille $m = p + n$,
k : nombre de voisins, R taux d'exemples positif à générer
Output: Un échantillon S'
begin
  Poser $New = R \times p$: nombre d'exemples à générer Syn,
  sélectionner aléatoirement New exemples parmis p et noter $setind$
   leur indice
  for $i \in setind$ **do**
    chercher les $k-$pp positifs de l'exemple $\mathbf{p}_i$,
    sélectionner aléatoirement l'un des plus plus proches voisins
     rNN_i **for** *attributs* $attr$ *de* $\mathbf{p}_i$ **do**
      choisir un nombre $\alpha \in [0, 1]$,
      poser $newattr = \alpha p_i[attr] + (1 - \alpha) rNN_i[attr]$
      poser $Syn_i[attr] = newattr$
    Poser ensuite $S' = S \cup Syn_i$
  Poser ensuite $S' = S$
return S'

Oversampling III



Oversampling IV





Oversampling V

BorderlineSMOTE

L'idée est très proche de SMOTE, mais on va cette fois-ci imposer une règle de sélection sur les exemples de la classe minoritaire qui seront utilisés pour la génération de données.

- On se fixe deux paramètres qui correspondent à un nombre de voisins k et k'. Un premier spécifique à SMOTE et un deuxième qui va servir à déterminer les exemples **borderline**.
- Un exemple borderline est un exemple dont la majorité des voisins (mais pas tous !) appartiennent à la classe majoritaire.
- Pour ces exemples là, on va ensuite générer des points à l'aide de la procédure SMOTE.

Oversampling VI

Sur le plan algorithmique, on va donc :

1. détecter tous les exemples de la classe minoritaire qui ont au moins $k'/2$ voisins dans la classe majoritaire (mais pas exactement k')
2. pour tout ces exemples on va appliquer l'algorithme SMOTE de façon à atteindre le ratio de positifs désiré.

Cet algorithme va donc se concentrer sur les exemples de la classe minoritaire qui sont difficiles à classer et laisser de côté les exemples faciles à classer (majorité de voisins dans la classe minoritaire) ou qu sont du bruits (k' voisins dans la classe majoritaire).

Oversampling VII

ADASYN

Il s'agit d'une version plus lisse que Borderline SMOTE. L'idée est très proche mais on cherche cette fois-ci à concentrer la génération sur les exemples plus difficiles à classer.

On va définir un score r_i de risque sur les exemples de la classe minoritaire qui correspond au nombre Δ_i de voisins dans la classe majoritaire sur ke nombre total de voisins, *i.e.*

$$r_i = \frac{\Delta_i}{k'}$$

Plus ce score est élevé, plus on va générer d'exemples autour de cette cette donnée.

Oversampling VIII

Si on souhaite créer N exemples au total, on va donc les répartir sur l'ensemble des données de la classe minoritaire.

Il faut donc transformer nos scores de risque r_i en une distribution

$$r'_i = \frac{r_i}{\sum_{j=1}^{p} r_i}$$

Et ainsi, pour chaque exemple de la classe minoritaire $\mathbf{x}_i$ on va créer exactement $N \times r'_i$ exemples.

Oversampling IX

Implémentation

Toutes les méthodes précédemment citées peuvent se trouver dans la librairie *imblearn.over_sampling* et les fonctions associées sont :

- **RandomOverSampler**
- **SMOTE**
- **BorderlineSMOTE**
- **ADASYN**

Pour toutes ces fonctions, le nombre de d'exemples générés est déterminé en fonction du ratio final de positifs souhaités dans notre jeu de données

$$\frac{n_{\text{minoritaire après sampling}}}{n}.$$

Oversampling X

Pour finir [Fernández et al., 2018]

- Permet de générer de nouveaux exemples de la classe minoritaire.
- Pour certaines variantes, on peut même se concentrer sur les zones difficiles (la frontière de décision)
- Mais cela augmente le nombre d'exemples
- Cela génère aussi parfois du bruit dans les données

Undersampling I

Aléatoire

On va supprimer, ou enlever aléatoirement des exemples de la classe minoritaire afin de réduire les déséquilibre mais sans toucher à la classe majoritaire cette fois-ci.

On sera ainsi susceptible de rendre les frontières de décisions beaucoup plus nettes entre les exemples de la classe minoritaire et majoritaire.

Malheureusement, en procédant ainsi, on risque de perdre de l'information et parfois une importante car le processus est totalement aléatoire. Voyons comment on peut fixer une bonne règle de suppression de ces exemples.

Undersampling II

Condensed Nearest Neighbor [Hart, 1968] :

C'est une méthode de pré-traitement utilisé pour accélérer l'algorithme du plus proche voisin

---

Algorithm 2: 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 modifés* **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'

Undersampling III

Algorithme Tomek Link [Tomek, 1976]

Il s'agit d'une méthode de nettoyage de données à l'image de *CNN*.

Pour se faire on considère des paires d'exemples avec des étiquettes différentes, si les deux exemples sont plus proches voisins de l'autre, on dit qu'ils forment un *lien Tomek*.

$\rightarrow$ ces exemples se trouvent donc à une frontière et peuvent potentiellement conduire à une mauvaise classification.

On décide donc de les retirer de notre jeu de données.
On peut aussi faire le choix de ne retirer que les exemples de la classe majoritaire !

Undersampling IV

Edited Nearest Neighbor [Wilson, 1972]

Il s'agit d'une version très proche de Tomek Link.
L'idée est de se fixer un paramètre k qui correspond au nombre de voisins que l'on va considérer pour le process de nettoyage (en général $k = 3$).
Pour chaque exemple de la classe majoritaire on va :

- déterminer ses k- plus proches voisins
- déterminer la classe majoritaire dans le k voisinage
- supprimer l'exemple de l'ensemble d'apprentissage si la prédiction ne coïncide pas avec le vrai label

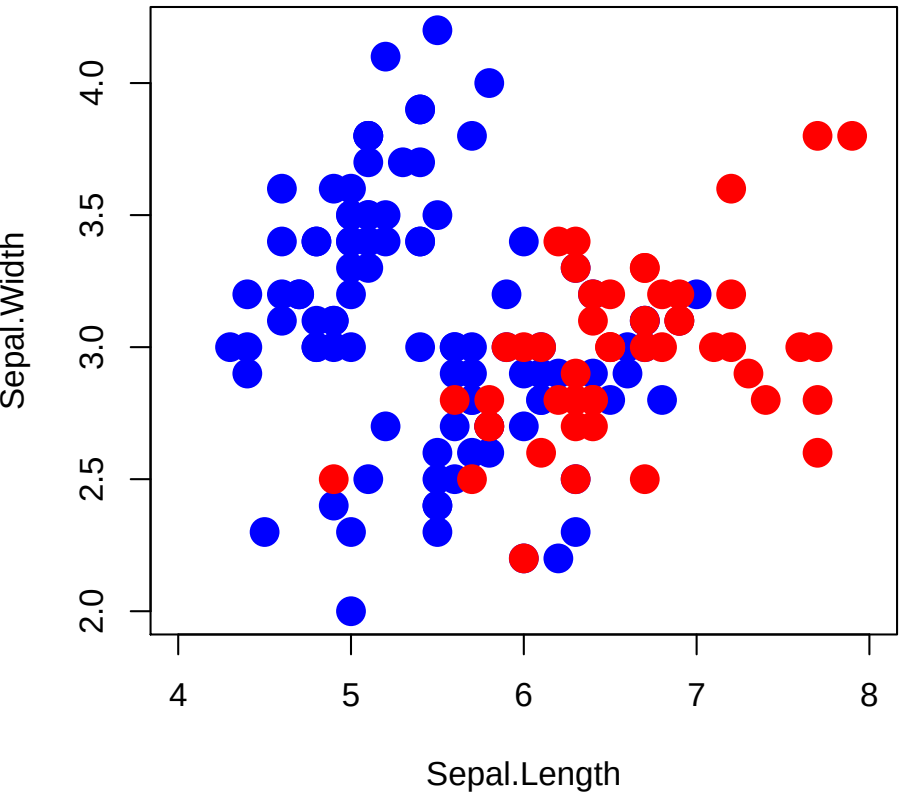
Undersampling V

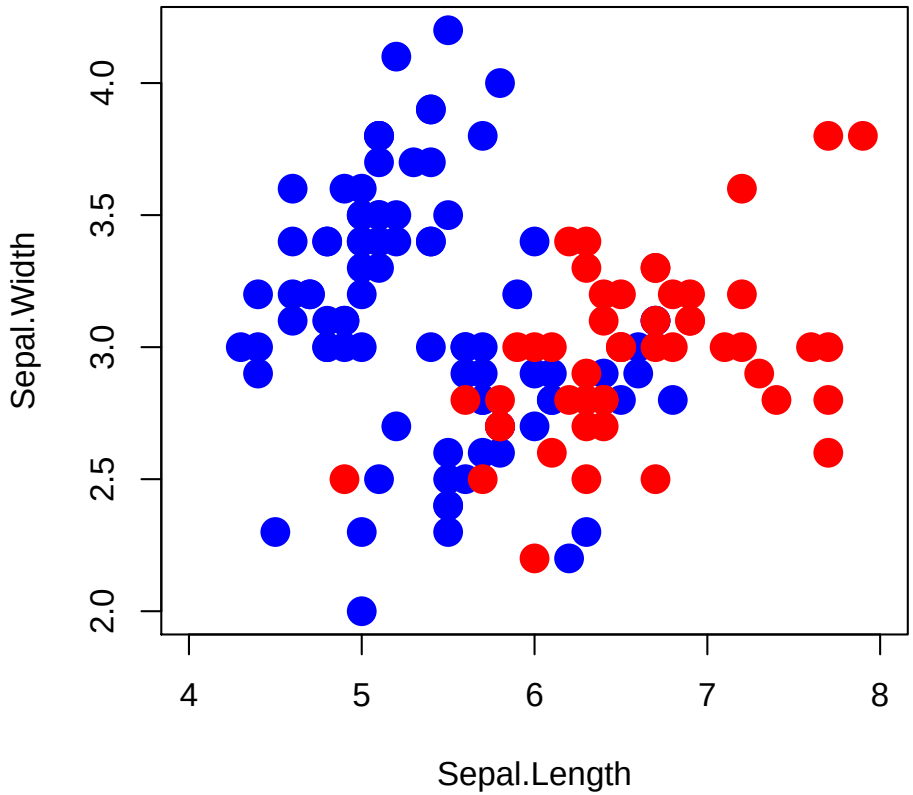
One Sided Selection [Tomek, 1976]

Cette dernière approche consiste à appliquer l'algorithme de *Condensed Nearest Neighbor* suivi ensuite par *Tomek Link* afin de réduire le temps de calcul de l'approche Tomek link lorsque les jeux de données sont volumineux.

Il permet également de réduire de façon plus conséquente la taille du jeu de données ce qui peut se révéler très intéressant d'un point de vue algorithmique et ainsi permettre d'apprendre nos modèles plus rapidement.

Undersampling VI





Cost-Sensitive Learning

Principe I

Cette deuxième approche consiste à modifier la façon dont les erreurs sont prises en compte par notre algorithme [Elkan, 2001].

On va partir du principe que les erreurs n'ont pas le même poids.

Si on souhaite faire en sorte que notre algorithme se focalise sur les exemples de la classe minoritaire, il va falloir lui indiquer que mal classer un tel exemple va entraîner une valeur importante de la loss, et que, au contraire, s'il classe mal un exemple de la classe majoritaire, cela aura une conséquence moindre.

Principe II

La pondération ainsi effectuée va se faire en définissant au préalable une matrice de coût comme cela présentée en Table 2. Elle est semblable à une matrice de confusion mais, au lieu de regarder le nombre de bonnes/mauvaises décisions prises par l'algorithme, on va associer des poids/coûts aux décisions prises par ce dernier.

	$h(\mathbf{x}) = 1$	$h(\mathbf{x}) = -1$
$y = 1$	c_{TP}	c_{FN}
$y = -1$	c_{FP}	c_{TN}

Table – Matrice de coût

Principe III

Dans notre contexte actuel, on va surtout se concentrer sur les valeurs de c_{FN} et c_{FP} qui définissent le poids des erreurs effectuées sur la classe minoritaire et majoritaire respectivement.

Par exemple, si on fixe $c_{FN} = 2$ et $c_{FP} = 1$, alors mal classer un exemple de la classe minoritaire aura un coût deux fois plus élevé que de mal classé un exemple de la majoritaire. En procédant ainsi, on sera en mesure d'accorder plus d'importance à la classe minoritaire.

Principe IV

Implémentation

Dans les implémentations standards, nous avons toujours $c_{FN} = c_{FP} = 1$, *i.e.* toutes les erreurs ont le même poids.

Il n'y a pas d'implémentation spécifique pour le cost-sensitive learning, il faudra le plus souvent régler le paramètre 'class_weight' des différents implémentations sous Python.

Principe V

Remarque

On pourrait aller plus loin dans la pondération des erreurs et être plus fin. Si on attribue des poids différentes à chaque classe, rien ne nous empêche de définir des poids différents à chaque exemple quel que soit sa classe.

Dans un contexte économique, voire bancaire (fraudes au assurances, impôts ou bancaires), il n'est pas rare d'attribuer un poids plus important aux fraudes d'un montant élevé qu'aux petites fraudes !
C'est d'ailleurs ce que l'on fera si on veut minimiser les pertes lors d'un processus de détection de fraudes.

Arbres et Forêts Aléatoires

Arbres de décisions I

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.

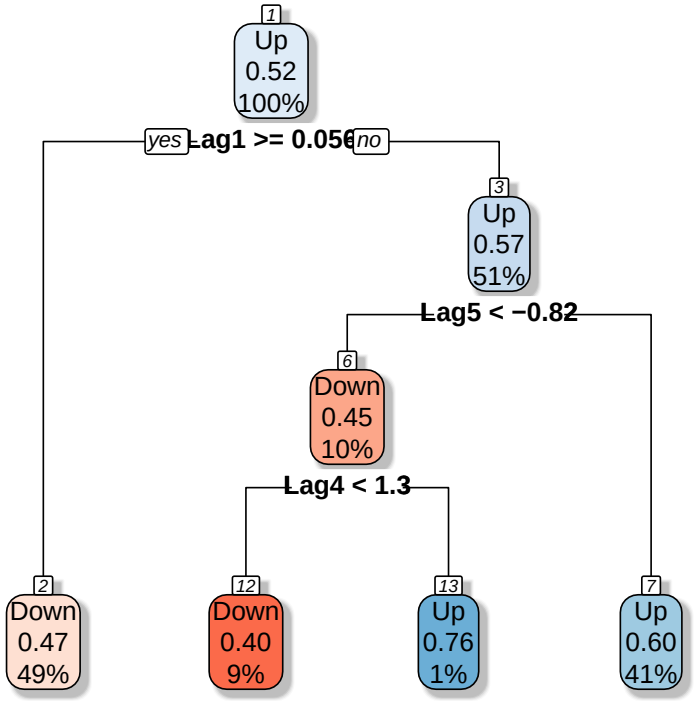
L'objectif de cet algorithme (CART) est de séparer les données en créant des groupes les plus **homogènes** possibles. C'est d'ailleurs cette notion **d'homogénéité** qui va servir de critère d'apprentissage des arbres.

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

Arbres de décisions II

- 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*

Arbres de décisions III



Arbres de décisions IV

En pratique

On ne construit pas les arbres jusqu'à l'obtention de feuilles pures (pourquoi ?). 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 !

Quel critère employer pour effectuer nos différentes séparations ?

Arbres de décisions V

- 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**

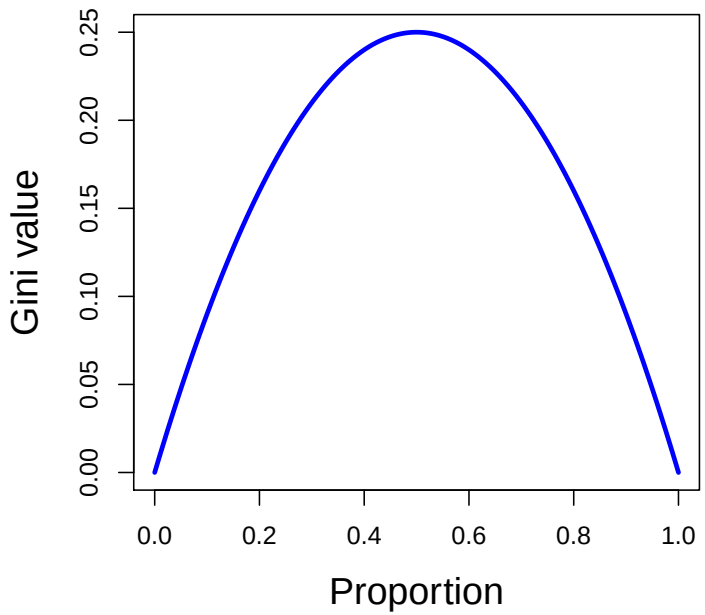
$$D = \sum_{j=1}^{C} \frac{m_j}{m} \left(1 - \frac{m_j}{m}\right).$$

- La variance V dans le cas de la régression

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

Arbres de décisions VI

Critères de séparations



Arbres de décisions VII

On choisit ensuite le couple $\theta^\star =$ (variable, valeur) tel que

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

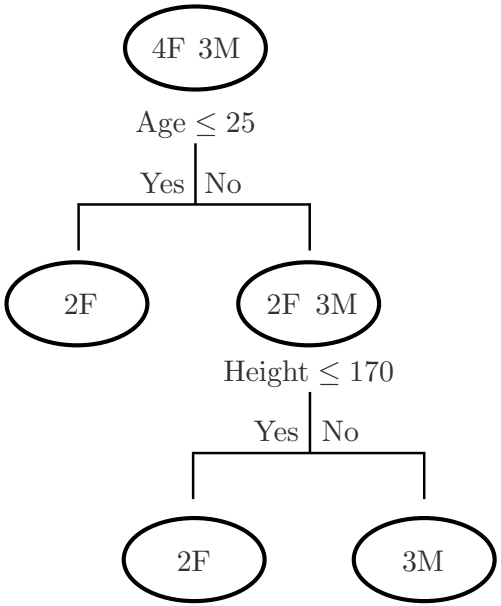
où Γ est l'un des critère précédemment mentionné.

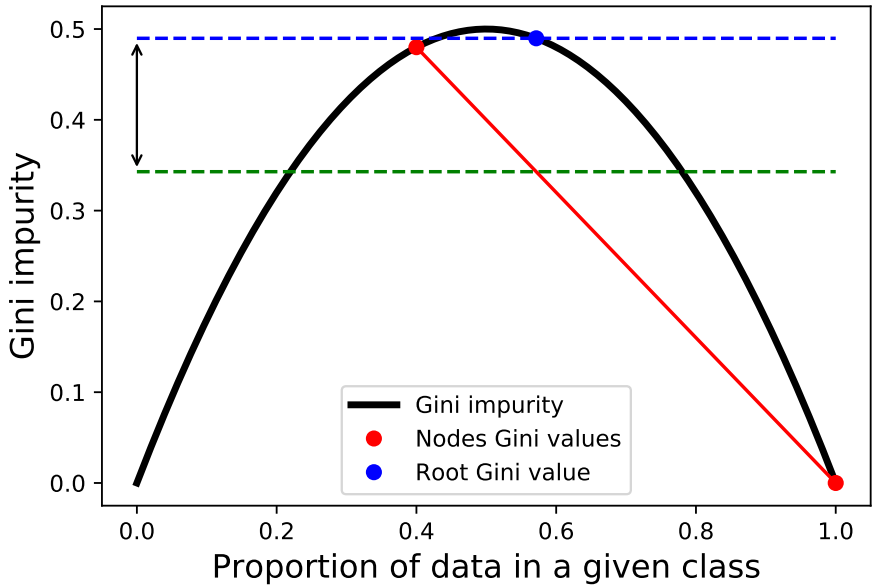
Arbres de décisions VIII

Exemples

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

$$\Gamma_{\text{node}} - \left(\frac{m_{\text{left}}}{m} \Gamma_{\text{left}} + \frac{m_{\text{right}}}{m} \Gamma_{\text{right}} \right).$$

Arbres de décisions IX

Prédiction

Quelle valeur affecter à chaque feuille ? En général on applique les règles suivantes :

- 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.

Arbres de décisions X

Pour finir

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 êtres 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)

Forêts Aléatoires I

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 de plusieurs **échantillonnages** de notre ensemble d'apprentissage.

$\rightarrow$ C'est le principe du bagging ou bootstrap aggregating

Forêts Aléatoires II

Bagging

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), mais on peut aussi décider d'attribuer un poids différent aux arbres selon leur performance.

Forêts Aléatoires III

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. Notre modèle sera donc meilleur en généralisation, *i.e.* sur des données non rencontrées jusqu'à présent.

$\rightarrow$ on peut même faire de l'échantillonnage sur les variables ! C'est le principe même des forêts aléatoires [Breiman, 2001].

Mais ce n'est pas la seule façon de faire pour combiner des modèles et éventuellement apprendre plusieurs arbres ... voyons une autre approche très efficace en pratique.

Boosting

Principe

Combiner plusieurs modèles qui ont un faible pouvoir prédictif, *i.e.* qui font à peine mieux que l'aléatoire afin de créer un modèle plus fort et ce, quelque soit la distribution de nos données.
On souhaite construire des modèles plus forts, robustes, *i.e.* des modèles, qui, s'ils sont faiblement perturbés, ne conduisent pas à une modification importante des résultats.

Cette idée du boosting répond à deux grands principes de la théorie de l'apprentissage PAC introduit à la fin des années 80' - strong et weak PAC learnability [Kearns and Valiant, 1994].

PAC Learnability I

Définition 7.1: Strong PAC learnability

Une classe de concept C est *strongly PAC learnable* avec une classe d'hypothèses $\mathcal{H}$ s'il existe un algorithme $\mathcal{A}$ tel que pour tout $c \in C$, pour toute distribution D, pour tout $\varepsilon \in (0, 1/2)$ et $\delta \in (0, 1/2)$ et ayant accès à un nombre d'exemples polynomial (en ε^{-1} et δ^{-1}) tirés de façon *i.i.d.* de D et étiquetés par c ; $\mathcal{A}$ apprend une hypothèse $h \in \mathcal{H}$ telle que $err(h) \leq \varepsilon$ avec probabilité au moins $1 - \delta$.

PAC Learnability II

Définition 7.2: Weak PAC learnability

Une classe de concept C est *weakly PAC learnable* avec une classe d'hypothèses $\mathcal{H}$ s'il existe un algorithme $\mathcal{A}$ et une valeur $\gamma > 0$ tels que pour tout $c \in C$, pour toute distribution $\mathcal{D}$, pour tout $\delta \in (0, 1/2)$ et ayant accès à un nombre d'exemples polynomial (en ε^{-1} et δ^{-1}) tirés de façon *i.i.d.* de D et étiquetés par c ; $\mathcal{A}$ apprend une hypothèse $h \in \mathcal{H}$ telle que $err(h) \leq 1/2 - \lambda$ avec probabilité au moins $1 - \delta$.

Comment pourriez-vous résumer ces deux définitions ? Est-ce que l'une implique l'autre ?

PAC Learnability III

On voir clairement qu'être strongly PAC learnable implique d'être weakly PAC learnable en regardant bien les énoncés et les résultats attendus sur les erreurs du classifieur.

Toute la question est maintenant de savoir s'il existe des algorithmes qui permettent à partir d'apprenants faibles issus d'un ensemble d'hypothèses $\mathcal{H}$ de construire un nouvel espace d'hypothèse $\mathcal{H}'$ dans lequel nous pourrions avoir un apprenant (ou une hypothèse) fort(e).

Algorithme boosting I

Principe du boosting

- On va apprendre une suite d'hypothèses $(h_t)_{t\in\mathbb{N}}$ comme ce que nous avons fait avec le bagging, c'est à dire sur des échantillons "différents".
- Les hypothèses apprises ne seront pas indépendantes, *i.e.* pas de possibilité d'apprentissage des modèles en parallèle. Nous devrons les apprendre itérativement.
- Le but est d'obtenir un apprenant fort à partir des apprenants faibles, cela peut ne peut se faire que si le modèle h_{t+1} est capable de corriger des erreurs effectuées par le modèle h_t.
- On va faire cela en pondérant, de façon itérative le poids de chaque exemple.

On va regarder un premier algorithme de boosting que l'on appelle **Adaboost** [Freund and Schapire, 1999].

Algorithme boosting II

Principe du boosting

On dispose initialement de notre échantillon S avec nos m exemples $(\mathbf{x}_i, y_i)$ et toutes les données ont **le même poids**, *i.e.* la même importance.

On va maintenant regarder comment une hypothèse h_{t+1} est apprise en fonction des performances du classifieur h_t.
Pour cela, plaçons nous à une étape t de notre algorithme où les exemples ont dont un poids égal à $w_i^{(t)}$. Une hypothèse h_t est alors apprise et nous pouvons évaluer son taux d'erreur en classification ε_t

$$\varepsilon_t = \sum_{i=1}^{m} w_i^{(t)} \mathbb{1}_{\{h_t(\mathbf{x}_i) y_i < 0\}}$$

A partir de cette erreur, nous allons déterminer une quantité α_t définie par :

Algorithme boosting III

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$$

et qui va permettre de quantifier l'importance de l'hypothèse h_t dans la décision finale, *i.e.* on va définir un poids sur le classifieur appris.
Par définition, si le classifieur appris es trop peu performant, son poids sera très faible (proche de 0) et ce poids sera d'autant plus élevé que le classifieur est bon.

Algorithme boosting IV

Le reste de la procédure consiste ensuite à trouver **une bonne repondération des exemples** de façon à ce que l'hypothèse qui sera apprise à l'itération suivante, puis se focaliser sur les erreurs commises par l'hypothèse actuelle, cela se fait par la mise à jour suivante :

$$w_i^{(t+1)} = w_i^{(t)} \frac{\exp(-\alpha_t y_i h_t(\mathbf{x}_i))}{Z_t},$$

où Z_t est un facteur de normalisation permettant d'avoir une distribution sur les poids des exemples. Nous verrons plus tard que ce facteur de normalisation est donné par $Z_t = 2\sqrt{\varepsilon_t(1-\varepsilon_t)}$. La fonction de repondration des exemples, va augmenter le poids des exempels mals classés et diminuer celui des exemples bien classés.

Algorithme boosting V

Input: Echantillon d'apprentissage S de taille m,
un nombre T de modèles
Output: Un modèle $H_T = \sum_{t=0}^{T} \alpha_t h_t$
begin
  Distribution uniforme $w_i^{(0)} = \frac{1}{m}$
  for $t = 1, ..., T$ **do**
    Apprendre un classifieur h_t à partir d'un algorithme $\mathcal{A}$
    Calculer l'erreur ε_t de l'algorithme.
    if $\varepsilon_t > 1/2$ **then**
      Stop
    else
      Calculer $\alpha_{(t)} = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$
      $w_i^{(t)} = w_i^{(t-1)} \frac{\exp(-\alpha_t y_i h_t(\mathbf{x}_i))}{Z_t}$
  Poser $H_T = \sum_{t=0}^{T} \alpha_t h_t$
  return H_T

Algorithme boosting VI

Proposition 7.1: Borne Erreur sur Adaboost

L'erreur empirique du classifieur obtenue à l'aide de Adaboost est bornée par

$$\mathcal{R}_S(H_T) \leq \exp\left[-2\sum_{t=1}^{T}\left(\frac{1}{2}-\varepsilon\right)^2\right].$$

De plus, si pour tout $t \in [\![1, T]\!]$, $\gamma \leq \left(\frac{1}{2}-\varepsilon_t\right)$, alors :

$$\mathcal{R}_S(H_T) \leq \exp\left(-2\gamma^2 T\right).$$

Algorithme boosting VII

Plan de la preuve (en exercice)

1. Exprimer $w_i^{(T+1)}$ en fonction de $w_i^{(0)} = \frac{1}{m}$.
2. Montre que $err(h) \leq \prod_{t=0}^{T} Z_t$ en utilisant ce qui précède
3. Calculer la valeur de $\alpha_{(t)}$ optimale à chaque itération
4. Montrer que $Z_t = 2\sqrt{\varepsilon_t(1-\varepsilon_t)}$
5. Conclure

A partir de ce résultat, il est possible d'obtenir des bornes en généralisation sur l'erreur de l'algorithme Adaboost [Mohri et al., 2012].

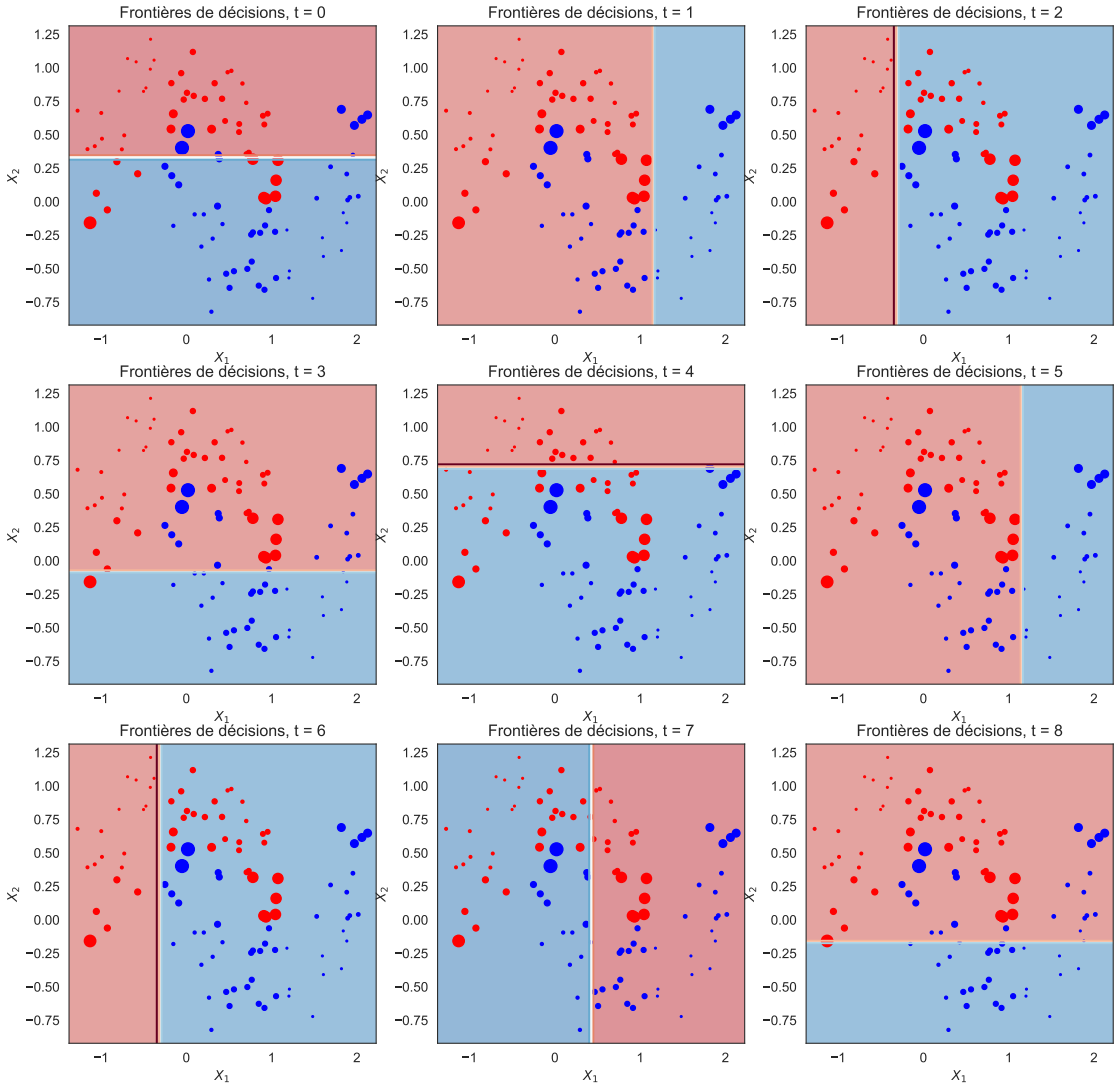
Algorithme boosting VIII

Implémentation

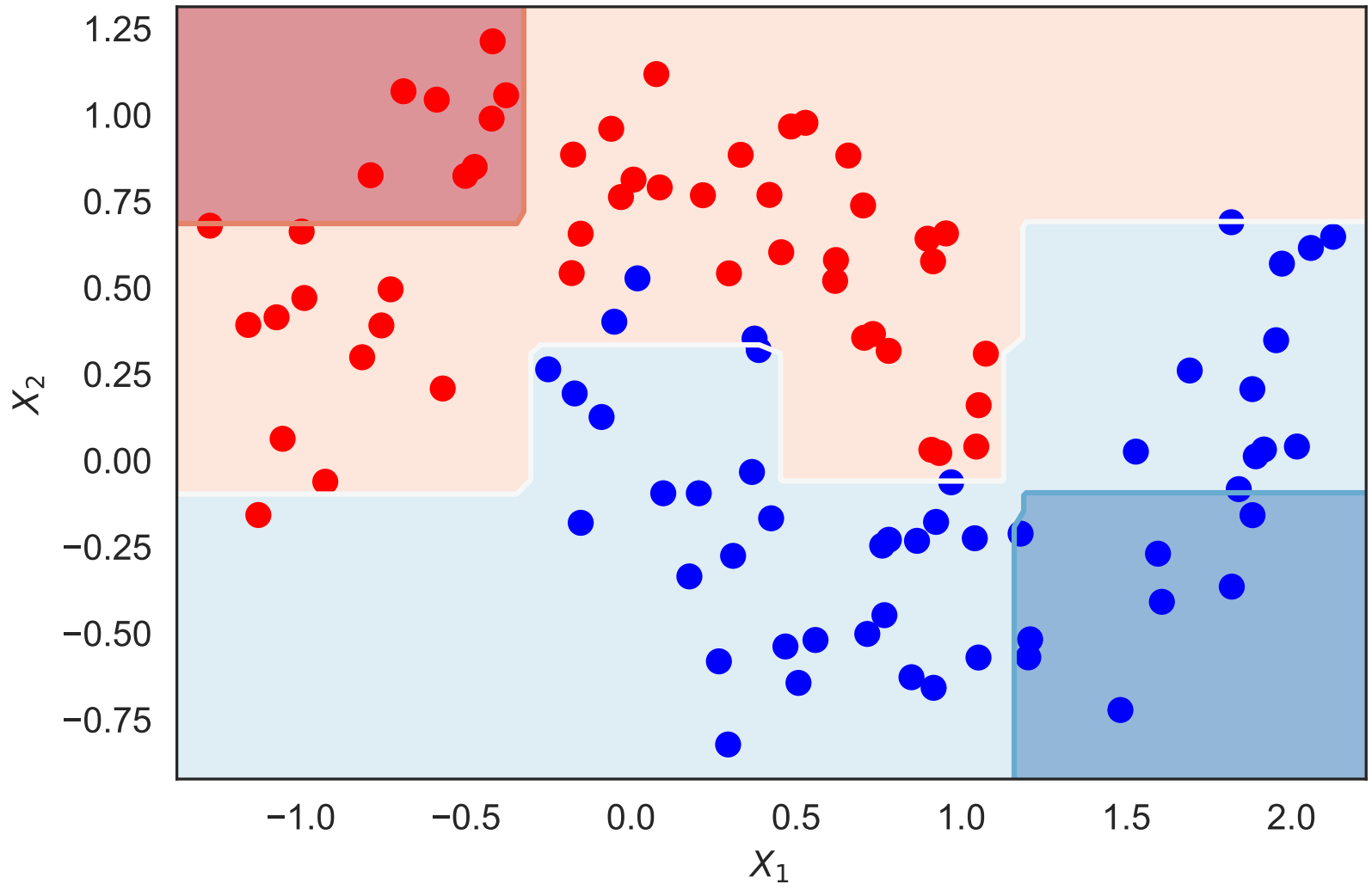
Vous pouvez regarder le fonction de la fonction **AdaboostClassifier** de la librairie *sklearn.ensemble*.
Les apprenants faibles ne sont pas spécifiés et vous pouvez utiliser n'importe quel type de modèle. Par défaut, la fonction choisit des arbres de décisions de profondeur égale à 1. Mais vous pouvez très bien choisir d'autres apprenants faibles, *e.g.* des modèles linéaires par exemple.

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.svm import SVC
svc=SVC(probability=True, kernel='linear')
abc=AdaBoostClassifier(n_estimators=50, base_estimator=svc,
learning_rate=1)
```

Algorithme boosting IX



Algorithme boosting X



Algorithme boosting XI

- Prédiction : $\text{sign}(H_T(\mathbf{x}_i)) = \text{sign}\left(\sum_{t=0}^{T} \alpha_t h_{(t)}(\mathbf{x}_i)\right)$.
- Simple d'utilisation de par sa construction
- Chaque hypothèse $h^{(t)}$ vérifie l'hypothèse *weakly PAC learnable*
- Il fournit des garanties sur la convergence de l'erreur d'entraînement (a fortiori, sur l'erreur en généralisation ou en phase de test) !

Algorithme boosting XII

On montre que le boosting :

- est performant (à la fois en théorie et en pratique)
- capable de retourner des hypothèses non linéaires à partir d'hypothèses linéaires $\rightarrow$ création de non linéarité

Mais qu'en est-il dans un contexte de Big Data ? Est-ce toujours envisageable d'utiliser une telle procédure ?

La réponse est oui mais pas forcément en utilisant un algorithme comme Adaboost et pas avec n'importe quel type de classifieur ... si on retournait sur les arbres ?

Gradient Tree Boosting ? I

- Nous avons vu plus tôt que les arbres sont capables d'apprendre de façon extrêmement précises les données (biais très faible)
- De plus, les combiner, en utilisant des méthodes d'échantillonnage, permettait de fortement réduire la variance des résultats (gain en stabilité)
- C'est un algorithme qui est peu gourmand en temps de calcul
- Mais qui est aussi facilement optimisable car parallélisable ! On ne fait que manipuler des listes.

Pour toutes ces raisons, c'est un algorithme que l'on pourrait largement privilégier dans ce contexte ... mais si on pouvait faire encore mieux ?

Gradient Tree Boosting ? II

• Après tout, on a montré que combiner plusieurs modèles ensembles permettait de créer un modèle plus fort

• Le processus itératif est relativement simple à mettre en œuvre et se concentre sur les erreurs effectuées par le modèle précédent.

Quid de cette histoire d'*hypothèses faibles* évoquée dans la présentation du boosting ?

On va simplement prendre des arbres moins profonds ! Donc avec un biais plus grand mais qui vont s'apprendre plus vite ! On va faire du *Gradient (Tree) Boosting*

Gradient Tree Boosting ? III

L'algorithme de **Gradient boosting** va se présenter comme un algorithme de descente de gradient dans un espace fonctionnelle, *i.e.* à chaque itération, nous allons apprendre un classifieur h_t en se basant sur les erreurs (que l'on appellera pseudo-résidus) effectuées par les précédents modèles.

$$H_t = H_{t-1} + \alpha_t h_t \tag{1}$$

où H_{t-1} est combinaison linéaire des $t-1$ premiers modèles avec leurs poids respectifs.

Gradient Tree Boosting ? IV

Les apprenants faibles (ou weak learner) sont entraînés sur les pseudo-résidus r_i du modèle courant. Ces pseudo-résidus sont définis comme le gradeint négatif, $-g_t$, de la fonction de loss ℓpar rapport à la prédiction courante $H_{t-1}(\mathbf{x}_i)$:

$$r_i = g_t(\mathbf{x}_i) = - \left[\frac{\partial \ell(y_i, H_{t-1}(\mathbf{x}_i))}{\partial H_{t-1}(\mathbf{x}_i)} \right].$$

Une fois les résidus r_i calculés, on résout le problème d'optimisation suivant :

$$(h_t, \alpha_t) = \arg \min_{\alpha, h} \sum_{i=1}^{m} (r_i - \alpha h(\mathbf{x}_i))^2.$$

Gradient Tree Boosting ? V

L'idée est d'apprendre un modèle, ainsi que le poids correspondant, de façon à minimiser, "combler" les erreurs effectuées par les précédentes itérations. Une fois que le modèle est appris, la mise à jour (1) est appliquée.

Gradient Boosting I

Algorithme de Gradient Boosting tel que présenté par [Friedman, 2000]

Input: Ensemble $S = \{\mathbf{x}_i, y_i\}_{i=1}^m$, une loss ℓ, nombre d'itérations T

Output: Un modèle sign $\left(H_0(\mathbf{x}) + \sum_{t=1}^T \alpha^t h_{a^t}(\mathbf{x})\right)$

begin

Hypothèse initiale $\quad H_0(\mathbf{x}_i) = \arg\min_{\rho \in \mathbb{R}} \sum_{i=1}^m \ell(y_i, \rho)\ \forall i = 1, \ldots, m$

for $t = 1, ..., T$ **do**

Calculer les pseudo-résidus :

$$\tilde{y}_i = -\frac{\partial \ell(y_i, H_{t-1}(\mathbf{x}_i))}{\partial H_{t-1}(\mathbf{x}_i)},\ \forall i = 1, ..., m$$

Apprendre un modèle pour les fitter :

$$a^t = \arg\min_{a \in \mathbb{R}^d} \sum_{i=1}^m (\tilde{y}_i - h_a(\mathbf{x}_i))^2$$

Apprendre le poids du classifieur :

$$\alpha_t = \arg\min_{a \in \mathbb{R}^+} \sum_{i=1}^m \ell(y_i, H_{t-1}(\mathbf{x}_i) + \alpha h_{a^t}(\mathbf{x}_i))$$

Mise à jour $H_t(\mathbf{x}_i) = H_{t-1}(\mathbf{x}_i) + \alpha_t h_{a^t}(\mathbf{x}_i)$

return H^t

Gradient Boosting II

En fait, il n'y en a pas ... enfin ce n'est pas tout à fait vrai ! Le gradient boosting peut être vu comme une généralisation plus flexible de notre algorithme *Adaboost* précédemment évoqué.

- *Adaboost* est basé sur la *l'exponential loss* alors que n'importe quelle loss est utilisable pour la *gradient boosting*.
- *Adaboost* travail directement sur le poids des données pour corriger les erreurs alors que le *gradient boosting* utilise le gradient

Le gradient boosting se présente en fait comme un algorithme d'optimisation dans un espace fonctionnel, l'espace des *prédictions*.

Il fonctionne comme un algorithme de descente de gradient à pas optimal (ou le plus profond).

Gradient Boosting III

Algorithme de Gradient Boosting d'un point de vue optimisation

---

Input: Ensemble $S = \{\mathbf{x}_i, y_i\}_{i=1}^m$, une loss ℓ, nombre d'itérations T

Output: Un modèle sign $\left(H_0(\mathbf{x}) + \sum_{t=0}^T \alpha_t h_{a^t}(\mathbf{x})\right)$

begin

Initialisation : $H_0(\mathbf{x}_i) = \arg\min_{\rho \in \mathbb{R}} \sum_{i=1}^m \ell(y_i, \rho)\ \forall i = 1, \ldots, m$

for $t = 1, ..., T$ **do**

Calcul du gradient : $\tilde{y}_i = -\dfrac{\partial \ell(y_i, H_{t-1}(\mathbf{x}_i))}{\partial H_{t-1}(\mathbf{x}_i)},\ \forall i = 1, ..., m$

Apprentissage d'un modèle qui approxime le gradient :

$a^t = \arg\min_{a \in \mathbb{R}^d} \sum_{i=1}^m (\tilde{y}_i - h_a(\mathbf{x}_i))^2$

Pas d'apprentissage optimal :

$\alpha_t = \arg\min_{a \in \mathbb{R}^+} \sum_{i=1}^m \ell(y_i, H_{t-1}(\mathbf{x}_i) + \alpha h_{a^t}(\mathbf{x}_i))$

Mise à jour du modèle : $H_t(\mathbf{x}_i) = H_{t-1}(\mathbf{x}_i) + \alpha_t h_{a^t}(\mathbf{x}_i)$

return H_T

Regardons maintenant un algorithme de Gradient Boosting très efficace en pratique.

XGBoost I

On va finir ces présentations sur le boosting en retournant sur nos arbres de décisions et présentant **XGBoost** [Chen and Guestrin, 2016] qui est un algorithme de *Gradient Tree Boosting*, *i.e.* qui faire du boosting sur des arbres.

- Utilise des arbres de faible profondeur (maximum 2 voire 3)
- Permet l'optimisation de n'importe quelle loss (entière customisable)
- Se base sur une approximation d'ordre 2 de la loss ℓ
- Apprentissage rapide, même sur des données volumineuses

Regardons son fonctionnement.

XGBoost II

On considère une loss ℓ (on ne fait d'hypothèse particulière dessus, sauf dérivabilité) et le problème d'optimisation suivant :

$$\min \sum_{i=1}^{m} \ell(y_i, \hat{y}_i^{(t-1)}) + \beta\mathcal{L} + \frac{\lambda}{2}\sum_{j=1}^{\mathcal{L}}(f_j^{(t)})^2. \tag{2}$$

où $\beta\mathcal{L}$ et $\frac{\lambda}{2}\sum_{j=1}^{\mathcal{L}}(h_t^{(j)})^2$ sont deux termes de régularisations qui contrôle le nombre mais aussi le poids des feuilles $f_t^{(j)}$ pour l'arbre appris à l'itération t.

Les modèles sont appris de façon additives, notons $\hat{y}^{(t-1)}$, la valeur prédite par les $t-1$ premières fonctions h_k, i.e. $\hat{y}_i^{(t-1)} = \sum_{k=1}^{t-1} h_t(\mathbf{x}_i) = H_{t-1}(\mathbf{x}_i)$.

XGBoost III

Regardons maintenant comment est appris le modèle suivant.

Pour cela, commençons par réécrire la quantité (2) à minimiser comme suit :

$$\sum_{i=1}^{m} \ell(y_i, \hat{y}_i^{(t-1)} + h_t(\mathbf{x}_i)) + \beta\mathcal{L} + \frac{\lambda}{2}\sum_{j=1}^{\mathcal{L}}(h_t^{(j)})^2. \tag{3}$$

En pratique, [Chen and Guestrin, 2016] considère simplement une approximation d'ordre de deux la fonction qu'ils souhaitent optimiser. Cette approximation d'ordre 2 est déterminée en fonction de la valeur prédite à l'itération précédente, *i.e.* $\hat{y}_i^{(t-1)}$. On notera respectivement g and f les dérivées première et seconde de notre fonction de loss ℓ par rapport à $\hat{y}^{(t-1)}$.

XGBoost IV

On peut réécrire l'équation (3) comme suit :

$$\sum_{i=1}^{m}\left[\ell(y_i,\hat{y}_i^{(t-1)})+h_t(\mathbf{x}_i)g(\mathbf{x}_i)+\frac{1}{2}h_t^2(\mathbf{x}_i)f(\mathbf{x}_i)\right]+\beta\mathcal{L}+\frac{\lambda}{2}\sum_{j=1}^{\mathcal{L}}(f_t^{(j)})^2. \tag{4}$$

Rappellons que l'on souhaite apprendre la fonction $\mathbf{f}_t=(h_t^{(j)})_{j=1,\dots,\mathcal{L}}$. Considérons alors une feuille j et notons I_j l'ensemble des indices i tels que $\mathbf{x}_i$ appartiennent à la feuille l_j. Ainsi, en utilisant (4), la fonction $h_t^{(j)}$ doit minimiser la quantité V_j pour un certain indice j :

$$V_j \;=\; \sum_{i\in I_j}\left[g(\mathbf{x}_i)h_t^{(j)}(\mathbf{x}_i)+\frac{1}{2}\left(\lambda+f(\mathbf{x}_i)\right)(h_t^{(j)}(\mathbf{x}_i))^2\right]. \tag{5}$$

XGBoost V

Cette fonction est convexe et admet donc un minimum, donné en résolvant l'*Equation d'Euler*, *i.e.* la fonction $f^{(t)}$ pour laquelle le gradient s'annule. Cette solution est donnée par :

$$h_t^{(j)} = -\frac{\sum_{i \in I_j} g(\mathbf{x}_i)}{\sum_{i \in I_j} f(\mathbf{x}_i) + \lambda}. \tag{6}$$

Focalisons nous maintenant sur la façon dont on va effectuer un split, une séparation à un noeud donné. Maintenant que l'on a trouvé le poids optimal à appliquer à chaque feuille (6), on peut calculer la valeur optimal $V_j^\star$ de la loss en utilisant (5). Ce qui nous donne

XGBoost VI

$$V_j^\star = \sum_{i\in I_j}\left[\underbrace{-g(\mathbf{x}_i)\frac{\sum_{i\in I_j} g(\mathbf{x}_i)}{\sum_{i\in I_j} f(\mathbf{x}_i)+\lambda}}\right.$$

$$\left.\underbrace{+\frac{1}{2}\left(\lambda + f(\mathbf{x}_i)\right)\left(-\frac{\sum_{i\in I_j} g(\mathbf{x}_i)}{\sum_{i\in I_j} f(\mathbf{x}_i)+\lambda}\right)^2}\right],$$

$$= -\frac{\left(\sum_{i\in I_j} g(\mathbf{x}_i)\right)^2}{\sum_{i\in I_j} f(\mathbf{x}_i)+\lambda} + \frac{1}{2}\frac{\left(\sum_{i\in I_j} g(\mathbf{x}_i)\right)^2}{\sum_{i\in I_j} f(\mathbf{x}_i)+\lambda},$$

$$V_j^\star = -\frac{1}{2}\frac{\left(\sum_{i\in I_j} g(\mathbf{x}_i)\right)^2}{\sum_{i\in I_j} f(\mathbf{x}_i)+\lambda}.$$

XGBoost VII

Cette formule est utilisée pour pour mesurer la pureté d'une feuille. Elle peut se voir comme une généralisation de l'impureté de Gini mais pour n'importe quelle loss ℓ. En utilisant cette nouvelle mesure, ils définissent ensuite leur critère de séparation, *i.e.* le gain associé à chaque séparation :

$$\frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g(\mathbf{x}_i)\right)^2}{\sum_{i\in I_L} f(\mathbf{x}_i)+\lambda}+\frac{\left(\sum_{i\in I_R} g(\mathbf{x}_i)\right)^2}{\sum_{i\in I_R} f(\mathbf{x}_i)+\lambda}-\frac{\left(\sum_{i\in I} g(\mathbf{x}_i)\right)^2}{\sum_{i\in I} f(\mathbf{x}_i)+\lambda}\right]-\beta,$$

où $I = I_L \cup I_R$ pour un arbre binaire and le paramètre β est utilisé pour contrôlé le nombre de feuilles

XGBoost VIII

Implémentation

Deux références à consulter pour l'implémentation de XGBoost : implémentation classique

Exemple utilisation

et customisation, définir sa propre loss et sa propre métrique d'évaluation

Loss et Métrique personnelles

Projet

A propos du projet I

Fraude par chèque ?

- Impayé (solde non disponible sur le compte)
- Faux chèque
  - Identité non réelle
  - Série de caractères incorrects dans la ligne CMC7

Quelques statistiques :

10 mois de transactions (20/01/2016 to 21/10/2016)

- environ 3.2 millions de transactions
- pour 195 millions d'euros
- 20 000 fraudes ou impayées (0.6%)
- représentent 2 millions d'euros (1.1%)

Apprentissage déséquilibré et Big Data

A propos du projet II

Les tâches quotidiennes de l'entreprise, valables pour d'autres entreprises

- gérer l'intégration continue des données avec les clients
- gérer la base de données (période + endroit de stockage)
- gérer un flux massif de données permanent
- apprendre à détecter des fraudes
- avoir des modèles qui répondent rapidement (20 ms environ) mais s'apprennent aussi en un temps "raisonnable"

Les données en quelques chiffres

- La base de données utilisée comporte environ 2.3 millions de transactions chacune identifiée par un ID.
- On utilisera un ensemble de 23 descripteurs variés relatifs à la transactions (détails fourni un peu plus tard). L'ensemble des descripteurs n'est pas à utiliser pour construire votre modèle, certains sont uniquement là à titre informatif. Il y a essentiellement des descripteurs quantitatifs, vous aurez aussi des variables ordinales et une variable catégorielle, à vous de voir si elles sont utiles et comment vous pouvez les utiliser.
- Taux de fraude dans les données 0.28% (en nombre) et 0.44% en montant, ce qui représente quand même un montant de fraude de $600k$ euros environ.

Quelques éléments I

• N'hésitez pas à consulter le tutoriel "sklearn" qui vous permettra de vous donner des éléments de base pour construire un modèle (point de vue code) :

https://scikit-learn.org/stable/tutorial/basic/tutorial.html

• Je vous invite aussi à consulter la librairie "imbalanced-learn" de python qui vous permettra de mettre facilement en place quelques méthodes abordées en classe comme les méthodes de sampling

https://imbalanced-learn.readthedocs.io/en/stable/user_guide.html

• Consultez aussi d'autres sites comme "Kaggle", "Medium" ou "MachineLearningMastery" qui permettront de trouver quelques idées + codes pour effectuer le projet.

Quelques éléments II

- XGBoost :

Implémentation XGBoost

- Nearest-Neighbor :

Implémentation Plus Proches Voisins

- Decision Trees

Implémentation arbres de décisions

- Random Forests

Implémentation forêts aléatoires

- SVM

Implémentation SVM

Quelques éléments III

- K-means

Implémentation K-means

- Sampling

Tutoriel Kaggle + code pour échantillonnage

- LOF

Calcul d'un score d'anormalité

- Auto-encodeurs

Implémentation auto-encodeurs

- Réseaux de neurones

Construction réseaux de neurones pas à pas

Quelques éléments IV

- Analyse discriminante linéaire

Implémentation LDA

- Analyse discriminante quadratique

Code QDA

- Cost-sensitive learning

Tutoriel cost-sensitive learning

- Méthodes ensemblistes

Boosting - Bagging

- Metric Learning

Un package pour le Metric Learning

Quelques conseils pour finir I

Soyez créatifs !

Il ne s'agit pas d'appliquer bêtement et simplement un algorithme existant comme un simple *Séparateur à Vaste Marge* et de dire que cela fonctionne ou non (déjà cela ne donnera pas de bons résultats, le problème est plus complexe).

→ il faut penser à combiner plusieurs modèles ensembles

- le résultat de plusieurs SVM avec une méthode d'échantillonnage (under ou over sampling),
- les combiner à l'aide d'une méthode de boosting
- pondérer les classes
- combiner SVM + pondération des classes + échantillonnage + ...
- les combinaisons sont multiples

Quelques conseils pour finir II

- L'algorithme standard/classique doit vous servir de baseline dans la construction de votre modèle.
- A vous de voir comment combiner un algorithme simple à d'autres méthodes pour améliorer ses performances.
- Rien ne vous interdit de combiner les résultats (bagging) de différents algorithmes : on peut imaginer créer un score à partir d'une LDA et ensuite créer un modèle à partir de forêts aléatoires et apprendre un modèle (ou construire un process) qui combine intelligemment les résultats de ces deux algorithmes.
- Rien ne vous empêche aussi de combiner des méthodes non supervisées (auto-encodeurs ou k-means) avec des algorithmes supervisés (SVM, Decision Trees, RF, ...)

Quelques conseils pour finir

- Les possibilités sont multiples, ne pas se priver et soyez imaginatifs et n'ayez surtout pas peur de tester, même si cela ne donne rien.
- La solution magique n'existe pas, il faut essayer et construire pas à pas votre modèle.
- N'hésitez pas à indiquer les démarches effectuées dans votre rapport en conservant les résultats des différentes expériences jusqu'à l'obtention de votre modèle final.
- Enfin ... soyez cohérent dans les expériences effectuées. Motivez le choix des algorithmes utilisés et pourquoi il est cohérent des les comparer entre eux.

Dernière chose ...

Les données que vous allez utiliser son datées, vous ne pourrez donc pas cross-valider vos modèles n'importe comment !

Pensez à définir correctement vos ensembles d'entraînement et de validation. L'ensemble de test, lui, sera fixé et vous est donné dans le sujet qui vous présente le projet ainsi que les caractéristiques des données.

Références I


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.


Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002).
Smote : Synthetic minority over-sampling technique.
Journal of Artificial Intelligence Research, 16(1) :321–357.

Références II


Chen, T. and Guestrin, C. (2016).
Xgboost : A scalable tree boosting system.
In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794. ACM.


Elkan, C. (2001).
The foundations of cost-sensitive learning.
In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2*, pages 973–978, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.


Fernández, A., Garcia, 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.

Références III


Freund, Y. and Schapire, R. E. (1999).
A short introduction to boosting.
In *In Proceedings of the Sixteenth IJCAI*, pages 1401–1406. Morgan Kaufmann.


Friedman, J. H. (2000).
Greedy function approximation : A gradient boosting machine.
Annals of Statistics, 29 :1189–1232.


Hart, P. (1968).
The condensed nearest neighbor rule.
IEEE Transactions on Information Theory, 14(3) :515–516.

Références IV


Kearns, M. and Valiant, L. (1994).
Cryptographic limitations on learning boolean formulae and finite automata.
Journal of the ACM (JACM), 41(1) :67–95.


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.


Tomek, I. (1976).
Two modifications of cnn.
IEEE Trans. Systems, Man and Cybernetics, 6 :769–772.

Références V


Wilson, D. L. (1972).
Asymptotic properties of nearest neighbor rules using edited data.
IEEE Transactions on Systems, Man, and Cybernetics,
SMC-2(3) :408–421.