Machine Learning
TD - Méthodes Ensemblistes

M2 Informatique - BI&A

Guillaume Metzler

Institut de Communication (ICOM)
Université de Lyon, Université Lumière Lyon 2
Laboratoire ERIC UR 3083, Lyon, France

guillaume.metzler@univ-lyon2.fr

Abstract

L'objectif de ce dernier TD est de vous faire travailler sur les méthodes ensemblistes. Si les méthodes ensemblistes reposant sur du bagging sont déjà connues pour la plupart, on pensera notamment à l'algorithme des Forêts Aléatoires, il existe une autre façon de combiner les modèles que l'on appelle le *Boosting*.

On se propose d'étudier des algorithmes dans le cadre de ce TP, en travaillant plus spécifiquement sur Adaboost afin de voir son fonctionnement mais aussi de mettre cet algortihme en pratique.

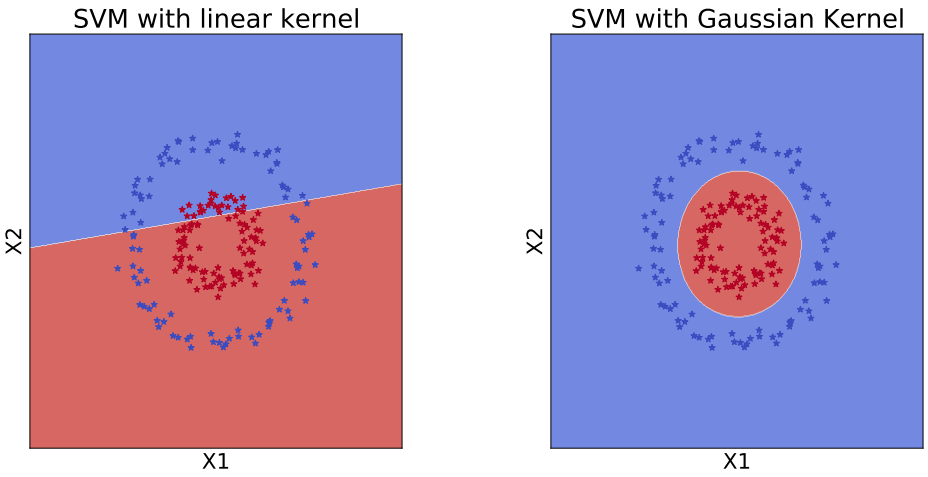

Figure 1: Illustration du pouvoir prédictif d'un modèle linéaire versus celui d'un modèle non linéaire

1 Introduction

Travailler avec des modèles simples ne permet pas de résoudre des tâches complexes, comme des problèmes de classification non linéaires par exemple.

Nous avons déjà pu mettre en oeuvre des algorithmes qui permettent de de résoudre des tâches complexes en utilisant des approches non linéaires, comme les méthodes à noyaux, *i.e.* SVM à noyau gaussien, pour résoudre de telles tâches (voir Figure 1).

On souhaite, à travers ce TD, savoir s'il existe une façon, à partir de modèles simples (comme les modèles linéaires) de construire des modèles beaucoup plus complexes et qui sont capables de résoudre des tâches plus complexes, comme des problèmes non linéairement séparables.

Pour faire cela, nous allons explorer des solutions du côté des *méthodes ensemblistes*.

2 Méthodes ensemblistes

Les méthodes ensemblistes sont des méthodes qui consistent à combiner plusieurs modèles afin de pouvoir générer un nouveau modèle qui soit potentiellement plus *performant* mais aussi plus *robuste*. Cette combinaison s'effectue généralement de deux façons :

- via du *bagging*
- ou du *boosting*

2.1 Bagging

Présentation Un algorithme emblématique reposant du bagging est celui des forêts aléatoires. En quelques mots, les méthodes de bagging consistent à combiner des modèles qui sont appris à partir d'informations différentes, *i.e.* à partir d'échantillons différents. Ces modèles sont ensuite combinés, généralement sommés, afin de créer un seul et uniquement modèle performant.

On considère un échantillon d'apprentissage S et on effectue la procédure suivante T fois, où T représente le nombre de modèles que l'on souhaite apprendre : (i) on tire un échantillon S_t avec remise à partir de S, (ii) on apprend un modèle h_t en en utilisant l'échantillon S_t.

Une fois que les T modèles sont appris, on se retrouve avec un modèle global H_T que l'on écrira :

$$H_T = \frac{1}{T} \sum_{t=1}^{T} h_t.$$

L'exemple le plus simple est celui de la combinaisons d'arbres pour construire une *forêt aléatoire* [Breiman, 2001].

1. Que peut-on dire des échantillons S_t vis-à-vis de S ?
2. Quel est l'avantage de cette procédure d'apprentissage de modèle sur le plan algorithmique, si on prend en compte le fait que chaque modèle est indépendant des autres ?
3. Quel est l'avantage d'apprendre des modèles avec des informations différentes ?

Implémentation Vous pouvez apprendre un modèle ensembliste en utilisant une méthode de *Bagging* à partir en utilisant la librairie `sklearn.ensemble` et plus précisément la fonction `BaggingClassifier`.

Cette fonction dépend de plusieurs arguments

- `base_estimator` : choix du modèle de base à partir duquel on effectuera notre combinaison, on pourra choisir n'importe quel classifieur ou régresseur (voir exemple après)
- `n_estimators` : nombre de modèles que l'on souhaite apprendre, il faut donc spécifier un nombre entier

- `max_features` : on pourra spécifier quelle proportion des variables employer pour l'apprentissage d'un modèle
- `oob_score` : *True* : calcul ce que l'on appelle l'*out of bag error* qui permet d'avoir une idée des performances du modèle en généralisation.

1. Quel est l'intérêt de faire de l'échantillonnage sur les variables ?
2. A quoi pourrait bien servir, sur le plan pratique, le paramètre `oob_score` dans la fonction `BaggingClassifier` de Python.

Exemples On donne ici un exemple qui permet d'apprendre un classifieur ensembliste, fondé sur la bagging, à partir de SVM linéaires[1].

```
from sklearn.svm import SVC
from sklearn.ensemble import BaggingClassifier
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=100, n_features=4, n_informative=2,
    n_redundant=0,random_state=0, shuffle=False)
clf = BaggingClassifier(base_estimator=SVC(), n_estimators=10, random_state=0).fit(X, y)
```

Un autre exemple où l'on effectue cette fois-ci une combinaison d'arbres de décisions *DecisionTreeClassifier* (ce qui correspond au paramètre par défaut de `base_estimator`).

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=100, n_features=4, n_informative=2,
    n_redundant=0,random_state=0, shuffle=False)
clf = BaggingClassifier(n_estimators=10, random_state=0).fit(X, y)
```

Mise en pratique

1. En considérant un jeu de données de votre choix et en prenant un classifieur de base qui soit un SVM et/ou arbre de décision, représenter votre d'erreur en entraînement et en test en fonction du paramètre `n_estimators` de la méthode de Bagging. Décrivez vos observations.

[1] Cet exemple est directemetn extraît de l'exemple présenté à l'adresse suivante : https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html.

2.2 Boosting

Si précédemment les modèles ont été appris les uns indépendamment des autres, l'idée du boosting est de rompre cette indépendance.

On suppose que les modèles appris ont un faible pouvoir prédictif et on souhaite créer une combinaison de ces modèles afin que cette dernière soit plus performante. Pour faire cela, on va apprendre nos modèles de façon itérative et cela, de sorte que le modèle appris à l'itération actuelle soit capable de corriger les erreurs effectuées par le modèle précédent.

Le premier algorithme de boosting qui a été développé et qui répond à cette problématique est l'algorithme *Adaboost* [Freund et al., 1999].
La présentation de cette procédure est donnée par l'Algorithme 1 et se présente comme suit.

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 :

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

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.

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 repondération des exemples, va augmenter le poids des exemples mal classés et diminuer celui des exemples bien classés.

Algorithm 1: Adaboost algorithm [Freund et al., 1999]

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

1. Expliquer le lien entre les valeurs de α_t et les performances de l'algorithme.
2. Expliquer le fonctionnement de la repondération des exemples.
3. A quelle loss classique vous fait penser la loss qui est utilisée lors de la repondération des erreurs.

L'algorithme Adaboost peut illustrer graphiquement cette algorithme comme présenté en Figure 2 et vérifie la propriété suivante :

Proposition 2.1: 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 :

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

Cette proposition, et plus précisément la démonstration de cette dernière permet d'expliquer la pondération des différents exemples au cours des itérations, *i.e.* les $w_i^{(t)}$ mais aussi la façon dont est calculée l'importance d'un classifieur α_t pour ces mêmes itérations.

1. En regardant de plus près le résultat de cette proposition, que devrions nous observer, sur le risque, dans le cas où l'on apprend une infinité de modèles avec Adaboost ?
2. Est-ce faisable en pratique ? Qu'est-ce qui pourrait empêcher cela ?

Implémentation Vous pouvez apprendre un modèle ensembliste en utilisant une méthode de type *Boosting* à partir en utilisant la librairie `sklearn.ensemble` et plus précisément la fonction `AdaBoostClassifier`.

Cette fonction dépend de plusieurs arguments, mais on se contentera uniquement des présentés ci-dessous :

- `base_estimator` : choix du modèle de base à partir duquel on effectuera notre combinaison, on pourra choisir n'importe quel classifieur ou régresseur (voir exemple après)
- `n_estimators` : nombre de modèles que l'on souhaite apprendre, il faut donc spécifier un nombre entier

Exemple On donne ici un exemple qui permet d'apprendre un classifieur ensembliste, fondé sur la bagging, à partir de SVM linéaires[2].

[2]Cet exemple est directemetn extraît de l'exemple présenté à l'adresse suivante : https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html.

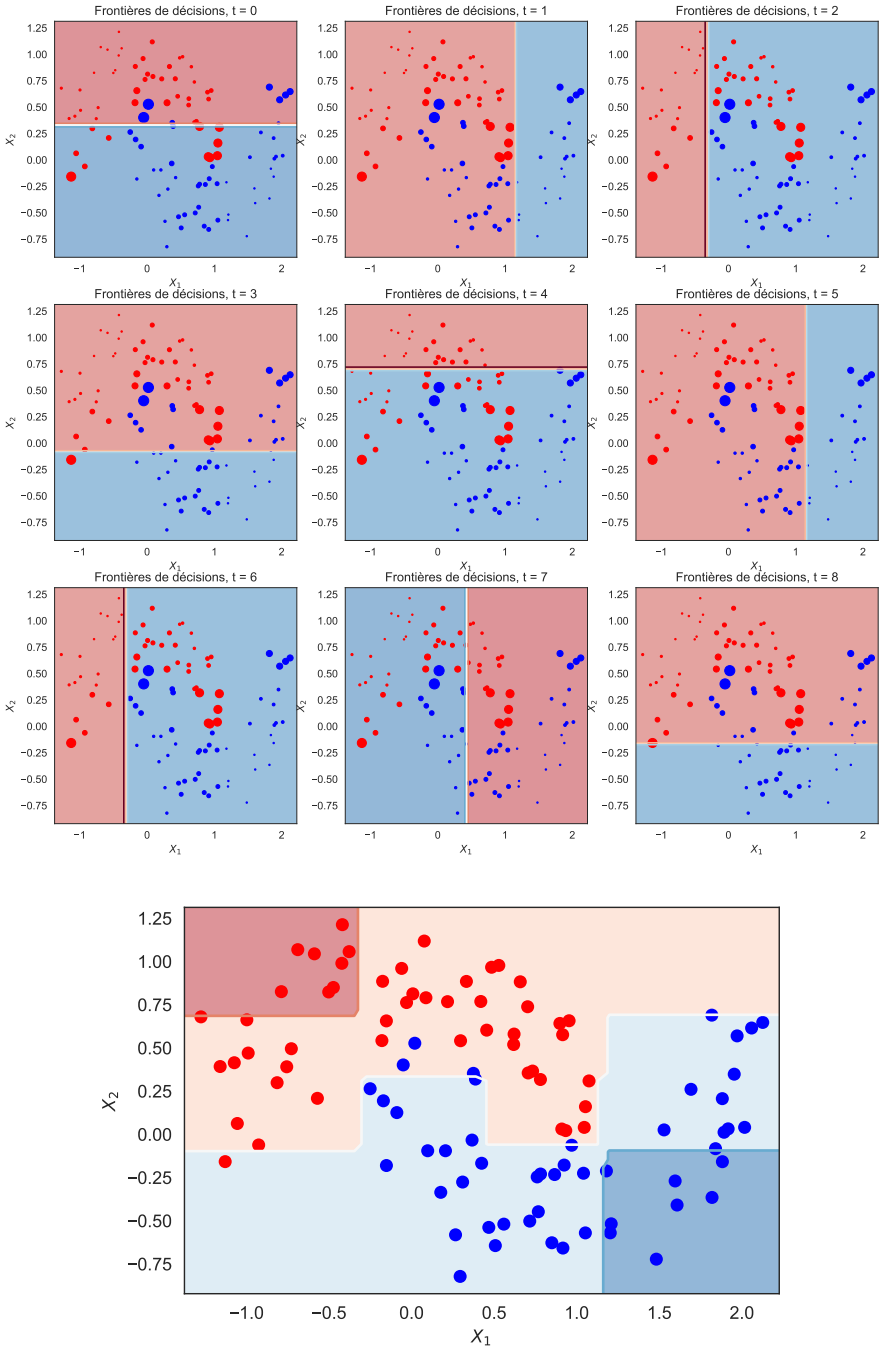

Figure 2: Illustration des différentes itérations de l'algorithme Adaboost lorsque l'on apprend un ensemble de 9 hypothèses

```
from sklearn.svm import SVC
from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=100, n_features=4, n_informative=2,
    n_redundant=0,random_state=0, shuffle=False)
clf = AdaBoostClassifier(base_estimator=SVC(), n_estimators=10, random_state=0).fit(X,
↪  y)
```

Un autre exemple où l'on effectue cette fois-ci une combinaison d'arbres de décisions *DecisionTreeClassifier* (ce qui correspond au paramètre par défaut de `base_estimator`).

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=100, n_features=4, n_informative=2,
    n_redundant=0,random_state=0, shuffle=False)
clf = AdaBoostClassifier(n_estimators=10, random_state=0).fit(X, y)
```

Mise en pratique

1. En considérant un jeu de données de votre choix (ou celui qui est donné ci-dessous) et en prenant un classifieur de base qui soit un SVM et/ou arbre de décision, représenter votre d'erreur en entraînement et en test en fonction du paramètre `n_estimators` de la méthode de Bagging. Décrivez vos observations.

   ```
   from sklearn.ensemble import AdaBoostClassifier
   from sklearn.datasets import make_moons
   X, y = make_moons(n_samples=400, noise=0.2)
   ```

2. Recoder vous même l'algorithme Adaboost en vous basant sur la présentation donnée en Algorithme 1.

References

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

[Freund et al., 1999] Freund, Y., Schapire, R., and Abe, N. (1999). A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612.