Agrégation de modèles

Résumé

*Les algorithmes décrits sont basés sur des stratégies adaptatives (*boosting, gradient boosting*) ou aléatoires (*bagging, random forest*) permettant d'améliorer l'ajustement par une combinaison ou agrégation d'un grand nombre de modèles tout en évitant ou contrôlant le sur-ajustement. Définitions, optimisation et principes d'utilisation de ces algorithmes.*

Retour au plan du cours

1 Introduction

Deux types d'algorithmes sont abordés. Ceux reposants sur une construction aléatoires d'une famille de modèles : *bagging* pour *bootstrap aggregating* (Breiman 1996)[2] et les forêts aléatoires (*random forests*) de Breiman (2001)[4] qui propose une amélioration du *bagging* spécifique aux modèles définis par des arbres binaires (CART). Ceux basés sur le *boosting* (Freund et Shapiro,1996)[8] et qui reposent sur une construction *adaptative*, déterministe ou aléatoire, d'une famille de modèles. Ces algorithmes se sont développés à la frontière entre apprentissage machine (*machine learning*) et Statistique. De nombreux articles comparatifs montrent leur efficacité sur des exemples de données simulées et surtout pour des problèmes réels complexes (Fernandez-Delgado et al. 2014)[7]). Elles participent régulièrement aux solutions gagnantes des concours de prévisions de type *Kaggle* et leurs propriétés théoriques sont un thème de recherche toujours actif.

Les principes du *bagging* ou du *boosting* s'appliquent à toute méthode de modélisation (régression, CART, réseaux de neurones) mais n'ont d'intérêt, et réduisent sensiblement l'erreur de prévision, que dans le cas de modèles *instables*, donc plutôt non linéaires. Ainsi, l'utilisation de ces algorithmes n'a guère de sens avec la régression multilinéaire ou l'analyse discriminante. Ils sont surtout mis en œuvre en association avec des arbres binaires comme modèles de base. En effet, l'instabilité déjà soulignés des arbres apparaît alors comme une propriété nécessaire à la réduction de la variance par agrégation de modèles.

La présentation des ces algorithmes toujours en évolution nécessite une mise à jour continuelle entrainant une progression chronologique des versions historiques (*bagging, adaboost*) à l'*extrem gradient boosting*. Ce choix ou plutôt l'adaptation à cette contrainte n'est sans doute pas optimal mais présente finalement quelques vertus pédagogiques en accompagnant l'accroissement de la complexité des algorithmes présentés.

2 Famille de modèles aléatoires

2.1 *Bagging*

Principe et algorithme

Soit Y une variable à expliquer quantitative ou qualitative, $X^1, \ldots, X^p$ les variables explicatives et $f(\mathbf{x})$ un modèle fonction de $\mathbf{x} = \{x^1, \ldots, x^p\} \in \mathbb{R}^p$. On note n le nombre d'observations et

$$\mathbf{z} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$$

un échantillon de loi F.

Considérant B *échantillons indépendants* notés $\{\mathbf{z}_b\}_{b=1,B}$, une prévision par *agrégation de modèles* est définie ci-dessous dans le cas où la variable à expliquer Y est :

- quantitative : $\widehat{f}_B(.) = \frac{1}{B}\sum_{b=1}^{B} \widehat{f}_{\mathbf{z}_b}(.)$,
- qualitative : $\widehat{f}_B(.) = \arg\max_j \text{card}\left\{b \mid \widehat{f}_{\mathbf{z}_b}(.) = j\right\}$.

Dans le premier cas, il s'agit d'une simple moyenne des résultats obtenus pour les modèles associés à chaque échantillon, dans le deuxième, un *comité* de modèles est constitué pour *voter* et *élire* la réponse la plus probable. Dans ce dernier cas, si le modèle retourne des probabilités associées à chaque modalité comme en régression logistique ou avec les arbres de décision, il est aussi simple de calculer des moyennes de ces probabilités.

Le principe est élémentaire, moyenner les prévisions de plusieurs modèles indépendants permet de *réduire la variance* et donc de réduire l'erreur de prévision.

Cependant, il n'est pas réaliste de considérer B échantillons indépendants.

Cela nécessiterait généralement trop de données. Ces échantillons sont donc remplacés par B réplications d'échantillons *bootstrap* obtenus chacun par n tirages avec remise selon la mesure empirique $\widehat{F}$. Ceci conduit à l'algorithme ci-dessous.

Algorithm 1 *Bagging*

Soit $\mathbf{x}_0$ à prévoir et
$\mathbf{z} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ un échantillon
for $b = 1$ à B **do**
  Tirer un échantillon bootstrap $\mathbf{z}_b^*$.
  Estimer $\widehat{f}_{\mathbf{z}_b}(\mathbf{x}_0)$ sur l'échantillon bootstrap.
end for
Calculer l'estimation moyenne $\widehat{f}_B(\mathbf{x}_0) = \frac{1}{B}\sum_{b=1}^{B} \widehat{f}_{\mathbf{z}_b}(\mathbf{x}_0)$ ou le résultat du vote.

Erreur out-of-bag

Il est naturel et techniquement facile d'accompagner ce calcul par une estimation *out-of-bag* (*o.o.b.*) de l'erreur de prévision car sans biais, ou plutôt pessimiste, comme en validation croisée.

Erreur *o.o.b.* : Pour chaque observation $(y_i, \mathbf{x}_i)$ considérer les seuls modèles estimés sur un échantillon *bootstrap* ne contenant pas cette observation (à peu près 1/3). Prévoir la valeur $\widehat{y}$ comme précédemment (moyenne ou vote) et calculer l'erreur de prévision associée ; moyenner sur toute les observations.

Utilisation

En pratique, CART est souvent utilisée comme méthode de base pour construire une famille de modèles c'est-à-dire d'arbres binaires. L'effet obtenu, par moyennage d'arbres, est une forme de "lissage" du pavage de l'espace des observations pour la construction des règles de décision. Trois stratégies d'élagage sont possibles :

1. laisser construire et garder un arbre complet pour chacun des échantillons en limitant le nombre minimale (5 par défaut) d'observation par feuille ;
2. construire un arbre d'au plus q feuilles ou de profondeur au plus q ;
3. construire à chaque fois l'arbre complet puis l'élaguer par validation croisée.

La première stratégie semble en pratique un bon compromis entre volume des calculs et qualité de prévision. Chaque arbre est alors affecté d'un faible biais et d'une grande variance mais la moyenne des arbres réduit avantageusement celle-ci. En revanche, l'élagage par validation croisée pénalise les calculs sans, en pratique, gain substantiel de qualité.

Cet algorithme a l'avantage de la simplicité, il s'adapte et se programme facilement quelque soit la méthode de modélisation mise en œuvre. Il pose néanmoins quelques problèmes :

- temps de calcul pour évaluer un nombre suffisant d'arbres jusqu'à ce que l'erreur de prévision *out-of-bag* ou sur un échantillon validation se stabilise et arrêt si elle tend à augmenter ;
- nécessiter de stocker tous les modèles de la combinaison afin de pouvoir utiliser cet outil de prévision sur d'autres données,
- l'amélioration de la qualité de prévision se fait au détriment de l'interprétabilité. Le modèle finalement obtenu devient une *boîte noire*.

2.2 Forêts aléatoires

Motivation

Dans les cas spécifique des modèles d'arbres binaires de décision (CART), Breiman (2001)[4] propose une amélioration du *bagging* par l'ajout d'une composante aléatoire. L'objectif est donc de rendre plus *indépendants* les arbres de l'agrégation en ajoutant du hasard dans le choix des variables qui interviennent dans les modèles. Depuis la publication initiale de l'algorithme, cette méthode a beaucoup été testée, comparée (Fernandez-Delgado et al. 2014)[7], (Caruana et al. 2008[5]), analysée. Elle devient dans beaucoup d'articles d'apprentissage machine la méthode à battre en matière de qualité de prévision alors que ses propriétés théoriques de convergence, difficiles à étudier, commencent à être publiées (Scornet et al. 2015)[13]. Néanmoins elle peut conduire aussi à de mauvais résultats notamment lorsque le problème sous-jacent est linéaire et donc qu'une simple régression PLS conduit à de bonnes prévisions même en grande dimension. C'est le cas par exemple de données de spectrométrie en proche infra-rouge (NIR).

Plus précisément, la variance de la moyenne de B variables indépendantes,

identiquement distribuées, chacune de variance σ^2, est σ^2/B. Si ces variables sont identiquement distribuées mais en supposant qu'elles sont corrélées deux à deux de corrélation ρ, Breiman (2001)[4] montre que la variance de la moyenne devient :

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2.$$

Comme dans le cas indépendant, le 2ème terme décroît avec B mais le premier limite considérablement l'avantage du *bagging* si la corrélation est élevée. C'est ce qui motive principalement la *randomisation* introduite dans l'algorithme ci-dessous afin de réduire ρ entre les prévisions fournies par chaque modèle.

Algorithme

Le *bagging* est appliqué à des arbres binaires de décision en ajoutant un tirage aléatoire de m variables explicatives parmi les p.

Algorithm 2 Forêts Aléatoires

Soit $\mathbf{x}_0$ à prévoir et
$\mathbf{z} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ un échantillon
for $b = 1$ à B **do**
  Tirer un échantillon bootstrap $\mathbf{z}_b^*$
  Estimer un arbre sur cet échantillon avec **randomisation** des variables : la recherche de chaque division optimale est précédée d'un tirage aléatoire d'un sous-ensemble de m prédicteurs.
end for
Calculer l'estimation moyenne $\widehat{f}_B(\mathbf{x}_0) = \frac{1}{B}\sum_{b=1}^{B} \widehat{f}_{\mathbf{z}_b}(\mathbf{x}_0)$ ou le résultat du vote.

Paramètres de l'algorithme

La stratégie d'élagage peut, dans le cas des forêts aléatoires, être plus élémentaire qu'avec le *bagging* en se limitant à des arbres de taille q relativement réduite voire même triviale avec $q = 2$ (*stump*). En effet, avec le seul *bagging*, des arbres limités à une seule fourche risquent d'être très semblables (fortement corrélés) car impliquant les mêmes quelques variables apparaissant comme les plus explicatives. Dans la stratégie par défaut de l'algorithme, c'est simplement le nombre minimum d'observation par nœuds qui limite la taille de l'arbre, il est fixé à par défaut à 5. Ce sont donc des arbres plutôt complets qui sont considérés, chacun de faible biais mais de variance importante.

La sélection aléatoire d'un nombre réduit de m prédicteurs potentiels à chaque étape de construction d'un arbre, accroît significativement la variabilité en mettant en avant nécessairement d'autres variables. Chaque modèle de base est évidemment moins performant, sous-optimal, mais, l'union faisant la force, l'agrégation conduit finalement à de bons résultats. Le nombre m de variables tirées aléatoirement peut, selon les exemples traités, être un paramètre sensible avec des choix par défaut pas toujours optimaux :

- $m = \sqrt{p}$ dans un problème de classification,
- $m = p/3$ dans un problème de régression.

Comme pour le *bagging*, l'évaluation itérative de l'erreur *out-of-bag* permet de contrôler le nombre B d'arbres de la forêt de même qu'éventuellement optimiser le choix de m. C'est néanmoins une procédure de validation croisée qui est préférablement opérée pour optimiser m.

Importance des variables

Comme pour tout modèle construit par agrégation ou boîte noire, il n'y a pas d'interprétation directe. Néanmoins des informations pertinentes sont obtenues par le calcul et la représentation graphique d'indices proportionnels à l'*importance* de chaque variable dans le modèle agrégé et donc de sa participation à la régression ou à la discrimination. C'est évidemment d'autant plus utile que les variables sont très nombreuses. Deux critères sont ainsi proposés pour évaluer l'importance de la $j$ème variable.

- Le premier (*Mean Decrease Accuracy*) repose sur une permutation aléatoire des valeurs de cette variable. Plus la qualité, estimée par une l'erreur *out-of-bag*, de la prévision est dégradée par la permutation des valeurs de cette variable, plus celle-ci est importante. Une fois le b-ème arbre construit, l'échantillon *out-of-bag* est prédit par cet arbre et l'erreur estimée enregistrée. Les valeurs de la j-ème variable sont aléatoirement permutées dans l'échantillon *out-of-bag* et l'erreur à nouveau calculée. La décroissance de la qualité de prévision est moyennée sur tous les arbres et utilisée pour évaluer l'importance de la variable j dans la forêt. Il s'agit donc d'une mesure globale mais indirecte de l'influence d'une variable sur la qualité des prévisions.

- Le deuxième (*Mean Decrease Gini*) est local, basé sur la décroissance de l'hétérogénéité définie à partir du critère de Gini ou de celui d'entropie. L'importance d'une variable est alors une somme pondérée des décroissances d'hétérogénéité induites lorsqu'elle est utilisée pour définir la division associée à un nœud.

Implémentations

- L'implémentation la plus utilisée est celle de la librairie `randomForest` de R qui ne fait qu'interfacer le programme original développé en Fortran77 par Léo Breiman et Adele Cutler qui maintient le site dédié à cet algorithme.
- Une alternative en R, plus efficace en temps de calcul surtout avec un volume important de données, consiste à utiliser la librairie `ranger`.
- Le site du logiciel Weka développé à l'université Waikato de Nouvelle Zélande propose une version en Java.
- Une version très efficace et proche de l'algorithme original est disponible dans la librairie `Scikit-learn` de Python.
- Une autre version adaptée aux données massives est proposée dans la librairie `MLlib` de Spark, technologie développée pour interfacer différentes architectures matérielles/logicielles avec des systèmes de gestion de fichiers de données distribuées (Hadoop). En plus du nombre d'arbres, de la profondeur maximum des arbres et du nombre de variables tirées au hasard à chaque recherche de division optimale pour construire un nœud, cette implémentation ajoute "innocemment" deux paramètres : `subsamplingRate` et `maxBins` nantis d'une valeur par défaut. Ces paramètres jouent un rôle important, certes pour réduire drastiquement le temps de calcul, mais, en contre partie, pour restreindre la précision de l'estimation. Ils règlent l'équilibre entre temps de calcul et précision de l'estimation comme le ferait un échantillonnage des données.

  `subsamplingRate = 1.0` sous échantillonne comme son nom l'indique avant la construction de chaque arbre. Avec la valeur par défaut, c'est la version classique des forêts aléatoires avec B Bootstrap mais si ce taux est faible, ce sont des échantillons de taille réduites mais plus distincts ou indépendants pour chaque arbre qui sont tirés. La variance est d'autant réduite (arbre plus indépendants) mais le biais augmente car chaque arbre est moins bien estimé sur un petit lot.

  `maxBins=32` est le nombre maximum de modalités qui sont considérées pour une variable qualitative ou encore le nombre de valeurs possibles pour une variable quantitative. Seules les modalités les plus fréquentes d'une variable qualitative sont prises en compte, les autres sont automatiquement regroupées dans une modalité `autre`. Comme précédemment, le temps de recherche d'une meilleure division est évidemment largement influencé par le nombre de modalités ou encore le nombre de valeurs possibles d'une variable quantitative en opérant des découpages en classe. Réduire le nombre de valeurs possibles est finalement une autre façon de réduire la taille de l'échantillon mais, il serait sans doute opportun de mieux contrôler ces paramètres notamment en guidant les regroupements des modalités pour éviter des contre sens. Ne vaut-il pas mieux sous-échantillonner préalablement les données plutôt que de restreindre brutalement le nombre de valeurs possibles ?

Autres utilisations

Devenu le *couteau suisse* de l'apprentissage, les forêts aléatoires sont utilisées à différentes fins (consulter le site dédié) :

- Similarité ou proximité entre observations. Après la construction de chaque arbre, incrémenter la similarité ou proximité de deux observations qui se trouvent dans la même feuille. Sommer sur la forêt, normaliser par le nombre d'arbres. Un positionnement multidimensionnel peut représenter ces similarités ou la matrice des dissimilarités qui en découle.
- Détection d'observations atypiques multidimensionnelles (*outliers*) ou de "nouveautés" (*novelties*) pour signifier qu'une observation n'appartient pas aux classes connues. Un critère d"anormalité" par rapport à une classe est basé sur la notion précédente de proximités (faible) d'une observation aux autres observations de sa classe.
- Classification non supervisée. Si aucune variables Y n'est à modéliser, l'idée est de se ramener au cas précédant en simulant des observations constituant une deuxième classe synthétique. à partir de celles connues (première classe). Pour ce faire, chaque colonne (variable) est aléatoi-

rement permutée détruisant ainsi la structure de corrélation entre les variables. Une forêt est estimée pour modéliser la variable ainsi créée puis les mêmes approches : matrice de dissimilarités, classification non supervisée à partir de cette matrice, positionnement multidimensionnel, détection d'observations atypiques, sont développées.
- Imputation de données manquantes.
- Modèles de durée de vie (*survival forest*).

3 Famille de modèles adaptatifs

3.1 Principes du *Boosting*

Le *boosting* diffère des approches précédentes par ses origines et ses principes. L'idée initiale, en apprentissage machine, était d'améliorer les compétences d'un *faible classifieur* c'est-à-dire celle d'un modèle de discrimination dont la probabilité de succès sur la prévision d'une variable qualitative est légèrement supérieure à celle d'un choix aléatoire. L'idée originale de Schapire de 1990 a été affinée par Freund et Schapire (1996)[8] qui ont décrit l'algorithme original *adaBoost* (*Adaptative boosting*) pour la prévision d'une variable binaire. De nombreuses études ont ensuite été publiées pour adapter cet algorithme à d'autres situations : k classes, régression, paramètre de *shrinkage* et rendre compte de ses performances sur différents jeux de données. Ces tests ont montré le réel intérêt pratique de ce type d'algorithme pour réduire sensiblement la variance (comme le *bagging*) mais aussi le biais de prévision comparativement à d'autres approches. En effet, comme les arbres sont identiquement distribués par *bagging*, l'espérance de B arbres est la même que l'espérance d'un arbre. Cela signifie que le biais d'arbres agrégés par *bagging* est le même que celui d'un seul arbre. Ce n'est plus le cas avec le *boosting*.

Cet algorithme fut considéré comme la meilleure méthode *off-the-shelf* c'est-à-dire ne nécessitant pas un long prétraitement des données ni un réglage fin de paramètres lors de la procédure d'apprentissage. Néanmoins, l'évolution vers de nouvelles versions (*extrem gradient boosting*) plus performantes en terme de prévision a conduit à multiplier le nombre de paramètres à régler et optimiser.

Le *boosting* adopte le même principe général que le *bagging* : construction d'une famille de modèles qui sont ensuite agrégés par une moyenne pondéré des estimations ou un vote. Il diffère nettement sur la façon de construire la famille qui est dans ce cas récurrente : chaque modèle est une version *adaptative* du précédent en donnant plus de poids, lors de l'estimation suivante, aux observations mal ajustées ou mal prédites. Intuitivement, cet algorithme concentre donc ses efforts sur les observations les plus difficiles à ajuster tandis que l'agrégation de l'ensemble des modèles réduit le risque de sur-ajustement.

Les algorithmes de *boosting* proposés diffèrent par différentes caractéristiques :
- la façon de pondérer c'est-à-dire de renforcer l'importance des observations mal estimées lors de l'itération précédente,
- leur objectif selon le type de la variable à prédire Y : binaire, qualitative à k classes, réelles ;
- la fonction perte, qui peut être choisie plus ou moins robuste aux valeurs atypiques, pour mesurer l'erreur d'ajustement ;
- la façon d'agréger, ou plutôt pondérer, les modèles de base successifs.

La littérature sur le sujet présente donc de très nombreuses versions de cet algorithme dont le dernier avatar, proposé dans une librairie `XGBoost` (*extrem gradient boosting*) connaît beaucoup de succès dans les concours de prévision *Kaggle*.

Cette section propose une présentation historique, car pédagogique, d'`adaBoost` à `XGBoost`.

3.2 Algorithme de base

Décrivons la version originale du *boosting* pour un problème de discrimination élémentaire à deux classes en notant δ la fonction de discrimination à valeurs dans $\{-1, 1\}$. Dans cette version, le modèle de base retourne l'identité d'une classe, il est encore nommé `adaBoost` discret. Il est facile de l'adapter à des modèles retournant une valeur réelle comme une probabilité d'appartenance à une classe.

Les poids de chaque observations sont initialisés à $1/n$ pour l'estimation du premier modèle puis évoluent à chaque itération donc pour chaque nouvelle estimation. L'importance d'une observation w_i est inchangée si elle est bien classée, elle croît sinon proportionnellement au défaut d'ajustement du modèle. L'agrégation finale des prévisions : $\sum_{m=1}^{M} c_m \delta_m(\boldsymbol{x}_0)$ est une combinaison pondérée par les qualités d'ajustement de chaque modèle. Sa valeur

Algorithm 3 `adaBoost` ou (*adaptive boosting*)

Soit $\mathbf{x}_0$ à prévoir et
$\mathbf{z} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ un échantillon
`Initialiser` les poids $\boldsymbol{w} = \{w_i = 1/n\ ;\ i = 1, \ldots, n\}$.
for $m = 1$ à M **do**
Estimer δ_m sur l'échantillon pondéré par $\boldsymbol{w}$.
Calculer le taux d'erreur apparent :

$$\widehat{\mathcal{E}}_p = \frac{\sum_{i=1}^n w_i \mathbf{1}\{\delta_m(\boldsymbol{x}_i) \neq y_i\}}{\sum_{i=1}^n w_i}.$$

Calculer les logit : $c_m = \log((1 - \widehat{\mathcal{E}}_p)/\widehat{\mathcal{E}}_p)$.
Calculer les nouvelles pondérations :

$$w_i \leftarrow w_i . \exp\left[c_m \mathbf{1}\{\delta_m(\boldsymbol{x}_i) \neq y_i\}\right]; i = 1, \ldots, n.$$

end for
Résultat du vote : $\widehat{f}_M(\boldsymbol{x}_0) = \text{signe}\left[\sum_{m=1}^M c_m \delta_m(\boldsymbol{x}_0)\right]$.

absolue appelée *marge* est proportionnelle à la confiance que l'on peut attribuer à son signe qui fournit le résultat de la prévision. Attention, un contrôle doit être ajouté en pratique pour bien vérifier que le classifieur de base est faible mais pas mauvais à savoir que c_m garde bien des valeurs positives ; que le taux d'erreur apparent ne soit pas supérieur à 50%.

Ce type d'algorithme est utilisé avec un arbre (CART) comme modèle de base. De nombreuses applications montrent que si le classifieur faible est un arbre trivial à deux feuilles (*stump*), `adaBoost` fait mieux qu'un arbre sophistiqué pour un volume de calcul comparable : autant de feuilles dans l'arbre que d'itérations dans `adaBoost`. Hastie et col. (2001)[11] discutent la meilleure stratégie d'élagage applicable à chaque modèle de base. Ils le comparent avec le niveau d'interaction requis dans un modèle d'analyse de variance. Le cas $q = 2$ correspondant à la seule prise en compte des effets principaux. Empiriquement ils recommandent une valeur comprise entre 4 et 8.

De nombreuses adaptations ont été proposées à partir de l'algorithme initial. Elles font intervenir différentes fonctions pertes offrant des propriétés de robustesse ou adaptées à une variable cible Y quantitative ou qualitative à plusieurs classes : `adaBoost` M1, M2, MH ou encore MR. Schapire (2002)[12] liste une bibliographie détaillée.

3.3 Version aléatoire

À la suite de Freund et Schapire (1996)[8], Breiman (1998)[3] développe aussi, sous le nom d'*Arcing* (*adaptively resample and combine*), une version aléatoire, et en pratique très proche, du *boosting*. Elle s'adapte à des classifieurs pour lesquels il est difficile voire impossible d'intégrer une pondération des observations dans l'estimation. Ainsi plutôt que de jouer sur les pondérations, à chaque itération, un nouvel échantillon est tiré avec remise, comme pour le bootstrap, mais selon des probabilités inversement proportionnelles à la qualité d'ajustement de l'itération précédente. La présence des observations difficiles à ajuster est ainsi renforcée pour que le modèle y consacre plus d'attention. L'algorithme *adaBoost* précédent est facile à adapter en ce sens en regardant celui développé ci-dessous pour la régression et qui adopte ce point de vue.

3.4 Pour la régression

Différentes adaptations du *boosting* ont été proposées pour le cas de la régression, c'est-à-dire lorsque la variable à prédire est quantitative. Voici l'algorithme de Drucker (1997) dans la présentation de Gey et Poggi (2002)[10] qui en étudient les performances empiriques en relation avec CART. Freund et Schapire (1996) ont proposé *adaBoost.R* avec le même objectif tandis que le point de vue de Friedman (2002)[9] est décrit dans la section suivante par l'algorithme de *gradient boosting machine*.

Algorithm 4 Boosting pour la régression

Soit $\mathbf{x}_0$ à prévoir et
$\mathbf{z} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ un échantillon
Initialiser $\boldsymbol{p}$ par la distribution uniforme $\mathbf{p} = \{p_i = 1/n \ ; \ i = 1, \ldots, n\}$.
for $m = 1$ à M **do**
Tirer avec remise dans $\mathbf{z}$ un échantillon $\mathbf{z}^*_m$ suivant $\mathbf{p}$.
Estimer $\widehat{f}_m$ sur l'échantillon $\mathbf{z}^*_m$.
Calculer à partir de l'échantillon initial $\mathbf{z}$:

$$\begin{aligned} l_m(i) &= l\left(y_i, \widehat{f}_m(\mathbf{x}_i)\right) \quad i = 1, \ldots, n; \quad (l : \text{fonction perte}) \\ \widehat{\mathcal{E}_m} &= \sum_{i=1}^{n} p_i l_m(i); \\ w_i &= g(l_m(i)) p_i. \quad (g \text{ continue non décroissante}) \end{aligned}$$

Calculer les nouvelles probabilités : $p_i \leftarrow \frac{w_i}{\sum_{i=1}^n w_i}$.
end for
Calculer $\widehat{f}(\mathbf{x}_0)$ moyenne ou médiane des prévisions $\widehat{f}_m(\mathbf{x}_0)$ pondérées par des coefficients $\log(\frac{1}{\beta_m})$.

Précisions :

- Dans cet algorithme la fonction perte l peut être exponentielle, quadratique ou, plus robuste, la valeur absolue. Le choix usuel de la fonction quadratique est retenu par Gey et Poggi (2002)[10].
- Notons $L_m = \sup_{i=1,\ldots,n} l_m(i)$ le maximum de l'erreur observée par le modèle $\widehat{f}_m$ sur l'échantillon initial. La fonction g est définie par :

$$g(l_m(i)) = \beta_m^{1-l_m(i)/L_m} \quad (1)$$

$$\text{avec} \quad \beta_m = \frac{\widehat{\mathcal{E}_m}}{L_m - \widehat{\mathcal{E}_m}}. \quad (2)$$

- Comme pour adaBoost discret, une condition supplémentaire est ajoutée à l'algorithme. Il est arrêté ou réinitialisé à des poids uniformes si l'erreur se dégrade trop : si $\widehat{\mathcal{E}_m} < 0.5 L_m$.

L'algorithme génère M prédicteurs construits sur des échantillons bootstrap $\mathbf{z}^*_m$ dont le tirage dépend de probabilités $\mathbf{p}$ mises à jour à chaque itération. Cette mise à jour est fonction d'un paramètre β_m qui est un indicateur de la performance, sur l'échantillon $\mathbf{z}$, du m-ième prédicteur estimé sur l'échantillon $\mathbf{z}^*_m$. La mise à jour des probabilités dépend donc à la fois de cet indicateur global β_m et de la qualité relative $l_m(i)/L_m$ de l'estimation du i-ème individu. L'estimation finale est enfin obtenue à la suite d'une moyenne ou médiane des prévisions pondérées par la qualité respective de chacune de ces prévisions. Gey et Poggi (2002)[10] conseille la médiane afin de s'affranchir de l'influence de prédicteurs très atypiques.

3.5 Modèle additif pas à pas

Le bon comportement du *boosting* par rapport à d'autres techniques de discrimination est difficile à expliquer ou justifier par des arguments théoriques. À la suite d'une proposition de Breiman en 1999 (rapport technique) de considérer le *boosting* comme un algorithme global d'optimisation, Hastie et col. (2001)[11] présentent le *boosting* dans le cas binaire sous la forme d'une approximation de la fonction f par un modèle additif construit pas à pas :

$$\widehat{f}(\boldsymbol{x}) = \sum_{m=1}^{M} c_m \delta(\boldsymbol{x}; \gamma_m)$$

est cette combinaison où c_m est un paramètre, δ le classifieur (faible) de base fonction de $\boldsymbol{x}$ et dépendant d'un paramètre γ_m. Si l est une fonction perte, il s'agit, à chaque étape, de résoudre :

$$(c_m, \gamma_m) = \arg\min_{(c,\gamma)} \sum_{i=1}^{n} l(y_i, \widehat{f}_{m-1}(\boldsymbol{x}_i) + c\delta(\boldsymbol{x}_i; \gamma));$$

$\widehat{f}_m(\boldsymbol{x}) = \widehat{f}_{m-1}(\boldsymbol{x}) + c_m\delta(\boldsymbol{x};\gamma_m)$ est alors une amélioration de l'ajustement précédent.

Dans le cas d'`adaBoost` pour l'ajustement d'une fonction binaire, la fonction perte utilisée est $l(y, f(\boldsymbol{x})) = \exp[-yf(\boldsymbol{x})]$. il s'agit donc de résoudre :

$$\begin{aligned}(c_m,\gamma_m) &= \arg\min_{(c,\gamma)}\sum_{i=1}^{n}\exp\left[-y_i(\widehat{f}_{m-1}(\boldsymbol{x}_i)+c\delta(\boldsymbol{x}_i;\gamma))\right];\\ &= \arg\min_{(c,\gamma)}\sum_{i=1}^{n} w_i^m\exp\left[-cy_i\delta(\boldsymbol{x}_i;\gamma)\right]\\ \text{avec}\quad w_i^m &= \exp[-y_i\widehat{f}_{m-1}(\boldsymbol{x}_i)];\end{aligned}$$

w_i^m ne dépendant ni de c ni de γ, il joue le rôle d'un poids fonction de la qualité de l'ajustement précédent. Quelques développements complémentaires montrent que la solution du problème de minimisation est obtenue en deux étapes : recherche du classifieur optimal puis optimisation du paramètre c_m.

$$\begin{aligned}\gamma_m &= \arg\min_{\gamma}\sum_{i=1}^{n}\mathbf{1}\{y_i\neq\delta(\boldsymbol{x}_i;\gamma)\},\\ c_m &= \frac{1}{2}\log\frac{1-\widehat{\mathcal{E}}_p}{\mathcal{E}_p}\end{aligned}$$

avec $\widehat{\mathcal{E}}_p$ erreur apparente de prévision tandis que les w_i sont mis à jour avec :

$$w_i^{(m)} = w_i^{(m-1)}\exp[-c_m].$$

On montre ainsi qu'`adaBoost` approche f pas à pas par un modèle additif en utilisant une fonction perte exponentielle tandis que d'autres types de *boosting* sont définis sur la base d'une autre fonction perte :

`adaBoost` $l(y, f(\boldsymbol{x})) = \exp[-yf(\boldsymbol{x})]$,

`LogitBoost` $l(y, f(\boldsymbol{x})) = \log_2(1+\exp[-2yf(\boldsymbol{x})]$,

L^2`Boost` $l(y, f(\boldsymbol{x})) = (y-f(\boldsymbol{x}))^2/2$.

D'autres fonctions pertes sont envisageables pour, en particulier, un algorithme plus robuste face à un échantillon d'apprentissage présentant des erreurs de classement dans le cas de la discrimination ou encore des valeurs atypiques (*outliers*) dans le cas de la régression. Hastie et col. (2001)[11] comparent les intérêts respectifs de plusieurs fonctions pertes. Celles jugées robustes (entropie en discrimination, valeur absolue en régression) conduisent à des algorithmes plus compliqués à mettre en œuvre.

3.6 *Boosting* et gradient adaptatif

Préambule

Dans le même esprit d'approximation adaptative, Friedman (2002)[9] a proposé sous l'acronyme MART (*multiple additive regression trees*) puis sous celui de GBM (*gradient boosting models*) une famille d'algorithmes basés sur une fonction perte supposée convexe et différentiable notée l. Le principe de base est le même que pour `adaBoost`, construire une séquence de modèles de sorte que chaque étape, chaque modèle ajouté à la combinaison, apparaisse comme un pas vers une meilleure solution. La principale innovation est que ce pas est franchi dans la direction du *gradient de la fonction perte*, afin d'améliorer les propriétés de convergence. Une deuxième idée consiste à approcher le gradient par un *arbre de régression* afin d'éviter un sur-apprentissage.

Le modèle adaptatif pas-à-pas précédent :

$$\widehat{f}_m(\boldsymbol{x}) = \widehat{f}_{m-1}(\boldsymbol{x}) + c_m\delta(\boldsymbol{x};\gamma_m)$$

est transformé en une descente de gradient :

$$\widehat{f}_m = \widehat{f}_{m-1} - \gamma_m\sum_{i=1}^{n}\nabla_{f_{m-1}}l(y_i, f_{m-1}(x_i)).$$

Plutôt que de chercher un meilleur classifieur comme avec adaBoost, le problème se simplifie en la recherche d'un meilleur pas de descente γ :

$$min_\gamma\sum_{i=1}^{n}\left[l\left(y_i, f_{m-1}(x_i) - \gamma\frac{\partial l(y_i, f_{m-1}(x_i))}{\partial f_{m-1}(x_i)}\right)\right].$$

Algorithme

L'algorithme ci-dessous décrit le cas de la régression, il peut être adapté à celui de la classification.

Algorithm 5 *Gradient Tree Boosting* pour la régression

Soit $\mathbf{x}_0$ à prévoir
Initialiser $\widehat{f}_0 = \arg\min_\gamma \sum_{i=1}^n l(y_i, \gamma)$
for $m = 1$ à M **do**
  Calculer $r_{mi} = -\left[\frac{\partial l(y_i, f(\boldsymbol{x}_i))}{\partial f(\boldsymbol{x}_i)}\right]_{f=f_{m-1}}$; $i = 1, \ldots, m$
  Ajuster un arbre de régression δ_m aux couples $(\boldsymbol{x}_i, r_{mi})_{i=1,\ldots,n}$
  Calculer γ_m en résolvant : $min_\gamma \sum_{i=1}^n l(y_i, f_{m-1}(\boldsymbol{x}_i) + \gamma\delta_m(\boldsymbol{x}_i))$.
  Mise à jour : $\widehat{f}_m(\boldsymbol{x}) = \widehat{f}_{m-1}(\boldsymbol{x}) + \gamma_m\delta_m(\boldsymbol{x})$
end for
Résultat : $\widehat{f}_M(\boldsymbol{x}_0)$.

L'algorithme est initialisé par un terme constant c'est-à-dire encore un arbre à une feuille. Les expressions du gradient reviennent simplement à calculer les termes r_{mj}, pour chaque nœud du modèle et chaque observation de ce nœud, à l'étape précédente. Le pas de descente γ est optimisé pour obtenir chaque mise à jour du modèle. Un algorithme de discrimination est similaire calculant autant de probabilités que de classes à prévoir.

Sur-ajustement et rétrécissement

Friedman (2002)[9] a également proposé une version de *stochastic gradient boosting* incluant un sous-échantillonnage aléatoire à chaque étape afin de construire comme en *bagging* une séquence de prédicteurs plus indépendants. Le taux de sous-échantillonnage est un autre paramètre à optimiser, il est présent dans la version de GBM implémentée dans `Scikit-learn`.

Une autre proposition consiste à ajouter un coefficient η de rétrécissement (*shrinkage*). Compris entre 0 et 1, celui-ci pénalise l'ajout d'un nouveau modèle dans l'agrégation et ralentit la convergence.

$$\widehat{f}_m(\boldsymbol{x}) = \widehat{f}_{m-1}(\boldsymbol{x}) + \eta\ \gamma_m\delta_m(\boldsymbol{x}).$$

Si sa valeur est petite ($< 0,1$) cela conduit à accroître le nombre d'arbres mais entraîne généralement une amélioration de la qualité de prévision. Le *boosting* est un algorithme qui peut effectivement converger exactement, donc éventuellement vers une situation de sur-apprentissage. En pratique, cette convergence peut être rendue suffisamment lente pour être mieux contrôlée.

Dans cette dernière version de l'algorithme, il est conseillé de *contrôler* le nombre d'itérations par un échantillon de validation ou par validation croisée.

En résumé, ce sont trois paramètres qu'il est d'usage de contrôler ou optimiser lors de la mise en œuvre de l'algorithme de *gradient boosting* implémenté en R (`gbm`) ou en pyhton (`scikit-learn`). La nomenclature est celle de Python (`scikit-learn`) mais les mêmes paramètres se retrouvent en R.

- la profondeur maximale des arbres : `max_depth`,
- le coefficient de rétrécissement : `learning_rate` ou *shrinkage*,
- le nombre d'arbres ou d'itérations `n_estimators`.

Auxquels s'ajoute le taux de sous-échantillonnage si le choix est fait du *stochastic gradient boosting*.

Outre le choix de la fonction perte les implémentations en R ou dans `scikit-learn` proposent plusieurs autres paramètres :

- `max_features` : nombre de variables tirés à la recherche de chaque division comme dans *random forest*,
- `min_sampes_split` : nombre minimal d'observations nécessaires pour diviser un nœud,
- `min_sampes_leaf` ou `min_weight_fraction_leaf` : nombre minimal d'observations dan schaque feuille pour conserver une division,
- `max_leaf_node` : contrôle comme `max_depth` la complexité d'un arbre.
- `subsample` : part d'échantillon tiré aléatoirement à chaque étape.
- ... consulter la documentation en ligne.

Ces paramètres jouant des rôles redondants sur le contrôle du sur-apprentissage, il n'est pas nécessaire de tous les optimiser. Certains sites proposent des stratégies testées sur les concours *Kaggle* pour conduire l'optimisation.

Comme pour les forêts aléatoire, des critères d'*importance des variables* sont ajoutés au *boosting* afin d'apporter quelques pistes de compréhension du modèle.

3.7 *Extrem Gradient Boosting*

Prélude

Plus récemment, Chen et Guestrin (2016)[6] ont proposé un dernier avatar du *boosting* avec l'*extrem gradient boosting*. La complexification est très sensible notamment avec le nombre de paramètres qu'il est nécessaire de prendre en compte dans l'optimisation de l'algorithme. Les coûts de calcul deviendraient assez rédhibitoires, aussi la librairie associée (`XGBoost`), disponible en R, Python, Julia, Scala et dans des environnements distribués (Spark) offre une parallélisation efficace des calculs avec notamment la possibilité d'accéder à la carte graphique (GPU) de l'ordinateur et propose une version approchée lorsque les données massives sont distribuées.

Mais, ce qui encourage réellement à s'intéresser à cette version du *boosting*, c'est son utilisation assez systématique dans les solutions gagnantes des concours *Kaggle* de prévision. Très schématiquement, si les données du concours sont des images, ce sont des algorithmes d'apprentissage profond qui l'emportent, sinon ce sont des combinaisons sophistiqués (usines à gaz) d'algorithmes incluant généralement `XGBoost` (cf. figure 1).

Il n'est pas dit que ces solutions gagnantes de type "usines à gaz" soient celles à rendre opérationnelles ou à industrialiser mais il semble nécessaire d'intégrer `XGBoost` dans la boîte du *data scientist* des outils à comparer. Même sans disposer des moyens de calcul adaptés à une optimisation de tous les paramètres, la mise en œuvre n'utilisant que les valeurs par défaut et une optimisation sommaire avec `caret`, qui offre des possibilités de parallélisation, même sous Windows, peut améliorer les solutions.

L'algorithme `XGBoost` inclut plusieurs fonctionnalités chacune associée avec un ou des paramètres supplémentaires à optimiser, en plus de ceux déjà décrit avec une nomenclature différente (`learning_rate`, `subsample`, `max_features`...) pour le *gradient boosting*. `XGBoost` est schématiquement décrit dans le cas de la régression. Se reporter à l'article original de Chen et Guestrin (2016)[6] pour des précisions, notamment sur les astuces d'implémentation et sur la dernière documentation en ligne pour la liste complète des paramètres. Un site propose des stratégies d'optimisation étudiées empiriquement sur les concours *Kaggle*.

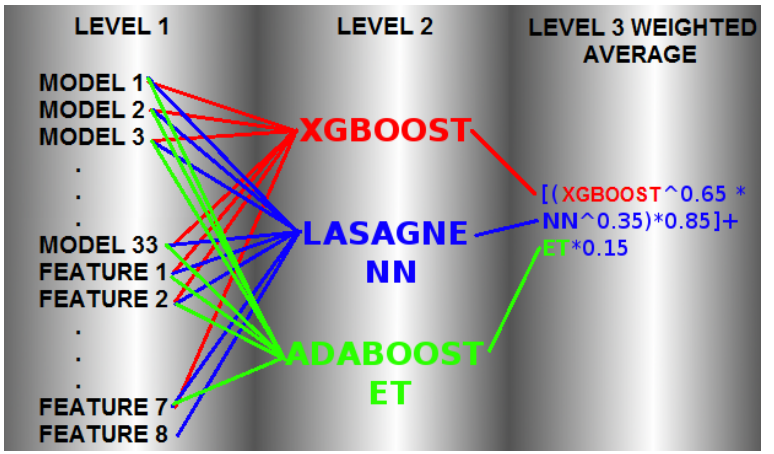

FIGURE 1 – *Solution gagnante d'un concours kaggle : Identify people who have a high degree of Psychopathy based on Twitter usage. Combinaison pondérée des combinaisons (boosting, réseaux de neurones) de trente trois modélisations (random forest, boosting, k plus proches voisins...) et 8 nouvelles variables (*features*) ad'hoc.*

Terme de régularisation

Une nouvelle fonction objectif $\mathcal{L}$ est considérée en complétant la fonction perte convexe différentiable l par un terme de régularisation :

$$\mathcal{L}(f) = \sum_{i=1}^{n} l(\widehat{[}y)_i, y_i) + \sum_{m=1}^{M} \Omega(\delta_m)$$

avec

$$\Omega(\delta) = \alpha|\delta| + \frac{1}{2}\beta||\boldsymbol{w}||^2,$$

où $|\delta|$ est le nombre de feuilles de l'arbre de régression δ et $\boldsymbol{w}$ le vecteur des valeurs attribuées à chacune de ses feuilles.

Cette pénalisation ou régularisation limite l'ajustement de l'arbre ajouté à chaque étape et contribue à éviter un sur-ajustement. Notamment lorsque des observations affectées d'erreurs importantes sont présentes, augmenter le nombre d'itérations peut provoquer une dégradation de la performance globale plutôt qu'une amélioration. Le terme Ω s'interprète comme une combinaison de régularisation *ridge* de coefficient β et de pénalisation Lasso de coefficient α.

Approximation du gradient

Une *astuce* consiste à approcher le gradient par un développement de Taylor au second ordre. Il suffit alors de sommer, pour chaque feuille, les valeurs des dérivées première et seconde de la fonction perte pour évaluer la fonction objectif pénalisées et maximiser sa décroissance lors de la recherche des divisions. La simplification introduite permet une parallélisation efficace de la construction des arbres.

Recherche des divisions

Lorsque les données sont volumineuses, éventuellement distribuées, l'algorithme original *glouton* de recherche d'une meilleure division nécessite trop de temps de calcul ou peut saturer l'espace mémoire. Une version approchée consiste à découper les variables quantitatives en classes en utilisant les quantiles de la distribution comme bornes. C'est un procédé similaire à celui utilisé dans l'implémentation des forêts aléatoires de la librairie `MLlib` de *Spark*. Dans le cas du *gradient boosting* c'est plus compliqué, Chen et Guestrin (2016)[6] détaillent un algorithme pour des quantiles pondérés.

Données manquantes

La gestion de données manquantes ou de zéros instrumentaux anormaux est prise en compte de façon originale dans l'implémentation de `XGBoost`. Celui-ci propose à chaque division une direction par défaut si une donnée est manquante. Pour palier cette absence, le gradient est calculé sur les seules valeurs présentes.

Autres paramètres

Enfin d'autres spécificités de l'implémentation améliorent les performances de l'algorithme : stockage mémoire en block compressé pour paralléliser les opérations très nombreuses de tris, tampons mémoire pour réduire les accès disque, blocs de données compressés distribués (*sharding*) sur plusieurs disques.

En plus des paramètres du *gradient boosting* dont certains change de nom par rapport à l'implémentation de `Scikit-learn`, d'autres paramètres sont à optimiser avec quelques précisions :

- `alpha` : coefficient de pénalisation Lasso (l_1),
- `lambda` : coefficient de régularisation de type *ridge* (l_2),
- `tree_method` : sélection, automatique par défaut, de l'algorithme exacte glouton ou celui approché par découpage des variables quantitatives guidé par le paramètre
- `sketch_eps` : qui contrôle le nombre de classes.
- `scale_pos_weight` : utile pour des classes déséquilibrées.
- Autres paramètres techniques liés à l'optimisation du temps de calcul.

Interprétation

L'interprétabilité des arbres de décision sont une des raisons de leur succès. Leur lecture ne nécessite pas de compétences particulières en statistique. Cette propriété est évidemment perdue par l'agrégation d'arbres ou de tout autre modèle. Néanmoins, surtout si le nombre de variables est très grand, il est important d'avoir une indication de l'importance relative des variables entrant dans la modélisation. Des critères d'importance des variables sont disponibles comme dans le cas des forêts aléatoires.

4 Super apprenti

D'autres stratégies ont été proposées dans l'optique d'une prévision "brute", sans sélection de variables ou objectif d'interprétation du modèle. Elles procèdent en deux principales étapes :

- la première consiste à estimer un ensemble de modèles variés appartenant à celles des méthodes précédentes, de la régression au *boosting* en passant par les réseaux de neurones.
- la deuxième construit une combinaison linéaire convexe (*super learner* Van der Laan et al. (2007)[14]) ou une régression locale, à partir des modèles précédents (COBRA de Biau et al. (2013)[1]) ou encore une architecture sophistiquée et très empirique utilisée dans les concours de prévision (cf. figure 1).

La dernière solution de "type usine à gaz" ne conduit généralement pas à des approches industrialisables et présentent des risques de sur-apprentissage, même sur l'échantillon test masqué lorsque des centaines voire milliers de concurrents s'affrontent avec des solutions ne différant qu'à la 2ème ou 3ème décimale du critère. Une part de hasard et donc d'artefact ne peut être exclue pour apparaître dans les toutes premières places.

Les autres solutions de combinaisons de modèles présentent des garanties plus théoriques dont celle du *Super learner*. Le principe de l'approche proposée par van der Laan et al. (2007) [14] est simple, il s'agit de calculer une combinaison convexe ou moyenne pondérée de plusieurs prévisions obtenues par plusieurs modèles. Les paramètres de la combinaison sont optimisés en minimisant un critère de validation croisée. La méthode est implémentée dans la librairie `SuperLearner` de R où toutes les combinaisons de méthodes ne sont pas possibles, seule une liste prédéfinie est implémentée à cette date (juin 2014) : `glm, random forest, gbm, mars, svm`. Son emploi est illustré dans le scénario d'analyse de données (QSAR) issues de criblage virtuel de molécules.

5 Exemples

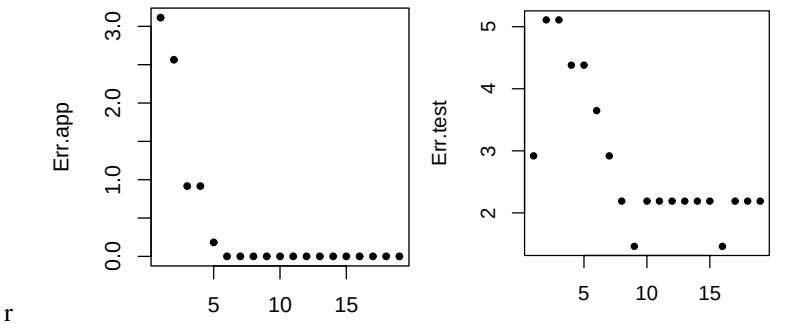

FIGURE 2 – *Cancer : Évolution des taux d'erreur (%) sur les échantillons d'apprentissage et de test en fonction du nombre d'arbres dans le modèle avec adaBoost.*

5.1 Cancer du sein

La prévision de l'échantillon test par ces algorithmes à des taux d'erreurs estimés de 4,4 et 2,2% pour cet exemple et avec les échantillons (apprentissage et test) tirés.

Il est remarquable de noter l'évolution des erreurs d'ajustement et de test (figure 2) en fonction du nombre d'arbres estimés par adaBoost. L'erreur d'apprentissage arrive rapidement à 0 tandis que celle de test continue à décroître avant d'atteindre un seuil. Cet algorithme est donc relativement robuste au sur-apprentissage avant, éventuellement, de se dégrader pour des raisons, sans doute, de précision numérique. Ce comportement à été relevé dans beaucoup d'exemples dans la littérature.

5.2 Concentration d'ozone

Malgré une bonne prévision quantitative, la prévision du dépassement de seuil reste difficile pour l'algorithme des forêts aléatoires. Par une régression ou une discrimination, le taux d'erreur obtenu est le même (12,5%) sur le même échantillon test et d'autres expérimentations sont nécessaires pour départager, ou non, les différentes méthodes. Il semble que, à travers plusieurs exemples, l'amélioration apportée à la prévision par des algorithmes d'agré-

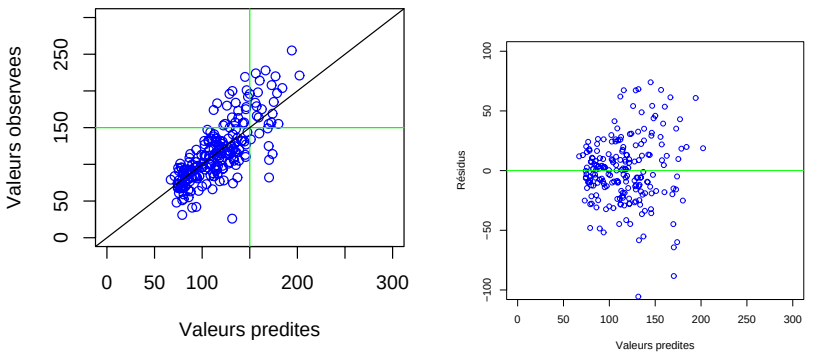

FIGURE 3 – *Ozone : Valeurs observées et résidus de l'échantillon test en fonction des valeurs prédites par une forêt aléatoire*

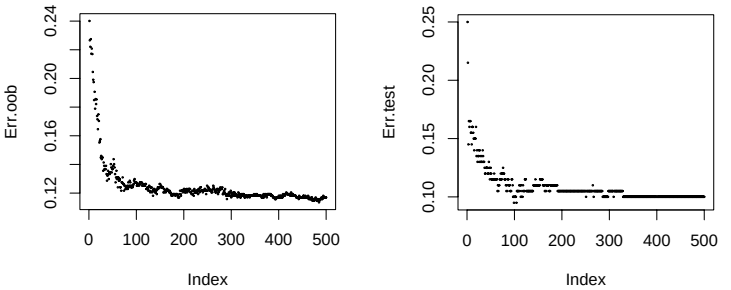

FIGURE 4 – *Banque : Évolution du taux de mal classés estimés "out-of-bag" et sur l'échantillon test en fonction du nombre d'arbres intervenant dans la combinaison de modèles.*

gation de modèles soit nettement plus probante dans des situations difficiles c'est-à-dire avec beaucoup de variables explicatives et des problèmes de multicolinéarité.

Comme les réseaux de neurones, les algorithmes d'agrégation de modèles sont des boîtes noires. Néanmoins dans le cas des forêts, les critères d'importance donnent des indications sur le rôle de celles-ci. Les voici ordonnées par ordre croissant du critère basé sur celui de Gini pour la construction des arbres.

```
jour station     lno    lno2 vmodule s_rmh2o   O3_pr   TEMPE
2.54   13.58   21.78   23.33   24.77   31.19   43.87   67.66
```

Les variables prépondérantes sont celles apparues dans la construction d'un seul arbre.

5.3 Données bancaires

Les arbres, qui acceptent à la fois des variables explicatives qualitatives et quantitatives en optimisant le découpage des variables quantitatives, se prêtent bien au traitement des données bancaires. on a vu qu'un seul arbre donnait des résultats semble-t-il très corrects. Naturellement les forêts constitués d'arbres se trouvent également performantes sur ces données en gagnant en stabilité et sans trop se poser de problème concernant l'optimisation de paramètres. Les TPs décrivent également les résultats proposés par les algorithmes de bagging et de boosting sur les arbres en faisant varier certains paramètres comme le *shrinkage* dans le cas du boosting.

Les graphiques de la figure 4 montrent bien l'insensibilité des forêts au surapprentissage. Les taux d'erreurs estimés, tant par bootstrap (out-of-bag), que sur un échantillon test, se stabilisent au bout de quelques centaines d'itérations. Il est même possible d'introduire dans le modèle toutes les variables quantitatives et qualitatives, avec certaines dupliquées, en laissant l'algorithme faire son choix. Cet algorithme conduit à un taux d'erreur de 10,5% sur l'échantillon test avec la matrice de confusion :

```
     Cnon Coui
Cnon  126   11
Coui   10   53
```

tandis que les coefficients d'importance :

```
QSMOY FACANL  RELAT DMVTPL QCREDL MOYRVL
20.97  26.77  29.98  36.81  40.31  50.01
```

mettent en évidence les variables les plus discriminantes. De son côté, le boosting (sans *shrinkage*) fournit des résultats comparables avec un taux d'erreur de 11%, le *gradient boosting* atteint 9,5% et `XGBoost` 8,8% sans gros efforts d'optimisation.

Références

[1] G. Biau, A. Ficher, B. Guedj et J. D. Malley, *COBRA : A Nonlinear Aggregation Strategy*, Journal of Multivariate Analysis **146** (2016), 18–28.

[2] L. Breiman, *Bagging predictors*, Machine Learning **26** (1996), n° 2, 123–140.

[3] _____, *Arcing classifiers*, Annals of Statistics **26** (1998), 801–849.

[4] _____, *Random forests*, Machine Learning **45** (2001), 5–32.

[5] Rich. Caruana, N. Karampatziakis et A. Yessenalina, *An Empirical Evaluation of Supervised Learning in High Dimensions*, Proceedings of the 25th International Conference on Machine Learning (New York, NY, USA), ICML '08, ACM, 2008, p. 96–103, ISBN 978-1-60558-205-4.

[6] Tianqi Chen et Carlos Guestrin, *XGBoost : A Scalable Tree Boosting System*, Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, 2016, p. 785–794.

[7] Manuel Fernandez-Delgado, Eva Cernadas, Senen Barro et Dinani Amorim, *Do we Need Hundreds of Classiers to Solve Real World Classication Problems ?*, IEEE Trans. Pattern Anal. Mach. Intell. **15** (2014), 3133–31.

[8] Y. Freund et R.E. Schapire, *Experiments with a new boosting algorithm*, Machine Learning : proceedings of the Thirteenth International Conference, Morgan Kaufman, 1996, San Francisco, p. 148–156.

[9] J. H. Friedman, *Stochastic gradient boosting*, Computational Statisrics and Data Analysis **38** (2002), .

[10] S. Gey et J. M. Poggi, *Boosting and instabillity for regression trees*, Rap. tech. 36, Université de Paris Sud, Mathématiques, 2002.

[11] T. Hastie, R. Tibshirani et J Friedman, *The elements of statistical learning : data mining, inference, and prediction*, Springer, 2009, Second edition.

[12] R. Schapire, *The boosting approach to machine learning. An overview*, MSRI workshop on non linear estimation and classification, 2002, p. .

[13] E. Scornet, G. Biau et J. P. Vert, *Consistency of random forests*, The Annals of Statistics (2015), à paraître.

[14] M. J. van der Laan, E. C. Polley et A. E. Hubbard, *Super learner*, Statistical Applications in Genetics and Molecular Biology **6 :1** (2007).