Introduction au boosting

Jean-Marc Lasgouttes, Inria de Paris
jean-marc.lasgouttes@inria.fr

http://mastere-esd.lasgouttes.net/boosting

Partie I. Introduction au cours

Organisation du cours

Matin Cours (3 heures)
- description de AdaBoost
- description de Gradient Boosting et XGBoost
- packages R implémentant les méthodes

Après midi TP (4 heures)
- application des méthodes sur un jeu de données
- rédaction d'un rapport rapide décrivant votre approche et vos résultats.

Références
- Freund, Y. and Schapire, R., A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 1997, 55 (1), 119-139
- J. H. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, *Annals of Statistics*, 2001, 29(5):1189-1232.
- Zhu, J., Zou, H., Rosset, S. and Hastie, T., Multi-class AdaBoost. *Statistics and its Interface* 2009, 2, 349–360.
- Hastie, T., Tibshirani, R., & Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction.* 2nd ed. New York: Springer.

Le problème de classification

Observations On dispose de données $\mathbf{x} \in \mathbb{E}$:
- $\mathbb{E} = \mathbb{R}^p$: p variables quantitatives (poids, taille, âge...)
- $\mathbb{E} = \{1, \dots, n_1\} \times \{1, \dots, n_2\} \times \cdots \times \{1, \dots, n_p\}$: p variables qualitatives (couleur des yeux, sexe, métier,...)
- ou un mélange de tout cela

La classification À chaque variable $\mathbf{x}$, on cherche à associer une variable $y \in \{-1, 1\}$
- « a survécu au naufrage du Titanic »
- « risque de faire un AVC dans l'année qui vient »
- « fraude le fisc »
- ...

Les données On dispose d'un échantillon de
- n observations $(\mathbf{x}_1, \dots, \mathbf{x}_n) \in \mathbb{E}^n$,
- des classifications $(y_1, \dots, y_n) \in \{-1, 1\}^n$

Objectif On cherche une fonction $G : \mathbb{E} \mapsto \{-1, 1\}$, telle que $G(\mathbf{x})$ soit une bonne prédiction du y correspondant

Boosting

Classifieurs faibles On se repose sur un ensemble de classifieurs $h : \mathbb{E} \mapsto \{-1, 1\}$ très simples qui permettent de prédire y juste un peu mieux que le hasard :

$$\epsilon = P(h(\mathbf{x}) \neq y) \leq \frac{1}{2} - \gamma,\ \gamma > 0$$

Exemple de classifieur faible Les plus utilisés sont
- arbres de décision (CART, *Classification And Regression Tree*) de faible profondeur ;
- stumps (souches), c'est-à-dire arbre de profondeur 1, par exemple pour le Titanic

$$\text{age} < 15 \implies \text{survie}, \quad \text{age} \geq 15 \implies \text{décès}.$$

Question Est-on capable de fabriquer séquentiellement un classifieur fort (erreur très petite) à partir d'un grand nombre de classifieurs faibles (erreur un peu plus petite que 0.5) ?

Réponse C'est le boosting !

Les approches ensemblistes

Approche générale on cherche à créer un classifieur fort en combinant plusieurs classifieurs plus simples

Bagging on entraîne les modèles sur des sous-ensemble des données
- approche parallèle
- pas très efficace pour réduire le biais
- évite le sur-ajustement

Random forest bagging + une partie des variables est utilisée pour chaque arbre
- mieux que bagging en tout point de vue

Boosting chaque modèle cherche à corriger les faiblesses du précédent
- approche itérative
- utilise des modèles très simples
- réduit le biais
- risque de sur-ajustement

Lequel choisir ? dépend de si le problème avec les données est plus le biais ($\rightarrow$ boosting) ou le sur-ajustement ($\rightarrow$ random forest).

Le (gradient) boosting est plus sensible aux paramètres.

Partie II. AdaBoost

Qu'est-ce que c'est ?

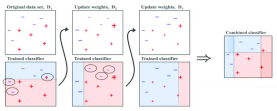
Signification *Ada*ptive *boost*ing

Qui ? Cet algorithme a été introduit en 1996 par Yoav Freund and Rob Shapire (prix Gödel 2003)

Quoi ? C'est le premier algorithme qui montre que les idées du boosting peuvent être implémentées de manière simple et efficaces

Caractéristiques d'AdaBoost
- produit une classification forte à partir de classifications faibles
- fonctionne en donnant plus d'importance aux observations difficiles à prédire
- très peu de paramètres (nombre de pas, complexité des classifieurs faibles)
- évite le sur-ajustement dans certain modèles
- peut aussi être utilisé pour des problèmes de régression

Exemple simple



Voir schéma page suivante
- règles faibles : stump sur l'abscisse ou l'ordonnée
- on augmente le poids des éléments mal classifiés à chaque itération
- Le classifieur final est une combinaison linéaire des classifieurs construits au fur et à mesure.

Algorithme AdaBoost

Entrée Les éléments nécessaires sont
- un échantillon $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$
- un ensemble de règles faibles
- le nombre M d'itérations

Initialisation on se donne des poids $(w_1, \ldots, w_n)$ uniformes

$$w_i \leftarrow \frac{1}{n}, i = 1, \ldots, n,$$

qui vérifient évidemment $w_1 + \cdots + w_n = 1$.

Itération pour $m = 1$ à M

1. ajuster un classifieur faible $g_m(\mathbf{x})$ sur l'échantillon pondéré par les poids w_i
2. calculer le taux d'erreur

$$\epsilon_m \leftarrow \sum_{i=1}^{n} w_i \mathbf{1}_{\{y_i \neq g_m(\mathbf{x}_i)\}}$$

3. calculer le poids de l'itération m :

$$\alpha_m \leftarrow \log \sqrt{(1-\epsilon_m)/\epsilon_m}$$

4. mettre à jour les poids des observations

$$\begin{aligned} w_i &\leftarrow \frac{1}{Z_m} w_i \exp\left[-\alpha_m y_i g_m(\mathbf{x}_i)\right] \\ &= \frac{1}{Z_m} w_i \times \begin{cases} e^{-\alpha_m}, & \text{si } y_i = g_m(\mathbf{x}_i), \\ e^{\alpha_m} & \text{sinon.} \end{cases} \end{aligned}$$

Sortie c'est le signe de la combinaison linéaire

$$\hat{g}_M(\mathbf{x}) = \operatorname{sign} \sum_{m=1}^{M} \alpha_m g_m(\mathbf{x})$$

Remarques

Ajustement L'étape 1 dépend évidemment des règles faibles choisies
- En général on cherche à minimiser ϵ_m
- Si on ne peut pas avoir de poids (arbres CART), on tire n valeurs (avec remise) de l'échantillon, suivant les poids w_i

Constante de normalisation pour que la somme des w_i reste 1. calcul de Z_m :

$$\begin{aligned} Z_m &= \sum_{i=1}^{n} w_i \left[e^{-\alpha_m} \mathbf{1}_{\{y_i = g_m(\mathbf{x}_i)\}} + e^{\alpha_m} \mathbf{1}_{\{y_i \neq g_m(\mathbf{x}_i)\}} \right] \\ &= (1-\epsilon_m)\sqrt{\epsilon_m/(1-\epsilon_m)} + \epsilon_m \sqrt{(1-\epsilon_m)/\epsilon_m} \\ &= 2\sqrt{(1-\epsilon_m)\epsilon_m}. \end{aligned}$$

Règle faible Elles ne doivent pas être trop faibles... On demande $\epsilon_m = 0.5 - \gamma_m$, avec $\gamma_m \geq \gamma$

Erreur empirique d'apprentissage Freund & Shapire ont montré que

$$\begin{aligned} L_n(\hat{g}_M) &= \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{y_i \neq \hat{g}_M(\mathbf{x}_i)} \\ &\leq \exp\left[-2 \sum_{i=1}^{M} \gamma_m^2 \right] \leq \exp(-2M\gamma^2) \end{aligned}$$

Erreur de généralisation

Définition C'est l'erreur moyenne attendue sur un échantillon de test

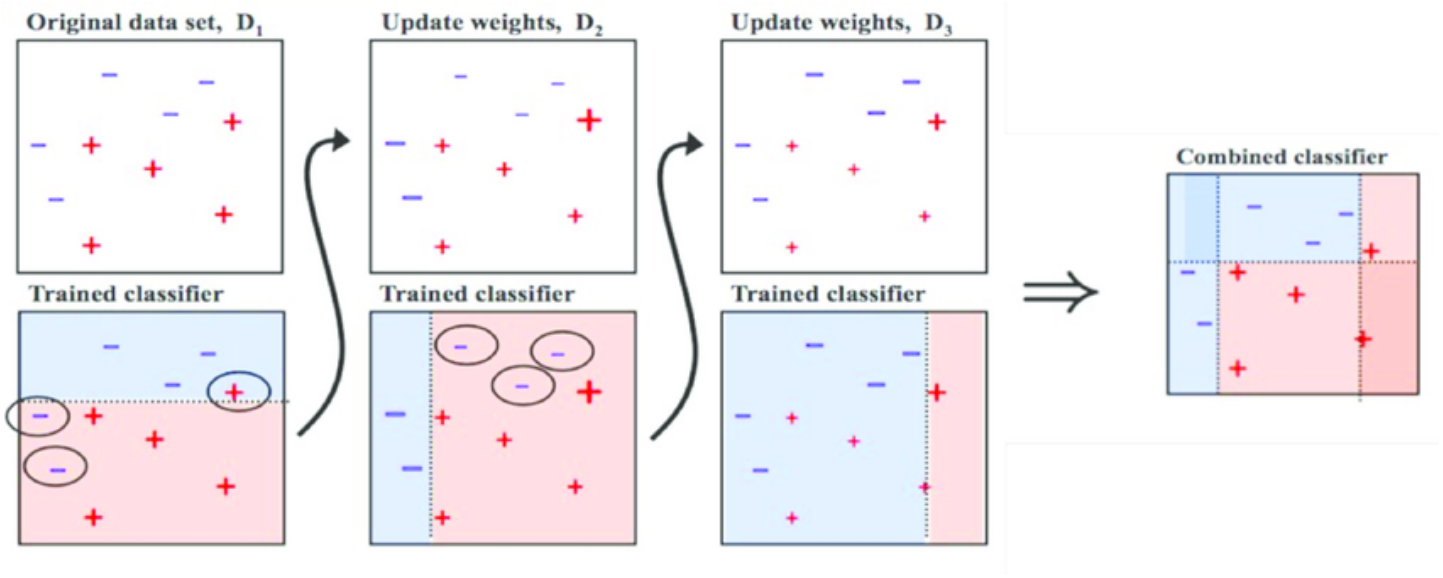
$$L(\hat{g}_M) = \mathbf{P}(Y \neq \hat{g}_M(\mathbf{X}))$$

Borne obtenue par Freund & Shapire

$$L(\hat{g}_M) \leq L_n(\hat{g}_M) + O\left(\sqrt{\frac{MV}{n}} \right),$$

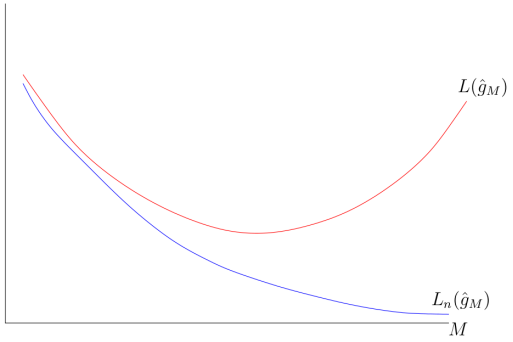
où V est la dimension de Vapnik-Chervonenkis de la famille de classifieurs faibles (3 dans l'exemple simple)

Interprétation Il peut y avoir du sur-ajustement
- si M est trop grand par rapport à n
- d'autant plus que V est grande (elle est grande si les règles peuvent être très complexes)



Problèmes de sur-ajustement (rappel)

Qu'est-ce que c'est ? C'est ce qui se passe quand en complexifiant le modèle l'erreur d'apprentissage baisse, alors que l'erreur de généralisation se remet à augmenter.



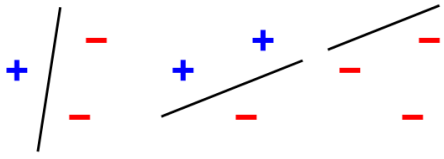
Dimension de Vapnik-Chervonenkis

Qu'est-ce que c'est ? C'est une mesure de la capacité d'un algorithme de classification statistique.

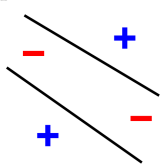
— cardinal du plus grand ensemble de points que l'algorithme peut pulvériser

Pulveriser ? ? Un modèle de classification f_θ pulvérise un ensemble de données $E = (x_1, x_2, \ldots, x_n)$ si, pour tout étiquetage de E, il existe θ tel que f_θ ne fasse aucune erreur dans l'évaluation de cet ensemble de données.

Exemple Une droite en dimension 2
On peut pulvériser 3 points

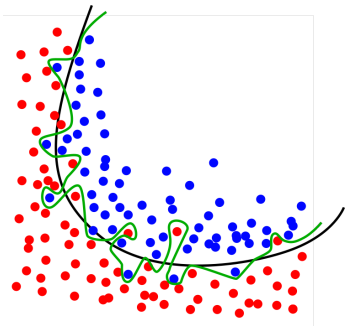


Mais pas 4 points !



Conséquence un modèle de dimension VC trop haute risque le sur-apprentissage par un modèle complexe trop adapté aux données d'apprentissage

Exemple Ici la ligne verte représente un modèle qui fait du sur-ajustement, la noire est meilleure.



SAMME : AdaBoost multi-classes

Principe On ne prédit plus une variable binaire mais $y \in \{1, \ldots, K\}$

SAMME ? Stagewise Additive Modeling using a Multi-class Exponential loss function

Entrée Les éléments nécessaires sont

— un échantillon $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$

— un ensemble de règles faibles meilleures que le hasard

$$P(h(\mathbf{x}) = y) \geq \frac{1}{K} + \gamma, \gamma > 0$$

— le nombre M d'itérations

Initialisation on calcule les poids de départ

$$w_i \leftarrow \frac{1}{n}, i = 1, \ldots, n$$

Itération pour $m = 1$ à M

1. ajuster un classifieur faible $g_m(\mathbf{x})$ sur l'échantillon pondéré par les poids w_i
2. calculer le taux d'erreur

$$\epsilon_m \leftarrow \sum_{i=1}^{n} w_i \mathbf{1}_{\{y_i \neq g_m(\mathbf{x}_i)\}}$$

3. calculer le poids de l'itération m :

$$\alpha_m \leftarrow \log(1-\epsilon_m)/\epsilon_m + \log(K-1)$$

4. mettre à jour les poids des observations

$$w_i \leftarrow \frac{1}{Z_m} w_i \exp\left[\alpha_m \mathbf{1}_{\left\{y_i \neq g_m(\mathbf{x}_i)\right\}}\right],$$

Z_m constante de normalisation.

Sortie elle est encore calculée à partir d'une combinaison linéaire

$$\hat{g}_M(\mathbf{x}) = \arg\max_k \sum_{m=1}^{M} \alpha_m \mathbf{1}_{\left\{g_m(\mathbf{x})=k\right\}}$$

Parenthèse : description des fonctions R

Exemple on obtient avec `?boosting` ou `help(boosting)` la description suivante

```
object <- boosting(formula, data, boos = TRUE,
                   mfinal = 100, control,...)
```

Comment lire ça ?

- les paramètres peuvent avoir une valeur par défaut (`TRUE` pour `boos`)

  Seuls les deux premiers paramètres sont obligatoires ici
- Si la valeur par défaut d'un paramètre vous convient, pas la peine de le spécifier
- si on ne donne pas le nom, ce sera le premier, le second...
- on peut abréger le nom si ce n'est pas ambigu (`mfi` pour `mfinal`, par ex.)
- en général, on ne donne que quelques paramètres
- La signification des paramètres est précisée dans l'aide, c'est utile de la lire

Formes équivalentes on écrit comme on préfère

```
object <- boosting(Y~., mesdonnees)
object <- boosting(Y~., mesdonnees, TRUE)
object <- boosting(formula = Y~.,
                   data = mesdonnees, boos = T)
object <- boosting(dat = mesdonnees, for = Y~.)
```

Adaboost en R avec adabag

Plusieurs packages mais tous n'implémentent pas la version originale (*adaboost.M1*).

- le plus rapide : `fastAdaboot` (écrit en C++), mais fonctionnalités assez basiques
- on choisit `adabag` qui implémente Adaboost et le bagging

Apprendre un modèle on récupère un objet de classe `boosting`

```
object <- boosting(formula, data, boos = TRUE,
                   mfinal = 100, control,...)
```

- `formula` : en général « `Y~.` » si `Y` est la variable qu'on veut prédire (doit être un facteur)
- `data` : les données d'entraînement
- `boos` : quand `TRUE` (le défaut), on utilise un échantillon de bootstrap en utilisant le poids de chaque observation ; sinon, on utilise l'ensemble des données associées à leur poids.
- `mfinal` : nombre total d'arbres M

Choix des arbres on utilise le paramètre `control` de la fonction `boosting`

```
..., control=rpart.control(maxdepth=10, ...)
```

- `maxdepth` contrôle la profondeur totale des arbres
- pour les autres arguments, voir la documentation de `rpart.control`.

Prédiction avec la fonction `predict.boosting` (on peut utiliser juste `predict` parce que le premier argument est de classe `boosting`)

```
pred <- predict(object, newdata,
                newmfinal=length(object$trees), ...)
```

- `object` est retourné par la fonction `boosting`
- `newdata` contient les données à tester
- `newmfinal` est le nombre d'arbres du modèle à utiliser (pour utiliser un modèle plus léger)

Résultat `pred` contient notamment les champs

- `class` : la classe prédite pour chaque individu de `newdata`
- `prob` : la probabilité *a posteriori* de chaque classe pour chaque individu
- `error` : l'erreur moyenne de prédiction
- `confusion` : la matrice de confusion

Influence du nombre d'arbres on considère tous les modèles intermédiaires qui ont été construits

```
evol <- errorevol(object, newdata, newmfinal=mfinal)
```

- `object` est retourné par la fonction `boosting`
- `newdata` contient les données à tester
- `newmfinal` est le nombre d'arbres à utiliser (tous les arbres par défaut)

On affiche avec la fonction `plot.errorevol`

```
plot(x, y = NULL, ...)
```

- `x` est un objet retourné par `errorevol`, par exemple sur les données de test
- `y` (optionnel) est un objet retourné par `errorevol`, typiquement sur les données d'apprentissage

Importance des variables avec la fonction `importanceplot`

```
importanceplot(object, ...)
```

Partie III. Modélisation additive linéaire

Modélisation additive linéaire

Contexte Presque le même que pour AdaBoost
- On a toujours une variable $y \in \{-1, 1\}$ à inférer à partir de règles faibles.
- Cette fois-ci, on se donne un fonction de coût (ou déviance) $L(y, g)$ que l'on cherche à minimiser

Approche On modélise à chaque fois le résidu produit par la solution précédente, on a donc

$$\hat{g}_M(\mathbf{x}) = \sum_{m=1}^{M} \beta_m g_m(\boldsymbol{x}) = \hat{g}_{M-1}(\mathbf{x}) + \beta_M g_M(\mathbf{x})$$

Algorithme Forward staging additive modeling

Entrée Les éléments nécessaires sont
- un échantillon $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$
- une fonction de coût $L(y, g)$
- un ensemble de règles faibles
- le nombre M d'itérations

Initialisation $\hat{g}_0(\mathbf{x}) = 0$.

Itération pour $m = 1$ à M

1. choisir une règle faible g_m et un coefficient β_m qui minimise

$$\sum_{i=1}^{n} L\big(y_i, \hat{g}_{m-1}(\mathbf{x}_i) + \beta_m g_m(\mathbf{x})\big)$$

2. $\hat{g}_m(\mathbf{x}) = \hat{g}_{m-1}(\mathbf{x}) + \beta_m g_m(\mathbf{x})$

Sortie la prédiction est $\operatorname{sign} \hat{g}_M(\mathbf{x})$

Les marges

Qu'est-ce que c'est ? Si on a une prédiction $\operatorname{sign} g(x)$ de la variable binaire y, alors la marge est la valeur $yg(x)$

Propriétés
- la marge est positive si l'objet est bien classifié, et négative sinon
- une grande marge positive est meilleure : elle sera peu sensible au bruit

Conséquences pour la fonction de coût
- elle doit pénalise les marges négatives…
- … mais si elle les pénalise trop, elle sera fragile par rapport aux données d'entraînement avec des labels faux

Pour Adaboost avec la package R `adabag`, on peut les calculer et les représenter avec les fonctions `margins` et `plot.margins`

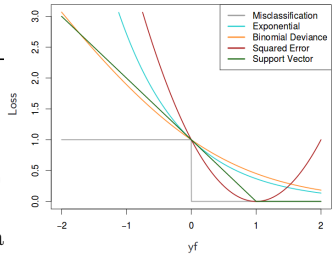
```
plot(margins(object))
```

Fonctions de coût pour la classification

Exponentielle $L(y, g) = \exp(-yg)$
- On peut prouver qu'on retrouve AdaBoost !!
- pourtant l'idée est très différente

Logistique déviance binomiale

$$L(y, g) = \log(1 + \exp(-2yg))$$



- Similaire à AdaBoost a priori
- Moins sensible aux observations mal classifiées

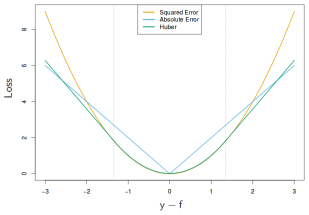
Quadratique $L(y, g) = (y - g)^2$, avec $y \in \mathbb{R}$
- pas bon, puisque le coût devient plus important quand yg est grand
- la fonction de coût doit être décroissante

Fonction de coût pour la régression

Qu'est-ce que c'est ? C'est le même problème, sauf que maintenant $y \in \mathbb{R}$

Quadratique

$$L(y, g) = \frac{1}{2}(y - g)^2$$



- sensible aux valeurs aberrantes (*outliers*)

Linéaire $L(y, g) = |y - g|$
- Plus robuste, mais moins précis pour les petites erreurs

Huber Utilisé pour les statistiques robustes

$$L(y, g) = \begin{cases} (y - g)^2 & \text{si } |y - g| \leq \delta \\ 2\delta|y - g| - \delta^2 & \text{sinon} \end{cases}$$

- combine les bonnes propriétés des deux fonctions précédentes

Partie IV. Gradient Boosting

Principe

Descente de gradient en analyse réelle
- on chercher le minimum d'une fonction convexe u : $\mathbb{R} \mapsto \mathbb{R}$,
- on fixe le paramètre $\lambda > 0$ et on utilise la récurrence

$$x_m = x_{m-1} - \lambda u'(x_{m-1})$$

Adaptation à notre problème
- Ici, on n'a plus un gradient sur une fonction, mais un gradient fonctionnel
  → On cherche une *fonction* minimale, pas un point
- il est facile de calculer le gradient aux points d'observation où y est connu
- par contre, on ne sait pas le faire aux autres points

Idée on va utiliser une règle faible pour modéliser le gradient

Algorithme de Gradient Boosting

Entrée Les éléments nécessaires sont
- un échantillon $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$
- une fonction de coût $L(y, g)$
- un ensemble de règles de régression faibles
- le nombre M d'itérations, le coefficient λ

Initialisation $\hat{g}_0(\mathbf{x}) = \arg\min_g \sum_{i=1}^n L(y_i, g(\mathbf{x}_i))$.

Itération pour $m = 1$ à M

1. calculer le gradient aux points d'observation

$$\dot{r}_{im} = \left[\frac{\partial L(y, g)}{\partial g}\right]_{y=y_i, g=\hat{g}_{m-1}(\mathbf{x}_i)}$$

2. ajuster une règle faible de régression g_m sur l'ensemble $(\mathbf{x}_1, \dot{r}_{1m}), \ldots, (\mathbf{x}_n, \dot{r}_{nm})$
3. $\hat{g}_m(\mathbf{x}) = \hat{g}_{m-1}(\mathbf{x}) - \lambda g_m(\mathbf{x})$

Sortie $\hat{g}_M(\mathbf{x})$ pour une régression, $\operatorname{sign} \hat{g}_M(\mathbf{x})$ pour une classification

Calcul des gradients

Contexte	coût	$\partial L(y, g)/\partial g$
Régression	$\frac{1}{2}(g-y)^2$	$g - y$
	$\lvert g - y\rvert$	$\operatorname{sign}(g - y)$
	Huber	$\begin{cases} g - y & \text{si } \lvert y - g\rvert \le \delta \\ \delta \operatorname{sign}(g - y) & \text{sinon} \end{cases}$
Classification	Logistique	$-\dfrac{2y}{1 + \exp(2yg)}$

Coefficient de régularisation

Qu'est-ce que c'est ? il réduit l'influence des nouveaux termes durant l'itération

Utilisation le coefficient de régularisation (shrinkage) λ fixe le risque d'apprentissage
- λ petit ($\ll 1$) : l'algorithme est plus lent mais limite le sur-apprentissage
- plus λ est petit, plus le nombre d'itérations M doit être grand
- λ vaut 1 pour Adaboost

Autres paramètres importants Il y a finalement assez peu de paramètres
- Nombre d'itérations M
- profondeur des arbres de décision : un stump est très rapide à calculer, mais un arbre plus profond est plus précis

Cas multi-classe

Modèle on écrit la probabilité d'être dans la classe $k \in \{1, \ldots, K\}$ comme

$$p_k(\mathbf{x}) = \frac{e^{g_k(\mathbf{x})}}{\sum_{\ell=1}^K e^{g_\ell(\mathbf{x})}}, \text{ avec } \sum_{\ell=1}^K g_\ell(\mathbf{x}) = 0.$$

Adaptation de l'algorithme on calcule les fonctions $\hat{\mathbf{g}}_m = (\hat{g}_{m1}, \ldots, \hat{g}_{mK})$ en même temps.

Coût la fonction de coût et son gradient pour la k-ième composante sont

$$L(y, \mathbf{g}) = -\sum_{k=1}^K \mathbf{1}_{\{y=k\}} \log p_k(\mathbf{x}), \quad \frac{\partial L(y, \mathbf{g})}{\partial g_k} = \mathbf{1}_{\{y=k\}} - p_k(\mathbf{x})$$

Sortie on calcule les $p_k(\mathbf{x})$ correspondant à $\hat{\mathbf{g}}_M$ et la prédiction est $\arg\max_k p_k(\mathbf{x})$.

Stochastic Gradient Boosting

Idée on n'utilise qu'une parte des données pour calculer les estimateurs g_m
- À chaque fois qu'on doit estimer le gradient, on sélectionne aléatoirement sans remplacement une fraction f des données
- L'algorithme n'est donc plus déterministe !

Propriétés le gain est double :
- exécution plus rapide
- meilleure précision, par réduction de la variance et du sur-ajustement

Valeur typique Friedman (2002), propose une valeur de $f = 0.4$ pour des petits jeux de données (≈ 500) et $f = 0.6$ pour une taille modérée (≈ 5000)

Gradient boosting en R avec gbm

Pourquoi gbm

— implemente le gradient boosting standard
— `gbm` est un bon équilibre entre simplicité et fonctionnalité

Modélisation

```
object <- gbm(formula, distribution = "bernoulli",
    data, n.trees = 100, interaction.depth = 1,
    shrinkage = 0.1, bag.fraction = 0.5,
    train.fraction = 1.0, ...)
```

— `formula`, `data` : comme pour `adaboost`
— `distribution` : `"bernoulli"` pour le coût logistique, `"adaboost"` pour l'exponentiel, `"huberized"`
— `n.trees` : nombre d'itérations M
— `interaction.depth` : profondeur des arbres (stumps : 1)
— `shrinkage` : paramètre de régularisation λ
— `bag.fraction` : fraction f de données à utiliser (gradient boosting stochastique)
— `train.fraction` : proportion des données à utiliser pour l'apprentissage
— et d'autres paramètres à voir dans l'aide

Prédiction avec `predict.gbm`

```
pred <- predict(object, newdata, n.trees, ...)
```

— `model` : l'objet retourné par gbm
— `newdata` : les données de test
— `n.trees` : le nombre d'arbres à utiliser. On peut spécifier un vecteur de tailles pour tout calculer à la fois.
— valeur retournée : liste de prédictions, positif pour valeur 1, négatifs sinon

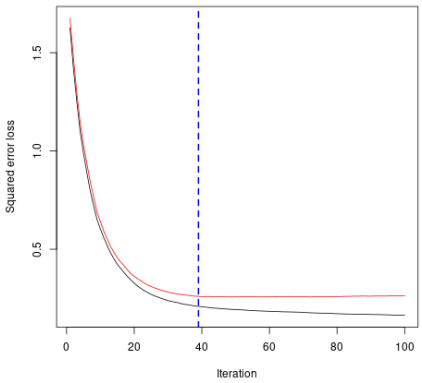
gbm : choix du nombre d'arbres

Méthode on cherche le meilleur sous modèle d'un modèle trop fourni

Fonction `gbm.perf` Calcule le nombre d'arbres idéal et trace des graphes d'erreur

```
best.trees <- gbm.perf(model, plot.it=TRUE,
method)
```

— `model` : l'objet retourné par gbm
— `plot.it` : si vrai, trace un plot de l'erreur sur l'échantillon de apprentissage (noir) et sur l'échantillon de test (rouge)
— `method` : indique la méthode utilisée pour calculer le nombre optimal d'itérations. `"OOB"` calcule l'estimé *out-of-the-bag* et `"test"` utilise la base de test
— valeur retournée : nombre optimal d'arbres



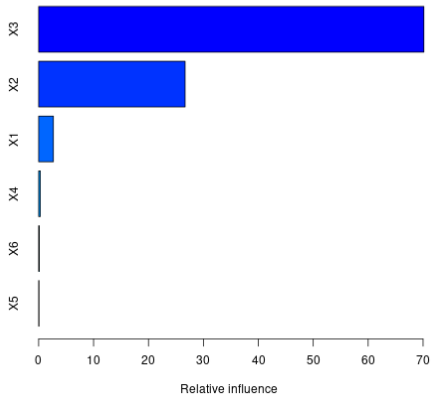
gbm : importance relative des variables

Méthode On ne l'expliquera pas ici, mais elle est décrite dans Friedman (2001).

Importance des variables Calcule et représente l'importance relative des variables dans la fonction de coût

```
summary(object, cBars, n.trees, plotit = TRUE, ...)
```

— `object` : l'objet retourné par gbm
— `cBars` : nombre des plus grandes valeurs à retenir (défaut : toutes)
— `n.trees` : le nombre d'arbres à utiliser (défaut : tous ceux du modèle)
— `plotit` : si `TRUE`, représenter les barres graphiquement
— valeur retournée : une table des influences relatives.



Partie V. XGBoost

eXtreme Gradient Boosting

Historique c'est une variante du gradient boosting qui a été utilisée par beaucoup de gagnants des compétitions en apprentissage

Particularités
- utilisation de l'algorithme de Newton-Raphson au lieu du gradient,
- pénalisation de la complexité des arbres,
- paramètre de randomisation,
- contraction proportionnelle du poids des feuilles.

Implémentation cet algorithme est utilisable de manière efficace dans python, R, Julia et Scala.

Principe

Approximation on suppose qu'on a une fonction $\hat{g}_{m-1}(\mathbf{x})$ et on cherche à minimiser

$$\sum_{i=1}^{n} L\big(y_i, \hat{g}_{m-1}(\mathbf{x}_i) + g_m(\mathbf{x}_i)\big)$$

que l'on développe au second ordre comme

$$\sum_{i=1}^{n} \left[L\big(y_i, \hat{g}_{m-1}(\mathbf{x}_i)\big) + \dot{r}_{im} g_m(\mathbf{x}_i) + \frac{1}{2}\ddot{r}_{im} g_m^2(\mathbf{x}_i) \right],$$

avec

$$\dot{r}_{im} = \left[\frac{\partial L(y,g)}{\partial g}\right]_{y=y_i, g=\hat{g}_{m-1}(\mathbf{x}_i)},$$
$$\ddot{r}_{im} = \left[\frac{\partial^2 L(y,g)}{\partial g^2}\right]_{y=y_i, g=\hat{g}_{m-1}(\mathbf{x}_i)}.$$

Évaluation et optimisation d'arbres

Paramétrisation de l'arbre si g_m est un arbre à T feuilles et $w_1, \ldots, w_T$ les scores de chaque feuille, on note

$$g_m(\mathbf{x}) = w_{q(\mathbf{x})}, \text{ pour } w \in \mathbb{R}^T,\ q : \mathbb{E}^n \mapsto \{1, \ldots, T\}$$

Pénalisation on ajoute un coût $\gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$, qui décourage les arbres complexes

Forme quadratique en combinant les deux termes et en enlevant les termes constants, on a un objectif de la forme

$$\text{obj} = \gamma T + \sum_{j=1}^{T} \left[\dot{R}_{jm} w_j + \frac{1}{2}\big(\ddot{R}_{jm} + \lambda\big) w_j^2 \right],$$

qui est minimal pour

$$w_j^* = -\frac{\dot{R}_{jm}}{\ddot{R}_{jm} + \lambda}, \qquad \text{obj}^* = \gamma T - \frac{1}{2}\sum_{j=1}^{T} \frac{\dot{R}_{jm}^2}{\ddot{R}_{jm} + \lambda}.$$

Utilisation sélection du meilleur arbre par un algorithme glouton.

Utilisation de XGBoost en R

Modélisation

```
object <- xgboost(data, label, params=list(),
                  nrounds, verbose = 1 ...)
```

- `data` : les données d'apprentissage
- `label` : les réponses souhaitées (0 ou 1 pour une classification simple)
- `params` : les paramètres de la méthode (dans une liste).
  - `objective` : la fonction de coût. Le défaut est `"reg:squarederror"`, pour une classification on prend `"binary:logistic"`
  - `max_depth` : profondeur maximale pour les arbres (défaut : 6)
  - `nthread` : nombre de processus à lancer en parallèle (par défaut, tous les cœurs disponibles)
  - et d'autres paramètres à voir dans l'aide
- `nrounds` : nombre maximum d'itérations
- `verbose` : si 1, donne des informations sur la performance
- et d'autres paramètres à voir dans l'aide

Prédiction avec `predict.xgb.Booster`

```
newlabel <- predict(object, newdata, ...)
```

- `object` : l'objet retourné par xgboost
- `newdata` : les données de test
- et d'autres paramètres à voir dans l'aide

Résultat
- les valeurs estimées pour une régression
- pour une classification binaire, un vecteur de valeurs entre 0 et 1
  `newlabel > 0.5` donne une liste de valeurs binaires

Épilogue : le choix des paramètres

Ça dépend Les différents auteurs de packages ont chacun leur approche !

gbm Par défaut on utilise des stumps, $\lambda = 0.001$ et $M = 100$. Dans la documentation, l'auteur dit « en pratique je mets λ à la plus petite valeur possible et je sélectionne M par validation croisée. La performance est meilleure quand λ est le plus petit possible, avec une utilité marginale décroissante quand λ décroît. (...) Je vise en général 3 000 à 10 000 itérations avec un λ entre 0.01 et 0.001. »

xgboost Par défaut, les arbres sont de profondeur maximale 6 et $\lambda = 0.3$. Il n'y a pas de valeur par défaut pour M.

Owen Zhang (vainqueur de la compétition « Avito » de Kaggle) propose
- $M = 10$ à 100, selon la taille des données
- $\lambda = \frac{2 \text{ à } 10}{M}$
- profondeur maximale des arbres parmi $[4, 6, 8, 10]$.