PAPER TITLE

1. Le *boosting*

1.1. **Introduction.**

Le fondement théorique du *boosting* est un article de de 1990 (Shapire (1990)) qui a démontré théoriquement que, sous certaines conditions, il est possible de transformer un modèle prédictif peu performant en un modèle prédictif très performant. Plus précisément, un modèle ayant un pouvoir prédictif arbitrairement élevé (appelé *strong learner*) peut être construit en combinant des modèles simples dont les prédictions ne sont que légèrement meilleures que le hasard (appelé *weak learners*). Le *boosting* est donc une méthode qui combine une approche ensembliste reposant sur un grand nombre de modèles simples avec un entraînement séquentiel: chaque modèle simple (souvent des arbres de décision peu profonds) tâche d'améliorer la prédiction globale en corrigeant les erreurs des prédictions précédentes à chaque étape. Bien qu'une approche de *boosting* puisse en théorie mobiliser différentes classes de *weak learners*, en pratique les *weak learners* utilisés par les algorithmes de *boosting* sont presque toujours des arbres de décision.

S'il existe plusieurs variantes, les algorithmes de *boosting* suivent la même logique :

- Un premier modèle simple et peu performant est entraîné sur les données.
- Un deuxième modèle est entraîné de façon à corriger les erreurs du premier modèle (par exemple en pondérant davantage les observations mal prédites);
- Ce processus est répété en ajoutant des modèles simples, chaque modèle corrigeant les erreurs commises par l'ensemble des modèles précédents;
- Tous ces modèles sont finalement combinés (souvent par une somme pondérée) pour obtenir un modèle complexe et performant.

En termes plus techniques, les algorithmes de *boosting* partagent trois caractéristiques communes:

- Ils visent à **trouver une approximation** $\hat{F}$“)`{=typst} d'une fonction inconnue $F^* : \mathbf{x} \mapsto y$ à partir d'un ensemble d'entraînement $(y_i, \mathbf{x_i})_{i=1,\dots,n}$;
- Ils supposent que la fonction F^* peut être approchée par une **somme pondérée de modèles simples** f de paramètres θ:

$$F(\mathbf{x}) = \sum_{m=1}^{M} \beta_m f(\mathbf{x}, \boldsymbol{\theta}_m)$$

- ils reposent sur une **modélisation additive par étapes**, qui décompose l'entraînement de ce modèle complexe en une **séquence d'entraînements de petits modèles**. Chaque étape de l'entraînement cherche le modèle simple f qui améliore la puissance prédictive du modèle complet, sans modifier les modèles précédents, puis l'ajoute de façon incrémentale à ces derniers:

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \hat{\beta}_m f\left(\mathbf{x}_i, \hat{\boldsymbol{\theta}}_{\mathbf{m}}\right)$$

METTRE ICI UNE FIGURE EN UNE DIMENSION, avec des points et des modèles en escalier qui s'affinent.

1.2. **Les premières approches du *boosting*.**

1.2.1. *Le* boosting *par repondération: Adaboost.*

Dans les années 1990, de nombreux travaux ont tâché de proposer des mise en application du *boosting* (Breiman (1998), Grove & Schuurmans (1998)) et ont comparé les mérites des différentes approches. Deux approches ressortent particulièrement de cette littérature: Adaboost (Adaptive Boosting, Freund & Schapire (1997)) et la *Gradient Boosting Machine* (Friedman (2001)). Ces deux approches reposent sur des principes très différents.

Le principe d'Adaboost consiste à pondérer les erreurs commises à chaque itération en donnant plus d'importance aux observations mal prédites, de façon à obliger les modèles simples à se concentrer sur les observations les plus difficiles à prédire. Voici une esquisse du fonctionnement d'AdaBoost:

- Un premier modèle simple est entraîné sur un jeu d'entraînement dans lequel toutes les observations ont le même poids.
- A l'issue de cette première itération, les observations mal prédites reçoivent une pondération plus élevé que les observations bien prédites, et un deuxième modèle est entraîné sur ce jeu d'entraînement pondéré.
- Ce deuxième modèle est ajouté au premier, puis on repondère à nouveau les observations en fonction de la qualité de prédiction de ce nouveau modèle.
- Cette procédure est répétée en ajoutant de nouveaux modèles et en ajustant les pondérations.

L'algorithme Adaboost a été au coeur de la littérature sur le *boosting* à la fin des années 1990 et dans les années 2000, en raison de ses performances sur les problèmes de classification binaire. Il a toutefois été progressivement remplacé par les algorithmes de *gradient boosting* inventé quelques années plus tard.

1.2.2. *L'invention du* boosting boosting *: la* Gradient Boosting Machine*.*

La *Gradient Boosting Machine* (GBM) propose une approche assez différente: elle introduit le *gradient boosting* en reformulant le *boosting* sous la forme d'un problème de descente de gradient. Voici une esquisse du fonctionnement de la *Gradient Boosting Machine*:

- Un premier modèle simple est entraîné sur un jeu d'entraînement, de façon à minimiser une fonction de perte qui mesure l'écart entre la variable à prédire et la prédiction du modèle.
- A l'issue de cette première itération, on calcule la dérivée partielle (*gradient*) de la fonction de perte par rapport à la prédiction en chaque point de l'ensemble d'entraînement. Ce gradient indique dans quelle direction et dans quelle ampleur la prédiction devrait être modifiée afin de réduire la perte.
- A la deuxième itération, on ajoute un deuxième modèle qui va tâcher d'améliorer le modèle complet en prédisant le mieux possible l'opposé de ce gradient.
- Ce deuxième modèle est ajouté au premier, puis on recalcule la dérivée partielle de la fonction de perte par rapport à la prédiction de ce nouveau modèle.
- Cette procédure est répétée en ajoutant de nouveaux modèles et en recalculant le gradient à chaque étape.

L'approche de *gradient boosting* proposée par Friedman (2001) présente deux grands avantages. D'une part, elle peut être utilisée avec n'importe quelle fonction de perte différentiable, ce qui permet d'appliquer le gradient boosting à de multiples problèmes (régression, classification binaire ou multiclasse, *learning-to-rank*...). D'autre part, elle offre souvent des performances comparables ou supérieures aux autres approches de *boosting*. Le *gradient boosting* d'arbres de décision (*Gradient boosted Decision Trees* - GBDT) est donc devenue l'approche de référence en matière de *boosting*. En particulier, les implémentations modernes du *gradient boosting* comme XGBoost, LightGBM, et CatBoost sont des extensions et améliorations de la *Gradient Boosting Machine*.

2. La mécanique du *gradient boosting*

Depuis la publication de Friedman (2001), la méthode de *gradient boosting* a connu de multiples développements et raffinements, parmi lesquels XGBoost (Chen & Guestrin (2016)), LightGBM (Ke et al. (2017)) et CatBoost (Prokhorenkova et al. (2018)). S'il existe quelques différences entre ces implémentations, elles partagent néanmoins la même mécanique d'ensemble, que la section qui suit va présenter en détail en s'appuyant sur l'implémentation proposée par XBGoost.Chen & Guestrin (2016, .)

Choses importantes à mettre en avant:

- Le boosting est fondamentalement différent des forêts aléatoires. See ESL, chapitre 10.
- Toute la mécanique est indépendante de la fonction de perte choisie. En particulier, elle est applicable indifféremment à des problèmes de classification et de régression.
- Les poids sont calculés par une formule explicite, ce qui rend les calculs extrêmement rapides.
- Comment on interprète le gradient et la hessienne: cas avec une fonction de perte quadratique.
- Le boosting est fait pour overfitter; contrairement aux RF, il n'y a pas de limite à l'overfitting. Donc lutter contre le surapprentissage est un élément particulièrement important de l'usage des algorithmes de boosting.

2.1. **Le modèle à entraîner.**

On veut entraîner un modèle comprenant K arbres de régression ou de classification:

$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$$

Chaque arbre f est défini par trois paramètres:

- sa structure qui est une fonction $q : \mathbb{R}^m \to \{1, ..., T\}$ qui à un vecteur d'inputs $\mathbf{x}$ de dimension m associe une feuille terminale de l'arbre);
- son nombre de feuilles terminales T;
- les valeurs figurant sur ses feuilles terminales $\mathbf{w} \in \mathbb{R}^T$ (appelées poids ou *weights*).

Le modèle est entraîné avec une **fonction-objectif** constituée d'une **fonction de perte** l et d'une **fonction de régularisation** Ω. La fonction de perte mesure la distance entre la prédiction $hat(y)$ et la vraie valeur y et présente généralement les propriétés suivantes: elle est convexe et dérivable deux fois, et atteint son minimum lorsque $hat(y) = y$. La

fonction de régularisation pénalise la complexité du modèle. Dans le cas présent, elle pénalise les arbres avec un grand nombre de feuilles (T élevé) et les arbres avec des poids élevés (w_t élevés en valeur absolue).

$$\mathcal{L}(\phi) = \underbrace{\sum_i l(\hat{y}_i, y_i)}_{\substack{\text{Perte sur les} \\ \text{observations}}} + \underbrace{\sum_k \Omega(f_k)}_{\substack{\text{Fonction de} \\ \text{régularisation}}} \text{ avec } \Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{t=1}^{T} w_j^2 \tag{1}$$

2.2. **Isoler le k-ième arbre.**

La fonction-objectif introduite précédemment est très complexe et ne peut être utilisée directement pour entraîner le modèle, car il faudrait entraîner tous les arbres en même temps. On va donc reformuler donc cette fonction objectif de façon à isoler le k-ième arbre, qui pourra ensuite être entraîné seul, une fois que les $k-1$ arbres précédents auront été entraînés. Pour cela, on note $\hat{y}_i^{(k)}$ la prédiction à l'issue de l'étape k: $\hat{y}_i^{(k)} = \sum_{j=1}^{t} f_j(\mathbf{x}_i)$, et on \$1 la fonction-objectif $\mathcal{L}^{(k)}$ au moment de l'entraînement du k-ième arbre:

$$\begin{aligned}
\mathcal{L}^{(t)} &= \sum_{i=1}^{n} l\Big(y_i, \hat{y}_i^{(t)}\Big) + \sum_{k=1}^{t} \Omega(f_k) \\
&= \sum_{i=1}^{n} l\Big(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)\Big) + \Omega(f_t) + constant
\end{aligned}$$

2.3. **Faire apparaître le gradient.**

Une fois isolé le k-ième arbre, on fait un développement limité d'ordre 2 de $l\Big(y_i, \hat{y}_i^{(k-1)} + f_k(\mathbf{x}_i)\Big)$ au voisinage de $\hat{y}_i^{(k-1)}$, en considérant que la prédiction du k-ième arbre $f_k(\mathbf{x}_i)$ est

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} \left[l\Big(y_i, \hat{y}_i^{(t-1)}\Big) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right] + \Omega(f_t)$$

avec

$$g_i = \frac{\partial l\Big(y_i, \hat{y}_i^{(t-1)}\Big)}{\partial \hat{y}_i^{(t-1)}} \text{ et } h_i = \frac{\partial^2 l\Big(y_i, \hat{y}_i^{(t-1)}\Big)}{\partial \hat{y}_i^{((t-1))^2}}$$

Les termes g_i et h_i désignent respectivement la dérivée première (le gradient) et la dérivée seconde (la hessienne) de la fonction de perte par rapport à la variable prédite. Il est important de noter que les termes (A) et (B) sont constants car les $k-1$ arbres

précédents ont déjà été entraînés et ne sont pas modifiés par l'entraînement du k-ième arbre. On peut donc retirer ces termes pour obtenir la fonction-objectif simplifiée qui sera utilisée pour l'entraînement du k-ième arbre.

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n}\left[g_i f_t(\mathbf{x}_i) + \frac{1}{2}h_i f_t^2(\mathbf{x}_i)\right] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_i^2$$

Cette expression est importante car elle montre qu'on est passé d'un problème complexe où il fallait entraîner un grand nombre d'arbres simultanément (équation Equation 1) à un problème beaucoup plus simple dans lequel il n'y a qu'un seul arbre à entraîner.

2.4. **Calculer les poids optimaux.**

A partir de l'expression précédente, il est possible de faire apparaître les poids w_j du k-ième arbre. Pour une structure d'arbre donnée ($q : \mathbb{R}^m \to \{1, ..., T\}$), on définit $I_j = \{i \mid q(\mathbf{x}_i) = j\}$ l'ensemble des observations situées sur la feuille j puis on réorganise $\tilde{\mathcal{L}}^{(k)}$:

$$\begin{aligned}
\tilde{\mathcal{L}}^{(t)} &= \sum_{j=1}^{T}\sum_{i\in I_j}\Big[\, g_i f_t(\mathbf{x}_i) \qquad\quad + \frac{1}{2}h_i f_t^2(\mathbf{x}_i)\,\Big] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_i^2 \\
&= \sum_{j=1}^{T}\sum_{i\in I_j}\Big[\, g_i w_j \qquad\quad + \frac{1}{2}h_i w_j^2\,\Big] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_i^2 \\
&= \sum_{j=1}^{T}\Big[\, w_j\sum_{i\in I_j} g_i \qquad + \frac{1}{2}w_j^2\sum_{i\in I_j} h_i + \lambda\,\Big] + \gamma T
\end{aligned}$$

Dans la dernière expression, on voit que la fonction de perte simplifiée se reformule comme une combinaison quadratique des poids w_j, dans laquelle les dérivées première et seconde de la fonction de perte interviennent sous forme de pondérations. Tout l'enjeu de l'entraînement devient donc de trouver les poids optimaux w_j qui minimiseront cette fonction de perte, compte tenu de ces opérations.

Il se trouve que le calculs de ces poids optimaux est très simple: pour une structure d'arbre donnée ($q : \mathbb{R}^m \to \{1, ..., T\}$), le poids optimal w_j^{} de la feuille j est donné par l'équation:

$$w_j^* = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i + \lambda}$$

Par conséquent, la valeur optimale de la fonction objectif pour l'arbre q est égale à

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{T}\frac{\left(\sum_{i\in I_j} g_i\right)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T \tag{2}$$

Cette équation est utile car elle permet de comparer simplement la qualité de deux arbres, et de déterminer lequel est le meilleur.

2.5. **Construire le k-ième arbre.**

Dans la mesure où elle permet de comparer des arbres, on pourrait penser que l'équation Equation 2 est suffisante pour choisir directement le k-ième arbre: il suffirait d'énumérer les arbres possibles, de calculer la qualité de chacun d'entre eux, et de retenir le meilleur. Bien que cette approche soit possible théoriquement, elle est inemployable en pratique car le nombre d'arbres possibles est extrêmement élevé. Par conséquent, le k-ième arbre n'est pas défini en une fois, mais construit de façon gloutonne:

REFERENCE A LA PARTIE CART/RF?

- on commence par le noeud racine et on cherche le *split* qui réduit au maximum la perte en séparant les données d'entraînement entre les deux noeuds-enfants.
- pour chaque noeud enfant, on cherche le *split* qui réduit au maximum la perte en séparant en deux la population de chacun de ces noeuds.
- Cette procédure recommence jusqu'à que l'arbre ait atteint sa taille maximale (définie par une combinaison d'hyperparamètres d$1s dans la partie **référence à ajouter**).

2.6. **Choisir les *splits*.**

Traduire split par critère de partition?

Reste à comprendre comment le critère de partition optimal est choisi à chaque étape de la construction de l'arbre. Imaginons qu'on envisage de décomposer la feuille I en deux nouvelles feuilles I_L et I_R (avec $I = I_L \cup I_R$), selon une condition logique reposant sur une variable et une valeur de cette variable (exemple: $x_6 > 11$). Par application de l'équation Equation 2, le gain potentiel induit par ce critère de partition est égal à:

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma \tag{3}$$

Cette dernière équation est au coeur de la mécanique du *gradient boosting* car elle permet de comparer les critères de partition possibles. Plus précisément, l'algorithme de

détermination des critère de partition (*split finding algorithm*) consiste en une double boucle sur les variables et les valeurs prises par ces variables, qui énumère un grand nombre de critères de partition et mesure le gain associé à chacun d'entre eux avec l'équation Equation 3. Le critère de partition retenu est simplement celui dont le gain est le plus élevé.

L'algorithme qui détermine les critère de partition est un enjeu de performance essentiel dans le *gradient boosting*. En effet, utiliser l'algorithme le plus simple (énumérer tous les critères de partition possibles, en balayant toutes les valeurs de toutes les variables) s'avère très coûteux dès lors que les données contiennent soit un grand nombre de variables, soit des variables continues prenant un grand nombre de valeurs. C'est pourquoi les algorithmes de détermination des critère de partition ont fait l'objet de multiples améliorations et optimisations visant à réduire leur coût computationnel sans dégrader la qualité des critères de partition.

2.7. **La suite.**

2.7.1. *Les moyens de lutter contre l'*overfitting*:.*

- le *shrinkage*;
- le subsampling des lignes et des colonnes;
- les différentes pénalisations.

2.7.2. *Les hyperparamètres.*

Hyperparamètre	Description	Valeur par défaut
`booster`	Le type de *weak learner* utilisé	`'gbtree'`
`learning_rate`	Le taux d'apprentissage	0.3
`max_depth`	La profondeur maximale des arbres	6
`max_leaves`	Le nombre maximal de feuilles des arbres	0
`min_child_weight`	Le poids minimal qu'une feuille doit contenir	1
`n_estimators`	Le nombre d'arbres	100
`lambda` ou `reg_lambda`	La pénalisation L2	1
`alpha` ou `reg_alpha`	La pénalisation L1	0
`gamma`	Le gain minimal nécessaire pour ajouter un noeud supplémentaire	0
`tree_method`	La méthode utilisée pour rechercher les splits	`'hist'`
`max_bin`	Le nombre utilisés pour discrétiser les variables continues	0
`subsample`	Le taux d'échantillonnage des données d'entraîenment	1
`sampling_method`	La méthode utilisée pour échantillonner les données d'entraînement	`'uniform'`
`colsample_bytree` `colsample_bylevel` `colsample_bynode`	Taux d'échantillonnage des colonnes par arbre, par niveau et par noeud	1, 1 et 1
`scale_pos_weight`	Le poids des observations de la classe positive (classification uniquement)	1
`sample_weight`	La pondération des données d'entraînement	1
`enable_categorical`	Activer le support des variables catégorielles	`False`
`max_cat_to_onehot`	Nombre de modalités en-deça duquel XGBoost utilise le *one-hot-encoding*	A COMPLETER
`max_cat_threshold`	Nombre maximal de catégories considérées dans le partitionnement optimal des variables catégorielles	A COMPLETER

Table 1: Les principaux hyperparamètres d'XGBoost

2.7.3. *La préparation des données.*

- les variables catégorielles:
  - ordonnées: passer en integer;
  - non-ordonnées: OHE ou approche de Fisher.
- les variables continues:
  - inutile de faire des transformations monotones.
  - Utile d'ajouter des transformations non monotones.

2.7.4. *Les fonctions de perte.*

2.8. **Liste des hyperparamètres d'une RF.**

Source: Probst et al. (2019)

- structure of each individual tree:
  - dudu
  - dudu
  - dudu
- structure and size of the forest:
- The level of randomness (je dirais plutôt :)

References

Breiman, L. (1998). Rejoinder: arcing classifiers. *The Annals of Statistics*, *26*(3), 841–849.

Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *Proceedings of the 22nd Acm Sigkdd International Conference on Knowledge Discovery and Data Mining*, 785–794.

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, *55*(1), 119–139.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 1189–1232.

Grove, A. J., & Schuurmans, D. (1998). Boosting in the limit: Maximizing the margin of learned ensembles. *AAAI/IAAI*, 692–699.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., & Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. *Advances in Neural Information Processing Systems*, *30*.

Probst, P., Wright, M. N., & Boulesteix, A.-L. (2019). Hyperparameters and tuning strategies for random forest. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, *9*(3), e1301.

Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. (2018). CatBoost: unbiased boosting with categorical features. *Advances in Neural Information Processing Systems*, *31*.

Shapire, R. (1990). The strength of weak learning. *Machine Learning*, *5*(2).