

INTRODUCTION AUX MÉTHODES ENSEMBLISTES

ABSTRACT

Voici un bien livre Typst.

1. SURVOL DES MÉTHODES ENSEMBLISTES

Principe: cette partie propose une présentation intuitive des méthodes ensemblistes, à destination notamment des *managers* sans bagage en *machine learning*. Elle ne contient aucune formalisation mathématique.

1.1. Principe des méthodes ensemblistes.

1.1.1. Pourquoi utiliser des méthodes ensemblistes?.

Avantages:

- Méthodes adaptées à un grand nombre de cas d'usage de la statistique publique:
 - Elles sont notamment applicables à tous les problèmes pour lesquels on utilise une régression linéaire ou une régression logistique);
 - Elles s'appliquent à des données tabulaires (enregistrements en lignes, variables en colonnes), situation très fréquente dans la statistique publique.
- Performances quasi systématiquement supérieures aux méthodes économétriques traditionnelles;
- Scalabilité: ces méthodes peuvent être appliquées à des données volumineuses;
- Coût d'entrée modéré (comparé à des approches plus avancées comme le *deep learning*)

Inconvénients:

- Bagage informatique minimal (une bonne maîtrise de Python ou R est un prérequis) -> comme toutes les méthodes statistiques je dirais. Et avec l'arrivée de chat GPT & co ou avec une bonne documentation le coût d'entrée est moindre.
- Temps d'entraînement potentiellement long, notamment pour l'optimisation des hyperparamètres.
- Ces méthodes peuvent nécessiter une puissance de calcul importante et/ou une mémoire vive de grande taille.
- Interprétabilité moindre que les méthodes économétriques traditionnelles (et encore, ça se discute)
- Risque de surapprentissage.

1.1.2. *L'union fait la force.*

Plutôt que de chercher à construire d'emblée un unique modèle très complexe, les approches ensemblistes visent à obtenir un modèle très performant en combinant un grand nombre de modèles simples.

Il existe quatre grandes approches ensemblistes:

- le *bagging*;
- la *random forest*
- le *stacking*;
- le *boosting*.

Le présent document se concentre sur deux approches: la *random forest* et le *boosting*.

Les méthodes ensemblistes consistent à entraîner plusieurs modèles de base, puis à combiner les résultats obtenus afin de produire une prédiction consolidée. Les modèles de base, dits "apprenants faibles" ("weak learners"), sont généralement peu complexes. Le choix de ces modèles et la manière dont leurs prédictions sont combinées sont des facteurs clés pour la performance de ces approches.

Les méthodes ensemblistes peuvent être divisées en deux grandes familles selon qu'elles s'appuient sur des modèles entraînés en parallèle ou de manière imbriquée ou séquentielle. Lorsque les modèles sont entraînés en parallèle, chaque modèle de base est entraîné pour la même tâche de prédiction en utilisant un sous-ensemble de l'échantillon d'entraînement, un sous-ensemble de variables, ou une combinaison des deux. Les techniques les plus populaires sont le bagging et la forêt aléatoire. Lorsque les modèles de base sont entraînés de manière séquentielle, appelée boosting, chaque modèle vise à minimiser l'erreur de prédiction du modèle précédent. Les implémentations les plus courantes du boosting sont actuellement XGBoost, CatBoost et LightGBM.

1.2. **Comment fonctionnent les méthodes ensemblistes?**

Quatre temps:

- les arbres de décision et de régression (CART);
- les forêts aléatoires;
- le boosting.

1.2.1. *Le point de départ: les arbres de décision et de régression.*

Présenter *decision tree* et *regression tree*. Reprendre des éléments du chapitre 9 de <https://bradleyboehmke.github.io/HOML/>

Principes d'un arbre:

- fonction constante par morceaux;
- partition de l'espace;
- interactions entre variables.

Illustration, et représentation graphique (sous forme d'arbre et de graphique).

1.2.2. Critères de performance et sélection d'un modèle.

La performance d'un modèle augmente généralement avec sa complexité, jusqu'à atteindre un maximum, puis diminue. L'objectif est d'obtenir un modèle qui minimise à la fois le sous-apprentissage (biais) et le sur-apprentissage (variance). C'est ce qu'on appelle le compromis biais/variance. Cette section présente très brièvement les critères utilisés pour évaluer et comparer les performances des modèles.

1.3. Le *bagging*, les *random forests* et le *boosting*.

Il existe plusieurs types de méthodes ensemblistes, toutes ayant en commun la combinaison de modèles élémentaires. Le présent document présente les 3 principales méthodes : le Bagging, la Random Forests et le Boosting.

1.3.1. Le bagging (*Bootstrap Aggregating*).

Présenter le *bagging* en reprenant des éléments du chapitre 10 de <https://bradleyboehmke.github.io/HOML>. Mettre une description de l'algorithme en pseudo-code?

- Présentation avec la figure en SVG;
- Illustration avec un cas d'usage de classification en deux dimensions.

Le bagging, ou Bootstrap Aggregating, est une méthode ensembliste qui comporte trois étapes principales :

Création de sous-échantillons : À partir du jeu de données initial, plusieurs sous-échantillons sont générés par échantillonnage aléatoire avec remise (bootstrapping). Chaque sous-échantillon a la même taille que le jeu de données original, mais peut contenir des observations répétées, tandis que d'autres peuvent être omises. Cette technique permet de diversifier les données d'entraînement en créant des échantillons variés, ce qui aide à réduire la variance et à améliorer la robustesse du modèle.

Entraînement parallèle : Un modèle distinct est entraîné sur chaque sous-échantillon de manière indépendante. Cette technique permet un gain d'efficacité et un meilleur contrôle du surapprentissage (overfitting).

Agrégation des prédictions : Les prédictions des modèles sont combinées pour produire le résultat final. En classification, la prédiction finale est souvent déterminée par un vote majoritaire, tandis qu'en régression, elle correspond généralement à la moyenne des prédictions. En combinant les prédictions de plusieurs modèles, le bagging renforce la stabilité et la performance globale de l'algorithme, notamment en réduisant la variance des prédictions.

Le bagging appliqué aux arbres de décision est la forme la plus courante de cette technique.

Le bagging est particulièrement efficace pour réduire la variance des modèles, ce qui les rend moins vulnérables au surapprentissage. Cette caractéristique est particulièrement utile dans les situations où la robustesse et la capacité de généralisation des modèles sont cruciales. De plus, comme le bagging repose sur des processus indépendants, l'exécution est plus rapide dans des environnements distribués.

Cependant, bien que chaque modèle de base soit construit indépendamment sur des sous-échantillons distincts, les variables utilisées pour générer ces modèles ne sont pas forcément indépendantes d'un modèle à l'autre. Dans le cas du bagging appliqué aux arbres de décision, cela conduit souvent à des arbres ayant une structure similaire.

Les forêts aléatoires apportent une amélioration à cette approche en réduisant cette corrélation entre les arbres, ce qui permet d'augmenter la précision de l'ensemble du modèle.

1.3.2. Les random forests.

Expliquer que les *random forests* sont une amélioration du *bagging*, en reprenant des éléments du chapitre 11 de <https://bradleyboehmke.github.io/HOML/>

- Présentation avec la figure en SVG;
- Difficile d'illustrer avec un exemple (car on ne peut pas vraiment représenter le *feature sampling*);
- Bien insister sur les avantages des RF: 1/ faible nombre d'hyperparamètres; 2/ faible sensibilité aux hyperparamètres; 3/ limite intrinsèque à l'overfitting.

1.3.3. Le boosting.

Reprendre des éléments du chapitre 12 de <https://bradleyboehmke.github.io/HOML/> et des éléments de la formation boosting.

Le *boosting* combine l'**approche ensembliste** avec une **modélisation additive par étapes** (*forward stagewise additive modeling*).

- Présentation;

- Avantage du boosting: performances particulièrement élevées.
- Inconvénients: 1/ nombre élevé d'hyperparamètres; 2/ sensibilité des performances aux hyperparamètres; 3/ risque élevé d'overfitting.
- Préciser qu'il est possible d'utiliser du subsampling par lignes et colonnes pour un algorithme de boosting. Ce point est abordé plus en détail dans la partie sur les hyperparamètres.

1.3.4. Comparaison RF-GBDT.

Les forêts aléatoires et le *gradient boosting* paraissent très similaires au premier abord: il s'agit de deux approches ensemblistes, qui construisent des modèles très prédictifs performants en combinant un grand nombre d'arbres de décision. Mais en réalité, ces deux approches présentent plusieurs différences fondamentales:

- Les deux approches reposent sur des fondements théoriques différents: la loi des grands nombres pour les forêts aléatoires, la théorie de l'apprentissage statistique pour le *boosting*.
- Les arbres n'ont pas le même statut dans les deux approches. Dans une forêt aléatoire, les arbres sont entraînés indépendamment les uns des autres et constituent chacun un modèle à part entière, qui peut être utilisé, représenté et interprété isolément. Dans un modèle de *boosting*, les arbres sont entraînés séquentiellement, ce qui implique que chaque arbre n'a pas aucun sens indépendamment de l'ensemble des arbres qui l'ont précédé dans l'entraînement.
- Les points d'attention dans l'entraînement ne sont pas les mêmes: arbitrage puissance-corrélation dans la RF, arbitrage puissance-overfitting dans le *boosting*.
- *overfitting*: borne théorique à l'*overfitting* dans les RF, contre pas de borne dans le *boosting*. Deux conséquences: 1/ lutter contre l'overfitting est essentiel dans l'usage du *boosting*; 2/ le *boosting* est plus sensible au bruit et aux erreurs sur y que la RF.
- Conditions d'utilisation: la RF peut être utilisée en OOB, pas le *boosting*.
- Complexité d'usage: peu d'hyperparamètres dans les RF, contre un grand nombre dans le *boosting*.

2. MÉTHODES ENSEMBLISTES: PRÉSENTATION MATHÉMATISÉE

3. LA BRIQUE ÉLÉMENTAIRE: L'ARBRE DE DÉCISION

Les arbres de décision sont des outils puissants en apprentissage automatique, utilisés pour des tâches de classification et de régression. Ces algorithmes non paramétriques consistent à diviser l'espace des caractéristiques en sous-ensembles homogènes à l'aide de règles simples, afin de faire des prédictions. Malgré leur simplicité apparente, les arbres de décision sont capables de saisir des relations complexes et non linéaires entre les variables (ou *caractéristiques*) d'un jeu de données.

3.1. Le principe fondamental : partitionner pour prédire.

Imaginez que vous souhaitiez prédire le prix d'une maison en fonction de sa superficie et de son nombre de pièces. L'espace des caractéristiques (superficie et nombre de pièces) est vaste, et les prix des maisons (la *réponse* à prédire) sont très variables. Pour prédire le prix des maisons, l'idée est de diviser cet espace en zones plus petites, où les maisons ont des prix similaires, et d'attribuer une prédiction identique à toutes les maisons situées dans la même zone.

3.1.1. Les défis du partitionnement optimal.

L'objectif principal est de trouver la partition de l'espace des caractéristiques qui offre les meilleures prédictions possibles. Cependant, cet objectif se heurte à plusieurs difficultés, et la complexité du problème augmente rapidement avec le nombre de caractéristiques et la taille de l'échantillon:

1. **Infinité des découpages possibles** : Il existe une infinité de façons de diviser l'espace des caractéristiques.
2. **Complexité de la paramétrisation** : Il est difficile de représenter tous ces découpages avec un nombre limité de paramètres.
3. **Optimisation complexe** : Même avec une paramétrisation, trouver le meilleur découpage nécessite une optimisation complexe, souvent irréaliste en pratique.

3.1.2. Les solutions apportées par les arbres de décision.

Pour surmonter ces difficultés, les méthodes d'arbres de décision, et notamment la plus célèbre, l'algorithme CART (Classification And Regression Tree, Breiman et al. (1984)), adoptent deux approches clés :

1. Simplification du partitionnement de l'espace

Au lieu d'explorer tous les découpages possibles, les arbres de décision partitionnent l'espace des caractéristiques en plusieurs régions distinctes (non chevauchantes) en appliquant des règles de décision simples. Les règles suivantes sont communément adoptées:

- **Découpages binaires simples** : À chaque étape, l'algorithme divise une région de l'espace en deux sous-régions en se basant sur une seule caractéristique (ou *variable*) et en définissant un seul seuil (ou *critère*) pour cette segmentation. Concrètement, cela revient à poser une question du type : "La valeur de la caractéristique X dépasse-t-elle un certain seuil ?" Par exemple : "La superficie de la maison est-elle supérieure à 100 m² ?". Les deux réponses possibles ("Oui" ou "Non") génèrent deux nouvelles sous-régions distinctes de l'espace, chacune correspondant à un sous-ensemble de données plus homogène.
- **Prédictions locales** : Lorsque l'algorithme s'arrête, une prédiction simple est faite dans chaque région. Il s'agit souvent de la moyenne des valeurs cibles dans cette région (régression) ou de la classe majoritaire (classification).

Ces règles de découpage rendent le problème d'optimisation plus simple mais également plus interprétable.

2. Optimisation gloutonne (greedy)

Plutôt que d'optimiser toutes les divisions simultanément, les arbres de décision utilisent une approche simplifiée, récursive et séquentielle :

- **Division étape par étape** : À chaque étape, l'arbre choisit la meilleure division possible sur la base d'un critère de réduction de l'hétérogénéité intra-région. En revanche, il ne prend pas en compte les étapes d'optimisation futures.
- **Critère local** : La décision est basée sur la réduction immédiate de l'impureté ou de l'erreur de prédiction (par exemple, la réduction de la variance pour la régression). Ce processus est répété pour chaque sous-région, ce qui permet d'affiner progressivement la partition de l'espace en fonction des caractéristiques les plus discriminantes.

Cette méthode dite "gloutonne" (*greedy*) s'avère efficace pour construire un partitionnement de l'espace des caractéristiques, car elle décompose un problème d'optimisation complexe en une succession de problèmes plus simples et plus rapides à résoudre. Le résultat obtenu n'est pas nécessairement un optimum global, mais il s'en approche raisonnablement et surtout rapidement.

Le terme "arbre de décision" provient de la structure descendante en forme d'arbre inversé qui émerge lorsqu'on utilise un algorithme glouton pour découper l'espace des caractéristiques en sous-ensemble de réponses homogènes de manière récursive. À chaque étape, deux nouvelles branches sont créées et forment une nouvelle partition de l'espace des caractéristiques.

Une fois entraîné, un arbre de décision est une fonction **constante par morceaux** défini sur l'espace des caractéristiques. En raison de leur nature **non-continue** et **non-différen-**

table, il est impossible d'utiliser des méthodes d'optimisation classiques reposant sur le calcul de gradients.

3.1.3. Terminologie et structure d'un arbre de décision.

Nous présentons la structure d'un arbre de décision et les principaux éléments qui le composent.

- **Nœud Racine (Root Node)** : Le nœud racine est le point de départ de l'arbre de décision, il est situé au sommet de l'arbre. Il contient l'ensemble des données d'entraînement avant toute division. À ce niveau, l'algorithme cherche la caractéristique la plus discriminante, c'est-à-dire celle qui permet de diviser les données de manière à optimiser une fonction de perte (comme l'indice de Gini pour la classification ou la variance pour la régression).
- **Nœuds Internes (Internal Nodes)** : Les nœuds internes sont les points intermédiaires où l'algorithme CART applique des règles de décision pour diviser les données en sous-ensembles plus petits. Chaque nœud interne représente une question ou condition basée sur une caractéristique particulière (par exemple, "La superficie de la maison est-elle supérieure à 100 m² ?"). À chaque étape, une seule caractéristique (la superficie) et un seul seuil (supérieur à 100) sont utilisés pour faire la division.
- **Branches**: Les branches sont les connexions entre les nœuds, elles illustrent le chemin que les données suivent en fonction des réponses aux questions posées dans les nœuds internes. Chaque branche correspond à une décision binaire, "Oui" ou "Non", qui oriente les observations vers une nouvelle subdivision de l'espace des caractéristiques.
- **Nœuds Terminaux ou Feuilles (Leaf Nodes ou Terminal Nodes)** : Les nœuds terminaux, situés à l'extrémité des branches, sont les points où le processus de division s'arrête. Ils fournissent la prédiction finale.
 - En **classification**, chaque feuille correspond à une classe prédite (par exemple, "Oui" ou "Non").
 - En **régression**, chaque feuille fournit une valeur numérique prédite (comme le prix estimé d'une maison).

Figure illustrative : Une représentation visuelle de la structure de l'arbre peut être utile ici pour illustrer les concepts de nœuds, branches et feuilles.

3.1.4. Illustration.

Supposons que nous souhaitions prédire le prix d'une maison en fonction de sa superficie et de son nombre de pièces. Un arbre de décision pourrait procéder ainsi :

1. **Première division** : “La superficie de la maison est-elle supérieure à 100 m² ?”
 - Oui : Aller à la branche de gauche.
 - Non : Aller à la branche de droite.
2. **Deuxième division (branche de gauche)** : “Le nombre de pièces est-il supérieur à 4 ?”
 - Oui : Prix élevé (par exemple, plus de 300 000 €).
 - Non : Prix moyen (par exemple, entre 200 000 € et 300 000 €).
3. **Deuxième division (branche de droite)** : “Le nombre de pièces est-il supérieur à 2 ?”
 - Oui : Prix moyen (par exemple, entre 150 000 € et 200 000 €).
 - Non : Prix bas (par exemple, moins de 150 000 €).

Cet arbre utilise des règles simples pour diviser l'espace des caractéristiques (superficie et nombre de pièces) en sous-groupes homogènes et fournir une prédiction (estimer le prix d'une maison).

Figure illustrative

3.2. L'algorithme CART, un partitionnement binaire récursif.

L'algorithme CART (Classification and Regression Trees) proposé par Breiman, Friedman, Olshen, & Stone (1984) est une méthode utilisée pour construire des arbres de décision, que ce soit pour des tâches de classification ou de régression. L'algorithme CART fonctionne en partitionnant l'espace des caractéristiques en sous-ensembles de manière récursive, en suivant une logique de décisions binaires à chaque étape. Ce processus est itératif et suit plusieurs étapes clés.

3.2.1. Définir une fonction d'impureté adaptée au problème.

La **fonction d'impureté** est une mesure locale utilisée dans la construction des arbres de décision pour évaluer la qualité des divisions à chaque nœud. Elle quantifie le degré d'hétérogénéité des observations dans un nœud par rapport à la variable cible (classe pour la classification, ou valeur continue pour la régression). Plus précisément, une mesure d'impureté est conçue pour croître avec la dispersion dans un nœud. Un nœud est dit **pur** lorsque toutes les observations qu'il contient appartiennent à la même classe (classification) ou présentent des valeurs similaires/identiques (régression).

L'algorithme CART utilise ce type de mesure pour choisir les divisions qui créent des sous-ensembles plus homogènes que le nœud parent. À chaque étape de construction, l'algorithme sélectionne la division qui réduit le plus l'impureté, afin de garantir des nœuds de plus en plus homogènes au fur et à mesure que l'arbre se développe.

Le choix de la fonction d'impureté dépend du type de problème :

- **Classification** : L'**indice de Gini** ou l'**entropie** sont très souvent utilisées pour évaluer la dispersion des classes dans chaque nœud.
- **Régression** : La **somme des erreurs quadratiques** (SSE) est souvent utilisée pour mesurer la variance des valeurs cibles dans chaque nœud.

3.2.1.1. Mesures d'impureté classiques pour les problèmes de classification.

Dans le cadre de la classification, l'objectif est de partitionner les données de manière à ce que chaque sous-ensemble (ou région) soit le plus homogène possible en termes de classe prédite. Plusieurs mesures d'impureté sont couramment utilisées pour évaluer la qualité des divisions.

Propriété-définition d'une mesure d'impureté

Pour un nœud t contenant K classes, une **mesure d'impureté** $I(t)$ est une fonction qui quantifie l'hétérogénéité des classes dans ce nœud. Elle doit satisfaire les propriétés suivantes :

- **Pureté maximale** : Lorsque toutes les observations du nœud appartiennent à une seule classe, c'est-à-dire que la proportion $p_k = 1$ pour une classe k et $p_j = 0$ pour toutes les autres classes $j \neq k$, l'impureté est minimale et $I(t) = 0$. Cela indique que le nœud est **entièrement pur**, ou homogène.
- **Impureté maximale** : Lorsque les observations sont réparties de manière uniforme entre toutes les classes, c'est-à-dire que $p_k = \frac{1}{K}$ pour chaque classe k , l'impureté atteint son maximum. Cette situation reflète une **impureté élevée**, car le nœud est très hétérogène et contient une forte incertitude sur la classe des observations.

1. L'indice de Gini

L'**indice de Gini** est l'une des fonctions de perte les plus couramment utilisées pour la classification. Il mesure la probabilité qu'un individu sélectionné au hasard dans un nœud soit mal classé si on lui attribue une classe au hasard, en fonction de la distribution des classes dans ce nœud.

Pour un nœud t contenant K classes, l'indice de Gini $G(t)$ est donné par :

$$G(t) = 1 - \sum_{k=1}^K p_k^2 \quad (1)$$

où p_k est la proportion d'observations appartenant à la classe k dans le nœud t .

Critère de choix : L'indice de Gini est souvent utilisé parce qu'il est simple à calculer et capture bien l'homogénéité des classes au sein d'un nœud. Il privilégie les partitions où une classe domine fortement dans chaque sous-ensemble.

2. L'entropie (ou entropie de Shannon)

L'**entropie** est une autre mesure de l'impureté utilisée dans les arbres de décision. Elle mesure la quantité d'incertitude ou de désordre dans un nœud, en s'appuyant sur la théorie de l'information.

Pour un nœud t contenant K classes, l'entropie $E(t)$ est définie par :

$$E(t) = - \sum_{k=1}^K p_k \log(p_k) \quad (2)$$

où p_k est la proportion d'observations de la classe k dans le nœud t .

Critère de choix : L'entropie a tendance à être plus sensible aux changements dans les distributions des classes que l'indice de Gini, car elle attribue un poids plus élevé aux événements rares (valeurs de p_k très faibles). Elle est souvent utilisée lorsque l'erreur de classification des classes minoritaires est particulièrement importante.

3. Taux d'erreur

Le **taux d'erreur** est une autre mesure de l'impureté parfois utilisée dans les arbres de décision. Il représente la proportion d'observations mal classées dans un nœud.

Pour un nœud t , le taux d'erreur $TE(t)$ est donné par :

$$TE(t) = 1 - \max(p_k) \quad (3)$$

où $\max(p_k)$ est la proportion d'observations appartenant à la classe majoritaire dans le nœud.

Critère de choix : Bien que le taux d'erreur soit simple à comprendre, il est moins souvent utilisé dans la construction des arbres de décision parce qu'il est moins sensible que l'indice de Gini ou l'entropie aux petits changements dans la distribution des classes.

3.2.1.2. Mesures d'impureté classiques pour les problèmes de régression.

Dans les problèmes de régression, l'objectif est de partitionner les données de manière à réduire au maximum la variabilité des valeurs au sein de chaque sous-ensemble. Pour mesurer cette variabilité, la somme des erreurs quadratiques (SSE) est la fonction d'impureté la plus couramment employée. Elle évalue l'impureté d'une région en quantifiant à quel point les valeurs de cette région s'écartent de la moyenne locale.

1. Somme des erreurs quadratiques (SSE) ou variance

La **somme des erreurs quadratiques** (ou **SSE**, pour *Sum of Squared Errors*) est une mesure qui quantifie la dispersion des valeurs dans un nœud par rapport à la moyenne des valeurs dans ce nœud.

Formule : Pour un nœud t , contenant N observations avec des valeurs y_i , la SSE est donnée par :

$$\text{SSE}(t) = \sum_{i=1}^N (y_i - \hat{y})^2 \quad (4)$$

où \hat{y} est la moyenne des valeurs y_i dans le nœud t .

Propriété :

- Si toutes les valeurs de y_i dans un nœud sont proches de la moyenne \hat{y} , la SSE sera faible, indiquant une homogénéité élevée dans le nœud.
- En revanche, une SSE élevée indique une grande variabilité dans les valeurs, donc un nœud impur.

Critère de choix : La somme des erreurs quadratiques (SSE) est particulièrement sensible aux écarts élevés entre les valeurs observées et la moyenne prédite. En cherchant à minimiser la SSE, les modèles visent à former des nœuds dans lesquels les valeurs des observations sont aussi proches que possible de la moyenne locale.

3.2.2. Identifier la partition binaire maximisant la réduction de l'impureté.

Une fois la mesure d'impureté définie, l'algorithme CART examine toutes les divisions binaires possibles de l'espace des caractéristiques. À chaque nœud, et pour chaque caractéristique, il cherche à identifier le **seuil optimal**, c'est-à-dire le seuil qui minimise le plus efficacement l'impureté des deux sous-ensembles générés. L'algorithme compare ensuite toutes les divisions potentielles (caractéristiques et seuils optimaux associés à chaque nœud) et sélectionne celle qui entraîne la réduction maximale de l'impureté.

Prenons l'exemple d'une caractéristique continue, telle que la superficie d'une maison :

- Si l'algorithme teste la règle "Superficie > 100 m²", il calcule la fonction de perte pour les deux sous-ensembles générés par cette règle ("Oui" et "Non").
- Ce processus est répété pour différentes valeurs seuils afin de trouver la partition qui minimise le plus efficacement l'impureté au sein des sous-ensembles.

3.2.3. Répéter le processus jusqu'à atteindre un critère d'arrêt.

L'algorithme CART poursuit le partitionnement de l'espace des caractéristiques en appliquant de manière récursive les mêmes étapes : identification de la caractéristique et du seuil optimal pour chaque nœud, puis sélection du partitionnement binaire qui maximise la réduction de l'impureté. Ce processus est répété jusqu'à ce qu'un **critère d'arrêt** soit atteint, par exemple :

- **Profondeur maximale de l'arbre :** Limiter le nombre de divisions successives pour éviter un arbre trop complexe.

- **Nombre minimum d'observations par feuille** : Empêcher la création de feuilles contenant très peu d'observations, ce qui réduirait la capacité du modèle à généraliser.
- **Réduction minimale de l'impureté à chaque étape**

3.2.4. *Elagage (pruning).*

3.2.5. *Prédire.*

Une fois l'arbre construit, la prédiction pour une nouvelle observation s'effectue en suivant les branches de l'arbre, en partant du nœud racine jusqu'à un nœud terminal (ou feuille). À chaque nœud interne, une décision est prise en fonction des valeurs des caractéristiques de l'observation, ce qui détermine la direction à suivre vers l'un des sous-ensembles. Ce cheminement se poursuit jusqu'à ce que l'observation atteigne une feuille, où la prédiction finale est effectuée.

- En **classification**, la classe attribuée est celle majoritaire dans la feuille atteinte.
- En **régression**, la valeur prédite est généralement la moyenne des valeurs cibles des observations dans la feuille.

3.2.6. *Critères de qualité et ajustements.*

Pour améliorer la performance de l'arbre, on peut ajuster les hyperparamètres tels que la profondeur maximale ou le nombre minimum d'observations dans une feuille. De plus, des techniques comme la **prédiction avec arbres multiples** (bagging, forêts aléatoires) permettent de surmonter les limites des arbres individuels, souvent sujets au surapprentissage.

3.3. **Avantages et limites de cette approche.**

3.3.1. *Avantages.*

- **Interprétabilité** : Les arbres de décision sont faciles à comprendre et à visualiser.
- **Simplicité** : Pas besoin de transformations complexes des données.
- **Flexibilité** : Ils peuvent gérer des caractéristiques numériques et catégorielles, ainsi que les valeurs manquantes.
- **Gestion des interactions** : Modèles non paramétriques, pas d'hypothèses sur les lois par les variables. Ils capturent naturellement les interactions entre les caractéristiques.

3.3.2. *Limites.*

- **Surapprentissage** : Les arbres trop profonds peuvent surapprendre les données d'entraînement.
- **Optimisation locale** : L'approche gloutonne peut conduire à des solutions sous-optimales globalement (optimum local).
- **Stabilité** : De petits changements dans les données peuvent entraîner des changements significatifs dans la structure de l'arbre (manque de robustesse).

4. LE BAGGING

Le bagging, ou “bootstrap aggregating”, est une méthode ensembliste qui vise à améliorer la stabilité et la précision des algorithmes d'apprentissage automatique en réduisant la variance des prédictions (Breiman (1996)). Elle repose sur la construction de plusieurs modèles (plusieurs versions d'un même modèle dans la plupart des cas) entraînés sur des échantillons distincts générés par des techniques de rééchantillonnage (*bootstrap*). Ces modèles sont ensuite combinés pour produire une prédiction agrégée, souvent plus robuste et généralisable que celle obtenue par un modèle unique.

4.1. Principe du bagging.

Le bagging comporte trois étapes principales:

- **L'échantillonnage bootstrap** : L'échantillonnage bootstrap consiste à créer des échantillons distincts en tirant aléatoirement avec remise des observations du jeu de données initial. Chaque échantillon *bootstrap* contient le même nombre d'observations que le jeu de données initial, mais certaines observations sont répétées (car sélectionnées plusieurs fois), tandis que d'autres sont omises.
- **L'entraînement de plusieurs modèles** : Un modèle (aussi appelé *apprenant de base* ou *weak learner*) est entraîné sur chaque échantillon bootstrap. Les modèles peuvent être des arbres de décision, des régressions ou tout autre algorithme d'apprentissage. Le bagging est particulièrement efficace avec des modèles instables, tels que les arbres de décision non élagués.
- **L'agrégation des prédictions** : Les prédictions de tous les modèles sont ensuite agrégées, en procédant généralement à la moyenne (ou à la médiane) des prédictions dans le cas de la régression, et au vote majoritaire (ou à la moyenne des probabilités prédites pour chaque classe) dans le cas de la classification, afin d'obtenir des prédictions plus précises et généralisables.

4.2. Pourquoi (et dans quelles situations) le bagging fonctionne.

Certains modèles sont très sensibles aux données d'entraînement, et leurs prédictions sont très instables d'un échantillon à l'autre. L'objectif du bagging est de construire un prédicteur plus précis en agrégeant les prédictions de plusieurs modèles entraînés sur des échantillons (légèrement) différents les uns des autres.

Breiman (1996) montre que cette méthode est particulièrement efficace lorsqu'elle est appliquée à des modèles très instables, dont les performances sont particulièrement sensibles aux variations du jeu de données d'entraînement, et peu biaisés.

Cette section vise à mieux comprendre comment (et sous quelles conditions) l'agrégation par bagging permet de construire un prédicteur plus performant.

Dans la suite, nous notons $\varphi(x, L)$ un prédicteur (d'une valeur numérique dans le cas de la *régression* ou d'un label dans le cas de la *classification*), entraîné sur un ensemble d'apprentissage L , et prenant en entrée un vecteur de caractéristiques x .

4.2.1. La régression: réduction de l'erreur quadratique moyenne par agrégation.

Dans le contexte de la **régression**, l'objectif est de prédire une valeur numérique Y à partir d'un vecteur de caractéristiques x . Un modèle de régression $\phi(x, L)$ est construit à partir d'un ensemble d'apprentissage L , et produit une estimation de Y pour chaque observation x .

4.2.1.1. Définition du prédicteur agrégé.

Dans le cas de la régression, le **prédicteur agrégé** est défini comme suit :

$$\phi_A(x) = E_L[\phi(x, L)] \quad (5)$$

où $\phi_A(x)$ représente la prédiction agrégée, $E_L[\cdot]$ correspond à l'espérance prise sur tous les échantillons d'apprentissage possibles L , chacun étant tiré selon la même distribution que le jeu de données initial, et $\phi(x, L)$ correspond à la prédiction du modèle construit sur l'échantillon d'apprentissage L .

4.2.1.2. La décomposition biais-variance.

Pour mieux comprendre comment l'agrégation améliore la performance globale d'un modèle individuel $\phi(x, L)$, revenons à la **décomposition biais-variance** de l'erreur quadratique moyenne (il s'agit de la mesure de performance classiquement considérée dans un problème de régression):

$$E_L[(Y - \phi(x, L))^2] = \underbrace{(E_L[\phi(x, L) - Y])^2}_{\text{Biais}^2} + \underbrace{E_L[(\phi(x, L) - E_L[\phi(x, L)])^2]}_{\text{Variance}} \quad (6)$$

- Le **biais** est la différence entre la valeur observée Y que l'on souhaite prédire et la prédiction moyenne $E_L[\phi(x, L)]$. Si le modèle est sous-ajusté, le biais sera élevé.
- La **variance** est la variabilité des prédictions $(\phi(x, L))$ autour de leur moyenne $(E_L[\phi(x, L)])$. Un modèle avec une variance élevée est très sensible aux fluctuations au sein des données d'entraînement: ses prédictions varient beaucoup lorsque les données d'entraînement se modifient.

L'équation 6 illustre l'**arbitrage biais-variance** qui est omniprésent en *machine learning*: plus la complexité d'un modèle s'accroît (exemple: la profondeur d'un arbre), plus son biais sera plus faible (car ses prédictions seront de plus en plus proches des données d'entraînement), et plus sa variance sera élevée (car ses prédictions, étant très proches des données d'entraînement, auront tendance à varier fortement d'un jeu d'entraînement à l'autre).

4.2.1.3. L'inégalité de Breiman (1996).

Breiman (1996) compare l'erreur quadratique moyenne d'un modèle individuel avec celle du modèle agrégé et démontre l'inégalité suivante :

$$(Y - \phi_A(x))^2 \leq E_L[(Y - \phi(x, L))^2] \quad (7)$$

- Le terme $(Y - \phi_A(x))^2$ représente l'erreur quadratique du **prédicteur agrégé** $\phi_A(x)$;
- Le terme $E_L[(Y - \phi(x, L))^2]$ est l'erreur quadratique moyenne d'un **prédicteur individuel** $\phi(x, L)$ entraîné sur un échantillon aléatoire L . Cette erreur varie en fonction des données d'entraînement.

Cette inégalité montre que **l'erreur quadratique moyenne du prédicteur agrégé est toujours inférieure ou égale à la moyenne des erreurs des prédicteurs individuels**. Puisque le biais du prédicteur agrégé est identique au biais du prédicteur individuel, alors l'inégalité précédente implique que la **variance du modèle agrégé** $\phi_A(x)$ est **toujours inférieure ou égale** à la variance moyenne d'un modèle individuel :

$$\text{Var}(\phi_A(x)) = \text{Var}(E_L[\phi(x, L)]) \leq E_L[\text{Var}(\phi(x, L))]$$

Autrement dit, le processus d'agrégation réduit l'erreur de prédiction globale en réduisant la **variance** des prédictions, tout en conservant un biais constant.

Ce résultat ouvre la voie à des considérations pratiques immédiates. Lorsque le modèle individuel est instable et présente une variance élevée, l'inégalité $\text{Var}(\phi_A(x)) \leq E_L[\text{Var}(\phi(x, L))]$ est forte, ce qui signifie que l'agrégation peut améliorer significativement la performance globale du modèle. En revanche, si $\phi(x, L)$ varie peu d'un ensemble d'entraînement à un autre (modèle stable avec variance faible), alors $\text{Var}(\phi_A(x))$ est proche de $E_L[\text{Var}(\phi(x, L))]$, et la réduction de variance apportée par l'agrégation est faible. Ainsi, **le bagging est particulièrement efficace pour les modèles instables**, tels que les arbres

de décision, mais moins efficace pour les modèles stables tels que les méthodes des k plus proches voisins.

4.2.2. La classification: vers un classificateur presque optimal par agrégation.

Dans le cas de la classification, le mécanisme de réduction de la variance par le bagging permet, sous une certaine condition, d'atteindre un **classificateur presque optimal** (*nearly optimal classifier*). Ce concept a été introduit par Breiman (1996) pour décrire un modèle qui tend à classer une observation dans la classe la plus probable, avec une performance approchant celle du classificateur Bayésien optimal (la meilleure performance théorique qu'un modèle de classification puisse atteindre).

Pour comprendre ce résultat, introduisons $Q(j | x) = E_L(1_{\varphi(x, L)=j}) = P(\varphi(x, L) = j)$, la probabilité qu'un modèle $\varphi(x, L)$ prédise la classe j pour l'observation x , et $P(j | x)$, la probabilité réelle (conditionnelle) que x appartienne à la classe j .

4.2.2.1. Définition : classificateur order-correct.

Un classificateur $\varphi(x, L)$ est dit **order-correct** pour une observation x si, en espérance, il identifie **correctement la classe la plus probable**, même s'il ne prédit pas toujours avec exactitude les probabilités associées à chaque classe $Q(j | x)$.

Cela signifie que si l'on considérait tous les ensemble de données possibles, et que l'on évaluait les prédictions du modèle en x , la majorité des prédictions correspondraient à la classe à laquelle il a la plus grande probabilité vraie d'appartenir $P(j | x)$.

Formellement, un prédicteur est dit "order-correct" pour une entrée x si :

$$\operatorname{argmax}_j Q(j | x) = \operatorname{argmax}_j P(j | x)$$

où $P(j | x)$ est la vraie probabilité que l'observation x appartienne à la classe j , et $Q(j | x)$ est la probabilité que x appartienne à la classe j prédite par le modèle $\varphi(x, L)$.

Un classificateur est **order-correct** si, pour **chaque** observation x , la classe qu'il prédit correspond à celle qui a la probabilité maximale $P(j | x)$ dans la distribution vraie.

4.2.2.2. Prédicteur agrégé en classification: le vote majoritaire.

Dans le cas de la classification, le prédicteur agrégé est défini par le **vote majoritaire**. Cela signifie que si K classificateurs sont entraînés sur K échantillons distincts, la classe prédite pour x est celle qui reçoit **le plus de votes** de la part des modèles individuels.

Formellement, le classificateur agrégé $\varphi_A(x)$ est défini par :

$$\varphi_A(x) = \operatorname{argmax}_j \sum_L I(\varphi(x, L) = j) = \operatorname{argmax}_j Q(j | x)$$

4.2.2.3. *Performance globale: convergence vers un classificateur presque optimal.*

Breiman (1996) montre que si chaque prédicteur individuel $\varphi(x, L)$ est order-correct pour une observation x , alors le prédicteur agrégé $\varphi A(x)$, obtenu par **vote majoritaire**, atteint la performance optimale pour cette observation, c'est-à-dire qu'il converge vers la classe ayant la probabilité maximale $P(j \mid x)$ pour l'observation x lorsque le nombre de prédicteurs individuels augmente. Le vote majoritaire permet ainsi de **réduire les erreurs aléatoires** des classificateurs individuels.

Le classificateur agrégé ϕA est optimal s'il prédit systématiquement la classe la plus probable pour l'observation x dans toutes les régions de l'espace.

Cependant, dans les régions de l'espace où les classificateurs individuels ne sont pas order-corrects (c'est-à-dire qu'ils se trompent majoritairement sur la classe d'appartenance), l'agrégation par vote majoritaire n'améliore pas les performances. Elles peuvent même se détériorer par rapport aux modèles individuels si l'agrégation conduit à amplifier des erreurs systématiques (biais).

4.3. **L'échantillage par bootstrap peut détériorer les performances théoriques du modèle agrégé.**

En pratique, au lieu d'utiliser tous les ensembles d'entraînement possibles L , le bagging repose sur un nombre limité d'échantillons bootstrap tirés avec remise à partir d'un même jeu de données initial, ce qui peut introduire des biais par rapport au prédicteur agrégé théorique.

Les échantillons bootstrap présentent les limites suivantes :

- Une **taille effective réduite par rapport au jeu de données initial**: Bien que chaque échantillon bootstrap présente le même nombre d'observations que le jeu de données initial, environ 1/3 des observations (uniques) du jeu initial sont absentes de chaque échantillon bootstrap (du fait du tirage avec remise). Cela peut limiter la capacité des modèles à capturer des relations complexes au sein des données (et aboutir à des modèles individuels sous-ajustés par rapport à ce qui serait attendu théoriquement), en particulier lorsque l'échantillon initial est de taille modeste.
- Une **dépendance entre échantillons** : Les échantillons bootstrap sont tirés dans le même jeu de données, ce qui génère une dépendance entre eux, qui réduit la diversité des modèles. Cela peut limiter l'efficacité de la réduction de variance dans le cas de la régression, voire accroître le biais dans le cas de la classification.
- Une **couverture incomplète de l'ensemble des échantillons possibles**: Les échantillons bootstrap ne couvrent pas l'ensemble des échantillons d'entraînement

possibles, ce qui peut introduire un biais supplémentaire par rapport au prédicteur agrégé théorique.

4.4. Le bagging en pratique.

4.4.1. *Quand utiliser le bagging en pratique.*

Le bagging est particulièrement utile lorsque les modèles individuels présentent une variance élevée et sont instables. Dans de tels cas, l'agrégation des prédictions peut réduire significativement la variance globale, améliorant ainsi la performance du modèle agrégé. Les situations où le bagging est recommandé incluent typiquement:

- Les modèles instables : Les modèles tels que les arbres de décision non élagués, qui sont sensibles aux variations des données d'entraînement, bénéficient grandement du bagging. L'agrégation atténue les fluctuations des prédictions dues aux différents échantillons.
- Les modèles avec biais faibles: En classification, si les modèles individuels sont order-corrects pour la majorité des observations, le bagging peut améliorer la précision en renforçant les prédictions correctes et en réduisant les erreurs aléatoires.

Inversement, le bagging peut être moins efficace ou même néfaste dans certaines situations :

- Les modèles stables avec variance faible : Si les modèles individuels sont déjà stables et présentent une faible variance (par exemple, la régression linéaire), le bagging n'apporte que peu d'amélioration, car la réduction de variance supplémentaire est minimale.
- La présence de biais élevée : Si les modèles individuels sont biaisés, entraînant des erreurs systématiques, le bagging peut amplifier ces erreurs plutôt que de les corriger. Dans de tels cas, il est préférable de s'attaquer d'abord au biais des modèles avant de considérer l'agrégation.
- Les échantillons de petite taille : Avec des ensembles de données limités, les échantillons bootstrap peuvent ne pas être suffisamment diversifiés ou représentatifs, ce qui réduit l'efficacité du bagging et peut augmenter le biais des modèles.

Ce qui qu'il faut retenir: le bagging peut améliorer substantiellement la performance des modèles d'apprentissage automatique lorsqu'il est appliqué dans des conditions appropriées. Il est essentiel d'évaluer la variance et le biais des modèles individuels, ainsi que la taille et la représentativité du jeu de données, pour déterminer si le bagging est une stratégie adaptée. Lorsqu'il est utilisé judicieusement, le bagging peut conduire à des modèles plus robustes et précis, exploitant efficacement la puissance de l'agrégation pour améliorer la performance des modèles individuels.

4.4.2. *Comment utiliser le bagging en pratique.*

4.4.2.1. *Combien de modèles agréger?*

“Optimal performance is often found by bagging 50–500 trees. Data sets that have a few strong predictors typically require less trees; whereas data sets with lots of noise or multiple strong predictors may need more. Using too many trees will not lead to overfitting. However, it’s important to realize that since multiple models are being run, the more iterations you perform the more computational and time requirements you will have. As these demands increase, performing k-fold CV can become computationally burdensome.”

4.4.2.2. *Evaluation du modèle: cross validation et échantillon Out-of-bag (OOB).*

“A benefit to creating ensembles via bagging, which is based on resampling with replacement, is that it can provide its own internal estimate of predictive performance with the out-of-bag (OOB) sample (see Section 2.4.2). The OOB sample can be used to test predictive performance and the results usually compare well compared to k-fold CV assuming your data set is sufficiently large (say $n \geq 1,000$). Consequently, as your data sets become larger and your bagging iterations increase, it is common to use the OOB error estimate as a proxy for predictive performance.”

4.5. **Mise en pratique (exemple avec code).**

Ou bien ne commencer les mises en pratique qu’avec les random forest ?

4.6. **Interprétation.**

5. LA FORÊT ALÉATOIRE

La forêt aléatoire (*random forests*) est une méthode ensembliste qui consiste à agréger plusieurs arbres de décision pour améliorer la précision et la robustesse des prédictions du modèle final. Cette méthode s’appuie sur la technique du bagging, qui consiste à entraîner chaque arbre sur un échantillon (*bootstrap*) tiré au hasard à partir du jeu de données initial. Toutefois, la forêt aléatoire va plus loin en introduisant un degré supplémentaire de randomisation : pour chaque division lors de la construction d’un arbre, elle **sélectionne aléatoirement** un sous-ensemble de variables sur lequel sera fondé le critère de séparation. Cette randomisation supplémentaire **réduit la corrélation** entre les arbres, ce qui permet de renforcer la performance globale du modèle agrégé.

5.1. Principe de la forêt aléatoire.

Les forêts aléatoires reposent sur trois éléments essentiels :

- **L'échantillonnage bootstrap**: Chaque arbre est construit à partir d'un échantillon bootstrap, un échantillon aléatoire tiré avec remise du jeu de données d'entraînement.
- **La sélection aléatoire de caractéristiques (*variables*)** : Lors de la construction de chaque arbre, et à chaque nœud de celui-ci, un sous-ensemble aléatoire de caractéristiques est sélectionné. La meilleure division est ensuite choisie parmi ces caractéristiques aléatoires.
- **L'agrégation des prédictions** : Comme pour le bagging, les prédictions de tous les arbres sont agrégées, en procédant généralement à la moyenne (ou à la médiane) des prédictions dans le cas de la régression, et au vote majoritaire (ou à la moyenne des probabilités prédites pour chaque classe) dans le cas de la classification, afin d'obtenir des prédictions plus précises et généralisables.

5.2. Pourquoi (et dans quelles situations) la random forest fonctionne.

5.3. Le *boosting*.

5.3.1. Introduction.

Le fondement théorique du *boosting* est un article 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} 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 :

$$F(\mathbf{x}) = \sum_{m=1}^M \beta_m f(\mathbf{x}, \theta_m) \quad (8)$$

- 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:

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \hat{\beta}_m f(\mathbf{x}_i, \hat{\theta}_m) \quad (9)$$

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

5.3.2. Les premières approches du boosting.

5.3.2.1. 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 élevée 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 répond 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.

5.3.2.2. L'invention du 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*.

5.4. 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 XGBoost. 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.

5.4.1. Le modèle à entraîner.

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

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

Chaque arbre f est défini par trois paramètres:

- sa structure qui est une fonction $q : \mathbb{R}^m \rightarrow \{1, \dots, T\}$ 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 $\text{hat}(y) = y$. 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).

$$\mathcal{L}(\phi) = \underbrace{\sum_i l(\hat{y}_i, y_i)}_{\text{Perte sur les observations}} + \underbrace{\sum_k \Omega(f_k)}_{\text{Fonction de régularisation}} \quad \text{avec} \quad \Omega(f) = \gamma T + \frac{1}{2} \lambda \sum_{t=1}^T w_t^2 \quad (10)$$

5.4.2. 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 : $\hat{y}_i^{(k)} = \sum_{j=1}^k f_j(\mathbf{x}_i)$, et on écrit la fonction-objectif $\mathcal{L}^{(k)}$ au moment de l'entraînement du k -ième arbre:

$$\begin{aligned} \mathcal{L}^{(k)} &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(k)}) + \sum_{j=1}^k \Omega(f_j) \\ &= \sum_{i=1}^n l\left(y_i, \hat{y}_i^{(k-1)} + \underbrace{f_k(\mathbf{x}_i)}_{k\text{-ième arbre}}\right) + \sum_{j=1}^{k-1} \Omega(f_j) + \Omega(f_k) \end{aligned} \quad (11)$$

5.4.3. Faire apparaître le gradient.

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

$$\mathcal{L}^{(k)} \approx \underbrace{\sum_{i=1}^n l(y_i, \hat{y}_i^{(k-1)}) + g_i f_k(\mathbf{x}_i)}_{(A)} + \frac{1}{2} h_i f_k^2(\mathbf{x}_i) + \underbrace{\sum_{j=1}^{k-1} \Omega(f_j) + \Omega(f_k)}_{(B)} \quad (12)$$

$$\text{avec} \quad g_i = \frac{\partial l(y_i, \hat{y}_i^{(k-1)})}{\partial \hat{y}_i^{(k-1)}} \quad \text{et} \quad h_i = \frac{\partial^2 l(y_i, \hat{y}_i^{(k-1)})}{\partial (\hat{y}_i^{(k-1)})^2} \quad (13)$$

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.

$$\tilde{\mathcal{L}}^{(k)} = \sum_{i=1}^n [g_i f_k(\mathbf{x}_i) + \frac{1}{2} h_i f_k^2(\mathbf{x}_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \quad (14)$$

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 10) à un problème beaucoup plus simple dans lequel il n'y a qu'un seul arbre à entraîner.

5.4.4. 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 ($q : \mathbb{R}^m \rightarrow \{1, \dots, T\}$), on définit $I_j = \{i \mid q(\mathbf{x}_i) = j\}$ l'ensemble des observations situées sur la feuille j puis on réorganise $\tilde{\mathcal{L}}^{(k)}$:

$$\begin{aligned} \tilde{\mathcal{L}}^{(k)} &= \sum_{j=1}^T \sum_{i \in I_j} \left[g_i f_k(\mathbf{x}_i) + \frac{1}{2} h_i f_k^2(\mathbf{x}_i) \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\ &= \sum_{j=1}^T \sum_{i \in I_j} \left[g_i w_j + \frac{1}{2} h_i w_j^2 \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\ &= \sum_{j=1}^T \left[w_j \left(\sum_{i \in I_j} g_i \right) + \frac{1}{2} w_j^2 \left(\sum_{i \in I_j} h_i + \lambda \right) \right] + \gamma T \end{aligned} \quad (15)$$

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 ($q : \mathbb{R}^m \rightarrow \{1, \dots, T\}$), le poids optimal w_j^* de la feuille j est donné par l'équation:

$$w_j^* = - \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \quad (16)$$

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

$$\tilde{\mathcal{L}}^{(k)}(q) = - \frac{1}{2} \sum_{j=1}^T \frac{\left(\sum_{i \in I_j} g_i \right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T \quad (17)$$

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

5.4.5. Construire le k -ième arbre.

Dans la mesure où elle permet de comparer des arbres, on pourrait penser que l'équation 17 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 décrits dans la partie **référence à ajouter**).

5.4.6. 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 17, le gain potentiel induit par ce critère de partition est égal à:

$$\text{Gain}_{\text{split}} = \frac{1}{2} \left[\frac{\left(\sum_{i \in I_L} g_i \right)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{\left(\sum_{i \in I_R} g_i \right)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{\left(\sum_{i \in I} g_i \right)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma \quad (18)$$

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 critères 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 18. Le critère de partition retenu est simplement celui dont le gain est le plus élevé.

L'algorithme qui détermine les critères 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 critères de partition ont fait l'objet de multiples améliorations et opti-

misations visant à réduire leur coût computationnel sans dégrader la qualité des critères de partition.

5.4.7. *La suite.*

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

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

5.4.7.2. *Les hyperparamètres.*

Table 1: Les principaux hyperparamètres d'XGBoost

Hyperparamètre	Description	Valeur par défaut
booster	Le type de <i>weak learner</i> 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'entraînement	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 <i>one-hot-encoding</i>	A COM- PLETER
max_cat_threshold	Nombre maximal de catégories considérées dans le partitionnement optimal des variables catégorielles	A COM- PLETER

5.4.7.3. 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.

5.4.7.4. *Les fonctions de perte.*

5.4.8. *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 :)

6. COMMENT (BIEN) UTILISER LES APPROCHES ENSEMBLISTES

6.1. **Guide d'entraînement des forêts aléatoires.**

Cette section rassemble et synthétise des recommandations sur l'entraînement des forêts aléatoires disponibles dans la littérature, en particulier dans Probst, Wright, & Boulesteix (2019).

6.1.1. *Mode de construction d'une forêt aléatoire.*

Le processus pour construire une Random Forest se résume comme suit :

- Sélectionnez le nombre d'arbres à construire (n_trees).
- Pour chaque arbre, effectuez les étapes suivantes :
 - Générer un échantillon bootstrap à partir du jeu de données.
 - Construire un arbre de décision à partir de cet échantillon :
 - À chaque nœud de l'arbre, sélectionner un sous-ensemble aléatoire de caractéristiques (mtry).
 - Trouver la meilleure division parmi ce sous-ensemble et créer des nœuds enfants.

- Arrêter la croissance de l'arbre selon des critères de fin spécifiques (comme une taille minimale de nœud), mais sans élaguer l'arbre.
- Agréger les arbres pour effectuer les prédictions finales :
 - Régression : la prédiction finale est la moyenne des prédictions de tous les arbres.
 - Classification : chaque arbre vote pour une classe, et la classe majoritaire est retenue.



6.1.2. *Quelle implémentation utiliser?*

Il existe de multiples implémentations des forêts aléatoires. Le présent document présente et recommande l'usage de deux implémentations de référence: le *package* R *ranger* et le *package* Python *scikit-learn*. Il est à noter qu'il est possible d'entraîner des forêts aléatoires avec les algorithmes XGBoost et LightGBM, mais il s'agit d'un usage avancé qui n'est recommandé en première approche. Cette approche est présentée dans la partie **REFERENCE A LA PARTIE USAGE AVANCE**.

6.1.3. *Quels sont les hyperparamètres des forêts aléatoires?*

Cette section décrit en détail les principaux hyperparamètres des forêts aléatoires listés dans le tableau Table 2. Les noms des hyperparamètres utilisés sont ceux figurant dans le *package* R *ranger*, et dans le *package* Python *scikit-learn*. Il arrive qu'ils portent un nom différent dans d'autres implémentations des *random forests*, mais il est généralement facile de s'y retrouver en lisant attentivement la documentation.

Table 2: Les principaux hyperparamètres des forêts aléatoires

Hyperparamètre		Description
 ranger	 scikit-learn	
mtry	max_features	Le nombre de variables candidates à chaque noeud
replacement		L'échantillonnage des données se fait-il avec ou sans remise?
sample.fraction	max_samples	Le taux d'échantillonnage des données
min.node.size	min_samples_leaf	Nombre minimal d'observations nécessaire pour qu'un noeud puisse être partagé
num.trees	n_estimators	Le nombre d'arbres
splitrule	criterion	Le critère de choix de la règle de division des noeuds intermédiaires
min.bucket	min_samples_split	Nombre minimal d'observations dans les noeuds terminaux
max.depth	max_depth	Profondeur maximale des arbres

- Le **nombre de variables candidates à chaque noeud** contrôle l'échantillonnage des variables lors de l'entraînement. La valeur par défaut est fréquemment \sqrt{p} pour la classification et $p/3$ pour la régression. C'est l'hyperparamètre qui a le plus fort effet sur la performance de la forêt aléatoire. Une valeur plus basse aboutit à des arbres plus différents, donc moins corrélés (car ils reposent sur des variables différentes), mais ces arbres peuvent être moins performants car ils reposent parfois sur des variables peu pertinentes. Inversement, une valeur plus élevée du nombre de variables candidates aboutit à des arbres plus performants, mais plus corrélés. C'est en particulier le cas si seulement certaines variables sont très prédictives, car ce sont ces variables qui apparaîtront dans la plupart des arbres.
- Le **taux d'échantillonnage** et le **mode de tirage** contrôlent le plan d'échantillonnage des données d'entraînement. Les valeurs par défaut varient d'une implémentation à l'autre; dans le cas de `ranger`, le taux d'échantillonnage est de 63,2% sans remise, et de 100% avec remise. L'implémentation `scikit-learn` ne propose pas le tirage sans remise. Ces hyperparamètres ont des effets sur la performance similaires à ceux du nombre de variables candidates, mais dans une moindre ampleur. Un taux d'échantillonnage plus faible aboutit à des arbres plus différents et donc moins corrélés (car ils sont entraînés sur des échantillons très différents), mais ces arbres peuvent être peu performants car ils sont entraînés sur des échantillons de petite taille.

Inversement, un taux d'échantillonnage élevé aboutit à des arbres plus performants mais plus corrélés. Les effets de l'échantillonnage avec ou sans remise sur la performance de la forêt aléatoire sont moins clairs et ne font pas consensus. Les travaux les plus récents suggèrent toutefois qu'il est préférable d'échantillonner sans remise (Probst, Wright, & Boulesteix (2019)).

- Le **nombre minimal d'observations dans les noeuds terminaux** contrôle la taille des noeuds terminaux. La valeur par défaut est faible dans la plupart des implémentations (entre 1 et 5). Il n'y a pas vraiment de consensus sur l'effet de cet hyperparamètre sur les performances. En revanche, il est certain que le temps d'entraînement décroît fortement avec cet hyperparamètre: une valeur faible implique des arbres très profonds, avec un grand nombre de noeuds. Il peut donc être utile de fixer ce nombre à une valeur plus élevée pour accélérer l'entraînement, en particulier si les données sont volumineuses.
- Le **nombre d'arbres** par défaut varie selon les implémentations (500 dans `ranger`, 100 dans `scikit-learn`). Il s'agit d'un hyperparamètre particulier car il n'est associé à aucun arbitrage en matière de performance: la performance de la forêt aléatoire croît avec le nombre d'arbres, puis se stabilise à un niveau approximativement constant. Le nombre optimal d'arbres est donc intuitivement celui à partir duquel la performance de la forêt ne croît plus (ce point est détaillé plus bas). Il est important de noter que ce nombre optimal dépend des autres hyperparamètres. Par exemple, un taux d'échantillonnage faible et un nombre faible de variables candidates à chaque noeud aboutissent à des arbres peu corrélés, mais peu performants, ce qui requiert probablement un plus grand nombre d'arbres.
- Le **critère de choix de la règle de division des noeuds intermédiaires**: la plupart des implémentations des forêts aléatoires retiennent par défaut l'impureté de Gini pour la classification et la variance pour la régression, même si d'autres critères de choix ont été proposés dans la littérature. A ce stade, aucun critère de choix ne paraît systématiquement supérieur aux autres en matière de performance. Le lecteur intéressé pourra se référer à la discussion détaillée dans Probst, Wright, & Boulesteix (2019).

6.1.4. *Comment entraîner une forêt aléatoire?*

Comme indiqué dans la partie **REFERENCE A AJOUTER**, la performance prédictive d'une forêt aléatoire varie en fonction de deux critères essentiels: elle croît avec le pouvoir prédictif moyen des arbres, et décroît avec la corrélation entre les arbres. La recherche d'arbres très prédictifs pouvant aboutir à augmenter la corrélation entre eux, l'objectif de

l'entraînement d'une forêt aléatoire revient à trouver le meilleur arbitrage possible entre pouvoir prédictif et corrélation.

REFERENCES

- Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24, 123–140.
- Breiman, L. (1998). Rejoinder: arcing classifiers. *The Annals of Statistics*, 26(3), 841–849.
- Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). Cart. *Classification and Regression Trees*.
- 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).

Email address:

URL: