UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE

TRAVAIL D'ÉTUDE ET DE RECHERCHE

MASTER 1 MAEF

Le boosting en apprentissage statistique et XGBoost

Réalisé par :
DE PAZ LUCIEN
CLEMENTE PIRES
Christopher

Encadré par :
JEAN MARC BARDET

Année Universitaire 2022-2023

Table des matières

Introduction

Le concept de **Machine Learning** est une branche de l'intelligence artificielle qui permet un apprentissage automatique par l'utilisation de données sans avoir été explicitement programmé pour cela. Apparu dès 1950, les avancées en statistiques et en informatique ont permis d'importantes évolutions et on peut désormais traiter d'importantes quantités de données en très peu de temps.

Parmi les techniques les plus performantes figure le boosting, une méthode d'apprentissage qui permet d'améliorer les performances prédictives, en combinant plusieurs modèles d'apprentissage. Au cœur du boosting on retrouve les arbres de décision, un modèle d'apprentissage supervisé qui se construit sur une succession de règles conditionnelles, et qui peut donner des prédictions numériques dans le cas d'un algorithme de régression, ou des prédictions non numériques dans le cas d'un algorithme de classification.

Toutefois, un arbre de décision prit individuellement peut se montrer limité en termes de capacité prédictive, on utilisera alors des méthodes ensemblistes pour combler ces faiblesses, en combinant plusieurs modèles d'apprentissage. Les méthodes ensemblistes les plus populaires sont le **bagging** et le **boosting**, et on s'intéressera particulièrement à cette seconde méthode, en étudiant son algorithme le plus connu : XGBoost, qui combine des techniques avancées de boosting tout en évitant le sur-apprentissage.

Afin de mieux comprendre cet algorithme, on s'intéressera dans un premier temps au fonctionnement des **arbres de décision**, qui sera complété par des méthodes ensemblistes, notamment les **random forest** ; avant de rentrer plus en détails sur le boosting sur lequel on effectuera une étude de cas sur **AdaBoost** ainsi que sur l'algorithme du **Gradient Boosting**. Enfin, nous travaillerons sur l'algorithme **XGBoost**, qui s'impose comme le plus performant et rapide de tous les algorithmes de boosting.

1. Arbre de décision

1.1 Fonctionnement des arbres de décision

Les **arbres de décision** sont une méthode très populaire de résolution de problèmes de classification et de régression, notamment grâce au fait qu'ils permettent de modéliser des relations complexes entre les variables d'entrée et celle de sortie, mais aussi car ils sont facilement compréhensibles. Ces arbres de décision sont construits à partir d'un ensemble de données, constitué de caractéristiques ainsi que d'une étiquette de classe. L'objectif va alors être de diviser ce jeu de données en sous-ensembles plus petits, en commençant par la sélection de la caractéristique qui va offrir la meilleure séparation. L'un des avantages principaux des arbres de décision est qu'ils sont très résistants au bruit dans les données car ils sont capables de remplacer les données manquantes par des valeurs estimées, ou d'ignorer les valeurs aberrantes.

Dans un arbre de décision, on retrouve 3 parties :

- **Le nœud racine** : premier nœud de l'arbre, qui contient l'ensemble des données utilisées.
- **Les branches** : avec une caractéristique associée à chacune d'entre elles, et qui vont donc représenter la manière dont l'ensemble des données est divisée en sous-ensembles.
- **Les nœuds terminaux (ou feuilles)** : qui vont être les derniers nœuds de l'arbre, représentent les sorties du modèle pour les valeurs d'entrées. Chacun de ces nœuds est associée à une étiquette de classe et correspond à la prédiction du modèle.

Il existe trois grandes problématiques liées à la construction d'un arbre de décision :

- **Comment bien choisir les critères associés à chaque branche ?**

Le critère de division permet de déterminer comment chaque nœud va être divisé en deux sous-ensembles distincts, l'objectif étant de maximiser les différences entre ces deux derniers. Pour cela, on peut utiliser divers critères, tels que l'indice de Gini, l'entropie ou encore l'erreur de classification.

- **Comment savoir quand nous arrêter ?**

Pour cela on utilise la règle d'arrêt, qui permet de déterminer quand arrêter la division des nœuds. Il en existe plusieurs, telles que la profondeur maximale qui limite la complexité de l'arbre, ou encore celle du nombre minimum d'observations par nœud, qui empêche les nœuds avec un faible nombre d'observations d'être à nouveau divisés. Le choix de la règle d'arrêt est important car si elle est trop stricte, le modèle peut rater des relations importantes entre les données, et si elle est au contraire trop souple, le modèle peut conduire à une mauvaise généralisation.

- **Comment interpréter les résultats ?**

Les résultats sont les étiquettes de classe prédites par l'arbre. On utilise donc une règle d'assignation qui attribue à chaque observation une étiquette de classe ou une valeur numérique. Le choix de cette règle va dépendre une nouvelle fois du type de problème (classification ou régression).

1.2 Construction d'un arbre de décision

1.2.1 Exemple de construction d'un arbre de classification

On va construire notre propre arbre de classification, avec le jeu de données suivant (on n'affiche que les 4 premières données) :

classification	espèce	gésier	nb membres	poil	carapace dorsal et ventral
Chélonien	tortue	0	4	0	1
actynoptérygien	thon	0	0	0	0
actynoptérygien	brochet	0	0	0	0
mammifère	chien	0	4	1	0

On va coder un arbre de classification linéaire sur Python, qui à partir de cette base de données va construire l'arbre de décision, en trouvant à chaque étape la meilleure séparation possible en évaluant le "coût" de la séparation. Pour cela, on peut utiliser divers critères de division, mais on utilisera ici **l'indice de Gini** afin de mesurer l'impureté de chaque nœud.

On dispose d'un échantillon de n observations indépendantes : $(X_i, Y_i), \forall i \in \{1, \dots, n\}$, Y_i est la variable à expliquer avec $Y_i \in \{0, \dots, K\}$ (K classes).

L'indice de Gini est alors donné par la formule suivante :

$I(h) = 1 - \sum_{k=1}^{K} p_{k,h}^2$, avec $p_{k,h} = \frac{N_{k,h}}{n_h}$ la proportion de la classe k dans le noeud h et $N_{k,h}$ le nombre d'observations dans le noeud h de la classe k.

On peut alors le coder comme suit :

```
def __gini__(self, dataset):
    # Calculer les indices Gini du dataset en param tre
    rows = dataset[self.target]
    class_dict = list(map(lambda x: {x: 1}, rows))
    class_dict_sum = reduce(self.__add_dict__, class_dict, {})
    if rows.empty: #On v rifie si rows est vide
        occu_class = np.array([])
    else :
        occu_class = np.fromiter(reduce(lambda x, y: {**x, **y}, map(lambda x: {x: 1},
        rows)).values(), dtype=float)
        pop = np.sum(occu_class)
        impurity = 1 - np.sum((occu_class / pop) ** 2)
    return impurity
```

On va coder une fonction qui cherchera pour quelle division les nœuds sont le plus pur possible, c'est à dire le coefficient de Gini le plus petit possible.

```
 def __find_best_split__(self, dataset):
        # Trouver la meilleure s paration de notre jeu de donn es
        df_eval = pd.DataFrame([], columns=('feature', 'value', 'nature', 'cost'))
        columns = dataset.columns[np.logical_not(dataset.columns == self.target)]
        for column in columns:
            if len(dataset[column].unique()) >= 10:
                df_eval = df_eval.append(self.__test_quanti__(dataset, column))
            elif len(dataset[column].unique()) > 1:
                df_eval = df_eval.append(self.__test_quali__(dataset, column))

        df_eval = df_eval.reset_index(drop=True)

        idx_cost_min = df_eval['cost'].idxmin(axis=0, skipna=True)

        return df_eval.iloc[idx_cost_min, :]
```

Fonction **split_evaluator** : cette fonction permet de calculer le coût d'un nœud, c'est une mesure de la pureté du nœud, donnée pour un nœud h selon la division d :

$$C(h,d) = \frac{n_g}{n} I(h_g) + \frac{n_d}{n} I(h_d)$$

avec n la population totale, n_g (respectivement n_d) la proportion de la population dans le sous-ensemble de gauche (respectivement de droite) et $I(h_g)$ l'indice de Gini à gauche.

On choisira de séparer la population initiale selon la coupure déterminée, ici par la fonction **__find_best_split__** , qui permet de déterminer pour quelle coupure le coût du nœud sera minimal :

$$\boxed{D^* = Argmin_d C(h,d)}$$

Grâce aux résultats de notre algorithme, on obtient l'arbre de classification :

```
poil == 0
--> True:
  nb membres == 2
  --> True:
    Predict oiseau
```

```
    Predict {'oiseau': 1.0}
  --> False:
    carapace dorsal et ventral == 1
    --> True:
      g sier == 0
      --> True:
        Predict Ch lonien
        Predict {'Ch lonien': 1.0}
      --> False:
        Predict crocodilien
        Predict {'crocodilien': 1.0}
    --> False:
      nb membres == 4
      --> True:
        Predict reptile
        Predict {'reptile': 1.0}
      --> False:
        Predict actynopt rygien
        Predict {'actynopt rygien': 0.4, 'insecte': 0.2, 'poisson': 0.2, 'reptile': 0.2}
--> False:
  Predict mammif re
  Predict {'mammif re': 1.0}
```

Que l'on peut représenter de manière plus visuelle :

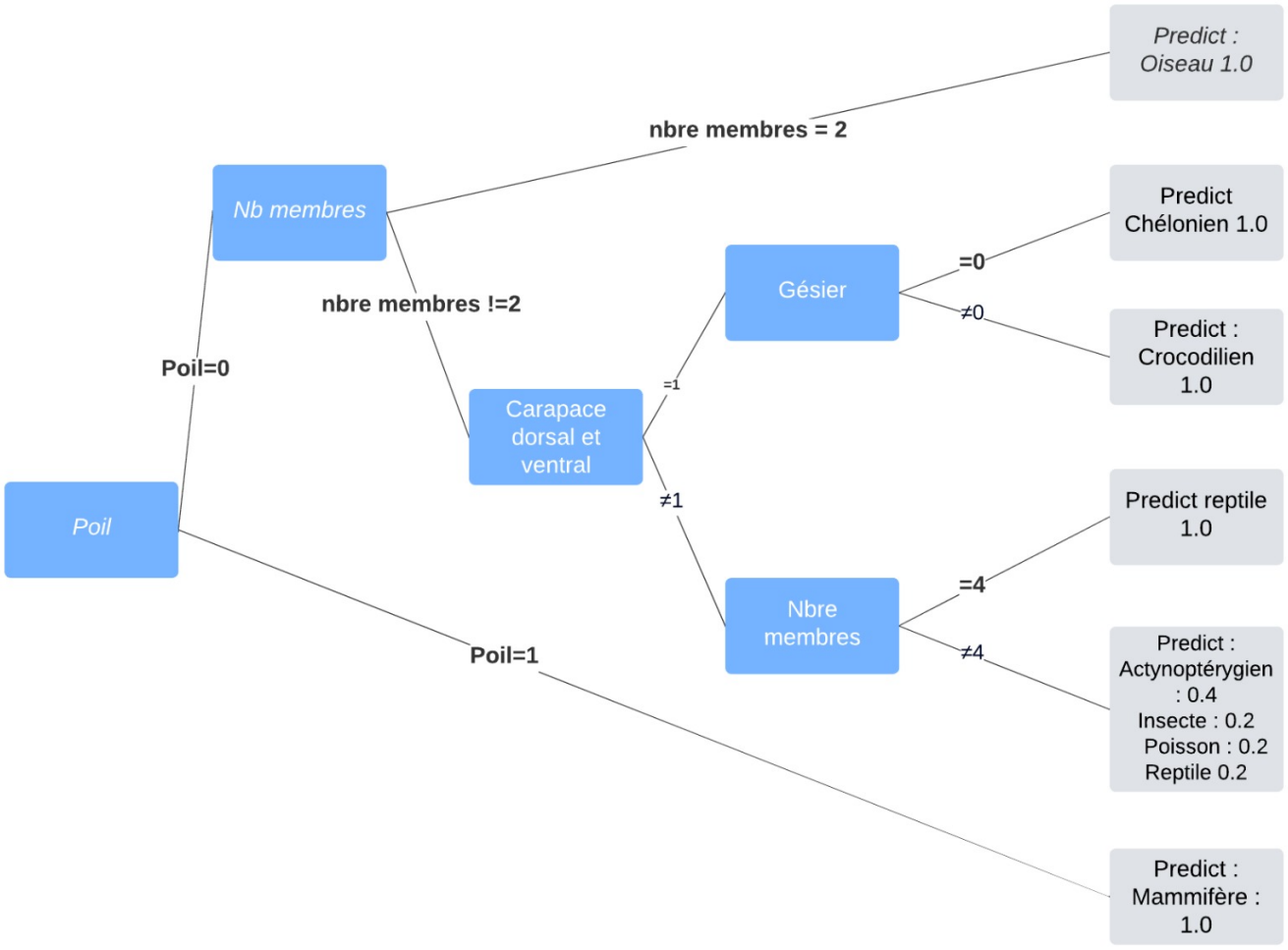


FIGURE 1.1 – Arbre de décision correspondant

1.2.2 Les arbres de régression

Dans le cas où la variable à expliquer Y est quantitative, nos feuilles contiendront des valeurs numériques et non des classes.
Critère de division : on utilisera cette fois la moyenne des carrés des erreurs plutôt que le coefficient de Gini pour chercher les divisions qui vont minimiser l'impureté, qui est donnée pour le nœud i par la formule :

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Similairement aux arbres de classification, le coût du nœud h est alors donné par la formule :

$$\boxed{J(h,d) = \frac{n_g}{n}MSE(h_g) + \frac{n_d}{n}MSE(h_d)}$$

1.3 Règle d'arrêt : comment définir la bonne taille pour un arbre ?

Les performances d'un arbre de décision vont principalement dépendre de la détermination de sa taille. Il est également clair que la taille de l'arbre augmente avec le nombre d'observations de notre jeu de données, et un arbre surdimensionné pourrait conduire à des calculs inutiles, voir des erreurs. La recherche de la taille optimale d'un arbre est donc un enjeu crucial dans sa construction, et il passe par les concepts de pré-élagage et de post-élagage.

1.3.1 Pré-élagage

Le **pré-élagage** consiste à la fixation d'une règle d'arrêt, qui arrête la construction de l'arbre. La difficulté de cette approche réside dans le fait que cette règle ne doit être ni trop restrictif sous peine d'avoir un arbre sous-dimensionné, ni trop souple sinon l'arbre pourrait être surdimensionné. On peut alors fixer certains paramètres, par exemple un nombre minimal d'observations pour qu'un nœud puisse à nouveau être divisé.

1.3.2 Post-élagage et méthode CART

Le **post-élagage** est quant à lui apparu avec la méthode **CART**, le principe est de construire dans un premier temps des arbres les plus purs possibles, en acceptant toutes les segmentations quitte à ce qu'elles ne soient pas pertinentes. On a alors la construction de l'arbre maximal, que l'on notera T_{max} , et qui correspond

à la partition la plus fine. On essaye ensuite de réduire l'arbre en utilisant un autre critère qui permet de comparer des arbres de tailles différentes, on parle de critère pénalisé. Pour cela, on va extraire une sous-suite en minimisant le critère minimalisé :

$$Crit_\alpha(T) \;=\; \gamma_n(\hat{s_T}) + \alpha.\frac{|T|}{n},\;\; \alpha \geq 0$$

Avec $|T|$ le nombre de feuilles pour l'arbre T, et γ_n l'erreur quadratique.

La phase finale consiste à l'aide par exemple d'une validation croisée (détaillée plus tard) de sélectionner le meilleur arbre de la suite construite.

1.4 Les arbres de décision : une méthode dépassée ?

Les arbres de décisions semblent être une méthode sans failles : performances comparables aux autres méthodes supervisées, résistante aux données atypiques et ne nécessite aucune hypothèse a priori sur la distribution des données. C'est également une méthode qui permet une représentation visuelle et une interprétation simple.

Cependant cette approche est **limitée** car les temps de calculs peuvent être très long pour de grandes bases de données ; mais également car les méthodes utilisées, tel que **l'algorithme CART** sont très instable : modifier légèrement les données de notre échantillon d'apprentissage peut entraîner des résultats très différents. Une autre de ses faiblesses provient de la nature de sa construction : les arbres étant construits pas-à-pas, on peut parler d'une certaine myopie vis-à-vis de la détection des combinaisons de variables.

Enfin, comme vu avec la règle d'arrêt, les arbres de décision sont sensibles à la taille de la base de données utilisée, et peuvent tomber dans le sur-apprentissage (over-fitting), c'est à dire un fonctionnement normal avec les données sur lesquels ils ont été entrainés, mais beaucoup moins avec un nouveau jeu de données.

1.5 Amélioration du modèle

Pour obtenir de meilleures prédictions et pallier les faiblesses des arbres de décision, on peut utiliser l'apprentissage ensembliste, qui utilise des algorithmes d'apprentissage, tels que le bagging, le boosting et les forêts aléatoires.

L'apprentissage ensembliste est une combinaison de plusieurs modèles d'apprentissage, dans le but d'obtenir une meilleure performance prédictive, avec l'idée que cette combinaison entraînera de meilleurs résultats que si l'on prenait chaque

modèle individuellement, et que chaque modèle peut compenser les faiblesses de l'autre.

1.5.1 Méthode bagging

Le **bagging** est un ensemble de méthodes introduit par Léo Breiman en 1996, dont le but est de réaliser un échantillonnage des données et d'entraîner l'algorithme de façon séparée sur chacun de ces échantillons, avant d'assembler les résultats obtenus.

Cette méthode a pour principaux avantages de réduire la variance, donc de combler une lacune des arbres de décision, mais aussi de corriger les erreurs de prédictions. Cependant, il est presque impossible de considérer ces estimateurs comme indépendants car ils dépendent tous du même échantillon. La méthode du bagging consiste à essayer de diminuer la dépendance entre les estimateurs en les construisant sur des échantillons **bootstrap**.

La première étape de cette méthode est de travailler les données, de telle sorte que nous allons fournir des données qui ne soient pas les mêmes pour tous nos modèles, dans un souci d'indépendance. Plutôt que de partitionner le jeu de données, on réalise un jeu de données bootstrap, ce qui consiste à la création d'un nouveau jeu de données à partir du jeu de données initial, qui seront de même taille. Ainsi, si on nomme E_1 le jeu de données initial et E_2 celui réalisé par bootstrap, de E_1 on choisit un individu aléatoire que l'on place dans E_2, et on répète cette étape jusqu'à qu'ils soient de la même taille. Il est important que ces tirages s'effectuent avec remise, c'est ce qui permet la création de données différentes.

Une fois cette partie de bootstrapping effectuée, on entraîne chacun de nos modèles séparément, en utilisant le même algorithme, puis on fait l'agrégation des données.

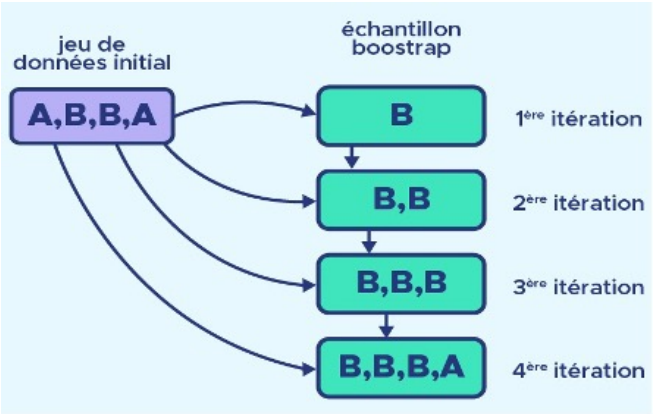


FIGURE 1.2 – Bootstrapping

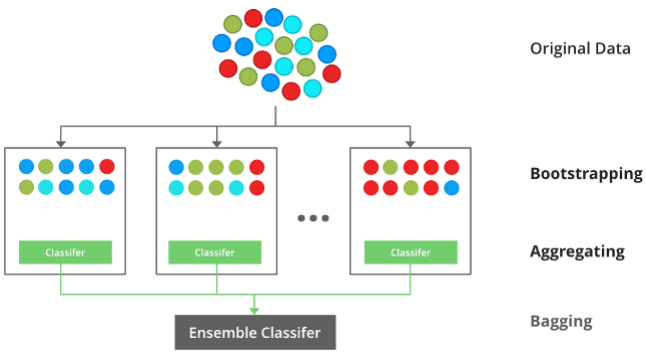


FIGURE 1.3 – Aggrégation

Détaillons mathématiquement ce qu'est le bagging.

On dispose de $L_n^1, \ldots, L_n^q$, où $L_n^i = (X_i, Y_i)_{i=1,\ldots,n}^{(l)} \forall l = 1, \ldots, q$, q échantillons d'apprentissages i.i.d de taille n.

A partir de ces q échantillons, on va construire q prédicteurs i.i.d, notés : $\hat{g}_1, \ldots, \hat{g}_q$.

Dans le cas de la régression la moyenne des valeurs prédites : $\boxed{\hat{g}_{bag}(x) = \frac{1}{q}\sum_{l=1}^{q} \hat{g}_l(x)}$,

et dans celui de la classification on applique un système de vote :

$\boxed{\hat{g}_{bag}(x) = Argmax_{k\in\{0,\ldots,K\}} \sum_{l=1}^{q} \mathbb{1}_{\hat{g}_l(x)=k}}$.

On note $m(x) = \mathrm{E}[Y|X = x]$ la fonction de régression et pour $x \in \mathbb{R}^p$, on considère l'erreur quadratique moyenne d'unn estimateur $\hat{m}$ et sa décomposition biais-variance :

$$\mathrm{E}[(\hat{m}(x) - m(x))^2] = \Big(\mathrm{E}[\hat{m}(x) - m(x)]\Big)^2 \; + \; \mathrm{Var}\,(\hat{m}(x))$$

La méthode bagging étant une méthode d'agrégation, elle consiste à agréger un nombre q d'estimateurs $\hat{m}_1, \ldots, \hat{m}_q$: $\hat{m}(x) = \frac{1}{q}\sum_{k=1}^{q} \hat{m}_k(x)$.

Remarquons que si on suppose les régresseurs $\hat{m}_1, \ldots, \hat{m}_q$ **i.i.d**, on a :

$$\mathrm{E}[\hat{m}(x)] \; = \; \mathrm{E}[\hat{m}_1(x)] \quad \text{et} \quad \mathrm{Var}(\hat{m}(x)) \; = \; \frac{1}{q}\mathrm{Var}\,(\hat{m}_1(x))$$

Le biais de l'estimateur agrégé est alors le même que celui des $\hat{m}_k$, mais avec une variance inférieure. Si on est dans le cas d'un jeu de données bootstrap, ce qui va nous intéresser ici, alors les prédicteurs $\hat{g}_1, \ldots, \hat{g}_q$ ne sont plus indépendants, et on a alors :

$$\mathrm{Var}\,[\hat{g}_{bag}(x)] \; = \; \mathrm{Var}\,[\frac{1}{q}\sum_{l=1}^{q} \hat{g}_l] \; = \; \left[\frac{1}{q^2}\sum_{l=1}^{q} \mathrm{Var}\,(\hat{g}_l) \; + \; 2\sum_{1\le i<j\le q} \mathrm{Cov}\,(X_i, Y_i)\right]$$

$$\Longleftrightarrow \; \boxed{\mathrm{Var}\,[\hat{g}_{bag}(x)] \; = \; \rho(x).\sigma^2(x) \; + \; \frac{1-\rho(x)}{q}\sigma^2(x)}$$

Avec cette formule, on constate donc que c'est le coefficient de corrélation $\rho(x)$ entre les estimateurs agrégés qui va quantifier le gain de cette procédure, c'est à

dire que la variance diminue d'autant plus que les estimateurs que l'on agrège seront décorrélés. Cependant, cela est conditionné au fait que les estimateurs soient sensibles à des perturbations de l'échantillon, car dans le cas contraire, le bagging n'apportera pas d'amélioration.

1.5.2 Random Forest

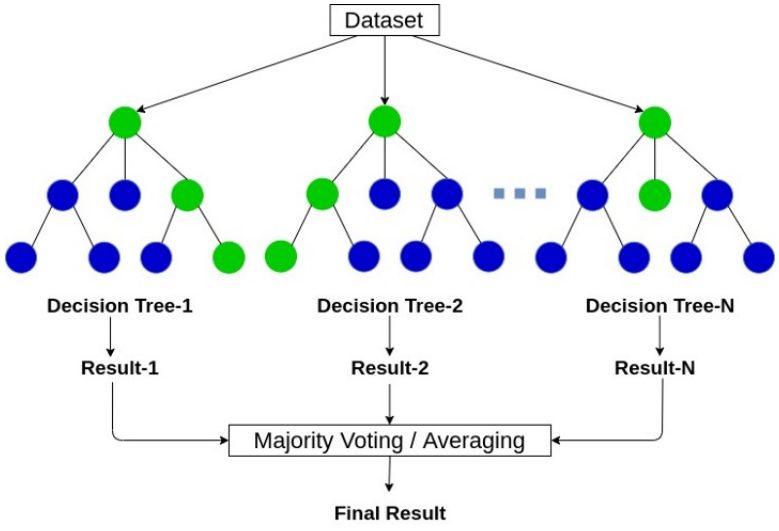
Les **forêts aléatoires** sont l'un des algorithmes les plus utilisés de la méthode bagging, ils utilisent l'algorithme CART afin de construire des arbres. L'objectif va être de diminuer la corrélation afin d'améliorer les résultats du bagging. C'est une agrégation d'arbres dépendants de variables aléatoires.

Nous allons coder cet algorithme pour mieux le comprendre :

- La première étape est de créer autant d'échantillons bootstrap que d'arbre que l'on va utiliser dans la forêt.

- On créer ensuite la forêt aléatoire à partir de ces données. Pour chaque arbre, on construit un arbre en suivant l'algorithme CART, pour chaque nœud on tire au hasard q variables parmi les p existantes dans notre échantillon, avant de sélectionner la meilleure division. En répétant ce processus un grand nombre de fois, on aura une forte variété d'arbre, ce qui fait la force de cet algorithme.

- Une fois cette étape d'apprentissage terminée, chaque arbre va définir une classe d'appartenance, puis un vote de majorité est effectué pour définir la classe prédite (si on est dans un contexte de classification) ou la valeur prédite (i.e moyenne des prédictions, dans le cas d'une régression).



Il est important de noter que puisque les tirages se font avec remise, il est possible que certaines observations ne soient pas sélectionnées, on parle alors

de données **out-of-bag**. Ces données restent toutefois importantes car elles permettent d'évaluer les performances de notre forêt aléatoire, en mesurant la proportion de ces données qui a été correctement classée.

C'est donc une procédure qui va permettre de fournir un estimateur des erreurs :

- $\mathrm{E}[(\hat{m}(X) - Y)^2]$ en régression
- $\mathbb{P}(\hat{m}(X) \neq Y)$ en classification

L'erreur out-of-bag est alors définie par :

- $\frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$ en régression
- $\frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{\hat{Y}_i \neq Y_i)}$ en classification

Comme tout algorithme, les random forest présentent aussi des faiblesses, malgré leur forte précision et leur robustesse aux valeurs aberrantes. En effet, c'est un modèle complexe de par sa nature agrégée, et peuvent donc être difficiles à interpréter dans certains cas. La suite logique est donc que les temps de calcul peuvent être très longs lorsque les jeux de données sont grands.

On va maintenant coder nous même notre propre algorithme Random Forest sur Python, à l'aide du dataset **"Iris"**, afin de visualiser ses performances.

On aboutit donc à ces résultats :

```
No. of variables tried at each split: 2

        OOB estimate of  error rate: 4.67%
Confusion matrix:
           setosa versicolor virginica class.error
setosa         50          0         0        0.00
versicolor      0         47         3        0.06
virginica       0          4        46        0.08
```

On obtient donc une matrice de prédictions, les bonnes prédictions se trouvent sur la diagonale tanndis que les erreurs sont disposées autour de la diagonnale. Il n'y a que très peu d'erreurs, avec seulement 7 observations mal prédites sur 150. Cela témoigne de la qualité prédictive de cette méthode.

2. Le Boosting en apprentissage statistique

2.1 L'apprentissage statistique

L'apprentissage statistique, aussi appelé apprentissage automatique ou Machine Learning en anglais, est une branche des domaines de l'informatique et de la statistique dont le but est de concevoir des techniques permettant aux ordinateurs d'effectuer des calculs et des prédiction à partir d'un jeu de données. L'objectif de ce procédé est d'élaborer des programmes informatiques qui peuvent s'améliorer au fil du temps de la manière la plus autonome possible, sans avoir été explicitement programmés.

L'apprentissage statistique se décompose en plusieurs catégories qui n'ont pas les mêmes objectifs : la classification, le clustering ou bien l'apprentissage par renforcement. Le domaine d'application de ces méthodes est très vaste, la reconnaissance d'image, la prédiction financière ou même la biologie et la cybersécurité.

Parmis les algorithmes les plus utilisés on peut citer la **régression linéaire**, qui consiste à estimer les paramètres d'une fonction linéaire des observations, mais aussi les réseaux de neuronnes artificiels et les arbres de décision que nous avons présenté dans le chapitre précédent.

L'ossature commune d'un problème d'apprentissage statistique ce compose le plus généralement d'une fonction objectif, souvent l'écart de prédiction que l'on doit minimiser, et d'un échantillon d'apprentissage ainsi qu'un autre destiné au test.

On peut donc formaliser le problème d'apprentissage de la manière suivante :

- **Données :** vecteurs aléatoires (x) générés suivant une densité $p(x)$
- **Fonction d'apprentissage :** $F = \{F_\theta\}_\theta$ avec θ les paramètres du modèle
- **Fonction coût :** $C_\theta(x)$ correspondant à la fonction F et les données x
- **Risque théorique :** $R_\theta = E_x[C_\theta(x)] = \int_x C_\theta(x)p(x)dx$
- **Solution optimale :** $F_{\theta^*} = Argmin_\theta R_\theta$

2.2 Principe et fonctionnement du boosting

2.2.1 Boosting et bagging

Le **boosting** en apprentissage statistique est une méthode ensembliste qui ne cesse d'évoluer permettant de résoudre des calculs toujours plus efficacement sur des jeux de données très volumineux. C'est en 1996 que le boosting voit le jour lors de la 13ème conférence internationale sur l'apprentissage automatique (ICML) où les chercheurs Robert Schapire et Yoav Freund ont introduit une nouvelle technique intitulée "Adapting Boosting". Cette innovation donnera ensuite naissance à l'algorithme **AdaBoost**, le premier programme de boosting en apprentissage statistique.

Le **bagging** en apprentissage statistique consiste à combiner un grand nombre d'algorithmes ayant des performances individuelles peu intéressantes afin d'en créer un beaucoup plus efficace. Les algorithmes faiblement performants sont appelés les "weak learners" et le résultat obtenu le "strong learner". Les weak learners peuvent avoir des stratégies et des natures différents, la seule condition est qu'ils doivent être indépendants les uns des autres.

Le boosting quant à lui diffère du bagging au niveau de la création des weak learners. En effet, il n'est plus nécessaire qu'ils soient indépendants mais au contraire, chaque weak learner est amélioré pour corriger les erreurs du précédent.

Enfin, ces deux méthodes jouent un rôle important dans le **compromis biais-variance** d'un modèle. Le bagging a tendance à réduire la variance du modèle, ce qui peut aider à surmonter le sur-apprentissage, tandis que le boosting permet plutôt de réduire son biais, car il se concentre sur les erreurs du modèle précédent, en augmentant le poids des données mal classées.

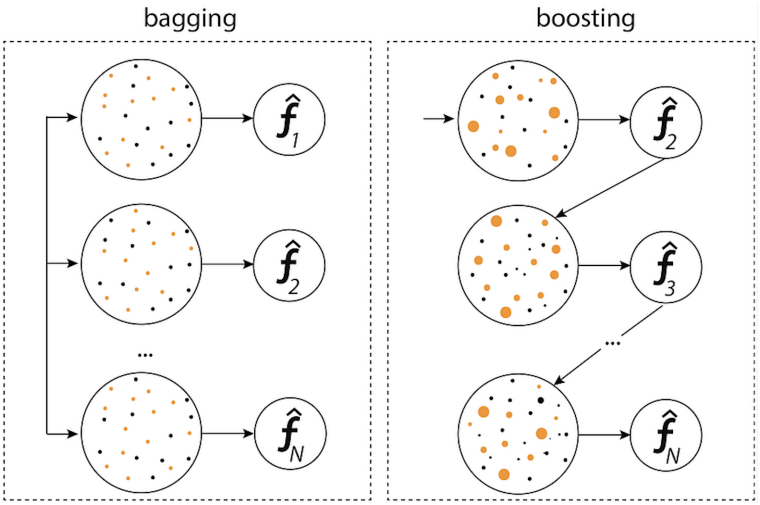


FIGURE 2.1 – Baggig versus Boosting

2.2.2 AdaBoost

Le principe général de l'algorithme **AdaBoost** est de construire des estimateurs de manière récursive, c'est à dire que chaque estimateur est une version adaptative du précédent en donnant un poids plus important aux observations mal ajustées ou dont la prédiction n'est pas assez précise. L'estimateur construit à l'étape n va donc se focaliser sur les observations mal estimées par l'estimateur de l'étape $n-1$.

L'algorithme AdaBoost s'exécute de la manière suivante :

Entrée :

- Un échantillon d'observations $(x_1, y_1), ..., (x_n, y_n)$.
- Une règle faible
- Un nombre d'itérations M

Initialisation : On définit des poids uniformes :

$$w_i = \frac{1}{n}, \quad \forall i = 1, ..., n \quad \text{tels que} \quad \sum_{i=1}^{n} w_i = 1.$$

Itération : Pour m allant de 1 à M :

1) Ajuster un classifieur faible $g_m(x)$ sur l'échantillon pondéré par les poids w_i
2) Calculer le taux d'erreur :

$$\varepsilon_m = \frac{\sum_{i=1}^{n} w_i \mathbb{1}_{y_i \neq g_m(x_i)}}{\sum_{i=1}^{n} w_i}$$

3) Calculer le poids de l'itération m : $\alpha_m = log\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right)$
4) Réajuster les poids des observations : $w_i = \frac{1}{Z_m} w_i \exp\left[\alpha_m \mathbb{1}_{y_i \neq g_m(x_i)}\right]$

Sortie : On affiche le signe de la combinaison linéaire : $\hat{g}_M(x) = signe\left(\sum_{m=1}^{M} \alpha_m g_m(x)\right)$

Remarques :

- Les α_m permettent de mesurer le poids de l'itération m dans le classifieur final. C'est à dire à quel point cette itération est importante, autrement dit cela permet de repérer le classifieur faible ayant fait le plus d'erreur.

- Lors de l'étape 4, le signe du α_m va influer sur la sortie de l'algorithme :
  - α_m **est positif :** l'échantillon est bien ordonné et dans ce cas, nous allons diminuer le poids de l'échantillon m par rapport à son poids précédent dans le classifieur final. Nous avons en effet déjà de bonnes performances.
  - α_m **est négatif :** l'échantillon n'est pas bien ordonné, il est nécessaire d'augmenter son poids afin que la même erreur ne se reproduise plus à l'itération suivante.
- Toujours lors de l'étape 4, la variable Z_m est une constante de normalisation permmettant d'assurer que la somme des poids w_i reste égale à 1 et vaut :

$$\boxed{Z_m = 2\sqrt{(1-\varepsilon_m)\varepsilon_m}}$$

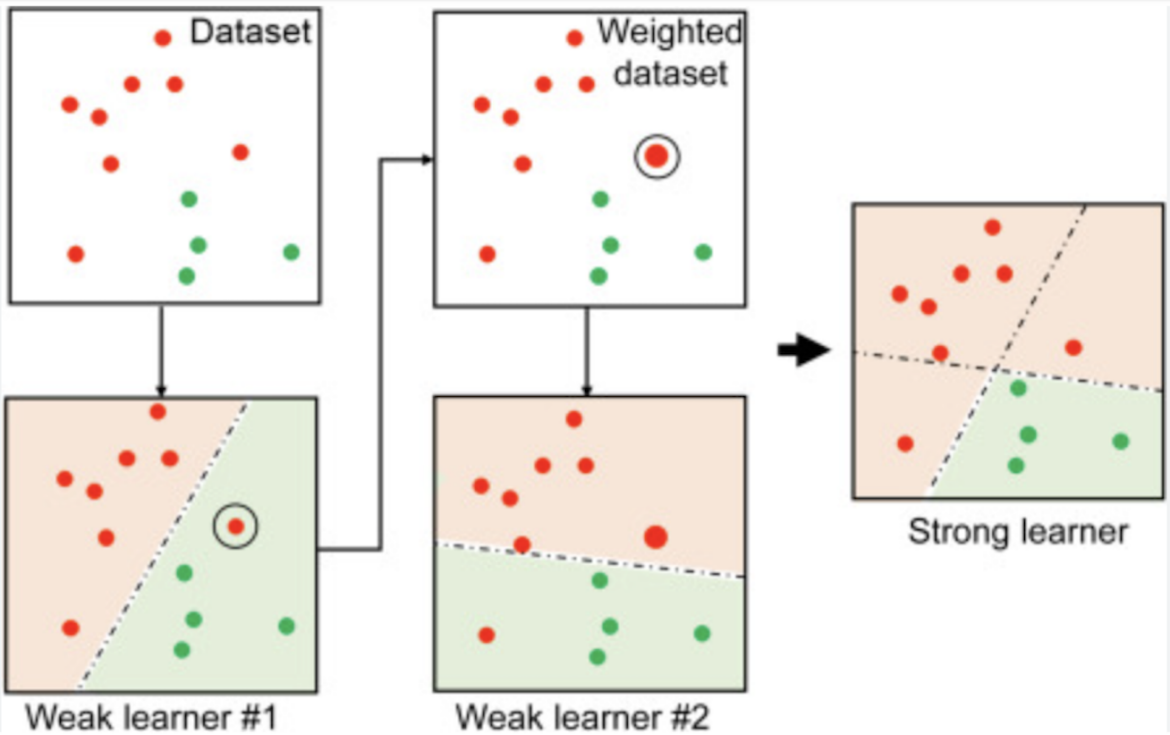


FIGURE 2.2 – Fonctionnement d'AdaBoost

Ce schéma représente bien la concaténation de plusieurs classifieurs faibles afin d'aboutir à un classifieur fort. En effet, lors des deux itérations, l'algorithme AdaBoost sépare de deux manières différentes le jeux de données et reprère ses erreurs. Enfin, nous remarquons que le classfieur fort est bien une **combinaison** des deux classifieurs faibles précédents.

2.2.3 Mise en pratique d'AdaBoost sur un problème de classification

Dans cette section, nous allons **tester** et **examiner** le fonctionnement de l'algorithme AdaBoost sur des données simulées numériquement.

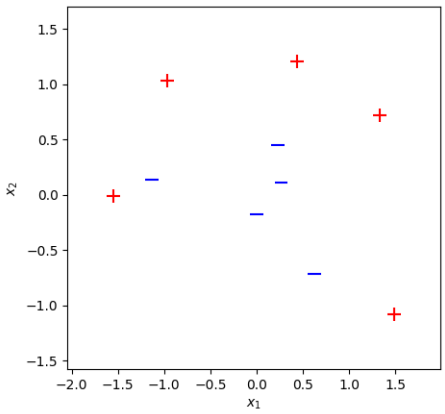
On introduit la fonction $plot\_adaboost()$ qui va nous permettre de visualiser nos données et de superposer la limite de décision des classifieurs faibles au fur et à mesure des itérations. Cette fonction est très complexe mais n'est pas essentielle à la compréhension de notre démonstration, on ne la détaillera pas ici.

On commence par générer un jeu de données à l'aide de la librairie **Sklearn**. On utilise la fonction $make\_gaussian\_quantiles$ afin de créer une base de données synthétique basée sur une distribution gaussienne multi-dimentionnelle. Cette fonction est particulièrement intéressante dans notre cas car on souhaite éviter le cas où les deux classes seraient linéairement séparables. Cela ne serait pas un exemple très pertinent quant au fonctionnement de l'algorithme. On définit donc la fonction suivante :

```
def make_dataset(n: int = 100, random_seed: int = None):
    n_per_class = int(n/2)
    if random_seed:
        np.random.seed(random_seed)
    X, y = make_gaussian_quantiles(n_samples=n, n_features=2,
    n_classes=2)
    return X, y*2-1

X, y = make_dataset(n=10, random_seed=10)
plot_adaboost(X, y)
```

Ici, on génère un échantillon de n observations (100 par défaut), réparties équitablement en deux classes distinctes, chaque observations possédant deux features. Si la valeur de $random\_seed$ est fournie, la fonction $np.random.seed$ est appelée pour initialiser le générateur de nombres aléatoires avec la graine spécifiée. Cela garantit la reproductibilité des échantillons générés si la même graine est utilisée. X contient les échantillons générés et y leurs étiquettes. L'expression $y*2-1$ transforme les étiquettes de classe de 0 et 1 en -1 et 1, respectivement.

On affiche ensuite nos données :



Une fois cette étape réalisée, nous allons implémenter manuellement l'algorithme AdaBoost afin de séparer notre jeu de données le plus rapidement et efficacement possible avec la technique du boosting pondéré. On commence par initialiser le modèle en créant une classe "AdaBoost" :

```
class AdaBoost:
    def __init__(self):
        self.stumps = None
        self.stump_weights = None
        self.errors = None
        self.sample_weights = None

    def _check_X_y(self, X, y):
        assert set(y) == {-1, 1}
        return X, y
```

Le fait de définir une classe permet d'encapsuler les données et les fonctionnalités associées en une seule entité, plus facile à manipuler et à réutilisée.

La classe "AdaBoost" contient les attributs suivants :
- **Stumps** : une liste contenant les weak learners
- **stump_weights** : une liste contenant leurs poids
- **errors** : une liste contenant leurs erreurs de classification
- **sample_weights** : une liste contenant les poids des échantillons utilisés lors de l'entraînement de chaque classifieur faible

Le constructeur de la classe est défini par la méthode "__init__()". Il initialise tous les attributs de la classe à "None". La méthode "_check_X_y()" est définie pour valider le format des données d'entrée.

Une fois ces éléments définis, on implémente manuellement l'algorithme :

```
def fit(self, X: np.ndarray, y: np.ndarray, iters: int):
    X, y = self._check_X_y(X, y)
    n = X.shape[0]
```

```
# Initialiser attributs
self.sample_weights = np.zeros(shape=(iters, n))
self.stumps = np.zeros(shape=iters, dtype=object)
self.stump_weights = np.zeros(shape=iters)
self.errors = np.zeros(shape=iters)

# Initialiser poids uniformment
self.sample_weights[0] = np.ones(shape=n) / n

for t in range(iters):
    # Weak learner
    curr_sample_weights = self.sample_weights[t]
    stump = DecisionTreeClassifier(max_depth=1, max_leaf_nodes=2)
    stump = stump.fit(X, y, sample_weight=curr_sample_weights)

    # Error et Stump_weights
    stump_pred = stump.predict(X)
    err = curr_sample_weights[(stump_pred != y)].sum()# / n
    stump_weight = np.log((1 - err) / err) / 2

    # Actualisation sample_weights
    new_sample_weights = (
        curr_sample_weights * np.exp(-stump_weight * y * stump_pred)
    )

    new_sample_weights /= new_sample_weights.sum()

    # Si it ration non finale
    if t+1 < iters:
        self.sample_weights[t+1] = new_sample_weights

    # R sultats
    self.stumps[t] = stump
    self.stump_weights[t] = stump_weight
    self.errors[t] = err

return self
```

Ce code peut paraître un peu complexe mais il reprend toutes les étapes de l'algorithme d'AdaBoost.

1. Après avoir vérifié le formatage des données, on initialise tous les attributs de la classe "AdaBoost".

2. On définit ensuite les premiers poids de chaque classifieur équiatblement.

3. On créer une boucle for dans laquelle on commence par donner un premier weak learner à l'algorithme, ici on suppose que les classifieurs sont des arbres de décisions, des **Stumps**, des arbres à deux modalités et de profondeur 1. Notre weak learner doit minimiser :

$$h_t(x)\varepsilon_t = \sum_{i=1}^{n} \mathbb{1}_{[h_t(x_i)\neq y_i]} w_i^t$$

4. On fixe un poids pour notre apprenant faible en fonction de sa précision : $\alpha_t = \frac{1}{2} ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$.

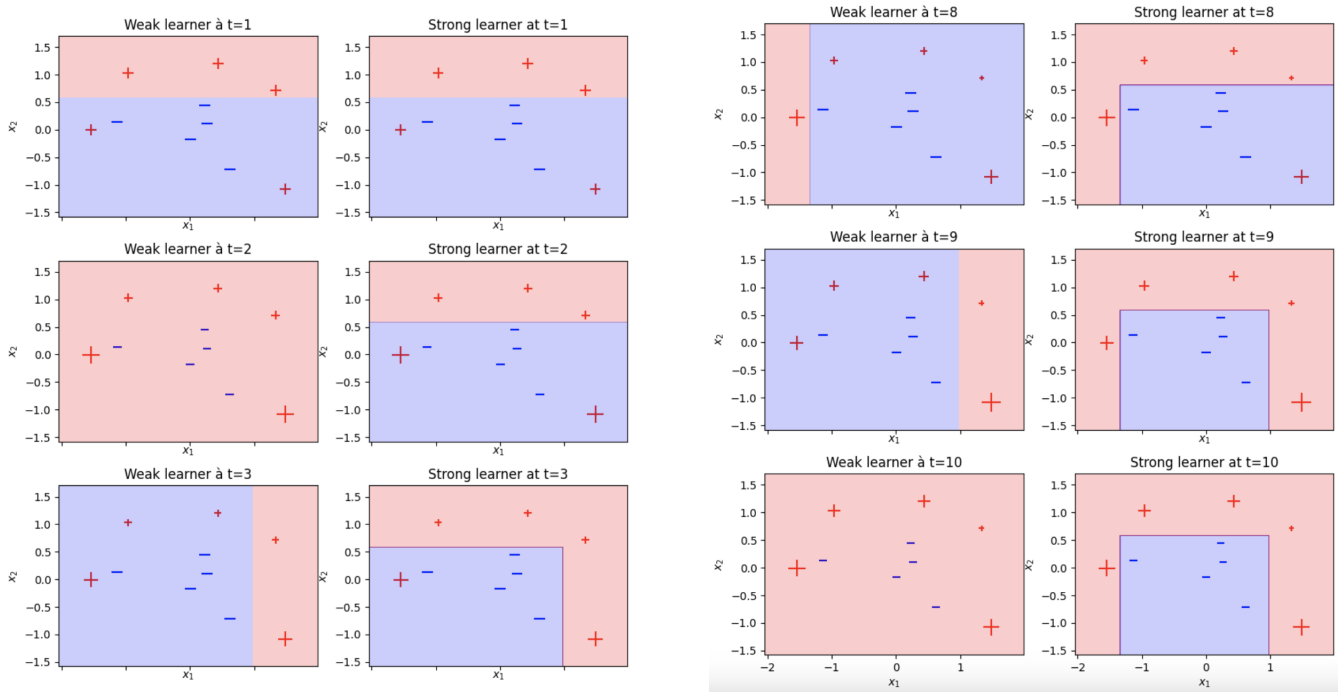
5. On augmente les pondérations des observations mal classées : $w_i^{(t+1)} = w_i^{(t)}.e^{-\alpha_t y_i h_t(x_i)}$.

6. Renormaliser les poids, de sorte que : $\sum_{i=1}^{n} w_i^{(t+1)} = 1$.

On va ensuite arriver à la fin des itérations et afficher la prédiction finale, c'est à dire l'aggrégation de toutes les prédictions des weak learners. On calcule la combinaison linéaire de la prédiction de chaque souche et de son poids de souche correspondant : $H_t(x) = sign\left(\sum_{t=1}^{T} a_t h_t(x)\right)$.

```
def predict(self, X):
    stump_preds = np.array([stump.predict(X) for stump in self.stumps])
    return np.sign(np.dot(self.stump_weights, stump_preds))
```

Afin de valider notre modèle, on va visualiser itérations après itérations l'évolution et l'apprentissage d'AdaBoost (à l'aide de fonctions d'affichage complexes) :



On remarque que notre algorithme a très bien fonctionné en séparant clairement les deux classes du dataset. On aurait même pu s'arrêter à l'itération 9 pour économiser des calculs, cela aurait pu être effectué par une recherche des hyperparamètres optimaux du modèle, telle que **"GridSearch"**.

Enfin, il y a des classifieurs faibles sans frontière pour certaines itérations. Par exemple, à l'itération 2, tous les points sont classés comme positifs, cela se produit parce que, compte tenu des poids d'échantillon actuels, l'erreur la plus faible est obtenue en prédisant simplement que tous les points de données sont positifs. Il n'y a aucun moyen de tracer une limite de décision linéaire pour classer correctement n'importe quel nombre de points sans faire au moins une erreur mais cela n'empêche pas l'algorithme de converger.

2.3 L'algorithme de Gradient Boosting

2.3.1 L'algorithme de descente de gradient

Maximiser ou minimiser une fonction est un des problèmes les plus répandus dans une multitude de domaines mathématiques et informatiques. Dans la pratique, il est commun de commencer par dériver la fonction objectif puis de trouver les valeurs l'annulant, tout en s'assurant du signe de la dérivée seconde. Ce processus est fiable mais lorsque la dimension devient trop importante, les solutions de l'équation $\nabla f(x) = 0$, qui jouent le rôle de paramètres, ne sont plus calculables dans un temps raisonnalble. Pour palier à ce problème, nous avons à notre disposition la descente de gradient.

L'algorithme de descente de gradient est un algorithme itératif permettant de minimiser une fonction convexe.

L'algorithme se dessine ainsi :

1. **Initialiser** avec x_0 (au hasard)
2. **Répéter** : $x_{t+1} = x_t - \alpha \nabla f(x_t)$
3. **Continuer** jusqu'à convergence vers x_t^*

Le paramètre α s'appelle le taux d'apprentissage ou learning rate en anglais. Sa valeur est primoridiale car c'est elle qui va indiquer à quelle vitesse l'algorithme va se dérouler. Un learning rate trop petit et la convergence vers la solution optimale sera trop longue tandis qu'un learning rate trop grand va créer des oscilations qui vont empêcher de tomber sur la solution minimisante. Un learning rate de **0.01** permet dans la plupart des cas un bon compromis.

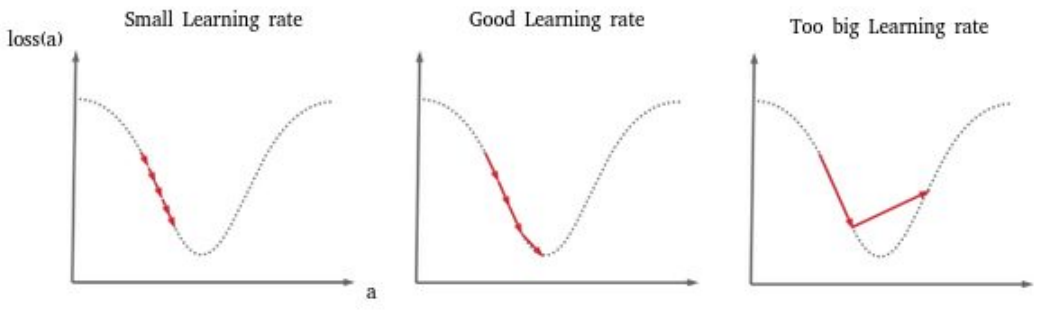


FIGURE 2.3 – Les différents taux d'apprentissage

Il est important de noter que cette algorithme doit être exclusivement utilisé dans le but de minimiser une fonction strictement convexe. En effet, dans le cas contraire, la descente de gradient risquerait de s'arrêter au niveau d'un minimum local et non global, ce qui rendrait inexacte notre minimisation.

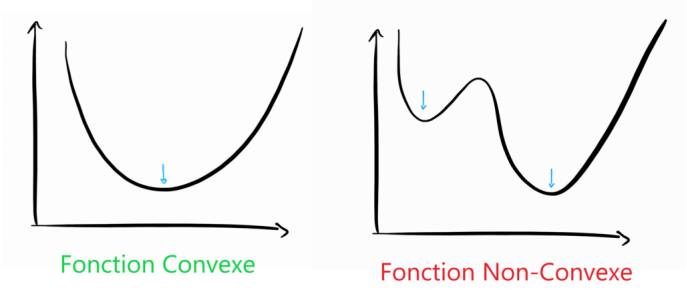


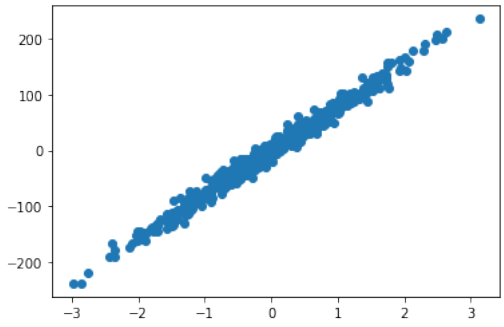
FIGURE 2.4 – Graphe d'une fonction convexe contre non-convexe

- Tout d'abord, on se place sur un point de la fonction de coût de manière totalement aléatoire. L'algorithme consiste ensuite à calculer la dérivée de la fonction de coût pour chacune de ses composantes, c'est à dire son gradient. Cela a pour but de trouver en quelque sorte la direction dans laquelle la fonction de coût décroit le plus rapidement.
- Grâce à cette étape nous allons progresser vers la solution optimale d'un pas déterminé par le learning rate (α). Nous avons atteint un nouveau point sur la fonction de coût plus proche du minimum global que nous voulons atteindre. En itérant cette étape autant de fois que nécéssaire, nous allons forcément tomber sur la solution optimale.
- Une fois le minimum atteint, nous n'avons qu'à lire la valeur des paramètres qui composent la fonction de coût et nous aurons trouver les paramètres idéaux du modèle.

Nous allons démontrer l'utilisation de cet algorithme via un petit exemple simple de régression linéaire en Python.

On commence par créer une base de données à l'aide de la fonction **make regression** appartenant à l'environnement *Sklearn* qui permet de générer un dataset, ici de 500 observations, propice à une régression linéaire, c'est à dire dont le nuage de point reflète une relation linéaire entre la variable explicative, x et la varibale à expliquer, ici y.

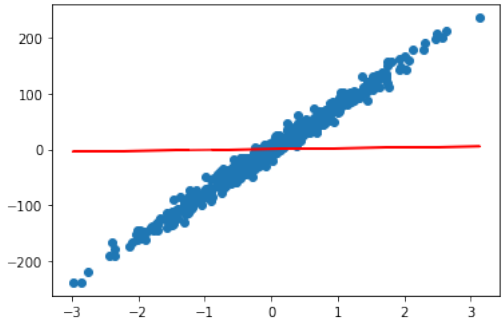
```
x,y = make_regression(n_samples=500, n_features=1, noise=10)
plt.scatter(x,y)
```



On va ensuite générer notre matrice de design X afin de pouvoir créer notre propore modèle afin d'effectuer des prédictions.

```
X = np.hstack((x,np.ones(x.shape)))
theta = np.random.randn(2,1)

def model(x,theta):
    return X.dot(theta)
```

On part d'un vecteur de paramètre θ complètement aléatoire. On créer ensuite notre modèle à partir de ce vecteur de paramètre. Pour l'instant, notre modèle ne modélise pas du tout notre jeu de données, comme le prouve le graphique qui suit, c'est n'est qu'une question de temps puisque nous allons effectuer une descente de gradient.



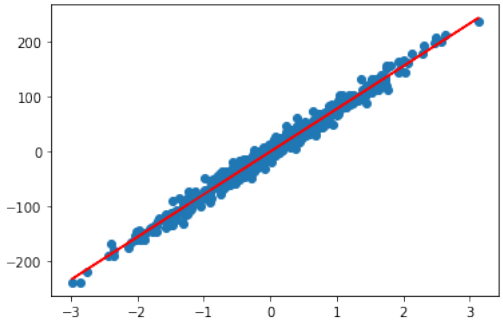
Nous allons maintenant coder notre fonnction de coût qui correspond à l'erreur quadratique moyenne afin de la minimiser par l'algorithme de la descente de gradient :

```
def fonction_cout(X,y,theta):
    n = len(y)
    return 1/(2*n)*np.sum((model(X,theta)-y)**2)

def grad(X,y,theta):
    n = len(y)
```

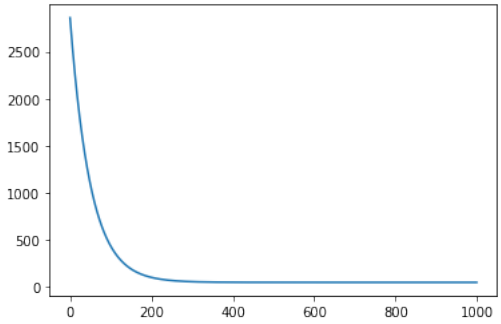
```
    return (1/n)*X.T.dot(model(X,theta)-y)

def descente_grad(X,y,theta,alpha,nb_iterations):
    cost=np.zeros(nb_iterations)
    for i in range(0,nb_iterations):
        theta = theta - alpha*grad(X,y,theta)
        cost[i]=fonction_cout(X,y,theta)
    return theta, cost
```

Ces trois fonctions permettent respectivement de calculer la fonction de coût appliquée à notre jeu de données, son gradient et l'algorithme de descente de gradient avec la formule de récurrence présentée plus tôt. On teste ensuite notre algorithme en affichant la valeur des paramètres après les 1000 itérations :

```
theta_final, cost = descente_grad(X,y,theta,alpha=0.01,
nb_iterations=1000)
```



On remarque tout de suite que notre modèle correspond beaucoup mieux aux données, les paramètres du vecteurs θ ont donc bien été choisis correspondants à la valeur minimale de la fonction de coût. On a donc un modèle bien plus précis qui possède une erreur quadratique moyenne nettement inférieure et toutà fait acceptable (passant de 2915 à 50!).



En traçant la courbe d'apprentissage, on remarque que la valeur minimale de la fonction coût est atteinte au bout de 300 itérations. Nous aurions pu nous arrêter à 300 itérations afin d'économiser du temps et du travail à l'ordinateur.

Enfin nous finnissons notre exemple en calculant le R^2 du modèle afin de vérifier qu'il explique bien nos données :

```
def R2(y,pred):
    a = ((y-pred)**2).sum()
    b = ((y-y.mean())**2).sum()
    return 1-a/b
```

Nous aboutissons à un R^2 de **0.98** qui est très proche de 1, par conséquent, la descente de gradient a bien permis de trouver les paramètres optimaux du modèle de regréssion linéaire en minimisant notre fonction de coût.

2.3.2 Fonctionnement du Gradient Boosting

L'algorithme du Gradient Boosting, comme AdaBoost, est un algorithme itératif combinant un ensemble de modèles simples (weak learners) afin de générer un classifieur plus puissant (strong learner). Les classifieurs faibles, c'est à dire des modèles ayant un faible pouvoir prédictif, seront dorénavant appelés des arbres, dont le fonctionnement sera détaillé dans la partie suivante. L'implémentation de l'algorithme de Gradient Boosting peut s'appliquer suivant différentes techniques telles que XGBoost, LightGBM et CatBoost qui ont toutes les trois des objectifs bien différents.

En utilisant cet algorithme, on travaille avec des weak learners durant chaque étape, c'est pourquoi les arbres n'auront jamais beaucoup de feuilles. Cela est plutôt avantageux car les weak learners permettent de **réduire la compléxité du modèle**, d'accélérer le processus et de se focaliser sur les **erreurs résiduelles**. La feuille d'un arbre de prédiction correspond aux nœuds finaux de l'arbre qui comportent les valeurs ou les classes prédites. Le Gradient Boosting va ensuite créer de nouveux arbres les uns après les autres, en se basant sur l'erreur de chaque nouvel arbre généré, jusqu'à atteindre le nombre d'arbre souhaité (M). Afin d'éviter le phénomène de sur-apprentissage, on va choisir un learning rate compris entre 0 et 1 afin de contrôler au mieux la contribution de chaque nouvel arbre. Notons que l'algorithme peut s'arrêter avant que le nombre d'arbres indiqué en hyper paramètre ne soit atteint, c'est par exemple le cas si l'ajout d'un nouvel arbre n'apporte plus assez de performance significative sur la réduction de l'erreur de prédiction.

L'algorithme de Gradient Boosting s'implémente de la manière suivante :

Algorithm 1 Algorithme Gradient Boosting

Require: Considérons n observations d'entraînement $(X_i, Y_i) \forall i = 1, \ldots, n$ et une fonction de perte différentiable $L(Y_i, F(X_i))$
Initialisation $F_0(x) = 0$
for $m = 1, \ldots, M$ **do**
Calculer $r_{im} = -\left[\frac{\partial L(Y_i, F(X_i))}{\partial F(X_i)}\right]_{F(X)=F_{m-1}(X)}$ pour $i = 1, \ldots, n$
Ajuster un arbre de décision faible sur $(X_1, r_{1m}), \ldots, (X_n, r_{nm})$, noter R_{mj} les feuilles pour $j = 1, \ldots, J_m$
for $j = 1, \ldots, J_m$ **do**
$\gamma_{jm} = \arg\min_\gamma \sum_{X_i \in R_{ij}} L(Y_i, F_{m-1}(X_i) + \gamma)$
end for
$F_m(X) = F_{m-1}(X) + \nu \sum_{j=1}^{J_m} \gamma_{jm}$
end for
return $F_M(X)$

Remarques :

- $F(x)$ correspond à la valeur prédictive.
- Les R_{mj} désigne la j-ème feuille de l'arbre de décision m.
- Les r_{im} sont appelés les pseudos résidus, ils servent de s'approcher le plus précisément de la solution optimale.
- Les γ_{jm} permettent de calculer les valeurs de chaque feuilles, dans le cadre d'une régression, la valeur sera la moyenne des résidus de l'échantillon contenus dans celle-ci.
- ν désigne le learning rate compris entre 0 et 1. Plus il se rapproche de 0, plus l'algorithme est lent mais évite le sur-apprentissage et le nombre d'itérations augmente. Dans AdaBoost, ν prend la valeur 1.

L'algorithme de Gradient Boosting peut se résumer ainsi :

- On cherhce à minimiser une fonction convexe $g : \mathbb{R} \longrightarrow \mathbb{R}$
- On fixe un learning rate $\lambda > 0$
- On utilise la relation de récurrence suivante :

$$x_n = x_{n-1} - \lambda g'(x_{n-1})$$

Cependant, ici notre but est de trouver un gradient fonctionnel, c'est à dire que l'on souhaite trouver une fonction et non plus un point.

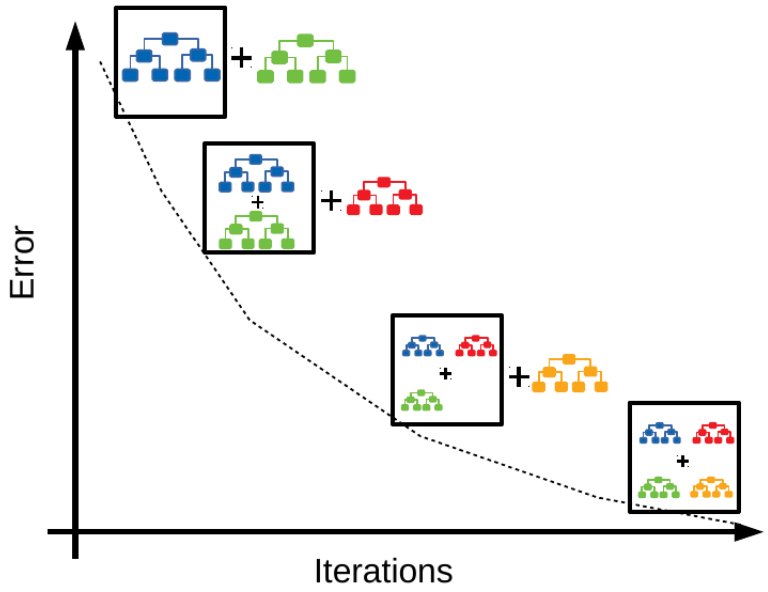
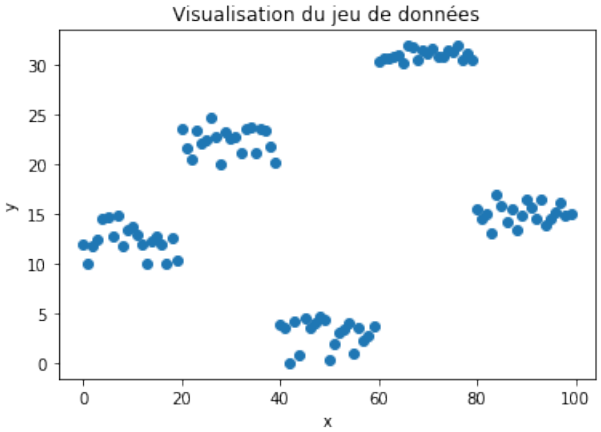


FIGURE 2.5 – Fonctionnement du Gradient Boosting

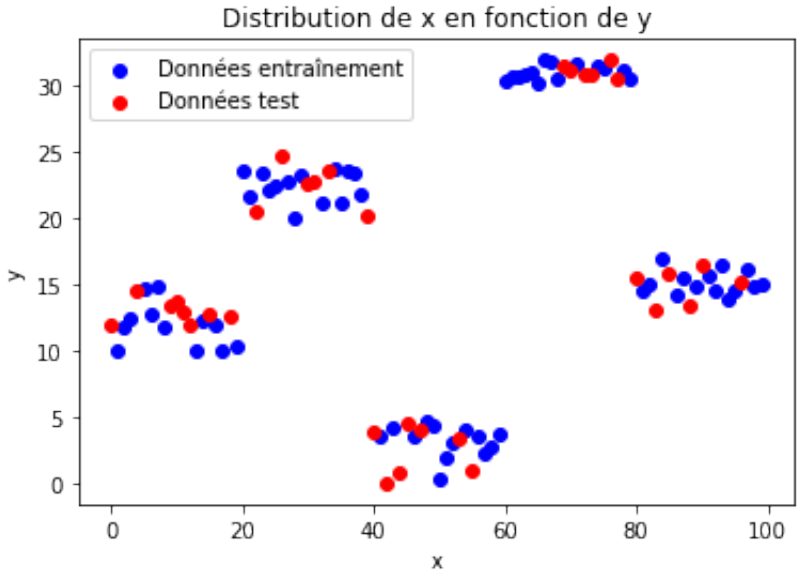
Nous allons maintenant observer un exemple simple d'utilisation de l'algorithme de Gradient Boosting, à l'aide du logiciel Python :

On commence par simuler un jeu de données, un vecteur X composé des entiers naturels entre 0 et 99 et un vecteur Y composé de 5 distributions d'une loi uniforme différente également de taille 100.

```
x = np.arange(0,100)
x = pd.DataFrame({'x':x})

y1 = np.random.uniform(10,15,20)
y2 = np.random.uniform(20,25,20)
y3 = np.random.uniform(0,5,20)
y4 = np.random.uniform(30,32,20)
y5 = np.random.uniform(13,17,20)
x['y'] = np.concatenate((y1,y2,y3,y4,y5))
```



On va maintenant séparer nos données en deux ensembles distincts, l'échantillon d'apprentissage et celui destiné aux tests. On accorde 33% des données pour l'échantillon test, on observe donc le graphique suivant :



Notre objectif est de **minimiser la fonction de coût** $f(\tilde{y}) = \sum_{i=0}^{100} (y_i - \tilde{y}_i)^2$, avec $\tilde{y}$ la prédiction de l'observation y, il s'agit donc de minimiser l'écart de prédiction élevé au carré. Pour cela, on va effectuer quelques itérations de l'algorithme de descente de gradient afin de mieux comprendre son comportement. On débute donc en créant un classifieur faible de départ, c'est à dire un arbre de décision peu profond.

```
tree1 = DecisionTreeRegressor(random_state=1234,max_depth=3)
model1 = tree1.fit(x_train,y_train)
y_pred1 = model1.predict(x_train)

errors = y_train - y_pred1
print(sum(abs(errors)**2)) -> 70.739
```

On a donc générer un arbre de décision que l'on a utilisé pour créer le modèle 1 qui renvoie les premières prédictions y_pred1. On calcul ensuite l'écart de cette prédiction avec nos données d'entraînement. Notre fonction objectif a pour valeur à cet instant : 70.739. On va maintenant se servir de ces termes d'erreurs afin de générer un nouvel arbre de décision qui va donc prendre en compte ces erreurs en essayant de les corriger un maximum.

```
tree2 = DecisionTreeRegressor(random_state=1234,max_depth=3)
model2 = tree2.fit(x_train,errors)
```

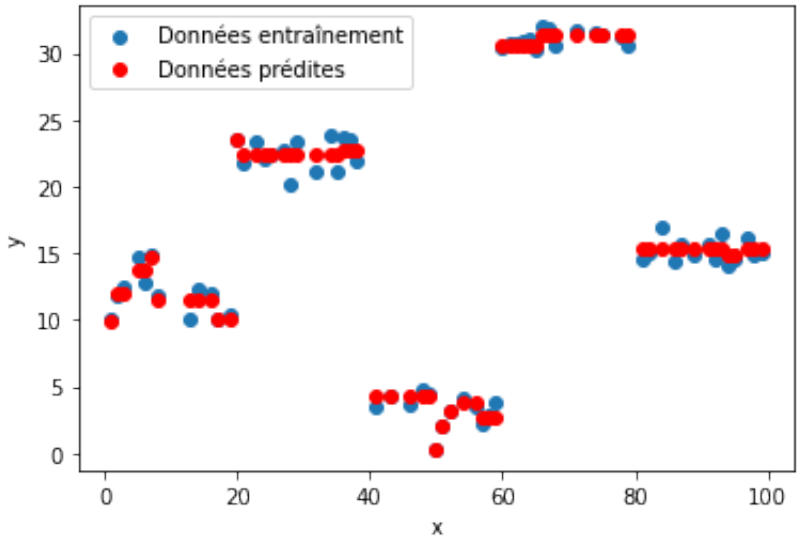
```
y_pred2 = model2.predict(x_train)

errors2 = errors - y_pred2
print(sum(abs(errors2)**2)) -> 44.095
```

On obtient donc un nouveau vecteur de prédictions qui ajuste mieux nos données d'entraînement. En effet, la somme des carrées résiduels a diminué jusqu'à 44.095! On va ensuite réitérer ces étapes les unes après les autres jusqu'à ce que notre fonction objectif ne diminue plus significativement. On se sert des erreurs comises aux étapes antérieures afin de générer d'autres classifieurs plus performants : c'est exactement le principe du **Boosting**!

On effectue une itérations de plus et on obtient une valeur de 33.802. On s'arrête ici pour nnotre exemple, on va donc générer notre classifieur final en sommant les trois classifieurs faibles précédents.

```
y_pred = y_pred1+y_pred2+y_pred3
```

En affichant nos prédictions par rapport à nos données d'entraînement, on voit que l'écart n'est pas si important et que notre modèle s'est finalement plutôt bien débrouillé en seulement trois itérations. On remarque ici tout l'intérêt du Boosting qui permet un gain de temps et une efficacité extrêmement significative.



Nous pouvons aller plus loin dans notre exemple en implémentant entièrement l'algorithme de Gradient Boosting afin d'effectuer un nombre important d'itérations. On va donc coder un programme qui reprend toutes les étapes de l'algorithme présenté à la page 16 (on ne donnera ici que les extraits les plus pertinants du code afin de ne pas surcharger le document).

```
def fit(self,x,y):
    self._model[0] = np.mean(y)
    # Calcul des résidus
    self._residual = y - self._model[0]
```

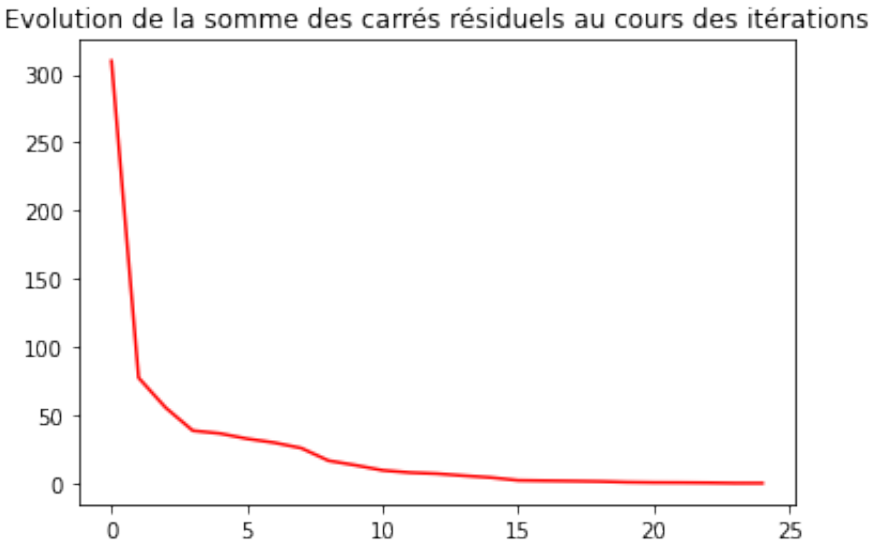
On initialise notre fonction en calculant la valeur au niveau de la feuille du premier arbre de décision, en cas d'une régression, on calcule la moyenne des observations.

```
        # Entraînement des arbres de la forêt
    for i in range(1, self._n_estimators+1) :
        # Initialiser le modèle d'arbre de décision
        estimator = DecisionTreeRegressor(max_depth= ...)

        # Entraîner le modèle sur les résidus
        self._model[i] = estimator.fit(x, self._residual)

        # Calculer la prédiction de la forêt actuel
        pred_tree = self._model[0] +
        self._learning_rate * np.sum([ self._model[j].predict
        (x) for j in range(1, i+1)], axis=0)

        # Calculer les nouveaux résidus
        self._residual =  y - pred_tree
```

On teste ensuite ce programme avec 25 classfieurs faibles et un learning rate de 0.8 :

```
gbt = GBTRegressor(n_estimators=25, random_state=123,
learning_rate=0.8, max_depth=3)
```

On obtient une somme des carrés résiduels de **0.237**!



On a tracé ici son évolution au cours des 25 itérations de l'algorithme, on remarque que 15 itérations auraient été suffisantes puisque les carrés résiduels ne diminuent plus significativement après 15 itérations.

2.4 Limites du Gradient Boosting

L'algorithme du Gradient Boosting présente plusieurs **limites** rendant parfois son utilisation moins pertinente. En effet, il s'agit d'un algorithme pouvant être très **coûteux en calcul**, c'est notamment le cas lorsque l'on utilise un nombre trop important d'arbre de décision. Cela peut rendre les calculs trop longs et ainsi surcharger la mémoire de la machine. De plus, à chaque nouvelle itération, le boosting du gradient a pour objectif de minimiser l'intégralité des erreurs du modèle. Ce processus peut parfois mettre l'accent sur des **points aberrants**, faussant alors davantage les prédictions. Enfin, c'est un algorithme qui nécessite une grande quantité d'**hyper-paramètres**, tels que le learning rate, la profondeur des arbres, le nombre d'itérations ..., il est donc important de bien les déterminer afin de ne pas trop impacter négativement les résultats. Dans la troisième partie, nous introduirons l'algorithme XGBoost, une méthode bien plus rapide et performante, qui est devenue petit à petit une référence dans le cadre du Boosting en apprentissage statistique.

3. Algorithme XGBoost

L'algorithme XGBoost a été inventé en 2014 par le chinois **Tianqi Chen** à l'Université de Washington lors d'un projet de recherche pour la Deep Machine Learning Community (DMLC). Il est maintenant devenu la star des algorithme de Boosting en apprentissage statistique, notamment en raison de son extrême rapidité et sa précision de haut niveau. Il a également remporté de nombreuses compétitions Kaggle en surpassant tous les algorithme classiques d'apprentissage statistique. C'est un algorithme accessible sur la plupart des languages informatiques et souvent utilisé aussi bien pour des problèmes de classification que de régression.

Le nom XGBoost vient de l'abréviation de **"eXtreme Gradient Boosting"**, cela implique donc que l'algorithme reprend toutes les caractéristiques du Gradient Boosting ainsi que des arbres de décision, précédemment décortiqués.

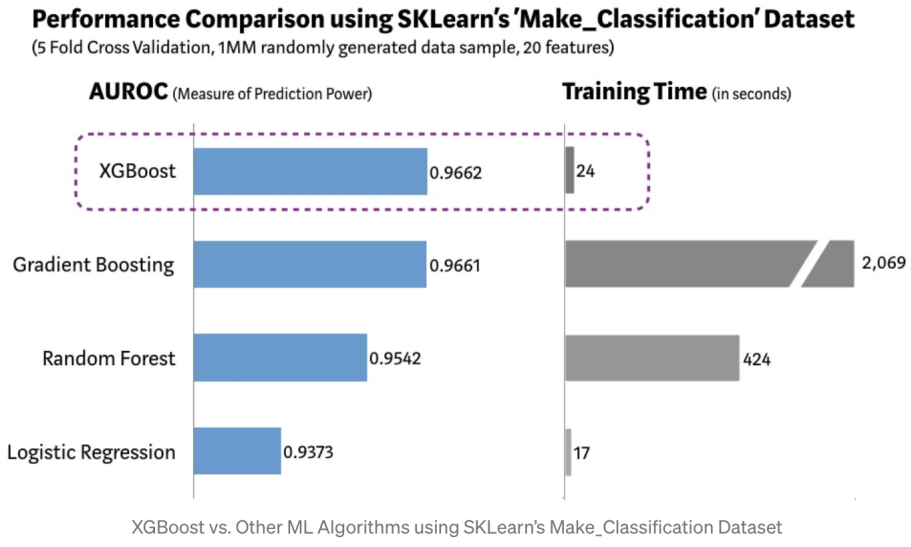


FIGURE 3.1 – Comparaison des algorithmes d'apprentissage statistique avec XGBoost

Il s'agit d'une version optimisée du Gradient Boosting qui utilise une fonction de perte personnalisable pour contrôler et optimiser les performances du modèle. XGBoost reprend le fonctionnement du Gradient Boosting mais s'effectue **10 fois** plus vite, d'où son surnom "Gradient Boosting on steroids" !

3.1 Construction de la fonction objectif

3.1.1 Fonction de perte personnalisable

On rappelle d'abord que la prédiction finale d'un algorithme de boosting est la somme des prédictions de tous les classifieurs faibles générés itérations après itérations. On note donc T le nombre d'arbres générés lors de la réalisation de l'algorithme et $\hat{y}_i^{(t)}$ la valeur de la feuille i de l'arbre t, c'est à dire la prédiction issue de l'arbre t de la donnée y_i, avec $i = 1, \ldots, n$. On a donc : $\hat{y}_i = \sum_{t=1}^{T} \hat{y}_i^{(t)}$.

La fonction de perte ou loss function en anglais a pour forme générique : $Loss = \sum_{i=1}^{n} L(\hat{y}_i, y_i)$, avec L une fonction convexe et différentiable.

- En **régression**, la fonction de perte est souvent $Loss = \frac{1}{2} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$.

- En **classification**, on préfèrera prendre le logarithme de la fonction de perte : $Loss = -\sum_{i=1}^{n} \Big(y_i.log(\hat{y}_i) + (1 - y_i).log(1 - \hat{y}_i) \Big)$, avec $y_i = 0$ ou 1.

XGBoost possède quant à lui une fonction objectif légèrement différente de la forme :

$$\boxed{Loss = \sum_{i=1}^{n} L(\hat{y}_i, y_i) + \sum_{t=1}^{T} P(g_t)}$$

avec $P(g_t) = \gamma.F_t + \frac{1}{2}.\lambda.||w_t||^2$ un terme de pénalisation (ou régularisation).

On note :

- F_t est le nombre de feuilles de l'arbre t
- $||w_t||^2$ la norme dans L^2 des valeurs de chaque feuille des arbres
- γ et λ les hyper-paramètres du modèle global permettant respectivement d'encourager l'élagage des arbres et de limiter l'overfitting

Avec cette fonction de pénalisation, l'algorithme fera en sorte de limiter les arbres qui ont un grand nombre de feuilles ou des valeurs de prédiction importantes. C'est ce procédé qui fait de XGBoost un algorithme unique permettant d'éviter au maximum le sur-apprentissage.

Un nouveau problème se pose alors : réussir à optimiser une fonction de perte aussi complexe que celle-ci, nous allons donc procéder à sa ré-écriture afin de la rendre plus exploitable.

3.1.2 Ré-écriture de la fonction objectif

Plusieurs éléments rendent l'optimisation de cette fonction objectif **trop complexe** et nécessite sa ré-écriture :

- La fonction $Loss$ fait intervenir deux sommes sur deux espaces différents, d'une part on somme sur le nombre d'observations afin de percevoir l'amélioration globale du modèle et d'autre part on somme sur le nombre d'arbre un terme de régularisation.
- Elle fait aussi intervenir $\hat{y}_i$ dans son expression, or $\hat{y}_i$ s'obtient de proche en proche et nécessite donc de connaître tous les arbres.

Les méthodes traditionnelles d'optimisation ne sont pas applicables dans notre cas, l'astuce qui va nous permettre de surpasser ce problème est d'utiliser le développement de Taylor.

En effet, les prédictions $\hat{y}_i$ sont calculées de proche en proche, on peut donc les modéliser comme suit :

A la première itération $t = 0$, on prend une constante comme première prédiction, par exemple 0 :

$$\hat{y}_i^{(0)} = 0$$

Puis à $t = 1$, on a :

$$\hat{y}_i^{(1)} = \hat{y}_i^{(0)} + f_1(x_i)$$

En itérant le processus, on a la relation suivante :

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)$$

Donc, à l'itération t, notre foction objectif vaudra :

$$Loss_t = \sum_{i=1}^{n} L(\hat{y}_i^{(t-1)} + f_t(x_i), y_i) + P(g_t)$$

Notre but est donc de trouver $f_t(x_i)$ qui minimise la fonction $Loss$.

On va prendre les mêmes principes que dans la descente de gradient, qui fait appel au gradient de la fonction. Cela revient donc à un développement de Taylor au 1er degré. On va pousser le developpement à l'ordre 2, ce qui permettra une simplification des écritures. On suppose que la quantité $f_t(x_i)$ est suffisamment

petite pour pouvoir appliquer la formule de Taylor au degré 2, on a donc la relation suivante :

$$Loss_t \approx \sum_{i=1}^{n} \left[L(\hat{y}_i^{(t-1)}, y_i) + g_i.f_t(x_i) + \frac{1}{2}.h_i.f_t^2(x_i) \right] + P(g_t)$$

avec $g_i = \frac{\partial L(\hat{y}_i^{(t-1)}, y_i)}{\partial \hat{y}_i^{(t-1)}}$ le gradient de la fonction L et $h_i = \frac{\partial^2 L(\hat{y}_i^{(t-1)}, y_i)}{\partial^2 \hat{y}_i^{(t-1)}}$ sa matrice Hessienne.

Ce qui a été fait ici est très important car maintenant la fonction $Loss_t$ ne dépend plus de l'itération t mais uniquement de l'itération $t-1$. Ce qui a été fait ici est en réalité l'application de la méthode de **Newton-Raphson** afin d'optimiser la Loss. C'est l'une des différence majeure de XGBoost avec l'algorithme de Gradient Boosting.

Reprenons l'expression que nous voulons minimiser à l'étape t :

$$f_t = Argmin_{f_t} \left\{ \sum_{i=1}^{n} \left[L(\hat{y}_i^{(t-1)}, y_i) + g_i.f_t(x_i) + \frac{1}{2}.h_i.f_t^2(x_i) \right] + P(g_t) \right\}$$

$$f_t = Argmin_{f_t} \left\{ \sum_{i=1}^{n} \left[L(\hat{y}_i^{(t-1)}, y_i) + g_i.f_t(x_i) + \frac{1}{2}.h_i.f_t^2(x_i) \right] \right\} + P(g_t)$$

Or, à l'étape t, $L(\hat{y}_i^{(t-1)}, y_i)$ est une constante donc il est inutile de la prendre en compte dans la minimisation, on a donc :

$$\boxed{f_t = Argmin_{f_t} \left\{ \sum_{i=1}^{n} \left[g_i.f_t(x_i) + \frac{1}{2}.h_i.f_t^2(x_i) \right] \right\} + P(g_t)}$$

Avec cette simplification, revenons maintenant au calcul de la fonction objectif, en précisant le terme de régularisation $P(g_t)$ pour l'arbre t :

$$Loss_t = \sum_{i=1}^{n} \left[g_i.f_t(x_i) + \frac{1}{2}.h_i.f_t^2(x_i) \right] + \gamma.F_t + \frac{1}{2}.\lambda.\sum_{j=1}^{F_t} w_j^2$$

On rappelle que F_t correspond au nombre de feuilles de l'arbre t. Définissons $f_t(x_i) = wj$ si on tombe sur la feuille j de l'arbre t dont la valeur est w_j.
Notons I_j l'ensemble des individus affectés à la feuille j.

En développant les sommes on obtient la relation suivante :

$$\boxed{Loss_t = \sum_{j=1}^{F_t} \left[\left(\sum_{i \in I_j} g_i \right).w_j \ + \ \frac{1}{2}.\left(\sum_{i \in I_j} h_i + \lambda \right).w_j^2 \right] \ + \ \gamma.F_t}$$

On peut désormais calculer facilement les w_j^* optimaux en résolvant l'équation suivante : $w_j = Argmin_{w_j} Loss_t \iff \frac{\partial Loss_t}{\partial w_j} = 0$.

On obtient alors les w_j^* optimaux tels que : $\boxed{w_j^* = -\frac{\sum\limits_{i \in I_j} g_i}{\sum\limits_{i \in I_j} h_i + \lambda}}$.

Cette formule permet de calculer la valeur optimale d'une feuille une fois l'arbre construit.

Maintenant, calculons la valeur de la fonction $Loss$ pour un arbre $t \in \{1, \ldots, T\}$. On a :

$$Loss_t = \sum_{j=1}^{F_t} \left[\left(\sum_{i \in I_j} g_i \right).w_j^* \ + \ \frac{1}{2}.\left(\sum_{i \in I_j} h_i + \lambda \right).w_j^{*2} \right] \ + \ \gamma.F_t$$

On remplace w_j^* par sa valeur que nous venons de trouver :

$$Loss_t = \sum_{j=1}^{F_t} \left[\left(\sum_{i \in I_j} g_i \right).- \frac{\sum\limits_{i \in I_j} g_i}{\sum\limits_{i \in I_j} h_i + \lambda} \ + \ \frac{1}{2}.\left(\sum_{i \in I_j} h_i + \lambda \right).\left(- \frac{\sum\limits_{i \in I_j} g_i}{\sum\limits_{i \in I_j} h_i + \lambda} \right)^2 \right] \ + \ \gamma.F_t$$

$$\iff Loss_t = \sum_{j=1}^{F_t} \left[- \frac{\left(\sum\limits_{i \in I_j} g_i \right)^2}{\sum\limits_{i \in I_j} h_i + \lambda} + \frac{1}{2}.\frac{\left(\sum\limits_{i \in I_j} g_i \right)^2}{\sum\limits_{i \in I_j} h_i + \lambda} \right] + \ \gamma.F_t$$

$$\iff \boxed{Loss_t = -\frac{1}{2}.\sum_{j=1}^{F_t} \frac{\left(\sum\limits_{i \in I_j} g_i \right)^2}{\sum\limits_{i \in I_j} h_i + \lambda} + \ \gamma.F_t}$$

Cette formule donne la perte pour l'arbre construit à l'étape t. Plus cette valeur est petite, meilleure sera la contribution de cet arbre. Cette formule nous servira plus tard à déterminer le split optimal pour l'arbre, c'est-à-dire quel est le critère à utiliser pour construire une nouvelle branche décisionnelle.

3.1.3 Critère de division des arbres de décision

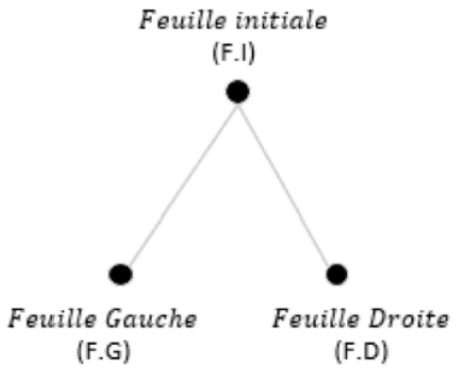
Une fois notre nouvelle fonction de perte déterminée précédemment, il sera facile de trouver un critère permettant de faire une nouvelle division (ou split), c'est à dire créer une nouvelle branche dans l'arbre.

Lors de chaque itération de l'algorithme, il sera nécessaire de savoir si il faut créer un **nouveau split** et si oui, quel **critère** utiliser afin de le réaliser.

Est-il nécessaire de faire un nouveau split ?

Afin de répondre à cette question, nous devons examiner la fonction de perte. Si le split permet de la diminuer, alors on l'effectue, sinon on s'arrête sans faire de split supplémentaire.

A la toute première itération, l'arbre ne possède qu'une seule feuille et donc une seule valeur. A l'itération suivante, nous avons un arbre de telle forme :



Il est important de souligner le fait que la la somme des pertes des deux nouvelles feuilles doit être inférieure à la perte de la feuille initiale afin que le split soit réalisable. C'est à dire que l'on effectue un split si et seulement si :

$$\boxed{Loss_{FG} + Loss_{FD} < Loss_{FI}}$$

On rappelle la formule permettant de calculer la perte sur un arbre, calculée précédemment :

$$Loss_t = -\frac{1}{2}.\sum_{j=1}^{F_t} \frac{\left(\sum\limits_{i\in I_j} g_i\right)^2}{\sum\limits_{i\in I_j} h_i + \lambda} + \gamma.F_t$$

Cette expression somme la perte sur toutes les feuilles de l'arbre, or ici nous n'avons besoin que de calculer la perte sur une seule feuille, F_t vaut donc 1. On a alors :

$$Loss = -\frac{1}{2}.\frac{\left(\sum_{i \in Feuille} g_i\right)^2}{\sum_{i \in Feuille} h_i + \lambda} + \gamma$$

Afin de savoir si un split est nécessaire, on va donc avoir l'équation :

$$-\frac{1}{2}.\frac{\left(\sum_{i \in FG} g_i\right)^2}{\sum_{i \in FG} h_i + \lambda} + \gamma - \frac{1}{2}.\frac{\left(\sum_{i \in FD} g_i\right)^2}{\sum_{i \in FD} h_i + \lambda} + \gamma < -\frac{1}{2}.\frac{\left(\sum_{i \in FI} g_i\right)^2}{\sum_{i \in FI} h_i + \lambda} + \gamma$$

$$\iff \boxed{\frac{\left(\sum_{i \in FG} g_i\right)^2}{\sum_{i \in FG} h_i + \lambda} + \frac{\left(\sum_{i \in FD} g_i\right)^2}{\sum_{i \in FD} h_i + \lambda} - \frac{\left(\sum_{i \in FI} g_i\right)^2}{\sum_{i \in FI} h_i + \lambda} - 2\gamma > 0}$$

Dans le cas où cette relation est vérifiée, un nouveau split sera donc crée. On remarque ici que les deux termes de régularisation λ et γ sont pris en compte, le nouveau split va faire converger la prédiction vers la solution, tout en respectant les contraintes de régularisation.

Nous savons maintenant si un split est réalisable ou non, nous voulons donc savoir quel critère utilisé afin de connaître la branche la plus apte à être divisée :

Quel critère utiliser ?

On introduit la fonction $Score$ qui permet de mesurer le gain effectué à l'issue d'un split.

$$Score = \frac{\left(\sum_{i \in FG} g_i\right)^2}{\sum_{i \in FG} h_i + \lambda} + \frac{\left(\sum_{i \in FD} g_i\right)^2}{\sum_{i \in FD} h_i + \lambda} - \frac{\left(\sum_{i \in FI} g_i\right)^2}{\sum_{i \in FI} h_i + \lambda} - 2\gamma$$

Une fois les feuilles pouvant effectuées un nouveau split déterminées, nous voulons connaître celle qui permettra de diviser le mieux les données et de progresser le plus rapidement vers la solution optimale. Pour cela, nous calculons le $Score$ de toutes les feuilles divisibles et on effectue la division sur celle qui présente le plus grand $Score$, c'est à dire celle où le gain est le plus élevé.

$\boxed{Score^* = Max\{Score_i \mid i \in D\}}$, où D désigne l'ensemble des feuilles divisibles.

3.1.4 Traitement des données manquantes

Dans cette partie nous allons expliquer comment XGBoost traite les **données manquantes** (valeurs nulles) d'une base de données. Il est crutial de ne pas négliger ces données car leurs valeurs nulles vont influer sur le $Score$ de chaque split et risquerait de ramifier les mauvaises branches de l'arbre. Pour remédier à ce problème, XGBoost effectue deux étapes supplémentaires lors de la détermination d'un nouveau split.

En cas de présence d'une donnée dont la feature est nulle, l'idée est de créer une branche par défaut qui recueillera toutes les données manquantes rencontrées. Afin de générer cette branche par défaut, l'algorithme va effectuer à chaque nouveau split une série d'étapes supplémentaires :

1. Construction d'un nouveau split en ne prenant en compte que les valeurs non nulles
2. Ce split possèdera donc un certain $Score$
3. Calcul du $Score$ en mettant toutes les features nulles sur la branche de gauche
4. Calcul du $Score$ en mettant toutes les features nulles sur la branche de droite
5. La branche ayant le **plus grand** $Score$ sera désignée branche par défaut

Ainsi, toutes les données présentant des features manquantes sont dirigées par défaut dans une branche spécifique de l'arbre afin de ne pas fausser la progression de l'algorithme.

FIGURE 3.2 – XGBoost, un algorithme très complet

3.1.5 Limites de l'algorithme et solutions

L'algorithme XGBoost possède tout de même **trois limites**.

Tout d'abord, nous l'avons vu, la recherche de splits peut être **fastidieuse** et assez longue en pratique, elle nécessite de faire beaucoup de calculs. Pour un jeu de données comportant n features et m valeurs pour chaque features, l'algorithme devra effectuer $n \times (m-1)$ calculs pour seulement un niveau de l'arbre afin de trouver le meilleur score ! Pour palier à ce problème, on peut faire les splits à partir de quantiles représentant la distribution statistique des données.

L'ensemble des données est divisé en **plusieurs petits sous-ensembles**, 10 quantiles chacun comprenant 10% des valeurs du dataset par exemple. Ainsi, les splits se feront aux frontières de chaque quantiles, certes le split idéal ne sera pas trouvé mais plutôt des approximations largement suffisantes. En présence d'un très gros volume de données, trouver les quantiles n'est pas une chose facile, on peut avoir recourt aux techniques de **Parallel learning et Weighted Quantile Sketch**. Les données sont séparées en plusieurs petits groupes et les quantiles sont donc calculés parallèlement puis rassemblés afin de trouver des quantiles approximatifs. Dans le cadre de XGBoost, ces quantiles ne sont pas tous identiques mais ils sont pondérés de sorte que la somme des poids dans chaque quantile soit approximativement la même.

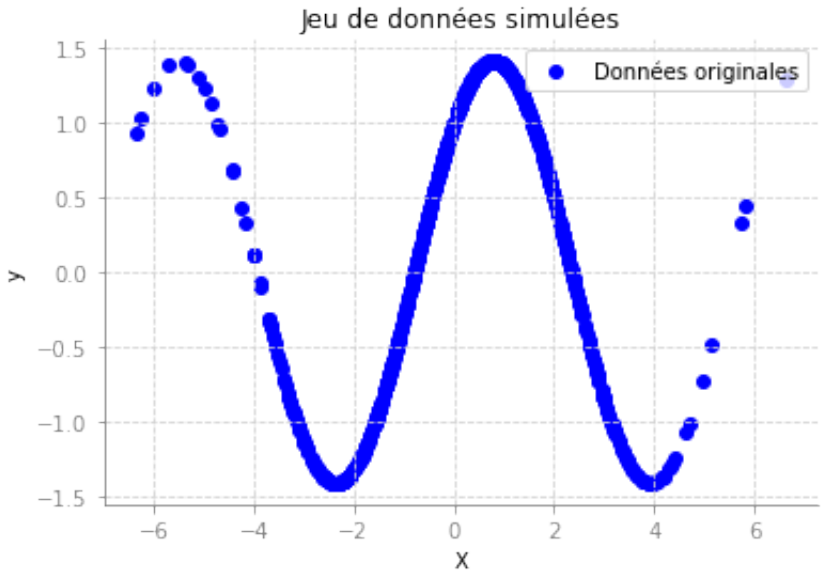
De plus, XGBoost prend parfois beaucoup de temps afin d'effectuer des **calculs de gradients et de matrices hessiennes** appliquées à chaque données. La solution est d'utilisé un **CPU** (Unité Centrale de Traitement) qui permet de mieux compresser les données et de stocker la formule du gradient et de la hessienne afin de pouvoir l'utiliser instantanément pour chaque observations.

Enfin, encore en cas de jeu de de données **trop volumineux**, les données doivent être stockées dans le disque dur lorsque la mémoire principale est saturée. Si le disque dur ne possède lui non plus plus assez de stockage, XGBoost met en place une technique appelé **Sharding**. Il s'agit de découper les données en subdivisions et chaque partition est conservée sur un coeur distinct, pour répartir la charge.

3.1.6 Application numérique de XGBoost

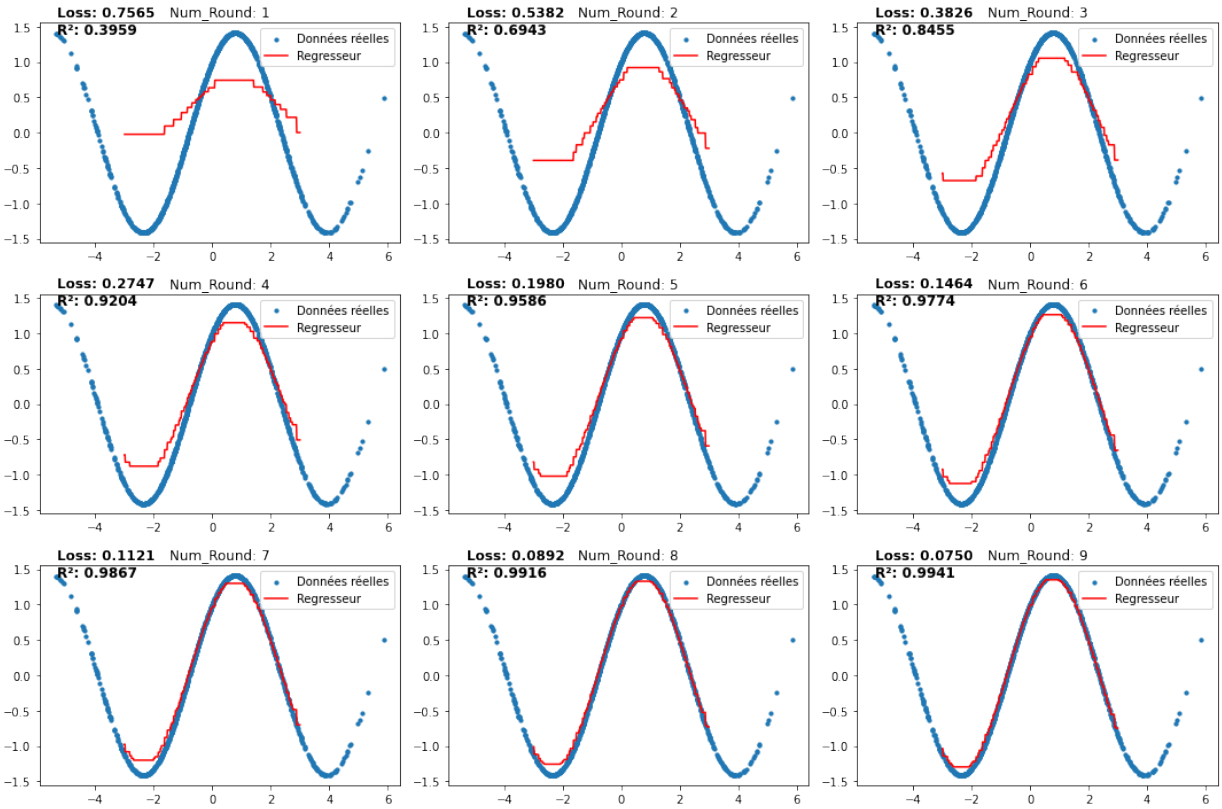
Dans cette partie, nous allons utiliser la bibliothèque **Sklearn**, appartenant à l'ennvironnement Python, afin de tester et de visualiser le fonctionnement de l'algorithme XGBoost.

On commence par simuler un jeu de données, ici on travaille en deux dimensions afin de pouvoir effectuer des graphiques interprétables. On trace 1000 points de la fonction : $f(x) = sin(x) + cos(x)$, on affiche ensuite nos données :



On souhaite effectuer une régression sur ces données afin d'en tirer des prédictions les plus fiables possibles. Pour cela, on va utiliser l'algorithme XGBoost avec la fonction suivante : $model = xgb.train()$.

On a ici indiqué au modèle XGBoost de n'effectuer que 9 itérations pour une représentation graphique plus lisible. On a ensuite ajouter à notre code des éléments permettant de visualiser itérations après itérations la progression de notre prédiction mais aussi l'évolution de la fonction de perte : "Loss", qui est ici la **RMSE** (racine de l'erreur quadratique moyenne) et notre métrique d'évaluation, le R^2. On obtient les graphiques suivants :



Obeservations :

- On trace en bleu nos données simulées originales et en rouge notre **régresseur**, c'est à dire notre classifieur fort, qui est une combinaison de tous les classifieurs faibles, pour chaque itérations.
- Dans le coin supérieur gauche, on affiche la valeur de notre Loss et du R^2.
- On remarque que la Loss diminue bien au fur et à mesure des itérations et le R^2 augmente en même temps jusqu'a être très proche de 1 !
- On voit que notre régresseur débute très loin de nos données réelles, les prédictions ne sont pas bonnes du tout. Avec le temps, il après de plus en plus efficacement de ses erreurs et corrige ses prédictions.
- A la 9ème itération, on voit que le régresseur s'est très bien adapté au modèle avec des métriques d'évaluations très satisfaisantes.

Remarques :

- Dans notre cas, on ne possède uniquement une seule variable explicative. Cela explique donc la rapidité de l'algorithme (9 itérations). En effet, si le jeu de données était très volumineux, le régresseur aurait mis plus de temps à s'ajuster au modèle aussi précisément.
- Afin de maximiser les performances de notre régresseur, on aurait pu utiliser une méthode de recherche des hyper-paramètres optimaux, telle que $GridSearch$. Cela aurait sans doute encore améliorer la vitesse de l'algorithme et certainement mieux contré le sur-apprentissage, mais dans notre exmple simple, la différence n'aurait pas été si significative que cela.

Conclusion

En conclusion, le boosting est une technique ensembliste permettant d'améliorer de manière **significative** les performances de modèles d'apprentissage statistique. Tout comme le **bagging**, le boosting procède à l'aggrégation de plusieurs classifieurs afin d'en tirer des prédictions. Cependant, le **boosting** consiste à combiner de manière séquentielle plusieurs modèles faibles pour former un modèle plus puissant, contrairement au bagging qui ne fait qu'assembler plusieurs classifieurs faibles indépendamment.

Nous avons d'abord examiné le fonctionnement des **arbres de décision**, en mettant l'accent sur leur construction et leur utilisation pour la classification et la régression. L'algorithme Random Forest permet d'amplifier leur efficacité en les combinant.
Ensuite, nous avons détaillé le principe des différents algorithmes de boosting, **AdaBoost, la descente de gradient et le Gradient Boosting**. Ils reposent tous sur la théorie des arbres dans le but d'aboutir à des prédictions toujours plus précises en minimisant le temps d'exécution.
Enfin, nous nous sommes intéressé à la star de ces algorithmes : **XGBoost**. Il se distingue par sa capacité à gérer efficacement des ensembles de données de grande taille et sa flexibilité pour traiter différentes tâches d'apprentissage. Il utilise une régularisation avancée en pénalisant la fonction de perte, lui permettant de contrôler la compléxité des modèles et d'éviter l'over-fitting.

Cependant, certaines situations rendent moins pertinentes son utilisation. C'est notamment le cas lorsque les données sont trop volumineuses et qu'elles comportent trop de valeurs aberrantes ou bruitées.

Une multitude de nouveaux algorithmes extrêmement puissants voient le jour, tels que **LightGBM** et **CatBoost**, montrant un grand potentiel. Combien de temps leur faudra t-ils pour détrôner XGBoost ?

Bibliographie

https ://eric.univ-lyon2.fr/ricco/cours/slides/gradient_boosting.pdf

https ://eric.univ-lyon2.fr/ricco/cours/slides/gradient_descent.pdf

https ://www.youtube.com/watch?v=iD_TL725wZs

https ://machinelearnia.com/descente-de-gradient/

https ://www.cs.toronto.edu/ mbrubake/teaching/C11/Handouts/AdaBoost.pdf

https ://who.rocq.inria.fr/Jean-Marc.Lasgouttes/mastere-esd/boosting/cours-boosting.pdf

https ://geoffruddock.com/adaboost-from-scratch-in-python/

https ://eric.univ-lyon2.fr/ricco/doc/tutoriel_arbre_revue_modulad_33.pdf

https ://math.unice.fr/ malot/CART.pdf

https ://perso.univ-rennes2.fr/system/files/users/rouviere_l/poly_apprentissage.pdf