# [Modélisez vos données avec les méthodes ensemblistes](https://openclassrooms.com/fr/courses/4470521-modelisez-vos-donnees-avec-les-methodes-ensemblistes)

Table des matières

[Modélisez vos données avec les méthodes ensemblistes 1](#_Toc57819536)

[**Améliorez la performance de vos algorithmes à l’aide des méthodes ensemblistes parallèles** 3](#_Toc57819537)

[**Les méthodes ensemblistes, késako** 3](#_Toc57819538)

[**D'abord, un peu de statistiques** 4](#_Toc57819539)

[**Les méthodes parallèles** 4](#_Toc57819540)

[**En pratique, il faut utiliser des apprenants faibles** 4](#_Toc57819541)

[**Conclusion** 5](#_Toc57819542)

[Contrôlez la variance à l’aide du bagging 6](#_Toc57819543)

[Comparaison avec un estimateur unique 7](#_Toc57819544)

[Conclusion 9](#_Toc57819545)

[Réduisez la corrélation entre les apprenants faibles à l’aide des forêts aléatoires 10](#_Toc57819546)

[Pour faire une forêt, il faut des arbres (de décision) 10](#_Toc57819547)

[Les forêts aléatoires, avantages sur le bagging 12](#_Toc57819548)

[Conclusion 14](#_Toc57819549)

[TP - Mesurez la puissance des forêts aléatoires 15](#_Toc57819550)

[Le dataset 15](#_Toc57819551)

[Application des forêts aléatoires 16](#_Toc57819552)

[Conclusion 18](#_Toc57819553)

[Partie 1 19](#_Toc57819554)

[Compétences évaluées 19](#_Toc57819555)

[ Question 1 19](#_Toc57819556)

[ Question 2 19](#_Toc57819557)

[ Question 3 19](#_Toc57819558)

[ Question 4 21](#_Toc57819559)

[ Question 5 21](#_Toc57819560)

[ Question 6 21](#_Toc57819561)

[ Question 7 22](#_Toc57819562)

[ Question 8 22](#_Toc57819563)

[**Initiez-vous aux méthodes séquentielles et au Boosting** 23](#_Toc57819564)

[**Les méthodes séquentielles, qu'est-ce que c'est ?** 23](#_Toc57819565)

[**Adaboost** 24](#_Toc57819566)

[**Boosting d'arbres aléatoires** 27](#_Toc57819567)

[**Conclusion** 27](#_Toc57819568)

[Décuplez les capacités du boosting : (X)GBoost 29](#_Toc57819569)

[Intuition du gradient boosting 29](#_Toc57819570)

[Formulation 30](#_Toc57819571)

[Contrôle du gradient boosting par les hyperparamètres 30](#_Toc57819572)

[Fonctions de pertes 30](#_Toc57819573)

[En pratique 31](#_Toc57819574)

[Conclusion 31](#_Toc57819575)

[Partie 2 32](#_Toc57819576)

[Compétences évaluées 32](#_Toc57819577)

[ Question 1 32](#_Toc57819578)

[ Question 2 32](#_Toc57819579)

[ Question 3 32](#_Toc57819580)

[ Question 4 33](#_Toc57819581)

[ Question 5 33](#_Toc57819582)

Après avoir étudié [les méthodes supervisées linéaires](https://openclassrooms.com/courses/entrainez-un-modele-predictif-lineaire) et [non-linéaire](https://openclassrooms.com/courses/utilisez-des-modeles-supervises-non-lineaires) les plus utilisés dans les cours précédent, il est temps d'aborder la famille des méthodes ensemblistes.

Êtes-vous prêt·e à décupler la puissance de vos modèles grâce aux méthodes ensemblistes ? C'est ce que nous allons voir dans ce cours, en nous intéressant à une famille d'algorithme parmi les plus performantes actuellement.

En effet, en utilisant de manière rusée notre jeu de données, nous pouvons exploiter tout son potentiel, en créant un grand nombre de petits modèles rapidement puis en développant un méta-modèle qui les rassemble.

Suivez ce cours pour apprendre les deux familles de modèles les plus utilisées par les data scientists : les**méthodes parallèles** avec les**forêts aléatoires** et les**méthodes séquentielles** dont le modèle phare est le**gradient boosting**.

**Objectifs pédagogiques**

À la fin de ce cours, vous saurez...

* Combiner différents algorithmes afin d’optimiser les performances de votre modèle global
* Utiliser les méthodes parallèles telles que le bagging ou les forêts aléatoires
* Utiliser les méthodes séquentielles telles que le boosting

**Prérequis**

Ce cours de Data Science se situe au croisement des mathématiques et de l'informatique. Pour en profiter pleinement, n'hésitez pas à vous rafraîchir la mémoire, avant ou pendant le cours, sur :

* Python pour le calcul numérique que nous utiliserons dans la partie TP du cours (librairie numpy et création de graphes avec pyplot)
* Quelques notions d'algèbre linéaire, telles que manipulation de vecteurs, multiplications de matrices, normes
* Quelques notions de probabilités et statistiques, telles que distribution de loi de probabilité et variance
* Le cours sur[**les méthodes supervisées linéaire**](https://openclassrooms.com/courses/entrainez-un-modele-predictif-lineaire) et [**non linéaires**](https://openclassrooms.com/courses/utilisez-des-modeles-supervises-non-lineaires)

**Améliorez la performance de vos algorithmes à l’aide des méthodes ensemblistes parallèles**

Les méthodes ensemblistes sont regroupées en deux sous-familles principales. Celles qui fonctionnent en parallèles et celles qui fonctionnent de manière séquentielle. Dans ce chapitre, nous allons d'abord étudier en détail ce que signifie "ensembliste" puis plus précisément les méthodes parallèles.

**Les méthodes ensemblistes, késako**

Ce cours est dédié aux méthodes ensemblistes. Comme leur nom l'indique, les méthodes d’ensembles suivent toutes le principe général suivant : **combiner des modèles avec des performances faibles permet d’obtenir un modèle prédictif plus efficace**. C’est l’effet de décision de groupe (*wisdom of crowds*)*,* que l’on applique déjà sur nous-mêmes en tant qu'êtres-humains

Je ne vois pas très bien, pourrais-tu donner un exemple de décision de groupe ?

Bien sûr, par exemple, la démocratie ! En effet, pour faire passer une loi, l’assemblée vote, on ne donne pas la responsabilité à une seule personne mais on préfère prendre la décision à la majorité de l’avis du groupe. On combine les différentes opinions dans l’objectif de prendre une meilleure décision.



La dernière élection présidentielle a élu au premier tour les deux candidats finalistes à la majorité des votes

Là ou ça devient intéressant, c’est si on prend le système de vote américain des grands électeurs. Les votes sont en effet pondérés selon une importance relative déterminée par le système. On peut faire un parallèle avec des méthodes que nous étudieront plus tard dans ce cours qui possèdent ce mécanisme de pondération.

Comme évoqué dans l’introduction, nous allons étudier en détail une première famille de méthodes, les méthodes dîtes parallèles.

**D'abord, un peu de statistiques**

Dans la nuit des temps (la fin des années 70), les statisticiens se sont penchés sur la technique d’échantillonnage (*sampling*), pour créer une nouvelle famille de méthodes, appelées *bootstrap*.

Imaginez vouloir estimer une variable à l’aide d’un échantillon de population. Par exemple, estimer l’âge moyen d’un·e étudiant·e d'OpenClassrooms, à l’aide d’un échantillon de 100 étudiant·es. La première chose à faire est bien entendu de calculer cette moyenne sur les 100 étudiant·es, et d’en déduire un intervalle de confiance associé en fonction du nombre étudiant·es total d'OpenClassrooms.

La méthode du *bootstrap* consiste à générer un ensemble d'échantillons **de même taille** que l'échantillon original à partir de celui-ci. En l'occurence ici, générer un échantillon de taille 100 à partir de notre échantillon original de 100 étudiant·e. Ces échantillon *bootstrap* sont créés en échantillonnant les entrées une par une **avec remplacement**.

Une fois un grand nombre d'échantillon générés, on peut calculer la moyenne sur chacun de ces échantillons. L'histogramme de répartition des moyennes sur ces différents échantillons nous donnera alors une idée de la variabilité de la moyenne, plus précis que l'intervalle de confiance classique avec une hypothèse de normalité.

**C'est cette première méthode, le *bootstrap*, qui a donnée naissance à la famille des méthodes parallèles.**

**Les méthodes parallèles**

La famille de méthodes de cette première partie sont appelées méthodes « parallèles ». C’est pour signifier qu’en fait, nous allons entraîner plusieurs modèles de manière indépendante (en parallèle) pour ensuite les regrouper afin de prendre une décision.

Plutôt simple non ? 😶

**En pratique, il faut utiliser des apprenants faibles**

En réalité, la théorie des méthodes ensemblistes est basée sur la supposition que l’on peut améliorer ce qu’on appelle des apprenants faibles en les combinant. Un apprenant faible désigne un modèle qui fait seulement légèrement mieux qu’un simple modèle aléatoire. Ils sont en général plus faciles à créer, et lorsqu'ils sont combinés avec des méthodes intelligentes, ils permettent ainsi d’être plus performants qu’un modèle unique. Par contre, ils sont plus complexes à créer et utiliser.

Nous allons commencer par utiliser les arbres de décisions, qui sont considérés comme des apprenants faibles quand ils sont de petite taille.

**Conclusion**

Les méthodes ensemblistes, notamment apparues à travers des méthodes pionnières comme le *bootstrap,*permettent de rendre des modèles individuels (souvent des apprenants faibles) plus performants en les combinant de manière intelligente. Une première méthode parallèle, le *bagging,* sera présentée dans le chapitre suivant.

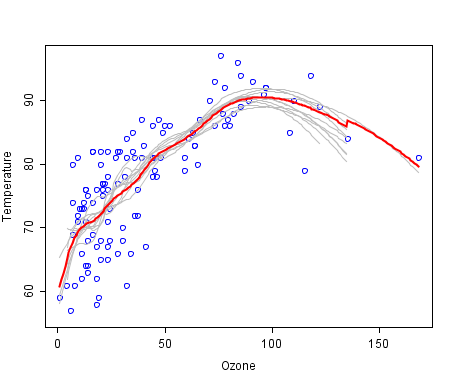
## Contrôlez la variance à l’aide du bagging

Le ***bagging*** qui est un diminutif de bootstrap aggregation (vous allez comprendre ce nom dans un instant) est une méthode permettant de combiner nos fameux apprenants faibles. Le principe est de créer un dataset différent pour entrainer nos différents apprenants. Pour cela on va appliquer notre méthode statistique du bootstrap : pour rappel, cela consiste à échantillonner avec replacement notre dataset de taille N afin d’obtenir un nouveau dataset de taille N. Chaque observation a une chance 1 / N d’être tirée et on répète ce processus N fois.

Attends une seconde, on échantillonne pour obtenir un dataset de même taille, quel est l’intérêt ?

Rappelez-vous, notre idée de base est de combiner les apprenants faibles. Pour générer des apprenants faibles (qui soient suffisamment différents de surcroît), une solution est de les entraîner sur un nouveau dataset. En considérant que l'on n'a pas une infinité de données, une solution est donc de “générer” (c'est-à-dire bootstraper) de nouveaux dataset qui “ressemblent” (en termes de distribution de probabilités) au dataset original. On échantillonne donc un dataset de taille N à partir de notre dataset original de taille N. 😉

Une fois que l’on a nos modèles, on effectue une prédiction simplement en effectuant un vote à la majorité pour la classification, ou la moyenne pour la régression.



On effectue une régression sur chaque échantillon, qu'on combine pour obtenir la régression finale

Et voilà pour l'algorithme du bagging 🙂

Mais ça semble un peu facile, pourquoi ça marche ?

On a créé un certain nombre d’apprenants faibles. La notion de “faible” désigne notamment le fait que nos modèles possèdent une forte variance. Le bagging permet de réduire la variance des estimateurs individuels et offre une prédiction plus performante et plus stable, dépendante du nombre d’apprenants faibles utilisés.

### Comparaison avec un estimateur unique

Nous allons observer l'effet des méthodes ensemblistes sur un petit jeu de données factice, afin de nous rendre compte de l'effet du bagging.

Commençons par créer un jeu de données d'exemple, à l'aide de la fonction make\_moons

from sklearn.datasets import make\_moons

X, y = make\_moons(n\_samples=100, noise=0.25)

On sépare ensuite le jeu de données en données d'entraînement et données test

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y)

On peut maintenant créer notre modèle de bagging à l'aide de la fonction BaggingClassifier  . Par défaut, le classifieur de base utilisé par Scikit learn est un arbre de décision donc inutile de le préciser. En revanche, on précise le nombre de classifieurs individuels utilisés (en l'occurence 5).

from sklearn.ensemble import BaggingClassifier

bagging = BaggingClassifier(n\_estimators=5)

bagging.fit(X\_train, y\_train)

On va maintenant afficher les différentes zones de classifications des arbres ainsi que du bagging de ces arbres. Pour cela j'ai utilisé deux fonctions d'affichage :  plot\_tree\_partition et plot\_2d\_separator  [disponibles ici](https://github.com/amueller/mglearn/tree/master/mglearn). Le plus simple est d'installer le packagemglearnavec la commande:

pip install mglearn

Puis d'importer les fonctions :

from mglearn.plot\_interactive\_tree import plot\_tree\_partition

from mglearn.plot\_2d\_separator import plot\_2d\_separator

from mglearn.tools import discrete\_scatter

Enfin, on obtient :

fig, axes = plt.subplots(2, 3, figsize=(20, 10))

for i, (ax, tree) in enumerate(zip(axes.ravel(), bagging.estimators\_)):

ax.set\_title("Tree {}".format(i))

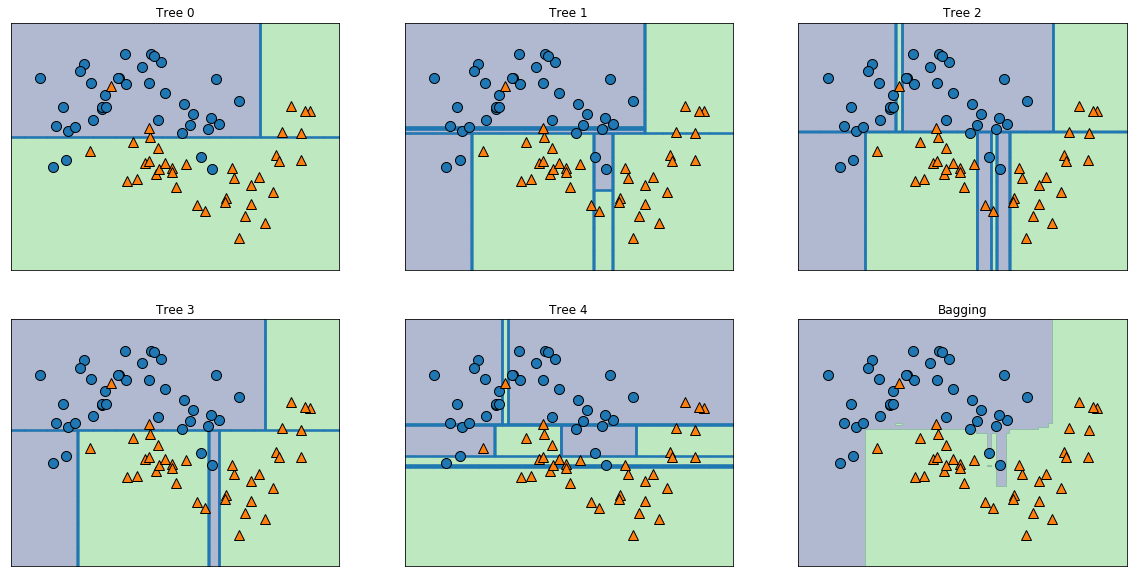
plot\_tree\_partition(X\_train, y\_train, tree, ax=ax)

plot\_2d\_separator(bagging, X\_train, fill=True, ax=axes[-1, -1],

alpha=.4)

axes[-1, -1].set\_title("Bagging")

discrete\_scatter(X\_train[:, 0], X\_train[:, 1], y\_train)



Les différents classifieurs

On peut voir que les arbres effectuent des erreurs (overfitting) relativement différentes, qui une fois combinés par le bagging sont lissées et donnent une bien meilleure performance. Et encore, n'on a utilisé que 5 arbres à des fins de démonstration, mais en réalité on va en utiliser bien plus en pratique (des centaines voir des milliers).

### Conclusion

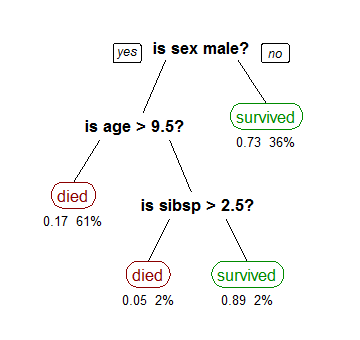
Nous avons vu l'algorithme fondamental du bagging qui est une base pour beaucoup d'algorithmes ensemblistes même si ce n'est pas celui que l'on va utiliser le plus souvent en pratique. Passons maintenant à un algorithme phare : les forêts aléatoires !

## Réduisez la corrélation entre les apprenants faibles à l’aide des forêts aléatoires

Dans ce chapitre nous allons étudier un modèle célèbre : les forêts aléatoires (ou random forests en anglais). Cette méthode est très utilisée par les data scientists car elle permet des performances plus que raisonnables pour la majorité des problèmes rencontrés.

### Pour faire une forêt, il faut des arbres (de décision)

Voyons un exemple (plutôt original 😬) ensemble. Étudions l’arbre de décision pour avoir la probabilité de survie des passagers du Titanic. La caractéristique **sibsp** représente le nombre total de proches (mari, frères et soeurs) :



Un arbre de décision d'exemple sur la probabilité de survie des passagers du Titanic

Comme on peut le voir, effectuer une classification est relativement simple. Il suffit de répondre aux questions en descendant dans l'arbre de décision. Par exemple, les garçons de moins de 9 ans et demi qui avaient moins de 3 proches présents sont classés comme ayant survécu. Ou bien une femme sera automatiquement classée comme ayant survécu par l'arbre de décision. On peut voir notamment comment il est facile d'effectuer de l'overfitting sur les données.

On parle surtout des arbres de classification binaire, dont les embranchements représentent des questions auxquelles on peut répondre par “oui” ou “non”. A force d’ajouter ces questions, on peut arriver à un intervalle de confiance assez élevé, même s'il y a un risque d'overfitting.

Les arbres de décision sont des modèles **non-paramétriques** et trouvent des règles en général assez puissantes. Ils peuvent traiter des très grands datasets et ils peuvent aussi utiliser des prédicateurs mixtes (catégoriques et nombres). Les variables redondantes sont éliminées, parce que l’arbre ne les prend même pas en compte.

Mais, les prédictions des arbres individuels sont souvent très médiocres, essentiellement à cause de la variance de la prédiction (avec beaucoup de bruit) et de la tendance à overfitter. C’est pour ça que les méthodes ensemblistes existent 😀

#### Intuition du fonctionnement d'un arbre de décision

Construire un arbre de décision binaire se résume au problème suivant : faire grandir un arbre embranchement par embranchement jusqu'à ce que le "gain d'information" soit maximisé. Si on créé un nouvel embranchement à partir de ce point, le gain sera diminué.

Comment trouve-t-on cet arbre ?

C’est fait à la base (algorithme CART, pour Classification And Regression Tree) par greedy, top-down recursive partitioning algorithm. Houla, ça fait beaucoup, voyons comment décomposer cet algorithme.

##### Première étape : faire grandir l'arbre

A chaque nœud, on doit donc trouver deux choses : quelle feature utiliser et quel est le point de séparation des deux zones, de sorte à minimiser l’erreur quadratique pour la régression et l’impureté pour la classification. On fait ainsi grandir l’arbre de la manière la plus grande possible (d’où l'adjectif greedy).

**Impureté**? Kesako?

L'impureté est la mesure probabilistique de la performance de l'arbre de décision. La plus utilisée, notamment pour les forêts aléatoires, est **l’index d’impureté de Gini**. C’est à l’aide de cette impureté que nous allons pouvoir choisir notre point de séparation, en testant toutes les valeurs possibles sur une feature choisie.

À noter qu'il est possible d'utiliser d'autres types d'heuristiques pour faire grandir notre arbre. Notamment **l'entropie** citée plus haut ou encore **le gain d'information**.

##### Seconde étape : élaguer l'arbre

Comme on peut l’imaginer, l’idée de coller autant aux données d’entraînement pour construire son modèle conduit facilement à l'overfitting. Une des techniques qui permet de réduire cet overfitting pour un arbre, c’est l’élagage. Pour élaguer l’arbre, il s'agit de prendre en compte le coût de la complexité du modèle. En effet, on ne peut faire grandir un arbre jusqu’à identifier chaque observation comme une feuille. On doit donc choisir entre la qualité de l’ajustement de l’arbre sur les observations et la complexité du modèle. Celui-ci ne doit pas être trop complexe sous peine d’overfitting.

### Les forêts aléatoires, avantages sur le bagging

L’élagage étudié au chapitre précédent est une manière de réduire l’overfitting. Une autre manière de le réduire est d’utiliser une méthode ensembliste, en l’occurence les forêts aléatoires. Les forêts aléatoires sont donc un ensemble d’arbres de décisions entraînés individuellement, légèrement différents les uns des autres. Pour prédire une nouvelle valeur, on effectue la classification pour chaque arbre de cette forêt. La forêt choisit la valeur ayant le plus de votes parmi tous ses arbres. Ce n'est pas plus compliqué que ça !

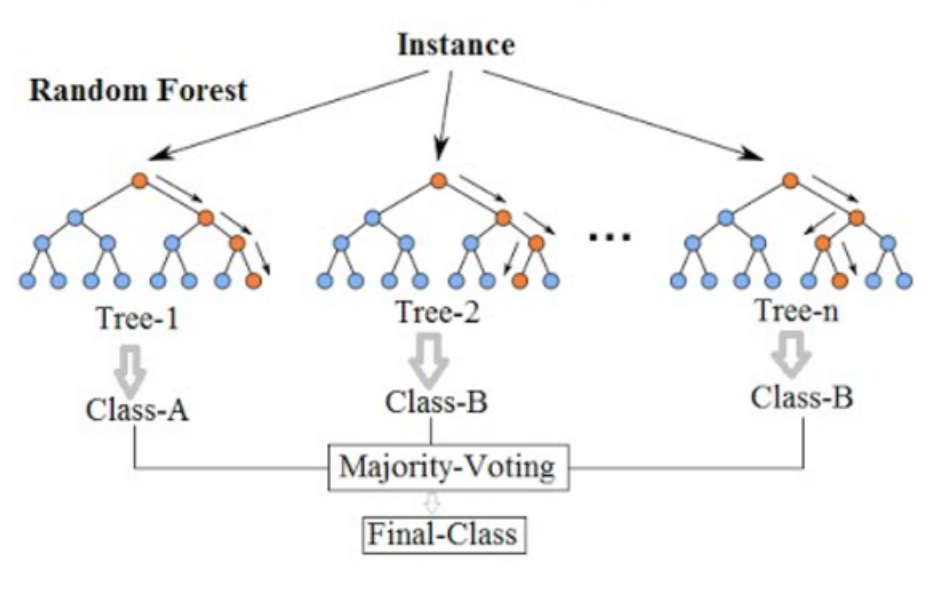


Illustration du fonctionnement d'une forêt aléatoire

L’idée derrière les forêts aléatoires est que chaque arbre de décision overfit (c'est prévu !) sur une partie des données, tout en possédant une bonne capacité de prédiction. Toute la subtilité réside, vous l’avez deviné, dans la sélection de l’échantillon à utiliser pour créer chaque arbre, car on veut qu’ils soient tous différents. Pour cela, on va bootstraper le jeu de données, c’est à dire que si on a un jeu de données de N entrées, alors on créé un échantillon aléatoire de N entrées (mais avec remplacement). Ce sera le jeude données qui nous servira pour cet arbre en question.

#### Créer notre forêt d'arbres de décision

Pour la création de chaque arbre de décision, on va différer légèrement de CART afin que nos arbres ne se ressemblent pas trop. Si on a M features d’entrée, on choisit un nombre m << M tel qu’a chaque nœud de notre arbre on sélectionne m features parmi les M aléatoirement et le meilleur « point de séparation ». Ce **m** est gardé constant durant la création de la forêt aléatoire. Tous les arbres sont créés sans élagage, contrairement à CART, on essaie ici de créer l’arbre avec les meilleures performances par validation croisée.

Pour résumer, les forêts aléatoires sont nommées ainsi parce qu’**on injecte de l’aléatoire dans la création de chaque arbre**. Premièrement avec la création d’échantillons aléatoires (bootstraping). Deuxièmement, avec la sélection des features sur lesquels effectuer le test de séparation.

La performance de notre forêt aléatoire dépend alors de deux paramètres :

* **La corrélation** entre n’importe quelle paire de deux arbres de la forêt. Si la corrélation augmente, l’erreur augmente (puisque la variance est plus forte).
* La **performance** de chaque arbre pris individuellement. Plus les performances de l'arbre individuel sont importantes, plus la forêt aura de chance d’être globalement performante.

Vous l’aurez deviné, le mystérieux**hyperparamètre m** permet à la fois d’augmenter la performance des arbres individuellement, puisque l’augmenter crée un critère de séparation plus précis. Il permet aussi de réduire la corrélation entre les différents arbres, puisque les points de séparation sont moins susceptibles d’être les mêmes. C’est bien sûr **au détriment de la vitesse d’exécution** de l’algorithme puisque l'on augmente la complexité du modèle.

Je rappelle que les forêts aléatoires sont une amélioration du bagging puisqu'elles créent des arbres plus différents structurellement en échantillonnant aléatoirement seulement un sous-ensemble des features disponibles, ce qui permet de réduire la corrélation entre les arbres et donc d'obtenir un meilleur modèle final

#### Quelques-uns des (nombreux) avantages des forêts aléatoires

Les forêts aléatoires sont un algorithme possédant de très bonnes performances assez facilement sur la plupart des problèmes que vous rencontrerez. C’est un go-to à tester dès le départ en abordant un problème de modélisation, sans trop besoin de faire de feature engineering en amont. De plus, il fonctionne bien sur les grosses bases de données, car il ne possède pas une si grosse complexité pour l’entraînement. Ceci-dit si vous êtes en présence d'un dataset volumineux, vous pouvez effectuer une parallélisation des calculs sur plusieurs CPU (avec l'argument n\_jobs dans scikit-learn).

Les forêts aléatoires permettent une certaine interprétabilité. Elles forment donc une black-box pas complètement « black » ! On peut en effet effectuer une estimation de l’importance des différentes features (nous verrons cela dans le TP).

Elles permettent de nombreuses applications telles que du clustering, de la détection d’outliers... en exploitant une méthode qui permet le calcul de proximité entre les paires de données.

Encore une fois, ce modèle est rapide. En plus, les forêts n’overfittent pas (!). Le problème de taille mémoire pour de gros jeux de données est le stockage du jeu de données lui-même.

Les forêts aléatoires sont donc à utiliser sans modération !

### Conclusion

En raison des nombreux avantages que l'on vient d'évoquer, les forêts aléatoires constituent un outil essentiel de l'arsenal du data scientist. Il convient de l'utiliser avec rigueur, cependant, dans le choix des hyperparamètres afin d'obtenir des performances satisfaisantes. Il faut aussi ajouter une bonne dose de feature engineering pour améliorer la rapidité, si cela est nécessaire.

## TP - Mesurez la puissance des forêts aléatoires

Le fonctionnement des forêts aléatoires doit maintenant être un peu plus clair pour vous. Il est temps de passer à la pratique en observant les performances de ce type de modèle sur des données réelles 🙂

Dans ce chapitre, on va appliquer l’algorithme des forêts aléatoires sur un exemple concret. Le jeu de données que j’ai choisi est assez connu : il permet de reconnaître l’activité physique à partir de données du smartphone. Il est simple mais possède de nombreuses variables (> 500) ce qui va nous permettre d’étudier un certain nombre de choses. Prêt·e ? Affutez votre Notebook et [téléchargez le nouveau dataset ici](https://www.kaggle.com/uciml/human-activity-recognition-with-smartphones).

### Le dataset

En plus de charger le dataset, vous pouvez aussi observer le fichier de description des différentes variables afin d’avoir une meilleure idée des données à disposition.

La connaissance du domaine est très importante pour un data scientist. Plus vous serez renseigné vos données, plus vous saurez créer des modèles efficaces rapidement. Vous ne devriez pas travailler à l’aveugle sur des variables muettes. C’est aussi important en amont pour avoir des hypothèses à confirmer par analyse exploratoire.

Dans un premier temps, étudions le dataset à notre disposition : le "[Human Activity Recognition Using Smartphones Data Set](https://www.kaggle.com/uciml/human-activity-recognition-with-smartphones)".

Ce jeu de données contient les logs de capteurs de smartphone sur une trentaine d'individus en train d'effectuer des activités (s'assoir, se mettre debout, marcher, etc). L'objectif sera de prédire à partir des logs de capteurs le type d'activités que le sujet est en train d'effectuer.

En regardant le fichier de description du dataset, on peut observer qu'il y a beaucoup de features (561). D'emblée, en observant ce qu'elles désignent, on peut se dire qu'il y a une certaine redondance entre toutes ces variables. Dans un premier temps, on va effectuer une modélisation "brute" sans se soucier de nettoyer le jeu de données.

Dans un second temps, on va utiliser cette première modélisation pour mieux comprendre le dataset et ainsi effectuer une seconde modélisation plus efficace en éliminant des variables peu importantes.

On commence par charger les données.

import pandas as pd

train = pd.read\_csv("train.csv")

test = pd.read\_csv("test.csv")

X\_train = train[train.columns[:-2]]

y\_train = train['Activity']

X\_test = test[test.columns[:-2]]

y\_test = test['Activity']

On regarde la taille des données.

print(train.shape)

(7352, 563)

Les données représentent des vecteurs de différentes mesures effectuées par le téléphone (accélération, secousses, etc). La dernière colonne représente l'activité, c'est ce qu'on va essayer de prédire à partir du reste. On va d'abord éliminer les features redondantes (intuitivement, les coordonnées polaires et cartésiennes doivent être corrélées par exemple ... ) Une première manière de faire serait de réfléchir et se renseigner sur le domaine d'études en question pour pouvoir éliminer des variables qui transmettent des informations similaires où n'influencent pas ou très peu la prédiction que l'on veut effectuer.

La seconde manière est d'utiliser justement une forêt aléatoire (!) de laquelle on va extraire l'importance des features qui la constituent, et ainsi déterminer quelles sont les features les plus importantes à partir de ça.

### Application des forêts aléatoires

Une fois les données chargées, on peut déclarer un nouveau modèle de forêts aléatoires pour la classification logiquement appelé dans scikit sklearn.RandomForestClassifier . On définit comme hyperparamètres 500 pour le nombre d'arbres.

from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier(n\_estimators=500, oob\_score=True)

On peut maintenant entraîner notre modèle sur les données brutes, sans autre forme de procès.

model = rfc.fit(X\_train, y\_train)

Voyons les performances de ce modèle sur le jeu de données d'entraînement

from sklearn.metrics import accuracy\_score

pred = rfc.predict(X\_test)

print("accuracy {:.2f}".format(accuracy\_score(y\_test, pred)))

0.93

Pas mal 😎  !

Maintenant que notre modèle est créé, on peut effectuer une sélection des features les plus importantes. Pour cela on va utiliser la fonction SelectFromModel qui utilise la propriété du modèle qu'on vient de créer model.feature\_importances qui permet d'évaluer l'importance relative des features fournies à la base (sur une échelle de 0 à 1). Intuitivement, cette importance est calculée en considérant que plus une feature est haute, plus elle contribue à une fraction plus élevée du jeu de donnée d'entraînement et donc des données au global. On considère donc qu'elle a plus d'importance que les features plus bas dans l'arbre. Cette fraction est utilisée comme estimateur de l'importance de la feature dans cet arbre, qu'on peut ensuite généraliser à la forêt entière.

Si on a peu de features, on pourrait les afficher sur un histogramme afin d'évaluer à l'œil si il n'y a pas déjà une sélection à faire [comme ici](http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html).

Donc en utilisant SelectFromModel avec un seuil d'importance choisi à l'aide de l'argument threshold, on peut créer une sélection des features qui sont les plus importantes à la création d'un modèle.

from sklearn.feature\_selection import SelectFromModel

select = SelectFromModel(rfc, prefit=True, threshold=0.003)

X\_train2 = select.transform(X\_train)

print(X\_train2.shape)

(7352, 84)

On a divisé par 5 le nombre de features utilisées, pas mal mais voyons si les performances restent similaires. À l'aide de l'argument threshold, on peut choisir le seuil d'importance que l'on souhaite pour les features à sélectionner.  On calcule en même temps le gain de temps car c'est ce qui nous intéresse principalement dans l'amélioration des performances.

import timeit

rfc2 = RandomForestClassifier(n\_estimators=500, oob\_score=True)

start\_time = timeit.default\_timer()

rfc2 = rfc2.fit(X\_train2, y\_train)

X\_test2 = select.transform(X\_test)

pred = rfc2.predict(X\_test2)

elapsed = timeit.default\_timer() - start\_time

accuracy = accuracy\_score(y\_test, pred)

print("accuracy {:.2f} time {:.2f}s".format(accuracy, elapsed))

accuracy 0.89 time 16.38s

Et pour le modèle avec toutes les features.

accuracy 0.93 time 50.04s

On a donc plus que divisé par trois le temps de calcul, sans trop perdre en performances ! C'est plus que respectable pour un premier jet.

### Conclusion

A vous maintenant de bidouiller afin d'améliorer les performances du modèle finale en modifiant les différents hyperparamètres de contrôle et trouver la bonne balance entre performances de classification / temps de calcul. Bien sûr le ratio souhaité dépendra de la problématique : est-ce qu'on veut une très bonne performance et peu importe le temps de calcul, ou bien un temps de calcul le plus rapide possible mais avec une performance minimum etc etc

# Partie 1

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Utiliser les méthodes parallèles telles que le bagging ou les forêts aléatoires

### Question 1

**Les méthodes parallèles sont une famille de méthodes qui...**

* + 

combinent plusieurs apprenants faibles pour créer un modèle performant

* + 

font des calculs distribués, d'où le nom « parallèle »

* + 

ont plus de variance que les apprenants individuels

*Les méthodes parallèles sont une sous-famille des méthodes ensemblistes dont la particularité est d'apprendre en combinant des apprenants faibles.*

### Question 2

**Les apprenants faibles sont un type de modèle d'apprentissage qui**

* + 

Font légèrement mieux qu'une réponse aléatoire

* + 

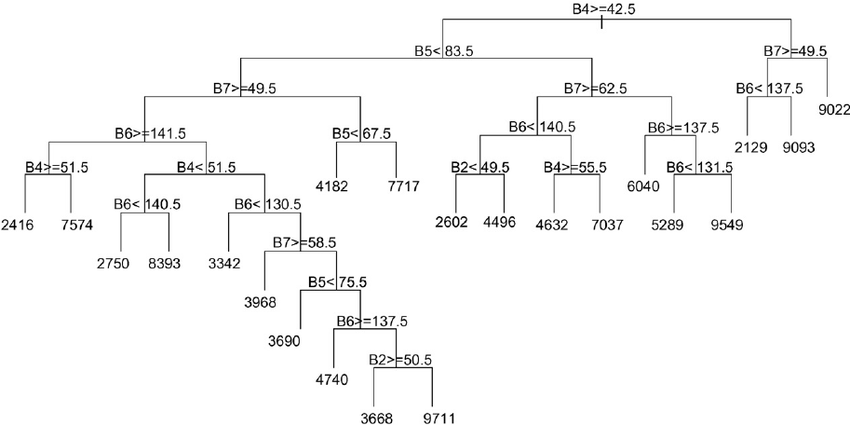
Font légèrement moins bien qu'une réponse aléatoire

* + 

Ne sont jamais des arbres de décision

*Si un apprenant fait moins bien qu'une réponse aléatoire, ça veut dire qu'ils n'ont pas saisi d'information relative au modèle. Si ils font légèrement mieux, ils ont saisi une partie de l'information, même si elle est petite. Les arbres de décision sont les apprenants faibles les plus utilisés.*

### Question 3

****

**A l'aide de l'arbre de décision ci-dessus, quelle est la valeur de la classification d'une observation ? À chaque embranchement, à gauche se trouvent les réponses positives au critère de décision, à droite négatives.**

B=(B1,B2,B3,B4,B5,B6,B7)=(11,81,54,55.1,92,142,63)

* + 

4632

* + 

4496

* + 

4182

* + 

7717

* + 

7037

* + 

9093

* + 

7574

*En suivant l'arbre avec l'observation fournie, on tombe naturellement sur la réponse 7037*

### Question 4

**Le bagging permet de**

*Attention, plusieurs réponses sont possibles.*

* + 

regrouper plusieurs jeux de données en un seul afin de créer un modèle robuste

* + 

combiner plusieurs apprenants faibles entrainés individuellement pour créer un modèle robuste

* + 

exploiter le jeu de données de manière intelligente en bootstrapant des nouveaux échantillons de données afin de créer un modèle robuste

*Le bagging (bootstrap aggregation) est un des premiers algorithmes ensemblistes parallèles et permet donc de****combiner plusieurs apprenants faibles entrainés individuellement pour créer un modèle robuste****. Il exploite ainsi le jeu de donnée au maximum en "augmentant" les données avec la technique du bootstrap pour entraîner les différents apprenants, ce qui rend le modèle final plus robuste.*

### Question 5

**A la différence du bagging, les forêts aléatoires...**

* + 

utilisent seulement un sous-ensemble des observations disponibles afin de rendre le modèle moins sujet aux corrélations entre les arbres.

* + 

utilisent seulement un sous-ensemble des features disponibles afin de rendre le modèle moins sujet à la corrélation entre les arbres.

*utilisent un sous-ensemble des features aléatoires permettant de créer une contrainte d'apprentissage pour chaque arbre différente des autres, ce qui réduit la corrélation entre eux.*

### Question 6

**On peut évaluer l'importance des features grâce aux forêts aléatoires...**

* + 

en pondérant les observations pour obtenir une importance relative.

* + 

en évaluant l'importance des features sur une partie des données test.

* + 

en évaluant l'importance des features par la fraction des décisions de l'arbre influencé par celles-ci.

*en observant la fraction d'un arbre qui utilise une feature, cela nous donne un bon ersatz de l'importance de cette feature dans le modèle puisque il impacte sur la décision finale.*

### Question 7

**Effectuer une sélection des features au préalable pour créer notre modèle final de forêt aléatoires permet d'obtenir**

* + 

Un gain de précision du modèle final

* + 

Un gain de robustesse du modèle final

* + 

Un gain de rapidité de temps de calcul

*Réduire la dimension des observations permet de simplifier les calculs dans le modèle et donc de les rendre plus rapides. On n'aura pas de gain de précision en général puisque il y a moins d'informations en entrée.*

### Question 8

**En statistique, lorsqu'on bootstrap cela signifie...**

* + 

que l'on récupère un nouveau dataset à partir des données de test

* + 

que l'on génère un nouveau dataset à partir des données d'entraînement, sans remise

* + 

que l'on génère un nouveau dataset à partir des données d'entraînement, avec remise

*La technique du bootstrap consiste à générer un  nouvel échantillon de donnée entrée par entrée avec remise*

**Initiez-vous aux méthodes séquentielles et au Boosting**

Dans cette seconde partie, intéressons-nous à la seconde famille de méthodes ensemblistes, les méthodes séquentielles.

**Les méthodes séquentielles, qu'est-ce que c'est ?**

Imaginez le problème suivant : on veut déterminer si un champignon est toxique ou pas, à partir de certaines caractéristiques (couleur, taille, odeur, etc).



Alors, une idée ?

Pour cela, nous disposons d’un jeu de données d’entraînement contenant un vecteur d’observations de différents champignons X, ainsi qu’un label binaire y∈{−1,1} indiquant s’il est effectivement toxique

Un modèle de classification C(X) retourne la prédiction binaire “toxique” ou “non toxique”. Ça peut être n’importe quel modèle de classification étudié précédemment (réseau de neurones, SVM, etc). De la même manière que les algorithmes de bagging, les algorithmes de boosting utilisent des algorithmes de classification dits “faibles” qui performent à peine mieux qu’un classifieur aléatoire.

On entraîne donc un premier classifieur **G1** sur le jeu de données. Comme il effectue un certain nombre d’erreurs, on décide d’augmenter le poids **w** associé aux observations qui induisent en erreur, afin de leur donner plus d’importance dans l’entraînement. On entraîne ainsi un second classifieur G2G2 avec ce nouveau jeu de données pondéré, ce qui lui permet de se concentrer sur ces observations qui induisent en erreur **G1**. On créé de la sorte une suite de classifieurs **G1,G2,...GM** qui s’appuient chacun sur les erreurs effectuées par le classifieur précédent pour repondérer un nouveau dataset sur lequel s’entraîner.



On peut maintenant combiner ces différents classifieurs afin d’obtenir notre classifieur final le plus performant.

**Adaboost**

Pour obtenir ce classifieur final, il nous suffit d’effectuer la somme pondérée des différents classifieurs



Comme un classifieur binaire retourne une réponse binaire, on récupère le signe de cette somme



Il manque quelque chose. C’est dommage de donner autant d’importance à chacun des classifieurs, même ceux qui performant mal au global. Voyons ce qu’on peut faire. Chaque classifieur **Gm** possède son taux d’erreur sur le jeu de données pondéré :



Pour accorder une plus grande importance à ceux qui ont le taux d’erreur global **err** le plus faible, on pondère notre somme par le poids .

Ce poids **αm** est calculé par minimisation séquentielle additive d’une fonction de perte exponentielle    

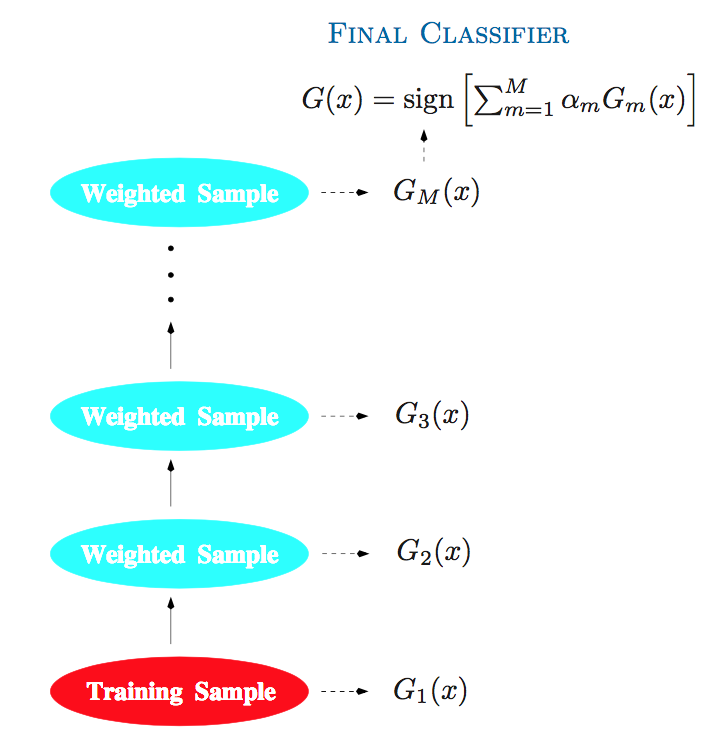
On a donc notre classifieur final



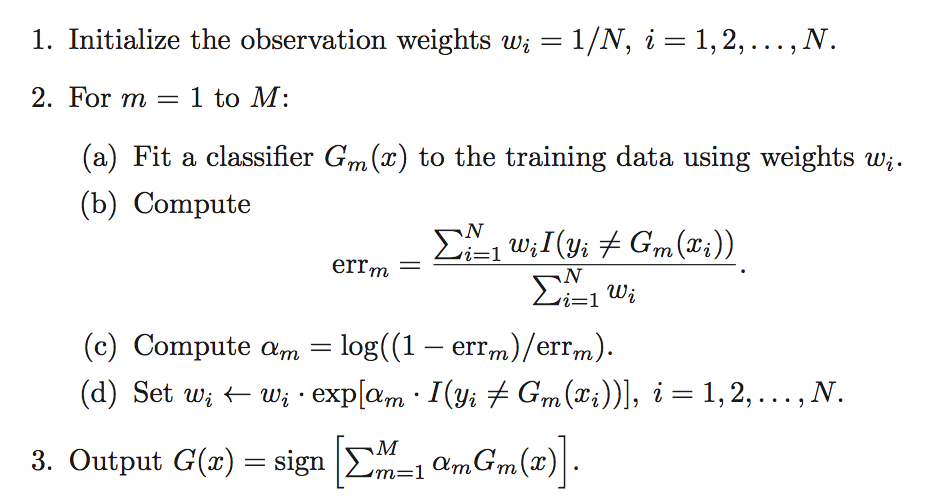
Il nous reste une dernière question... Comment pondérer le jeu de données à chaque étape ? Chaque classifieur est entraîné sur un jeu de donnée pondéré qui accorde plus d’importance aux observations qui ont induit en erreur le classifieur précédent. On va donc utiliser notre poids **αm**. Pour chaque observation **xi**, on pondère par le poids



De la même manière, ces poids sont calculés dans le cadre d’une minimisation *forward stagewise additive modeling*



L’algorithme final du Adaboost pour une classification binaire est donc :



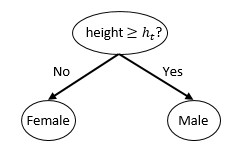
L’Adaboost peut aussi être utilisé en régression ou en classification multi classes, mais le principe reste le même.

Je rappelle les différents choix et hyperparamètres qui caractérisent l’algorithme :

* la fonction de perte utilisée
* le type de classifieur. Le plus utilisé est l’arbre de décision mais on pourrait techniquement utiliser un réseau de neurones ou un SVM.
* la taille M de la séquence de classifieurs.

**Boosting d'arbres aléatoires**

Avec l’Adaboost, on va le plus souvent utiliser nos bons vieux apprenants faibles que sont les arbres aléatoires. Mais on va même faire encore plus simple que ça, on va utiliser ce qu’on appelle en anglais des stumps c’est à dire des arbres qui ne font qu’une seule séparation, avec un seul noeud !



Un stump est un arbre de décision très simple

**Conclusion**

Maintenant qu'on a une meilleure idée de la technique du boosting originale, on peut passer à la plus utilisée de nos jours : le gradient boosting !

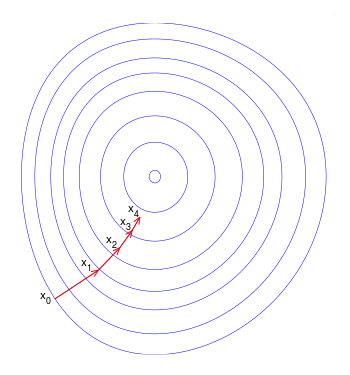
## Décuplez les capacités du boosting : (X)GBoost

Maintenant qu’on a vu comment booster nos arbres afin de les rendre plus performants, nous allons étudier l’algorithme le plus utilisé quand il s’agit d’utiliser des arbres de décision : le **gradient boosting**. C’est en fait une généralisation du boosting dans lequel on va utiliser la fonction de perte de manière similaire à une descente de gradient. A la manière du boosting classique, il n’est pas forcément utilisé avec des arbres de décision même s’ils restent notre classe préférée d’apprenants faibles.

### Intuition du gradient boosting

Le gradient boosting a vu naissance lorsque des chercheurs ont replacé l’algorithme de l’Adaboost dans un cadre statistique plus formel dans lequel il n’était qu’une “simple” optimisation numérique visant à minimiser une fonction de perte de manière successive à l’instar d’une descente de gradient.

Pour rappel, la descente de gradient minimise une fonction objective en suivant la “pente” de cette fonction à l’aide de son gradient ou bien d’une approximation de celui-ci.



Descente de gradient

On va utiliser ici une variation de la descente de gradient appelée **descente de gradient fonctionnelle**, puisqu’on travaille dans un espace fonctionnel lorsqu'on parle des classifieurs de l'algorithme du boosting 😉

En terme statistique, on a appelé cette famille de modèle “stage-wise additive models” a contrario des “stepwise additive models”, parce qu’on fige les apprenants précédents contrairement aux stepwise models dans lesquels on va modifier les apprenants précédents.

Cette généralisation de l’Adaboost à l’aide de ce framework statistique a permis d’utiliser différentes fonctions de pertes différentiables, ce qui a permis plus d’applications que la simple classification binaire mais aussi pouvoir faire des régressions ou encore des classification multi-classes.

### Formulation

Pour résumer, l’algorithme du gradient boosting a besoin de 3 éléments principaux :

* Une fonction de perte à optimiser, qui doit être différentiable, qui permet de résoudre votre problème.
* Un apprenant faible pour effectuer des prédictions. On peut ne plus utiliser uniquement des stumps mais des arbres un peu plus grands, de 4 à 8 niveaux (soyons fous).
* Un modèle additif pour combiner nos apprenants faibles afin de minimiser notre fonction de perte. C’est à dire avancer dans notre fonction de perte en suivant le gradient, ce qui pourra être effectué en ajoutant un arbre de décision supplémentaire. On effectue cette procédure en paramétrant l’arbre, et ensuite en modifiant ces paramètres en allant dans la direction du gradient en diminuant la perte résiduelle que l’on a vue plus haut.

### Contrôle du gradient boosting par les hyperparamètres

Le gradient boosting est un algorithme qui overfit relativement rapidement. On a donc besoin de méthodes de régularisation, qui pénalisent différentes parties de l’algorithme en permettant ainsi de réduire l’overfitting global.

Les différents paramètres qui permettent de limiter l’overfitting sont : **le nombre d’arbres**, **la profondeur des arbres**, **le nombre d’observations utilisées par séparation d’un arbre**, et le **minimum d’amélioration apporté** par chaque nouvelle séparation dans un arbre.

### Fonctions de pertes

Maintenant qu’on peut utiliser n’importe quelle fonction de perte, un monde de possibilités s’offre à nous ! Reste à choisir celle qui correspond à notre problème. On présentera ici les plus courantes.

L’algorithme Adaboost est basé sur une perte dite exponentielle bien qu’encore une fois, la formulation sous forme de modèle additif ait été découverte a posteriori.

#### Utilisation du gradient boosting en classification

En classification, la première fonction de perte que l’on peut utiliser est celle originelle c’est à dire **la perte exponentielle**, qui permet en fait d'obtenir l'algorithme de l'AdaBoost. Cependant, dans un environnement instable/bruyant, on peut préférer utiliser la fonction appelée **déviance binomiale**, beaucoup moins sujet au variation du dataset. Ce sont les deux fonctions de pertes à utiliser pour une classification binaire.

#### Utilisation du gradient boosting en régression

En régression on peut utiliser la perte des moindres carrés classique ou bien la perte de norme 1 aussi classique .

### En pratique

En pratique, on utilisera la fonction de sklearn sklearn.ensemble.GradientBoostingClassifier qui fonctionne de manière très similaire au forêts aléatoires pour la classification et sklearn.ensemble.GradientBoostingRegressor pour la régression.

Les hyperparamètres à optimiser quand on travaille avec le gradient boosting sont **learning\_rate** qui permet le shrinkage pour la régularisation et le nombre de classifieurs **n\_estimators** qui détermine le nombre d'arbres à utiliser. Pour plus de détails sur l'utilisation de ce classifieur, [rendez-vous sur ce petit guide de scikit-learn](http://scikit-learn.org/stable/modules/ensemble.html#gradient-boosting).

Il existe un autre package très utilisé, notamment dans les compétitions Kaggle, qui s'appelle XGBoost [**que je vous encourage à tester**](https://github.com/dmlc/xgboost) afin d'expérimenter ce type d'utilisation sur une recherche de performances tant sur la précision que sur la vitesse de calcul.

### Conclusion

Le gradient boosting est généralement plus complexe à utiliser que les forêts aléatoires en raison du plus grand nombre d'hyperparamètres à gérer. Un second frein est le fait qu'en général il faut effectuer plus de feature engineering ce qui rend l'utilisation moins brute que les forêts aléatoires, qui, elles, fonctionnent mieux en plus grande dimension ([> 4000 dimensions d'après cette étude empirique](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.3232)).

# Partie 2

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Savoir utiliser le package xgboost
* Comprendre le concept de boosting
* Découvrir les méthodes séquentielles

### Question 1

**Les méthodes séquentielles diffèrent des méthodes parallèles...**

* + 

car elles utilisent différents datasets pour créer chacun des classifieurs faibles

* + 

car elles retravaillent les datasets à la suite afin de prendre en compte les performances du modèle précédent

*Les méthodes sont dîtes séquentielles car elles créent des nouveaux apprenants faibles à la suite en prenant en compte l'apprenant précédent*

### Question 2

**Adaboost est un algorithme qui utilise comme classifieur de base...**

* + 

uniquement des arbres de décision

* + 

uniquement des stumps

* + 

tous types de classifieurs classiques (SVM réseaux de neurones, etc.)

* + 

tous les types de classifieurs (techniquement)

*Techniquement adaboost peut utiliser tout type de classifieur mais on utilisera en général des stumps ou des petits arbres de décision*

### Question 3

**Le gradient boosting**

* + 

Utilise la méthode de descente de gradient classique

* + 

Utilise une version modifiée de la descente de gradient

* + 

Techniquement n'utilise pas la descente de gradient

*C'est une pseudo-descente de gradient dans un espace fonctionnel et non pas scalaire comme la méthode originale*

### Question 4

**La descente de gradient utilise différentes fonctions de pertes afin de**

* + 

Pouvoir faire de l'apprentissage supervisé ou non supervisé

* + 

Pouvoir faire de la classification ou de la régression

* + 

Pour pouvoir gérer certains cas spécifiques

*La principale raison d'utiliser une fonction de perte différente est le type de problème (classification ou régression)*

### Question 5

**L'algorithme original de l'Adaboost est en fait un gradient boosting avec une fonction de perte**

* + 

Huber

* + 

Quadratique

* + 

Exponentielle

*On peut retrouver la fonction de perte exponentielle à partir de l'algorithme de l'adaboost en effectuant quelques transformations*