Les algorithmes de Classification: Random Forest

1. Introduction

2. Rappel sur le principe d'arbre de décision et notation mathématique critère de division, critère d'arrêt et élagage

3. Le feature engineering et l'encodage des variables qualitatives

2. Les forêts aléatoires (Random Forest) en théorie

3. Le principe de construction d'une suite emboîtée d'arbres : ensemble learning

5. L'analyse et la validation du modèle : test train split et accuracy score

3. L'importance des hyperparametres avec la librairie scikit-learn

Pré-requis :
- Le cours : La Régression linéaire multiple
- Mathématique niveau terminale S/ES
- De la bonne humeur et de la motivation 🚀

Contexte :

Le but de ce cours est de vous faire comprendre les principes théoriques des arbres de décision, qui permettent d'établir des règles successives permettant de classer des observations ou de faire des régressions. Les arbres de décisions sont une technique particulièrement apprécié dans le domaine du machine learning supervisé car ils sont facilement explicables. On peut les retrouver dans les domaines du risque, du marketing et bien d'autres.

Partie 1. Introduction

TYPE DE BLOC : Introduction

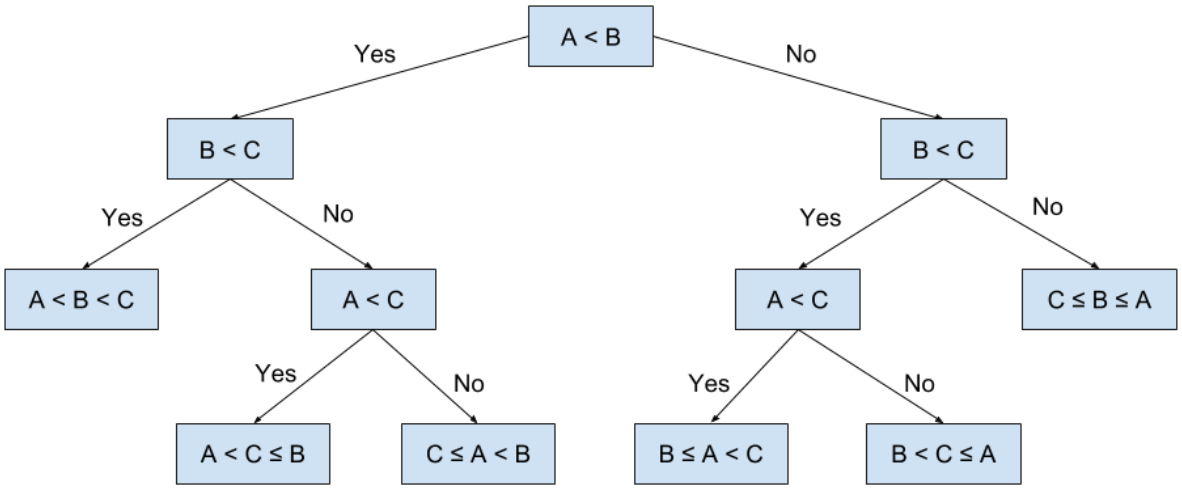
Contenu du bloc : Rappels sur le principe d'arbre de décision, des notations mathématiques, critère de division, critère d'arrêt et élagage de l'arbre.

1.1. Rappel sur le principe d'arbre de décision

Dans un premier temps nous nous intéresserons au Regression Trees qui interviennent dans le cas d'une variable cible quantitative. Leur avantage réside dans leur représentation graphique aisément lisible. L'arbre est composé de trois types d'éléments :

- La racine, où réside l'ensemble des données d'apprentissage.
- Les noeuds/branches, qui représente les points à partir de la racine où les données sont séparées en deux groupes selon un critère lié aux variables explicatives.
- Les feuilles, qui sont les noeuds terminaux de l'arbre et auxquels sont associés un valeur dans le cas où **Y** est quantitative et une classe lorsque **Y** est qualitative.

Ainsi à partir de la racine on défini un noeud qui divise l'ensemble des données selon un critère lié à une variable explicative, pour chacune des deux branches ainsi créées on répète le même procédé, et ainsi de suite jusqu'à ce qu'aucune division ne satisfasse le critère de construction d'un noeud et on définit alors un noeud terminal ou feuille.



Voilà un exemple d'arbre avec en haut la racine et la première division et l'enchaînement des branches, jusqu'aux feuilles. Par ailleurs, on constate immédiatement l'aspect très clair et visuel de l'arbre.

Imaginons que nous avons un programme qui demande trois valeurs à un utilisateur et qui a pour but de classer ces trois valeurs. Pour répondre au problème on utilise un arbre de décision comme celui ci-dessus. Voyons comment lire cette représentation en arbre, pour cela il faut commencer par regarder le noeud tout en haut et vérifier si la condition s'applique. Comme indiqué sur le schéma, si oui nous allons a gauche vers le noeud inférieure vérifier (ou non) la nouvelle condition. Et cela jusqu'à qu'il n'existe plus de noeud inférieur.

Pour se faire cet algorithme a besoin de plusieurs choses :

- Un **critère de sélection de la meilleure division**
- Une règle pour **définir si un noeud est terminal**
- Une **méthode pour assigner à chaque feuille une valeur ou une classe**

1.2. La notation mathématique du problème : critère de division

Comme énoncé dans la partie précédente afin de construire un arbre de décision il nous faut un critère de sélection de la meilleure division pour un noeud.

Un noeud est admissible si les branches qui en découlent portent des noeuds non vide (c'est à dire qu'au moins une observation appartient à chacun des noeuds enfant). Pour un noeud parent contenant M observations il existe M - 1 divisions admissibles si la variables sélectionnée pour la division est quantitative ou qualitative ordinale, et $2^{m-1} - 1$ divisions admissibles si la variable sélectionnée est nominale.

Afin de sélectionner la meilleure division admissible, on construit une fonction d'hétérogénéité qui présente deux propriétés remarquables :

- Elle vaut zéro lorsque tous les individus appartiennent à la même modalité ou présentent la même valeur de Y.
- Elle est maximale lorsque les valeurs de Y sont équiréparties dans le noeud.

On cherche donc la division qui minimise la somme des hétérogénéités des noeuds enfants.

1.3. La notation mathématique du problème : critère d'arrêt

Un noeud donné sera terminal lorsqu'il est homogène, c'est à dire que toutes les observations dans le noeud présente la même valeur ou la même modalité de Y, lorsqu'il n'existe plus de divisions admissibles, ou bien lorsque le nombre d'observations dans le noeud est inférieur à une valeur définie à l'avance, en général de l'ordre de quelques unités.

Dans le cas ou Y est une variable quantitative

Dans le cas de la régression, l'hétérogénéité du noeud k s'écrit de la manière suivante :

$Card(k)$ parfois aussi noté #(k) ou encore $|k|$ est le nombre d'éléments dans le noeud k, et $\bar{y}_k$ est la moyenne des valeurs de Y parmi les observations du noeud k. Ce qui fait la variance du noeud k. La division retenue est celle pour laquelle : $H_{kG} + H_{kD}$ la somme de

$$H_k = \frac{1}{Card(k)} \cdot \sum_{i \in k} (y_i - \bar{y}_k)^2$$

l'hétérogénéité de la branche gauche et de la branche droite est minimale. Car rappelons le l'objectif est de partager les individus en deux groupes les plus homogènes par rapport à notre variable cible.

Dans le cas ou Y est une variable qualitative

Soit Y une variable qualitative à **m** modalités ou catégories numérotées de 1 à m. La fonction d'hétérogénéité privilégiée la plupart du temps est la concentration de GINI qui se note de la façon suivante :

$$H_k = \sum_{i=1}^{m} p_k^i \cdot (1 - p_k^i)$$

Où p_k^i est la proportion de la classe i de Y dans le noeud k.

Comme précédemment, il s'agit pour chaque nœud de rechercher, parmi les divisions admissible, celle qui maximise la décroissance de l'hétérogénéité.

1.4. L'élagage d'un arbre

Le problème des modèles LASSO et RIDGE est qu'il y a un risque important comme dans tout problème d'apprentissage supervisé est celui du sur-apprentissage. Le sur-apprentissage signifie que votre algorithme c'est tellement entraîné sur vos données qu'il perd sa capacité à généraliser ses prédictions. Le critère d'arrêt défini pour la construction de l'arbre est souvent propice au sur-apprentissage puisqu'il est très probables que la plupart des feuilles de l'arbre ne contiennent que quelques observations.

Ainsi l'arbre de décision tel quel sera très instable, son biais est quasi-nul voir nul par définition, il dépend très fortement des observations de la base d'apprentissage et sera potentiellement peu généralisable aux nouvelles données.

Le problème est donc de trouver un arbre intermédiaire qui vérifie un compromis biais variance intéressant pour les besoins de l'estimation de Y. C'est pour cela que nous allons voir les forets (en ensemble d'arbres) qui vont répondre à ce problème, c'est ce que nous allons voir dans la partie suivante.

1.5. Le feature engineering et l'encodage des variables qualitatives

Tous les modèles que l'on utilise avec les librairies (scikit-learn, tensorflow, pytorch ...) procèdent à ce qu'on appelle de l'optimisation numérique, c'est à dire qu'ils font des calculs statistiques sur les données qu'on leurs fournit en input.

Seulement, **pour effectuer des calculs numériques, ces variables doivent être transformés numériquement.** Cela devient problématique lorsque l'on utilise des variables qualitatives et non ordinales (date, lieu, texte, ...). Heureusement, il existe diverses méthodes afin d'encoder ces variables de manière numérique.

1.5.1 L'encodage par dictionnaire

L'encodage par dictionnaire est fréquemment utilisé en NLP pour assimilier un mot particulier à un indice.

Pour illustrer l'exemple d'un encodage par dictionnaire nous utiliserons ici la fonction **LabelEncoder()** de la librairie scikit-learn. Par exemple, si on considère la phrase le chat adore le poisson, alors la phrase pourra être encodée par 0 1 2 0 3.

```
import numpy as np
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

X = [ "le chat adore le poisson", "le chien aime la promenade", "le cheval adore galoper" ]
label_encoder = LabelEncoder()
X_labels = label_encoder.fit_transform(" ".join(X).split(" "))
X_labels
```

```
array([7, 2, 0, 7, 8, 7, 4, 1, 6, 9, 7, 3, 0, 5])
```

Dans le cas du texte, il existe beaucoup de méthodes pour encoder numériquement les variables : par fréquence ou codage, par TF-IDF, par dictionnaire ou encore par word embedding. Ci-dessous un autre exemple d'utilisation de la fonction **LabelEncoder()** avec cette fois des redondances dans l'input.

```
X = [ "bleu", "rouge", "vert", "rouge", "jaune", "orange", "bleu", "vert", "rouge", "rouge", "orange" ]
label_encoder = LabelEncoder()
X_labels = label_encoder.fit_transform(" ".join(X).split(" "))
print("Classes :", label_encoder.classes_)
print("Encodage par labels :", X_labels)
```

```
Classes : ['bleu' 'jaune' 'orange' 'rouge' 'vert']
Encodage par labels : [0 3 4 3 1 2 0 4 3 3 2]
```

1.5.2 L'encodage par vecteur one-hot

Un autre type d'encodage, particulièrement utilisé en NLP (traitement du langage naturel) est l'encodage one-hot.

Le principe est le suivant : on dispose d'une variable à modalités (pouvant prendre valeurs différentes) identifiées de à . Pour représenter numériquement une observation de cette variable, on crée un vecteur de taille dont toutes les composantes sont nulles, sauf une composante qui vaut et qui correspond à la ¯ème modalité.

Par exemple, considérons quatre villes : Paris, Marseille, Nice et Bordeaux. On dispose d'une observation dont la valeur est Nice. Dans ce cas, le vecteur one-hot associé à cette observation est :

$$(0 \quad 0 \quad 1 \quad 0)$$

Car Nice est bien la troisième modalité de la variable ville.

```
X = np.asarray([ ["Paris"], ["Marseille"], ["Paris"], ["Nice"], ["Bordeaux"], ["Bordeaux"], ["Marseille"] ])
# Création de l'objet Label Encoder / l'option .ravel() signifie que X est un array en 1 dimension
label_encoder = LabelEncoder()
X_labels = label_encoder.fit_transform(X.ravel())
print("Classes :", label_encoder.classes_)
print("Encodage par labels :", X_labels)
# Redimensionnement matriciel
X_labels = X_labels.reshape(len(X_labels), 1)
# Objet One Hot Encoder
one_hot_encoder = OneHotEncoder(sparse=False)
print("Encodage one-hot :")
print(one_hot_encoder.fit_transform(X_labels))
```

```
Classes : ['Bordeaux' 'Marseille' 'Nice' 'Paris']
Encodage par labels : [3 1 3 2 0 0 1]
Encodage one-hot :
[[0. 0. 0. 1.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]]
```

Partie 2. Les forêts aléatoires (Random Forest) en théorie

TYPE DE BLOC : Définition

Contenu du bloc : Explication du principe de constructions d'une forêt d' arbres et introduction au principe d'apprentissage par ensemble.

Afin de bien comprendre le principe de forêt aléatoire commençons par définir deux choses la méthode Boostrap ou le Bootstraping et l'apprentissage par ensemble.

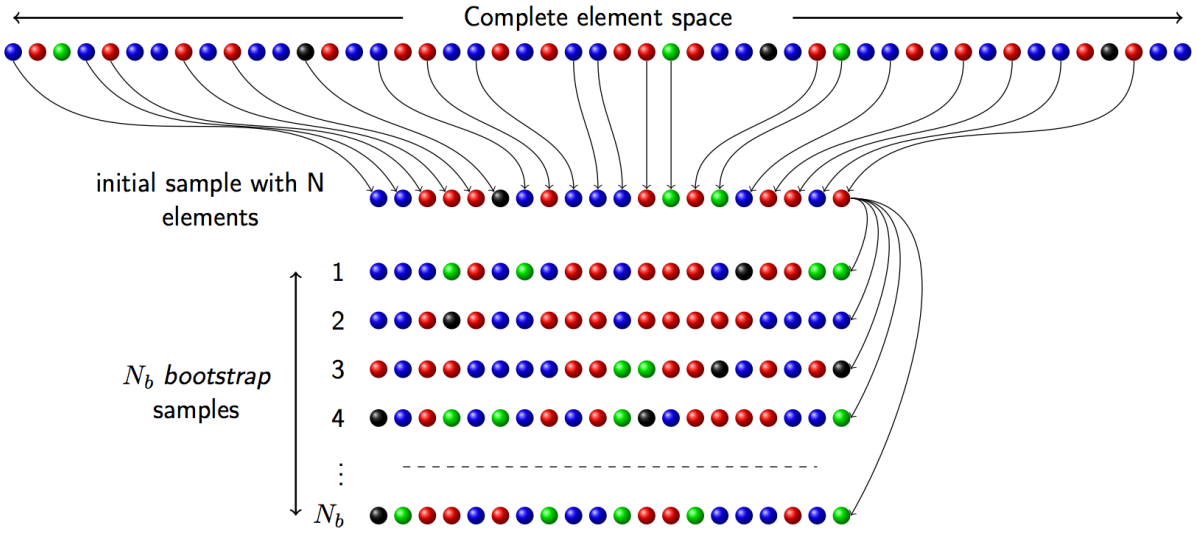
Commencons par le Bootstrapping; la méthode Boostrap est un procédé qui permet d'augmenter artificiellement le nombre d'observation d'un échantillon de données sans pour autant modifier la distribution des variables présentes dans le jeu de données. Le principe est simple, on dispose d'un jeu de données contenant n observations, pour créer un échantillon de taille m on tire avec remise m observations parmi le jeu de données original

Chaque observation du jeu de données original a $\frac{1}{n}$ chance d'être tiré (c'est un tirage avec remise équiprobable).

L'équiprobabilité du tirage est essentielle afin que la loi de distribution de l'échantillon soit la même que celle de la base initiale.

Un exemple avec le schéma ci-dessous :

Ou l'ensemble de la population est représentée par toute les billes de couleurs (complete élément space). On construit un premier sample (échantillon) de valeurs dans la rangée de bille du dessous et enfin on utilise cet échantillon pour créer N_b nouveaux échantillons.



2.1. Ensemble learning

Le principe de l'ensemble Learning (l'apprentissage par ensemble en français) repose sur un principe simple : « l'union fait la force ».

De façon plus concrète l'apprentissage par ensemble regroupe plusieurs modèles appelés estimateurs faibles (weak learner en anglais) et sont tous entraînés a résoudre le même problèmes pour produire de meilleurs résultats. L'hypothèse principale derrière ce raisonnement est que quand des estimateurs faibles sont correctement agencés nous pouvons obtenir de bien meilleurs résultats qu'avec un unique modèle.

On appelle estimateur faible, un estimateur (ou modèle) qui peut être utilisé dans une construction de modèles plus complexes (un modèle formé de plusieurs estimateurs faibles). Souvent un estimateur faible va être moins performant soit parce qu'ils ont un biais élevé (modèles avec peu de variables explicatives, par exemple) soit parce qu'ils ont trop de variance pour être robustes (modèles avec trop de variables explicatives, par exemple).

L'idée des méthodes d'apprentissage par ensemble est d'essayer de réduire le biais et/ou la variance de ces estimateurs faibles en combinant plusieurs d'entre eux afin de créer un estimateur fort (ou modèle d'ensemble) qui obtient de meilleures performances.

Maintenant que vous avez compris l'idée, on peut se poser la question de comment combiner des estimateurs pour former un modèle d'ensemble. Pour cela, on parle généralement de trois type d'apprentissage par ensemble :

- **Bagging**, technique qui consiste souvent à prendre des estimateurs faibles et homogènes qui apprennent indépendamment les uns des autres en parallèle et les combine en suivant une statistique comme la moyenne des estimations.
- **Boosting**, technique qui consiste souvent à prendre des estimateurs faibles et homogènes, qui apprennent séquentiellement de manière adaptative (modèle de base qui dépend des précédents) et les combine selon une certaine stratégie.
- **Stacking**, technique qui consiste souvent à prendre des estimateurs faibles et non homogènes, qui apprennent en parallèle et combine les estimateurs en entraînant un méta-estimateurs (ou meta-modèle) pour produire une prédiction basée sur les différentes prédictions des estimateurs faibles.

Nous n'allons pas rentrer dans les détails mathématiques de ces méthodes dans ce cours. Par ailleurs, on peut dire de façon imagée que le bagging se concentrera principalement sur l'obtention d'un modèle avec moins de variance alors que le boosting et le stacking essaieront principalement de produire des modèles forts moins biaisés (même si la variance peut également être réduite).

La première idée derrière les random forest est d'effectuer un bagging de plusieurs arbres aléatoires. Plusieurs élagages des arbres ainsi construits sont possible :

- On peut conserver les arbres complets et éventuellement limiter le nombre minimum d'observations au niveau des noeuds terminaux.
- Conserver au plus **q** feuilles ou limiter la profondeur de l'arbre à **q** niveaux de noeuds.

En général on retiendra la première stratégie, car elle représente un bon compromis entre qualité d'estimation et quantité de calculs. Chaque arbre ainsi construit aura un biais très faible et une grande variance, cependant le fait d'agréger les modèles entre eux participe justement à réduire cette variance.

Cet algorithme est peu complexe à mettre en place, ce qui est un grand avantage, cependant le nombre de modèles à calculer avant que l'erreur de test (appelée aussi erreur de validation qui va mesurer la performance de notre modèle) se stabilise peut être très important.

Le modèle final sera volumineux en terme d'espace disque car il est nécessaire de stocker la structure complète de tous les arbres pour pouvoir faire des prévisions. Enfin la multiplication du nombre d'arbres participants au modèle rend plus difficile, voir impossible l'interprétation du modèle comme cela était possible avec un seul arbre.

La seconde idée consiste à améliorer la méthode du bagging afin de créer des random forest s'appuyant sur des échantillons de données les plus "indépendants" possible. Non seulement l'aléatoire intervient lors de la sélection des observations lors de la construction des échantillons d'apprentissage, mais on fera interviendra également dans le choix des variables explicatives retenues pour chaque échantillon sur lequel on construira un arbre aléatoire ce que nous allons détailler plus tard.

Ce double appel à l'aléatoire pour la sélection des observations et des variables explicatives a plusieurs avantages :

- Il permet de s'approcher de l'hypothèse d'indépendance des échantillons,
- Il réduit le nombre de calculs à effectuer pour le construction de chaque arbre
- Il réduit les risques d'erreurs liés à d'éventuelles corrélations entre variables explicatives.

2.2. Construire une forêt aléatoire

Les random forest (en français, forêts d'arbres aléatoires) s'inscrivent dans le champ des méthodes de partitionnement récursif, qu'on connait plutôt sous l'acronyme CART (Classification And Regression Tree). On rappelle que nous parlons de classification lorsque la variable cible est qualitative (catégorielle). Tout comme les CART les Random Forest peuvent estimer des variables quantitatives et qualitatives.

L'algorithme de construction de l'arbre peut se structurer en trois étape de la façon suivante :

Étape 1 : créer un bootstrap dataset

Comme vu dans la partie précédent sur la méthode de bootstrap on utilise cette méthode pour générer des échantillons de données à partir de notre jeu de donnée originale.

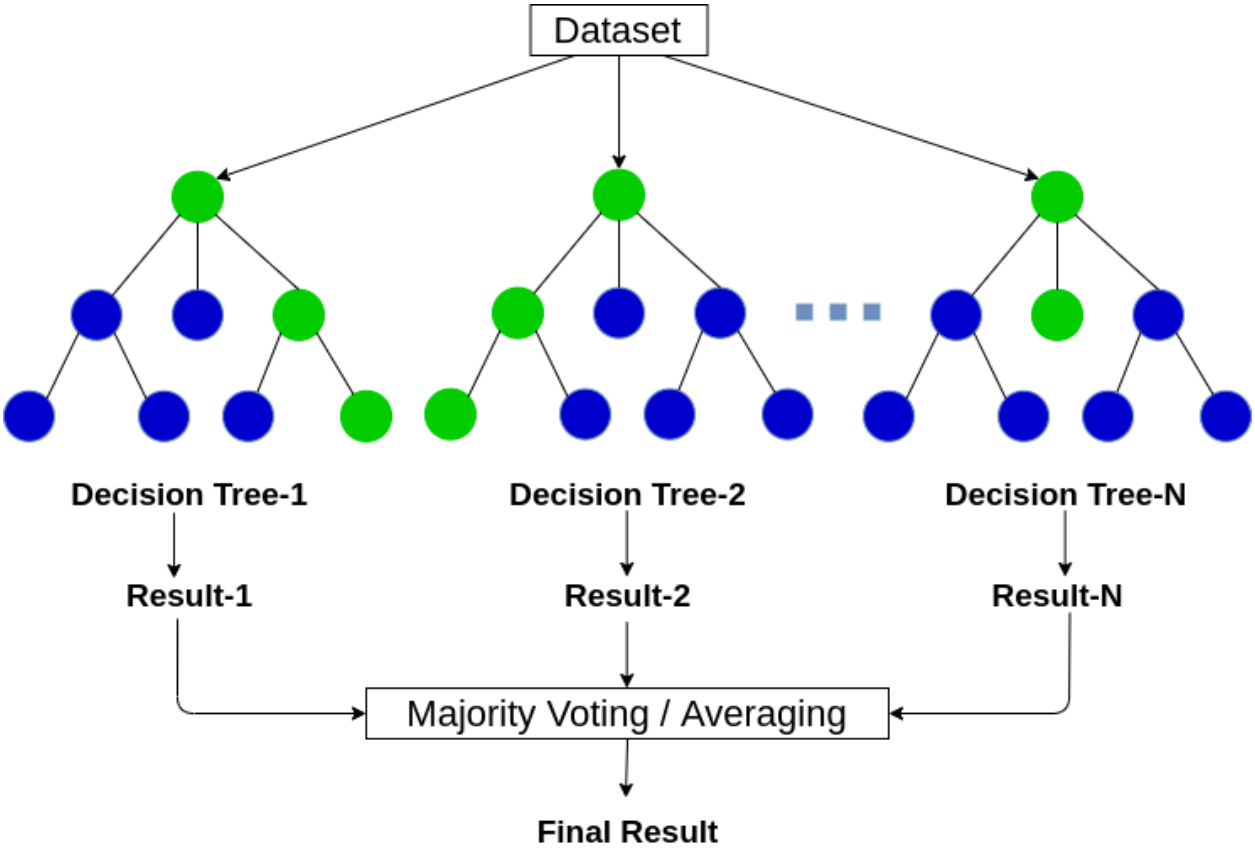
Étape 2 : créer un arbre de décision en utilisant le bootstrap dataset

Chaque arbre crée à partir du bootstrap dataset pourra utiliser (aléatoirement) des variables explicatives différentes à chaque nouvel itération.

Étape 3 : répéter les étapes 1 et 2 un nombre limité de fois

Ci-dessous une représentation de trois arbres formés à partir du même jeu de données qui donnent trois résultats différents.

La décision finale sera ensuite un vote parmi les arbres qui compose la forêt ou bien une moyenne de ces derniers.

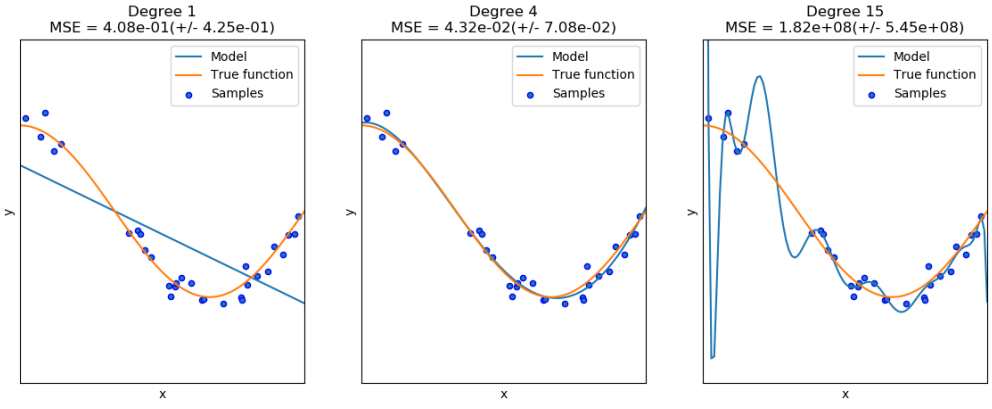


2.3. L'analyse et la validation du modèle

Dans cette partie nous allons voir des concepts essentiels dans le domaine du machine learning : le principe de sur/sous apprentissage ainsi que le spliting.

Les notions de sur et sous apprentissage qui s'expliquent ainsi :

- Le **sous-apprentissage** est le fait qu'un modèle soit trop simple pour être une bonne estimation de la variable cible.
- Le **sur-apprentissage** est le phénomène inverse, lorsqu'on construit un modèle très complexe qui adhère parfaitement aux données d'apprentissage, mais inutile en pratique car il est très improbable qu'il se généralise bien à de nouvelles données inconnues.



La figure ci-dessus représente de gauche à droite, une situation de sous-apprentissage, une situation de bonne estimation, et une situation de sur-apprentissage. En effet, le modèle de la figure de droite « colle beaucoup trop aux données » il ne représente pas assez bien l'allure générale de l'ensemble de point.

La question qu'on peut se poser maintenant est la suivante : comment se protéger du sur-apprentissage ?

2.3.1. Le principe du test train split, de la k-fold cross validation et de l'accuracy score

Dans les parties précédentes nous avons parlé d'erreur de test ou d'erreur de validation qui vont mesurer la performance de notre modèle. Nous allons donc voir en détail dans cette partie comment mesurer la performance d'un modèle.

Les situations de sous-apprentissage interviennent peu en pratique, ou elles sont souvent dues à un manque de données pertinentes ou d'autres problèmes qui ne peuvent pas être réglés directement par les data sciences. Le véritable ennemi des data scientists est le sur-apprentissage, car il donne l'illusion de la performance, mais est en réalité un piège et. Vous allons voir comment ne pas tomber dedans.

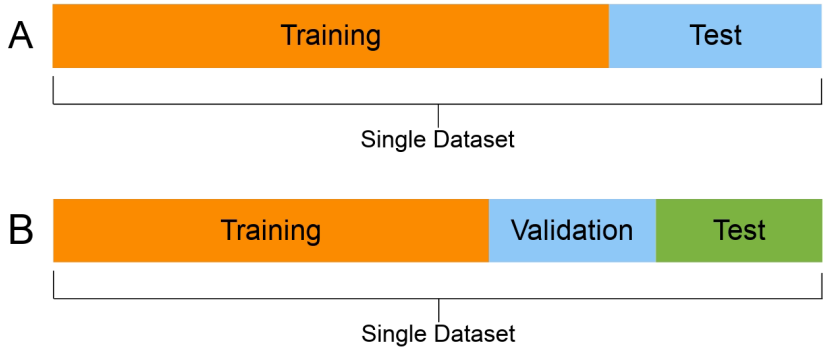
Le principe du test train split

Le mot split signifie diviser/partager en anglais, le principe de split de données c'est simplement la division d'un ensemble de données en plusieurs parties distincts. Il est dans la coutume d'entrainer un modèle à l'aide d'une partie et de tester son efficacité sur une autre partie.

Il est courant de voir ce processus de spliter en 3 parties :

- Les données que nous utilisons pour concevoir un (ou plusieurs) modèles **Training dataset**
- Les données que nous utilisons pour affiner un (ou plusieurs) modèles : **Validation dataset**
- Les données que nous utilisons pour tester un (ou plusieurs) modèles **Testing set**

On peut illustrer ce partage à l'aide de la figure ci-dessous :



Ci dessous un exemple avec des données de la Princeton University à propos de la discrimination salariale entre homme et femme. Vous pouvez accéder aux données avec le lien suivant : https://data.princeton.edu/wws509/datasets/salary.dat

On importe les données avec la libraire Pandas et on utilise la fonction **train_test_split()** de scikit-learn tel que :

```
from sklearn.model_selection import train_test_split
#définition des features et de la target
features = ['sx', 'rk', 'yr', 'dg', 'yd']
target = ['sl']
#split
x_train, x_test, y_train, y_test = train_test_split(df[features],df[target],test_size=0.2)
```

Cette méthode un peu particulière car elle retourne 4 variables et plus précisément 4 objet Dataframe qui correspondent à :

- **x_train** : les variables explicatives (features) pour entrainer votre modele
- **y_train** : la variable cible (target) sur laquelle votre modele essaie de faire des prédictions
- **x_test** : les variables explicatives (features) pour tester votre modele (il ne les connait pas, il ne s'est pas entrainer avec)
- **y_test** : la variable cible (target) sur laquelle votre modele va se tester (il ne la connait pas, il va donc tester ses prédictions)

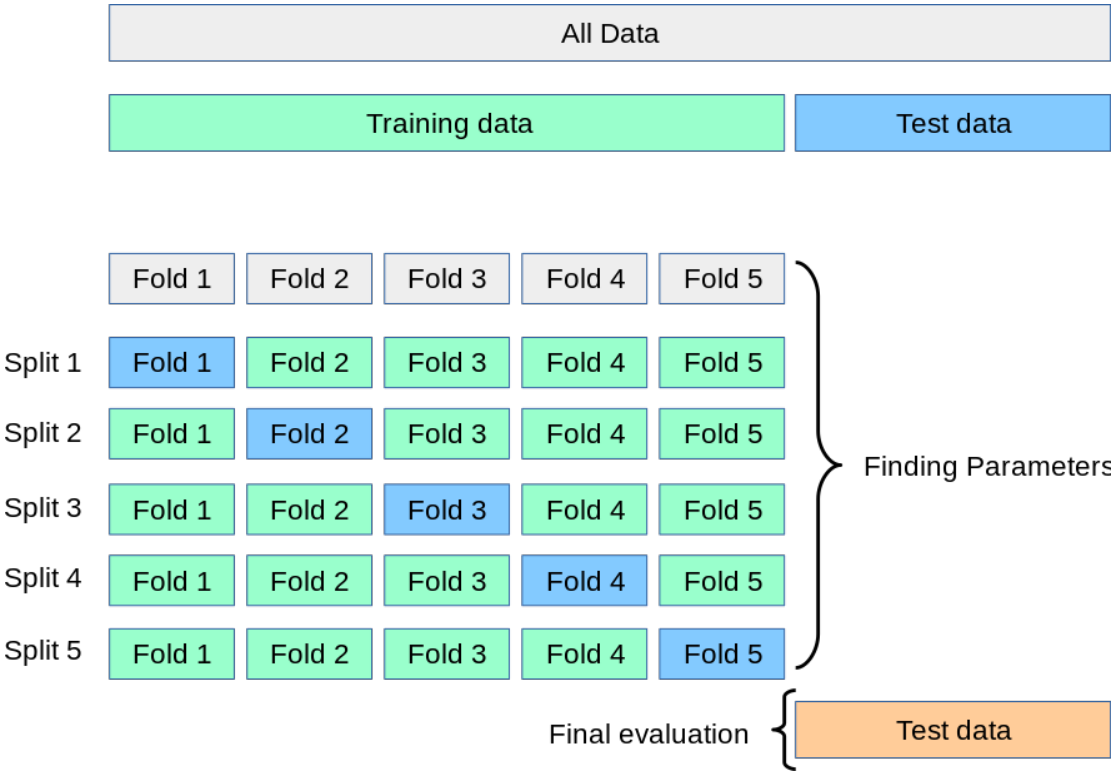
Quant au paramètre **test_size** il signifie la taille de votre échantillons de test, ici il est de 0.2 soit 20% des données initiales.

La k-fold cross validation

Une manière simple et efficace de se garantir de ce piège est de pratiquer la cross-validation (ou **k-fold cross validation**). C'est un procédé qui consiste à choisir un entier k (souvent on choisit 10 par défaut), on répartit les observations au hasard dans k groupes de taille égale. Puis on répète k fois la méthode suivante :

- On isole un groupe i parmi 10 groupes, qu'on appellera base de test, et on rassemble les 9 autres, qu'on appellera base d'apprentissage.
- On estime le modèle choisi à l'aide de la base d'apprentissage.
- On calcule l'erreur commise par le modèle i sur la base de test (le groupe i) que l'on compare à l'erreur commise sur la base d'apprentissage après optimisation.

La comparaison de l'erreur d'apprentissage et l'erreur de test permet de comprendre le réel pouvoir explicatif d'un modèle, car elle quantifie la performance du modèle sur des données inconnues par rapport à sa performance sur des données connues.



La figure ci dessus illustre le principe de la k-fold cross-validation. Chaque itération produit des résultat en termes d'erreurs de test et d'apprentissage dont on se sert pour évaluer le modèle.

Pour calculer ces erreurs on se base en général sur la fonction de coût qu'on a choisit pour optimiser le modèle, ou bien tout simplement la moyenne des erreur au carré. En général on s'attend à ce que l'erreur de test et de validation soient du même ordre de grandeur, et on espère que l'erreur de manière générale sera petite par rapport aux valeurs prises par la variable cible.

L'accuracy score

L'accuracy score est une mesure de la performance de votre algorithmes sur des données qu'il n'a jamais rencontré auparavant.

Le score d'accuracy indique le pourcentage de bonnes prédictions de votre algorithme sur l'ensemble de données de test. C'est un indicateur tres apprécié car il est assez simple à comprendre.

Ce n'est évidement pas le seul indicateur que nous pouvons développer pour analyser nos modèles. Nous verrons dans un cours dédié tout les outils pour analyser les résultats d'un modele de machine learning.

Avantages des forêts aléatoires

1. Ils utilisent l'apprentissage par ensemble, de cette façon on peut réduire le problème du sur-apprentissage (lorsqu'on construit un modèle très complexe qui adhère parfaitement aux données d'apprentissage, mais inutile en pratique car il est très improbable qu'il se généralise bien à de nouvelles données inconnues)

2. Ils peuvent être utilisé pour résoudre à la fois des problèmes de classification et de régression, ce qui est assez pratique.

3. Ils fonctionnent bien avec les variables catégoriques et continues, ce qui n'est pas le cas de tout les algorithmes.

4. Ils peuvent gérer automatiquement les valeurs manquantes.

5. Aucun scaling (mise à l'échelle des données) des variables n'est requis : aucune normalisation n'est requise dans le cas des Random Forest car ils utilisent une approche basée sur des règles de décision et non un calcul de distance.

6. Ils sont généralement résistants aux valeurs aberrantes et peuvent les gérer automatiquement.

Désavantages des forêts aléatoires

1. La complexité : ils nécessitent beaucoup de puissance de calcul car l'algorithme construit plusieurs arbres pour combiner leurs sorties.

2. Le temps d'entrainement : ils nécessitent beaucoup de temps pour s'entrainer du au nombre d'arbres à entrainer.

3. L'interprétabilité : encore une fois en raison de l'ensemble d'arbres, il est compliqué de déterminer la signification de chaque classe contrairement au arbre de décision simple ou la lecture de l'arbre est claire.

QUIZ 1

Question 1 - dans le cas d'une variable cible qualitatives il s'agit pour chaque nœud de rechercher, parmi les divisions admissible, celle qui maximise la décroissance de l'hétérogénéité:

vrai

faux

Explication : Comme dans le cas d'une variable quantitative, il s'agit pour chaque nœud de rechercher, parmi les divisions admissibles, celle qui maximise la décroissance de l'hétérogénéité.La fonction d'hétérogénéité privilégiée la plupart du temps est la concentration de GINI (cf formule partie 1.3.)

Question 2 - L'encodage de variable dans le cas du machine learning c'est le fait de transformer une variable quantitative en variable qualitative ?

vrai

faux

Explication : L'encodage c'est l'ensemble des techniques pour formaliser l'information afin de pouvoir la manipuler, la stocker ou la transmettre. Dans notre cas il s'agit de transformer une variable qualitative en variable quantitative afin qu'elle puisse être traitée par un algorithme.

Question 3 - La racine d'un arbre de décision est-elle la première division ?

vrai

faux

Explication : En effet, la racine est la première division, là où réside l'ensemble des données d'apprentissage.

Question 4 - L'encodage par méthode one hot et par dictionnaire sont-elles deux méthodes équivalentes ?

vrai

faux

Explication : Ce sont deux méthode différentes car elle ne propose pas la même façon d'encoder une même information.

Question 5 - Le nombre de noeuds (ou le nombre d'étages) d'un arbre de décision n'influence pas ses performances ?

vrai

faux

Explication : En effet, le critère d'arrêt défini pour la construction de l'arbre est souvent propice au sur-apprentissage (cela signifie que votre algorithme s'est trop entraîné sur vos données qu'il perd sa capacité à généraliser ses prédictions, ce qui n'est pas bon) puisqu'il est très probable que la plupart des feuilles de l'arbre ne contiennent que quelques observations.

Partie 3. L'importance des hyperparametres avec la librairie scikit-learn

TYPE DE BLOC : Définition

Contenu du bloc : Explication du principe de recherche d' hyperparametres dans une forêt aléatoire et manipulation d'un RandomizedSearchCV.

3.1. L'importance de la variation d' un hyperparamètre

Comme on l'a vu précédemment, afin de bien démarrer les Random Forest, il est nécessaire de comprendre dans le détail le fonctionnement du processus d'optimisation (de calibrage) des arbres de décisions. En effet, rappelons que l'idée derrière les Random Forest est une méthode d'ensemble (de paquets), c'est-à-dire qu'elle se base sur plusieurs arbres afin d'obtenir un modèle complexe.

Commençons par un exemple afin de comprendre le fonctionnement d'un hyperparametre dans le cas d'un arbre de régression : la simulation d'une fonction sinus. Si cela vous rappelle des mauvais souvenirs d'école avec des notions abstraites. Sachez que les fonctions sinusoïdales sont partout dans ce qui nous entoure comme le déplacement des ondes, la simulation de vibrations pour tester la résistance des matériaux et bien d'autres choses encore.

Notre tache va donc être de construire un arbre de décision et plus précisément un CART afin de prédire une fonction sinus dans l'intervalle [0;5] en faisant varier un hyperparamètre (le paramètre de profondeur de l'arbre)

Pour cet exemple, nous simulons une fonction sinus avec des bruits aléatoires et indépendants à l'aide de la librairie Numpy de la façon suivante :

```
# le random dataset
rng = np.random.RandomState(1)
X = np.sort(5 * rng.rand(80, 1), axis=0)
y = np.sin(X).ravel()
# le bruit ajouté
y[::5] += 3 * (0.5 - rng.rand(16))
```

Nous allons créer arbres de régression ayant une profondeur différente (respectivement 2, 3, 4 et 6). Pour chacun des arbres :

- On crée l'objet scikit-learn permettant d'effectuer la régression par un arbre de décision
- On optimise l'arbre sur nos données
- On prédit l'arbre sur des données simulés

```
# Fit regression model
regressors = []
depths = [2, 3, 4, 6]
X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]
Y_test = []
for p in depths:
    reg = DecisionTreeRegressor(max_depth=p)  # Création d'un arbre de régression de profondeur p
    reg.fit(X, y)  # Processus d'optimisation de l'arbre
    Y_test.append(reg.predict(X_test))  # On prédit sur les données et on ajoute à la liste des val
eurs prédites
    regressors.append(reg)
```

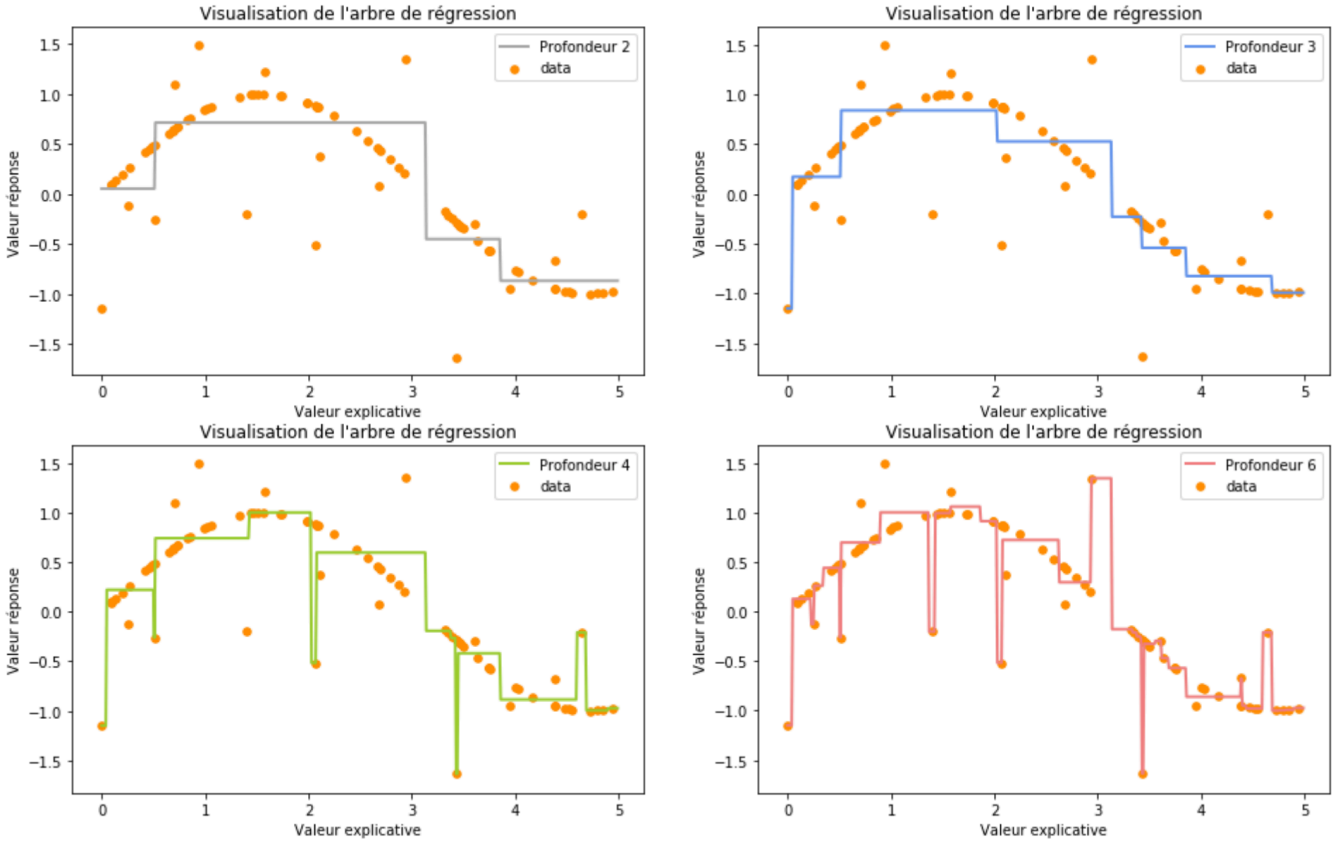
Enfin, nous pouvons afficher les données d'entrées et les valeurs prédites pour chaque arbre.

```
plt.figure(figsize=(16, 10))
colors = [ "darkgray", "cornflowerblue", "yellowgreen", "lightcoral"]

for i, reg in zip(range(len(regressors)), regressors):
    plt.subplot(220 + i + 1)
    plt.plot(X_test, Y_test[i], color=colors[i], label="Profondeur {0}".format(depths[i]), linewidt
h=2)
    plt.scatter(X, y, s=30, c="darkorange", label="data")
    plt.xlabel("Valeur explicative")
    plt.ylabel("Valeur réponse")
    plt.title("Visualisation de l'arbre de régression")
    plt.legend()

plt.show()
```

Ce qui nous donne les graphiques suivants :



Analyse des graphiques

Nous n'allons pas revenir sur le principe de coupure ici. Cet exemple est la pour vous illustrer l'effet d'un hyperparametre (la profondeur de notre arbre) sur le sur-apprentissage.

3.2. Random Search

Nous avons maintenant une vague idée de l'influence des hyperparamètres sur notre model. On peut donc imaginer que trouver le meilleur modele c'est trouver les hyperparamètres optimaux sur nos données.

La question qu'on se pose maintenant est la suivante : comment trouver les hyperparamètres optimaux ? On peutessayer des les rechercher sur un certain intervalle ou plage de valeur (une liste ou un dictionnaire en pratique) et évaluer le résultat de notre algorithme pour chaque hyperparamètre / combinaison d'hyperparamètres.

Nous allons pour cela utiliser la méthode **RandomizedSearchCV()** de scikit-Learn, nous pouvons définir une grille de plages d'hyperparamètres et sélectionner au hasard à partir de la grille, en effectuant un K-FoldCV (K-Fold Cross Validation) avec chaque des combinaisons de valeurs.

On va illustrer cette exemple avec avec des données de la Princeton University à propos de la discrimination salariale entre homme et femme. Vous pouvez accéder aux données avec le lien suivant : https://data.princeton.edu/wws509/datasets/salary.dat

On va commencer par encoder les variables catégoriques avec la fonction **LabelEncoder()** tel que :

```
sx_label_encoder, rk_label_encoder, dg_label_encoder = LabelEncoder(), LabelEncoder(), LabelEncoder()
df.rk=rk_label_encoder.fit_transform(df.rk)
df.dg=dg_label_encoder.fit_transform(df.dg)
df.sx=sx_label_encoder.fit_transform(df.sx)
df.head()
```

	sx	rk	yr	dg	yd	sl
0	1	2	25	0	35	36350
1	1	2	13	0	22	35350
2	1	2	10	0	23	28200
3	0	2	7	0	27	26775
4	1	2	19	1	30	33696

```
from sklearn.model_selection import train_test_split
#définition des features et de la target
features = ['sx', 'rk', 'yr', 'dg', 'yd']
target = ['sl']
#split
x_train, x_test, y_train, y_test = train_test_split(df[features],df[target],test_size=0.2)
```

On va ensuite procéder à un split :

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(random_state = 42)
from pprint import pprint
#affichage des parametres par defaut
print('Parameters currently in use:\n')
pprint(rf.get_params())
```

```
Parameters currently in use:

{'bootstrap': True,
 'ccp_alpha': 0.0,
 'criterion': 'mse',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n estimators': 100,
```

On va commencer par regarder les paramètres par défauts du modele de random Forest avec la fonction **get_params()**.

```
rf_ex.best_params_
```

```
{'bootstrap': False,
 'max_depth': 40,
 'max_features': 'auto',
 'min_samples_leaf': 2,
 'min_samples_split': 10,
 'n_estimators': 12}
```

```python
from sklearn.model_selection import RandomizedSearchCV
#nombre d'arbre dans la random forest
n_estimators = [int(x) for x in np.linspace(start = 2, stop = 100, num = 10)]
#nombre de features dans chaque split
max_features = ['auto', 'sqrt']
#profondeur : nombre de niveau dans l'arbre
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
#nombre minimum de sample pour faire un split
min_samples_split = [2, 5, 10]
#nombre minimum de sample pour former une feuille
min_samples_leaf = [1, 2, 4]
#methode de selection de sample pour chaque arbre
bootstrap = [True, False]

#creation du grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap
              }

pprint(random_grid)
```

```
{'bootstrap': [True, False],
 'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, None],
 'max_features': ['auto', 'sqrt'],
 'min_samples_leaf': [1, 2, 4],
 'min_samples_split': [2, 5, 10],
 'n_estimators': [2, 12, 23, 34, 45, 56, 67, 78, 89, 100]}
```

```
#créons le modele de random forest
rf = RandomForestRegressor(random_state = 42)
#random search of parameters, en utilisant 3 fold cross validation
#recherche dans 100 differentes combinaisons et utilisation de tout les core du processeur
rf_ex = RandomizedSearchCV(estimator=rf, param_distributions=random_grid,
                           n_iter = 100, cv = 3, verbose=2, random_state=42, n_jobs=-1,
                           return_train_score=True)

#fit du modele
target = np.array(y_train)
rf_random.fit(x_train.values,target.ravel());
```

```
Fitting 3 folds for each of 100 candidates, totalling 300 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  58 tasks      | elapsed:    1.8s
[Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed:    7.9s finished
```

```
rf_ex.best_estimator_
```

```
RandomForestRegressor(bootstrap=False, ccp_alpha=0.0, criterion='mse',
                      max_depth=40, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=2,
                      min_samples_split=10, min_weight_fraction_leaf=0.0,
                      n_estimators=12, n_jobs=None, oob_score=False,
                      random_state=42, verbose=0, warm_start=False)
```

On va ensuite créer un random_grid qui va nous permettre de rechercher les hyperparametre optimaux tel que :

\On va maintenant utiliser le random grid pour chercher ces hyperparametre optimaux avec le module **RandomizedSearchCV()**

```
from sklearn.metrics import r2_score
y_pred = rf_ex.predict(x_test)
r2_score(y_test, y_pred)
```

```
0.8068959309342355
```

Les forêts aléatoires n'ont plus de secrets pour vous maintenant 😎

QUIZZ

1 - dans le cas d'une variable cible qualitatives il s'agit pour chaque nœud de rechercher, parmi les divisions admissible, celle qui maximise la décroissance de l'hétérogénéité:

vrai

faux

(explication : voir partie 1.3.)

2 - Dans la partie 3.1. à votre avis parmi les 4 figures laquelle correspond le plus à un sur-apprentissage ?

- 1
- 2
- 3
- **4**

(explication : la dernière figure en bas à droite est trop proche des points comme expliqué dans la partie 3.1.)

3 - Dans la partie 3.1. à votre avis parmi les 4 figures ci-dessus laquelle parait la plus appropriée ?

- 1
- **2**
- 3

• 4

(explication : la deuxième figure en haut à droite parfait un bon compris pour approcher les points sans trop aller chercher le bruitcomme expliqué dans la partie 3.1.)

4 - Si on veut évaluer la performance de notre algorithme il faut absolument séparer nos données en une partie de test et une partie d'entraînement ?

• **vrai**

• faux

(explication : voir fin de partie 2.3.1.)

5 - En théorie il y a un nombre infini d’hyperparamètres pour un algorithme :

• **vrai**

• faux

(explication : en effet, dans la théorie il existe un nombre infini d’hyperparametre et un nombre encore plus grand de combinaison c’est pour cela qu’on réduit cet espace de recherche en utilisant des tableaux pour chaque hyperparametre)