COURS 4

APPRENTISSAGE SUPERVISÉ : CLASSIFICATION (SUITE)

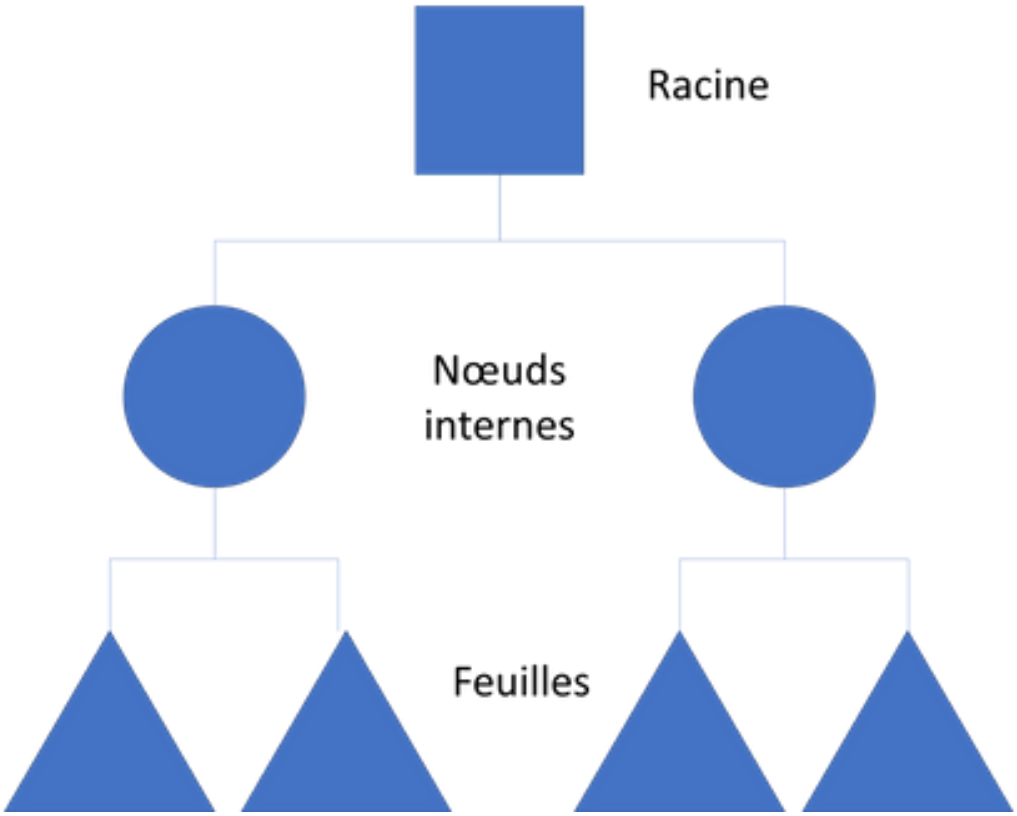
1. ARBRES DE DÉCISIONS
2. FORÊTS ALÉATOIRES (BAGGING)
3. BOOSTING DE GRADIENT
4. CLASSIFICATION NAÏVE BAYÉSIENNE
5. K PLUS PROCHES VOISINS
6. SÉLECTION DE CARACTÉRISTIQUES
7. CLASSIFICATION MULTICLASSE

1. ARBRES DE DÉCISIONS

Les arbres de décisions constituent une autre alternative aux algorithmes de « régression logistique » et de « machines à vecteurs de support ».

- Les deux approches précédentes n'utilisaient qu'une seule frontière décisionnelle pour classifier les données.

- Les arbres de décisions utilisent quant à eux plusieurs frontières décisionnelles.

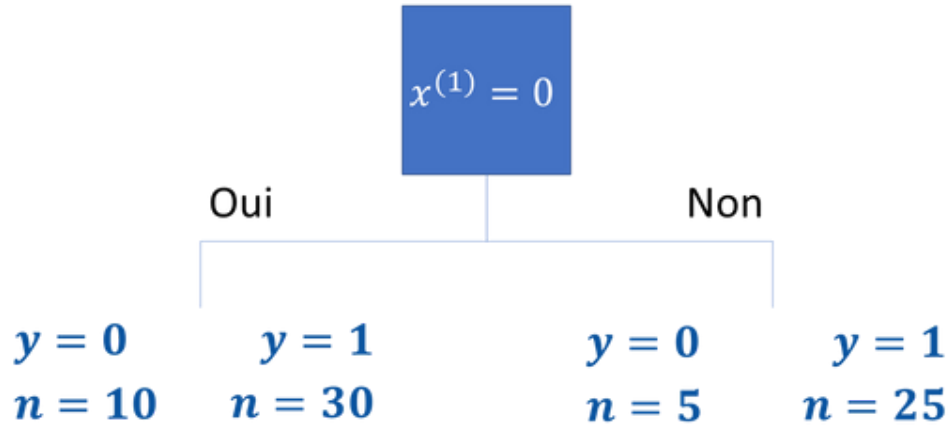


1. ARBRES DE DÉCISIONS

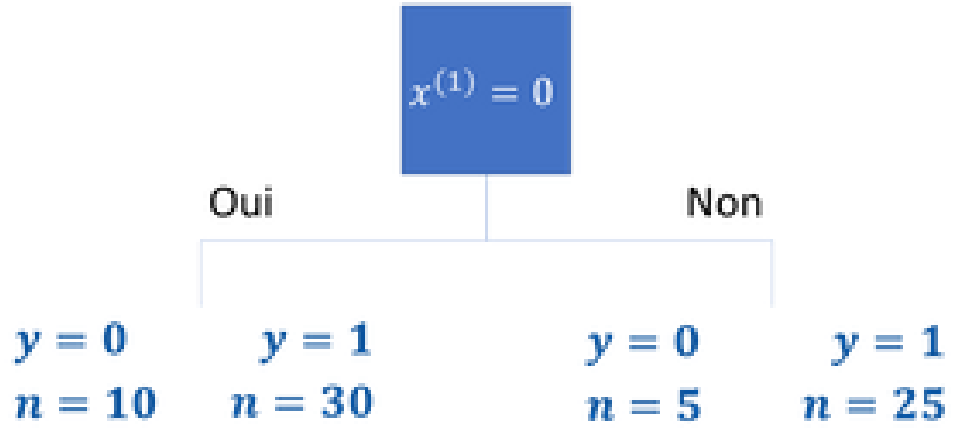
À chaque nouveau nœud, l'algorithme doit sélectionner:

- Une caractéristique (i.e. une variable indépendante) de l'ensemble de données.
- Un critère de décision à l'intérieur de cette caractéristique.

Par exemple, on pourrait sélectionner une caractéristique **catégorielle**, $x^{(1)}$, et séparer les données selon que $x^{(1)} = 0$ ou $x^{(1)} = 1$.



1. ARBRES DE DÉCISIONS



Cette question devient un « candidat » pour le nœud.

- On doit évaluer la qualité du candidat.
- Dans *scikit-learn*, la méthode d'évaluation utilisée par défaut est nommée « **gini** » et utilise le calcul suivant :

$$1 - (\textit{probabilité de } y = 0)^2 - (\textit{probabilité de } y = 1)^2$$

Plus le score est bas, meilleur est le candidat.

- Si l'une des feuilles est « pure » ($y = 0$ pour tous les cas ou $y = 1$ pour tous les cas), alors le score gini est de « 0 ».
- Si dans 50 % des cas $y = 0$, alors le score gini est de « 0.5 ».

Comme les côtés « Oui » et « Non » ne sont pas nécessairement de même taille, on prend la moyenne pondérée des deux.

Exemple. Ici, on a :

- Oui :

$$1 - \left(\frac{10}{10+30}\right)^2 - \left(\frac{30}{10+30}\right)^2 = 1 - 0.0625 - 0.5625 = 0.375$$

- Non :

$$1 - \left(\frac{5}{5+25}\right)^2 - \left(\frac{25}{5+25}\right)^2$$

1. ARBRES DE DÉCISIONS

Si la caractéristique, nommons la $x^{(2)}$, est **continue** (plutôt que catégorielle), on procède de la même manière.

1. ARBRES DE DÉCISIONS

L'algorithme évalue tous les candidats possibles et sélectionne celui correspondant au score *gini* pondéré le plus bas.

Ensuite, on reprend le même processus pour chaque nouveau nœud.

La construction de l'arbre s'arrête selon deux critères possibles:

- Le nombre d'exemples minimum pour qu'on ajoute un nœud de décision.
- Une profondeur maximale (i.e. le nombre d'étages de l'arbre de décisions).

Si on met le « nombre d'exemples minimum » à 2 et qu'on ne spécifie pas de profondeur maximum, alors, la construction de l'arbre va se poursuivre jusqu'à ce que toutes les **feuilles** soient dites **pures**.

- À l'intérieur de chacune des feuilles,
  tous les cas seront $y = 0$ ou
  tous les cas seront $y = 1$.

Problème si toutes les feuilles sont pures…:

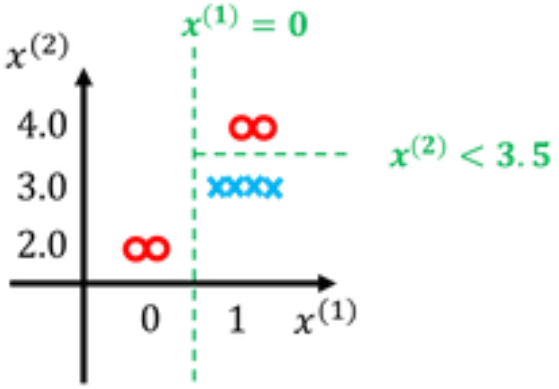
- Surapprentissage (on minimise seulement le biais).

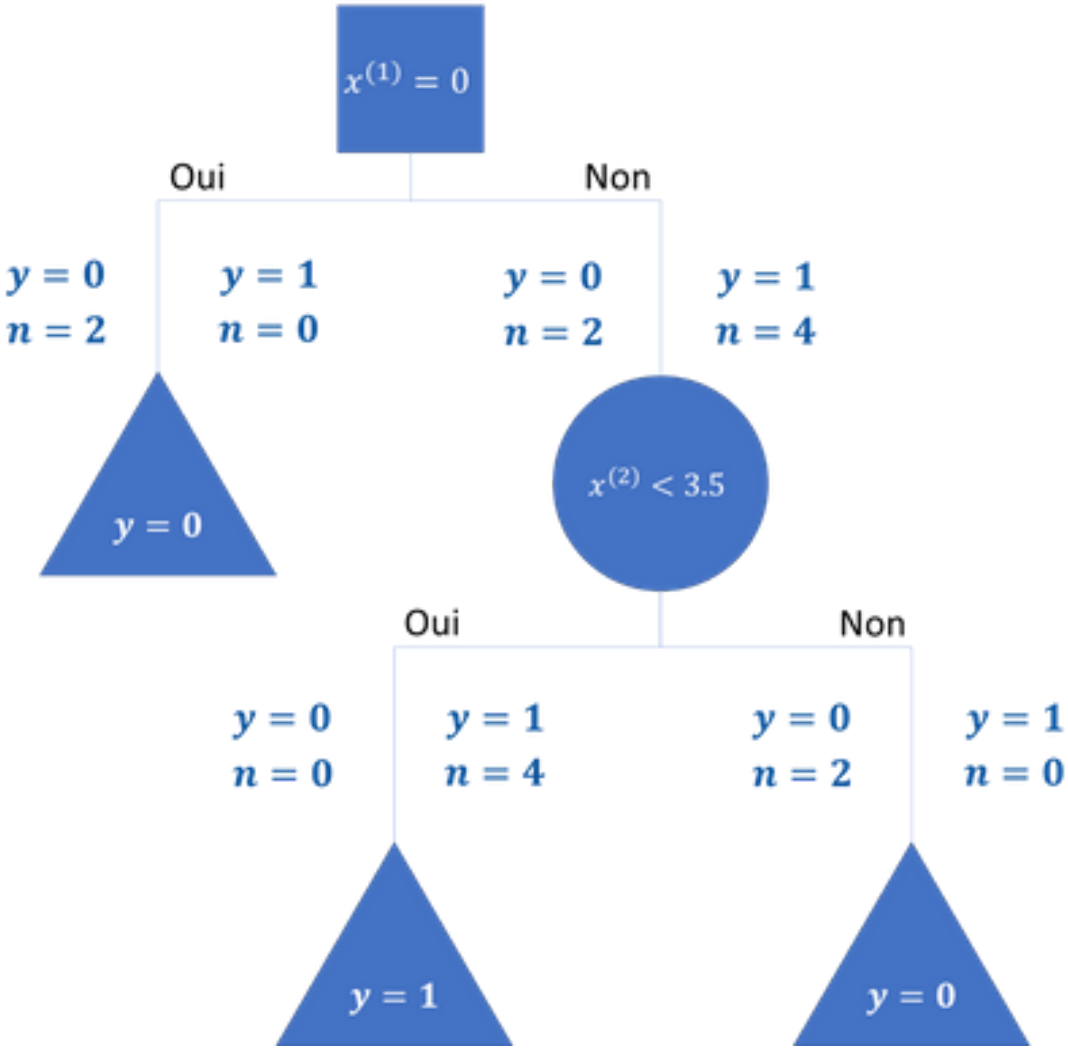
On ajoute du biais (et on diminue la variance) en… :

- …augmentant le nombre d'exemples minimum pour qu'on ajoute un nouveau nœud.
- …en diminuant la profondeur maximale de l'arbre.

1. ARBRES DE DÉCISIONS

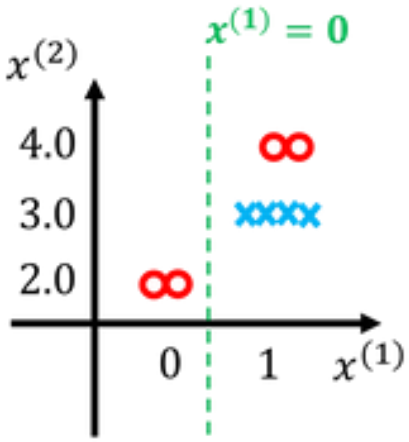
Illustrons dans un cas avec deux caractéristiques $x^{(1)}$ et $x^{(2)}$:

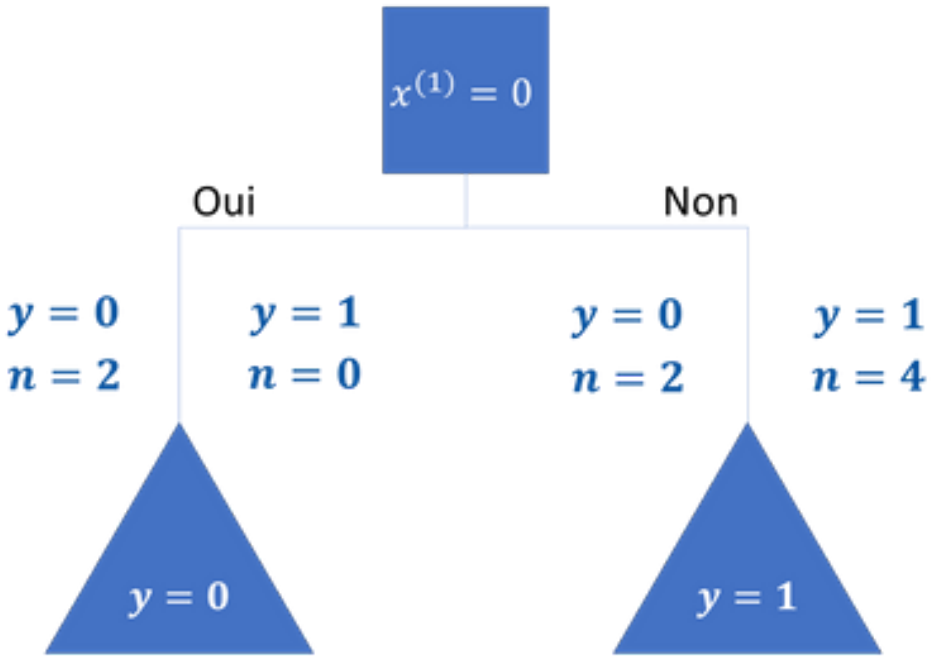




1. ARBRES DE DÉCISIONS

Notons que si nous avions accepté une profondeur maximale de « 1 », on aurait eu :





Quand une feuille n'est pas pure, la décision est prise à la majorité.

1. ARBRES DE DÉCISIONS

Essayons la méthode dans Scikit Learn.

L'hyperparamètre correspondant à la profondeur maximale est nommé :

- ***max_depth***
- Valeur par défaut : ***None***

L'hyperparamètre correspondant au nombre minimum d'exemplaire pour créer un nouveau nœud est nommé :

- ***min_samples_split***
- Valeur par défaut : ***2***

Donc, le modèle par défaut est à risque de surapprentissage.

In [1]:

```
import warnings
warnings.filterwarnings("ignore")
warnings.filterwarnings(action='ignore',category=DeprecationWarning
)
warnings.filterwarnings(action='ignore',category=FutureWarning)

# ----------------------------------------------------------------
------------
# ÉTAPE 1 : importer les librairies utiles
# ----------------------------------------------------------------
------------

import pandas as pd
import numpy as np

# ----------------------------------------------------------------
------------
# ÉTAPE 2 : importer les fonctions utiles
# ----------------------------------------------------------------
------------

# Importer les fonctions de prétraitement
from sklearn.preprocessing import StandardScaler

# Importer une fonction qui nous permette de construire aléatoireme
nt les ensembles "Entraînement" et "Test"
from sklearn.model_selection import train_test_split

# Importer le modèle de régression logistique de sklearn
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier

# Importer la fonction de validation croisée
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Importer la fonction permettant d'afficher le rapport de classifi
cation
from sklearn.metrics import roc_auc_score

np.random.seed(42)

# ----------------------------------------------------------------
------------
# ÉTAPE 3 : importer et préparer le jeu de données
# ----------------------------------------------------------------
------------

# Importons un ensemble de données
data = pd.read_csv('../data/sim_data_signature_small.csv')
data = data.dropna()

# Noms des colonnes correspondant aux caractéristiques et à la cibl
```

```
e
features_cols = ['PSQ_SS', 'PHQ9TT', 'CEVQOTT', 'DAST10TT', 'AUDITT
T', 'STAIYTT', 'AGE', 'SEXE', 'SES']
target_col = 'WHODASTTB'

# Données importées (X: caractéristiques, y: cible)
X = data.loc[:, features_cols]
y = data['WHODASTTB']

# Séparation en données d'entraînement et en données de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2)

# Standatdisation des entrées
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

print('Data ready')

# ----------------------------------------------------------------
------------
# ÉTAPE 4 : définir et entraîner le modèle
# ----------------------------------------------------------------
------------

# Définir le modèle
model = DecisionTreeClassifier()

# Définir les hyperparamètres
hyperparams = {'max_depth':[1, 2, 4, 8, 12, 16], 'min_samples_spli
t':[2, 4, 6, 8, 10, 12, 14, 16]}

# Définir les plus de la validation croisée
cv_folds = StratifiedKFold(n_splits=5, random_state=42)

# Définir le type de score utilisé pour sélectionner les hyperparam
ètres dans la validation croisée
scoring='roc_auc'

# Réaliser la validation croisée avec grille de recherche pour les
 hyperparamètres.
cv_valid = GridSearchCV(estimator=model, param_grid=hyperparams, cv
=cv_folds, scoring=scoring, iid=False)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
model = cv_valid.best_estimator_
print('\nMeilleurs hyperparamètres: \n', best_params)
print('\nScore = \n', best_score)

# Entraîner le modèle final avec toutes les données d'entraînement
model.fit(X_train,y_train)
```

```
Data ready

Meilleurs hyperparamètres:
 {'max_depth': 4, 'min_samples_split': 10}

Score =
 0.7270538693786748
```

Out[1]:

```
DecisionTreeClassifier(class_weight=None, criterion='gi
ni', max_depth=4,
                       max_features=None, max_leaf_node
s=None,
                       min_impurity_decrease=0.0, min_i
mpurity_split=None,
                       min_samples_leaf=1, min_samples_
split=10,
                       min_weight_fraction_leaf=0.0, pr
esort=False,
                       random_state=None, splitter='bes
t')
```

In [2]:

```
# On teste l'algorithme final en prédisant de nouvelles données.
y_pred = model.predict(X_test)

# On évalue les prédictions de l'algorithme final.
auc = roc_auc_score(y_test, y_pred)

# On affiche le résultat
print('\nTest AUC = ', auc)

# On construit une image d'un arbre de décisions
tree.export_graphviz(
            model,
            out_file = './img/_tree.dot',
            feature_names = features_cols,
            class_names = target_col,
            filled = True,
            rounded = True)
```

```
Test AUC =  0.6262553802008608
```

1. ARBRES DE DÉCISIONS

En général, les arbres de décision demeurent à risque de surapprentissage.

Ainsi, les arbres de décisions ont généralement… :

- Une variance relativement élevée.
- Un biais relativement faible.

Pour améliorer leurs performances on utilise couramment des méthodes dites « d'ensembles ».

Il existe deux types de méthodes d'ensembles :

- **Bagging**
- **Boosting**

Nous allons maintenant voir ces deux techniques.

2. FORÊTS ALÉATOIRES (BAGGING)

2. FORÊTS ALÉATOIRES

Le bagging (***B**ootstrap **AGG**regation*) est une technique visant principalement à **réduire la variance** du modèle.

L'idée générale est de… :

1. …entraîner un ensemble de modèles indépendants les uns des autres utilisant chacun une version différente de l'ensemble de données (***bootstrap***).

2. …utiliser l'ensemble des prédictions de tous les modèles entraînés pour faire une prédiction sur une nouvelle donnée (***aggregation*).**

2. FORÊTS ALÉATOIRES

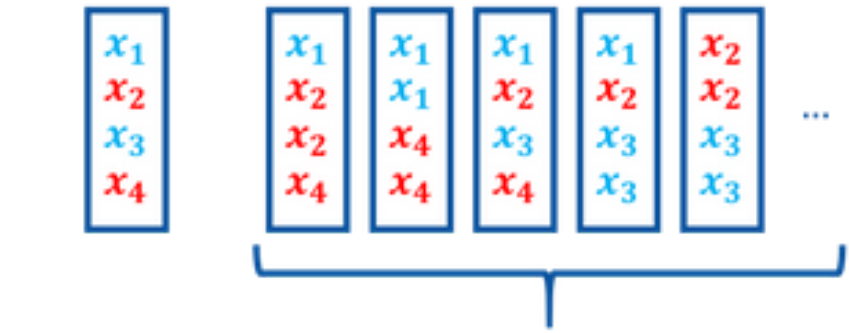
1. …entraîner un ensemble de modèles indépendants les uns des autres.

Dans le contexte des arbres de décisions, chaque arbre de décision est construit avec un ensemble de données différent.

Chaque version de l'ensemble de données est construit de la même manière, en utilisant une méthode de type *bootstrap* :

- On construit le nouvel ensemble de données en tirant aléatoirement *N* exemples à partir de l'ensemble d'entraînement.

- Le tirage aléatoire est réalisé « avec remise ».
  - Ceci signifie que si un exemple a déjà été tiré au hasard, et qu'il fait donc déjà partie du nouvel ensemble, il a néanmoins autant de chances que les autres d'être tiré au hasard pour les exemples suivants du même nouvel ensemble.



Ensemble d'entraînement **original**

Ensembles obtenus par **bootstrap**

2. FORÊTS ALÉATOIRES

1. …entraîner un ensemble de modèles indépendants les uns des autres.

Chaque arbre est donc entraîné sur un ensemble de données distinct.

Toutefois, chaque arbre de décision est construit de la même manière que si on en avait qu'un seul..

…ou presque… :

- La version du *bagging* la plus utilisée avec les arbres de décisions est appelée « **Forêt aléatoire** » *(random forest*).

Dans cette version, on ajoute un hyperparamètre à ceux déjà utilisés (profondeur maximum et nombre d'exemples minimum pour construire un nouveau nœud).

- **On peut maintenant également sélectionner le nombre de caractéristiques utilisées par l'arbre de décision.**

Ceci permet d'augmenter davantage la variance des modèles.

- Toutefois, on augmente ainsi le biais de chaque arbre de décision considéré individuellement.

2. FORÊTS ALÉATOIRES

Et c'est ici qu'entre en jeu le deuxième élément de des méthodes de *bagging* :

2. …utiliser l'ensemble des prédictions de tous les modèles entraînés pour faire une prédiction sur une nouvelle donnée (***aggregation*).**

Spécifiquement, dans les forêts aléatoires utilisées en classification, chaque arbre va tenter de prédire la classe d'un nouvel exemple.

- **La classe recevant le plus de votes est choisie.**

Cette agrégation des résultats permet généralement de conserver un biais relativement bas, tout en diminuer la variance du modèle

- Les méthodes de bagging, et ici spécifiquement la méthode des forêts aléatoires, permettent ainsi de diminuer l'erreur de généralisation par rapport à une méthode utilisant un arbre de décision unique.

2. FORÊTS ALÉATOIRES

Notons que les méthodes de bagging (incluant les forêts aléatoires) peuvent être utilisées tant dans un contexte de classification que dans un contexte de régression.

Dans le contexte de régression, au moment de l'agrégation, on prend généralement la moyenne des résultats des différents arbres.

- Les résiduels correspondent à la somme des carrés de l'erreur (comme dans le cas d'une régression linéaire).

Essayons un exemple de classification avec forêt aléatoire dans *scikit-learn*.

In [3]:

```
import warnings
warnings.filterwarnings("ignore")
warnings.filterwarnings(action='ignore',category=DeprecationWarning
)
warnings.filterwarnings(action='ignore',category=FutureWarning)

# ----------------------------------------------------------------
------------
# ÉTAPE 1 : importer les librairies utiles
# ----------------------------------------------------------------
------------

import pandas as pd
import numpy as np

# ----------------------------------------------------------------
------------
# ÉTAPE 2 : importer les fonctions utiles
# ----------------------------------------------------------------
------------

# Importer les fonctions de prétraitement
from sklearn.preprocessing import StandardScaler

# Importer une fonction qui nous permette de construire aléatoireme
nt les ensembles "Entraînement" et "Test"
from sklearn.model_selection import train_test_split

# Importer le modèle de régression logistique de sklearn
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier

# Importer la fonction de validation croisée
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Importer la fonction permettant d'afficher le rapport de classifi
cation
from sklearn.metrics import roc_auc_score

np.random.seed(42)

# ----------------------------------------------------------------
------------
# ÉTAPE 3 : importer et préparer le jeu de données
# ----------------------------------------------------------------
------------

# Importons un ensemble de données
data = pd.read_csv('../data/sim_data_signature_small.csv')
data = data.dropna()

# Colonnes correspondant à des caractéristiques
```

```
features_cols = ['PSQ_SS', 'PHQ9TT', 'CEVQOTT', 'DAST10TT', 'AUDITT
T', 'STAIYTT', 'AGE', 'SEXE', 'SES']
target_col = 'WHODASTTB'

# Données importées (X: caractéristiques, y: cible)
X = data.loc[:, features_cols]
y = data['WHODASTTB']

# Séparation en données d'entraînement et en données de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2)

# Standatdisation des entrées
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

print('Data ready')

# ----------------------------------------------------------------
------------
# ÉTAPE 4 : définir et entraîner le modèle
# ----------------------------------------------------------------
------------

# Définir le modèle
model = RandomForestClassifier()

# Définir les hyperparamètres
hyperparams = {'n_estimators':[5], \
                      'max_depth':[1, 2, 4, 8, 12, 16], \
                      'min_samples_split':[2, 4, 6, 8, 10, 12, 14,
16], \
                      'max_features': [1, 3, 5, 7, 9]}

# Définir les plus de la validation croisée
cv_folds = StratifiedKFold(n_splits=5, random_state=42)

# Définir le type de score utilisé pour sélectionner les hyperparam
ètres dans la validation croisée
scoring='roc_auc'

# Réaliser la validation croisée avec grille de recherche pour les
 hyperparamètres.
cv_valid = GridSearchCV(estimator=model, param_grid=hyperparams, cv
=cv_folds, scoring=scoring, iid=False)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
model = cv_valid.best_estimator_
print('\nMeilleurs hyperparamètres: \n', best_params)
print('\nScore = \n', best_score)

# Entraîner le modèle final avec toutes les données d'entraînement
model.fit(X_train,y_train)
```

```
Data ready

Meilleurs hyperparamètres:
 {'max_depth': 4, 'max_features': 7, 'min_samples_spli
t': 2, 'n_estimators': 5}

Score =
 0.7453678690608613
```

Out[3]:

```
RandomForestClassifier(bootstrap=True, class_weight=Non
e, criterion='gini',
                       max_depth=4, max_features=7, max
_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_i
mpurity_split=None,
                       min_samples_leaf=1, min_samples_
split=2,
                       min_weight_fraction_leaf=0.0, n_
estimators=5,
                       n_jobs=None, oob_score=False, ra
ndom_state=None,
                       verbose=0, warm_start=False)
```

In [4]:

```
# On teste l'algorithme final en prédisant de nouvelles données.
y_pred = model.predict(X_test)

# On évalue les prédictions de l'algorithme final.
auc = roc_auc_score(y_test, y_pred)

# On affiche le résultat
print('\nTest AUC = ', auc)

# On construit une image d'un arbre de décisions
tree.export_graphviz(
            model[0],
            out_file = './img/_tree.dot',
            feature_names = features_cols,
            class_names = target_col,
            filled = True,
            rounded = True)
```

```
Test AUC =  0.68974175035868
```

3. BOOSTING DE GRADIENT

3. BOOSTING DE GRADIENT

Le *boosting* est une méthode qui permet de réduire à la fois le biais et la variance.

Comme dans le cas du *bagging*, on souhaite mettre à profit un ensemble de modèles.

Toutefois, ici, chaque nouveau modèle (par exemple chaque arbre de décisions), constitue une « extension » des modèles précédents.

L'expression « boosting » vient du fait que la méthode transforme un ensemble « d'apprenants **faibles** » en un unique « apprenant **fort** ».

3. BOOSTING DE GRADIENT

Ici, on n'a pas besoin de construire de nouveaux ensembles de données.

	$x^{(1)}$	$x^{(2)}$	$x^{(\dots)}$	$x^{(D)}$	y
x_1	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(\dots)}$	$x_1^{(D)}$	0
x_2	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(\dots)}$	$x_2^{(D)}$	1
x_3	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(\dots)}$	$x_3^{(D)}$	1
x_4	$x_4^{(1)}$	$x_4^{(2)}$	$x_4^{(\dots)}$	$x_4^{(D)}$	1
x_5	$x_5^{(1)}$	$x_5^{(2)}$	$x_5^{(\dots)}$	$x_5^{(D)}$	0
x_6	$x_6^{(1)}$	$x_6^{(2)}$	$x_6^{(\dots)}$	$x_6^{(D)}$	0
x_7	$x_7^{(1)}$	$x_7^{(2)}$	$x_7^{(\dots)}$	$x_7^{(D)}$	1
x_8	$x_8^{(1)}$	$x_8^{(2)}$	$x_8^{(\dots)}$	$x_8^{(D)}$	1

On va construire une séquence d'arbres de décisions où chaque nouvel arbre va venir améliorer les prédictions

On commence généralement avec un premier arbre constitué d'une simple feuille.

Cette feuille fait une prédiction utilisant le **logarithme des « chances »** d'obtenir la valeur 1 sur la cible (y), si on ne connaît **aucune** caractéristique.

On le calcule ainsi :

$$ln\left(\frac{n_{y=1}}{n_{y\neq 1}}\right)$$

Ici, on a donc :

$$ln\left(\frac{5}{3}\right) = 0.51$$

3. BOOSTING DE GRADIENT

	$x^{(1)}$	$x^{(2)}$	$x^{(\dots)}$	$x^{(D)}$	y
x_1	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(\dots)}$	$x_1^{(D)}$	0
x_2	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(\dots)}$	$x_2^{(D)}$	1
x_3	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(\dots)}$	$x_3^{(D)}$	1
x_4	$x_4^{(1)}$	$x_4^{(2)}$	$x_4^{(\dots)}$	$x_4^{(D)}$	1
x_5	$x_5^{(1)}$	$x_5^{(2)}$	$x_5^{(\dots)}$	$x_5^{(D)}$	0
x_6	$x_6^{(1)}$	$x_6^{(2)}$	$x_6^{(\dots)}$	$x_6^{(D)}$	0
x_7	$x_7^{(1)}$	$x_7^{(2)}$	$x_7^{(\dots)}$	$x_7^{(D)}$	1
x_8	$x_8^{(1)}$	$x_8^{(2)}$	$x_8^{(\dots)}$	$x_8^{(D)}$	1

On peut alors calculer la probabilité que $y = 1$ ainsi :

$$Probabilité(y = 1) = \frac{e^{ln\left(\frac{n_{y=1}}{n_{y\neq 1}}\right)}}{1 + e^{ln\left(\frac{n_{y=1}}{n_{y\neq 1}}\right)}}$$

Ici, on a donc :

$$Probabilité(y = 1) = \frac{e^{ln(0.51)}}{1 + e^{ln(0.51)}} = 0.625$$

3. BOOSTING DE GRADIENT

	$x^{(1)}$	$x^{(2)}$	$x^{(\dots)}$	$x^{(D)}$	y
x_1	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(\dots)}$	$x_1^{(D)}$	0
x_2	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(\dots)}$	$x_2^{(D)}$	1
x_3	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(\dots)}$	$x_3^{(D)}$	1
x_4	$x_4^{(1)}$	$x_4^{(2)}$	$x_4^{(\dots)}$	$x_4^{(D)}$	1
x_5	$x_5^{(1)}$	$x_5^{(2)}$	$x_5^{(\dots)}$	$x_5^{(D)}$	0
x_6	$x_6^{(1)}$	$x_6^{(2)}$	$x_6^{(\dots)}$	$x_6^{(D)}$	0
x_7	$x_7^{(1)}$	$x_7^{(2)}$	$x_7^{(\dots)}$	$x_7^{(D)}$	1
x_8	$x_8^{(1)}$	$x_8^{(2)}$	$x_8^{(\dots)}$	$x_8^{(D)}$	1

On utilise alors généralement 0.5 comme seuil de décision pour déterminer si la valeur de $\boldsymbol{y}$ prédite par le modèle est égale à « 1 » ou si elle est égale à « 0 ».

- On classerait alors les nouveaux exemples ainsi :

$$\begin{cases} 1, & Probabilité(y = 1) \geq 0.5 \\ 0, & Probabilité(y = 1) < 0.5 \end{cases}$$

- Sur la base de ce premier modèle, qui correspond à une simple constante, où $\hat{y} = 0.625$, on classerait donc tous le nouveaux exemples dans la classe $y = 1$.

3. BOOSTING DE GRADIENT

La première étape du modèle est donc sur le point d'être complétée.

Notre premier arbre est constitué d'une simple feuille qui prédit la valeur de y de 0.625.

Il ne nous reste plus qu'à calculer l'erreur de prédiction de notre premier modèle, communément appelée « résiduels » (nommée R_1 dans le tableau suivant).

- $R_1 = y - \hat{y}_1$

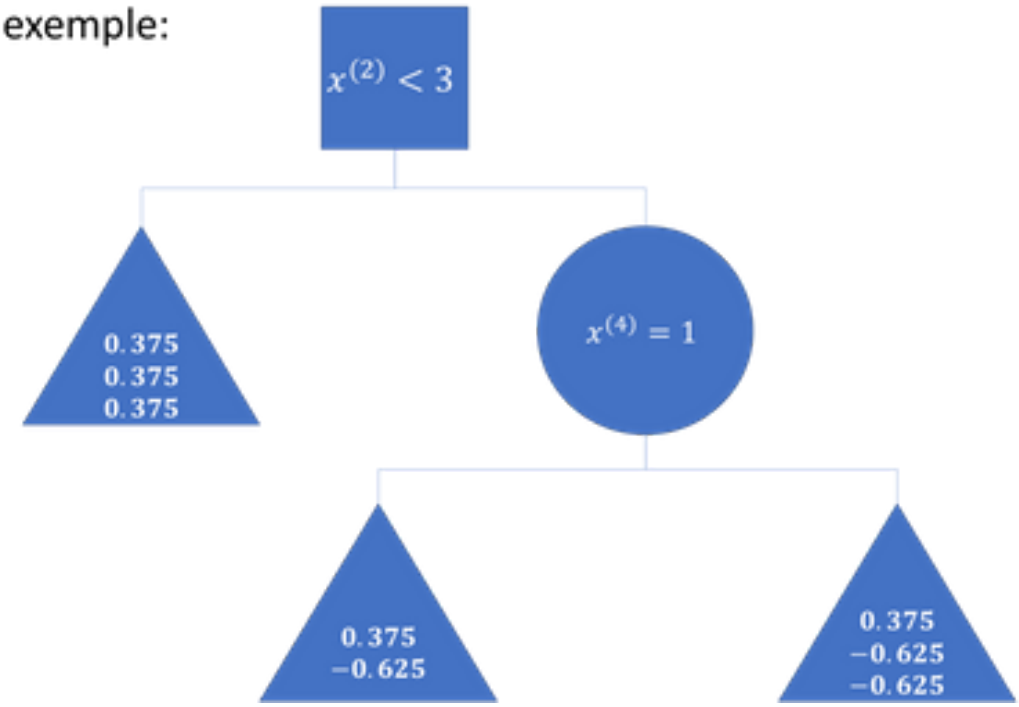
	$x^{(1)}$	$x^{(2)}$	$x^{(\dots)}$	$x^{(D)}$	y	R_1
x_1	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(\dots)}$	$x_1^{(D)}$	0	-0.625
x_2	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(\dots)}$	$x_2^{(D)}$	1	0.375
x_3	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(\dots)}$	$x_3^{(D)}$	1	0.375
x_4	$x_4^{(1)}$	$x_4^{(2)}$	$x_4^{(\dots)}$	$x_4^{(D)}$	1	0.375
x_5	$x_5^{(1)}$	$x_5^{(2)}$	$x_5^{(\dots)}$	$x_5^{(D)}$	0	-0.625
x_6	$x_6^{(1)}$	$x_6^{(2)}$	$x_6^{(\dots)}$	$x_6^{(D)}$	0	-0.625
x_7	$x_7^{(1)}$	$x_7^{(2)}$	$x_7^{(\dots)}$	$x_7^{(D)}$	1	0.375
x_8	$x_8^{(1)}$	$x_8^{(2)}$	$x_8^{(\dots)}$	$x_8^{(D)}$	1	0.375

3. BOOSTING DE GRADIENT

À la deuxième étape, on construirait un arbre de décisions cherchant à **prédire**... les **résiduels** R_1.

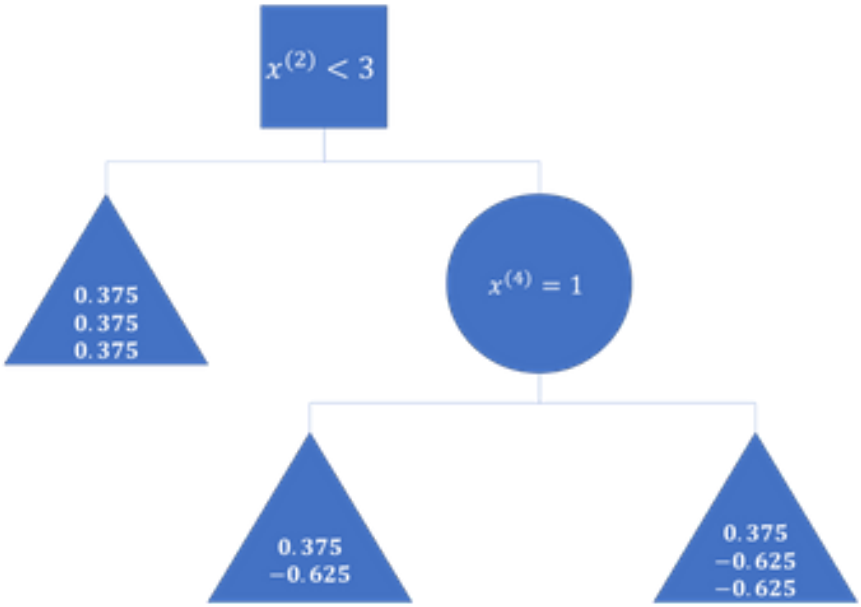
	$x^{(1)}$	$x^{(2)}$	$x^{(\dots)}$	$x^{(D)}$	y	R_1
x_1	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(\dots)}$	$x_1^{(D)}$	0	-0.625
x_2	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(\dots)}$	$x_2^{(D)}$	1	0.375
x_3	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(\dots)}$	$x_3^{(D)}$	1	0.375
x_4	$x_4^{(1)}$	$x_4^{(2)}$	$x_4^{(\dots)}$	$x_4^{(D)}$	1	0.375
x_5	$x_5^{(1)}$	$x_5^{(2)}$	$x_5^{(\dots)}$	$x_5^{(D)}$	0	-0.625
x_6	$x_6^{(1)}$	$x_6^{(2)}$	$x_6^{(\dots)}$	$x_6^{(D)}$	0	-0.625
x_7	$x_7^{(1)}$	$x_7^{(2)}$	$x_7^{(\dots)}$	$x_7^{(D)}$	1	0.375
x_8	$x_8^{(1)}$	$x_8^{(2)}$	$x_8^{(\dots)}$	$x_8^{(D)}$	1	0.375

Par exemple:



3. BOOSTING DE GRADIENT

On doit lors calculer la contribution du deuxième arbre de décisions aux prédictions.



Pour ce faire, on fait d'abord le calcul suivant :

$$\frac{\sum Résiduels_i}{\sum[\ Probabilités\ précédentes_i \times (1 - Probabilités\ précédentes_i)]}$$

Ici, on aurait donc :

$$\frac{0.375 + 0.375 + 0.375}{3\big(0.625 \times (1 - 0.625)\big)} = 1.600$$

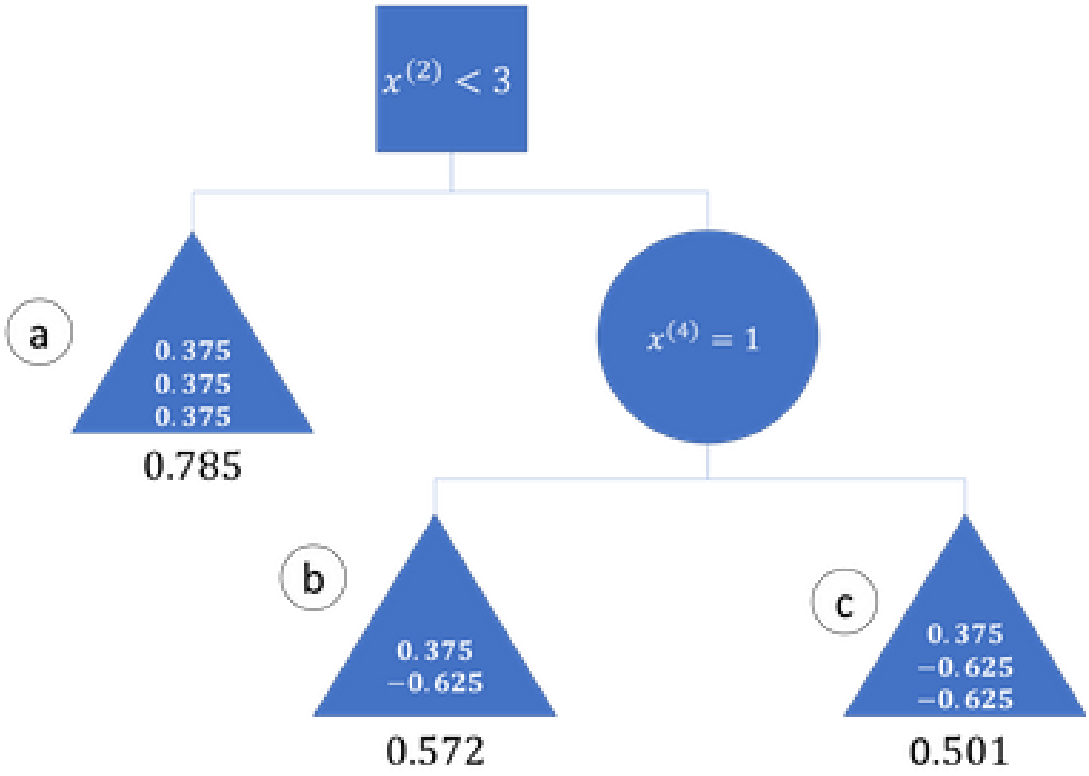
$$\frac{0.375 - 0.625}{2\big(0.625 \times (1 - 0.625)\big)} = -0.533$$

$$\frac{0.375 + -0.625 - 0.625}{3\big(0.625 \times (1 - 0.625)\big)} = -1.24$$

3. BOOSTING DE GRADIENT

On doit alors calculer la contribution du deuxième arbre de décisions aux prédictions.

3. BOOSTING DE GRADIENT



On aurait alors les nouveaux résiduels (R_2) suivants :

	$x^{(1)}$	$x^{(2)}$	$x^{(\dots)}$	$x^{(D)}$	y	R_1	R_2	
x_1	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(\dots)}$	$x_1^{(D)}$	0	−0.625	0.501	c
x_2	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(\dots)}$	$x_2^{(D)}$	1	0.375	0.785	a
x_3	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(\dots)}$	$x_3^{(D)}$	1	0.375	0.785	a
x_4	$x_4^{(1)}$	$x_4^{(2)}$	$x_4^{(\dots)}$	$x_4^{(D)}$	1	0.375	0.785	a
x_5	$x_5^{(1)}$	$x_5^{(2)}$	$x_5^{(\dots)}$	$x_5^{(D)}$	0	−0.625	0.501	c
x_6	$x_6^{(1)}$	$x_6^{(2)}$	$x_6^{(\dots)}$	$x_6^{(D)}$	0	−0.625	0.572	b
x_7	$x_7^{(1)}$	$x_7^{(2)}$	$x_7^{(\dots)}$	$x_7^{(D)}$	1	0.375	0.572	b
x_8	$x_8^{(1)}$	$x_8^{(2)}$	$x_8^{(\dots)}$	$x_8^{(D)}$	1	0.375	0.501	c

3. BOOSTING DE GRADIENT

On ajouterait ainsi les contributions de tous les arbres, de manière à obtenir une prédiction finale pour chaque nouvel exemple, de la forme :

$$Prédiction = 0.625 + 0.1(1.600 + \cdots + \cdots) = \cdots$$

Le **nombre d'arbres de décisions** dans la séquence constitue un **hyperparamètre**.

Aussi, la **vitesse d'apprentissage** constitue un autre **hyperparamètre**.

- On ne veut pas prédire parfaitement chaque classe, car on ne souhaite pas que la variance du modèle soit trop élevée.

La **profondeur maximale** de chaque arbre est également un **hyperparamètre** important.

- On souhaite construire un modèle avec un faible biais, mais on souhaite maintenir la variance faible. Chaque arbre est donc peu profond.

3. BOOSTING DE GRADIENT

Notons que les méthodes de boosting (incluant le gradient boosting) peuvent être utilisées tant dans un contexte de classification que dans un contexte de régression.

Dans le contexte de régression :

- La première feuille correspond à la moyenne de y.
- Les résiduels correspondent à la somme des carrés de l'erreur (comme dans le cas d'une régression linéaire).

Essayons un exemple de classification avec boosting de gradient dans Scikit Learn.

In [5]:

```
import warnings
warnings.filterwarnings("ignore")
warnings.filterwarnings(action='ignore',category=DeprecationWarning
)
warnings.filterwarnings(action='ignore',category=FutureWarning)

# ----------------------------------------------------------------
------------
# ÉTAPE 1 : importer les librairies utiles
# ----------------------------------------------------------------
------------

import pandas as pd
import numpy as np

# ----------------------------------------------------------------
------------
# ÉTAPE 2 : importer les fonctions utiles
# ----------------------------------------------------------------
------------

# Importer les fonctions de prétraitement
from sklearn.preprocessing import StandardScaler

# Importer une fonction qui nous permette de construire aléatoireme
nt les ensembles "Entraînement" et "Test"
from sklearn.model_selection import train_test_split

# Importer le modèle de régression logistique de sklearn
from sklearn import tree
from sklearn.ensemble import GradientBoostingClassifier

# Importer la fonction de validation croisée
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Importer la fonction permettant d'afficher le rapport de classifi
cation
from sklearn.metrics import roc_auc_score

np.random.seed(42)

# ----------------------------------------------------------------
------------
# ÉTAPE 3 : importer et préparer le jeu de données
# ----------------------------------------------------------------
------------

# Importons un ensemble de données
data = pd.read_csv('../data/sim_data_signature_small.csv')
data = data.dropna()

# Colonnes correspondant à des caractéristiques
```

```
features_cols = ['PSQ_SS', 'PHQ9TT', 'CEVQOTT', 'DAST10TT', 'AUDITT
T', 'STAIYTT', 'AGE', 'SEXE', 'SES']
target_col = 'WHODASTTB'

# Données importées (X: caractéristiques, y: cible)
X = data.loc[:, features_cols]
y = data['WHODASTTB']

# Séparation en données d'entraînement et en données de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2)

# Standatdisation des entrées
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

print('Data ready')

# ----------------------------------------------------------------
------------
# ÉTAPE 4 : définir et entraîner le modèle
# ----------------------------------------------------------------
------------

# Définir le modèle
model = GradientBoostingClassifier()

# Définir les hyperparamètres
hyperparams = {'learning_rate':[0.1, 0.2, 0.4, 0.8], \
                      'n_estimators':[5], \
                      'max_depth':[1, 2, 4, 8, 12, 16], \
                      'min_samples_split':[2, 4, 6, 8, 10, 12, 14,
16]}

# Définir les plus de la validation croisée
cv_folds = StratifiedKFold(n_splits=5, random_state=42)

# Définir le type de score utilisé pour sélectionner les hyperparam
ètres dans la validation croisée
scoring='roc_auc'

# Réaliser la validation croisée avec grille de recherche pour les
 hyperparamètres.
cv_valid = GridSearchCV(estimator=model, param_grid=hyperparams, cv
=cv_folds, scoring=scoring, iid=False)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
model = cv_valid.best_estimator_
print('\nMeilleurs hyperparamètres: \n', best_params)
print('\nScore = \n', best_score)

# Entraîner le modèle final avec toutes les données d'entraînement
model.fit(X_train,y_train)
```

```
Data ready

Meilleurs hyperparamètres:
 {'learning_rate': 0.1, 'max_depth': 4, 'min_samples_sp
lit': 10, 'n_estimators': 5}

Score =
 0.7540402034006038
```

Out[5]:

```
GradientBoostingClassifier(criterion='friedman_mse', in
it=None,
                           learning_rate=0.1, loss='dev
iance', max_depth=4,
                           max_features=None, max_leaf_
nodes=None,
                           min_impurity_decrease=0.0, m
in_impurity_split=None,
                           min_samples_leaf=1, min_samp
les_split=10,
                           min_weight_fraction_leaf=0.
0, n_estimators=5,
                           n_iter_no_change=None, preso
rt='auto',
                           random_state=None, subsample
=1.0, tol=0.0001,
                           validation_fraction=0.1, ver
bose=0,
                           warm_start=False)
```

In [6]:

```
# On teste l'algorithme final en prédisant de nouvelles données.
y_pred = model.predict(X_test)

# On évalue les prédictions de l'algorithme final.
auc = roc_auc_score(y_test, y_pred)

# On affiche le résultat
print('\nTest AUC = ', auc)

# On construit une image d'un arbre de décisions
tree.export_graphviz(
            model[1, 0],
            out_file = './img/_tree.dot',
            feature_names = features_cols,
            class_names = target_col,
            filled = True,
            rounded = True)
```

```
Test AUC =  0.665351506456241
```

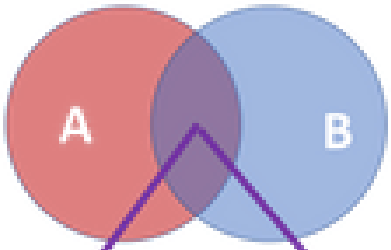
4. CLASSIFICATION NAÏVE BAYÉSIENNE

4. Classification naïve bayésienne

Basée sur le théorème de Bayes

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Illustrons le théorème



Réécrivons l'équation

$$\boldsymbol{P(A,B) = P(A|B)P(B) = P(B|A)P(A)}$$

En mots :

- A : on fait un pic-nic
- B : il fait soleil
- $P(A|B)$: la probabilité qu'on faisse un pic-nic s'il fait soleil
- $P(B)$: la probabilité qu'il fasse soleil
- $P(B|A)$: la probabilité qu'il fasse soleil si on fait un pic-nic
- $P(A)$: la probabilité qu'on fasse un pic-nic

Et donc :

$$P(\text{on fait un pic-nic, il fait soleil})$$
$$=$$
$$P(\text{on fait un pic-nic} \mid \text{il fait soleil}) \times P(il\ fait\ soleil)$$
$$=$$
$$P(\text{il fait soleil} \mid \text{on fait un pic-nic}) \times P(on\ fait\ un\ \text{pic-nic})$$

Or, on s'intéresse généralement à seulement l'un des termes.

Par exemple :

- Qu'elle est la probabilité qu'on fasse un pic-nic cette après-midi état donné la température ?

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Si on reprend notre structure de données en apprentissage

supervisé, on a :

4. Classification naïve bayésienne

Prenons l'exemple suivant :

	$x^{(1)}$	$x^{(2)}$	y
x_1	*Soleil* (1)	*Venteux* (0)	Rien (0)
x_2	*Pluie* (0)	*Calme* (1)	Pic-nic (1)
x_3	*Soleil* (1)	*Calme* (1)	Pic-nic (1)
x_4	*Soleil* (1)	*Venteux* (0)	Pic-nic (1)
x_5	*Pluie* (0)	*Venteux* (0)	Rien (0)
x_6	*Soleil* (1)	*Calme* (1)	Rien (0)
x_7	*Soleil* (1)	*Calme* (1)	Pic–nic (1)
x_8	*Soleil* (1)	*Calme* (1)	Pic–nic (1)

Disons qu'on a un nouvel exemplaire où « il fait soleil » et le temps est « calme ».

➢ On veut classifier l'exemple dans « pic-nic » ou « rien ».

Rappelons l'équation :

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

Trouvons d'abord la probabilité qu'il y ait un pic-nic :

$$P\left(y = 1|x^{(1)} = 1, x^{(2)} = 1\right) = \frac{P\left(x^{(1)} = 1, x^{(2)} = 1|y = 1\right)P(y = 1)}{P(x^{(1)} = 1, x^{(2)} = 1)}$$

On a alors :

$P(y = 1) = 5/8 = 0.625$

$P(X) = P\left(x^{(1)} = 1\right) \times P\left(x^{(2)} = 1\right) = (6/8) \times (5/8) = 0.469$

$P(X|y) = P\left(x^{(1)} = 1, x^{(2)} = 1|y = 1\right) = 3/5 = 0.600$

Ce qui nous donne une probabilité de

$$P\left(y = 1|x^{(1)} = 1, x^{(2)} = 1\right) = \frac{0.600 \times 0.625}{0.469} = 0.800$$

4. Classification naïve bayésienne

Prenons l'exemple suivant :

	$x^{(1)}$	$x^{(2)}$	y
x_1	*Soleil* (1)	*Venteux* (0)	Rien (0)
x_2	*Pluie* (0)	*Calme* (1)	Pic-nic (1)
x_3	*Soleil* (1)	*Calme* (1)	Pic-nic (1)
x_4	*Soleil* (1)	*Venteux* (0)	Pic-nic (1)
x_5	*Pluie* (0)	*Venteux* (0)	Rien (0)
x_6	*Soleil* (1)	*Calme* (1)	Rien (0)
x_7	*Soleil* (1)	*Calme* (1)	Pic–nic (1)
x_8	*Soleil* (1)	*Calme* (1)	Pic–nic (1)

Disons qu'on a un nouvel exemplaire où « il fait soleil » et le temps est « calme ».

➢ On veut classifier l'exemple dans « pic-nic » ou « rien ».

Rappelons l'équation :

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

Trouvons maintenant la probabilité qu'il n'y ait rien :

$$P\left(y = 0|x^{(1)} = 1, x^{(2)} = 1\right) = \frac{P\left(x^{(1)} = 1, x^{(2)} = 1|y = 0\right)P(y = 0)}{P(x^{(1)} = 1, x^{(2)} = 1)}$$

On a alors :

$P(y = 0) = 3/8 = 0.375$

$P(X) = P\left(x^{(1)} = 1\right) \times P\left(x^{(2)} = 1\right) = (6/8) \times (5/8) = 0.469$

$P(X|y) = P\left(x^{(1)} = 1, x^{(2)} = 1|y = 0\right) = 1/3 = 0.333$

Ce qui nous donne une probabilité de

$$P\left(y = 0|x^{(1)} = 1, x^{(2)} = 1\right) = \frac{0.333 \times 0.375}{0.469} 0.266$$

4. Classification naïve bayésienne

Prenons l'exemple suivant :

	$x^{(1)}$	$x^{(2)}$	y
x_1	*Soleil* (1)	*Venteux* (0)	Rien (0)
x_2	*Pluie* (0)	*Calme* (1)	Pic-nic (1)
x_3	*Soleil* (1)	*Calme* (1)	Pic-nic (1)
x_4	*Soleil* (1)	*Venteux* (0)	Pic-nic (1)
x_5	*Pluie* (0)	*Venteux* (0)	Rien (0)
x_6	*Soleil* (1)	*Calme* (1)	Rien (0)
x_7	*Soleil* (1)	*Calme* (1)	Pic–nic (1)
x_8	*Soleil* (1)	*Calme* (1)	Pic–nic (1)

Ainsi, étant donné l'exemple, il y a 80 % de chances de faire un pic-nic et 27 % de chances de ne rien faire.

- On classifiera donc l'exemple dans « faire un pic-nic ».

Comme certaines caractéristiques sont généralement continues, on assume généralement que celles-ci sont distribuées « normalement » pour calculer $P(X|y)$ et calculer $P(X)$ et on utilise la version « GaussianNB » dans *scikit-learn*.

Notons que la méthode est qualifiée de « naïve » parce que le calcul de

$$P(X) = P\left(x^{(1)} = 1\right) \times P\left(x^{(2)} = 1\right) \times \cdots$$

prend pour acquis que les caractéristiques sont indépendantes, ce qui n'est jamais vraiment le cas dans la réalité.

- La méthode fonctionne généralement bien même si ce postulat n'est pas respecté.

Enfin, on n'a généralement aucun hyperparamètre à ajuster (il existe cependant « *var_smoothing* » dans *scikit-*

learn, mais on n'a généralement pas besoin de le modifier).

In [7]:

```
import warnings
warnings.filterwarnings("ignore")
warnings.filterwarnings(action='ignore',category=DeprecationWarning
)
warnings.filterwarnings(action='ignore',category=FutureWarning)

# ----------------------------------------------------------------
------------
# ÉTAPE 1 : importer les librairies utiles
# ----------------------------------------------------------------
------------

import pandas as pd
import numpy as np

# ----------------------------------------------------------------
------------
# ÉTAPE 2 : importer les fonctions utiles
# ----------------------------------------------------------------
------------

# Importer les fonctions de prétraitement
from sklearn.preprocessing import StandardScaler

# Importer une fonction qui nous permette de construire aléatoireme
nt les ensembles "Entraînement" et "Test"
from sklearn.model_selection import train_test_split

# Importer le modèle de régression logistique de sklearn
from sklearn.naive_bayes import GaussianNB

# Importer la fonction de validation croisée
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Importer la fonction permettant d'afficher le rapport de classifi
cation
from sklearn.metrics import roc_auc_score

np.random.seed(42)

# ----------------------------------------------------------------
------------
# ÉTAPE 3 : importer et préparer le jeu de données
# ----------------------------------------------------------------
------------

# Importons un ensemble de données
data = pd.read_csv('../data/sim_data_signature_small.csv')
data = data.dropna()

# Colonnes correspondant à des caractéristiques
features_cols = ['PSQ_SS', 'PHQ9TT', 'CEVQOTT', 'DAST10TT', 'AUDITT
```

```
T', 'STAIYTT', 'AGE', 'SEXE', 'SES']

# Données importées (X: caractéristiques, y: cible)
X = data.loc[:, features_cols]
y = data['WHODASTTB']

# Séparation en données d'entraînement et en données de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2)

# Standatdisation des entrées
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

print('Data ready')

# ----------------------------------------------------------------
------------
# ÉTAPE 4 : définir et entraîner le modèle
# ----------------------------------------------------------------
------------

# Définir le modèle
model = GaussianNB()

# Définir les hyperparamètres
hyperparams = {'var_smoothing':[1e-8, 1e-9, 1e-10]}

# Définir les plus de la validation croisée
cv_folds = StratifiedKFold(n_splits=5, random_state=42)

# Définir le type de score utilisé pour sélectionner les hyperparam
ètres dans la validation croisée
scoring='roc_auc'

# Réaliser la validation croisée avec grille de recherche pour les
 hyperparamètres.
cv_valid = GridSearchCV(estimator=model, param_grid=hyperparams, cv
=cv_folds, scoring=scoring, iid=False)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
model = cv_valid.best_estimator_
print('\nMeilleurs hyperparamètres: \n', best_params)
print('\nScore = \n', best_score)

# Entraîner le modèle final avec toutes les données d'entraînement
model.fit(X_train,y_train)
```

```
Data ready

Meilleurs hyperparamètres:
 {'var_smoothing': 1e-08}

Score =
 0.7162084856189417
```

Out[7]:

```
GaussianNB(priors=None, var_smoothing=1e-08)
```

In [8]:

```
# On teste l'algorithme final en prédisant de nouvelles données.
y_pred = model.predict(X_test)

# On évalue les prédictions de l'algorithme final.
auc = roc_auc_score(y_test, y_pred)

# On affiche le résultat
print('\nTest AUC = ', auc)
```

```
Test AUC =  0.5893113342898135
```

5. K PLUS PROCHES VOISINS

5. K plus proches voisins

Cette méthode est purement basée sur
la proximité des exemples.

Étapes :

1. On fixe le nombre de voisins utilisés dans la décision (i.e. *k*).

2. Pour un nouvel exemple dont on doit prédire la classe, on vérifie les classes des *k* exemples les plus proches (i.e. les *k* voisins).

3. La classe la plus fréquente (dans ces *k* voisins) est la classe prédite pour le nouvel exemple.
   - Si égalité, on tire au hasard.

Dans l'exemple suivant pour le nouvel exemple ▲ :

In [9]:

```
import warnings
warnings.filterwarnings("ignore")
warnings.filterwarnings(action='ignore',category=DeprecationWarning
)
warnings.filterwarnings(action='ignore',category=FutureWarning)

# ----------------------------------------------------------------
------------
# ÉTAPE 1 : importer les librairies utiles
# ----------------------------------------------------------------
------------

import pandas as pd
import numpy as np

# ----------------------------------------------------------------
------------
# ÉTAPE 2 : importer les fonctions utiles
# ----------------------------------------------------------------
------------

# Importer les fonctions de prétraitement
from sklearn.preprocessing import StandardScaler

# Importer une fonction qui nous permette de construire aléatoireme
nt les ensembles "Entraînement" et "Test"
from sklearn.model_selection import train_test_split

# Importer le modèle de régression logistique de sklearn
from sklearn.neighbors import KNeighborsClassifier

# Importer la fonction de validation croisée
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Importer la fonction permettant d'afficher le rapport de classifi
cation
from sklearn.metrics import roc_auc_score

np.random.seed(42)

# ----------------------------------------------------------------
------------
# ÉTAPE 3 : importer et préparer le jeu de données
# ----------------------------------------------------------------
------------

# Importons un ensemble de données
data = pd.read_csv('../data/sim_data_signature_small.csv')
data = data.dropna()

# Colonnes correspondant à des caractéristiques
features_cols = ['PSQ_SS', 'PHQ9TT', 'CEVQOTT', 'DAST10TT', 'AUDITT
```

```
T', 'STAIYTT', 'AGE', 'SEXE', 'SES']

# Données importées (X: caractéristiques, y: cible)
X = data.loc[:, features_cols]
y = data['WHODASTTB']

# Séparation en données d'entraînement et en données de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2)

# Standatdisation des entrées
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

print('Data ready')

# ----------------------------------------------------------------
------------
# ÉTAPE 4 : définir et entraîner le modèle
# ----------------------------------------------------------------
------------

# Définir le modèle
model = KNeighborsClassifier()

# Définir les hyperparamètres
hyperparams = {'n_neighbors':[1, 2, 4, 8, 16], 'weights':['uniform'
, 'distance']}

# Définir les plus de la validation croisée
cv_folds = StratifiedKFold(n_splits=5, random_state=42)

# Définir le type de score utilisé pour sélectionner les hyperparam
ètres dans la validation croisée
scoring='roc_auc'

# Réaliser la validation croisée avec grille de recherche pour les
 hyperparamètres.
cv_valid = GridSearchCV(estimator=model, param_grid=hyperparams, cv
=cv_folds, scoring=scoring, iid=False)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
model = cv_valid.best_estimator_
print('\nMeilleurs hyperparamètres: \n', best_params)
print('\nScore = \n', best_score)

# Entraîner le modèle final avec toutes les données d'entraînement
model.fit(X_train,y_train)
```

```
Data ready

Meilleurs hyperparamètres:
 {'n_neighbors': 16, 'weights': 'distance'}

Score =
 0.6926744001271252
```

Out[9]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, me
tric='minkowski',
                     metric_params=None, n_jobs=None, n
_neighbors=16, p=2,
                     weights='distance')
```

In [10]:

```
# On teste l'algorithme final en prédisant de nouvelles données.
y_pred = model.predict(X_test)

# On évalue les prédictions de l'algorithme final.
auc = roc_auc_score(y_test, y_pred)

# On affiche le résultat
print('\nTest AUC = ', auc)
```

```
Test AUC =  0.706599713055954
```

RÉCAPITULATIF

Voici certains avantages/inconvénients de ces méthodes de classifications :

Régression logistique

- Avantages
  - On a toujours un minimum global.
  - Un peu interprétable.
  - Bon avec données larges.
- Inconvénients
  - Peu efficace avec solutions non-linéaires.
  - Sensible à la colinéarité.
  - Sensible aux valeurs aberrantes.

Machines à vecteurs de support

- Avantages
  - On a toujours un minimum global.
  - Efficace avec solutions non-linéaire (astuce du noyau).
  - Peu sensible aux valeurs aberrantes.
  - Bon avec données larges.
- Inconvénients
  - Peu efficace en termes de temps de computation avec grands ensembles de données.

RÉCAPITULATIF

Voici certains avantages/inconvénients de ces méthodes de classifications :

Arbres de décisions

- Avantages
  - Faciles à interpréter
  - Bon avec solutions complexes.
  - Bon avec grands ensembles de données.
- Inconvénients
  - Peu robuste (variance élevée).

Forêts aléatoires

- Avantages
  - Bon avec solutions complexes.
  - Bon avec grands ensembles de données.
  - Robuste (variance faible)
- Inconvénients
  - Difficiles à interpréter.
  - Longs à entraîner.

Boosting de gradient

- Avantages
  - Bon avec solutions très complexes.
  - Bon avec grands ensembles de données.
  - Biais généralement plus faible que « forêts aléatoires ».
- Inconvénients
  - Variance généralement plus élevée que « forêts aléatoires ».
  - Longs à entraîner

RÉCAPITULATIF

Voici certains avantages/inconvénients de ces méthodes de classifications :

Classification naïve bayésienne

- Avantages
  - Pas d'entraînement requis.
  - Peu d'hyperparamètres.
  - Très rapide
- Inconvénients
  - Sensible à la colinéarité

K plus proches voisins

- Avantages
  - Pas d'entraînement requis.
- Inconvénients
  - K peut être difficile à ajuster.
  - Peu être lent si l'ensemble de données est grand.

6. SÉLECTION DES CARACTÉRISTIQUES

6. SÉLECTION DE CARACTÉRISTIQES

Permet d'augmenter l'interprétabilité des différents algorithmes.

- Trois types:
  1. Sélection par « filtrage »
  2. *Sélection à l'aide d'un « wrapper »*
  3. *Sélection* « intégrée » (*embedded*)

Filtrage

- Sélection en amont de l'algorithme (prétraitement).
  - Retrait des caractéristiques ayant une faible variance. (*e.g. sklearn : VarianceThreshold*)

Wrapper

- Sélection en fonction des modèles entraînés.
  - Sélection des caractéristiques les plus importantes. (*e.g. sklearn : SelectFromModel*)

Intégrée

- Sélection durant l'entraînement du modèle.
  - Sélection des caractéristiques les plus importantes. (*e.g. sklearn : LogisticRegression(penalty='l1')*)

EXEMPLE "FILTRAGE"

In [11]:

```
import warnings
warnings.filterwarnings("ignore")
warnings.filterwarnings(action='ignore',category=DeprecationWarning
)
warnings.filterwarnings(action='ignore',category=FutureWarning)

# ----------------------------------------------------------------
------------
# ÉTAPE 1 : importer les librairies utiles
# ----------------------------------------------------------------
------------

import pandas as pd
import numpy as np

# ----------------------------------------------------------------
------------
# ÉTAPE 2 : importer les fonctions utiles
# ----------------------------------------------------------------
------------

# Importer les fonctions de prétraitement
from sklearn.preprocessing import StandardScaler

# Importer une fonction de sélection de données
from sklearn.feature_selection import VarianceThreshold

# Importer une fonction qui nous permette de construire aléatoireme
nt les ensembles "Entraînement" et "Test"
from sklearn.model_selection import train_test_split

# Importer le modèle de régression logistique de sklearn
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier

# Importer la fonction de validation croisée
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Importer la fonction permettant d'afficher le rapport de classifi
cation
from sklearn.metrics import roc_auc_score

np.random.seed(42)

# ----------------------------------------------------------------
------------
# ÉTAPE 3 : importer et préparer le jeu de données
# ----------------------------------------------------------------
------------

# Importons un ensemble de données
data = pd.read_csv('../data/sim_data_signature_small.csv')
```

```
data = data.dropna()

# Noms des colonnes correspondant aux caractéristiques et à la cibl
e
features_cols = ['PSQ_SS', 'PHQ9TT', 'CEVQOTT', 'DAST10TT', 'AUDITT
T', 'STAIYTT', 'AGE', 'SEXE', 'SES']
target_col = 'WHODASTTB'

# Données importées (X: caractéristiques, y: cible)
X = data.loc[:, features_cols]
y = data['WHODASTTB']

# Séparation en données d'entraînement et en données de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2)

# Standatdisation des entrées
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)


print(X_train.shape)
sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
sel.fit_transform(X_train)
sel.fit_transform(X_test)
print(X_train.shape)

print('Data ready')

# ----------------------------------------------------------------
------------
# ÉTAPE 4 : définir et entraîner le modèle
# ----------------------------------------------------------------
------------

# Définir le modèle
model = DecisionTreeClassifier()

# Définir les hyperparamètres
hyperparams = {'max_depth':[1, 2, 4, 8, 12, 16], 'min_samples_spli
t':[2, 4, 6, 8, 10, 12, 14, 16]}

# Définir les plus de la validation croisée
cv_folds = StratifiedKFold(n_splits=5, random_state=42)

# Définir le type de score utilisé pour sélectionner les hyperparam
ètres dans la validation croisée
scoring='roc_auc'

# Réaliser la validation croisée avec grille de recherche pour les
 hyperparamètres.
cv_valid = GridSearchCV(estimator=model, param_grid=hyperparams, cv
=cv_folds, scoring=scoring, iid=False)
cv_valid.fit(X_train, y_train)
```

```
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
model = cv_valid.best_estimator_
print('\nMeilleurs hyperparamètres: \n', best_params)
print('\nScore = \n', best_score)

# Entraîner le modèle final avec toutes les données d'entraînement
model.fit(X_train,y_train)
```

```
(297, 9)
(297, 9)
Data ready

Meilleurs hyperparamètres:
 {'max_depth': 4, 'min_samples_split': 10}

Score =
 0.7270538693786748
```

Out[11]:

```
DecisionTreeClassifier(class_weight=None, criterion='gi
ni', max_depth=4,
                       max_features=None, max_leaf_node
s=None,
                       min_impurity_decrease=0.0, min_i
mpurity_split=None,
                       min_samples_leaf=1, min_samples_
split=10,
                       min_weight_fraction_leaf=0.0, pr
esort=False,
                       random_state=None, splitter='bes
t')
```

In [12]:

```
# On teste l'algorithme final en prédisant de nouvelles données.
y_pred = model.predict(X_test)

# On évalue les prédictions de l'algorithme final.
auc = roc_auc_score(y_test, y_pred)

# On affiche le résultat
print('\nTest AUC = ', auc)
```

```
Test AUC =  0.6262553802008608
```

EXEMPLE "WRAPPER"

In [13]:

```
import warnings
warnings.filterwarnings("ignore")
warnings.filterwarnings(action='ignore',category=DeprecationWarning
)
warnings.filterwarnings(action='ignore',category=FutureWarning)

# ----------------------------------------------------------------
------------
# ÉTAPE 1 : importer les librairies utiles
# ----------------------------------------------------------------
------------

import pandas as pd
import numpy as np

# ----------------------------------------------------------------
------------
# ÉTAPE 2 : importer les fonctions utiles
# ----------------------------------------------------------------
------------

# Importer les fonctions de prétraitement
from sklearn.preprocessing import StandardScaler

# Importer une fonction de sélection de données
from sklearn.feature_selection import VarianceThreshold, SelectFrom
Model

# Importer une fonction qui nous permette de construire aléatoireme
nt les ensembles "Entraînement" et "Test"
from sklearn.model_selection import train_test_split

# Importer le modèle de régression logistique de sklearn
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier

# Importer la fonction de validation croisée
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Importer la fonction permettant d'afficher le rapport de classifi
cation
from sklearn.metrics import roc_auc_score

np.random.seed(42)

# ----------------------------------------------------------------
------------
# ÉTAPE 3 : importer et préparer le jeu de données
# ----------------------------------------------------------------
------------

# Importons un ensemble de données
```

```
data = pd.read_csv('../data/sim_data_signature_small.csv')
data = data.dropna()

# Noms des colonnes correspondant aux caractéristiques et à la cibl
e
features_cols = ['PSQ_SS', 'PHQ9TT', 'CEVQOTT', 'DAST10TT', 'AUDITT
T', 'STAIYTT', 'AGE', 'SEXE', 'SES']
target_col = 'WHODASTTB'

# Données importées (X: caractéristiques, y: cible)
X = data.loc[:, features_cols]
y = data['WHODASTTB']

# Séparation en données d'entraînement et en données de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2)

# Standatdisation des entrées
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)


print(X_train.shape)
sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
sel.fit_transform(X_train)
sel.fit_transform(X_test)
print(X_train.shape)

print('Data ready')

# ----------------------------------------------------------------
------------
# ÉTAPE 4 : définir et entraîner le modèle
# ----------------------------------------------------------------
------------

# Définir le modèle
model = DecisionTreeClassifier()

# Définir les hyperparamètres
hyperparams = {'max_depth':[1, 2, 4, 8, 12, 16], 'min_samples_spli
t':[2, 4, 6, 8, 10, 12, 14, 16]}

# Définir les plus de la validation croisée
cv_folds = StratifiedKFold(n_splits=5, random_state=42)

# Définir le type de score utilisé pour sélectionner les hyperparam
ètres dans la validation croisée
scoring='roc_auc'

# Réaliser une première itération de la validation croisée avec gri
lle de recherche pour les hyperparamètres.
print('\n\nITÉRATION 1\n')
print('\nNombre de caractéristiques :\n', X_train.shape[1])
```

```
# Réaliser une validation croisée avec grille de recherche pour les
hyperparamètres
cv_valid = GridSearchCV(estimator=model, param_grid=hyperparams, cv
=cv_folds, scoring=scoring, iid=False)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
model = cv_valid.best_estimator_
print('\nMeilleurs hyperparamètres: \n', best_params)
print('\nScore = \n', best_score)
print('\nImportance des caractéristiques : \n', model.feature_impor
tances_)

print(X_train.shape)
# Sélectionner les caractéristiques les plus importantes
features_new = SelectFromModel(model, threshold=0.1, prefit=True, m
ax_features=3)
X_train = features_new.transform(X_train)
X_test = features_new.transform(X_test)
print(X_train.shape)

# Réaliser une deuxième itération de la validation croisée avec gri
lle de recherche pour les hyperparamètres.
print('\n\nITÉRATION 2\n')
print('\nNombre de caractéristiques :\n', X_train.shape[1])
# Réaliser une nouvelle validation croisée avec grille de recherche
pour les hyperparamètres
cv_valid = GridSearchCV(estimator=model, param_grid=hyperparams, cv
=cv_folds, scoring=scoring, iid=False)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
model = cv_valid.best_estimator_
print('\nMeilleurs hyperparamètres: \n', best_params)
print('\nScore = \n', best_score)
print('\nImportance des caractéristiques : \n', model.feature_impor
tances_)

# Entraîner le modèle final avec toutes les données d'entraînement
model.fit(X_train,y_train)
```

```
(297, 9)
(297, 9)
Data ready


ITÉRATION 1


Nombre de caractéristiques :
 9

Meilleurs hyperparamètres:
 {'max_depth': 4, 'min_samples_split': 10}

Score =
 0.7270538693786748

Importance des caractéristiques :
 [0.0255661  0.52284761 0.         0.04807339 0.1180450
9 0.1492882
 0.1361796  0.         0.        ]
(297, 9)
(297, 3)


ITÉRATION 2


Nombre de caractéristiques :
 3

Meilleurs hyperparamètres:
 {'max_depth': 2, 'min_samples_split': 2}

Score =
 0.7329334180835849

Importance des caractéristiques :
 [0.86004273 0.13995727 0.        ]

Out[13]:

DecisionTreeClassifier(class_weight=None, criterion='gi
ni', max_depth=2,
                       max_features=None, max_leaf_node
s=None,
                       min_impurity_decrease=0.0, min_i
mpurity_split=None,
                       min_samples_leaf=1, min_samples_
split=2,
                       min_weight_fraction_leaf=0.0, pr
esort=False,
                       random_state=None, splitter='bes
t')
```

In [14]:

```
# On teste l'algorithme final en prédisant de nouvelles données.
y_pred = model.predict(X_test)

# On évalue les prédictions de l'algorithme final.
auc = roc_auc_score(y_test, y_pred)

# On affiche le résultat
print('\nTest AUC = ', auc)
```

```
Test AUC =  0.6603299856527978
```

EXEMPLE "INTÉGRÉE"

In [15]:

```
import warnings
warnings.filterwarnings("ignore")
warnings.filterwarnings(action='ignore',category=DeprecationWarning
)
warnings.filterwarnings(action='ignore',category=FutureWarning)

# ----------------------------------------------------------------
------------
# ÉTAPE 1 : importer les librairies utiles
# ----------------------------------------------------------------
------------

# Importer les librairies utiles pour l'analyse
import pandas as pd
import numpy as np


# ----------------------------------------------------------------
------------
# ÉTAPE 2 : importer les fonctions utiles
# ----------------------------------------------------------------
------------

# Importer les fonctions de prétraitement
from sklearn.preprocessing import StandardScaler

# Importer une fonction qui nous permette de construire aléatoireme
nt les ensembles "Entraînement" et "Test"
from sklearn.model_selection import train_test_split

# Importer le modèle de régression logistique de sklearn
from sklearn.linear_model import LogisticRegression

# Importer la fonction de validation croisée
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Importer la fonction permettant d'afficher le rapport de classifi
cation
from sklearn.metrics import roc_auc_score

np.random.seed(42)

# ----------------------------------------------------------------
------------
# ÉTAPE 3 : importer et préparer le jeu de données
# ----------------------------------------------------------------
------------

# Importons un ensemble de données
data = pd.read_csv('../data/sim_data_signature_small.csv')
data = data.dropna()
```

```
# Colonnes correspondant à des caractéristiques
features_cols = ['PSQ_SS', 'PHQ9TT', 'CEVQOTT', 'DAST10TT', 'AUDITT
T', 'STAIYTT', 'AGE', 'SEXE', 'SES']

# Données importées (X: caractéristiques, y: cible)
X = data.loc[:, features_cols]
y = data['WHODASTTB']

# Séparation en données d'entraînement et en données de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2)

# Standatdisation des entrées
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

print('Data ready')

# ----------------------------------------------------------------
------------
# ÉTAPE 4 : définir et entraîner le modèle
# ----------------------------------------------------------------
------------

# Définir le modèle
#model = LogisticRegression()
model = LogisticRegression(penalty='l1')

# Définir les hyperparamètres
hyperparams = {'C':[.0001, .001, .01, .1, 1, 10, 100, 1000, 10000]}

# Définir les plus de la validation croisée
cv_folds = StratifiedKFold(n_splits=5, random_state=42)

# Définir le type de score utilisé pour sélectionner les hyperparam
ètres dans la validation croisée
scoring='roc_auc'

# Réaliser la validation croisée avec grille de recherche pour les
 hyperparamètres.
cv_valid = GridSearchCV(estimator=model, param_grid=hyperparams, cv
=cv_folds, scoring=scoring, iid=False)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
model = cv_valid.best_estimator_
print('\nMeilleurs hyperparamètres: \n', best_params)
print('\nScore = \n', best_score)
print('\nImportance des caractéristiques : \n', model.coef_)

# Entraîner le modèle final avec toutes les données d'entraînement
model.fit(X_train,y_train)
```

```
Data ready

Meilleurs hyperparamètres:
 {'C': 0.1}

Score =
 0.7475011918004131

Importance des caractéristiques :
 [[0.         0.80137513 0.         0.         0.
0.00362856
  0.         0.         0.        ]]
```

Out[15]:

```
LogisticRegression(C=0.1, class_weight=None, dual=Fals
e, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None,
max_iter=100,
                   multi_class='warn', n_jobs=None, pen
alty='l1',
                   random_state=None, solver='warn', to
l=0.0001, verbose=0,
                   warm_start=False)
```

In [16]:

```
# On teste l'algorithme final en prédisant de nouvelles données.
y_pred = model.predict(X_test)

# On évalue les prédictions de l'algorithme final.
auc = roc_auc_score(y_test, y_pred)

# On affiche le résultat
print('\nTest AUC = ', auc)
```

```
Test AUC =  0.6406025824964132
```

7. CLASSIFICATION MULTICLASSE

7. CLASSIFICATION MULTICLASSE

Chacun des algorithmes de classification présentés peuvent fonctionner avec plus de deux classes.

Au moment d'évaluer le modèle (pour sélectionner les hyperparamètres ou pour évaluer le modèle final), la matrice de confusion ressemble à ceci.

Exemple pour trois classes :

Pour la classe 1		VALEURS RÉELLES		
		Classe 1	Classe 2	Classe 3
	Classe 1	VP	FP	FP

7. CLASSIFICATION MULTICLASSE

On a alors trois option pour calculer le score global:

- Pondéré (*weighted*)
  - Les scores sont calculés pour chaque classe séparément.
  - On utilise alors une moyenne pondérée en fonction du nombre de valeurs réelles positives.
  - Plus de poids pour les classes ayant plus d'exemples.

- Micro (*micro*)
  - Les scores sont calculés pour toutes les classes simultanément (nombres totaux de VP, FP, VN, FN.

- Macro (*macro*)
  - Les scores sont calculés pour chaque classe séparément.
  - Les classes ayant moins d'exemplaires sont aussi importantes que les classes ayant beaucoup d'exemplaires.

7. CLASSIFICATION MULTICLASSE

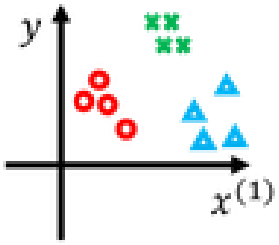
Pour entraîner le modèle, la stratégie est différente selon le type d'algorithme.

- Régression logistique
  - Un contre tous (*one vs rest*) : *multi_class='ovr''*
- Machines à vecteurs de supports
  - Un contre un (*one vs one*)
- Arbres de décisions
  - Extension naturelle
- Forêts aléatoires
  - Extension naturelle
- Boosting de gradient
  - Extension naturelle
- Classification naïve bayesienne
  - Multinomiale : utiliser *MultinomialNB()* *plutôt que GaussianNB()*
- K plus proches voisins
  - Extension naturelle

7. CLASSIFICATION MULTICLASSE

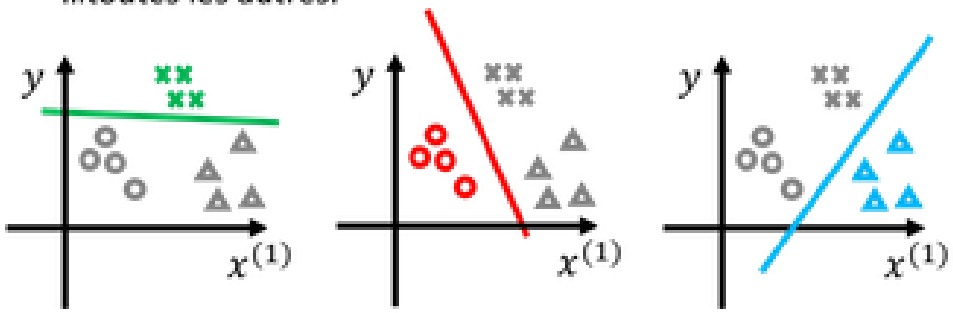
Régression logistique

- **Un contre tous (*one vs rest*) : *multi_class='ovr'***



On applique l'algorithme une fois par paire de classes.

- Chaque fois, on cherche à séparer une classe de…
  …toutes les autres.



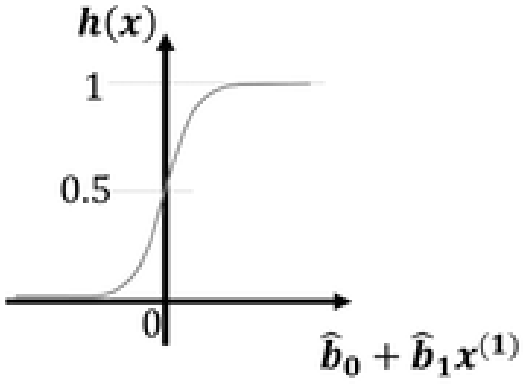
Frontière 1:	***Frontière 2:***	***Frontière 3:***
$\hat{b}_0^{(1)} + \hat{b}_1^{(1)} x^{(1)}$	$\hat{b}_0^{(2)} + \hat{b}_1^{(2)} x^{(1)}$	$\hat{b}_0^{(3)} + \hat{b}_1^{(3)} x^{(1)}$

Rappelons qu'à chaque fois, la véritable sortie est en fait :

$$h^{(1)}(x) = \frac{1}{1 + e^{-\left(\hat{b}_0^{(1)} + \hat{b}_1^{(1)} x^{(1)}\right)}}$$

$$h^{(2)}(x) = \frac{1}{1 + e^{-\left(\hat{b}_0^{(2)} + \hat{b}_1^{(2)} x^{(1)}\right)}}$$

$$h^{(3)}(x) = \frac{1}{1 + e^{-\left(\hat{b}_0^{(3)} + \hat{b}_1^{(3)} x^{(1)}\right)}}$$



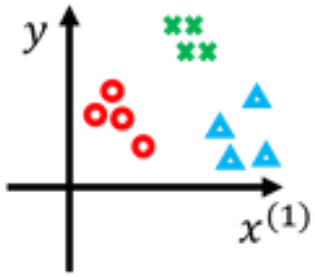
Pour classer un nouvel exemplaire, on calcule la valeur $h(x)$ pour chacune des trois frontières et on le classe dans la classe ayant la valeur la plus élevée.

$$classe = \underset{i}{\operatorname{argmax}}\, h^{(i)}(x)$$

7. CLASSIFICATION MULTICLASSE

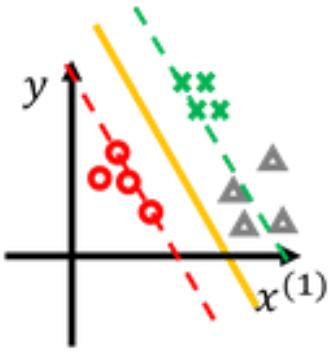
Machines à vecteurs de supports

- **Un contre un (*one vs one*)**

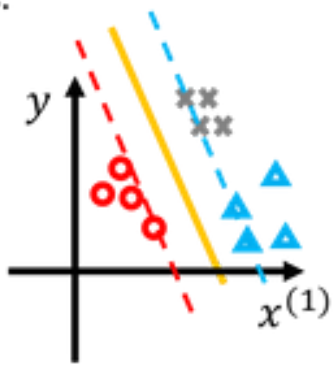


On applique l'algorithme une fois par paire de classes.

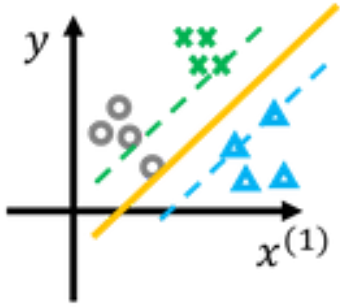
- Chaque fois, on cherche à séparer une classe de…
  …toutes les autres.



$\boldsymbol{Frontière\ 1}$:
$\widehat{b}_0^{(1)} + \widehat{b}_1^{(1)} x^{(1)}$



$\boldsymbol{Frontière\ 2}$:
$\widehat{b}_0^{(2)} + \widehat{b}_1^{(2)} x^{(1)}$



$\boldsymbol{Frontière\ 3}$:
$\widehat{b}_0^{(3)} + \widehat{b}_1^{(3)} x^{(1)}$

- Pour classer un nouvel exemplaire…:
  - On calcule la Classe choisie pour chaque classifieur binaire.
  - On additionne le nombre de votes pour chaque classe.
  - La classe ayant le plus de votes l'emporte.

In [17]:

```
import warnings
warnings.filterwarnings("ignore")
warnings.filterwarnings(action='ignore',category=DeprecationWarning
)
warnings.filterwarnings(action='ignore',category=FutureWarning)

# ----------------------------------------------------------------
------------
# ÉTAPE 1 : importer les librairies utiles
# ----------------------------------------------------------------
------------

import pandas as pd
import numpy as np

# ----------------------------------------------------------------
------------
# ÉTAPE 2 : importer les fonctions utiles
# ----------------------------------------------------------------
------------

# Importer les fonctions de prétraitement
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Importer une fonction qui nous permette de construire aléatoireme
nt les ensembles "Entraînement" et "Test"
from sklearn.model_selection import train_test_split

# Importer le modèle de régression logistique de sklearn
from sklearn.linear_model import LogisticRegression
from sklearn import svm
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoosti
ngClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import KNeighborsClassifier

# Importer la fonction de validation croisée
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Importer la fonction permettant d'afficher le rapport de classifi
cation
from sklearn.metrics import f1_score, make_scorer

np.random.seed(42)

# ----------------------------------------------------------------
------------
# ÉTAPE 3 : importer et préparer le jeu de données
# ----------------------------------------------------------------
------------
```

```
# Importons un ensemble de données
data = pd.read_csv('../data/sim_data_signature_small.csv')
# Retirer les lignes comportant des données manquantes
data = data.dropna()
# Retirer les sujets contrôles
data[data.CONTROL != 1]

features_cols = ['PSQ_SS', 'PHQ9TT', 'CEVQOTT', 'DAST10TT', 'AUDITT
T', 'STAIYTT', 'AGE', 'SEXE', 'SES']

X = data.loc[:, features_cols]
y = data['CIMDX']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2)

scaler = MinMaxScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

print('Data ready')

# ----------------------------------------------------------------
------------
# ÉTAPE 4 : définir et entraîner le modèle
# ----------------------------------------------------------------
------------

# Définir le modèle
model_logistic = LogisticRegression(multi_class='ovr')
model_linear = svm.SVC(kernel='linear')
model_rbf = svm.SVC(kernel='rbf')
model_tree = DecisionTreeClassifier()
model_forest = RandomForestClassifier()
model_boosting = GradientBoostingClassifier()
model_nb = MultinomialNB()
model_knn = KNeighborsClassifier()

# Définir les hyperparamètres
hyperparams_logistic = {'C':[.0001, .001, .01, .1, 1, 10, 100, 1000
, 1000]}
hyperparams_linear = {'C':[.0001, .001, .01, .1, 1, 10, 100, 1000,
1000]}
hyperparams_rbf = {'C':[.0001, .001, .01, .1, 1, 10, 100, 1000, 100
0], 'gamma':[.0001, .001, .01, .1, 1, 10, 100, 1000, 1000]}
hyperparams_tree = {'max_depth':[1, 2, 4, 8, 12, 16], 'min_samples_
split':[2, 4, 6, 8, 10, 12, 14, 16]}
hyperparams_forest = {'n_estimators':[5], \
                      'max_depth':[1, 2, 4, 8, 12, 16], \
                      'min_samples_split':[2, 4, 6, 8, 10, 12, 14,
16], \
                      'max_features': [1, 3, 5, 7, 9]}
hyperparams_boosting = {'learning_rate':[0.1, 0.2, 0.4, 0.8], \
```

```
                      'n_estimators':[5], \
                      'max_depth':[1, 2, 4, 8, 12, 16], \
                      'min_samples_split':[2, 4, 6, 8, 10, 12, 14,
16], \
                      'max_features': [1, 3, 5, 7, 9]}
hyperparams_nb = {'alpha':[1]}
hyperparams_knn = {'n_neighbors':[1, 2, 4, 8, 16], 'weights':['unif
orm', 'distance']}


cv_folds = StratifiedKFold(n_splits=3, random_state=42)

scoring = make_scorer(f1_score , average='macro')

cv_valid = GridSearchCV(estimator=model_logistic, param_grid=hyperp
arams_linear, cv=cv_folds, scoring=scoring)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
print('\nlogistic')
print('\nMeilleur paramètre: ', best_params)
print('\nScore = ', best_score)

cv_valid = GridSearchCV(estimator=model_linear, param_grid=hyperpar
ams_linear, cv=cv_folds, scoring=scoring)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
print('\nSVC_linear')
print('\nMeilleur paramètre: ', best_params)
print('\nScore = ', best_score)

cv_valid = GridSearchCV(estimator=model_rbf, param_grid=hyperparams
_rbf, cv=cv_folds, scoring=scoring)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
print('\nSVC_rbf')
print('\nMeilleur paramètre: ', best_params)
print('\nScore = ', best_score)

cv_valid = GridSearchCV(estimator=model_tree, param_grid=hyperparam
s_tree, cv=cv_folds, scoring=scoring)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
print('\ntree')
print('\nMeilleur paramètre: ', best_params)
print('\nScore = ', best_score)

cv_valid = GridSearchCV(estimator=model_forest, param_grid=hyperpar
ams_forest, cv=cv_folds, scoring=scoring)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
```

```
print('\nforest')
print('\nMeilleur paramètre: ', best_params)
print('\nScore = ', best_score)

cv_valid = GridSearchCV(estimator=model_boosting, param_grid=hyperp
arams_boosting, cv=cv_folds, scoring=scoring)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
print('\nboosting')
print('\nMeilleur paramètre: ', best_params)
print('\nScore = ', best_score)

cv_valid = GridSearchCV(estimator=model_nb, param_grid=hyperparams_
nb, cv=cv_folds, scoring=scoring)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
print('\nnaive bayes')
print('\nMeilleur paramètre: ', best_params)
print('\nScore = ', best_score)

cv_valid = GridSearchCV(estimator=model_knn, param_grid=hyperparams
_knn, cv=cv_folds, scoring=scoring)
cv_valid.fit(X_train, y_train)
best_params = cv_valid.best_params_
best_score = cv_valid.best_score_
print('\nknn')
print('\nMeilleur paramètre: ', best_params)
print('\nScore = ', best_score)

print("\n\n Analyse terminée")
```

```
Data ready

logistic

Meilleur paramètre:  {'C': 1000}

Score =  0.272270737503952

SVC_linear

Meilleur paramètre:  {'C': 10}

Score =  0.27650306407984937

SVC_rbf

Meilleur paramètre:  {'C': 1000, 'gamma': 1}

Score =  0.3809556829162604

tree

Meilleur paramètre:  {'max_depth': 12, 'min_samples_spl
it': 6}

Score =  0.28160113589197633

forest

Meilleur paramètre:  {'max_depth': 8, 'max_features':
9, 'min_samples_split': 2, 'n_estimators': 5}

Score =  0.36359885148800425

boosting

Meilleur paramètre:  {'learning_rate': 0.8, 'max_dept
h': 8, 'max_features': 3, 'min_samples_split': 2, 'n_es
timators': 5}

Score =  0.4012201909204111

naive bayes

Meilleur paramètre:  {'alpha': 1}

Score =  0.20031841689265756

knn

Meilleur paramètre:  {'n_neighbors': 1, 'weights': 'uni
form'}

Score =  0.43204561551884185
```

Analyse terminée