Arbres de décision et forêts d'arbres décisionnels

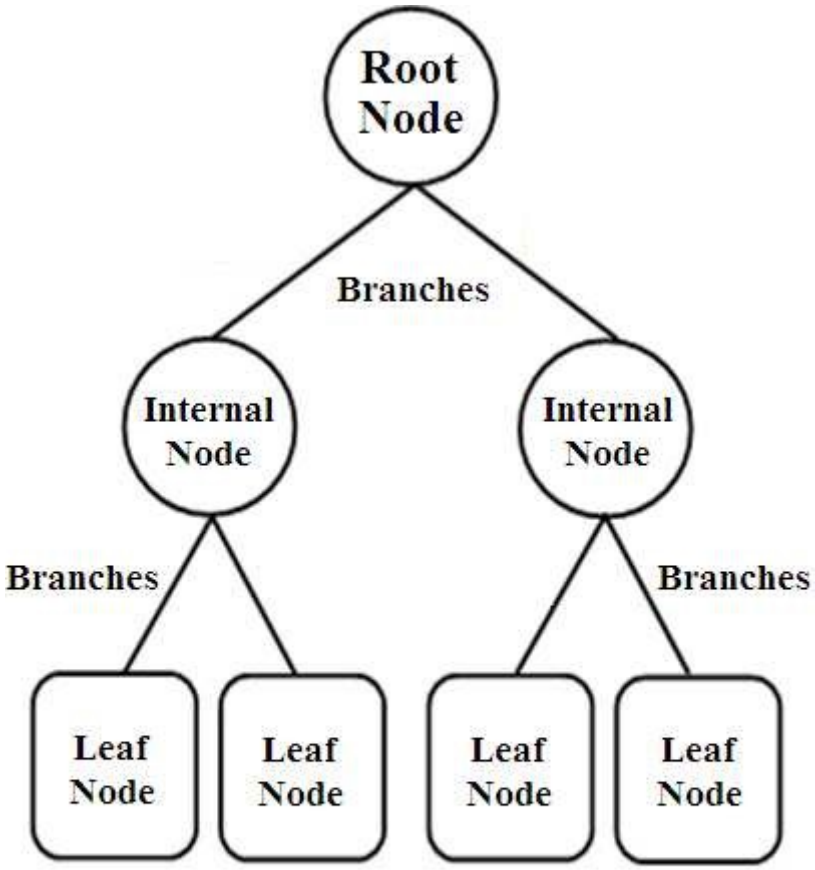
I. Arbres de décision

1) Principe

Les arbres de décisions sont une technique d'apprentissage supervisé. Ils permettent de bâtir des modèles de classification ou de régression sous forme de structures en arbres.

Ils cassent les jeux de donnés en sous-ensembles de plus en plus petit tandis qu'un arbre de décision associé est développé par incrément.

Le résultat final est un arbre avec des noeuds de décision (*decision nodes*) et des feuilles (*leaf nodes*). Voir figure ci-dessous :



(a)

Contrairement aux régressions linéaires que l'on a vues précédemment, les arbres de décision sont des méthodes d'apprentissage non paramétriques.

Note : Les arbres de décision sont au fondement des modèles basés sur des arbres tels que les *random forest* et les *gradient boosting*.

2) Fonctionnement

Le modèle entraîné sur des données d'apprentissage va créer un ensemble de séquences de règles de décision qu'il déduit à partir de la donnée.

Tout comme pour les modèles de régression linéaire, l'objectif est de prédire la valeur d'une variable cible à partir de nouvelles valeurs de variables *features*.

L'arbre va venir approximer la cible en utilisant cet ensemble de règle (du type "si, alors, sinon...").

Ceci fonctionne pour des données qualitatives ainsi que des données numériques.

3) Implémentation en python

Pour implémenter un arbre de décision en python on va utiliser `DecisionTreeRegressor()` du module *tree* de *sklearn*.

On crée ensuite un objet qui va accueillir notre instance de `DecisionTreeRegressor()` .

Cet objet est notre régresseur. On le *fit* aux données comme un modèle linéaire et il est bon pour de la prédiction avec un `.predict` .

```
from sklearn import tree
tree_reg = tree.DecisionTreeRegressor()

tree_reg.fit(X_train, Y_train)

Y_predit = tree_reg.predict(X_train)
```

4) Intérêt

Les arbres de décision ont plusieurs intérêts. Ils sont intéressants lorsqu'on est dans le cas où l'on souhaite comprendre la séquence de décisions prises par le modèle réalisant la prédiction.

- Ils sont réputés simples à comprendre et à visualiser.
- Ils nécessitent peu de préparation de données.
- Leur coût d'utilisation est logarithmique (donc il évolue moins brusquement qu'un coût exponentiel).
- Ils sont adaptés aux données numériques et catégorielles.
- Ils sont capables de traiter des problèmes de classification multi-classe (plusieurs labels possibles).
- Comme dit au début, ce sont des modèles transparents. On parle de **modèle en boîte blanche** par opposition aux modèles en boîte noire à faible explicabilité. Avec les arbres de décision, le résultat est facile à conceptualiser et à visualiser.

5) Limites

Les arbres décisionnels peuvent avoir les défauts de leurs qualités : plus ils sont complexes et plus ils expliquent à la perfection les données d'apprentissage. On se retrouve alors dans un cas de **sur-apprentissage** (*overfitting*) où le modèle ne sert plus à rien pour expliquer de nouvelles données.

Pour diminuer ce risque on peut jouer sur la profondeur maximale du modèle (la *max_depth*) et un et nombre minimal d'exemples par feuille (*min_samples_leaf*).

On peut aussi pratiquer la **validation croisée** et privilégier les datasets les plus conséquents possible.

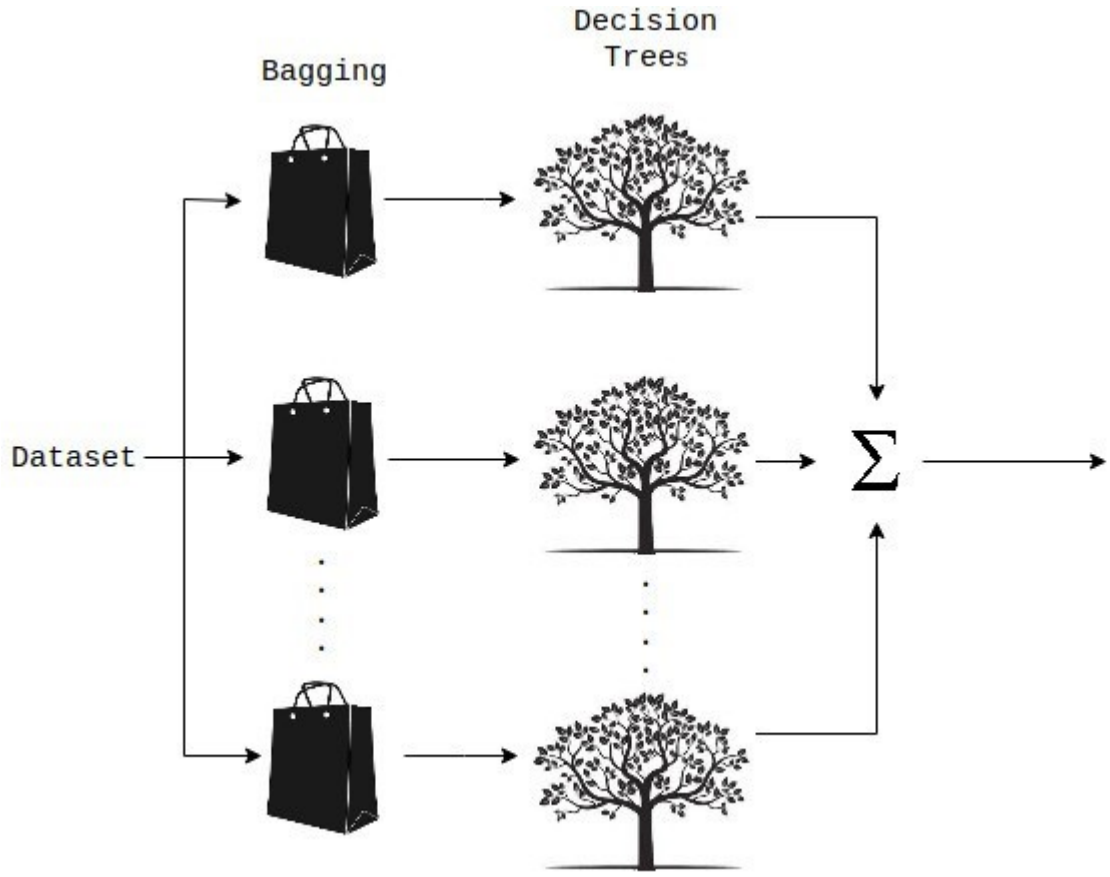
Autre soucis possible : les arbres générés ne sont pas toujours **équilibrés** et le temps de parcours de l'abre par le modèle lors de la prédiction n'est alors plus logarithmique.

Il est recommandé d'ajuster notre bdd avant construction afin d'éviter qu'une classe ne domine les autres (en termes de nombre d'exemples d'apprentissage).

II. Forêts d'arbres décisionnels

1) Principe

Les forêts d'arbres décisionnels (*random forests* en anglais) sont également des modèles adaptés à la régression comme à la classification. En réalité il ne s'agit pas d'un simple algorithme mais plutôt ce qu'on appelle un meta-algorithme d'apprentissage dit "ensembliste". Les forêts utilisent plusieurs arbres de décision ainsi qu'une technique appellée *bootstrap aggregation* ou encore *bagging*.



Cette technique de *bagging* consiste à entraîner chaque arbre décisionnel sur un jeu de données différent. Cette technique est conçue pour améliorer la stabilité et la précision de l'algorithme. Il réduit la variance et permet d'éviter le surapprentissage.

2) Fonctionnement

Le dataset est découpé en sous-ensembles aléatoires constitués d'échantillons (d'où le *random* dans le nom de la technique *random forest*).

On entraîne un modèle sur chaque sous-ensemble. On a autant de modèles (d'arbres) que de sous-ensembles.

On combine les résultats des modèles via un système de vote et cela donne un "super-résultat" final

L'idée est de construire ainsi un modèle plus robuste que ses composants (arbres) pris séparément.

3) Implémentation en python

Afin d'implémenter une forêt de décision en python, il faut récupérer le régresseur `RandomForestRegressor` dans le sous-module *ensemble* de *sklearn*.

Tout comme pour les autres modèles similaires, on instancie un objet régresseur : `rndfor_reg = RandomForestRegressor()`

Ensuite, on peut utiliser une grille de paramètres pour le modèle (nombre d'arbres et de features gérées), du type : `param_grid = {'n_estimators' : [3, 10, 30],'max_features' : [2, 4, 6, 8]}`

Cette grille permet de cross-valider les différentes combinaisons entre estimateurs et nombre maximal de features.

Pour ce faire, on va employer la méthode GridSearchCV disponible dans *model_selection* :

```
from sklearn.model_selection import GridSearchCV
gridlog = GridSearchCV(rndfor_reg, param_grid, cv = 5)
```

Ci-dessus, gridlog est un objet GridSearchCV, du coup on a accès à un certain nombre d'attributs très utiles. Mais il faut d'abord fit notre objet GridSearchCV à la donnée : `gridlog.fit(X_train, Y_train.values.ravel())`

Ensuite, on peut trouver les paramètres optimaux pour le régresseur avec `gridlog.best_params_` .

On peut aussi afficher toutes les données issues du `GridSearchCV` et ce pour chaque combinaison des paramètres de la variable `param_grid` . On les affiche avec `gridlog.cv_results_` . On obtient alors les écart-type, les moyennes, les scores (dont la stratégie est à définir avec le paramètre scoring)...

Pour la prédiction, on peut directement utiliser le meilleur estimateur (celui qui est créé avec les meilleurs paramètres) :

```
#On crée un modèle spécial à partir de nos paramètres optimisés :
best_rndfor_reg = gridlog.best_estimator_

#Le ravel sur le Y est nécessaire pour éviter un warning. On fit notre
modèle :
best_rndfor_reg.fit(X, Y.values.ravel())

#On prédit :
Y_predit = best_rndfor_reg.predict(X_test)
```

4) Intérêt

Les forêts décisionnelles sont pratiques pour gérer des relations complexes et non linéaires entre les variables.

Ils sont suffisamment robustes pour gérer des datasets de grande dimensions.

Ils supportent plutôt bien les valeurs manquantes.

Ils sont puissants et donnent des résultats précis.

Ils peuvent être entraînés rapidement car les arbres décisionnels qui les composent ne dépendent pas les uns des autres et leur apprentissage peut donc être parallélisé.

5) Limites

Malgré l'intérêt du *bagging* les modèles de forêts décisionnelles restent connus pour présenter des risques de surapprentissage et donc pour être moins précis pour les problèmes de régression.

Aussi, un grand nombre d'arbres peut rendre l'algorithme trop lent et inefficace pour de la prédiction en temps réel. En général ces modèles sont entraînés plutôt rapidement mais prédisent lentement une fois entraînés.