TP3

Les arbres de décision

Les arbres de décision sont des méthodes d'apprentissage non paramétriques utilisées pour des problèmes de classification et de régression. L'objectif est de créer un modèle qui prédit les valeurs de la variable cible, en se basant sur un ensemble de séquences de règles de décision déduites à partir des données d'apprentissage. L'arbre approxime donc la cible par une succession de règles `if-then-else`. Ce paradigme s'applique aussi bien à des données catégorielles qu'à des données numériques. Plus l'arbre généré est complexe, mieux le modèle « explique » les donnés d'apprentissage mais plus le risque de sur-apprentissage (*over-fitting*) est élevé.
Les arbres de décision ont plusieurs **avantages** qui les rendent intéressants dans des contextesoù il est utile de comprendre la séquence de décisions prise par le modèle :

- Ils sont simples à comprendre et à visualiser.
- Ils nécessitent peu de préparation des données (normalisation, etc.).
- Le coût d'utilisation des arbres est logarithmique.
- Ils sont capables d'utiliser des données catégorielles et numériques.
- Ils sont capables de traiter des problèmes multi-classe.
- Modèle en boîte blanche : le résultat est facile à conceptualiser et à visualiser.

Ces modèles présentent néanmoins deux **désavantages** majeurs :

- Sur-apprentissage : parfois les arbres générés sont trop complexes et généralisent mal. Choisir des bonnes valeurs pour les paramètres profondeur maximale (`max_depth`) et nombre minimal d'exemples par feuille (`min_samples_leaf`) permet d'éviter ce problème.
- Il peut arriver que les arbres générés ne soient pas équilibrés (ce qui implique que le temps de parcours n'est plus logarithmique). Il est donc recommandé d'ajuster la base de données avant la construction, pour éviter qu'une classe domine largement les autres (en termes de nombre d'exemples d'apprentissage).

I/ Arbres pour la classification

Dans scikit-learn, la classe sklearn.tree.DecisionTreeClassifier permet de réaliser une classification multi-classe à l'aide d'un arbre de décision.
On commence par importer les bons modules et construire l'objet arbre :

```
from sklearn import tree
clf = tree.DecisionTreeClassifier()
```

Pour l'exemple, nous pouvons définir un jeu de données minimaliste (deux points, chacun dans une classe) :

```
X = [[0, 0], [1, 1]]
y = [0, 1]
```

L'arbre se construit comme d'habitude à l'aide de la méthode `.fit(X, y)` :

```
clf = clf.fit(X, y)
```
La prédiction sur de nouveaux échantillons se fait de façon habituelle avec .predict(X) :
```
clf.predict([[2., 2.]])
```
On peut aussi prédire la probabilité de chaque classe pour un échantillon (qui est calculée comme la fraction de données d'apprentissage dans chaque feuille) :
```
clf.predict_proba([[2., 2.]])
```

II/ Classification des données Iris

DecisionTreeClassifier est capable de gérer des problèmes de classification à plusieurs classes (par exemple, avec les étiquettes 0, 1, … K-1). Dans cet exemple nous allons travailler avec la base de données Iris, facilement accessible dans sklearn. Cette base contient 150 instances d'iris (un type de plante, chaque observation décrit sa morphologie). L'objectif est de classer chaque instance en une des trois catégories : *Iris setosa*, *Iris virginica* ou *Iris versicolor*.
Une des classes est linéairement séparable par rapport aux deux autres, mais les deux autres ne sont pas séparables une par rapport à l'autre.

Les attributs du jeu de données sont :

- longueur de sépale,
- largeur de sépale,
- longueur de pétale,
- largeur de pétale,
- classe : Iris Setosa, Iris Versicolor ou Iris Virginica.

Un échantillon : (4.9,3.6,1.4,0.1, "Iris-setosa")

Le jeu de données Iris étant très commun, scikit-learn propose une fonction native permettant de le charger en mémoire :

```
from sklearn.datasets import load_iris

iris = load_iris()
X, y = iris.data, iris.target
```

Question

Calculer les statistiques (moyenne et écart-type) des quatre variables explicatives :
longueur de sépale, largeur de sépale, longueur de pétale et largeur de pétale.

Question

Combien y a-t-il d'exemples de chaque classe ?

Avant de construire le modèle, séparons le jeu de données en deux : 70% pour l'apprentissage, 30% pour le test.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7,
random_state=0)
```

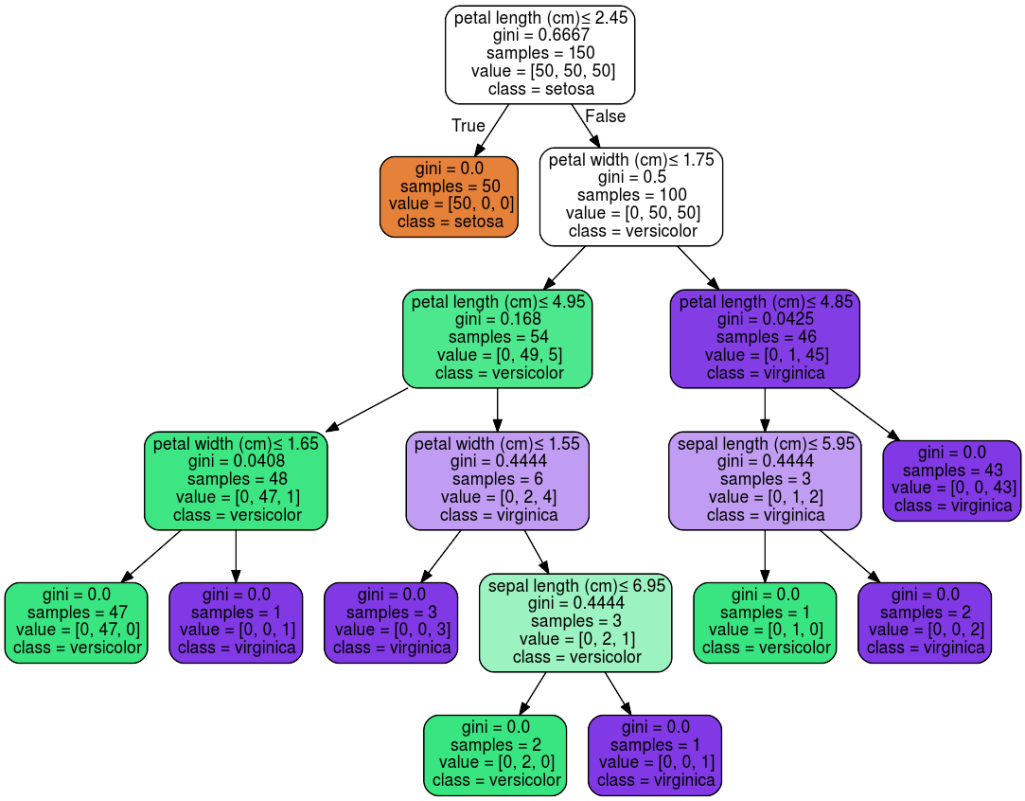
Nous pouvons désormais construire un arbre de décision sur ces données :

```
from sklearn import tree
clf = tree.DecisionTreeClassifier()
clf.fit(X_train, y_train)
```

Une fois l'apprentissage terminé, nous pouvons visualiser l'arbre, soit avec matplotlib en passant par la méthode plot_tree. Par exemple, avec matplotlib :

```
tree.plot_tree(clf, filled=True)
```

L'image générée doit ressembler à ceci :



Une fois le modèle construit, il est possible de l'utiliser pour la prédiction sur de nouvelles données :

```
clf.predict(X_test)
```

On peut de cette façon calculer le score en test :

```
clf.score(X_test, y_test)
```

Question :

Changez les valeurs de parametres max_depth et min_samples_leaf. Que constatez-vous ?
Voir la documentation des arbres de décision. Le paramètre max_depth est un seuil sur la profondeur maximale de l'arbre. Le paramètre min_samples_leaf donne le nombre minimal d'échantillons dans un nœud feuille. Ils permettent de mettre des contraintes sur la construction de l'arbre et donc de contrôler indirectement le phénomène de sur-apprentissage.
Exemple d'utilisation :

```
clf = tree.DecisionTreeClassifier(max_depth = 3)
```

ou bien :

```
clf = tree.DecisionTreeClassifier(min_samples_leaf = 20)
```

Voici l'arbre obtenu avec min_samples_leaf=20. Il a plus d'éléments dans les feuilles, donc moins de nœuds et l'arbre est moins profond.

Question :

Le problème ici étant particulièrement simple, refaites une division apprentissage/test avec 5% des données en apprentissage et 95% test.

Calculez le taux d'éléments mal classifiés sur l'ensemble de test.

Faites varier (ou mieux, réalisez une recherche par grille avec GridSearchCV) les valeurs des paramètres max_depth et min_samples_leaf pour mesurer leur impact sur ce score.

III/ Affichage de la surface de décision

Pour une paire d'attributs, c'est-à-dire pour des observations en deux dimensions, nous pouvons visualiser la surface de décision en 2 dimensions. D'abord on discrétise le domaine bidimensionnel avec un pas constant et ensuite on évalue le modèle sur chaque point de la grille.

Dans cet exemple, nous ne gardons que la longueur et la largeur des pétales.

```
import numpy as np
import matplotlib.pyplot as plt

# Paramètres
n_classes = 3
plot_colors = "bry" # blue-red-yellow
plot_step = 0.02

# Choisir les attributs longueur et largeur des pétales
pair = [2, 3]

# On ne garde seulement les deux attributs
X = iris.data[:, pair]
y = iris.target

# Apprentissage de l'arbre
clf = tree.DecisionTreeClassifier().fit(X, y)

# Affichage de la surface de décision
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
```

```
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step), np.arange(y_min,
y_max, plot_step))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.xlabel(iris.feature_names[pair[0]])
plt.ylabel(iris.feature_names[pair[1]])
plt.axis("tight")

# Affichage des points d'apprentissage
for i, color in zip(range(n_classes), plot_colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[i],
cmap=plt.cm.Paired)
plt.axis("tight")
plt.suptitle("Decision surface of a decision tree using paired features")
plt.legend()
plt.show()
```

Question :

Refaire l'affichage pour les autres paires d'attributs. Sur quelle paire la séparation entre les classes est la plus marquée ?

IV/ Arbres de décision pour la régression

Pour la régression avec les arbres de décision, scikit-learn offre la classe `DecisionTreeRegressor`. Comme pour la classification, la méthode `fit(...)` prend en entrée le paramètre X (attributs des observations).
Attention : les `y` ne sont pas des étiquettes de classes mais des valeurs réelles.

```
from sklearn import tree

X = [[0, 0], [2, 2]]
y = [0.5, 2.5]
clf = tree.DecisionTreeRegressor()
clf = clf.fit(X, y)
clf.predict([[1, 1]])
```

Dans l'exemple suivant nous allons construire un signal sinusoïdal affecté par un bruit blanc et nous allons apprendre un arbre de régression sur ces données d'apprentissage.

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.tree import DecisionTreeRegressor

# Créer les données d'apprentissage
np.random.seed(0)
X = np.sort(5 * np.random.rand(80, 1), axis=0)
y = np.sin(X).ravel()
```

```
fig = plt.figure(figsize=(12, 4))
fig.add_subplot(121)
plt.plot(X, y)
plt.title("Signal sinusoïdal pur")

# On ajoute un bruit aléatoire tous les 5 échantillons
y[::5] += 3 * (0.5 - np.random.rand(16))
fig.add_subplot(122)
plt.plot(X, y)
plt.title("Signal sinusoïdal bruité")
```

L'objectif est de régresser ce signal `y` à partir des valeurs de `x`. Pour cela, nous utilisons un arbre de régression.

```
# Apprendre le modèle
reg = DecisionTreeRegressor(max_depth=2)
reg.fit(X, y)

# Prédiction sur la même plage de valeurs
X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]
y_pred = reg.predict(X_test)

# Affichage des résultats
plt.figure()
plt.scatter(X, y, c="darkorange", label="Exemples d'apprentissage")
plt.plot(X_test,    y_pred,    color="cornflowerblue",    label="Prédiction",
linewidth=2)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Régression par un arbre de décision")
plt.legend()
plt.show()
```

Question :

Changer la valeur du paramètre `max_depth`. Que se passe-t-il si on prend une valeur trop grande ? Trop petite ? Changer le taux d'éléments affectés par le bruit (le `y[::5]`). Quand tous les éléments sont affectés par le bruit, faut-il préférer une valeur élevée ou faible pour `max_depth` ?

Question :

Pour approfondir, chargez la base de données Diabètes du module `sklearn.datasets` et faire une partition aléatoire en partie apprentissage et partie test (70% apprentissage, 30% test). Construire un modèle d'arbre de regression sur cette base. Calculer l'erreur quadratique moyenne sur l'ensemble de test. Faire un *grid search* pour trouver la valeur du paramètre `max_depth` qui minimise cette erreur.