1 Introduction

Implémentation des arbres de décision avec la librairie Scikit-Learn (0.22.1) sous Python. Représentation graphique de l'arbre et appréhension des variables prédictives qualitatives.

Tous les ans, préalablement à chacune de mes séances sur machine avec les étudiants, je fais un travail de mise à jour des instructions et indications de résultats retranscrits dans ma fiche de TD (travaux dirigés). Il faut dire que les packages sous R et Python ne se soucient pas toujours de compatibilités descendantes ou ascendantes. Une instruction valable hier peut ne pas fonctionner aujourd'hui ou, pire, fournir un résultat différent parce que les paramètres par défaut ont été modifiés ou les algorithmes sous-jacents améliorés. La situation est moins critique lorsque des fonctionnalités additionnelles sont proposées. Encore faut-il les connaître. La veille technologique est indissociable de notre activité, et j'y passe vraiment beaucoup de temps.

Concernant ma séance consacrée aux arbres de décision sous Python justement, où nous utilisons la libraire Scikit-Learn (Decision Trees), j'avais pour habitude d'annoncer à mes étudiants qu'il n'était pas possible de disposer – simplement – d'une représentation graphique de l'arbre, à l'instar de ce que nous fournirait le package ''rpart.plot'' pour les arbres ''rpart'' sous R par exemple. La nécessité d'installer un outil externe (voir ''Random Forest et Boosting avec R et Python'', novembre 2015 ; section 4.3.3) rendait la manipulation rédhibitoire dans une séance où nous travaillons en temps (très) restreint avec des machines (très) protégées. Je me suis rendu compte récemment au détour d'une requête Google, assez heureuse je dois l'avouer, que la situation a évolué avec la version 0.21.0 de Scikit-Learn (Mai 2019). Nous allons vérifier cela dans ce tutoriel. Nous en profiterons pour étudier les manipulations à réaliser pour pouvoir appliquer les dits-arbres sur des variables prédictives (explicatives) catégorielles. L'outil ne sait pas les appréhender de manière native… pour l'instant (version 0.22.1, février 2020).

2 Construction et représentation d'un arbre avec Scikit-Learn

2.1 Vérification de la version de Scikit-Learn

Nous devrions commencer toute session d'analyse avec l'identification des versions des packages que nous utilisons. Concernant ''scikit-learn'', nous travaillons avec…

```
#vérification de la version de scikit-learn
import sklearn
print(sklearn.__version__)
```

```
0.22.1
```

2.2 Importation et expertise des données

Nous traitons un exemple très simple dans un premier temps. Nous utilisons la base ultra-connue ''Breast Cancer Wisconsin''. Nous cherchons à expliquer la variable ''classe'' décrivant la nature maligne (malignant) ou non (begnin) de cellules à partir de leurs caractéristiques (clump, ucellsize, …, mitoses ; 9 variables numériques).

```
#modifier le working directory
import os
os.chdir("... votre dossier de travail ...")

#importer les données - utilisation de la librairie pandas
import pandas
df = pandas.read_excel("breast.xlsx",sheet_name = 0)

#dimension du data frame
print(df.shape)
```

```
(699, 10)
```

Nous disposons de 699 observations et 10 variables. Voici les premières lignes de notre dataset.

```
#affichage des premières lignes
print(df.head())
```

```
   clump  ucellsize  ucellshape  ...  normnucl  mitoses     classe
0      4          2           2  ...         1        1     begnin
1      1          1           1  ...         1        1     begnin
2      2          1           1  ...         1        1     begnin
3     10          6           6  ...         7        1  malignant
4      4          1           1  ...         1        1     begnin
```

Nous affichons les informations sur le type des variables.

```
#information sur les variables
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 699 entries, 0 to 698
Data columns (total 10 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   clump       699 non-null    int64
 1   ucellsize   699 non-null    int64
 2   ucellshape  699 non-null    int64
 3   mgadhesion  699 non-null    int64
 4   sepics      699 non-null    int64
 5   bnuclei     699 non-null    int64
 6   bchromatin  699 non-null    int64
 7   normnucl    699 non-null    int64
 8   mitoses     699 non-null    int64
 9   classe      699 non-null    object
dtypes: int64(9), object(1)
```

La variable cible ''classe'' est la seule non-numérique, le type ''object'' lui est associé.

Nous affichons la fréquence absolue des classes…

```
#vérifier la distribution absolue des classes
print(df.classe.value_counts())
```

```
begnin       458
malignant    241
Name: classe, dtype: int64
```

… puis relative.

```
#la distribution relative
print(df.classe.value_counts(normalize=True))
```

```
begnin       0.655222
malignant    0.344778
Name: classe, dtype: float64
```

Ces informations sont importantes lorsque nous aurons à inspecter les résultats.

2.3 Partition en échantillons d'apprentissage et de test

Nous cherchons à appliquer le schéma type de l'analyse prédictive : scinder les données en échantillons d'apprentissage et de test, développer le modèle (estimer ses paramètres) sur le premier, en évaluer les performances sur le second à travers la confrontation des classes observées et prédites.

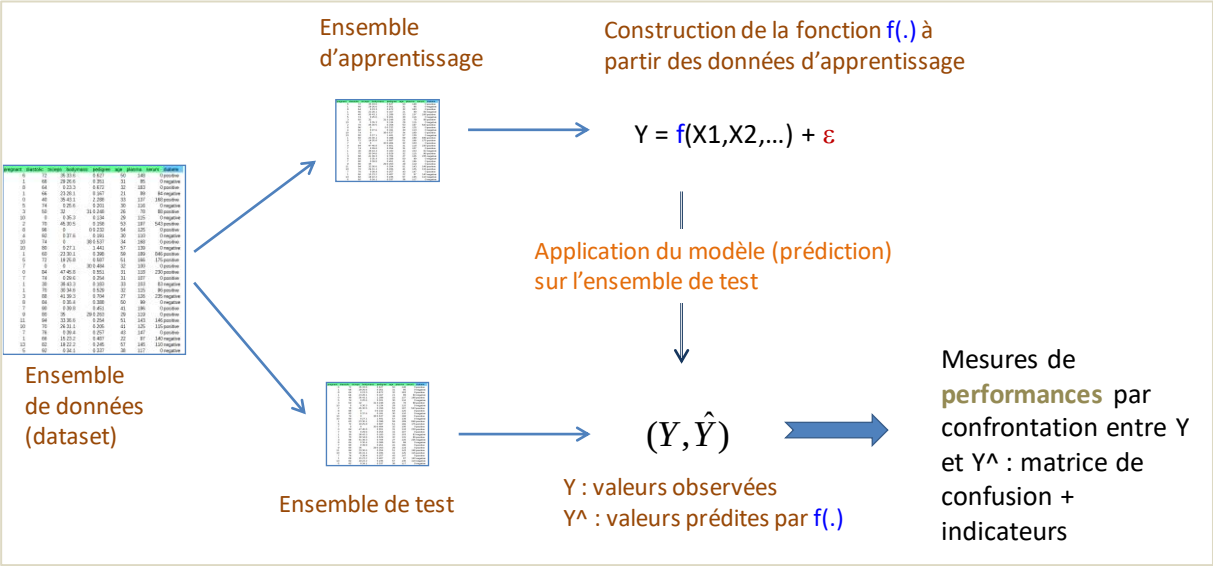


Figure 1 - Schéma type de travail en analyse prédictive

Nous souhaitons réserver 399 observations pour l'apprentissage et 300 pour le test, avec un échantillonnage stratifié (stratify) c.-à-d. respectant les proportions des classes dans les deux sous-ensembles. Nous fixons (random_state = 1) pour que l'expérimentation soit reproductible.

```
#subdiviser les données en échantillons d'apprentissage et de test
```

```
from sklearn.model_selection import train_test_split
dfTrain, dfTest = train_test_split(df,test_size=300,random_state=1,stratify=df.classe)
```

Nous vérifions les dimensions des données.

```
#vérification des dimensions
print(dfTrain.shape) #(399, 10)
print(dfTest.shape) #(300, 10)
```

Nous affichons les distributions relatives des classes en apprentissage…

```
#vérification des distributions en apprentissage
print(dfTrain.classe.value_counts(normalize=True))
```

```
begnin       0.654135
malignant    0.345865
Name: classe, dtype: float64
```

… et en test.

```
#vérification des distributions en test
print(dfTest.classe.value_counts(normalize=True))
```

```
begnin       0.656667
malignant    0.343333
Name: classe, dtype: float64
```

Les proportions sont respectées.

2.4 Instanciation et modélisation

Nous instancions un arbre de décision ''DecisionTreeClassifier'' de la librairie Sckit-Learn (https://scikit-learn.org/stable/modules/tree.html ; ''Classification'') avec deux paramètres : un sommet n'est pas segmenté s'il est composé de moins de 30 individus (min_samples_split = 30) ; une segmentation est validée si et seulement si les feuilles générées comportent tous au moins 10 observations (min_samples_leaf = 10).

```
#instanciation de l'arbre
from sklearn.tree import DecisionTreeClassifier
arbreFirst = DecisionTreeClassifier(min_samples_split=30,min_samples_leaf=10)
```

Nous lançons le processus de modélisation sur les données d'apprentissage en spécifiant la matrice (X) des variables prédictives, et le vecteur (y) de la variable cible.

```
#construction de l'arbre
arbreFirst.fit(X = dfTrain.iloc[:,:-1], y = dfTrain.classe)
```

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=None, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=10, min_samples_split=30,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

La console affiche l'ensemble des paramètres utilisés lors de la modélisation.

2.5 Affichage graphique de l'arbre

L'affichage de l'arbre était un des obstacles de l'utilisation de cet outil. Auparavant, il fallait générer un fichier au format particulier (**.dot**), que l'on faisait interpréter par un outil externe à installer au préalable. Ce n'était pas vraiment ''user friendly''. Aujourd'hui, depuis la version 0.21 de Scikit-Learn, nous disposons d'une fonction dédiée à la génération de la représentation graphique directement dans la console. Voyons ce qu'il en est.

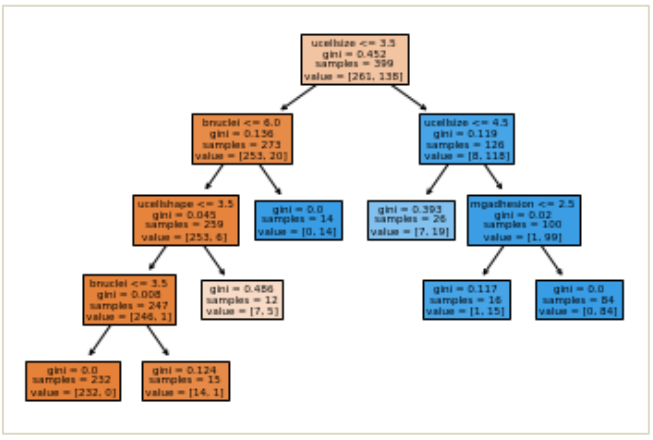
La fonction prend en paramètre l'arbre généré par l'apprentissage, la liste des noms des variables prédictives (feature_names), les sommets peuvent être coloriés selon la classe majoritaire (filled = True).

```
#affichage graphique de l'arbre - depuis sklearn 0.21
#https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html#sklearn.tree.plot_tree
from sklearn.tree import plot_tree
plot_tree(arbreFirst,feature_names = list(df.columns[:-1]),filled=True)
```

L'outil affiche la description textuelle de l'arbre.

```
[Text(182.61818181818182, 195.696, 'ucellsize <= 3.5\ngini = 0.452\nsamples = 399\nvalue = [261, 138]'),
 Text(121.74545454545455, 152.208, 'bnuclei <= 6.0\ngini = 0.136\nsamples = 273\nvalue = [253, 20]'),
 Text(91.30909090909091, 108.72, 'ucellshape <= 3.5\ngini = 0.045\nsamples = 259\nvalue = [253, 6]'),
 Text(60.872727272727275, 65.232, 'bnuclei <= 3.5\ngini = 0.008\nsamples = 247\nvalue = [246, 1]'),
 Text(30.436363636363637, 21.744, 'gini = 0.0\nsamples = 232\nvalue = [232, 0]'),
 Text(91.30909090909091, 21.744, 'gini = 0.124\nsamples = 15\nvalue = [14, 1]'),
 Text(121.74545454545455, 65.232, 'gini = 0.486\nsamples = 12\nvalue = [7, 5]'),
 Text(152.1818181818182, 108.72, 'gini = 0.0\nsamples = 14\nvalue = [0, 14]'),
 Text(243.4909090909091, 152.208, 'ucellsize <= 4.5\ngini = 0.119\nsamples = 126\nvalue = [8, 118]'),
 Text(213.05454545454546, 108.72, 'gini = 0.393\nsamples = 26\nvalue = [7, 19]'),
 Text(273.92727272727274, 108.72, 'mgadhesion <= 2.5\ngini = 0.02\nsamples = 100\nvalue = [1, 99]'),
 Text(243.4909090909091, 65.232, 'gini = 0.117\nsamples = 16\nvalue = [1, 15]'),
 Text(304.3636363636364, 65.232, 'gini = 0.0\nsamples = 84\nvalue = [0, 84]')]
```

Puis sous sa forme graphique.



Si elle n'est pas très lisible, nous pouvons en moduler la taille.

```
#affichage plus grand pour une meilleure lisibilité
import matplotlib.pyplot as plt
plt.figure(figsize=(10,10))
plot_tree(arbreFirst,feature_names = list(df.columns[:-1]),filled=True)
plt.show()
```

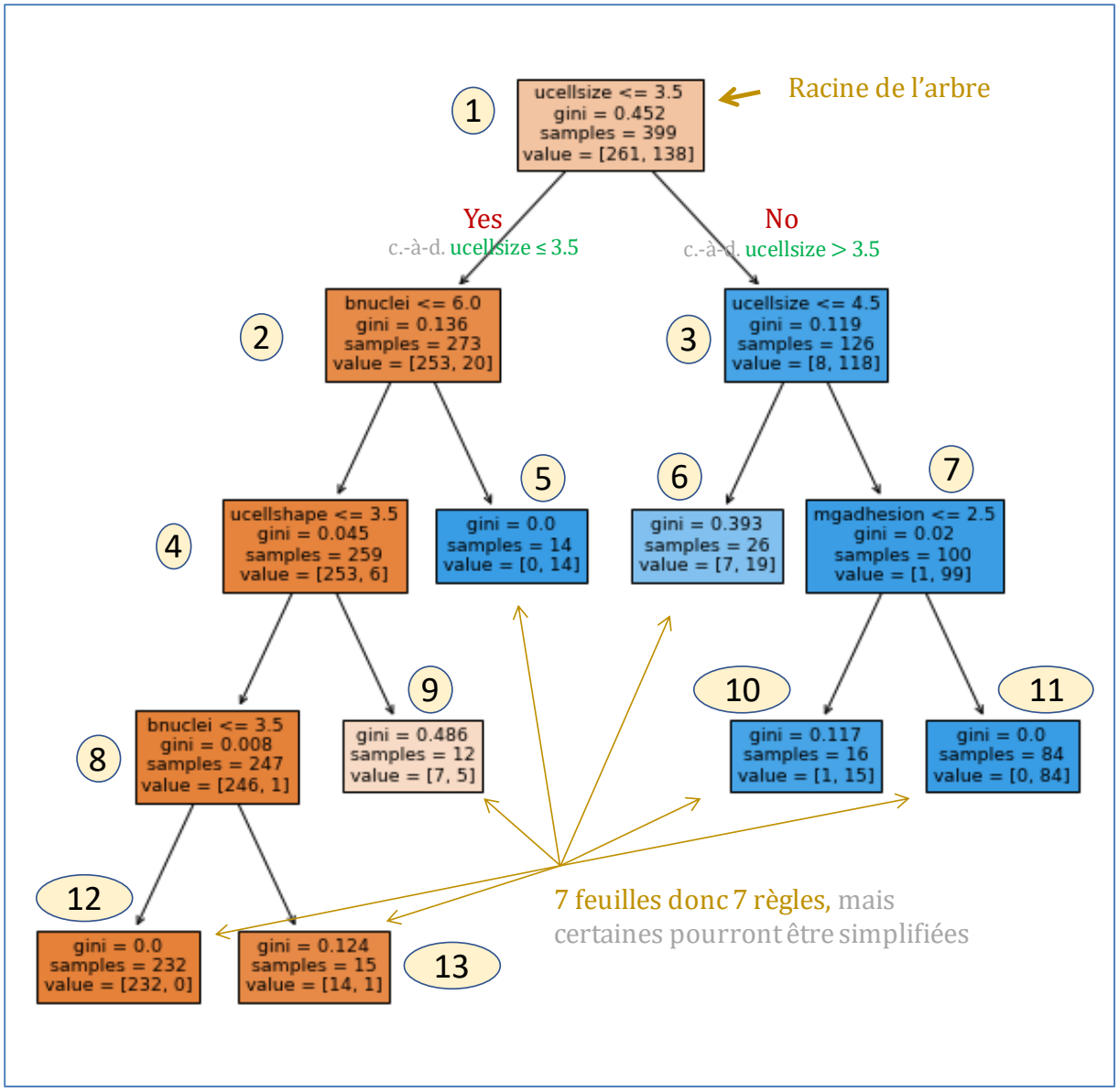


Figure 2 - Premier arbre de décision sur les données "breast"

Que lisons-nous ?

- L'arbre est composé de 7 feuilles (sommets n°12, 13, 9, 5, 6, 10, 11). Il produit donc 7 règles prédictives matérialisées par les chemins partant de la racine aux feuilles.

- Nous observons l'effectif de l'échantillon d'apprentissage sur la racine de l'arbre ($n_{Racine} = n = samples = 399$) avec 261 ''begnin'' et 138 ''malignant'' (dans l'ordre alphabétique).
- Les sommets sont teintés (c'est le rôle de l'option filled = True) selon la classe majoritaire qu'ils portent, avec plus ou moins d'intensité selon la concentration des effectifs. Ici, visiblement, le bleu est dévolu à ''malignant'', l'orange à ''begnin''.
- La concentration des classes est calculée à l'aide de l'indice de Gini (on parle aussi de mesure d'impureté [de l'anglais ''impurity''] ou de mesure de diversité). Pour la racine (sommet n°1), nous avons (K = 2, nombre de modalités de la variable ''classe'') :

$$G(Racine) = \sum_{k=1}^{K} \frac{n_{k,Racine}}{n_{Racine}}\left(1 - \frac{n_{k,Racine}}{n_{Racine}}\right) = \frac{261}{399}\left(1 - \frac{261}{399}\right) + \frac{138}{399}\left(1 - \frac{138}{399}\right) = 0.452$$

- ''ucellsize'' est la variable de segmentation sur la racine, avec la condition ''$ucellsize \leq 3.5$''.
- La branche gauche de la racine (sommet n°2) correspond à la proposition vraie de la condition c.-à-d. $ucellsize \leq 3.5$. Nous lisons sur le sommet enfant qu'elle correspond à (samples = 273) observations, avec 253 ''begnin'' et 20 ''malignant''. Nous avons G(sommet) = 0.136. Plus la valeur de l'indice de Gini est faible, plus les classes sont concentrées sur un sommet.
- La branche droite de la racine (sommet n°3) correspond à la négation de la proposition c.-à-d. ''ucellsize > 3.5'', elle concerne 126 observations avec 8 ''begnin'' et 118 ''malignant''.
- Lorsque les variables de segmentation sont quantitatives, il n'est pas rare qu'elles interviennent plusieurs fois dans l'arbre, mais avec des seuils de découpage différents. C'est le cas de variables ''ucellsize'' et ''bnuclei'' ici.
- Nous pouvons simplifier l'arbre en retirant les feuilles issues du même père qui portent des conclusions identiques. En procédant ainsi de bas en haut (bottom-up), nous effectuons un processus (simplifié) de post-élagage qui permet de réduire la taille de l'arbre sans modifier en aucune manière ses propriétés prédictives. Suivant cette idée, nous devrions aboutir à un arbre avec 3 feuilles dans notre exemple, avec exactement le même comportement en classement. Nous y reviendrons plus bas en modifiant les paramètres d'apprentissage (section 0).

Clairement, la possibilité d'afficher l'arbre sous une forme graphique à l'aide d'instructions simples joue énormément en faveur de l'attractivité de l'outil. L'absence de cette fonctionnalité constituait une des critiques récurrentes des pro-R (rpart / rpart.plot) par rapport au tandem Python / Scikit-Learn. Elle vient d'être balayée.

2.6 Affichage sous forme de règles imbriquées de l'arbre

L'affichage graphique est sympathique mais devient peu lisible dès lors que la taille de l'arbre augmente. Scikit-Learn propose une sortie alternative textuelle, sous la forme de règles imbriquées, à la manière de la surcharge de la fonction print() de rpart sous R.

Le paramètres de la fonction sont identiques à celle de l'affichage graphique, mais il n'est plus question de colorier les sommets bien évidemment. Nous demandons à ce que les effectifs sur les feuilles soient spécifiées (show_weights = True).

```
#affichage sous forme de règles
#plus facile à appréhender quand l'arbre est très grand
from sklearn.tree import export_text
tree_rules = export_text(arbreFirst,feature_names = list(df.columns[:-1]),show_weights=True)
```

```
|--- ucellsize <= 3.50
|   |--- bnuclei <= 6.00
|   |   |--- ucellshape <= 3.50
|   |   |   |--- bnuclei <= 3.50
|   |   |   |   |--- weights: [232.00, 0.00] class: begnin
|   |   |   |--- bnuclei >  3.50
|   |   |   |   |--- weights: [14.00, 1.00] class: begnin
|   |   |--- ucellshape >  3.50
|   |   |   |--- weights: [7.00, 5.00] class: begnin
|   |--- bnuclei >  6.00
|   |   |--- weights: [0.00, 14.00] class: malignant
|--- ucellsize >  3.50
|   |--- ucellsize <= 4.50
|   |   |--- weights: [7.00, 19.00] class: malignant
|   |--- ucellsize >  4.50
|   |   |--- mgadhesion <= 2.50
|   |   |   |--- weights: [1.00, 15.00] class: malignant
|   |   |--- mgadhesion >  2.50
|   |   |   |--- weights: [0.00, 84.00] class: malignant
```

(J'ai colorié les feuilles à la main) En faisant le parallèle avec l'arbre graphique (Figure 2), nous distinguons bien les successions de segmentations. Les effectifs sur les feuilles correspondent bien évidemment.

2.7 Importance des variables

Autre outil d'interprétation, Scikit-Learn peut afficher l'importance des variables, en s'en tenant exclusivement à celles qui apparaissent explicitement dans l'arbre, contrairement à d'autres outils (''rpart'' de R par exemple qui s'appuie sur le mécanisme du ''surrogate variables'').

Nous récupérons le champ ''.feature_importances_'' de l'arbre que nous plaçons dans un data frame Pandas pour pouvoir afficher les contributions des variables associées à leurs noms, et triées de manière décroissante.

```
#importance des variables
impVarFirst={"Variable":df.columns[:-1],"Importance":arbreFirst.feature_importances_}
print(pandas.DataFrame(impVarFirst).sort_values(by="Importance",ascending=False))
```

```
     Variable  Importance
1   ucellsize    0.816627
5     bnuclei    0.158477
2  ucellshape    0.024243
3  mgadhesion    0.000653
0       clump    0.000000
4      sepics    0.000000
6  bchromatin    0.000000
7    normnucl    0.000000
8     mitoses    0.000000
```

Sans surprise, seules les variables qui apparaissent dans l'arbre présentent une valeur non-nulle. ''ucellsize'' est la plus importante, puis viennent ''bnuclei'', ''ucellshape'' et, très marginalement, ''mgadhesion''. Comment sont calculées ces valeurs ?

2.7.1 Qualité globale de l'arbre

La qualité globale de l'arbre peut être quantifiée par la différence entre l'indice de Gini de la racine (n°1), et la moyenne pondérée des Gini des L feuilles (n°12, 13, 9, 5, 6, 10, 11), chacune avec un effectif n_l :

$$\Delta_{Arbre} = G(Racine) - \sum_{l=1}^{L} \frac{n_l}{n} G(l)$$

Dans notre arbre,

$$\Delta_{Arbre} = 0.452 - \left(\frac{232}{399} \times 0.0 + \frac{15}{399} \times 0.124 + \cdots + \frac{16}{399} \times 0.117 + \frac{84}{399} \times 0.0\right) = 0.40284608$$

Cette quantité peut être également exprimée sous la forme d'une somme pondérée des contributions locales de chaque opération de segmentation.

2.7.2 Contribution d'une variable apparaissant une fois

Pour mesurer la contribution d'une segmentation dans l'arbre, nous effectuons la différence entre l'indice de Gini du sommet à segmenter et la moyenne pondérée des Gini de ses feuilles. Cette différence étant pondérée par le poids du sommet traité.

Pour le sommet n°4 où la variable ''ucellshape'' intervient, et qui a généré les sommets n°8 et 9, nous avons :

$$\Delta_{segmentation} = \frac{259}{399} \times \left[0.045 - \left(\frac{247}{259} \times 0.008 + \frac{12}{259} \times 0.486\right)\right] = 0.00976634$$

Elle correspond également à la contribution de la variable dans l'arbre si elle n'apparaît qu'une seule fois.

Cette valeur est ensuite ramenée à la qualité globale de l'arbre pour que la somme des contributions fasse 1. Dans notre cas :

$$CTR(ucellshape) = \frac{0.00976634}{0.40284608} = 0.024243$$

Et c'est bien la valeur associée à ''ucellshape'' dans le tableau de ''feature importances'' proposé par Scikit-Learn.

2.7.3 Contribution d'une variable apparaissant plusieurs fois

Lorsqu'une variable apparaît plusieurs fois, nous additionnons les contributions des segmentations dans lesquelles elle intervient. Pour ''ucellsize'' par exemple, qui opère lors des partitions des sommets n° 1 et 3, nous avons…

$$CTR(ucellsize) = \frac{0.3289749 + 0.00695077}{0.40284608} = 0.816627$$

… comme nous l'indique le tableau fourni par Scikit-Learn.

2.8 Evaluation en test

2.8.1 Prédiction en test

Pour évaluer les performances prédictives de l'arbre, nous l'appliquons sur l'échantillon test composé de 300 observations. Nous obtenons une prédiction :

```
#prédiction sur l'échantillon test
predFirst = arbreFirst.predict(X=dfTest.iloc[:,:-1])

#distribution des predictions
import numpy
print(numpy.unique(predFirst,return_counts=True))
```

```
(array(['begnin', 'malignant'], dtype=object), array([199, 101], dtype=int64))
```

La classe ''begnin'' a été assignée à 199 observations, 101 pour ''malignant''.

Mais est-ce à juste titre ? Pour le savoir, nous confrontons les classes observées et prédites via la matrice de confusion.

2.8.2 Matrice de confusion

Nous aurions pu utiliser un outil tableau croisé standard, type ''crosstab'' de la librairie Pandas, mais ''Scikit-Learn'' propose un module dédié ''metrics'' qui se charge de toute la partie évaluation des classifieurs. Nous aurions tort de nous en priver.

Nous calculons la matrice de confusion :

```
#matrice de confusion
from sklearn import metrics
print(metrics.confusion_matrix(dfTest.classe,predFirst))
```

```
[[189   8]
 [ 10  93]]
```

Les lignes et colonnes sont dans l'ordre alphabétique des modalités. De fait, avec les étiquettes, notre tableau se présenterait comme suit :

		classes prédictes	
		begnin	malignant
classes observées	begnin	189	8
	malignant	10	93

2.8.3 Indicateurs de performances

Nous pouvons en déduire les indicateurs de performances usuels. Le taux de reconnaissance :

```
#taux de reconnaissance - (189+93)/300
print(metrics.accuracy_score(dfTest.classe,predFirst))
```

```
0.94
```

A partir duquel nous pouvons dériver le taux d'erreur :

```
#taux d'erreur - (10+8)/300
print(1.0 - metrics.accuracy_score(dfTest.classe,predFirst))
```

```
0.06
```

Et si ''classe = mailgnant'' est la modalité cible que l'on cherche à identifier en priorité (pos_label = 'malignant'), nous pouvons calculer.

Le rappel ou sensibilité :

```
#rappel - sensibilité - 93/(10+93)
print(metrics.recall_score(dfTest.classe,predFirst,pos_label='malignant'))
```

```
0.9029126213592233
```

La précision :

```
#précision - 93/(8+93)
print(metrics.precision_score(dfTest.classe,predFirst,pos_label='malignant'))
```

```
0.9207920792079208
```

Et le F1-Score qui est une moyenne harmonique entre rappel et précision :

```
#F1-score
print(metrics.f1_score(dfTest.classe,predFirst,pos_label='malignant'))
```

```
0.911764705882353
```

Rapport de prédiction. Scikit-Learn propose un rapport global intégrant ces différents éléments avec la fonction classification_report() :

```
#rapport de prédiction
print(metrics.classification_report(dfTest.classe,predFirst))
```

```
              precision    recall  f1-score   support

      begnin       0.95      0.96      0.95       197
   malignant       0.92      0.90      0.91       103

    accuracy                           0.94       300
   macro avg       0.94      0.93      0.93       300
weighted avg       0.94      0.94      0.94       300
```

2.9 Modification des paramètres d'apprentissage

Notre arbre (Figure 2) paraît surdimensionné. Nous avions remarqué notamment que plusieurs feuilles issues du même sommet père portaient des conclusions identiques.

Dans cette section, nous introduisons un nouveau paramètre pour réduire la taille de l'arbre. Nous spécifions (max_leaf_nodes = 3) c.-à-d. nous souhaitons obtenir un arbre qui produit 3 règles au maximum. Dixit la documentation, l'outil effectue en priorité les segmentations qui maximisent les contributions.

```
#modifier les paramètres d'apprentissage
arbreSecond = DecisionTreeClassifier(min_samples_split=30,min_samples_leaf=10,max_leaf_nodes=3)

#construction de l'arbre
arbreSecond.fit(X = dfTrain.iloc[:,:-1], y = dfTrain.classe)

#affichage graphique de l'arbre
plot_tree(arbreSecond,feature_names = list(df.columns[:-1]),filled=True)
```

Nous obtenons une nouvelle version de l'arbre de décision (Figure 3).

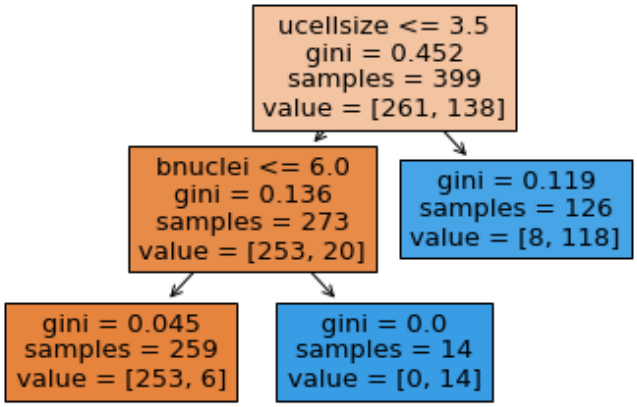


Figure 3 - Second arbre de décision sur les données "breast"

L'arbre est fortement simplifié, tout en maintenant ses qualités prédictives…

```
#prédiction sur l'échantillon test
predSecond = arbreSecond.predict(X=dfTest.iloc[:,:-1])

#matrice de confusion
print(metrics.confusion_matrix(dfTest.classe,predSecond))
```

```
[[189   8]
 [ 10  93]]
```

… puisque nous obtenons exactement la même matrice de confusion sur l'échantillon test, et par conséquent des valeurs identiques des indicateurs de performances.

```
#taux de reconnaissance
print(metrics.accuracy_score(dfTest.classe,predSecond))
```

```
0.94
```

3 Cas des prédictives exclusivement qualitatives

Un second écueil me chagrine lors de l'utilisation des arbres de décision de Scikit-Learn : l'impossibilité d'introduire des variables prédictives qualitatives dans le modèle. Je le contournais facilement en choisissant opportunément les bases à utiliser en TD. Mais cette solution – ce n'en est pas vraiment une d'ailleurs – n'est pas satisfaisante. Je profite de ce tutoriel pour montrer comment surmonter cet obstacle par un codage judicieux des variables.

3.1 Importation des données

Nous traitons une version simplifiée de la base ''Congressional Voting Records'' maintenant. Elle recense les votes de parlementaires américains {(y)es, (n)o, (_?) on ne sait ce qu'ils ont voté} qui sont repartis en 2 groupes politiques {democrat, republican} que l'on cherche à identifier.

Nous chargeons et inspectons les données.

```
#vérification de la version de scikit-learn
import sklearn
print(sklearn.__version__) #0.22.1

#modifier le working directory
import os
os.chdir("... votre dossier de travail ...")

#importer les données
import pandas
dfVote = pandas.read_excel("vote_simplified.xlsx",sheet_name = 0)

#dimensions
print(dfVote.shape) #(435, 7)

#liste et type des variables
print(dfVote.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 435 entries, 0 to 434
Data columns (total 7 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   adoption_of_the_budget_re  435 non-null    object
 1   physician_fee_freeze       435 non-null    object
 2   mx_missile                 435 non-null    object
 3   superfund_right_to_sue     435 non-null    object
 4   crime                      435 non-null    object
 5   duty_free_exports          435 non-null    object
 6   groupe                     435 non-null    object
```

''groupe'' est la variable cible. Les autres sont les thèmes sur lesquels les parlementaires se sont prononcés. Pandas attribue le type ''object'' pour ces variables dont les valeurs sont représentées par des chaînes de caractères comme nous le constatons en affichant les premières lignes. Pandas propose l'équivalent du ''factor'' de R avec le type ''category'', mais les variables n'ont pas été encodées en ce sens lors de notre importation avec les options par défaut.

```
#affichage des premières lignes
print(dfVote.head())
```

```
  adoption_of_the_budget_re physician_fee_freeze  ... duty_free_exports      groupe
0                         n                    y  ...                n  republican
1                         n                    y  ...                n  republican
2                         y                   _?  ...                n    democrat
3                         y                    n  ...                n    democrat
4                         y                    n  ...                y    democrat
```

3.2 Construction de l'arbre sans recodage

A priori, les algorithmes d'arbres de décision n'ont aucun problème à manipuler les variables prédictives qualitatives. C'est le cas de la très grande majorité des logiciels. Confiants, nous instancions un arbre de décision avec une profondeur maximale de 3 (max_depth = 3) pour faciliter sa lecture, l'essentiel étant ailleurs dans cette section.

```
#instanciation de l'arbre
from sklearn.tree import DecisionTreeClassifier
arbreVote = DecisionTreeClassifier(max_depth = 3)

#construction de l'arbre
arbreVote.fit(X = dfVote.iloc[:,:-1], y = dfVote.groupe)
```

```
Traceback (most recent call last):

  File "<ipython-input-3-770dae6c0d59>", line 5, in <module>
    arbreVote.fit(X = dfVote.iloc[:,:-1], y = dfVote.groupe)

  File "D:\Logiciels\Anaconda3\lib\site-packages\sklearn\tree\_classes.py", line 877, in fit
    X_idx_sorted=X_idx_sorted)

  File "D:\Logiciels\Anaconda3\lib\site-packages\sklearn\tree\_classes.py", line 149, in fit
    X = check_array(X, dtype=DTYPE, accept_sparse="csc")

  File "D:\Logiciels\Anaconda3\lib\site-packages\sklearn\utils\validation.py", line 531, in check_array
    array = np.asarray(array, order=order, dtype=dtype)

  File "D:\Logiciels\Anaconda3\lib\site-packages\numpy\core\_asarray.py", line 85, in asarray
    return array(a, dtype, copy=False, order=order)

ValueError: could not convert string to float: 'n'
```

Patatras ! L'interpréteur nous annonce qu'il ne sait pas convertir les valeurs proposées en flottant. En effectuant quelques recherches sur le web, je me suis rendu compte que l'outil ne sait pas appréhender les explicatives non-numériques à ce jour (version 0.22.1, février 2020). On est mal. Il faut passer par un recodage. Mais de quel type ?

3.3 Codage disjonctif complet des prédictives

Codage 0/1 des prédictives. Je propose de passer par un codage disjonctif complet (''Codage disjonctif complet'', mars 2008) c.-à-d. un codage 0/1 où toutes les modalités sont représentées (ce qui n'est pas le cas dans le tutoriel sur la régression logistique que j'ai mis en lien, une des modalités sert de référence). Nous énumérons tout d'abord les colonnes constituant la base pour mieux situer les opérations qui viendront.

```
#liste des variables
print(dfVote.columns)
```

```
Index(['adoption_of_the_budget_re', 'physician_fee_freeze', 'mx_missile',
       'superfund_right_to_sue', 'crime', 'duty_free_exports', 'groupe'],
      dtype='object')
```

Nous devons recoder toutes les variables, sauf la cible ''groupe'' qui est en dernière position. Pour recoder les variables, nous utilisons la fonction get_dummies() de la librairie Pandas.

```
#encodage des prédictives - toutes les colonnes sauf la dernière
dfVoteBis = pandas.get_dummies(dfVote[dfVote.columns[:-1]])
print(dfVoteBis.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 435 entries, 0 to 434
Data columns (total 18 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   adoption_of_the_budget_re__?  435 non-null    uint8
 1   adoption_of_the_budget_re_n   435 non-null    uint8
 2   adoption_of_the_budget_re_y   435 non-null    uint8
 3   physician_fee_freeze__?       435 non-null    uint8
 4   physician_fee_freeze_n        435 non-null    uint8
 5   physician_fee_freeze_y        435 non-null    uint8
 6   mx_missile__?                 435 non-null    uint8
 7   mx_missile_n                  435 non-null    uint8
 8   mx_missile_y                  435 non-null    uint8
 9   superfund_right_to_sue__?     435 non-null    uint8
 10  superfund_right_to_sue_n      435 non-null    uint8
 11  superfund_right_to_sue_y      435 non-null    uint8
 12  crime__?                      435 non-null    uint8
 13  crime_n                       435 non-null    uint8
 14  crime_y                       435 non-null    uint8
 15  duty_free_exports__?          435 non-null    uint8
 16  duty_free_exports_n           435 non-null    uint8
 17  duty_free_exports_y           435 non-null    uint8
dtypes: uint8(18)
```

Toutes les variables maintenant sont des indicatrices 0/1 codées en entier. Nous remarquons également que, pour chaque variable, toutes les modalités sont représentées (y, n, _?).

Vérification pour la variable ''crime''. Pour vérifier l'opération, nous nous intéressons à la variable ''crime''. Nous affichons la fréquence relative de ses modalités.

```
#répartition de la variable crime par ex.
print(dfVote['crime'].value_counts(normalize=True))
```

```
y     0.570115
n     0.390805
_?    0.039080
Name: crime, dtype: float64
```

Nous devrions obtenir exactement les mêmes résultats – les mêmes proportions – en calculant les moyennes des colonnes constituées de valeurs 0/1.

```
#moyennes par variables -- proportions
print(dfVoteBis.apply(func='mean',axis=0))
```

```
adoption_of_the_budget_re__?    0.025287
adoption_of_the_budget_re_n     0.393103
adoption_of_the_budget_re_y     0.581609
physician_fee_freeze__?         0.025287
physician_fee_freeze_n          0.567816
physician_fee_freeze_y          0.406897
mx_missile__?                   0.050575
mx_missile_n                    0.473563
mx_missile_y                    0.475862
superfund_right_to_sue__?       0.057471
superfund_right_to_sue_n        0.462069
superfund_right_to_sue_y        0.480460
crime__?                        0.039080
crime_n                         0.390805
crime_y                         0.570115
duty_free_exports__?            0.064368
duty_free_exports_n             0.535632
duty_free_exports_y             0.400000
dtype: float64
```

Et, effectivement, nous avons les bonnes valeurs pour la variable ''crime''. Il n'y a pas de pertes d'informations durant le recodage.

Compléter la base avec la variable cible. Nous complétons la base avec la variable cible ''groupe'', qu'il n'était pas nécessaire de recoder, mais qui fait partie de l'analyse.

```
#ajouter la variable 'groupe'
dfVoteBis['groupe'] = dfVote.groupe
print(dfVoteBis.info())
```

```
RangeIndex: 435 entries, 0 to 434
Data columns (total 19 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   adoption_of_the_budget_re__?  435 non-null    uint8
 1   adoption_of_the_budget_re_n   435 non-null    uint8
 2   adoption_of_the_budget_re_y   435 non-null    uint8
 3   physician_fee_freeze__?       435 non-null    uint8
 4   physician_fee_freeze_n        435 non-null    uint8
 5   physician_fee_freeze_y        435 non-null    uint8
 6   mx_missile__?                 435 non-null    uint8
 7   mx_missile_n                  435 non-null    uint8
 8   mx_missile_y                  435 non-null    uint8
 9   superfund_right_to_sue__?     435 non-null    uint8
 10  superfund_right_to_sue_n      435 non-null    uint8
 11  superfund_right_to_sue_y      435 non-null    uint8
 12  crime__?                      435 non-null    uint8
 13  crime_n                       435 non-null    uint8
 14  crime_y                       435 non-null    uint8
 15  duty_free_exports__?          435 non-null    uint8
 16  duty_free_exports_n           435 non-null    uint8
```

```
 17  duty_free_exports_y           435 non-null    uint8
 18  groupe                        435 non-null    object
dtypes: object(1), uint8(18)
```

Des 7 variables initiales, nous travaillons maintenant avec une base comportant 18 variables.

Remarque sur le schéma apprentissage-test. J'ai toujours pour habitude de dire aux étudiants qu'il faut calculer tous les paramètres de recodage sur l'échantillon d'apprentissage avant de l'appliquer sur l'échantillon test. Ce principe ne s'applique pas ici puisque nous n'effectuons aucun calcul à partir des données pour procéder au recodage. Dans le cas présent, qui est particulier, il est donc possible de réaliser ce pré-traitement sur la totalité de la base avant de partitionner le dataset en échantillons d'apprentissage et de test.

3.4 Construction de l'arbre de décision après recodage

Nous créons une nouvelle instance de l'arbre de décision et nous lançons la modélisation sur la totalité des données. Il s'agit avant tout d'un exercice de style destiné à montrer l'intérêt du codage disjonctif complet ici, le schéma apprentissage-test n'a pas lieu d'être.

```
#réinstancier l'arbre de décision
arbreVote = DecisionTreeClassifier(max_depth = 3)
#construction de l'arbre sur les données
arbreVote.fit(X = dfVoteBis.iloc[:,:-1], y = dfVoteBis.groupe)
#affichage graphique de l'arbre - depuis sklearn 0.21
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
plt.figure(figsize=(20,7.5))
plot_tree(arbreVote,feature_names = list(dfVoteBis.columns[:-1]),filled=True)
plt.show()
```

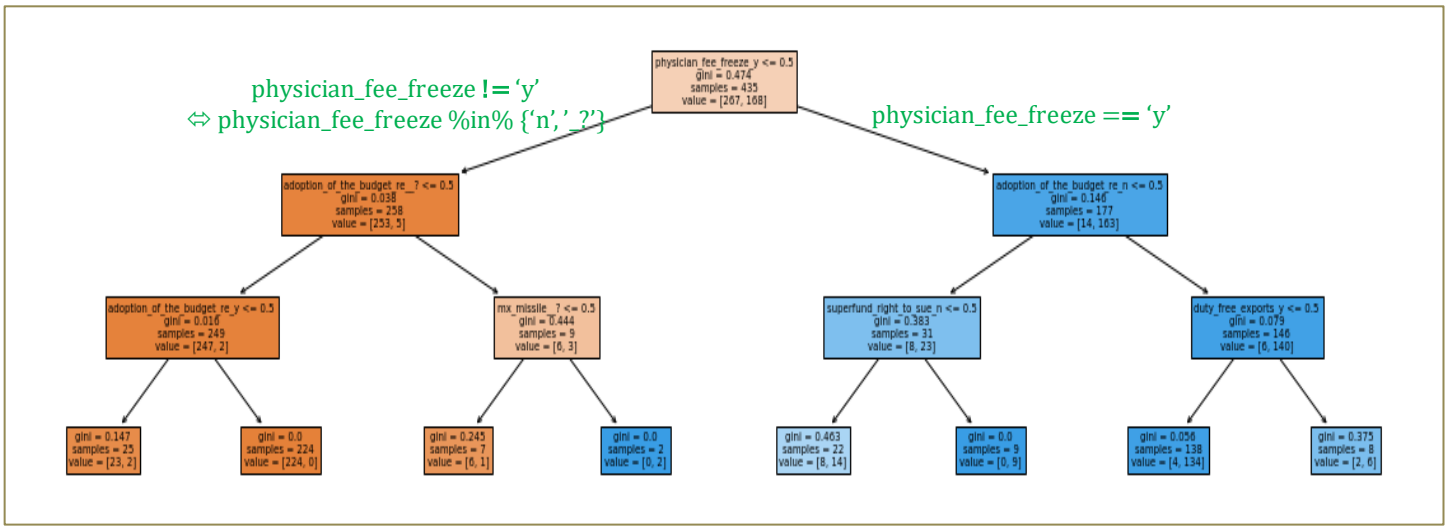


Figure 4 - Arbre de décision sur la base "vote"

La lecture de l'arbre nécessite un petit exercice d'interprétation. Prenons l'exemple de la racine de l'arbre :

- Elle est segmentée avec la condition (physician_fee_freeze_y ≤ 0.5). C'est variable 0/1 encodée avec la proposition (physician_fee_freeze = 'y') qui a été utilisée.
- La branche gauche est activée lorsque la condition est vérifiée c.-à-d. lorsque (physician_fee_freeze_yes == 0) puisqu'elle est binaire. Si l'on revient à la variable initiale, cette condition correspond donc à (physician_fee_freeze **!=** 'y').
- A contrario, la branche droite est activée lorsque (physician_fee_freeze_y > 0.5) c.-à.-d. (physician_fee_freeze_y == 1), ou encore (physician_fee_freeze == 'y').

Sous la forme d'un tableau croisé sur la variable originelle, voici le fruit de la segmentation de la racine :

		physician_fee_freeze	
		n, _?	y
Group	democrat	253	14
	republican	5	163

Le reste de l'arbre se lit de la même manière.

Remarque : Notons que cette transformation n'est pas anodine. L'algorithme explore différemment l'espace des solutions. Surtout lorsque leur nombre est élevé, plutôt qu'un regroupement binaire des modalités, nous avons – conséquence du recodage – une stratégie ''une modalité contre les autres''. Ce n'est pas mieux ou moins bien, c'est tout simplement différent, et elle (cette stratégie) a un impact sur les résultats.

4 Cas des prédictives mixtes

Comment faire lorsque la base est composée d'un mix de variables prédictives qualitative et quantitatives ? Il faut repérer les variables non-numériques et les transformer, puis mettre le tout, variables codées 0/1 et celles initialement numériques, dans un seul data frame.

4.1 Importation des données

Nous utilisons la base ''Statlog Heart Dataset'' dans cette section.

```
#importer les données
import pandas
```

```
dfHeart = pandas.read_excel("heart_dataset.xlsx",sheet_name = 0)

#dimensions
print(dfHeart.shape)

#liste des variables
print(dfHeart.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 270 entries, 0 to 269
Data columns (total 13 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   age            270 non-null    int64
 1   sexe           270 non-null    object
 2   type_douleur   270 non-null    object
 3   pression       270 non-null    int64
 4   cholester      270 non-null    int64
 5   sucre          270 non-null    object
 6   electro        270 non-null    object
 7   taux_max       270 non-null    int64
 8   angine         270 non-null    object
 9   depression     270 non-null    int64
 10  pic            270 non-null    int64
 11  vaisseau       270 non-null    object
 12  coeur          270 non-null    object
dtypes: int64(6), object(7)
```

Outre la variable cible ''heart'', forcément qualitative, nous disposons de 6 variables prédictives qualitatives (object : sexe, type_douleur, sucre, electro, angine, vaisseau) et 6 quantitatives (numériques : age, pression, cholester, taux_max, depression, pic).

L'idée serait donc de repérer automatiquement les prédictives qualitatives, de les coder en 0/1, puis de les intégrer dans un nouveau data frame en compagnie des quantitatives.

4.2 Codage disjonctif des prédictives qualitatives

Pour identifier les qualitatives, nous parcourons les variables prédictives candidates et nous comparons leur type avec la constante ''object_'' de la librairie Numpy (https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.dtypes.html).

```
#liste des variables quanti
import numpy
lstQuali =  [var for var in dfHeart.columns[:-1] if dfHeart[var].dtype == numpy.object_]
print(lstQuali)
```

```
['sexe', 'type_douleur', 'sucre', 'electro', 'angine', 'vaisseau']
```

Nous leur appliquons un codage disjonctif complet pour obtenir un nouveau data frame.

```
#recoder en 0/1 ces variables
dfQualiEncoded = pandas.get_dummies(dfHeart[lstQuali])
```

```
print(dfQualiEncoded.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 270 entries, 0 to 269
Data columns (total 17 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   sexe_feminin    270 non-null    uint8
 1   sexe_masculin   270 non-null    uint8
 2   type_douleur_A  270 non-null    uint8
 3   type_douleur_B  270 non-null    uint8
 4   type_douleur_C  270 non-null    uint8
 5   type_douleur_D  270 non-null    uint8
 6   sucre_A         270 non-null    uint8
 7   sucre_B         270 non-null    uint8
 8   electro_A       270 non-null    uint8
 9   electro_B       270 non-null    uint8
 10  electro_C       270 non-null    uint8
 11  angine_non      270 non-null    uint8
 12  angine_oui      270 non-null    uint8
 13  vaisseau_A      270 non-null    uint8
 14  vaisseau_B      270 non-null    uint8
 15  vaisseau_C      270 non-null    uint8
 16  vaisseau_D      270 non-null    uint8
dtypes: uint8(17)
```

4.3 Constitution du data.frame de travail

Nous construisons par ailleurs la liste des variables numériques (différent de 'object_').

```
#liste des variables quantitatives
lstQuanti = [var for var in dfHeart.columns[:-1] if dfHeart[var].dtype != numpy.object_]
print(lstQuanti)
```

```
['age', 'pression', 'cholester', 'taux_max', 'depression', 'pic']
```

Nous concaténons le data frame correspond à celui composé des variables 0/1.

```
#réunir dummies et quantitatives dans le même data frame
dfNew = pandas.concat([dfQualiEncoded,dfHeart[lstQuanti]],axis=1)
print(dfNew.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 270 entries, 0 to 269
Data columns (total 23 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   sexe_feminin    270 non-null    uint8
 1   sexe_masculin   270 non-null    uint8
 2   type_douleur_A  270 non-null    uint8
 3   type_douleur_B  270 non-null    uint8
 4   type_douleur_C  270 non-null    uint8
 5   type_douleur_D  270 non-null    uint8
 6   sucre_A         270 non-null    uint8
 7   sucre_B         270 non-null    uint8
 8   electro_A       270 non-null    uint8
 9   electro_B       270 non-null    uint8
 10  electro_C       270 non-null    uint8
 11  angine_non      270 non-null    uint8
```

```
 12  angine_oui      270 non-null    uint8
 13  vaisseau_A      270 non-null    uint8
 14  vaisseau_B      270 non-null    uint8
 15  vaisseau_C      270 non-null    uint8
 16  vaisseau_D      270 non-null    uint8
 17  age             270 non-null    int64
 18  pression        270 non-null    int64
 19  cholester       270 non-null    int64
 20  taux_max        270 non-null    int64
 21  depression      270 non-null    int64
 22  pic             270 non-null    int64
dtypes: int64(6), uint8(17)
```

Et nous n'oublions pas bien sûr de lui associer la variable cible ''cœur''.

```
#rajouter la variable cible
dfNew['coeur'] = dfHeart.coeur
print(dfNew.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 270 entries, 0 to 269
Data columns (total 24 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   sexe_feminin    270 non-null    uint8
 1   sexe_masculin   270 non-null    uint8
 2   type_douleur_A  270 non-null    uint8
 3   type_douleur_B  270 non-null    uint8
 4   type_douleur_C  270 non-null    uint8
 5   type_douleur_D  270 non-null    uint8
 6   sucre_A         270 non-null    uint8
 7   sucre_B         270 non-null    uint8
 8   electro_A       270 non-null    uint8
 9   electro_B       270 non-null    uint8
 10  electro_C       270 non-null    uint8
 11  angine_non      270 non-null    uint8
 12  angine_oui      270 non-null    uint8
 13  vaisseau_A      270 non-null    uint8
 14  vaisseau_B      270 non-null    uint8
 15  vaisseau_C      270 non-null    uint8
 16  vaisseau_D      270 non-null    uint8
 17  age             270 non-null    int64
 18  pression        270 non-null    int64
 19  cholester       270 non-null    int64
 20  taux_max        270 non-null    int64
 21  depression      270 non-null    int64
 22  pic             270 non-null    int64
 23  coeur           270 non-null    object
dtypes: int64(6), object(1), uint8(17)
```

4.4 Construction de l'arbre

Nous pouvons dès lors élaborer l'arbre de décision. Nous limitons sa profondeur à (max_depth = 2) pour en faciliter la lecture.

```
#instanciation de l'arbre
from sklearn.tree import DecisionTreeClassifier
arbreHeart = DecisionTreeClassifier(max_depth = 2)
```

```
#construction de l'arbre
arbreHeart.fit(X = dfNew.iloc[:,:-1], y = dfNew.coeur)

#affichage graphique de l'arbre - depuis sklearn 0.21
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
plt.figure(figsize=(15,10))
plot_tree(arbreHeart,feature_names = list(dfNew.columns[:-1]),filled=True)
plt.show()
```

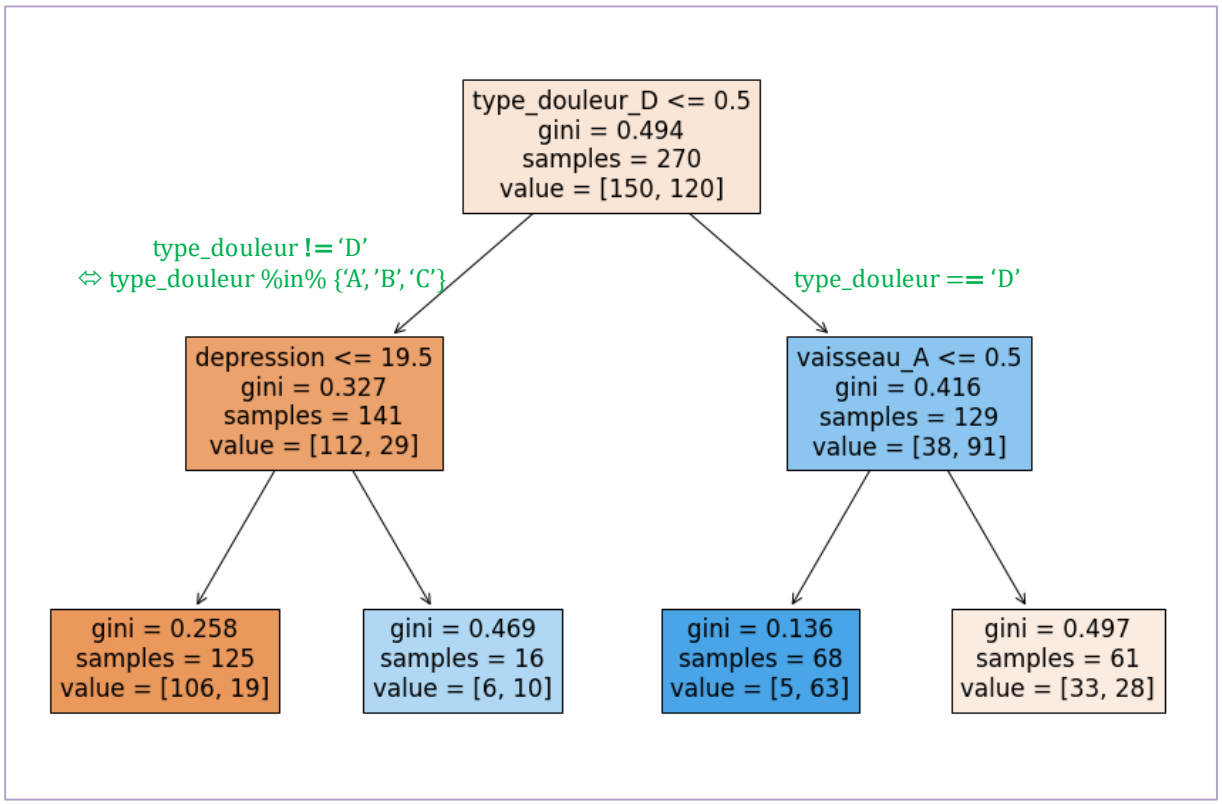


Figure 5 - Arbre de décision sur la base "heart"

La segmentation de la racine de l'arbre porte sur la variable recodée ''type_douleur_D'' c.-à-d. la présence ou non de la modalité 'D' pour la variable ''type_douleur''. Avec :

- Sur la branche gauche les observations répondant à la condition (type_douleur_D ≤ 0.5) soit (type_douleur != 'D') ou encore (type_douleur ∈ {'A', 'B', 'C'}).
- La branche droite correspond à la condition (type_douleur_D > 0.5) soit (type_douleur = 'D').

Nous observons que les descripteurs quantitatifs et qualitatifs (sous leur forme recodée) apparaissent maintenant de manière indifférenciée dans l'arbre.

Remarque : Comme pour les bases à variables prédictives exclusivement qualitatives, nous pouvons effectuer l'opération de transformation avant de procéder à la partition des données en échantillons d'apprentissage et de test.

5 Conclusion

Les packages évoluent et c'est tant mieux. Il faut suivre simplement. Dans ce tutoriel, nous avons exploré une nouvelle fonctionnalité de Scikit-Learn pour Python, la possibilité de représenter graphiquement un arbre de décision avec des commandes succinctes, sans utilisation d'outils additionnels. Nous en avons profité pour montrer comment appréhender les variables explicatives catégorielles qui ne sont pas, à l'heure actuelle (février 2020), supportées nativement par la classe de calcul.

6 Références

Scikit-Learn, ''Decision Trees'', https://scikit-learn.org/stable/modules/tree.html

R. Rakotomalala, ''Arbres de décision'', Revue Modulad, numéro 33, pp. 163-187, 2005.

Tutoriel Tanagra, ''Introduction à R – Arbre de décision'', mars 2012.