Bermudes

Zeste de savoir

Les arbres de décisions

12 août 2019

Table des matières

Ce cours a pour objectif de vous apprendre (ou de vous rappeler si vous connaissez déjà) comment générer un arbre de décision avec les algorithmes ID3 et C4.5 inventés par Ross Quinlan dans les années **1980** et **1990**. Je vous montrerai le principe de ces algorithmes et leur utilité à l'aide d'un exemple (celui utilisé par Quinlan lui-même), et je vous montrerai également les pseudo-codes pour pouvoir vous laisser la possibilité de les implémenter, ce que je vous accompagnerai à faire en Python3.

Pour comprendre les points de théorie traités, il est nécessaire de savoir manipuler l'indice sommatoire ($\sum$), les logarithmes (et exponentielles). Il faut également être à l'aise avec la notion d'ensembles et de sous-ensembles et bien sûr la notion d'arbre informatique (commencer par les arbres binaires, puis les arbres n-aires puis les arbres quelconques pour ceux ne connaissant pas cette notion).

Si vous arrivez à la fin, vous aurez un moyen infaillible de gagner au *Qui est-ce ?* , c'est moi qui vous le dis ! Mais il vous faut bien entendu arriver jusqu'à la conclusion Vous verrez que le *Qui est-ce ?* est un exemple qui s'applique très bien à ce que nous allons accomplir.

1. Comprendre le concept

Dans cette partie, nous allons voir la théorie résidant derrière le machine learning et plus précisément les arbres de décisions. Restez bien attentifs et n'hésitez pas à relire les passages qui vous semblent compliqués car il est évidemment important de bien saisir ce qui est expliqué ici pour pouvoir comprendre la suite.

1.1. Les origines

Je vais commencer par introduire tous les termes un peu compliqués et les concepts qui vont revenir tout au long de ce tutoriel.

###1. Mise en situation

Mettons-nous tout alors en situation, et laissez-moi vous exposer en quoi ce que je vous propose est intéressant. Figurez-vous que j'ai un ami qui s'appelle René et qui me demande s'il peut s'inscrire au MIT. Alors comme vous pouvez vous en douter, ce n'est pas moi qui décide de qui peut ou ne peut pas s'inscrire dans cet établissement réputé. Je lui ai donc dit de se référer aux responsables des inscriptions. Jusque là, vous vous demandez pourquoi je vous parle de notre cher René et vous avez tout à fait raison. Mais une petite minute, j'y arrive.

Le MIT avait des problèmes financiers parce qu'il nécessitait des dizaines de personnes préposées aux décisions d'inscriptions car ces derniers étaient extrêmement sollicités et qu'il fallait les payer, mais c'est maintenant une période révolue : tout a été informatisé ! Bien entendu, il faut trouver une manière de savoir comment expliquer à l'ordinateur comment choisir... À nouveau, ça ne va pas vous surprendre, l'ordinateur va l'apprendre tout seul ! Quoi ? Si, ça vous surprend ? Tiens donc... L'ordinateur va se créer une structure interne pour réussir à prendre des décisions. C'est justement ce que nous allons faire nous aussi (quelle coïncidence !).

Il y a bien entendu plusieurs manières pour faire un arbre de décision : on peut gentiment expliquer à l'ordinateur en le programmant étape par étape comment il doit faire pour prendre une décision (c'est la méthode facile) ou alors on peut lui demander de trouver tout seul comment le générer (c'est la méthode compliquée). Je vous laisse deviner laquelle nous allons développer ici.

Ne perdez pas de vue notre ami René, je vais en reparler dans un instant pour vous expliquer le fonctionnement plus précis d'un arbre de décision. Mais avant ça, je vais devoir vous parler de l'*apprentissage automatique* !

###2. Apprentissage automatique Commençons par l'apprentissage automatique, plus connu sous son nom anglais : *Machine Learning*. C'est une branche de l'IA qui consiste à trouver des méthodes qui permettraient la résolution de problèmes compliqués à résoudre avec des algorithmes triviaux. Bon qu'est-ce qu'un algorithme trivial ? Il faudrait déjà commencer par

définir ce qu'est un algorithme. Si ça vous intéresse vraiment fort, n'hésitez pas à aller lire le tutoriel sur ce site (ici).

? Bon, qu'est-ce qu'un algorithme ?

Un algorithme, c'est une suite finie d'instructions non ambigües exprimant la résolution d'un problème. Voilà, vous êtes contents ? Qu'est-ce que ça veut dire tout ça ? Bon, allons-y étape par étape. Une suite finie d'instructions est une liste d'instructions qui est très clairement définie et pour laquelle le nombre d'étapes (instructions) est fini (fini ici est le contraire d'infini). Passons à la caractéristique de non-ambigüité. Ça veut tout simplement dire que chaque instruction de la liste est très claire sur ce qu'elle fait et ne peut pas être confondue avec une autre.

? Et trivial, ça veut dire quoi ?

On va considérer ici qu'un algorithme trivial c'est un algorithme (forcément) qui est très simple et surtout qui s'exécute en un temps plus qu'acceptable. On parle de complexité polynomiale pour les intéressés.

> De manière générale, l'apprentissage automatique, c'est apprendre à faire mieux dans le futur sur base de ce qui a été expérimenté dans le passé.
>
> *source*

Les chercheurs tentent donc de découvrir l'algorithme (ou plus justement la succession d'algorithmes) le plus efficace pour permettre au programme d'apprendre de ce qui lui est donné. Cet algorithme se veut le plus général possible :

> Nous cherchons des algorithmes qui peuvent être facilement applicables à une large classe de problèmes d'apprentissage.
>
> *source*

Le fondement même de cette branche de l'IA est de trouver une méthode de résolution de problèmes sans que des programmeurs ne doivent adapter l'algorithme en fonction du problème posé. Vous pouvez voir sur la figure suivante le fonctionnement général de l'apprentissage automatique. Les notions et le vocabulaire qui y sont relatifs sont expliqués juste après.

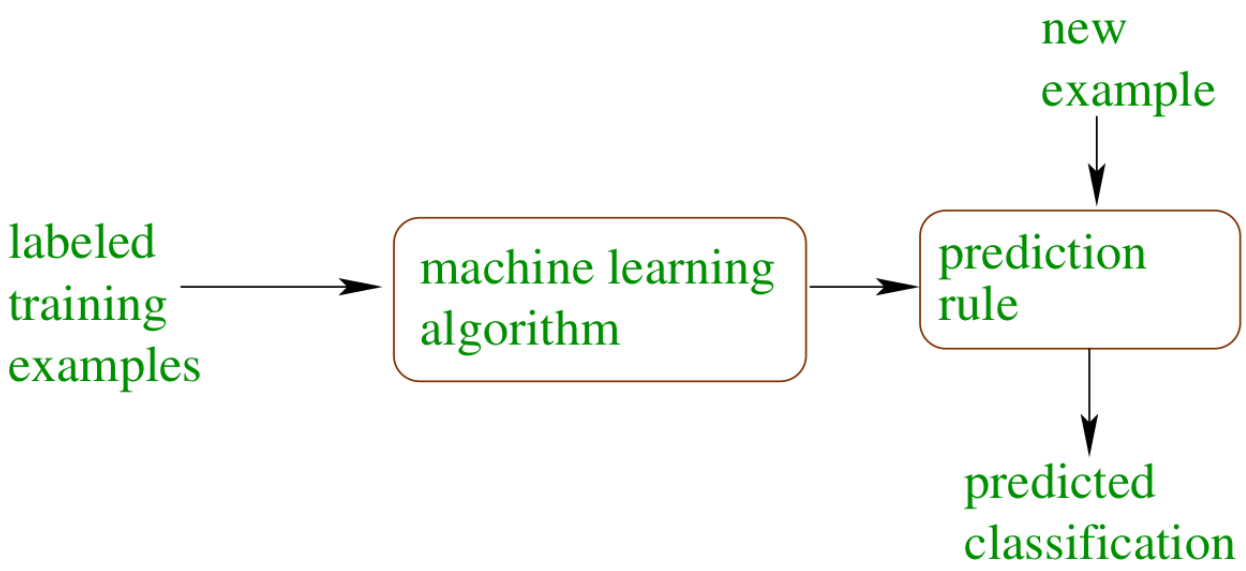

FIGURE 1.1. – fonctionnement d'un problème typique d'apprentissage automatique (Fig. 1)

L'apprentissage automatique a un jargon bien particulier que l'on va détailler ici.

> Attention, le vocabulaire expliqué ici est un vocabulaire relatif à une branche précise de l'apprentissage automatique : l'apprentissage automatique supervisé. C'est uniquement de celui-ci qu'il sera question ici. Plus d'informations sur le sujet viendront plus tard dans des cours/articles dédiés.

Ce que l'on va faire, c'est **classer** (ou classifier) des objets. C'est le principe des algorithmes que je vais vous détailler dans ce cours. Ces objets vont avoir plusieurs **caractéristiques** (que l'on peut également appeler **attributs**). En fonction de ces attributs que l'on va explorer et tester, on va finir par attribuer une **classe** à chaque objet. Chacun de ces objets est appelé **exemple**. On va donc manipuler des exemples composés d'attributs pour leur attribuer une classe.

Il existe deux types d'exemples : les **exemples étiquetés** et les **exemples non étiquetés**. Leur différence est très simple. Les exemples étiquetés sont ceux qui sont pré-classés et qui donc vont être utilisés pour que le programme sache comment classer les exemples suivants. Les exemples suivants, vous l'aurez deviné, sont eux les exemples non étiquetés. La méthode qui va être utilisée pour déterminer comment étiqueter les exemples non étiquetés sur base des exemples étiquetés s'appelle **l'algorithme d'apprentissage**. Nous allons en voir deux ici : l'algorithme ID3 et son successeur, l'algorithme C4.5. Cet algorithme d'apprentissage (peu importe duquel il est question) va nous permettre d'extraire une règle que l'on appelle **concept** qui va nous dire comment on fait pour étiqueter un exemple.

Récapitulons :

nous allons créer un programme composé d'un **algorithme d'apprentissage** qui va nous permettre d'analyser des **exemples étiquetés** (eux-mêmes composés d'**attributs**) pour pouvoir **classer** d'autres exemples mais cette fois-ci des **exemples non étiquetés**, et ce à l'aide d'un **concept**.

Pour rendre les choses aussi simples que possible, nous allons considérer qu'il n'y a que deux classes possibles que nous pouvons appeler **0** et **1**. Nous allons également supposer qu'il y a toujours moyen de lier l'exemple à son étiquette. En réalité, il est possible que ça ne soit pas le cas, mais c'est tout de même plus simple de vous expliquer comment ça marche si tout se déroule bien !

> Notez que l'histoire du nombre d'étiquettes réduit à **2** n'est pas tout le temps d'application. Forcément, par soucis d'optimisation, on cherche toujours à limiter le nombre de possibilités et de tournures que peut prendre un problème. On tente donc de réduire le nombre d'étiquettes autant que possible. Mais on peut très bien être confronté à un problème de classification à **3**, **4** ou encore **150** classes !

Rassurez-moi, tout le monde est encore vivant à la fin de cette première partie un peu ardue ?

Revenons-en au déroulement général d'un problème d'apprentissage automatique. Le fonctionnement est le suivant : l'algorithme central est le premier élément de l'image, il est implémenté avant de faire fonctionner le programme. C'est d'ailleurs deux de ces algorithmes que nous allons étudier ici. L'étape suivante est de fournir un set d'exemples étiquetés pour que le programme puisse comprendre (grâce à l'algorithme central) par quel concept (quelle règle) les exemples ont été étiquetés. Une fois le concept extrait, on n'y touche plus, et on soumet de nouveaux exemples, cette fois-ci non étiquetés !, à ce concept afin que le programme fasse une analyse systématique et réussisse à étiqueter lui-même les nouveaux exemples fournis. Donc si vous regardez de plus près la figure 1, vous vous apercevrez qu'il y a présence de deux axes : un axe horizontal qui est le premier expérimenté, et qui n'est expérimenté qu'une seule fois, et puis il y a l'axe vertical qui, quant à lui, ne peut être opéré qu'après avoir opéré l'axe horizontal, et est utilisé plus d'une fois (sinon quel en serait l'intérêt ?).

1.2. État des lieux

###1. Les arbres de décisions En anglais *decision trees*, les arbres de décisions sont utilisés entre autres dans l'apprentissage automatique. Mais attention, c'est loin d'être leur seul domaine d'application ! Ce modèle est appelé arbre car il est composé de nœuds ayant chacun un certain nombre (variable) de fils. Le principe est de partir d'en haut de l'arbre (du nœud père de tous les autres, que l'on appelle *racine*) et de tester l'attribut en question et de suivre le chemin.

Juste avant d'en voir un exemple, il serait intéressant de donner une définition d'un arbre de décision.

> Un arbre de décision est un arbre[1] dont les nœuds représentent un choix sur un attribut, les arcs représentent les possibilités pour l'attribut testé et les feuilles représentent les décisions en fonction des compositions d'attributs.

Voici un exemple d'arbre de décision :

1. Donc un graphe connexe acyclique.

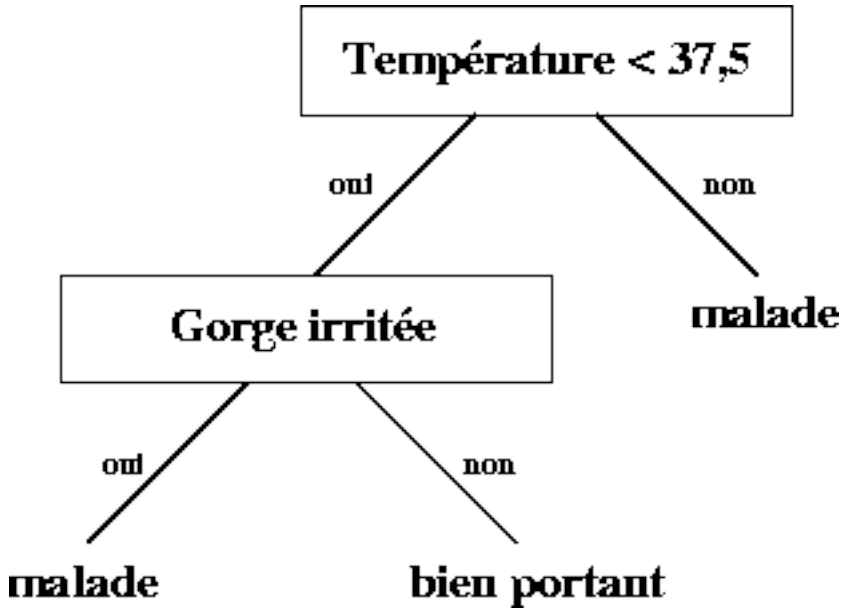

FIGURE 1.2. – exemple d'arbre de décision répondant à "Le patient est-il malade ?" (Fig. 2)

Laissez-moi vous expliquer la méthodologie pour parcourir un arbre de décision en *machine learning*. Pour ce, il faut déjà savoir à quoi il sert.

Considérons que cet arbre-ci est une méthode systématique pour traiter les cas d'inscriptions au MIT. Vous souvenez-vous de René, mon bon ami ? C'est ici qu'il réintervient : nous allons donc suivre étape par étape le traitement de sa demande d'inscription au MIT.

On commence tout en haut, donc par la question

> est diplômé du lycée (ou de l'enseignement secondaire) ?
>
> *Ordinateur du MIT en charge des inscriptions*

Ici, deux choix s'offrent à nous comme on peut le voir sur l'image. Deux branches partent de la première case. Ces deux branches sont **NON** et **OUI**. René a donc deux possibilités : soit il a un diplôme d'études supérieures, soit il n'en a pas. Jusque-là, tout le monde est d'accord avec moi, non ? Fort heureusement pour nous (et surtout pour René !), ce dernier a bien fini ses études. On suit donc la branche **OUI** (celle de droite) et on laisse tomber tout ce qui suit la branche de gauche.

On continue de descendre et on arrive à la question

> a plus de 18 ans ?
>
> *Toujours le même ordinateur*

À nouveau, deux choix s'offrent à nous : soit René a plus de 18 ans, soit il n'a pas plus de 18 ans.

i Et pour ceux qui rouspéteraient parce qu'on ne laisse pas à René la possibilité d'avoir 18 ans pile, je leur répondrai qu'il attendra le lendemain pour venir s'inscrire si c'est le cas, na !

C'est le jour de chance de notre ami René : il a eu 18 ans la semaine passée. Il peut donc passer sur la branche **OUI** (la branche de gauche). À nouveau, on oublie le reste des branches (donc uniquement la branche de droite).

On continue de descendre... Et l'ordinateur nous harcèle de questions : il nous demande maintenant

> a des parents riches ?
>
> *Encore et toujours cet ordinateur*

Euh... Bah René vient d'une famille relativement aisée mais mine de rien, on ne peut pas les qualifier de riches. Donc regardons ce que l'arbre nous propose : **OUI**, **MOYEN**, ou **NON**. Choisissons **MOYEN** parce que sa famille n'est ni pauvre ni riche. Alors on fait encore et toujours le même procédé, on suit la branche **MOYEN** et on oublie les deux autres. L'ordinateur en redemande :

> a droit à une bourse ?
>
> *si si... C'est toujours le même, pourquoi il aurait changé ?*

C'est là que ça coince. René est un pauvre étudiant français qui n'a pas droit à avoir une bourse aux États-Unis d'Amérique. On est alors contraints de suivre la branche **NON** de l'arbre et à nouveau d'oublier le reste.

Maintenant, l'ordinateur ne nous demande plus rien. Mais il nous affiche ceci :

> Non
>
> *Je vais le répéter à chaque fois ?*

Pourquoi ? C'est assez simple : on a continué sur la branche, mais on ne rencontre plus de nœud. On a rencontré une feuille. C'est le nom que l'on donne à une extrémité d'un arbre. Le fait de tomber sur une feuille nous fait arrêter de chercher vu que l'on vient de recevoir la réponse du MIT. Et cette réponse est non. Je vous laisse réessayer avec d'autres personnes que René qui ont d'autres caractéristiques ou plutôt d'autres **attributs**. Si ce mot ne vous dit rien, je vous recommande vivement de remonter d'un chapitre et de relire le **(§1.1.)**.

###2. ID3 ID3 est un des nombreux algorithmes possibles pour générer un arbre de décision, et c'est également celui qui va être développé dans la première partie de ce cours. Cet algorithme à été développé en **1986** par Ross Quinlan. Il l'a publié dans le magazine *Machine Learning* parmi d'autres articles.

###3. C4.5 C4.5 est en quelque sorte le petit frère d'ID3. C4.5 a également été développé par Ross Quinlan dans les années **1990** et a aussi été publié dans *Machine Learning*.

Je n'ai pas vraiment détaillé ces deux derniers chapitres car c'est ce que nous allons faire dans toute la suite de ce cours donc ne soyez pas impatients !

Il est donc nécessaire que vous ayez **bien compris** les notions abordées ici car vous en aurez besoin pour suivre la suite de ce cours ! Si vous n'avez pas retenu tout le vocabulaire exposé au point **2**, ce n'est pas très grave, vous pourrez y retourner en cas de trou de mémoire. Cependant, si vous n'avez pas saisi ce qu'était un arbre de décision et comment ça s'utilise ce machin là, il est préférable pour vous de relire ce chapitre car la suite sera beaucoup plus ardue sinon.

Quoi qu'il en soit, préparez-vous à entrer dans le vif du sujet avec l'exposition de l'algorithme ID3 !

Maintenant que vous savez ce qu'est un arbre de décision, nous allons pouvoir nous lancer dans la construction d'une telle structure.

2. source

2. Une première version : ID3

Dans cette partie, nous allons voir le principe de l'algorithme ID3 à l'aide d'un exemple. Ensuite je vous donnerai le pseudo-code afin de nous lancer dans une implémentation en Python3.

2.1. L'algorithme ID3

C'est, pour commencer, cet algorithme qui va être expliqué. *ID3* veut dire ***I**terative **D**ichotomiser **3***. Il va être expliqué à l'aide d'un premier exemple, puis l'algorithme sera explicité, et enfin quelques améliorations possibles seront expliquées.

###1. Exemple de playTennis Voici l'exemple utilisé par Quinlan en personne pour expliquer son algorithme dans le magazine *Machine Learning*. L'algorithme ID3 part d'un tableau (de manière plus générale, d'un *set*, ensemble en anglais) d'exemples étiquetés duquel va découler un arbre qui pourra prédire les étiquettes de nouveaux exemples non étiquetés donnés par après. Comme vous pouvez le voir, ID3 est bien un algorithme d'apprentissage automatique vu qu'il est bien question d'exemples, d'étiquettes, d'attributs, de classification, etc. et c'est bien un arbre de décision parce que... parce que c'est un arbre pardi !

####1.1. Tableau de valeurs

Voici le tableau contenant les informations nécessaires à l'explication de l'algorithme tel qu'expliqué par Ross Quinlan.

Jour	**Attributs des exemples**				**Classe**
	Prévisions	Température	Humidité	Vent	
1	**Ensoleillé**	**Chaud**	**Élevée**	**Faible**	**Non**
2	Ensoleillé	Chaud	Élevée	Fort	Non
3	Nuageux	Chaud	Élevée	Faible	Oui
4	Pluvieux	Moyen	Élevée	Faible	Oui
5	Pluvieux	Frais	Normale	Faible	Oui
6	Pluvieux	Frais	Normale	Fort	Non
7	Nuageux	Frais	Normale	Fort	Oui
8	Ensoleillé	Moyen	Élevée	Faible	Non
9	Ensoleillé	Frais	Normale	Faible	Oui

10	Pluvieux	Moyen	Normale	Faible	Oui
11	Ensoleillé	Moyen	Normale	Fort	Oui
12	Nuageux	Moyen	Élevée	Fort	Oui
13	Nuageux	Chaud	Normale	Faible	Oui
14	Pluvieux	Moyen	Élevée	Fort	Non

Table : Ensemble d'exemples pour playTennis

Prévisions, **Température**, **Humidité** et **Vent** sont les quatre attributs qui déterminent chacun des exemples qui vont être fournis. On peut voir qu'il existe uniquement **36** exemples différents pour cette configuration d'attributs :

$$= 3 \times 3 \times 2 \times 2 = 9 \times 4 = 36$$

1.2. Arbre complet et optimisé généré par l'algorithme

Voici ce à quoi on devrait arriver à la fin de ce chapitre : un arbre tout beau tout propre généré sur base du tableau que je viens de vous donner. Si vous testez chaque branche avec la méthodologie expliquée au-dessus **(§1.2.)**, vous vous apercevrez que l'arbre classe tous les exemples sans faute.

![l'arbre généré par ID3 de l'exemple playTennis (Fig. 3)](/media/galleries/4955/8108f502-02e8-4f08-aab9-a0b4abda6c94.gif)

2. Explication de l'algorithme L'algorithme ID3 se base sur le concept d'attributs et de classe de l'apprentissage automatique (sur classification discrète[$^{d}iscrte$]).$Cetalgorithmerecherchel'attributleplusper$

Pour trouver l'attribut à tester, Quinlan parle d'**entropie**. Pour définir l'entropie, il faut d'abord rappeler que la quantité minimale de données redondantes à ajouter pour qu'un message ayant une probabilité p d'arriver sans être corrompu est $-\log_2(p)$. Une telle nécessité de stocker des données redondantes est assez évidente dans le cas de codes correcteurs d'erreurs par exemple[^{C}CE].$Nevouslaissezpasavoirparle**-**devantle$log : le logarithme d'un nombre compris entre **0** et **1** est toujours négatif. Il faut donc prendre son opposé avec le **-**. Rappelons également que l'entropie est la longueur minimale nécessaire pour coder la classe d'un membre pris au hasard dans le set d'exemples S. C'est en réalité de l'**entropie de Shannon** qu'il est question. On l'appelle ainsi car c'est une notion trouvée par l'ingénieur américain Claude Shannon.[^{E}S]

[^{C}CE] : $Pourplusd'informations, c'est[ici](https://fr.wikipedia.org/wiki/Code_correcteur)etpluslarge$
$//fr.wikipedia.org/wiki/Th[^{E}S] : Plusdedtails : [ici](http://www.yann-ollivier.org/entropie/entropi$

Tel que p_c est la proportion d'exemples de S ayant pour classe résultante c. Dans l'exemple d'au-dessus, les seules classes possibles sont **Oui** et **Non**. donc l'entropie vaut

$$-p_{\text{Oui}} \times \log_2(p_{\text{Oui}}) - p_{\text{Non}} \times \log_2(p_{\text{Non}})$$

D'ailleurs voici à quoi ressemble la fonction d'entropie pour un ensemble à deux classes possibles (Fig. 4)

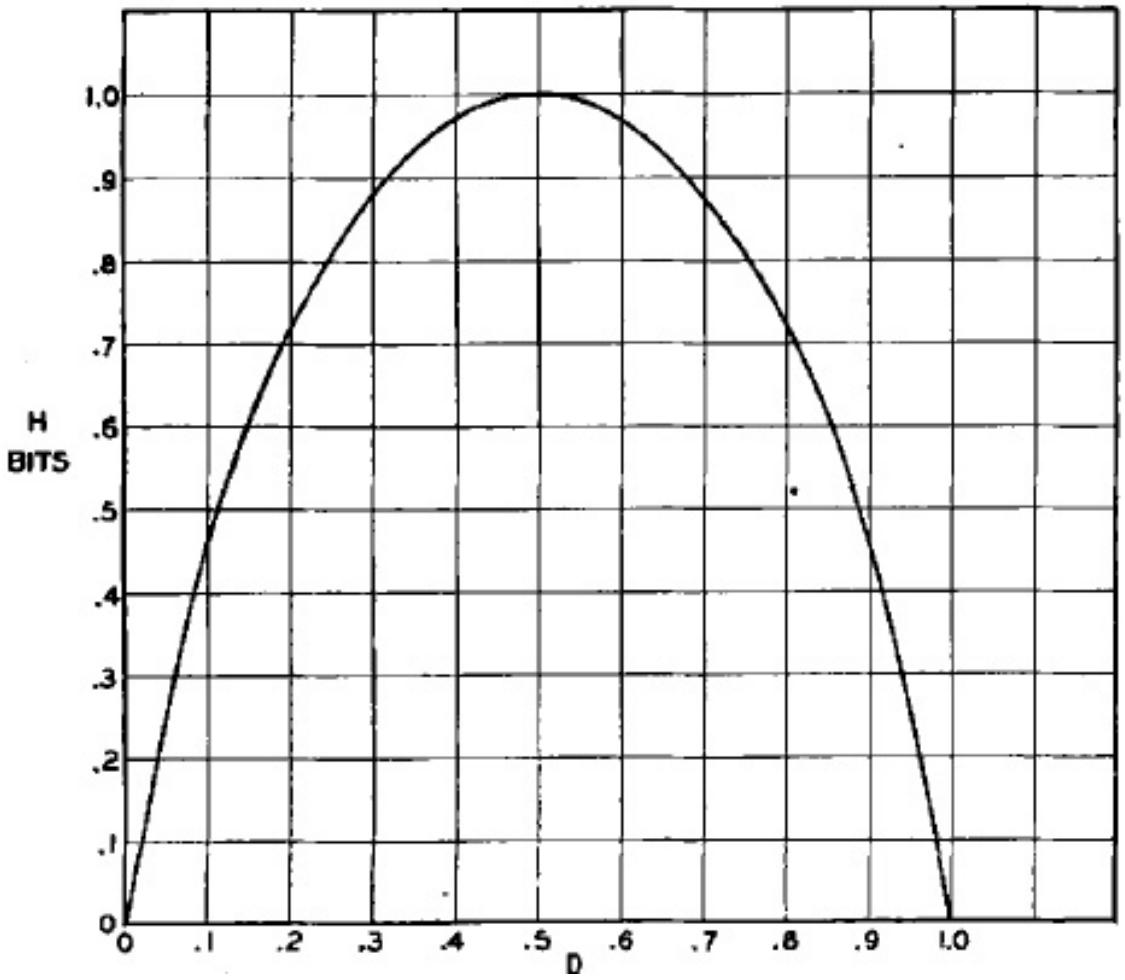

FIGURE 2.1. – le graphique d'entropie de Shannon pour un ensemble n'ayant que deux étiquettes (Fig. 4)

Remarquons quelque chose d'intéressant : cette fonction est parfaitement symétrique en prenant l'axe $x = \frac{1}{2}$. Et pour cause : la fonction d'entropie représente le "désordre" au sein du set (dans notre cas, ou du message dans le cas du code correcteur d'erreurs). Donc qu'il y ait **5** éléments d'une classe et **3** d'une autre ou inversement, le "désordre" reste le même car les proportions restent inchangées. On peut également voir trois points intéressants sur ce graphique (il y a en réalité une infinité de points intéressants pour ceux comme moi à qui ce graphe parle mais bon mon but est de ne pas vous effrayer donc nous allons nous intéresser qu'à ces trois là maintenant). Ces trois points sont les points d'abscisse **0**, **0.5** et **1**. Vous voyez où je veux en venir ? Si l'entropie représente le désordre du set, c'est logique que le set ait l'entropie la plus élevée quand il est le plus désordonné. Et quand est-ce qu'il est le plus désordonné ? Quand il a autant d'objets de la première classe que d'objets de la deuxième classe. Logique non ? Et inversement : quand le set a une entropie nulle (qui vaut **0**), c'est parce que c'est à ce moment que le désordre est le moins élevé. Et quand est-ce que le désordre est le moins élevé ? Quand tous les objets sont de la même classe. Donc quand la probabilité est soit minimum (**0**) soit maximum (**1**).

Initialement, l'algorithme prend tout le set $S = \{J_1, J_2, J_3, ..., J_{14}\}$. Et comme **9** des **14** exemples donnent la réponse (ou classe) **Oui** et **5** sur **14** donnent la réponse (ou classe) **Non**,

$$p_{\text{Non}} = \frac{5}{14}$$

On peut donc calculer[$^a pprox$]que

Il faut savoir que, comme on peut le voir sur la Fig. 4,

$$\forall S : 0 \leq \text{Entropie}(S) \leq 1$$

Ce qui veut dire que pour tout ensemble S, le désordre de S est toujours compris entre **0** et **1**.

Maintenant que nous savons que l'entropie initiale du set est de **0.94**, il nous faut savoir quel attribut tester en premier, puis en second, ..., puis en n-ième.

Pour savoir quel attribut tester, il faut connaître la notion de gain d'entropie. Le gain est défini par un set d'exemples et par un attribut. Cette formule va donc servir à calculer ce que cet attribut apporte au désordre du set. Plus un attribut contribue au désordre, plus il est important de le tester pour séparer le set en plus petits sets ayant une entropie moins élevée.

$$\text{Gain}(S, A) = \text{Entropie}(S) - \sum_{v \in \text{ valeurs}(A)} \frac{|S_v|}{|S|} \times \text{Entropie}(S_v)$$

L'attribut qui va être testé à ce nœud de l'arbre est le nœud qui va le plus réduire l'entropie. C'est logique : quand l'entropie vaut zéro, c'est qu'il n'y a qu'une seule classe représentée :

$$-1 \times \log_2(1) - n \times (0 \times \log_2(0)) = 0 - n \times 0 = 0$$

i

Ici, j'ai mis un **n** en évidence pour dire qu'il peut y avoir autant de classes que l'on veut. Mais si la probabilité d'une classe est **1**, on a beau avoir **150** classes, leur probabilité sera nulle à chaque fois. On peut donc mettre le nombre de classes en évidence. Mais attention, si je mets **n** en évidence, c'est parce qu'il y a **n** classes de probabilité nulle, mais il y a aussi une classe de probabilité **1**. Il y a donc **n+1** classes différentes.

Toujours dans notre exemple du point **2**, en considérant S comme le set initial, pour déterminer l'attribut à tester, il faut calculer le gain de tous les attributs :

####2.1. Calcul de gain d'entropie

#####2.1.1. Prévisions L'attribut **Prévisions** a trois valeurs possibles : {Ensoleillé, Nuageux, Pluvieux}.

$$
\begin{aligned}
\text{Gain}(S, \text{Prévisions}) = \text{Entropie}(S) \quad & - \frac{5}{14} \times \text{Entropie}(S_{\text{Ensoleillé}}) \\
& - \frac{4}{14} \times \text{Entropie}(S_{\text{Nuageux}}) \\
& - \frac{5}{14} \times \text{Entropie}(S_{\text{Pluvieux}}) \\
= 0.94 \quad & - \frac{5}{14} \times \left(-\frac{3}{5} \times \log_2\left(\frac{3}{5}\right) - \frac{2}{5} \times \log_2\left(\frac{2}{5}\right) \right) \\
& - \frac{4}{14} \times \left(-\frac{0}{4} \times \log_2\left(\frac{0}{4}\right) - \frac{4}{4} \times \log_2\left(\frac{4}{4}\right) \right) \\
& - \frac{5}{14} \times \left(-\frac{3}{5} \times \log_2\left(\frac{3}{5}\right) - \frac{2}{5} \times \log_2\left(\frac{2}{5}\right) \right) \\
= 0.94 \quad & - 0.357 \times (0.97) \\
& - 0.286 \times (0) \\
& - 0.357 \times (0.97) \\
= 0.24742 &
\end{aligned}
$$

#####2.1.2. Vent L'attribut **Vent** a quant à lui deux valeurs possibles : {Faible, Fort}.

$$
\begin{aligned}
\text{Gain}(S, \text{Vent}) = \text{Entropie}(S) \quad & - \frac{8}{14} \times \left(-\frac{6}{8} \times \log_2\left(\frac{6}{8}\right) - \frac{2}{8} \times \log_2\left(\frac{2}{8}\right) \right) \\
& - \frac{6}{14} \times \left(-\frac{3}{6} \times \log_2\left(\frac{3}{6}\right) - \frac{3}{6} \times \log_2\left(\frac{3}{6}\right) \right) \\
= 0.94 \quad & - 0.571 \times (0.811) \\
& - 0.428 \times 1 \\
= 0.048 &
\end{aligned}
$$

#####2.1.3. Humidité **Humidité** a également deux valeurs possibles : {Élevée, Normale}.

$$
\begin{aligned}
\text{Gain}(S, \text{Humidité}) = \text{Entropie}(S) \quad & - \frac{7}{14} \times \left(-\frac{4}{7} \times \log_2\left(\frac{4}{7}\right) - \frac{3}{7} \times \log_2\left(\frac{3}{7}\right) \right) \\
& - \frac{7}{14} \times \left(-\frac{6}{7} \times \log_2\left(\frac{6}{7}\right) - \frac{1}{7} \times \log_2\left(\frac{1}{7}\right) \right) \\
= 0.94 \quad & - 0.49 \\
& - 0.296 \\
= 0.153 &
\end{aligned}
$$

#####2.1.4. Température **Température** a trois valeurs différentes possibles : {Chaud, Chaud, Frais}.

$$\begin{aligned}
\text{Gain}(S, \text{Température}) = \text{Entropie}(S) \quad & - \frac{4}{14} \times \left(-\frac{2}{4} \times \log_2\left(\frac{2}{4}\right) - \frac{2}{4} \times \log_2\left(\frac{2}{4}\right) \right) \\
& - \frac{6}{14} \times \left(-\frac{4}{6} \times \log_2\left(\frac{4}{6}\right) - \frac{2}{6} \times \log_2\left(\frac{2}{6}\right) \right) \\
& - \frac{4}{14} \times \left(-\frac{3}{4} \times \log_2\left(\frac{3}{4}\right) - \frac{1}{4} \times \log_2\left(\frac{1}{4}\right) \right) \\
= 0.94 \quad & - 0.286 \\
& - 0.394 \\
& - 0.232 \\
= 0.028 &
\end{aligned}$$

Récapitulons : $\text{Gain}(S, \text{Température}) < \text{Gain}(S, \text{Vent}) < \text{Gain}(S, \text{Humidité}) < \text{Gain}(S, \text{Prévisions})$. On peut voir que le plus grand gain est pour **Prévisions**. C'est donc **Prévisions** qui est le premier attribut testé dans l'arbre. Si on regarde chaque nœud fils, on remarque que pour le nœud **Nuageux**, tous les résultats sont positifs. Il n'y a donc pas d'attribut à tester ici, on peut directement étiqueter **Oui**. Voici à quoi ressemble notre arbre.

/media/galleries/822/a9a95fc2-33c6-4a0b-8586-7

FIGURE 2.2. – arbre à la première itération de sa réalisation (Fig. 5)

Image perdue

J'ai utilisé une notation entre crochets de la forme **[n+, m-]**. Cette notation veut dire que dans l'ensemble ayant **n+m** éléments, **n** sont positifs et **m** sont négatifs. Dans notre cas, les positifs sont les exemples étiquetés **Oui** et les négatifs sont ceux étiquetés **Non** bien entendu.

Il faut maintenant continuer à mettre des nœuds après **Ensoleillé** et **Pluvieux** car tous les exemples ne donnent pas le même résultat. Déterminons donc pour **Ensoleillé** quel est le meilleur attribut à tester en utilisant à nouveau le gain. Cependant il n'est plus utile de tester le gain de **Prévisions** étant donné qu'il vient d'être utilisé. Je vous donne juste les résultats, je vous laisse faire les calculs vous-même pour vous entrainer.

Rappel : pour ceux n'ayant pas de fonctionnalité $\log_2$ sur leur calculatrice :

$$\log_2(f(x)) = \frac{\log(f(x))}{\log(2)}$$

Voici donc ce que vous devez normalement obtenir :

$$\text{Gain}(S_{\text{Ensoleillé}}, \text{Température}) = 0.571$$
$$\text{Gain}(S_{\text{Ensoleillé}}, \text{Humidité}) = 0.971$$
$$\text{Gain}(S_{\text{Ensoleillé}}, \text{Vent}) = 0.019$$

Donc récapitulons à nouveau :

Le plus grand gain est pour **Humidité**. D'ailleurs on peut voir que le gain est égal à l'entropie de $S_{\text{Ensoleillé}}$. Ça veut dire que tous les fils de **Humidité** donneront une étiquette. Voilà notre arbre jusqu'à présent. Il nous reste à continuer l'arbre du côté de **Pluvieux**. Voici les gains pour les différents attributs (à nouveau, il n'est pas nécessaire de tester **Prévisions** qui vient de l'être mais il faut tester **Humidité** qui n'a pas été testé de ce côté-là de la branche) :

$$\text{Gain}(S_{\text{Pluvieux}}, \text{Température}) = 0.019$$
$$\text{Gain}(S_{\text{Pluvieux}}, \text{Humidité}) = 0.019$$
$$\text{Gain}(S_{\text{Pluvieux}}, \text{Vent}) = 0.971$$

Récapitulons une fois de plus :

Le plus grand gain est à nouveau de **0.971** et est pour **Vent**. Il nous faut donc tester **Vent** et vu que le gain est égal à l'entropie de S_{Pluvieux}, chaque nœud fils de **Vent** sera une étiquette.

Voici donc notre arbre. Il n'y a plus rien à faire étant donné qu'il ne reste aucun nœud qui n'est pas étiqueté. Le travail est donc accompli et voilà l'arbre que nous avons généré.

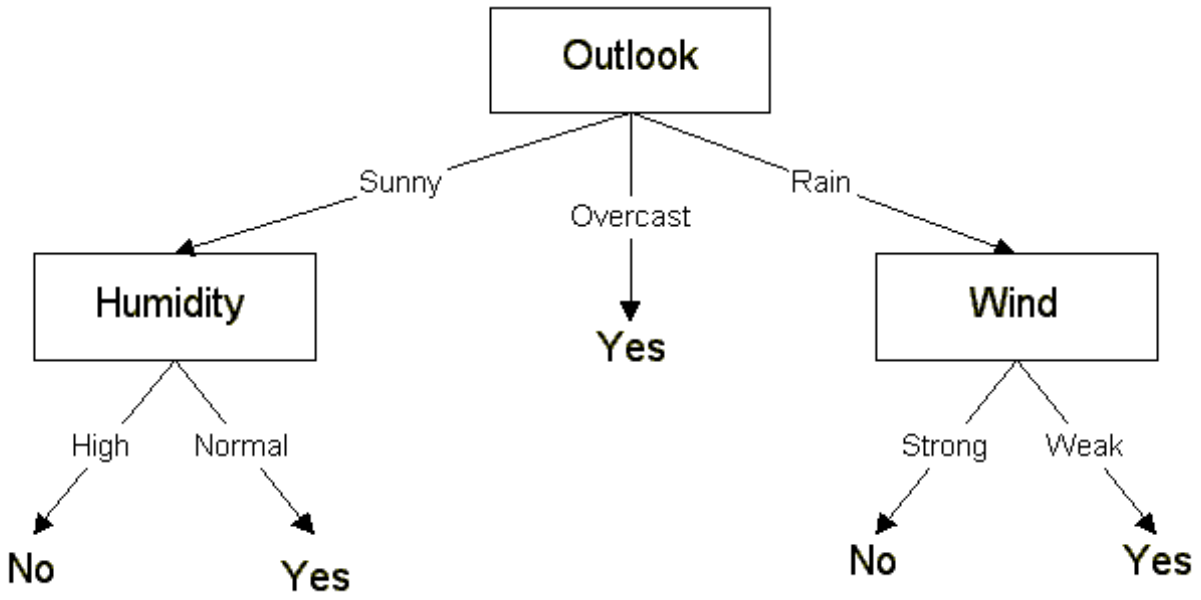

FIGURE 2.3. – l'arbre généré par ID3 de l'exemple playTennis (Fig. 6)

Si vous comparez l'arbre que nous venons de générer avec l'arbre que je vous ai donné en début de tutoriel, vous pouvez constater que c'est exactement le même. L'arbre est optimisé car il donne une étiquette en maximum deux tests alors qu'il y a 4 attributs différents.

Vous ne voyez peut-être pas l'utilité de cet arbre parce qu'ici il y a maximum **36** combinaisons des attributs et **14** nous sont déjà données. Mais imaginez maintenant que vous voulez faire un arbre qui va déterminer un diagnostic pour des malades. Comme il existe des milliers de maladies et encore plus de symptômes, l'arbre sera plus difficile à faire à la main. Ceci dit, tant que tous les exemples possibles n'ont pas été donnés il est possible que l'arbre contienne des erreurs. Mais nous y reviendrons plus tard.

###3. Algorithme et pseudo-code de l'algorithme ID3 Voici l'algorithme de génération d'un arbre de décision selon ID3 sous forme de pseudo-code. Ce chapitre est pour les programmeurs. Si vous êtes uniquement venus pour les maths, ce chapitre ne vous concerne pas directement et vous n'êtes pas obligés de lire ce qui suit.

Je ne vais pas détailler le pseudo-code, ça devrait sembler relativement clair. Et si vous avez tout de même besoin d'aide à la compréhension, regardez dans les commentaires, peut-être trouverez-vous votre réponse. Et si vous ne l'y trouvez pas, posez votre question sur les forums, c'est à ça qu'ils servent.

```
1  FONCTION ID3
2          Entrée :
3                  - exemples = liste d'exemples étiquetés
4                  - questions = liste des attributs non utilisés
                       jusqu'à présent
5          Retour :
6                  - un nœud
7  DEBUT
8          SI exemples est vide, alors
9                  Finir la fonction sans construire de nœud
10         FIN SI
11
12         SI tous les exemples sont la même classe, alors
13                 Retourner une feuille ayant cette classe
14         FIN SI
15
16         SI questions est vide, alors
17                 Retourner une feuille avec la classe la plus
                       fréquente
18         FIN SI
19
20         q = attribut optimal (avec le plus grand gain d'entropie)
21
22         n = nouveau nœud créé qui testera l'attribut q
23         POUR CHAQUE v = valeur possible de q, FAIRE
24                 e = l'ensemble des éléments de exemples ayant v
                       comme valeur à l'attribut q
```

```
                chaque nœud fils de n est créé par ID3(e,
                    questions\{q})
        FIN POUR CHAQUE
        Retourner n
FIN
```

Notez que si la condition `SI questions est vide` est remplie, ça veut dire que deux exemples identiques mais étiquetés différemment ont été mis dans le set d'exemples de test.

###4. Améliorations Ici sont présentées d'abord les améliorations que l'on peut apporter aux notions vues au **(§2.)**.[5]

Commençons par la fonction d'Entropie. Une petite étude mathématique nous montre que la probabilité d'une classe dans un set n'est autre que la longueur du sous-set de cette classe divisée par la longueur du set initial. Donc

$$
\begin{aligned}
\text{Entropie}(S) &= \sum_{c \in \text{classes}(S)} -p_c \times \log_2(p_c) \\
&= \sum_{c \in \text{classes}(S)} -\frac{|c|}{|S|} \times \log_2\left(\frac{|c|}{|S|}\right) \\
&= -\frac{1}{|S|} \sum_{c \in \text{classes}(S)} |c| \times \log_2\left(\frac{|c|}{|S|}\right) \\
&= -\frac{1}{|S|} \sum_{c \in \text{classes}(S)} |c| \times (\log_2|c| - \log_2|S|) \\
&= -\frac{1}{|S|} \left(\sum_{c \in \text{classes}(S)} |c| \times \log_2|c| - \sum_{c \in \text{classes}(S)} |c| \times \log_2|S| \right) \\
&= -\frac{1}{|S|} \left(\sum_{c \in \text{classes}(S)} |c| \times \log_2|c| - \log_2|S| \sum_{c \in \text{classes}(S)} |c| \right) \\
&= -\frac{1}{|S|} \left(\sum_{c \in \text{classes}(S)} |c| \times \log_2|c| - |S| \times \log_2|S| \right) \\
&= -\frac{1}{|S|} \sum_{c \in \text{classes}(S)} |c| \times \log_2|c| + \frac{|S|}{|S|} \times \log_2|S| \\
&= 1 \times \log_2|S| - \frac{1}{|S|} \sum_{c \in \text{classes}(S)} |c| \times \log_2|c| \\
&= \log_2|S| - \frac{\sum_{c \in \text{classes}(S)} |c| \times \log_2|c|}{|S|}
\end{aligned}
$$

Alors cette petite démonstration peut vous paraître rude, mais elle est très bien car elle nous permet de faire beaucoup moins de calculs. C'est très pratique tant pour programmer (car le programme sera plus rapide) que pour faire les calculs de tête (vous risquez beaucoup moins

de faire des erreurs). On a donc simplifié notre splendide fonction d'entropie et on a gardé un indice sommatoire tout de même. Ouf !

? Quid de la fonction de Gain(S, A) ?

Cette fonction varie entre **0** et Entropie(S) vu que :

$$0 < \sum_{v \in \text{valeurs}(A)} \frac{|S_v|}{|S|} \times \text{Entropie}(S_v) < \text{Entropie}(S)$$

Quand la somme vaut **0**, Gain(S, A) vaut Entropie(S) ce qui est son maximum. C'est donc cet attribut qui va être choisi. Alors que si la somme vaut Entropie(S), gain vaut **0** et donc cet attribut n'est pas intéressant à tester à ce moment vu qu'il ne réduit pas l'entropie (qui est, rappelons-le, le désordre des classifications du set). La seule chose qui nous intéresse dans cette fonction est donc la somme qui doit être la plus petite possible (il nous faut avoir le moins de désordre). On peut donc changer la fonction comme ceci :

$$\text{Perte}(S, A) = \sum_{v \in \text{valeurs}(A)} \frac{|S_v|}{|S|} \times \text{Entropie}(S_v) = \frac{1}{|S|} \times \sum_{v \in \text{valeurs}(A)} |S_v| \times \text{Entropie}(S_v)$$

Sauf que je viens de vous montrer comment améliorer la fonction d'entropie. On peut donc encore simplifier la fonction de gain !

$$\begin{aligned}
\text{Perte}(S, A) &= \frac{1}{|S|} \times \sum_{v \in \text{valeurs}(A)} |S_v| \times \textit{Entropie}(S_v) \\
&= \frac{1}{|S|} \times \sum_{v \in \text{valeurs}(A)} |S_v| \times \left(\log_2 |S_v| - \frac{1}{|S_v|} \times \sum_{c \in \text{classes}(S_v)} |c| \times \log_2 |c| \right) \\
&= \frac{1}{|S|} \times \left(\sum_{v \in \text{valeurs}(A)} |S_v| \times \log_2 |S_v| - \sum_{v \in \text{valeurs}(A)} \frac{|S_v|}{|S_v|} \times \sum_{c \in \text{classes}(S_v)} |c| \times \log_2 |c| \right) \\
&= \frac{1}{|S|} \times \left(\sum_{v \in \text{valeurs}(A)} |S_v| \times \log_2 |S_v| - \sum_{v \in \text{valeurs}(A)} \sum_{c \in \text{classes}(S_v)} |c| \times \log_2 |c| \right)
\end{aligned}$$

Je ne pense pas que cette amélioration-ci ait réellement une influence en termes de performances lors d'un programme mais c'est plus facile de faire les calculs à la main avec cette formule. La double somme peut faire un petit peu peur, mais ce n'est pas bien compliqué : ça veut dire qu'on fait la somme sur tous les **v**, et que pour chaque **v**, on ajoute la somme sur tous les **c**.

PS : faites bien attention, ici c'est bien un indice sommatoire sur c dans les classes de S_v et plus juste dans S. La raison est bien simple, c'est ce qui est expliqué juste au-dessus, c'est lié à la double somme.

Ces améliorations ne changent rien à l'algorithme. Ce sont juste de légères optimisations pour éviter de trop calculer. C'est pour ceux qui programment leur algorithme mais également pour ceux qui font leurs calculs sur papier et à la calculette (ou pour les plus balèzes, qui font les calculs de logarithmes avec leurs tables) et qui ne veulent pas tout le temps calculer les mêmes choses ce qui augmente la probabilité d'erreurs.

2.2. Une implémentation

###5. Implémentation

Je vais vous proposer une implémentation détaillée en python3. Je vas vous guider étape par étape dans la réalisation de ce programme. Et pour ceux qui ne savent pas vraiment programmer ce n'est pas grave, le code complet sera mis à la fin.

Le code que je vous propose est une implémentation que j'ai choisie et que j'ai développée. Elle n'est pas la plus performante mais les choix sont avant tout pédagogiques.
Le code qui suit est pseudo-orienté objet (dans la mesures de ce que Python permet), il vous faut être relativement à l'aise avec cette notion pour pouvoir le comprendre !

####5.1. Les types de données nécessaires

Commençons par déterminer de quelles structures de données nous allons avoir besoin. Voici ce que je propose, je vous laisse le lire, puis j'expliquerai.

```
classe Nœud:
    * enfants: dict
    * attribut_testé: str

classe Feuille:
    * étiquette: str

classe Exemple:
    * étiquette: str
    * attributs: dict

```

3. ici, la classification **discrète** s'oppose à la classification **continue**. Les mathématiques discrètes concernent les opérations sur des ensembles finis (ex : les possibilités d'un jet de dé : $\{1, 2, 3, 4, 5, 6\}$) alors que les mathématiques continues concernent les opérations sur des ensembles infinis (ex : $\mathbb{R}$)

4. j'ai tendance à arrondir à la seconde décimale histoire que tout le monde ait une idée de l'ordre de grandeur pour pouvoir faire des comparaisons. Mais pour une question de lisibilité, je préfère mettre le symbole $=$ que le symbole $\simeq$. La valeur ici n'est donc pas précisément 0.94 mais s'en rapproche assez fort. On peut donc se permettre une approximation.

5. ce chapitre n'est pas indispensable à la lecture. C'est un développement mathématique sur les logarithmes qui permet de simplifier l'écriture et/ou le sens des formules. Vous pouvez soit l'ignorer et passer à la suite, soit regarder à quelles formules on arrive sans regarder le reste, soit suivre toute la pseudo-démonstration (qui s'apparente plus à un développement et à des substitutions). Je vous conseille de faire ce que votre cerveau pourra tolérer en terme de maths, je ne vous en voudrai pas si vous ne le lisez même pas

```
classe Ensemble:
    * noms_attributs: list de str
    * liste_exemples: list de Exemple

classe Arbre_ID3:
    * arbre: Nœud/Feuille
    * ensemble: Ensemble
```

Voilà le moment où je dois vous l'expliquer. J'ai donc fait un découpage en classes. Peut-être auriez-vous fait autrement mais voilà, c'est ce que j'ai choisi.

? Pourquoi ces classes là ?

Bon pour commencer, laissez-moi expliquer la classe Arbre_ID3. Cette classe contient deux informations importantes :

— l'*arbre* en lui-même ;
— et l'*ensemble* de données sur lequel il est construit.

Cet ensemble, il est de classe Ensemble (oui je sais c'est pas très imaginatif mais au moins, c'est clair). La classe Ensemble quant à elle contient également deux informations mais ce ne sont pas du tout les mêmes :

— le *nom des **attributs*** qui sont à utiliser ;
— et les *exemples*.

Le nom des attributs est stocké dans une liste (qui est un type interne à Python[6]) et les exemples sont également stockés dans une liste. Mais que contiennent ces listes me direz-vous !

? ... Mais que contiennent ces listes ?

Merci de poser la question, ça me permet de vous expliquer la classe Exemple ! La première liste, *le nom des attributs* ne contient que des chaines de caractères, ou des *Strings*. Ce type s'appelle `str` en Python. Par contre, la seconde liste, celle qui contient les exemples contient des... Exemples ! donc des objets de la classe Exemple.

Cette classe Exemple contient deux informations importantes :

— l'*étiquette* de l'exemple ;
— et ses *attributs*.

L'étiquette est une `str` alors que les attributs sont un dictionnaire ayant pour clef le nom de l'attribut et comme valeur la valeur de l'attribut. Logique, non ?

Je crois qu'on en a fini pour vous expliquer ce qui était contenu dans la variable *ensemble* de la classe Arbre_ID3... Mais il nous reste la variable *arbre* !

Cette variable *arbre* est soit un objet de type Feuille soit un objet de type Nœud. Étant donné que la notion d'arbre est un prérequis pour ce cours, je considère que vous connaissez ces deux termes et je ne m'attarderai pas dessus.

La classe Feuille ne contient qu'un élément de type `str`. C'est vrai que ce n'était pas obligatoire d'en faire une classe, mais je préférais pour une raison de clarté.

La classe Nœud quant à elle contient la valeur de l'attribut testé dans un `str` et un dictionnaire de Nœuds. Effectivement, c'est bien le principe : pour faire un arbre n-aire, il nous faut pouvoir mettre autant de fils que possible à chaque nœud.

Nous avons maintenant fait le tour des variables, nous pouvons passer à la phase de développement du code, du vrai !

####5.2. Le code le vrai

La première chose que doit faire notre code, c'est savoir sur quelles données il va travailler. On pourrait soit rentrer toutes nos données de la classe Ensemble à la main, mais j'ai le sentiment que ça va être long, fastidieux et inutile... Je vous propose donc de créer une fonction qui va récupérer des données dans un fichier !

#####5.2.1. implémentation des types de données basiques

Cette fonction est une méthode de la classe Ensemble. Pourquoi ? C'est simple : on veut faire une fonction qui va remplir notre classe Ensemble et donc créer et modifier ses éléments. Cette méthode est donc dans notre classe Ensemble. Je propose même que ça fasse partie de notre constructeur ! je vais tout de même laisser la possibilité de créer un Ensemble simple, vous comprendrez plus tard pourquoi.

Voici donc le début de notre classe Ensemble qui contient un constructeur et une fonction qui nous permet d'alléger le code du constructeur.

```
1  class Ensemble:
2      """
3          Un ensemble contient deux valeurs :
4              - les noms des attributs (list)
5              - les exemples (list)
6      """
7
8      def __init__(self, chemin=""):
9          """
10             chemin est l'emplacement du fichier contenant les données.
11             Cette variable peut être non précisée en quel cas les variables
12                 seront initialisées comme des listes vides.
13         """
14         #Python est un langage à typage dynamique fort,
15         #il faut donc vérifier que l'utilisateur ne fait pas
               n'importe quoi
16         #en passant autre chose qu'un str
17         if not isinstance(chemin, str):
18             raise TypeError("chemin doit être un str et non {}" \
19                             .format(type(chemin)))
20         if chemin == "":
21             #initialisation en listes vides
22             self.liste_attributs = list()
```

```
23              self.liste_exemples = list()
24          else:
25              with open(chemin, 'r') as fichier:
26                  #on stocke chaque mot de la première ligne dans
                        liste_attributs
27                  self.liste_attributs = \
28
                                        fichier.readline().lower().strip().split('
29                  #ensuite on stocke la liste d'exemples dans
                        liste_exemples
30                  self.liste_exemples = self.liste_en_exemples(
31
                                              fichier.read().strip().lower().spl
32                                        self.liste_attributs
33                                    )
34
35      @staticmethod #fonction statique car ne dépend pas de l'objet
            mais est commune à toute instance de la classe
36      def liste_en_exemples(exemples, noms_attributs):
37      """
38          retourne une liste d'exemples sur base d'une liste de str contenant
39          les valeurs et d'une liste de str contenant les noms des attributs
40      """
41      #on initialise la liste à retourner
42      ret = list()
43      for ligne in exemples:
44          #on stocke chaque mot de la ligne dans une liste attributs
45          attributs = ligne.lower().strip().split(' ')
46          #met l'étiquette par défaut si elle n'est pas dans la ligne
47          etiquette = attributs[-1] if len(attributs) !=
                len(noms_attributs) \
48                                      else ""
49          #on ajoute un objet de type Exemple contenant la ligne
50          ret.append(Exemple(noms_attributs,
51                             attributs[:len(noms_attributs)],
52                             etiquette))
53          return ret
```

Voici comment nous allons coder le constructeur d'Exemple : cette fonction doit prendre en paramètres la liste contenant le nom des attributs, puis la liste contenant la valeur de chaque attribut, puis l'étiquette.

Voilà donc le début de notre classe `Exemple` :

```
1   class Exemple:
2       """
3           Un exemple contient 2 valeurs :
4               - un dictionnaire d'attributs (dict)
```

```
            - une étiquette (str)
    """

    def __init__(self, noms_attributs, valeurs_attributs,
        etiquette=""):
        """
            etiquette peut être non précisée en quel cas on aurait
            un exemple non étiqueté
        """
        #si on a un problème de types
        if not isinstance(noms_attributs, list) \
        or not isinstance(valeurs_attributs, list):
            raise
                TypeError("noms_attributs et valeurs_attributs doivent être"
                \
                            " des listes et pas des {0} et {1}" \
                            .format(type(noms_attributs),
                                    type(valeurs_attributs)))
        if not isinstance(etiquette, str):
            raise
                TypeError("etiquette doit être un str et pas un {}"
                \
                            .format(type(etiquette)))
        #si les deux listes n'ont pas le même nombre d'éléments
        if len(valeurs_attributs) != len(noms_attributs):
            raise
                ValueError("noms_attributs et valeurs_attributs doivent "
                \
                            "avoir le même nombre d'éléments")
        self.etiquette = etiquette
        self.dict_attributs = dict()
        #on ajoute chaque attribut au dictionnaire
        for i in range(len(noms_attributs)):
            self.dict_attributs[noms_attributs[i]] =
                valeurs_attributs[i]
```

Lançons-nous maintenant dans la création de l'arbre en lui-même. Il nous faut donc manipuler la classe Arbre_ID3, et donc, la créer ! Le constructeur doit prendre une variable, à savoir le chemin vers le fichier de données à envoyer au constructeur de sa variable ensemble :

```
class Arbre_ID3:
    """
        Un arbre ID3 contient deux valeurs :
            - un ensemble d'exemples (Ensemble)
            - un arbre (Noeud)
    """

    def __init__(self, chemin=""):
```

```
9           """
10              chemin est l'emplacement du fichier contenant les données
11          """
12          #initialisation de l'ensemble avec le fichier dans chemin
13          self.ensemble = Ensemble(chemin)
14          #initialisation du noeud principal de l'arbre
15          self.arbre = None
```

Voilà la base de notre classe. Sauf que jusqu'ici, on ne sait qu'initialiser les données, nous ce qu'on veut faire, c'est bien plus !

Il nous faut donc pouvoir créer l'arbre. Voici la méthode de Arbre_ID3 que je vous propose :

```
1  def construire(self):
2      """
3          génère l'arbre sur base de l'ensemble pré-chargé
4      """
5      self.arbre = self.__construire_arbre(self.ensemble)
```

Alors, vous aimez ?

Bon d'accord, je le reconnais, je n'ai pas codé la réalisation de l'arbre... Mais c'est fait exprès ! J'ai créé une petite fonction toute simple à appeler par l'utilisateur pour que l'arbre se crée. Cette fonction appelle la fonction *privée* `__construire_arbre`.

#####5.2.2. La construction de l'arbre

Nous allons maintenant coder cette fonction, mais étape par étape.

######5.2.2.1. Les tests

```
1  def __construire_arbre(self, ensemble):
2      """
3          fonction privée et récursive pour la génération de l'arbre
4      """
5      if not isinstance(ensemble, Ensemble):
6          raise
               TypeError("ensemble doit être un Ensemble et non un {}"
               \
7                            .format(type(ensemble)))
```

Voici la base de la définition de notre fonction (qui est donc toujours dans la classe Arbre_ID3). Si vous reprenez le pseudo-code que je vous ai donné deux chapitres au-dessus, vous pouvez voir qu'il nous faut tester si la liste d'exemples est vide, et si elle l'est, retourner une erreur. Nous allons utiliser les exceptions qui sont faites pour cela en python. Voici donc le test :

```python
#si la liste est vide
if len(ensemble) == 0:
    raise ValueError("la liste d'exemples ne peut être vide !")
```

Ici, je fais appel à la fonction `len` sur une classe qui n'est pas prévue à la base. Il nous faut donc ajouter la fonction `__len__` à notre classe `Ensemble`. Étant donné qu'un minimum de connaissances en Python3 orienté objet est nécessaire, ce ne sera pas approfondi.

Voici donc la fameuse méthode qui nous permet de calculer la longueur de l'ensemble :

```python
def __len__(self):
    """
        retourne la longueur de l'ensemble
    """
    return len(self.liste_exemples)
```

Je confirme, rien de bien compliqué, cependant, je n'ai jamais dit le contraire ! On aurait très bien pu faire sans, mais c'est quand même plus lisible comme ça.

Après cela, il faut ajouter le test des classes. Comment on peut savoir si tous les exemples ont la même étiquette ? Facile, il suffit de les tester un par un et de voir si elle est différente du précédent (ou du premier)... Oui mais pour montrer qu'on a bien compris l'affaire, on peut tester l'entropie ! Que vaut l'entropie quand tous les exemples ont la même étiquette ? Elle vaut 0 !

```python
#testons si tous les exemples ont la même étiquette
if ensemble.entropie() == 0:
    #on retourne l'étiquette en question
    return Feuille(ensemble.liste_exemples[0].etiquette)
```

Ouille ! J'ai introduit ici beaucoup de choses ! Premièrement, j'ai créé une méthode entropie dans la classe Ensemble. Deuxièmement, j'ai créé la classe Feuille et son constructeur. Et le constructeur prend un seul paramètre (en plus de `self` bien entendu) de type `str`. Ce paramètre est l'étiquette vu que c'est la seule valeur que contient la classe Feuille.

Commençons par la classe Feuille, c'est le plus simple. Il suffit de créer une nouvelle classe et son constructeur, voici le code :

```python
class Feuille:
    """
        Une feuille contient uniquement une valeur:
            - l'étiquette (str)
```

```
    """

    def __init__(self, etiquette):
        """
            etiquette doit obligatoirement être un str
        """
        if not isinstance(etiquette, str):
            raise
                TypeError("etiquette doit être un str et pas un {}"
                \
                                .format(type(etiquette)))
        self.etiquette = etiquette
```

Ce n'est pas bien compliqué. Par contre, pour la méthode d'entropie, ça risque de l'être un peu plus.

######5.2.2.2. Le calcul d'entropie

Pour plus de clarté dans le code, je n'implémenterai pas l'*amélioration* proposée au chapitre précédent. Cependant, libre à vous de le faire si vous voulez !

Pour coder notre fonction, nous allons avoir besoin du logarithme inclus dans le module math. Donc :

```
from math import log
```

Ensuite nous allons créer la méthode dans notre classe Ensemble. Voici la méthode que je vous propose :

```
def entropie(self):
    """
        retourne l'entropie de Shannon de l'ensemble
    """
    #initialisation de la variable retournée
    ret = 0
    #pour chaque étiquette de l'ensemble
    for etiquette in self.etiquettes_possibles():
        #on crée un sous-ensemble qui ne contient que les éléments
            de
        #self ayant etiquette comme étiquette
        sous_ensemble = self.sous_ensemble_etiquette(etiquette)
        #on ajoute |c| * log_2(|c|) à ret
        longueur_sous_ensemble = len(sous_ensemble)
        ret += longueur_sous_ensemble * log(longueur_sous_ensemble,
            2)
```

```
15      #on retourne log_2(|S|) - ret/|S|
16      return log(len(self), 2) - ret/len(self)
```

A nouveau, ce code introduit de nouvelles fonctions de la classe Ensemble : `etiquettes_possibles()` et `sous_ensemble_etiquette()`.

Voici la fonction etiquettes_possibles. Elle a pour but de renvoyer une liste contenant toutes les étiquettes possibes dans l'ensemble d'exemples. :

```
1  def etiquettes_possibles(self):
2      """
3          retourne une liste contenant les étiquettes de l'ensemble
4      """
5      #on initialise la valeur de retour
6      ret = list()
7      #pour chaque exemple de l'ensemble
8      for exemple in self.liste_exemples:
9          #si l'étiquette n'est pas déjà dans la liste
10         if not exemple.etiquette in ret:
11             #on l'ajoute
12             ret.append(exemple.etiquette)
13     return ret
```

Et voici la fonction sous_ensemble_etiquette(). Son but à elle est de créer un sous-ensemble (comme son nom l'indique) ne reprenant que les exemples ayant la bonne étiquette :

```
1  def sous_ensemble_etiquette(self, etiquette):
2      """
3          retourne un ensemble contenant uniquement les exemples ayant
4          etiquette comme étiquette
5      """
6      #initialisation de la valeur de retour
7      ret = Ensemble()
8      #on copie la liste d'attributs
9      ret.liste_attributs = self.liste_attributs[:]
10     #pour chaque exemple de l'ensemble initial
11     for exemple in self.liste_exemples:
12         #si l'étiquette est bonne
13         if exemple.etiquette == etiquette:
14             #on l'ajoute au sous-ensemble
15             ret.liste_exemples.append(exemple)
16     return ret
```

On peut donc reprendre le code de notre fonction de construction !

######5.2.2.3. La liste d'attributs

Il nous faut maintenant tester si la liste d'attributs à tester est vide. Si elle l'est, il faut retourner une feuille avec l'étiquette la plus fréquente. Ce qui se fait de la manière suivante :

```
#s'il ne reste d'attribut à tester
if len(ensemble.liste_attributs) == 0:
    max, etiquette_finale = 0, ""
    #on teste toutes les étiquettes possibles de l'ensemble
    for etiquette in ensemble.etiquettes_possibles():
        sous_ensemble = ensemble.sous_ensemble_etiquette(etiquette)
        #si c'est la plus fréquente, c'est celle qu'on choisit
        if len(sous_ensemble) > max:
            max, etiquette_finale = len(sous_ensemble), etiquette
    #et on la retourne dans une feuille
    return Feuille(etiquette_finale)
```

######5.2.2.4. Les nœuds enfants

Maintenant, il faut coder la partie la plus importante : la partie récursive !

C'est maintenant que l'on va pouvoir utiliser la puissance de la récursivité pour pouvoir générer tout l'arbre.

Commençons par trouver l'attribut optimal avec ceci :

```
a_tester = self.attribut_optimal()
```

La méthode attribut_optimal de la classe Ensemble peut être définie comme ceci :

```
def attribut_optimal(self, ID3=True):
    """
        retourne un str avec le nom de l'attribut à tester
    """
    max, ret = float("-inf"), ""
    #pour chaque attribut
    for attribut in self.liste_attributs:
        gain = self.gain_entropie(attribut)
        #si le gain d'entropie est la plus grande
        if gain >= max:
            #on le garde en mémoire
            max, ret = gain, attribut
    #et on le retourne
    return ret
```

Cette fonction nécessite que l'on code la fonction de gain d'entropie. Ici, deux choix s'offrent à nous :

— soit on prend la forme simple à implémenter mais non optimisée que voici :

```
def gain_entropie(self, nom_attribut):
    """
        retourne la perte d'entropie selon la définition de Ross Quinlan
    """
    somme = 0
    #pour chaque valeur de l'attribut en question
    for valeur in self.valeurs_possibles_attribut(nom_attribut):
        #déclaration de Sv
        sous_ensemble = self.sous_ensemble_attribut(nom_attribut,
            valeur)
        #somme = somme sur v de |Sv| * Entropie(Sv)
        somme += len(sous_ensemble) * sous_ensemble.entropie()
    #Gain(S, A) = Entropie(S) - 1/|S| * somme
    return self.entropie() - somme/len(self)
```

— soit on prend la version *optimisée* donnée au chapitre précédent que je n'implémenterai pas comme mentionné plus haut.

Dans les deux cas, il est nécessaire de faire quelques fonctions en plus... Ces fonctions sont les suivantes : `valeurs_possibles_attribut()` et `sous_ensemble_attribut()`.

La fonction valeurs_possibles_attribut() doit renvoyer une liste contenant toutes les valeurs que peut prendre l'attribut passé en paramètre. Voici son implémentation :

```
def valeurs_possibles_attribut(self, nom_attribut):
    """
        retourne une liste contenant toutes les
        valeurs possibles de l'attribut
    """
    ret = list()
    #pour chaque exemple
    for exemple in self.liste_exemples:
        #si cette valeur n'est pas encore dans la liste
        if not exemple.dict_attributs[nom_attribut] in ret:
            #on l'ajoute
            ret.append(exemple.dict_attributs[nom_attribut])
    #et on retourne la liste
    return ret
```

Et sous_ensemble_attribut() doit renvoyer un sous-ensemble contenant uniquement une certaine valeur pour l'attribut. La voici :

```
def sous_ensemble_attribut(self, nom_attribut, valeur):
    """
        retourne un sous-ensemble contenant uniquement les exemples ayant
```

```
        la bonne valeur pour l'attribut
    """
    ret = Ensemble()
    #on prend tous les attributs sauf celui passé en paramètre
    ret.liste_attributs = self.liste_attributs[:]
    ret.liste_attributs.remove(nom_attribut)
    #pour chaque exemple de l'ensemble
    for exemple in self.liste_exemples:
        #s'il a la bonne valeur
        if exemple.dict_attributs[nom_attribut] == valeur:
            #on l'ajoute
            ret.liste_exemples.append(exemple)
    #et on retourne la liste
    return ret
```

Il nous reste une petite partie de la fonction de construction à faire, on doit faire la création du nœud à retourner. Et pour ce faire, il nous faut créer la classe Noeud ! La voici :

```
class Noeud:
    """
        Un noeud a deux valeurs :
            - un dictionnaire d'enfants (dict)
            - l'attribut testé (str)
    """

    def __init__(self, attribut):
        """
            attribut_teste est le nom de l'attribut stocké dans un str
        """
        if not isinstance(attribut, str):
            raise
                TypeError("attribute_teste doit être un str et pas un {}"
                \
                          .format(type(attribut)))
        #initialisation des valeurs de l'objet
        self.enfants = dict()
        self.attribut_teste = attribut
```

La suite et fin de notre fonction est donc la suivante :

```
#si on arrive ici, on retourne d'office un nœud et pas une feuille
noeud = Noeud(a_tester)
#pour chaque valeur que peut prendre l'attribut à tester
for valeur in ensemble.valeurs_possibles_attribut(a_tester):
    #on crée un sous-ensemble
```

```
    sous_ensemble = ensemble.sous_ensemble_attribut(a_tester,
        valeur)
    #et on en crée un nouveau nœud
    noeud.enfants[valeur] = self.__construire_arbre(sous_ensemble)
#on retourne le nœud que l'on vient de créer
return noeud
```

Voici d'ailleurs le code complet de la fonction :

```
def __construire_arbre(self, ensemble):
    """
        fonction privée et récursive pour la génération de l'arbre
    """
    if not isinstance(ensemble, Ensemble):
        raise
            TypeError("ensemble doit être un Ensemble et non un {}"
            \
                        .format(type(ensemble)))
    #si la liste est vide
    if len(ensemble) == 0:
        raise ValueError("la liste d'exemples ne peut être vide !")
    #testons si tous les exemples ont la même étiquette
    if ensemble.entropie() == 0:
        #on retourne l'étiquette en question
        return Feuille(ensemble.liste_exemples[0].etiquette)
    #s'il ne reste d'attribut à tester
    if len(ensemble.liste_attributs) == 0:
        max, etiquette_finale = 0, ""
        #on teste toutes les étiquettes possibles de l'ensemble
        for etiquette in ensemble.etiquettes_possibles():
            sous_ensemble =
                ensemble.sous_ensemble_etiquette(etiquette)
            #si c'est la plus fréquente, c'est celle qu'on choisit
            if len(sous_ensemble) > max:
                max, etiquette_finale = len(sous_ensemble),
                    etiquette
        #et on la retourne dans une feuille
        return Feuille(etiquette_finale)

    a_tester = ensemble.attribut_optimal()
    #si on arrive ici, on retourne d'office un nœud et pas une
        feuille
    noeud = Noeud(a_tester)
    #pour chaque valeur que peut prendre l'attribut à tester
    for valeur in ensemble.valeurs_possibles_attribut(a_tester):
        #on crée un sous-ensemble
        sous_ensemble = ensemble.sous_ensemble_attribut(a_tester,
            valeur)
```

```
34          #et on en crée un nouveau nœud
35          noeud.enfants[valeur] =
                self.__construire_arbre(sous_ensemble)
36      #on retourne le nœud que l'on vient de créer
37      return noeud
```

?

Maintenant, nous avons pu créer un arbre... Mais comment on sait si tout a bien fonctionné ?

C'est une excellente question ! C'est simple, on va l'afficher

On peut donc :

— soit surcharger la fonction `__str__` pour pouvoir l'utiliser dans un print ;
— soit créer une nouvelle fonction pour pouvoir faire un affichage standard.

C'est personnellement la deuxième solution que je préfère car notre affichage est un peu spécial et tient sur plusieurs lignes. Je ne vois donc pas réellement comment dans quelle circonstance notre classe serait affichée dans un print si ce n'est quand elle est toute seule. Et c'est donc celle que je vais implémenter. Ceci-dit, libre à vous de faire l'autre, voire de faire les deux !

#####5.3. L'affichage de l'arbre

Je vous propose donc de faire une méthode de la sorte :

```
1  def afficher(self):
2      """
3          affiche l'entièreté de l'arbre à l'écran
4      """
5      self.__afficher_arbre(self.arbre)
```

Effectivement, je propose que l'on fasse comme pour la construction, une fonction privée. D'ailleurs, la voici. Comme vous pouvez le voir, j'ai utilisé une certaine convention de notation qui ne tient qu'à moi mais je la trouvais claire et jolie donc je l'ai gardée !

```
1  def __afficher_arbre(self, noeud, nb_tabs=0):
2      """
3          selon la convention :
4              <texte> <-> nom de l'attribut
5              -<texte> <-> valeur de l'attribut
6              .<texte> <-> feuille
7      """
8      #si on a affaire à un nœud
9      if isinstance(noeud, Noeud):
10         #on affiche le nom de l'attribut testé
11         print('\t' * nb_tabs + noeud.attribut_teste)
12         #on parcourt ses enfants
```

```
13          for enfant in noeud.enfants:
14              #on affiche la valeur de l'attribut
15              print('\t' * nb_tabs + '-' + str(enfant))
16              self.__afficher_arbre(noeud.enfants[enfant], nb_tabs+1)
17      #si c'est une feuille
18      elif isinstance(noeud, Feuille):
19          #on affiche l'étiquette
20          print('\t' * nb_tabs + '.' + noeud.etiquette)
21      else:
22          raise
                TypeError("noeud doit être un Noeud/Feuille et pas un {}"
                \
23                          .format(type(noeud)))
```

Nous avons donc maintenant fini l'implémentation de l'algorithme ID3 !

Sauf que comme ça, il ne nous sert à rien ! L'intérêt de ces arbres de décisions, c'est de pouvoir étiqueter des exemples externes. Il nous faut donc une méthode pour étiqueter un nouvel exemple. Cette méthode fait donc bien entendu partie de la classe Arbre_ID3. Voici ce que je vous propose :

```
1  def etiqueter(self, exemple):
2      """
3          assigne la bonne étiquette à l'exemple passé en paramètre
4      """
5      #on initialise le nœud actuel avec le haut de l'arbre
6      noeud_actuel = self.arbre
7      #tant que l'on est sur un nœud et pas sur une feuille,
8      #on continue d'explorer
9      while not isinstance(noeud_actuel, Feuille):
10         #pour savoir quel est le prochain nœud, on récupère d'abord
11         #l'attribut testé avec noeud_actuel.attribut_teste puis on
               récupère
12         #la valeur de l'exemple pour cet attribut avec
13         #exemple.dict_attributs[noeud_actuel.attribut_teste]
14         #puis on prend l'enfant de noeud_actuel ayant cette valeur.
15         valeur =
               exemple.dict_attributs[noeud_actuel.attribut_teste]
16         noeud_actuel = noeud_actuel.enfants[valeur]
17     #on finit en donnant comme étiquette l'étiquette
18     #contenue dans la feuille finale
19     exemple.etiquette = noeud_actuel.etiquette
```

Je vous laisse améliorer le code si bon vous semble. Je vous le donne en entier au cas où certains auraient mal copié ou se seraient trompés ou n'auraient pas réussi à tout comprendre.

Contenu masqué n°1

Je vous donne également l'exemple utilisé dans ce cours pour que vous puissiez voir à quel point votre code est magnifique !

Le voici :

```
Prévisions Température Humidité Vent
Ensoleillé Chaud Élevée Faible Non
Ensoleillé Chaud Élevée Fort Non
Nuageux Chaud Élevée Faible Oui
Pluvieux Moyen Élevée Faible Oui
Pluvieux Frais Normale Faible Oui
Pluvieux Frais Normale Fort Non
Nuageux Frais Normale Fort Oui
Ensoleillé Moyen Élevée Faible Non
Ensoleillé Frais Normale Faible Oui
Pluvieux Moyen Normale Faible Oui
Ensoleillé Moyen Normale Fort Oui
Nuageux Moyen Élevée Fort Oui
Nuageux Chaud Normale Faible Oui
Pluvieux Moyen Élevée Fort Non
```

PS : il se peut qu'il y ait des problèmes d'affichage d'accents avec les différents terminaux et invites de commande. Donc soit vous vous débarassez froidement des accents, soit vous configurez votre terminal, soit vous utilisez l'exemple en anglais que voici :

```
Outlook Temperature Humidity Wind
Sunny Hot High Weak No
Sunny Hot High Strong No
Overcast Hot High Weak Yes
Rain Mild High Weak Yes
Rain Cool Normal Weak Yes
Rain Cool Normal Strong No
Overcast Cool Normal Strong Yes
Sunny Mild High Weak No
Sunny Cool Normal Weak Yes
Rain Mild Normal Weak Yes
Sunny Mild Normal Strong Yes
Overcast Mild High Strong Yes
Overcast Hot Normal Weak Yes
Rain Mild High Strong No
```

---

6. on parle de *built-in type*

Et voilà, nous en avons fini pour l'algorithme ID3 et son implémentation. On souffle un bon coup, et puis on se lance sur son successeur : C4.5 !

Contenu masqué

Contenu masqué n°1

```
"""
    écrit par poupou9779 pour Zeste de Savoir
"""

from math import log

class Feuille:
    """
        Une feuille contient uniquement une valeur:
            - l'étiquette (str)
    """

    def __init__(self, etiquette):
        """
            etiquette doit obligatoirement être un str
        """
        if not isinstance(etiquette, str):
            raise
                TypeError("etiquette doit être un str et pas un {}"
                \
                                .format(type(etiquette)))
        self.etiquette = etiquette

class Noeud:
    """
        Un noeud a deux valeurs :
            - un dictionnaire d'enfants (dict)
            - l'attribut testé (str)
    """

    def __init__(self, attribut):
        """
            attribut_teste est le nom de l'attribut stocké dans un str
        """
        if not isinstance(attribut, str):
            raise
                TypeError("attribute_teste doit être un str et pas un {}"
                \
                                .format(type(attribut)))
```

```
36          #initialisation des valeurs de l'objet
37          self.enfants = dict()
38          self.attribut_teste = attribut
39
40  class Exemple:
41      """
42          Un exemple contient 2 valeurs :
43              - un dictionnaire d'attributs (dict)
44              - une étiquette (str)
45      """
46
47      def __init__(self, noms_attributs, valeurs_attributs,
            etiquette=""):
48          """
49              etiquette peut être non précisée en quel cas on aurait
50              un exemple non étiqueté
51          """
52          #si on a un problème de types
53          if not isinstance(noms_attributs, list) \
54          or not isinstance(valeurs_attributs, list):
55              raise
                    TypeError("noms_attributs et valeurs_attributs doivent être"
                    \
56                              " des listes et pas des {0} et {1}" \
57                              .format(type(noms_attributs),
58                                      type(valeurs_attributs)))
59          if not isinstance(etiquette, str):
60              raise
                    TypeError("etiquette doit être un str et pas un {}"
                    \
61                              .format(type(etiquette)))
62          #si les deux listes n'ont pas le même nombre d'éléments
63          if len(valeurs_attributs) != len(noms_attributs):
64              raise
                    ValueError("noms_attributs et valeurs_attributs doivent "
                    \
65                               "avoir le même nombre d'éléments")
66          self.etiquette = etiquette
67          self.dict_attributs = dict()
68          #on ajoute chaque attribut au dictionnaire
69          for i in range(len(noms_attributs)):
70              self.dict_attributs[noms_attributs[i]] =
                    valeurs_attributs[i]
71
72  class Ensemble:
73      """
74          Un ensemble contient deux valeurs :
75              - les noms des attributs (list)
76              - les exemples (list)
77      """
```

```
78
79      def __init__(self, chemin=""):
80          """
81              chemin est l'emplacement du fichier contenant les données.
82              Cette variable peut être non précisée en quel cas les variables
83                  seront initialisées comme des listes vides.
84          """
85          #Python est un langage à typage dynamique fort,
86          #il faut donc vérifier que l'utilisateur ne fait pas
                n'importe quoi
87          #en passant autre chose qu'un str
88          if not isinstance(chemin, str):
89              raise TypeError("chemin doit être un str et non {}" \
90                              .format(type(chemin)))
91          if chemin == "":
92              #initialisation en listes vides
93              self.liste_attributs = list()
94              self.liste_exemples = list()
95          else:
96              with open(chemin, 'r') as fichier:
97                  #on stocke chaque mot de la première ligne dans
                        liste_attributs
98                  self.liste_attributs = \
99
                                        fichier.readline().lower().strip().split('
100                 #ensuite on stocke la liste d'exemples dans
                        liste_exemples
101                 self.liste_exemples = self.liste_en_exemples(
102
                                                fichier.read().strip().lower().spli
103                                       self.liste_attributs
104                                     )
105
106     def __len__(self):
107         """
108             retourne la longueur de l'ensemble
109         """
110         return len(self.liste_exemples)
111
112     @staticmethod
113     def liste_en_exemples(exemples, noms_attributs):
114         """
115             retourne une liste d'exemples sur base d'une liste de str contenant
116             les valeurs et d'une liste de str contenant les noms des attributs
117         """
118         #on initialise la liste à retourner
119         ret = list()
120         for ligne in exemples:
121             #on stocke chaque mot de la ligne dans une liste
                    attributs
```

```
122             attributs = ligne.lower().strip().split(' ')
123             #met l'étiquette par défaut si elle n'est pas dans la
                    ligne
124             etiquette = attributs[-1] if len(attributs) !=
                    len(noms_attributs) \
125                                       else ""
126             #on ajoute un objet de type Exemple contenant la ligne
127             ret.append(Exemple(noms_attributs,
128                                attributs[:len(noms_attributs)],
129                                etiquette))
130         return ret
131
132     def etiquettes_possibles(self):
133         """
134             retourne une liste contenant les étiquettes de l'ensemble
135         """
136         #on initialise la valeur de retour
137         ret = list()
138         #pour chaque exemple de l'ensemble
139         for exemple in self.liste_exemples:
140             #si l'étiquette n'est pas déjà dans la liste
141             if not exemple.etiquette in ret:
142                 #on l'ajoute
143                 ret.append(exemple.etiquette)
144         return ret
145
146     def sous_ensemble_etiquette(self, etiquette):
147         """
148             retourne un ensemble contenant uniquement les exemples ayant
149             etiquette comme étiquette
150         """
151         #initialisation de la valeur de retour
152         ret = Ensemble()
153         #on copie la liste d'attributs
154         ret.liste_attributs = self.liste_attributs[:]
155         #pour chaque exemple de l'ensemble initial
156         for exemple in self.liste_exemples:
157             #si l'étiquette est bonne
158             if exemple.etiquette == etiquette:
159                 #on l'ajoute au sous-ensemble
160                 ret.liste_exemples.append(exemple)
161         return ret
162
163     def sous_ensemble_attribut(self, nom_attribut, valeur):
164         """
165             retourne un sous-ensemble contenant uniquement les exemples ayant
166             la bonne valeur pour l'attribut
167         """
168         ret = Ensemble()
169         #on prend tous les attributs sauf celui passé en paramètre
```

```
170         ret.liste_attributs = self.liste_attributs[:]
171         ret.liste_attributs.remove(nom_attribut)
172         #pour chaque exemple de l'ensemble
173         for exemple in self.liste_exemples:
174             #s'il a la bonne valeur
175             if exemple.dict_attributs[nom_attribut] == valeur:
176                 #on l'ajoute
177                 ret.liste_exemples.append(exemple)
178         #et on retourne la liste
179         return ret
180 
181     def entropie(self):
182         """
183             retourne l'entropie de Shannon de l'ensemble
184         """
185         #initialisation de la variable retournée
186         ret = 0
187         #pour chaque étiquette de l'ensemble
188         for etiquette in self.etiquettes_possibles():
189             #on crée un sous-ensemble qui ne contient que les
                    éléments de
190             #self ayant etiquette comme étiquette
191             sous_ensemble = self.sous_ensemble_etiquette(etiquette)
192             #on ajoute |c| * log_2(|c|) à ret
193             longueur_sous_ensemble = len(sous_ensemble)
194             ret += longueur_sous_ensemble *
                    log(longueur_sous_ensemble, 2)
195         #on retourne log_2(|S|) - ret/|S|
196         return log(len(self), 2) - ret/len(self)
197 
198     def attribut_optimal(self):
199         """
200             retourne un str avec le nom de l'attribut à tester
201         """
202         max, ret = float("-inf"), ""
203         #pour chaque attribut
204         for attribut in self.liste_attributs:
205             gain = self.gain_entropie(attribut)
206             #si le gain d'entropie est le plus grand
207             if gain >= max:
208                 #on le garde en mémoire
209                 max, ret = gain, attribut
210         #et on le retourne
211         return ret
212 
213     def valeurs_possibles_attribut(self, nom_attribut):
214         """
215             retourne une liste contenant toutes les
216             valeurs possibles de l'attribut
217         """
```

```
218         ret = list()
219         #pour chaque exemple
220         for exemple in self.liste_exemples:
221             #si cette valeur n'est pas encore dans la liste
222             if not exemple.dict_attributs[nom_attribut] in ret:
223                 #on l'ajoute
224                 ret.append(exemple.dict_attributs[nom_attribut])
225         #et on retourne la liste
226         return ret
227 
228     def gain_entropie(self, nom_attribut):
229         """
230             retourne la perte d'entropie selon la définition de Ross Quinlan
231         """
232         somme = 0
233         #pour chaque valeur de l'attribut en question
234         for valeur in self.valeurs_possibles_attribut(nom_attribut):
235             #déclaration de Sv
236             sous_ensemble =
                    self.sous_ensemble_attribut(nom_attribut, valeur)
237             #somme = somme sur v de |Sv| * Entropie(Sv)
238             somme += len(sous_ensemble) * sous_ensemble.entropie()
239         #Gain(S, A) = Entropie(S) - 1/|S| * somme
240         return self.entropie() - somme/len(self)
241 
242 
243 class Arbre_ID3:
244     """
245         Un arbre ID3 contient deux valeurs :
246             - un ensemble d'exemples (Ensemble)
247             - un arbre (Noeud)
248     """
249 
250     def __init__(self, chemin=""):
251         """
252             chemin est l'emplacement du fichier contenant les données
253         """
254         #initialisation de l'ensemble avec le fichier dans chemin
255         self.ensemble = Ensemble(chemin)
256         #initialisation du noeud principal de l'arbre
257         self.arbre = None
258 
259     def construire(self):
260         """
261             génère l'arbre sur base de l'ensemble pré-chargé
262         """
263         self.arbre = self.__construire_arbre(self.ensemble)
264 
265     def __construire_arbre(self, ensemble):
266         """
```

```
267             fonction privée et récursive pour la génération de l'arbre
268         """
269         if not isinstance(ensemble, Ensemble):
270             raise
                TypeError("ensemble doit être un Ensemble et non un {}"
                \
271                           .format(type(ensemble)))
272         #si la liste est vide
273         if len(ensemble) == 0:
274             raise
                ValueError("la liste d'exemples ne peut être vide !")
275         #testons si tous les exemples ont la même étiquette
276         if ensemble.entropie() == 0:
277             #on retourne l'étiquette en question
278             return Feuille(ensemble.liste_exemples[0].etiquette)
279         #s'il ne reste d'attribut à tester
280         if len(ensemble.liste_attributs) == 0:
281             max, etiquette_finale = 0, ""
282             #on teste toutes les étiquettes possibles de l'ensemble
283             for etiquette in ensemble.etiquettes_possibles():
284                 sous_ensemble =
                        ensemble.sous_ensemble_etiquette(etiquette)
285                 #si c'est la plus fréquente, c'est celle qu'on
                    choisit
286                 if len(sous_ensemble) > max:
287                     max, etiquette_finale = len(sous_ensemble),
                            etiquette
288             #et on la retourne dans une feuille
289             return Feuille(etiquette_finale)
290
291         a_tester = ensemble.attribut_optimal()
292         #si on arrive ici, on retourne d'office un nœud et pas une
                feuille
293         noeud = Noeud(a_tester)
294         #pour chaque valeur que peut prendre l'attribut à tester
295         for valeur in ensemble.valeurs_possibles_attribut(a_tester):
296             #on crée un sous-ensemble
297             sous_ensemble =
                    ensemble.sous_ensemble_attribut(a_tester, valeur)
298             #et on en crée un nouveau nœud
299             noeud.enfants[valeur] =
                    self.__construire_arbre(sous_ensemble)
300         #on retourne le nœud que l'on vient de créer
301         return noeud
302
303     def afficher(self):
304         """
305             affiche l'entièreté de l'arbre à l'écran
306         """
307         self.__afficher_arbre(self.arbre)
```

```
308
309     def __afficher_arbre(self, noeud, nb_tabs=0):
310         """
311             selon la convention :
312                 <texte> <-> nom de l'attribut
313                 -<texte> <-> valeur de l'attribut
314                 .<texte> <-> feuille
315         """
316         #si on a affaire à un nœud
317         if isinstance(noeud, Noeud):
318             #on affiche le nom de l'attribut testé
319             print('\t' * nb_tabs + noeud.attribut_teste)
320             #on parcourt ses enfants
321             for enfant in noeud.enfants:
322                 #on affiche la valeur de l'attribut
323                 print('\t' * nb_tabs + '-' + str(enfant))
324                 self.__afficher_arbre(noeud.enfants[enfant],
                        nb_tabs+1)
325         #si c'est une feuille
326         elif isinstance(noeud, Feuille):
327             #on affiche l'étiquette
328             print('\t' * nb_tabs + '.' + noeud.etiquette)
329         else:
330             raise
                    TypeError("noeud doit être un Noeud/Feuille et pas un {}"
                    \
331                               .format(type(noeud)))
332
333     def etiqueter(self, exemple):
334         """
335             assigne la bonne étiquette à l'exemple passé en paramètre
336         """
337         #on initialise le nœud actuel avec le haut de l'arbre
338         noeud_actuel = self.arbre
339         #tant que l'on est sur un nœud et pas sur une feuille,
340         #on continue d'explorer
341         while not isinstance(noeud_actuel, Feuille):
342             #pour savoir quel est le prochain nœud, on récupère
                    d'abord
343             #l'attribut testé avec noeud_actuel.attribut_teste puis
                    on récupère
344             #la valeur de l'exemple pour cet attribut avec
345             #exemple.dict_attributs[noeud_actuel.attribut_teste]
346             #puis on prend l'enfant de noeud_actuel ayant cette
                    valeur.
347             valeur =
                    exemple.dict_attributs[noeud_actuel.attribut_teste]
348             noeud_actuel = noeud_actuel.enfants[valeur]
349         #on finit en donnant comme étiquette l'étiquette
350         #contenue dans la feuille finale
```

```
351         exemple.etiquette = noeud_actuel.etiquette
352
353 def exemple_utilisation():
354     #exemple d'utilisation
355     arbre = Arbre_ID3('datas PlayTennis.txt')
356     arbre.construire()
357     arbre.afficher()
358     # il faut mettre les mots dans la même langue que l'ensemble
            choisi
359     exemple = Exemple(["outlook", "temperature", "humidity",
            "wind"],
360                       ["sunny", "29.5", "normal", "strong"])
361     print("etiquette : '{}'".format(exemple.etiquette))
362     arbre.etiqueter(exemple)
363     print("etiquette : '{}'".format(exemple.etiquette))
364
365 if __name__ == "__main__":
366     exemple_utilisation()
```

Retourner au texte.

3. Une amélioration : C 4.5

L'algorithme C4.5 est une évolution de l'algorithme ID3. Donc j'espère que vous avez bien suivi la partie précédente, car ici, nous allons découvrir C4.5 et à nouveau l'implémenter.

3.1. L'algorithme C 4.5

L'algorithme C4.5 a également été inventé par Ross Quinlan, il en a fait un livre édité en **1993**. C'est un algorithme qui est basé sur ID3 mais qui a quelques éléments en plus :

— une adaptation de la fonction de gain qui n'a plus tendance à aller vers l'attribut avec le plus de valeurs possibles ;
— la possibilité de gérer des attributs avec des valeurs manquantes ;
— la possibilité de post-élaguer son arbre pour éviter l'"*overfitting*" ;
— la possibilité de manipuler des valeurs continues (en les "discrétisant" lors de la mise en arbre).

? Discrétiser ?

Exactement. Le principe est de pouvoir manipuler **toutes** les valeurs réelles. Or en suivant l'algo ID3, on devrait alors avoir un nombre **infini** de branches à chaque nœud. Le principe est donc de remplacer notre infinité de nombres réels par un nombre fini d'intervalles. Mais gardons cela pour la fin.

Toutes les améliorations vont être expliquées l'une après l'autre, et bien entendu, pour continuer sur notre lancée, elles seront implémentées toujours en Python3 afin d'enrichir le code du chapitre précédent. Pour ce faire, nous allons avoir besoin d'une nouvelle classe : la classe `Arbre_C45` qui va ressembler très fort à la classe `Arbre_ID3`. Elle va lui ressembler tellement fort qu'en réalité, pour se simplifier la vie (du moins dans un premier temps), nous allons faire descendre notre nouvelle classe de cette ancienne de la sorte :

```
1 class Arbre_C45(Arbre_ID3):
2     """
3         cette classe découle directement (hérite) de la classe Arbre_ID3
4     """
```

Les améliorations se feront sur les données que nous avons codées jusqu'à présent également.

###1. Adaptation de la fonction de gain Commençons par la nouvelle fonction de gain. Cette fonction s'appelle ratio de gain et est définie de la sorte :

$$\text{Ratio de gain}(S, A) = \frac{\text{Gain}(S, A)}{\text{Split d'Entropie}(S, A)}$$

tel que

$$\text{Split d'Entropie}(S, A) = -\sum_{v \in \text{valeurs}(A)} \frac{|S_v|}{|S|} \times \log_2\left(\frac{|S_v|}{|S|}\right)$$

Cette fonction s'utilise de la même manière que la fonction $\text{Gain}(S, A)$ de l'algorithme ID3.

En python (en lien avec le code du chapitre précédent), cette fonction pourrait ressembler à ceci :

```
def ratio_gain(self, nom_attribut):
    """
        retourne le ratio de gain (C4.5) de l'ensemble
    """
    split = self.split_entropie(nom_attribut)
    gain = self.gain_entropie(nom_attribut)
    return gain/split if split != 0 else float("inf")
```

Attention, n'oubliez pas de gérer une éventuelle division par zéro !

La fonction ici est très simple car la fonction de gain d'entropie est déjà faite, il ne nous reste plus que la fonction de split d'entropie, mais rassurez-vous elle n'est en rien compliquée !

```
def split_entropie(self, nom_attribut):
    """
        retourne le split d'entropie du set selon l'attribut en question
    """
    ret = 0
    for valeur in self.valeurs_possibles_attribut(nom_attribut):
        sous_ensemble = self.sous_ensemble_attribut(nom_attribut,
            valeur)
        ret += len(sous_ensemble) * log(len(sous_ensemble), 2)
    return log(len(self), 2) - ret/len(self)
```

Faire ceci implique également un changement de la fonction `attribut optimal` : utiliser la fonction `ratio_gain`. Voici donc ce qu'est devenue ma fonction :

```
def attribut_optimal(self, ID3=True):
    """
```

```
3           retourne un str avec le nom de l'attribut à tester
4       """
5       max, ret = float("-inf"), ""
6       #pour chaque attribut
7       for attribut in self.liste_attributs:
8           if ID3:
9               gain = self.gain_entropie(attribut)
10          else:
11              gain = self.ratio_gain(attribut)
12          #si le gain d'entropie est le plus grande
13          if gain >= max:
14              #on le garde en mémoire
15              max, ret = gain, attribut
16      #et on le retourne
17      return ret
```

J'ai choisi d'utiliser un booléen pour savoir quelle méthode utiliser. Il vaut True par défaut comme ça il ne nous est pas nécessaire de retoucher à notre classe `Arbre_ID3`. Mais la valeur par défaut n'est pas du tout obligatoire.

###2. Manipulation de valeurs continues Penchons-nous ensuite sur la discrétisation des valeurs. Pour ce faire, reprenons notre tableau du chapitre 2.1.2 mais rendons la valeur de **Humidité** continue et non plus discrète[??]. On va la représenter par un pourcentage.

Voici notre tableau :

Jour	**Attributs des exemples**				**Classe**
	Prévisions	Température	Humidité	Vent	
1	**Ensoleillé**	**Chaud**	**85**	**Faible**	**Non**
2	Ensoleillé	Chaud	90	Fort	Non
3	Nuageux	Chaud	78	Faible	Oui
4	Pluvieux	Moyen	96	Faible	Oui
5	Pluvieux	Frais	80	Faible	Oui
6	Pluvieux	Frais	70	Fort	Non
7	Nuageux	Frais	65	Fort	Oui
8	Ensoleillé	Moyen	95	Faible	Non
9	Ensoleillé	Frais	70	Faible	Oui
10	Pluvieux	Moyen	80	Faible	Oui
11	Ensoleillé	Moyen	70	Fort	Oui

12	Nuageux	Moyen	90	Fort	Oui
13	Nuageux	Chaud	75	Faible	Oui
14	Pluvieux	Moyen	80	Fort	Non

Table : Ensemble d'exemples adapté aux données continues

Si vous désirez calculer le gain de cet attribut, imaginons que vous n'avez qu'une seule fois chaque valeur. Le gain serait :

$$\text{Gain}(S, \text{Humidité}) = |S| \times \frac{1}{|S|} \times 0 = 0$$

Effectivement, le gain vaut **0** vu que si vous n'avez qu'une seule fois chaque valeur, vous aurez un nœud avec $|S|$ fils (ici **14**) et chacun donnera un résultat différent. Mais ce n'est pas ce qui nous intéresse parce que la donnée étant continue, si on tente d'étiqueter un exemple ayant **61.2** comme température, l'arbre se retrouve inefficace car on ne sait pas avancer vu qu'aucune branche du nœud en question n'a cette valeur. Pour une valeur continue, les branches ne doivent pas représenter une valeur précise, mais un ensemble délimité par une borne inférieure et une borne supérieure.

Tentons de discrétiser l'attribut **Humidité** pour cet ensemble d'exemples. Il faut commencer par trier les exemples dans l'ordre croissant (ou décroissant, ça n'a pas d'importance).

Humidité	**Classe**
65	Oui
70	Non
70	Oui
70	Oui
75	Oui
78	Oui
80	Oui
80	Oui
80	Non
85	Non
90	Non
90	Oui
95	Non
96	Oui

TABLE 3.3. – Valeurs de **Humidité** isolées avec l'étiquette correspondante

J'ai uniquement gardé les éléments importants pour la compréhension de la discrétisation, à savoir l'attribut **Humidité** et l'étiquette (ou la classe si vous préférez). Il y aura autant de valeurs différentes pour l'attribut **Humidité** qu'il y a de changements de classe. Je m'explique : on commence au premier exemple : **(65, Oui)**, si on regarde le second exemple, on a **(70, Non)**. On a donc eu un changement de classification entre **Humidité = 65** et **Humidité = 70**. On fait donc la moyenne et on a une première valeur pour :

$$\text{Humidité} =]-\infty, 67.5[$$

Continuons : nous en sommes à **(70, Non)**. Si on lit le suivant, on obtient **(70, Oui)**. Étant donné qu'il y a à nouveau changement de classification, on crée un nouvel élément à nouveau en faisant la moyenne. On a donc :

$$\text{Humidité} =]-\infty, 67.5[\cup [67.5\, 70[$$

Vous continuez ce procédé jusqu'au bout et vous devriez obtenir :

$$\text{Humidité} =]-\infty, 67.5[\cup [67.5\, 70[\cup [70, 80[\cup [80, 90[\cup$$
$$[90, 92.5[\cup [92.5, 95.5[\cup [95.5, +\infty[$$

Il y a donc 7 classes. Et de plus nous avons des étiquettes différentes alors que la valeur de **Humidité** est la même. Ça ne fait rien, on traite ça normalement, de toute façon, la moyenne arithmétique entre **80** et **80**, c'est encore plus simple à calculer, !a fait **80** !

Voici le pseudo code qui vous permet de faire cette discrétisation :

DISCRÉTISATION : Entrée : - S = le set d'exemples étiquetés - A = attribut à discrétiser Retour : - rien DÉBUT trier S en fonction de A p = -∞ c = classe du premier élément de S POUR CHAQUE e = exemple de S SI la classe de e est différente de c ajouter comme valeur à A [p, (A de (e) + (A de (e-1))/2] c = classe de e p = e FIN SI FIN POUR CHAQUE ajouter comme valeur à A [p, +∞[FIN

Ce pseudo-code permet donc de changer toutes les valeurs continues en valeurs discrètes (des intervalles).

[[a]] |Cependant, attention ! Il ne faut pas se contenter de faire la discrétisation une fois au début et puis tout garder comme ça pour la simple et bonne raison que les ensembles vont diminuer petit à petit et que donc les intervalles n'auront plus beaucoup de sens. Il faut donc discrétiser à chaque fois juste avant de calculer le ratio de gain. Et si l'attribut n'est pas choisi, il faut garder les valeurs continues afin de pouvoir les re-discrétiser de manière plus efficace par la suite (une fois l'ensemble d'exemples raccourci).

Maintenant, comment allons-nous faire pour discrétiser nos valeurs ? Nous allons déjà devoir changer la fonction pour générer l'arbre. Donc je vous invite à prendre le code de la génération de l'arbre ID3 (à savoir les méthodes 'construire' et '$_{construire_arbre}$`):

"'python class Arbre$_C$45($Arbre_ID3$) : """$UnarbreC4.5hrited'unarbreID3maisseconstruitdiffremment$"""

def construire(self) : """ génère l'arbre sur base de l'ensemble pré-chargé """ self.arbre = self.$_{construire arbre(self.ensemble)}$

def $_{construire arbre(self,ensemble):"""fonctionpriveetrcursivepouragnrationdel'arbre"""ifnotisinstance(ensemble,Ensemble):raiseTypeErro}$

a$_t ester = ensemble.attribut_o ptimal(ID3 = False)sionarriveici, onretourned'officeunnudetpasunefeuil$
$Noeud(a_t ester)pourchaquevaleurquepeutprendrel'attributtesterforvaleurinensemble.valeurs_p ossibles_a t$
$oncreunsous-ensemblesous_e nsemble = ensemble.sous_e nsemble_a ttribut(a_t ester, valeur)etonencreunnou$
$self._{construire arbre(sous_e nsemble)onretournelenudquel'onvientdecrerreturnnoeud}$```

[[a]] |N'oubliez pas de préciser le booléen 'ID=False' lors de l'appel à 'attribut$_o ptimal$` $sinonvousn'utiliserez p$

Sauf que nous allons devoir rajouter des choses : nous devons discrétiser les attributs qui doivent l'être avant de choisir l'attribut optimal, et nous devons restituer les valeurs des attributs discrétisés n'ayant pas été choisis.

"'python hl$_l ines$ = "2729"$def_{construire arbre(self,ensemble):"""fonctionpriveetrcursivepouragnrationdel'arbre"""ifnotisinstance(ens}$

ICI : discrétisation a$_t ester = ensemble.attribut_o ptimal(ID3 = False)ICI : restitutiondesvaleurs$

si on arrive ici, on retourne d'office un nœud et pas une feuille noeud = Noeud(a$_t ester)pourchaquevaleurquep$
$oncreunsous-ensemblesous_e nsemble = ensemble.sous_e nsemble_a ttribut(a_t ester, valeur)etonencreunnou$
$self._{construire arbre(sous_e nsemble)onretournelenudquel'onvientdecrerreturnnoeud}$```

Dans l'ordre, que devons-nous faire ? Nous devons :

+ sauvegarder les valeurs continues ; + les discrétiser ; + déterminer l'attribut optimal ; + restituer les valeurs non-utilisées.

Commençons donc par les sauvegarder. C'est très simple, je vous propose ceci (qui donc se place juste avant l'appel à 'attribut$_o ptimal$`) :

"'python sauvegarde$_v aleurs = ensemble.sauvegarder_v aleurs_d iscretes()$```

Ça déborde de simplicité si ce n'est le fait qu'il faille coder la fonction 'sauvegarder$_v aleurs_d iscretes$`.$Oui, jes a$
$moiconfiance.M$

Cette fonction, que doit-elle faire ? Elle doit regarder chaque attribut, regarder s'il est discrétisable, d'il l'est, le discrétiser. Dit comme ça, ça n'a rien de compliqué. Voyez par vous même :

"'python def sauvegarder$_v aleurs_d iscretes(self)$: """$renvoieunelistedetuplescontenant : +lenomdel'attribu$
$lalistedetouteslesvaleursdecetattribut"""ret = list()pourchaqueattributforattributinself.liste_a ttributs :$
$s'ilestdiscrtisableifself.est_d iscretisable(attribut) : onsauvegardelesvaleursdechaqueexemplevaleurs_a ttr$
$[exemple.dict_a ttributs[attribut]\ forexemple inself.liste_e xemples]ret.append((attribut, valeurs_a ttribut))re$

Cette méthode fait partie de la classe 'Ensemble' et non pas de la classe 'Arbre$_C 45$`, $nevousmlangezpaslespin$
$lenomdel'attributdontilestquestionetlalistedetouteslesvaleurs.Lecodedelafonction`est_d iscretisable`est*$
$*trs**simple, levoici$:

"'python def est$_d iscretisable(self, nom_a ttribut)$: """$renvoieTruesil'attributenquestionestunattributcontin$
$silavaleurdel'attributn'estpasunnombretry : float(exemple.dict_a ttributs[nom_a ttribut])alorsl'attributn'es$
$returnFalsesitouslesexemplessontdiscrtisablespourcetattribut, l'attributestdiscrtisablereturnTrue$```

On tente une conversion en 'float', si tout marche bien, c'est que l'attribut est discrétisable. Par contre, s'il y en a ne fut-ce qu'un seul qui ne veut pas, c'est que l'attribut n'est pas discrétisable et qu'il faut donc le considérer comme un attribut discret.

Jusqu'à présent, nous n'avons fait que la partie où on sauvegarde les valeurs avant de les discrétiser. Donc le plus gros du boulot reste encore à faire ! Il nous faut maintenant faire la discrétisation à proprement parler. Il faut donc que nous changions chaque attribut discrétisable en intervalles. Mais attention, nous n'avons plus besoin de retester lesquels sont discrétisables et lesquels ne le sont pas étant donné que ceux qui le sont sont stockés dans notre liste ‘sauvegarde$_v aleurs$`.

Nous allons donc devoir parcourir notre liste fraichement construite pour récupérer le nom de tous les attributs qui sont à discrétiser.

“‘python pour chaque valeur à discrétiser for attribut, valeurs in sauvegarde$_v aleurs$: $ondiscrtise!ensemble.di$

Oui, je sais, ça ne nous dit pas ce qu'il y a dans la fameuse fonction ‘discretiser‘ de la classe ‘Ensemble‘ mais ça nous permet de garder une fonction de construction de taille *relativement* correcte bien qu'elle soit déjà assez longue !

C'est cette fonction ‘discretiser‘ qui va utiliser le pseudo-code que je vous ai donné un peu plus haut. Voilà plus ou moins l'ordre des opérations faites dessus :

Trier les valeurs Parcourir les valeurs Quand on a un changement de classe (étiquette) on ajoute un nouvel intervalle à la liste des intervalles Ajouter l'intervalle restant (celui qui va jusque $+\infty$) Parcourir les valeurs Remplacer la valeur continue par l'intervalle qui lui correspond

Commençons par déclarer et initialiser les variables :

“‘python liste$_i ntervallesvacontenirlalistedesintervallesrsultantdeladiscrtisationliste_i ntervalles =$
$list()valeurs_t rieesestinitialiseaveclesvaleursdel'attributdechaqueexemplevaleurs_t riees = [(i, self.liste_e$
$itemgetter(1))indice_b orne_i nf = 0$```

Je fais usage du module ‘operator‘ duquel j'ai importé ‘itemgetter‘ ce qui me permet de trier la liste selon le n$^e lment(icilavaleurcontinuetantdonnquelapremirevaleurestl'indicedel'exemple)$.

Il faut ensuite parcourir la liste et ajouter les intervalles au fur et à mesure.

“‘python on parcourt la liste triée for i in range(1, len(valeurs$_t riees$)) : $s'ilyaunchangementd'tiquetteif self.l$
$self.liste_e xemples[valeurs_t riees[indice_b orne_i nf]][0]].etiquette : lesbornes(infetsup)sontunemoyennedel'l$
$lepremierintervallecommence-\infty carilfautpouvoirgrertouteslesvaleursiflen(liste_i ntervalles) ==$
$0 : borne_i nf = float(" - inf") else : borne_i nf = (float(valeurs_t riees[indice_b orne_i nf][1]) +$
$float(valeurs_t riees[indice_b orne_i nf-1][1]))/2borne_s up = (float(valeurs_t riees[i][1])+ float(valeurs_t riees[$
$1][1]))/2unefoislesbornestrouves, onajoutel'intervallelalisteliste_i ntervalles.append((borne_i nf, borne_s up)$
i```

Je fais donc varier i pour parcourir toute la liste, je regarde ensuite si le i$^e lmentalammetiquettequeceluil'indi$

[[a]] |Je fais commencer ma boucle sur ‘i‘ à **1** et pas **0** pour la simple et bonne raison que l'indice **0** est géré par le fait que ‘indice$_b orne_i nf$`$estmis * *0 * *$.

On ajoute ensuite le dernier intervalle (celui qui va jusqu'à $+\infty$) de la sorte :

“‘python il ne faut pas oublier le dernier intervalle qui va jusqu'à $+\infty$! liste$_i ntervalles.append((liste_i ntervalle$
$inf"))$```

Il nous faut maintenant encore changer les valeurs continues en les intervalles correspondant. Pour ce faire, il nous faut parcourir tous les exemples et puis parcourir tous les intervalles jusqu'à trouver le bon. Heureusement, nous avons créé les intervalles dans l'ordre, les exemples

étant triés. Ça fait que nous pouvons à chaque fois tester si la valeur continue est inférieure à la borne supérieure car elle est d'office plus grande ou égale à la borne inférieure.

“‘python pour chaque exemple for exemple in self.liste$_exemples$: $for intervalle in liste_intervalles$: $ontrouvelebonintervalle if float(exemple.dict_attributs[nom_attribut]) < intervalle[1] : enondiscrtiseexem$ $intervallebreak$```

[[a]] |Beaucoup de personnes/écoles/institutions/... *interdisent* l'utilisation du ‘break‘. Vous pouvez l'enlever, ça ne changera rien au fonctionnement du programme, mais ça nous empêche de faire des tours de boucle inutiles.

Nous avons donc fini notre fonction ‘discretiser‘ de laquelle voici le code complet récapitulé :

“‘python def discretiser(self, nom$_attribut$) : """$remplacechaquevaleurdel'attributenquestionparl'ensemblea$ $list()valeurs_trieesestinitialiseaveclesvaleursdel'attributdechaqueexemplevaleurs_triees = [(i, self.liste_e$ $itemgetter(1))indice_borne_inf = 0onparcourtlalistetrie for iinrange(1, len(valeurs_triees)) : s'ilyaunchan$ $self.liste_exemples[valeurs_triees[indice_borne_inf][0]].etiquette : lesbornes(inf et sup)sontunemoyennedel'l$ $lepremierintervallecommence-\infty carilfautpouvoirgrertouteslesvaleursiflen(liste_intervalles) ==$ $0 : borne_inf = float("-inf")else : borne_inf = (float(valeurs_triees[indice_borne_inf][1]) +$ $float(valeurs_triees[indice_borne_inf-1][1]))/2borne_sup = (float(valeurs_triees[i][1])+ float(valeurs_triees[$ $1][1]))/2unefoislesbornestrouves, onajoutel'intervallelalisteliste_intervalles.append((borne_inf, borne_sup)$ $iilnefautpasoublierledernierintervallequivajusqu'+\infty !liste_intervalles.append((liste_intervalles[-1][1], f$ $inf")))$

pour chaque exemple for exemple in self.liste$_exemples$: $for intervalle in liste_intervalles$: $ontrouvelebonintervalle if float(exemple.dict_attributs[nom_attribut]) < intervalle[1] : enondiscrtiseexem$ $intervallebreak$```

Ah nous avançons ! Il nous reste encore le fait de restituer les valeurs continues n'ayant pas été choisies. Pour ça, il nous suffit de reparcourir la liste ‘sauvegarde$_valeurs$`etregardersilenomdel'attributsauvegar

“‘python pour chaque attribut sauvegardé for attribut, valeurs in sauvegarde$_valeurs$: $sicen'estpasl'attributc$ $a_tester : onremetlesanciennesvaleurscontinues for iinrange(len(valeurs)) : ensemble.liste_exemples[i].di$ $valeurs[i]$```$Nousavonsdoncgalementfinieladiscrtisation!Flicitations!$

Je vous remet le code de la fonction de génération C 4.5 pour que vous soyez sûrs d'avoir tout noté comme il faut (le code en *entier* sera repris à la toute fin).

“‘python def $_construire_arbre(self,ensemble):if notisinstance(ensemble,Ensemble):raiseTypeError("ensembledoittreunEnsembleetnonun"$

ne pas oublier de sauver les valeurs pour pouvoir les restituer au cas où l'attribut discrétisé n'est pas choisi sauvegarde$_valeurs = ensemble.sauvegarder_valeurs_discretes()pourchaquevaleurdiscrtiser for at$ $ondiscrtiseensemble.discretiser(attribut)onrcuprel'attributoptimalATTENTION : prciserID3 =$ $False pourutiliserleratiodegain a_tester = ensemble.attribut_optimal(ID3 = False)pourchaqueattributsau$ $sicen'estpasl'attributchoisiif attribut != a_tester : onremetlesanciennesvaleurscontinues for iinrange(len$ $ensemble.liste_exemples[i].dict_attributs[attribut] = valeurs[i]sionarriveici, onretourned'of iceunnudetp$ $Noeud(a_tester)pourchaquevaleurquepeutprendrel'attributtester forvaleurinensemble.valeurs_possibles_a$ $oncreunsous-ensemblesous_ensemble = ensemble.sous_ensemble_attribut(a_tester, valeur)etonencreunnou$ $self._construire_arbre(sous_ensemble)onretournelenudquel'onvientdecrerreturnnoeud$```

[$^d iscrete$] : $Attention : ici, oncontinued'utiliserdesnombresquel'onpourraitqualifierdediscretsvuquel'on$

Manipuler les données manquantes

Qu'en est-il de la manipulation de données manquantes ? Reprenons le tableau que je vous ai donné au **(§1.1.)** mais cette fois-ci, nous allons corrompre volontairement le tableau de manière à ce que deux données soient illisibles :

->

+——+————+————-+———-+———+———+ | Jour | Attributs des exemples | Classe | +——+————+————-+———-+———+———+ | | Prévisions | Température | Humidité | Vent | | +======+============+=============+==========+========+=
| 1 | Ensoleille | Chaud | Élevée | Faible | Non | +——+————+————-+———-+———+———+
| 2 | Ensoleille | Chaud | Élevée | Fort | Non | +——+————+————-+———-+———+———+
| 3 | Nuageux | Chaud | Élevée | Faible | Oui | +——+————+————-+———-+———+———+
| 4 | Pluvieux | Moyen | Élevée | Faible | Oui | +——+————+————-+———-+———+———+
| 5 | Pluvieux | Frais | Normale | Faible | Oui | +——+————+————-+———-+———+———+
| 6 | Pluvieux | Frais | Normale | Fort | Non | +——+————+————-+———-+———+———+
| 7 | ?? | Frais | Normale | Fort | Oui | +——+————+————-+———-+———+———+ | 8 | Ensoleille | Moyen | Élevée | Faible | Non | +——+————+————-+———-+———+———+
| 9 | Ensoleille | Frais | Normale | Faible | Oui | +——+————+————-+———-+———+———+
| 10 | Pluvieux | ?? | Normale | Faible | Oui | +——+————+————-+———-+———+———+
| 11 | Ensoleille | Moyen | Normale | Fort | Oui | +——+————+————-+———-+———+———+
| 12 | Nuageux | Moyen | Élevée | Fort | Oui | +——+————+————-+———-+———+———+
| 13 | Nuageux | Chaud | Normale | Faible | Oui | +——+————+————-+———-+———+———+
| 14 | Pluvieux | Moyen | Élevée | Fort | Non | +——+————+————-+———-+———+———+

Table : Ensemble adapté aux données manquantes

<-

J'ai remplacé la **Prévision** du **7** et la **Température** du **10** par ** ??**. Nous avons deux possibilités qui s'offrent à nous :

- soit nous supprimons méchamment ces exemples de notre set ; - soit nous tentons d'assigner une valeur à la valeur manquante.

Bien entendu, la première méthode est la plus sûre dans le sens où on ne prend pas le risque de créer un exemple qui n'existe pas, mais par contre ça diminue notre set d'exemples alors que dans notre cas il n'est déjà pas très grand (un véritable problème de *machine learning* peut contenir plusieurs centaines de milliers d'exemples pour plusieurs centaines d'attributs). L'algorithme C4.5 nous propose de déterminer une valeur à cet attribut manquant à l'exemple. Comment faire pour savoir quelle est la bonne valeur ? A nouveau, ici nous n'avons que maximum **3** valeurs possibles par attribut (en ne considérant que les attributs discrets) mais il est possible qu'un attribut ait **50** valeurs possibles.

[[q]] |Alors comment savoir laquelle mettre ?

- soit on assigne à cet attribut la valeur que l'attribut a le plus souvent dans le set ; - soit on assigne à cet attribut la valeur que l'attribut a le plus souvent pour la même classification.

[[q]] |Qu'est-ce que ça veut dire tout ça ?

Le premier cas veut dire que nous allons regarder dans les **13** exemples qui nous restent (**14** dans notre set moins celui qui est inconnu) quelle est la valeur qui revient le plus souvent.

Pour assigner une valeur à la **Prévision** de notre **7**e exemple, nous allons regarder le compte des valeurs de l'attribut Prévision dans tout notre set :

On peut donc voir que les valeurs les plus assignées sont **Ensoleillé** et **Pluvieux**. Nous devrons donc mettre à la place de notre **??** soit **Ensoleillé** soit **Pluvieux**. Ceci-dit, si vous regardez la valeur que l'on est censés obtenir, vous verrez que c'est **Pluvieux**. On n'obtient donc pas le bon résultat. Si on regarde pour l'attribut **Température** du 10e exemple, il faut procéder de la même manière, à savoir compter le nombre d'occurrences de chaque valeur de l'attribut **Température** :

$$\#\text{Température} = \#\text{Moyen} + \#\text{Chaud} + \#\text{Frais} = 4 + 5 + 4 = 13$$

Là, on peut voir que c'est la valeur **Chaud** qui est le plus assignée, et si on regarde dans le tableau complet, c'est bien la valeur que nous devons obtenir. Donc on voit que cette méthode peut fonctionner mais pas dans tous les cas.

Maintenant, regardons alors la seconde méthode : pour trouver quelle valeur mettre à l'attribut **Prévision** du **7e**, il faut tout d'abord regarder la classe du **7e** exemple, qui est **Oui**. Faisons ensuite un sous-ensemble contenant uniquement les exemples étiquetés **Oui**. Le voici :

Jour	**Attributs des exemples**				**Classe**
	Prévisions	Température	Humidité	Vent	
3	**Nuageux**	**Chaud**	**Élevée**	**Faible**	**Oui**
4	Pluvieux	Moyen	Élevée	Faible	Oui
5	Pluvieux	Frais	Normale	Faible	Oui
7	??	Frais	Normale	Fort	Oui
9	Ensoleille	Frais	Normale	Faible	Oui
10	Pluvieux	??	Normale	Faible	Oui
11	Ensoleille	Moyen	Normale	Fort	Oui
12	Nuageux	Moyen	Élevée	Fort	Oui
13	Nuageux	Chaud	Normale	Faible	Oui

Table : Sous-ensemble de l'ensemble d'exemples de playTennis avec uniquement les exemples étiquetés **Oui**

Et là, il faut faire exactement le même procédé que dans la première méthode à savoir compter le nombre d'occurrences de chaque valeur :

$$\#\text{Prévision} = \#\text{Ensoleillé} + \#\text{Nuageux} + \#\text{Pluvieux} = 2 + 3 + 3 = 8$$

On voit donc que la valeur à assigner à l'attribut **Prévision** du **7e** exemple est soit **Nuageux** soit **Pluvieux**. C'est en réalité **Nuageux**, donc pour cette valeur, cette seconde méthode s'est avérée plus efficace que la première. Maintenant cherchons la valeur de l'attribut **Température** du 10e exemple : l'étiquette étant la même, on garde le même tableau de sous-ensemble.

$$\#\text{Température} = \#\text{Moyen} + \#\text{Chaud} + \#\text{Frais} = 2 + 3 + 3 = 8$$

La valeur à assigner est donc soit **Chaud** soit **Pluvieux**. Et en réalité, c'est **Chaud**. Donc cette méthode fonctionne également. Je vous conseille personnellement d'utiliser la seconde méthode que la première car son taux d'exactitude est légèrement plus élevé que la première.

Cependant, que faire quand le nombre d'occurrences est ex æquo ? C'est vrai, c'est le cas que nous avons eu ici au-dessus. Premièrement, laissez-moi vous dire que moins il y a d'exemples dans le set, plus vous avez de chances de tomber sur des comptes ex æquo. Alors que faire ? Lequel choisir ? Moi je vous dirais, choisissez soit n'importe lequel (donc au hasard) ce qui n'est pas très professionnel, soit refaites un sous-ensemble avec un attribut commun, puis un autre etc. Ceci dit, plus votre set est grand, moins il est grave que votre valeur ne soit pas exacte. Mais attention, ce n'est pas une raison pour la choisir complètement au hasard ! Personnellement, je choisis au hasard une des valeurs maximum quand il y en a et ça suffit amplement. De plus, ceci n'arrive que très rarement.

Voici le pseudo-code pour la complétion d'exemples corrompus.

```
FONCTION compléter_exemples
        Entrée :
                - exemples = liste d'exemples étiquetés à compléter
                    (données manquantes)
        Retour :
                - rien
DEBUT
        POUR CHAQUE e dans exemples, FAIRE
                SI e n'est pas complet, alors
                        y = classe de e
                        S = sous-ensemble de exemples ayant pour
                            classe y
                        A = attribut manquant de e
                        A de e = A le plus fréquent dans S
                FIN SI
        FIN POUR CHAQUE
FIN
```

Maintenant, nous allons l'implémenter. Pour ce faire, il faut déjà déterminer où nous allons devoir modifier le code. Il va falloir demander de restaurer l'ensemble juste avant de lancer la génération. Il faut donc changer la méthode `construire`. Une petite ligne suffit :

```
def construire(self):
    """
        retourne l'arbre au complet
    """
    #si le set est corrompu (attributs manquants), on le restaure
    self.ensemble.restaurer_valeurs_manquantes()
    self.arbre = self.__construire_arbre(self.ensemble)
```

Ce qui nous montre bien qu'il faut *encore une fois* aller modifier la classe Ensemble. Il nous faut une fonction `restaurer_valeurs_manquantes` qui ne prend aucun paramètre. Comment doit fonctionner cette fonction ? Elle doit tout d'abord regarder pour chaque exemple si au moins un attribut a `'?'` pour valeur. Si oui, le déterminer et modifier ce même `'?'`.

On commence donc par chercher quels attributs valent `'?'`.

```
def restaurer_valeurs_manquantes(self):
    """
        change chaque '?' dans les attributs, le remplacer par
        la valeur de l'attribut la plus fréquente dans le set
        (pour les exemples ayant la même étiquette)
    """
    #pour chaque exemple
    for i in range(len(self.liste_exemples)):
        #on check chaque chaque attribut
        for nom_attribut in self.liste_exemples[i].dict_attributs:
            #si l'attribut en question vaut '?'
            if self.liste_exemples[i].dict_attributs[nom_attribut]
                == '?':
```

Une fois trouvés, on fait un sous-ensemble selon l'étiquette de l'exemple en question, et ensuite, on récupère la valeur la plus présente.

```
def restaurer_valeurs_manquantes(self):
    """
        change chaque '?' dans les attributs, le remplacer par
        la valeur de l'attribut la plus fréquente dans le set
        (pour les exemples ayant la même étiquette)
    """
    #pour chaque exemple
    for i in range(len(self.liste_exemples)):
        #on check chaque chaque attribut
        for nom_attribut in self.liste_exemples[i].dict_attributs:
            #si l'attribut en question vaut '?'
            if self.liste_exemples[i].dict_attributs[nom_attribut]
                == '?':
```

```
13                  #on isole les éléments ayant la même étiquette
14                  sous_ensemble = self.sous_ensemble_etiquette(
15
                                                self.liste_exemples[i].etiquette)
16                  #et on récupère la valeur de ce même attribut la
                        plus
17                  #fréquente pour l'assigner à la place du '?'
18                  self.liste_exemples[i].dict_attributs[nom_attribut]
                        = \
19
                          sous_ensemble.valeur_plus_frequente_attribut(nom_attribu
```

Il nous faut encore maintenant déterminer la fonction `valeur_plus_frequente_attribut`. Cette fonction doit donc également se situer dans la classe `Ensemble` et prendre un argument : le nom de l'attribut que l'on explore. Je vous la donne en un coup car elle n'est pas très compliquée :

```
1  def valeur_plus_frequente_attribut(self, nom_attribut):
2      """
3          renvoie un str avec la valeur la plus fréquente de
4          l'attribut en question dans l'ensemble
5      """
6      dict_frequences = dict()
7      #pour chaque exemple de l'ensemble
8      for exemple in self.liste_exemples:
9          #on regarde quelle est sa valeur pour l'attribut en
               question
10         #si elle n'a pas encore été rencontrée
11         if exemple.dict_attributs[nom_attribut] not in
               dict_frequences:
12             #on l'ajoute au dictionnaire
13             dict_frequences[exemple.dict_attributs[nom_attribut]] =
                   0
14         dict_frequences[exemple.dict_attributs[nom_attribut]] += 1
15     #on retourne la clef pour laquelle la valeur est la plus élevée
16     return max(dict_frequences, key=dict_frequences.get)
```

Je stocke au fur et à mesure les fréquences dans un dictionnaire et au final, je renvoie la *clef* correspondant à la plus grande valeur du dictionnaire étant donné que la clef est la valeur de l'attribut la plus fréquente.

Nous avons maintenant fini la restauration de données corrompues. Je vous avais bien dit que ça n'avait rien de sorcier !

3.2. Élagage de l'arbre

Terminons en parlant de l'élagage et de l'"*overfitting*".

? Qu'est-ce que l'*overfitting* ?

C'est une très bonne question. "To fit" en anglais veut dire "correspondre". L'*overfitting* est donc le fait de **trop** correspondre. Alors bon, comme ça, ça ne veut pas dire grand-chose. Mais vous devez comprendre par là qu'il existe trois catégories de règles[7] pour chaque problème de *machine learning* :

— les règles *underfitting* ;
— les règles *fitting* ;
— les règles *overfitting*.

La première catégorie est un ensemble de règles qui sont très mauvaises car elles ne parviennent pas à classer correctement les éléments du set d'exemples. La seconde catégorie, les règles *fitting* sont de bonnes règles car elles correspondent relativement bien aux exemples, et encore mieux, elles permettent de prédire relativement bien la classe des nouveaux exemples non étiquetés. Et la dernière, les règles *overfitting* sont mauvaises car elles correspondent parfaitement au set d'exemples ce qui fait qu'elles faillissent à une prédiction des classes des nouveaux exemples non étiquetés. L'*overfitting* arrive fréquemment lorsque l'on travaille avec un set d'exemples trop petit. Prenons le set d'exemples sur lequel nous avons travaillé, nous avons **14** des **36** exemples possibles. $\frac{14}{36} \simeq 39\%$. Ça veut donc dire que nous disposons de **39%** des cas possibles et donc que l'arbre nous sert à deviner les **61%** qui restent. Autant vous dire que nous n'avons pas beaucoup d'exemples au final. Nous risquons donc de rencontrer le phénomène d'*overfitting*. Imaginez l'exemple suivant :

Jour	**Attributs des exemples**				**Classe**
	Prévisions	Température	Humidité	Vent	
15	**Pluvieux**	**Chaud**	**Faible**	**Faible**	**Non**

Table : Nouvel exemple

Notre arbre échouerait à l'étiqueter correctement si nous le lui demandions : l'arbre analyserait tout d'abord l'attribut **Prévisions** pour lequel il suivrait la branche **Pluvieux**. Ensuite il analyserait l'attribut **Vent** et suivra la branche **Faible**. Et là il l'étiquetterait **Oui** alors que nous le voulons étiqueté **Non**.

Il y a deux moyens de se débarrasser de l'*overfitting* d'un arbre :

— ajouter des exemples étiquetés pour la création de l'arbre ;
— élaguer l'arbre.

Effectivement, comme je vous ai dit que l'*overfitting* était majoritairement dû à un manque d'exemples d'entrainement, le fait d'en rajouter diminuera l'*overfitting*. Mais ce n'est pas le seul

moyen. Si vous ne pouvez pas rajouter d'exemples à votre set mais que vous rencontrez tout de même de l'*overfitting* dans votre arbre, il vous faut faire autrement : il vous faut l'élaguer.

? Comment fait-on pour élaguer un arbre ?

C'est ce que je m'apprête à vous expliquer. Il existe deux types d'élagage :

— le pré-élagage ;
— le post-élagage.

La différence entre ces deux types d'élagage est que le premier se fait pendant la génération de l'arbre et que le second se fait après la génération. L'avantage du pré-élagage est le fait qu'aucun nouvel exemple ne doit être apporté au set d'entrainement. Son inconvénient est qu'il est difficile à mettre en place parce qu'on ne sait pas vraiment quand l'arrêter. L'avantage du post-élagage est qu'il est assez intuitif et qu'il est facile à mettre en œuvre. Son inconvénient est qu'il nécessite de nouveaux exemples.

Laissez-moi vous expliquer comment chacune de ces deux méthodes fonctionnent. La première est un peu plus compliquée donc commençons par la seconde. Le principe est le suivant : une fois l'arbre construit, il faut voir nœud par nœud s'il y a moyen de transformer le nœud en feuille. Il faut bien entendu partir de tout en haut et observer pour chaque fils. Voici le pseudo-code pour ce procédé :

```
1  FONCTION élaguer
2      Entrée :
3          - nœud = l'arbre (ou sous-arbre) généré par l'algorithme
               C4.5 (on peut également élaguer un arbre ID3)
4          - exemples = set d'exemples pré-étiquetés (non utilisés
               pour l'élaboration de l'arbre !)
5      Retour :
6          - rien
7  DEBUT
8      POUR CHAQUE classe c, FAIRE
9          SI taux d'erreur(nœud remplacé par c, exemples) < taux
               d'erreur(nœud, exemples), alors
10             remplacer nœud par c
11             ARRÊTER FONCTION
12         FIN SI
13     FIN POUR CHAQUE
14
15     POUR CHAQUE fils de nœud f, FAIRE
16         s = sous-ensemble de exemples ayant pour l'attribut
               nœud.attribut la valeur des exemples arrivant en f
17         élaguer(f, S)
18     FIN POUR CHAQUE
19 FIN
```

Pour l'autre méthode, il y a plusieurs possibilités :

— soit on choisit un seuil pour le nombre d'itérations de la récursivité de l'arbre. Par exemple, on met un seuil à la **10e** ligne ce qui fait qu'à la **10e** ligne maximum dans l'arbre, tous les exemples sont étiquetés ;
— soit on s'arrête quand le fait de recréer un nœud n'a pas une grande influence statistiquement.

Pas besoin de vous dire que la première n'est pas spécialement une bonne méthode du fait que c'est très arbitraire et que le seuil est très difficile à estimer. La seconde méthode est déjà plus sûre mais n'est pas moins ardue à mettre en œuvre.

Je vous conseille donc d'utiliser la méthode du post-élagage tant que possible, et quand vous n'avez pas d'autre choix que d'utiliser la méthode de pré-élagage, je vous conseille d'utiliser la seconde méthode, mais là, à vous de trouver ce qu'est "*une grande influence statistique*".

Au niveau de l'implémentation, j'ai choisi de m'occuper de la méthode du *post-élagage*. Il va donc nous falloir une fonction d'élagage dans notre classe `Arbre_C45`. Je propose la même entourloupe que pour les fonctions de génération et d'affichage : une méthode publique qui fera appel à une méthode privée récursive. Voici donc ma fonction d'élagage :

```
def elaguer(self):
    """
        modifie l'arbre en élaguant suivant l'ensemble de travail
        d'élagage donné dans self.chemin_elagage
    """
    #ATTENTION : si le chemin n'a pas été donné, on n'élague pas !
    if self.chemin_elagage != "":
        self.arbre = self.__elaguer_noeud(self.arbre,

                                          Ensemble(self.chemin_elagage))
```

Vous pouvez y voir que j'utilise une variable `chemin_elagage` de la classe `Arbre_C45` alors que nous ne l'avons nul part déclarée. C'est parce que j'ai changé ma fonction d'initialisation de la classe (son constructeur). Il nous faut également savoir où se trouve le fichier contenant l'ensemble d'exemples permettant l'élagage. Ce même fichier est nécessaire dès l'initialisation :

```
def __init__(self, chemin_données="", chemin_elagage=""):
    """
        chemin_données est l'emplacement du fichier contenant les données
        chemin_elagage est l'emplacement d'un autre set permettant
        d'élaguer l'arbre
    """
    #initialisation de l'ensemble avec le fichier dans
        chemin_données
    self.ensemble = Ensemble(chemin_données)
    #initialisation du nœud principal de l'arbre
    self.arbre = None
    self.chemin_elagage = chemin_elagage
```

J'ai donc créé notre nouvelle variable et je lui ai donné une valeur par défaut si l'utilisateur ne désire pas élaguer l'arbre.

Maintenant, il nous faut déterminer la fonction `__elaguer_noeud`. Cette fonction, comme vous pouvez le voir dans le code ci-dessus, prend deux paramètres (en plus du paramètre `self` bien entendu). Ces paramètres sont un nœud (celui à élaguer) et un ensemble (d'élagage). Notre déclaration doit donc ressembler à ceci :

```
def __elaguer_noeud(self, noeud, ensemble_elagage):
    """
        élague le noeud passé en paramètre et le retourne
    """
```

Le code de cette fonction doit donc s'arrêter si il reçoit une Feuille en paramètre et pas un Nœud. En effet, si c'est une feuille qui est envoyée il n'y a rien à optimiser.

Nous avons donc déjà ceci :

```
#si on est sur une feuille, on ne va pas plus loin
if isinstance(noeud, Feuille):
    return noeud
```

Ensuite, voilà ce qui va se dérouler : on va tester pour chaque étiquette par laquelle on pourrait remplacer le nœud que vaut le taux d'erreur. Si ce taux d'erreur est inférieur au taux d'erreur de l'arbre tel quel, alors on va remplacer le nœud par une feuille contenant l'étiquette en question.

```
min_erreur, etiquette_gardee = 1.0, ""
proportion_initiale = self.taux_erreur(ensemble_elagage)
sauvegarde = self.arbre
#pour chaque étiquette
for etiquette in ensemble_elagage.etiquettes_possibles():
    self.arbre = Feuille(etiquette)
    #on calcule le taux d'erreur si on remplace le nœud par
    #l'étiquette en question
    taux_erreur_actuel = self.taux_erreur(ensemble_elagage)
    #on sauvegarde le meilleur taux
    if taux_erreur_actuel < min_erreur:
        min_erreur, etiquette_gardee = taux_erreur_actuel,
            etiquette
#s'il existe un taux avantageux on élague à cet endroit
if min_erreur <= proportion_initiale:
    return Feuille(etiquette_gardee)
else:
    self.arbre = sauvegarde
#on n'oublie pas de restaurer la valeur du sommet de l'arbre !
sauvegarde, self.arbre = self.arbre, sauvegarde
```

Ici, je cherche donc quelle est l'étiquette qui donne le meilleur taux d'erreur (donc le plus faible). Et si ce taux est inférieur au taux d'erreur initial (donc celui de l'arbre non-élagué (du moins de ce nœud non-élagué)), alors je décide de changer le nœud en feuille.

Par contre, pour pouvoir calculer le taux d'erreur qui se trouve directement dans la classe `Arbre_C45`, il faut que self.arbre pointe directement sur le nœud voulu. Nous verrons la fonction `taux_erreur` dans un instant.

Maintenant, si le nœud n'a pas été élagué parce que ce n'était pas avantageux, il nous faut tenter de procéder à la même chose pour chaque enfant de ce même nœud. Pour ça, il faut boucler sur chaque enfant et tenter de le discrétiser :

```
#on teste chaque enfant pour voir s'il est élagable
for enfant in noeud.enfants:
    sous_ensemble = ensemble_elagage.sous_ensemble_attribut(
                                            noeud.attribut_teste,
                                            enfant)
    #s'il l'est, on l'élague
    if len(sous_ensemble) != 0:
        noeud.enfants[enfant] = self.__elaguer_noeud(
                                            noeud.enfants[enfant],
                                            sous_ensemble)
```

Après ça, il nous faut renvoyer le nœud ayant (peut-être) des enfants élagués :

```
return noeud
```

Voici donc à quoi ressemble notre fonction d'élagage de nœud :

```
def __elaguer_noeud(self, noeud, ensemble_elagage):
    """
        élague le noeud passé en paramètre et le retourne
    """
    #si on est sur une feuille, on ne va pas plus loin
    if isinstance(noeud, Feuille):
        return noeud
    min_erreur, etiquette_gardee = 1.0, ""
    proportion_initiale = self.taux_erreur(ensemble_elagage)
    sauvegarde = self.arbre
    #pour chaque étiquette
    for etiquette in ensemble_elagage.etiquettes_possibles():
        self.arbre = Feuille(etiquette)
        #on calcule le taux d'erreur si on remplace le nœud par
        #l'étiquette en question
        taux_erreur_actuel = self.taux_erreur(ensemble_elagage)
        #on sauvegarde le meilleur taux
```

```
18          if taux_erreur_actuel < min_erreur:
19              min_erreur, etiquette_gardee = taux_erreur_actuel,
                    etiquette
20      #s'il existe un taux avantageux on élague à cet endroit
21      if min_erreur <= proportion_initiale:
22          return Feuille(etiquette_gardee)
23      else:
24          self.arbre = sauvegarde
25      #on n'oublie pas de restaurer la valeur du sommet de l'arbre !
26      sauvegarde, self.arbre = self.arbre, sauvegarde
27      #on teste chaque enfant pour voir s'il est élagable
28      for enfant in noeud.enfants:
29          sous_ensemble = ensemble_elagage.sous_ensemble_attribut(
30
                                                        noeud.attribut_teste,
31                                                  enfant)
32          #s'il l'est, on l'élague
33          if len(sous_ensemble) != 0:
34              noeud.enfants[enfant] = self.__elaguer_noeud(
35
                                                        noeud.enfants[enfant],
36                                                  sous_ensemble)
37      #et au final, on renvoie le nœud aux enfants peut-être élagués
38      return noeud
```

Il ne nous reste plus que la fonction qui nous permet de calculer le taux d'erreur d'un arbre par rapport à un ensemble d'exemples. Le principe de cette fonction est de tenter d'étiqueter chaque ensemble du set, et ensuite de regarder s'il a été bien étiqueté.

Voici donc ce que je peux vous proposer :

```
1   def taux_erreur(self, ensemble):
2       """
3           renvoie un nombre dans [0, 1] correspondant à la proportion
4           d'exemples dans ensemble qui se font étiqueter correctement
5           avec l'arbre tel quel
6       """
7       compteur_etiquetages_incorrects = 0
8       #pour chaque exemple
9       for exemple in ensemble.liste_exemples:
10          #on garde son étiquette
11          etiquette = exemple.etiquette
12          self.etiqueter(exemple)
13          #et on la compare à l'étiquette donnée par l'arbre
14          if etiquette != exemple.etiquette:
15              #si elles sont différentes, on augmente la proportion
16              #d'étiquetages incorrects et on rétablit la bonne
                    étiquette
```

```
17                compteur_etiquetages_incorrects += 1
18                exemple.etiquette = etiquette
19        return compteur_etiquetages_incorrects/len(ensemble)
```

J'utilise un compteur pour trouver le nombre d'exemples étiquetés correctement et puis je divise le compteur par la longueur de l'ensemble, ce qui me donne une proportion.

Attention : il faut faire très attention lors de l'élagage car si cette étape fait disparaitre des valeurs d'un attribut, il se peut que vous ayez une erreur lorsque vous tenterez d'étiqueter un nouvel exemple. Vous risquez ce même problème en *devinant* un attribut manquant. Car en changeant l'ensemble, l'arbre peut être assez différent, et certaines branches peuvent se voir ajoutées, modifiées, ou encore supprimées. Si c'est le cas, l'étiquetage ne fonctionnera pas. Peut être devriez-vous gérer un étiquetage impossible.

Nous en avons non seulement fini avec l'élagage d'arbre, mais nous en avons également fini avec l'algorithme C 4.5 ! Toutes mes félicitations. Je vais une dernière fois vous donner l'entièreté du code qui a été travaillé tout au long de ce cours. Ce code est susceptible de changer sans spécialement que le tuto soit mis à jour. Si vous voulez y participer, je vous invite à suivre le dépôt GitHub qui lui est associé. Toutes les propositions sont les bienvenues !

Voici donc le code comme promis (comportant tout le code ID3 et C 4.5 ainsi qu'une fonction d'exemples pour pouvoir exploiter les différentes possibilités de tout ce travail) :

Contenu masqué n°2

C'est ainsi que s'achève notre long mais passionant périple. Merci beaucoup de m'avoir lu (pour ceux qui sont allés jusqu'au bout). En espérant que ceci vous a plu !

#1. Sources :

[1] SCHAPIRE R, *Theoretical Machine Learning* [Document électronique], http ://www.cs.princeton.edu/courses/archive/spr08/cos511/scribe_notes/0204.pdf

[2] QUINLAN R, *Induction of Decision Trees*, Machine Learning, 1986

[3] QUINLAN R, *C4.5 programs for machine learning*, Kaufmann, 1993

[4] Lecture slides for textbook *Machine Learning*, Tom M. Mitchell, McGraw Hill, 1997, http ://www.cs.cmu.edu/afs/cs/project/theo-20/www/mlbook/ch3.pdf

#2. Liens

Pour aller plus loin, voici quelques liens sur le machine learning et l'intelligence artificielle en général :

7. je vous rappelle que l'on parle également de **concept**.

— DE LESPINAY JP, *Conscience artificielle et robotique : fin de l'évolution humaine !* [Document électronique], http ://www.automatesintelligents.com/echanges/2009/mar/conscienceartificielle.pdf
— GANASCIA JG, *L'Intelligence Artificielle*, Paris, Le Cavalier Bleu, 2007, coll. "idées reçues".
— RUSSEL R, NORVIG P, *Intelligence Artificielle*, Paris, Pearson, 2010, coll. "Pearson Education"[8]
— POTTER S, *What AI can get from neurosciences* [Document électronique], https ://neurolab.gatech.edu/wp/wp-content/uploads/potter/publications/Potter-NeuroscienceForAIchapter.pdf

— POTTER S, Neuroingineering : *Neuroscience – APPLIED* [Vidéo d'un séminaire datant du 19 octobre 2012], www.youtube.com

Contenu masqué n°3

Contenu masqué

Contenu masqué n°2

```
"""
   écrit par poupou9779 pour Zeste de Savoir
"""

from math import log
from operator import itemgetter

class Exemple:
    """
        Un exemple contient 2 valeurs :
            - un dictionnaire d'attributs (dict)
            - une étiquette (str)
    """

    def __init__(self, noms_attributs, valeurs_attributs,
        etiquette=""):
        """
            etiquette peut être non précisée en quel cas on aurait
            un exemple non étiqueté
        """
        #si on a un problème de types
        if not isinstance(noms_attributs, list) \
        or not isinstance(valeurs_attributs, list):
```

8. La bible dans le domaine !

```
23              raise
                    TypeError("noms_attributs et valeurs_attributs doivent être"
                    \
24                              " des listes et pas des {0} et {1}" \
25                              .format(type(noms_attributs),
26                                      type(valeurs_attributs)))
27          if not isinstance(etiquette, str):
28              raise
                    TypeError("etiquette doit être un str et pas un {}"
                    \
29                              .format(type(etiquette)))
30          #si les deux listes n'ont pas le même nombre d'éléments
31          if len(valeurs_attributs) != len(noms_attributs):
32              raise
                    ValueError("noms_attributs et valeurs_attributs doivent "
                    \
33                               "avoir le même nombre d'éléments")
34          self.etiquette = etiquette
35          self.dict_attributs = dict()
36          #on ajoute chaque attribut au dictionnaire
37          for i in range(len(noms_attributs)):
38              self.dict_attributs[noms_attributs[i]] =
                    valeurs_attributs[i]
39
40  class Ensemble:
41      """
42          Un ensemble contient deux valeurs :
43              - les noms des attributs (list)
44              - les exemples (list)
45      """
46
47      def __init__(self, chemin=""):
48          """
49              chemin est l'emplacement du fichier contenant les données.
50              Cette variable peut être non précisée en quel cas les variables
51                  seront initialisées comme des listes vides.
52          """
53          #Python est un langage à typage dynamique fort,
54          #il faut donc vérifier que l'utilisateur ne fait pas
                n'importe quoi
55          #en passant autre chose qu'un str
56          if not isinstance(chemin, str):
57              raise TypeError("chemin doit être un str et non {}" \
58                              .format(type(chemin)))
59          if chemin == "":
60              #initialisation en listes vides
61              self.liste_attributs = list()
62              self.liste_exemples = list()
63          else:
64              with open(chemin, 'r') as fichier:
```

```
65                #on stocke chaque mot de la première ligne dans
                      liste_attributs
66                self.liste_attributs = \
67
                                       fichier.readline().lower().strip().split('
68                #ensuite on stocke la liste d'exemples dans
                      liste_exemples
69                self.liste_exemples = self.liste_en_exemples(
70
                                            fichier.read().strip().lower().spli
71                                      self.liste_attributs
72                                    )
73
74        def __len__(self):
75            """
76                retourne la longueur de l'ensemble
77            """
78            return len(self.liste_exemples)
79
80        @staticmethod
81        def liste_en_exemples(exemples, noms_attributs):
82            """
83                retourne une liste d'exemples sur base d'une liste de str contenant
84                les valeurs et d'une liste de str contenant les noms des attributs
85            """
86            #on initialise la liste à retourner
87            ret = list()
88            for ligne in exemples:
89                #on stocke chaque mot de la ligne dans une liste
                      attributs
90                attributs = ligne.lower().strip().split(' ')
91                #met l'étiquette par défaut si elle n'est pas dans la
                      ligne
92                etiquette = attributs[-1] if len(attributs) !=
                      len(noms_attributs) \
93                                          else ""
94                #on ajoute un objet de type Exemple contenant la ligne
95                ret.append(Exemple(noms_attributs,
96                                   attributs[:len(noms_attributs)],
97                                   etiquette))
98            return ret
99
100       def etiquettes_possibles(self):
101           """
102               retourne une liste contenant les étiquettes de l'ensemble
103           """
104           #on initialise la valeur de retour
105           ret = list()
106           #pour chaque exemple de l'ensemble
107           for exemple in self.liste_exemples:
```

```
108             #si l'étiquette n'est pas déjà dans la liste
109             if not exemple.etiquette in ret:
110                 #on l'ajoute
111                 ret.append(exemple.etiquette)
112         return ret
113 
114     def sous_ensemble_etiquette(self, etiquette):
115         """
116             retourne un ensemble contenant uniquement les exemples ayant
117             etiquette comme étiquette
118         """
119         #initialisation de la valeur de retour
120         ret = Ensemble()
121         #on copie la liste d'attributs
122         ret.liste_attributs = self.liste_attributs[:]
123         #pour chaque exemple de l'ensemble initial
124         for exemple in self.liste_exemples:
125             #si l'étiquette est bonne
126             if exemple.etiquette == etiquette:
127                 #on l'ajoute au sous-ensemble
128                 ret.liste_exemples.append(exemple)
129         return ret
130 
131     def sous_ensemble_attribut(self, nom_attribut, valeur):
132         """
133             retourne un sous-ensemble contenant uniquement les exemples ayant
134             la bonne valeur pour l'attribut
135         """
136         ret = Ensemble()
137         #on prend tous les attributs sauf celui passé en paramètre
138         ret.liste_attributs = self.liste_attributs[:]
139         ret.liste_attributs.remove(nom_attribut)
140         #pour chaque exemple de l'ensemble
141         for exemple in self.liste_exemples:
142             #s'il a la bonne valeur
143             if exemple.dict_attributs[nom_attribut] == valeur:
144                 #on l'ajoute
145                 ret.liste_exemples.append(exemple)
146         #et on retourne la liste
147         return ret
148 
149     def entropie(self):
150         """
151             retourne l'entropie de Shannon de l'ensemble
152         """
153         #initialisation de la variable retournée
154         ret = 0
155         #pour chaque étiquette de l'ensemble
156         for etiquette in self.etiquettes_possibles():
```

```
157             #on crée un sous-ensemble qui ne contient que les
                    éléments de
158             #self ayant etiquette comme étiquette
159             sous_ensemble = self.sous_ensemble_etiquette(etiquette)
160             #on ajoute |c| * log_2(|c|) à ret
161             longueur_sous_ensemble = len(sous_ensemble)
162             ret += longueur_sous_ensemble *
                    log(longueur_sous_ensemble, 2)
163         #on retourne log_2(|S|) - ret/|S|
164         return log(len(self), 2) - ret/len(self)
165
166     def attribut_optimal(self, ID3=True):
167         """
168             retourne un str avec le nom de l'attribut à tester
169         """
170         max, ret = float("-inf"), ""
171         #pour chaque attribut
172         for attribut in self.liste_attributs:
173             if ID3:
174                 gain = self.gain_entropie(attribut)
175             else:
176                 gain = self.ratio_gain(attribut)
177             #si le gain d'entropie est le plus grande
178             if gain >= max:
179                 #on le garde en mémoire
180                 max, ret = gain, attribut
181         #et on le retourne
182         return ret
183
184     def valeurs_possibles_attribut(self, nom_attribut):
185         """
186             retourne une liste contenant toutes les
187             valeurs possibles de l'attribut
188         """
189         ret = list()
190         #pour chaque exemple
191         for exemple in self.liste_exemples:
192             #si cette valeur n'est pas encore dans la liste
193             if not exemple.dict_attributs[nom_attribut] in ret:
194                 #on l'ajoute
195                 ret.append(exemple.dict_attributs[nom_attribut])
196         #et on retourne la liste
197         return ret
198
199     def gain_entropie(self, nom_attribut):
200         """
201             retourne la perte d'entropie selon la définition de Ross Quinlan
202         """
203         somme = 0
204         #pour chaque valeur de l'attribut en question
```

```
205         for valeur in self.valeurs_possibles_attribut(nom_attribut):
206             #déclaration de Sv
207             sous_ensemble =
                    self.sous_ensemble_attribut(nom_attribut, valeur)
208             #somme = somme sur v de |Sv| * Entropie(Sv)
209             somme += len(sous_ensemble) * sous_ensemble.entropie()
210         #Gain(S, A) = Entropie(S) - 1/|S| * somme
211         return self.entropie() - somme/len(self)
212
213     def ratio_gain(self, nom_attribut):
214         """
215             retourne le ratio de gain (C4.5) de l'ensemble
216         """
217         split = self.split_entropie(nom_attribut)
218         gain = self.gain_entropie(nom_attribut)
219         return gain/split if split != 0 else float("inf")
220
221     def split_entropie(self, nom_attribut):
222         """
223             retourne le split d'entropie du set selon l'attribut en question
224         """
225         ret = 0
226         for valeur in self.valeurs_possibles_attribut(nom_attribut):
227             sous_ensemble =
                    self.sous_ensemble_attribut(nom_attribut, valeur)
228             ret += len(sous_ensemble) * log(len(sous_ensemble), 2)
229         return log(len(self), 2) - ret/len(self)
230
231     def est_discretisable(self, nom_attribut):
232         """
233             renvoie True si l'attribut en question est un attribut continu
234         """
235         #pour chaque exemple
236         for exemple in self.liste_exemples:
237             #si la valeur de l'attribut n'est pas un nombre
238             try:
239                 float(exemple.dict_attributs[nom_attribut])
240             #alors l'attribut n'est pas discrétisable
241             except ValueError:
242                 return False
243         #si tous les exemples sont discrétisables pour cet attribut,
244         #l'attribut est  discrétisable
245         return True
246
247     def sauvegarder_valeurs_discretes(self):
248         """
249             renvoie une liste de tuples contenant :
250                 + le nom de l'attribut
251                 + la liste de toutes les valeurs de cet attribut
252         """
```

```
253         ret = list()
254         #pour chaque attribut
255         for attribut in self.liste_attributs:
256             #s'il est discrétisable
257             if self.est_discretisable(attribut):
258                 #on sauvegarde les valeurs de chaque exemple
259                 valeurs_attribut = [exemple.dict_attributs[attribut]
                        \
260                                     for exemple in
                                            self.liste_exemples]
261                 ret.append((attribut, valeurs_attribut))
262         return ret
263 
264     def discretiser(self, nom_attribut):
265         """
266             remplace chaque valeur de l'attribut en question par l'ensemble
267             auquel il appartient
268         """
269         #liste_intervalles va contenir la liste des intervalles
                résultant
270         #de la discrétisation
271         liste_intervalles = list()
272         #valeurs_triees est initialisée avec les valeurs de
273         #l'attribut de chaque exemple
274         valeurs_triees = \
275                 [(i,
                     self.liste_exemples[i].dict_attributs[nom_attribut])
                     \
276                                 for i in range(len(self))]
277         #et ensuite est triée selon ces valeurs
278         valeurs_triees.sort(key=itemgetter(1))
279         indice_borne_inf = 0
280         #on parcourt la liste triée
281         for i in range(1, len(valeurs_triees)):
282             #s'il y a un changement d'étiquette
283             if self.liste_exemples[valeurs_triees[i][0]].etiquette
                    !=  \
284 
                    self.liste_exemples[valeurs_triees[indice_borne_inf][0]].etique
285                 #les bornes (inf et sup) sont une moyenne de
286                 #l'élément courant et du précédent
287                 #ATTENTION : le premier intervalle commence à -8 car
                        il faut
288                 #pouvoir gérer toutes les valeurs
289                 if len(liste_intervalles) == 0:
290                     borne_inf = float("-inf")
291                 else:
292                     borne_inf =
                            (float(valeurs_triees[indice_borne_inf][1])
                            + \
```

```
293
                                        float(valeurs_triees[indice_borne_inf-1][1]
294                 borne_sup = (float(valeurs_triees[i][1]) + \
295                              float(valeurs_triees[i-1][1])) / 2
296                 #une fois les bornes trouvées, on ajoute
                        l'intervalle à la liste
297                 liste_intervalles.append((borne_inf, borne_sup))
298                 indice_borne_inf = i
299         #il ne faut pas oublier le dernier intervalle qui va jusqu'à
                +8 !
300         liste_intervalles.append((liste_intervalles[-1][1],
                float("+inf")))
301
302         #pour chaque exemple
303         for exemple in self.liste_exemples:
304             for intervalle in liste_intervalles:
305                 #on trouve le bon intervalle
306                 if float(exemple.dict_attributs[nom_attribut]) <
                        intervalle[1]:
307                     #en on discrétise
308                     exemple.dict_attributs[nom_attribut] =
                            intervalle
309                     break
310
311     def valeur_plus_frequente_attribut(self, nom_attribut):
312         """
313             renvoie un str avec la valeur la plus fréquente de
314             l'attribut en question dans l'ensemble
315         """
316         dict_frequences = dict()
317         #pour chaque exemple de l'ensemble
318         for exemple in self.liste_exemples:
319             #on regarde quelle est sa valeur pour l'attribut en
                    question
320             #si elle n'a pas encore été rencontrée
321             if exemple.dict_attributs[nom_attribut] not in
                    dict_frequences:
322                 #on l'ajoute au dictionnaire
323
                        dict_frequences[exemple.dict_attributs[nom_attribut]]
                        = 0
324             dict_frequences[exemple.dict_attributs[nom_attribut]] +=
                    1
325         #on retourne la clef pour laquelle la valeur est la plus
                élevée
326         return max(dict_frequences, key=dict_frequences.get)
327
328     def restaurer_valeurs_manquantes(self):
329         """
330             change chaque '?' dans les attributs, le remplacer par
```

```
331             la valeur de l'attribut la plus fréquente dans le set
332             (pour les exemples ayant la même étiquette)
333         """
334         #pour chaque exemple
335         for i in range(len(self.liste_exemples)):
336             #on check chaque chaque attribut
337             for nom_attribut in
                    self.liste_exemples[i].dict_attributs:
338                 #si l'attribut en question vaut '?'
339                 if
                        self.liste_exemples[i].dict_attributs[nom_attribut]
                        == '?':
340                     #on isole les éléments ayant la même étiquette
341                     sous_ensemble = self.sous_ensemble_etiquette(
342
                                                  self.liste_exemples[i].etiquette)
343                     #et on récupère la valeur de ce même attribut la
                            plus
344                     #fréquente pour l'assigner à la place du '?'
345
                        self.liste_exemples[i].dict_attributs[nom_attribut]
                        = \
346
                          sous_ensemble.valeur_plus_frequente_attribut(nom_attr
347
348 class Feuille:
349     """
350         Une feuille contient uniquement une valeur:
351             - l'étiquette (str)
352     """
353
354     def __init__(self, etiquette):
355         """
356             etiquette doit obligatoirement être un str
357         """
358         if not isinstance(etiquette, str):
359             raise
                    TypeError("etiquette doit être un str et pas un {}"
                    \
360                               .format(type(etiquette)))
361         self.etiquette = etiquette
362
363 class Noeud:
364     """
365         Un noeud a deux valeurs :
366             - un dictionnaire d'enfants (dict)
367             - l'attribut testé (str)
368     """
369
370     def __init__(self, attribut):
```

```
371         """
372             attribut_teste est le nom de l'attribut stocké dans un str
373         """
374         if not isinstance(attribut, str):
375             raise
                TypeError("attribute_teste doit être un str et pas un {}"
                \
376                           .format(type(attribut)))
377         #initialisation des valeurs de l'objet
378         self.enfants = dict()
379         self.attribut_teste = attribut
380 
381 class Arbre_ID3:
382     """
383         Un arbre ID3 contient deux valeurs :
384             - un ensemble d'exemples (Ensemble)
385             - un arbre (Noeud)
386     """
387 
388     def __init__(self, chemin=""):
389         """
390             chemin est l'emplacement du fichier contenant les données
391         """
392         #initialisation de l'ensemble avec le fichier dans chemin
393         self.ensemble = Ensemble(chemin)
394         #initialisation du noeud principal de l'arbre
395         self.arbre = None
396 
397     def construire(self):
398         """
399             génère l'arbre sur base de l'ensemble pré-chargé
400         """
401         self.arbre = self.__construire_arbre(self.ensemble)
402 
403     def __construire_arbre(self, ensemble):
404         """
405             fonction privée et récursive pour a génération de l'arbre
406         """
407         if not isinstance(ensemble, Ensemble):
408             raise
                TypeError("ensemble doit être un Ensemble et non un {}"
                \
409                           .format(type(ensemble)))
410         #si la liste est vide
411         if len(ensemble) == 0:
412             raise
                ValueError("la liste d'exemples ne peut être vide !")
413         #testons si tous les exemples ont la même étiquette
414         if ensemble.entropie() == 0:
415             #on retourne l'étiquette en question
```

```
416             return Feuille(ensemble.liste_exemples[0].etiquette)
417         #s'il ne reste d'attribut à tester
418         if len(ensemble.liste_attributs) == 0:
419             max, etiquette_finale = 0, ""
420             #on teste toutes les étiquettes possibles de l'ensemble
421             for etiquette in ensemble.etiquettes_possibles():
422                 sous_ensemble =
                        ensemble.sous_ensemble_etiquette(etiquette)
423                 #si c'est la plus fréquente, c'est celle qu'on
                        choisit
424                 if len(sous_ensemble) > max:
425                     max, etiquette_finale = len(sous_ensemble),
                            etiquette
426             #et on la retourne dans une feuille
427             return Feuille(etiquette_finale)
428 
429         a_tester = ensemble.attribut_optimal()
430         #si on arrive ici, on retourne d'office un nœud et pas une
                feuille
431         noeud = Noeud(a_tester)
432         #pour chaque valeur que peut prendre l'attribut à tester
433         for valeur in ensemble.valeurs_possibles_attribut(a_tester):
434             #on crée un sous-ensemble
435             sous_ensemble =
                    ensemble.sous_ensemble_attribut(a_tester, valeur)
436             #et on en crée un nouveau nœud
437             noeud.enfants[valeur] =
                    self.__construire_arbre(sous_ensemble)
438         #on retourne le nœud que l'on vient de créer
439         return noeud
440 
441     def afficher(self):
442         """
443             affiche l'entièreté de l'arbre à l'écran
444         """
445         self.__afficher_arbre(self.arbre)
446 
447     def __afficher_arbre(self, noeud, nb_tabs=0):
448         """
449             selon la convention :
450                 <texte> <-> nom de l'attribut
451                 -<texte> <-> valeur de l'attribut
452                 .<texte> <-> feuille
453         """
454         #si on a affaire à un nœud
455         if isinstance(noeud, Noeud):
456             #on affiche le nom de l'attribut testé
457             print('\t' * nb_tabs + noeud.attribut_teste)
458             #on parcourt ses enfants
459             for enfant in noeud.enfants:
```

```
460                 #on affiche la valeur de l'attribut
461                 print('\t' * nb_tabs + '-' + str(enfant))
462                 self.__afficher_arbre(noeud.enfants[enfant],
                        nb_tabs+1)
463         #si c'est une feuille
464         elif isinstance(noeud, Feuille):
465             #on affiche l'étiquette
466             print('\t' * nb_tabs + '.' + noeud.etiquette)
467         else:
468             raise
                TypeError("noeud doit être un Noeud/Feuille et pas un {}"
                \
469                           .format(type(noeud)))
470
471     def etiqueter(self, exemple):
472         """
473             assigne la bonne étiquette à l'exemple passé en paramètre
474         """
475         #on initialise le nœud actuel avec le haut de l'arbre
476         noeud_actuel = self.arbre
477         #tant que l'on est sur un nœud et pas sur une feuille,
478         #on continue d'explorer
479         while not isinstance(noeud_actuel, Feuille):
480             #pour savoir quel est le prochain nœud, on récupère
                    d'abord
481             #l'attribut testé avec noeud_actuel.attribut_teste puis
                    on récupère
482             #la valeur de l'exemple pour cet attribut avec
483             #exemple.dict_attributs[noeud_actuel.attribut_teste]
484             #puis on prend l'enfant de noeud_actuel ayant cette
                    valeur.
485             valeur =
                    exemple.dict_attributs[noeud_actuel.attribut_teste]
486             noeud_actuel = noeud_actuel.enfants[valeur]
487         #on finit en donnant comme étiquette l'étiquette
488         #contenue dans la feuille finale
489         exemple.etiquette = noeud_actuel.etiquette
490
491 class Arbre_C45(Arbre_ID3):
492     """
493         Un arbre C4.5 hérite d'un arbre ID3 mais se construit différemment
494     """
495
496     def __init__(self, chemin_données="", chemin_elagage=""):
497         """
498             chemin_données est l'emplacement du fichier contenant les données
499             chemin_elagage est l'emplacement d'un autre set permettant
500             d'élaguer l'arbre
501         """
```

```
502         #initialisation de l'ensemble avec le fichier dans
                chemin_données
503         self.ensemble = Ensemble(chemin_données)
504         #initialisation du nœud principal de l'arbre
505         self.arbre = None
506         self.chemin_elagage = chemin_elagage
507 
508     def construire(self):
509         """
510             retourne l'arbre au complet
511         """
512         #si le set est corrompu (attributs manquants), on le
                restaure
513         self.ensemble.restaurer_valeurs_manquantes()
514         self.arbre = self.__construire_arbre(self.ensemble)
515 
516     def __construire_arbre(self, ensemble):
517         if not isinstance(ensemble, Ensemble):
518             raise
                    TypeError("ensemble doit être un Ensemble et non un {}"
                    \
519                               .format(type(ensemble)))
520         #si la liste est vide
521         if len(ensemble) == 0:
522             raise
                    ValueError("la liste d'exemples ne peut être vide !")
523         #testons si tous les exemples ont la même étiquette
524         if ensemble.entropie() == 0:
525             #on retourne l'étiquette en question
526             return Feuille(ensemble.liste_exemples[0].etiquette)
527         #s'il ne reste d'attribut à tester
528         if len(ensemble.liste_attributs) == 0:
529             max, etiquette_finale = 0, ""
530             #on teste toutes les étiquettes possibles de l'ensemble
531             for etiquette in ensemble.etiquettes_possibles():
532                 sous_ensemble =
                        ensemble.sous_ensemble_etiquette(etiquette)
533                 #si c'est la plus fréquente, c'est celle qu'on
                        choisit
534                 if len(sous_ensemble) > max:
535                     max, etiquette_finale = len(sous_ensemble),
                            etiquette
536             #et on la retourne dans une feuille
537             return Feuille(etiquette_finale)
538 
539         #ne pas oublier de sauver les valeurs pour pouvoir les
                restituer au cas
540         #où l'attribut discrétisé n'est pas choisi
541         sauvegarde_valeurs =
                ensemble.sauvegarder_valeurs_discretes()
```

```
542         #pour chaque valeur à discrétiser
543         for attribut, valeurs in sauvegarde_valeurs:
544             #on discrétise
545             ensemble.discretiser(attribut)
546         #on récupère l'attribut optimal
547         #ATTENTION : préciser ID3=False pour utiliser le ratio de
                gain
548         a_tester = ensemble.attribut_optimal(ID3=False)
549         #pour chaque attribut sauvegardé
550         for attribut, valeurs in sauvegarde_valeurs:
551             #si ce n'est pas l'attribut choisi
552             if attribut != a_tester:
553                 #on remet les anciennes valeurs continues
554                 for i in range(len(valeurs)):
555 
                            ensemble.liste_exemples[i].dict_attributs[attribut]
                            = \
556 
                                                                          valeurs
557         #si on arrive ici, on retourne d'office un nœud et pas une
                feuille
558         noeud = Noeud(a_tester)
559         #pour chaque valeur que peut prendre l'attribut à tester
560         for valeur in ensemble.valeurs_possibles_attribut(a_tester):
561             #on crée un sous-ensemble
562             sous_ensemble =
                    ensemble.sous_ensemble_attribut(a_tester, valeur)
563             #et on en crée un nouveau nœud
564             noeud.enfants[valeur] =
                    self.__construire_arbre(sous_ensemble)
565         #on retourne le nœud que l'on vient de créer
566         return noeud
567 
568     def etiqueter(self, exemple):
569         #on initialise le nœud actuel avec le haut de l'arbre
570         noeud_actuel = self.arbre
571         #tant que l'on est sur un nœud et pas sur une feuille
572         while isinstance(noeud_actuel, Noeud):
573             #valeur == valeur de l'exemple à étiqueter pour
                    l'attribut du nœud
574             valeur =
                    exemple.dict_attributs[noeud_actuel.attribut_teste]
575             #si valeur représente un nombre
576             try:
577                 valeur = float(valeur)
578             #si ça ne marche pas, tout va bien : c'est une valeur
                    discrète
579             except:
580                 pass
```

```
581             #si c'est une valeur continue, on la transforme en
                    intervalle
582             else:
583                 for intervalle in noeud_actuel.enfants:
584                     if valeur < intervalle[1] and valeur >=
                            intervalle[0]:
585                         valeur = intervalle
586                         break
587             finally:
588                 #mais il faut bien faire avancer le nœud
589                 noeud_actuel = noeud_actuel.enfants[valeur]
590         #une fois l'exploration terminée, on étiquette l'exemple
591         exemple.etiquette = noeud_actuel.etiquette
592 
593     def taux_erreur(self, ensemble):
594         """
595             renvoie un nombre dans [0, 1] correspondant à la proportion
596             d'exemples dans ensemble qui se font étiqueter correctement
597             avec l'arbre tel quel
598         """
599         compteur_etiquetages_incorrects = 0
600         #pour chaque exemple
601         for exemple in ensemble.liste_exemples:
602             #on garde son étiquette
603             etiquette = exemple.etiquette
604             self.etiqueter(exemple)
605             #et on la compare à l'étiquette donnée par l'arbre
606             if etiquette != exemple.etiquette:
607                 #si elles sont différentes, on augmente la
                        proportion
608                 #d'étiquetages incorrects et on rétablit la bonne
                        étiquette
609                 compteur_etiquetages_incorrects += 1
610                 exemple.etiquette = etiquette
611         return compteur_etiquetages_incorrects/len(ensemble)
612 
613     def elaguer(self):
614         """
615             modifie l'arbre en élaguant suivant l'ensemble de travail
616             d'élagage donné dans self.chemin_elagage
617         """
618         #ATTENTION : si le chemin n'a pas été donné, on n'élague pas
                !
619         if self.chemin_elagage != "":
620             self.arbre = self.__elaguer_noeud(self.arbre,
621 
                                                   Ensemble(self.chemin_elagage))
622 
623     def __elaguer_noeud(self, noeud, ensemble_elagage):
624         """
```

```
625             élague le noeud passé en paramètre et le retourne
626         """
627         #si on est sur une feuille, on ne va pas plus loin
628         if isinstance(noeud, Feuille):
629             return noeud
630         min_erreur, etiquette_gardee = 1.0, ""
631         proportion_initiale = self.taux_erreur(ensemble_elagage)
632         sauvegarde = self.arbre
633         #pour chaque étiquette
634         for etiquette in ensemble_elagage.etiquettes_possibles():
635             self.arbre = Feuille(etiquette)
636             #on calcule le taux d'erreur si on remplace le nœud par
637             #l'étiquette en question
638             taux_erreur_actuel = self.taux_erreur(ensemble_elagage)
639             #on sauvegarde le meilleur taux
640             if taux_erreur_actuel < min_erreur:
641                 min_erreur, etiquette_gardee = taux_erreur_actuel,
                        etiquette
642         #s'il existe un taux avantageux on élague à cet endroit
643         if min_erreur <= proportion_initiale:
644             return Feuille(etiquette_gardee)
645         else:
646             self.arbre = sauvegarde
647         #on n'oublie pas de restaurer la valeur du sommet de l'arbre
              !
648         sauvegarde, self.arbre = self.arbre, sauvegarde
649         #on teste chaque enfant pour voir s'il est élagable
650         for enfant in noeud.enfants:
651             sous_ensemble = ensemble_elagage.sous_ensemble_attribut(
652
                                                          noeud.attribut_teste,
653                                                     enfant)
654             #s'il l'est, on l'élague
655             if len(sous_ensemble) != 0:
656                 noeud.enfants[enfant] = self.__elaguer_noeud(
657
                                                          noeud.enfants[enfant],
658                                                     sous_ensemble)
659         #et au final, on renvoie le nœud aux enfants peut-être
              élagués
660         return noeud
661
662 def exemple_utilisation():
663     """
664         exemples d'utilisation
665     """
666     with open("datas PlayTennis.txt") as tmp:
667
              print("Exemple d'arbre avec ID3 sur le fichier datas PlayTennis.txt
                "\n\n{}\n\n" .format("".join(tmp.readlines())))
668
```

```
669     arbre = Arbre_ID3("datas PlayTennis.txt")
670     arbre.construire()
671     arbre.afficher()
672     exemple = Exemple(["outlook", "temperature", "humidity",
            "wind"],
673                       ["sunny",   "cool",        "normal",
                              "strong"])
674     print("etiquette : '{}'".format(exemple.etiquette))
675     arbre.etiqueter(exemple)
676     print("etiquette : '{}'\n\n".format(exemple.etiquette))
677 
678     print('-' * 35)
679 
680     with open("datas continues.txt") as tmp:
681         print("\n\nExemple d'arbre avec C4.5 sur le fichier "
682               "'datas continues.txt' :\n\n{}\n\n" \
683               .format("".join(tmp.readlines())))
684     arbre = Arbre_C45("datas continues.txt")
685     arbre.construire()
686     arbre.afficher()
687     exemple = Exemple(["outlook", "temperature", "humidity",
            "wind"],
688                       ["sunny",   "cool",        "60",   "strong"])
689     print("etiquette : '{}'".format(exemple.etiquette))
690     arbre.etiqueter(exemple)
691     print("etiquette : '{}'".format(exemple.etiquette))
692 
693     print('-' * 35)
694 
695     with open("datas manquantes.txt") as tmp:
696         print("\n\nExemple d'arbre avec C4.5 sur le fichier " \
697               "datas manquantes.txt :\n\n{}\n\n" \
698               .format("".join(tmp.readlines())))
699     arbre = Arbre_C45("datas manquantes.txt")
700     arbre.construire()
701     arbre.afficher()
702     exemple = Exemple(["outlook", "temperature", "humidity",
            "wind"],
703                       ["sunny",   "cool",        "normal",
                              "strong"])
704     print("etiquette : '{}'".format(exemple.etiquette))
705     arbre.etiqueter(exemple)
706     print("etiquette : '{}'".format(exemple.etiquette))
707 
708     with open("datas PlayTennis.txt") as tmp:
709         print("\n\nExemple d'arbre avec C4.5 sur le fichier " \
710               "datas PlayTennis.txt :\n\n{}\n\n" \
711               .format("".join(tmp.readlines())))
712     arbre = Arbre_C45("datas PlayTennis.txt", "datas élagage.txt")
713     arbre.construire()
```

```
714     arbre.afficher()
715     print("\n\nÉlagage de l'arbre\n\n")
716     arbre.elaguer()
717     arbre.afficher()
718     exemple = Exemple(["outlook", "temperature", "humidity",
            "wind"],
719                       ["sunny",   "cool",        "normal",
                              "strong"])
720     print("etiquette : '{}'".format(exemple.etiquette))
721     arbre.etiqueter(exemple)
722     print("etiquette : '{}'".format(exemple.etiquette))
723
724 if __name__ == "__main__":
725     exemple_utilisation()
```

Retourner au texte.

Contenu masqué n°3

Comme promis, voici le dataset qui vous permet de faire fonctionner l'arbre ID3 sur les données du *Qui est-ce ?*. Je vous le donne uniquement parce que je m'étais beaucoup amusé à tricher contre ma petite soeur qui ne comprenait pas comment je faisais pour gagner sans même regarder la plaque . Le voici donc :

```
1  sexe couleur_pilosite est_euro couleur_yeux lunettes chapeau chauve
       moustache barbe
2  g noir non brun non non non non non frank
3  f marron oui brun non non non non non holly
4  g blond oui brun non non non oui non hans
5  g roux oui bleu non non non oui non alfred
6  f roux oui brun oui oui non non non sarah
7  g marron oui brun non oui non non non bernard
8  g roux oui brun non non oui non non herman
9  f blanc oui bleu oui non non non non betty
10 g roux oui brun non non oui non oui bill
11 g roux oui brun non non non non non frederick
12 g blanc oui brun non non non non non victor
13 g blanc oui brun oui non oui oui non charles
14 g blanc oui brun oui non non non non paul
15 g blond oui brun non non non oui oui luk
16 g marron oui bleu non non non non non robert
17 g blond oui brun non oui non non non eric
18 f marron oui brun non oui non non non maria
19 g blanc oui brun non oui non non non joe
20 g noir oui bleu oui non oui non non albert
21 g noir non brun non non non non oui mario
22 g noir non brun non non non oui non max
```

```
f blond oui bleu non non non non non anita
f noir non brun oui non non non non sally
g marron oui brun non non oui oui oui roger
```

Retourner au texte.

Liste des abréviations

IA Intelligence Artificielle. 2, 3