INFO F-103, Algorithmique
Rapport projet 2

ROCCA Manuel

May 7, 2024

Contents

1 Introduction

Les données et leur analyse possède une place prépondérante dans notre société actuelle. Tout naturellement, nous cherchons à les classer, indexer, etc... Diverses techniques existent pour arriver à nos fin. C'est pourquoi, dans le cadre de ce projet d'algorithmique, nous nous sommes intéressés aux arbres de décision afin de classer un ensemble de plus de 8000 champignons.
L'arbre de décision implémenté se sert du calcul de l'entropie ainsi que du gain d'information pour déterminer l'impureté d'un ensemble et de le diviser efficacement. Il sera représenté à l'aide d'objets 'Node' correspondant aux noeuds d'un arbre ainsi que d'objets 'Edge', représentant les branches de l'arbre. De plus, nous allons parcourir cette arbre de diverses manières. Pour l'affichage, nous utiliserons un parcours préfixé exhaustif, tandis que pour vérifier si un champignon est comestible ou non, nous parcourerons l'abre sur base de attributs de ce champignon.
Un ensemble de tests furent également demandés afin de vérifier certains cas imprévus lors de l'analyse d'un ensemble de données quelconques. Ces tests porterons sur la construction de l'arbre, son parcours ainsi que sa transformation en règles booléennes.

2 Fonctionnement général

Nous allons ici décrire plus précisément comment est implémenté cet arbre.

2.1 Construction de l'arbre

L'arbre est construit en se basant sur un pseudo-code donné dans les consignes (cf. consignes), de manière récursive:

1. Choisir l'attribut A qui sépare au mieux les champignons.
2. Créer le noeud associé r.
3. Pour chaque valeur possible de cet attribut A:
   (a) Construire le sous-arbre $T_A = v$ sur base des champignons ayant la valeur v pour leur attribut A.
   (b) Ajouter la racine de $T_A = v$ aux enfants de r.

Le point 1 se fait à l'aide de la fonction *get_info_gain(attribute_values: dict, parent_entropy: int, total_mushrooms: int) -> int* qui se charge de calculer le gain d'information pour un attribut. Le plus grand gain d'information sera ainsi retenu que l'attribut associé. En suivant le pseudo-code donné, nous créons ensuite un premier noeud comportant le premier attribut choisi comme racine de notre arbre. Précédemment, la fonction *get_attribute_values(attribute: str, mushrooms: list[Mushroom]) -> dict* nous a permis de récuperer, pour valeur de l'attribut choisi, un ensemble de champignons correspondant. Toujours en suivant le pseudo-code, nous ferons donc un appel récursif sur chaque sous-ensemble de champignon sur la fonction de construction de l'arbre pour créer des noeuds enfants qu'on ajoutera à la racine à l'aide d'une branche.

La formule du gain d'information utilisée est définie par:

$$I(C|A) = H(C) - \sum_v p_A = v \cdot H(C_A = v)$$

2.2 Parcours de l'arbre

Nous allons ici adresser les parcours principaux, à savoir celui de la fonction *is_edible(root: Node, mushroom: Mushroom) -> bool* et celui de *display(tree: Node, indent = 0) -> None*.

2.2.1 Fonction: is_ebible

Cette fonction prend l'arbre construit en paramètre ainsi qu'un champignon avec certains attributs quelconques sauf l'attribut edible. Nous cherchons, dans toutes les branches de la racine, laquelle correspond au champignon donné. Une fois cette branche trouvée, nous faisons un appel récursif avec comme arbre, le fils de cette branche. Les cas de bases sont si l'attribut de l'arbre sont "Yes" ou "No", à savoir les informations sur la comestibilité du champignon.

2.2.2 Fonction: display

Cette fonction utilise un parcours préfixé, comme expliqué au cours théorique et aux travaux pratiques sur les arbres. Nous affichons donc d'abord la racine avant de parcourir chaque fils, de gauche à droite, récursivement. Le cas de base vérifie si l'arbre donné en paramètre est une feuille ou non, à l'aide de la méthode *is_leaf* de la classe *Node*.

2.3 Autres fonctions

2.3.1 Fonction: bool_tree

La fonction *bool_tree(tree: Node) -> str* transforme l'arbre en un ensemble de règles booléennes sous forme de chaîne de caractères. Nous n'entrerons pas dans ici dans les détails de la construction de la réponse car elle se fait sur base d'un parcours similaire aux deux autres cités ci-dessus.2.2

2.3.2 Fonction: to_python

La fonction *to_python(dt: Node, path: str) -> None* retranscrit l'arbre en un ensemble de conditions de if/else dans un fichier python. L'écriture est faite localement dans la fonction mais l'élément écrit est récupéré via la fonction *write_python(tree : Node, f, indent = 4, ret = ") -> str*. Cette dernière parcourt l'arbre une fois de plus de manière similaire aux fonctions précédemment expliquées.2.2

3 Complexité

3.1 Temporelle

La complexité temporelle n'est ici que peu intéressante. Cela est dû au fait que le programme se compose essentiellement de boucles *for* qui vont itérer dans un certain ensemble. Nous soulignons quand même les nombreux appels récursifs qui impliquent une complexité temporelle plutôt élevée.

3.2 Spatiale

La complexité spatiale, en revanche, s'avère plutôt intéressante car élevée. Les appels récursifs impliquent évidemment une utilisation importante du stack et le grand nombre d'objets *Mushroom* possédant chacun plus de 20 attriubts utilisent une quantité non négligeable de mémoire. De plus, l'arbre est stocké sous forme d'objets *Node* et *Edge* qui augmentent encore la quantité de mémoire requise par ce programme.

4 Tests personnalisés

4.1 Partie 1

Les tests de la partie 1 portent sur la construction de l'arbre et du chargement des données à partir du fichier csv. En revanche, ces deux aspects sont déjà évalués dans les tests fournis. C'est pourquoi nos tests vérifient d'autres parties plus spécifiques.

4.1.1 Test: test_attribute_values

Ce test vérifie le bon fonctionnement de la fonction *get_attribute_values(attribute: str, mushrooms: list[Mushroom]) -> dict*
, expliquée au point 2.1 Cette vérification est faite à l'aide d'une fonction *get_values_of_attribute(mushrooms, attribute : str)* qui va récupérer chaque valeur pour un attribut donné sous forme d'une liste de *string* pour les comparer aux clés du dictionnaire donné par *get_attribute_values*.

4.1.2 Test: test_information_gain

Nous vérifions ici si, pour un ensemble donné, la valeur de gain d'information calculée par la fonction *get_info_gain (attribute_values: dict, parent_entropy: int, total_mushrooms: int) -> int* est correcte. Les valeurs auxquelles nous comparons les résultats des appels de fonction sont tout naturellement calculés à la main.

4.2 Partie 2

Les deux tests que nous proposons pour la phase 2 vérifient un parcours de l'arbre interactif ajouté en plus des choses demandées dans les consignes. Nous nous sommes permis d'importer, du module *unittest*, *mock*. Ce sous-module permet, grâce à *mock.patch*, de simuler des input lors du test du programme. Nous testons donc simplement la validité d'un chemin.
Il est tout naturel que nous nous sommes renseignés au sujet de ce module. Voici les sources utilisées:

- [Documentation patch]
- [Documentation mock]

4.3 Arbre booléen

La dernière série de test se focalise sur l'arbre de décision booléen. Ce dernier étant un *string*, nous avons donc dû nous adapter. Autrement dit, les tests diffèrent des autres.

4.3.1 Test: test_right_attributes_in_bool_tree

Ce test va comparer le résultat de deux fonctions, *tree_attributes_to_set(tree, res = [])* et *bool_string_to_set(bool_tree)*. Les deux fonctions vont retourner un ensemble de toutes les valeurs d'attributs présents dans l'arbre classique et l'arbre booléen respectivement.

4.3.2 Test: test_bool_tree_string

Cette fonction nous permet de vérifier de choses simples dans l'arbre booléen. Par exemple, nous vérifions si chaque paranthèse ouvrante possède une paranthèse fermante correspondante, ou encore si un "*Attribut = valeur*" recherché est bien présent dans le *string*.

```
Mushroom decision tree's boolean expression:

(odor = Almond) OR (odor = Anise) OR (odor = None AND (spore-print-color = Brown) OR (spore-print-color = Black) OR (spore-print-color = Chocolate) OR
(spore-print-color = Orange) OR (spore-print-color = Yellow) OR (spore-print-color = Buff) OR (spore-print-color = White AND (habitat = Waste) OR (habi
tat = Grasses) OR (habitat = Paths) OR (habitat = Leaves AND (cap-color = Cinnamon) OR (cap-color = Brown)) OR (habitat = Woods AND (gill-size = Broad)
)))
```

Figure 1: L'arbre booléen dans un terminal.