TP 20: Quels sont les champignons toxiques ?

MP2I Descartes, Tours - Vladislav Tempez

5 juillet 2023

L'objectif de ce TP est d'implémenter l'algorithme ID3 et de l'utiliser pour construire un arbre de décision classifiant les champignons en deux catégories : comestible ou toxique.

1 Préparation des données

Le jeu de données provient de `https://archive.ics.uci.edu/ml/datasets/mushroom`. Les champignons y sont décrits par 22 caractéristiques (features dans le code) dont l'intitulé est le suivant :

1. cap-shape
2. cap-surface
3. cap-color
4. bruises ?
5. odor
6. gill-attachment
7. gill-spacing
8. gill-size
9. gill-color
10. stalk-shape
11. stalk-root
12. stalk-surface-above-ring
13. stalk-surface-below-ring
14. stalk-color-above-ring
15. stalk-color-below-ring
16. veil-type
17. veil-color
18. ring-number
19. ring-type
20. spore-print-color
21. population
22. habitat

Ces caractéristiques correspondent principalement à des aspects de la physiologie du champignon (comment se présente le chapeau, les lamelles, la tige, etc) et sont toutes catégorielles (elles prennent un nombre fini de valeurs). Le détail des valeurs prises est indiqué dans le fichier `mushroom_signature.data`. Les données elles-mêmes sont contenues dans le fichier `mushroom.csv` sous la forme d'une ligne par champignon, contenant, séparées par des virgules, la classe (e pour edible et p pour poisonous) suivie des différentes valeurs de chacun des features. Chaque valeur est résumée en une seule lettre, la correspondant est dans le fichier `.data`.

1.1 Lecture des fichiers de données

Le code permettant de lire le fichier de données et de construire une représentation de ces données en mémoire est fourni dans les fichiers `data_read.c` et `data_read.h`. Le fichier `data_read.h` contient en particulier la signature des structures de données permettant de manipuler :

— La signature d'une donnée, qui indique le nombre des features, l'ensemble des noms de features (sous forme d'un tableau), l'ensemble des valeurs possibles pour ces features (sous forme d'un tableau), le nombre de valeurs possible pour chaque feature, l'ensemble des classes (sous forme d'un tableau) et le nombre de classes. C'est cette signature qui permet d'interpréter correction les structures suivantes.

— Un point de l'ensemble d'apprentissage, correspondant à un champignon via la structure `data_value`. Cette structure contient un identifiant, un tableau des valeurs prises pour chaque feature et la classe du champignon. Les valeurs et la classe ne sont pas représentées directement par la lettre présente dans le fichier `.csv` mais par le numéro de cette valeur (dans l'ordre de lecture du fichier `data`) La correspondance numéro valeur peut être faite via la signature.

— Un data set qui contient un tableau de pointeurs vers les différents points qu'il contient, le nombre de points et un pointeur vers la signature des données. Attention, comme ce data set ne stocke par directement les points mais des pointeurs vers ceux-ci, les points peuvent être partagés par plusieurs data set et il convient de faire attention à la libération de ceux-ci.

En plus de ces définitions de type, le fichier `read_data.c` contient une fonction `data_signature_t* read_data_signature(char* signature_filename)` qui initialise une signature à l'aide du contenu du fichier `.data`. Celui-ci contient, ligne par ligne et après les lignes de commentaires (qui commencent par `//`) :

— le nombre de features

— pour chaque feature le nom de celle-ci et le nombre de valeurs possibles

— suivant le nom d'une feature, une ligne par valeur possible

— après la liste des features le nombre de classe

— le nom de chacune des classes, à raison d'un par ligne

On trouve aussi une fonction `data_set_t* read_data(char* data_filename, data_signature_t* signature)` qui initialise un data set à partir du contenu d'un fichier `.csv` donné en argument et de la signature de ces données. Sont aussi implémentées deux fonctions de libération de la mémoire réservée par les structures de data set (sauf la signature) et de signature.

1.2 Manipulation des data sets

1. Pour pouvoir évaluer le modèle appris à l'avenir, on souhaite pouvoir séparer un ensemble de données en deux, avec d'un côté les données utilisées pour construire le modèle et de l'autre celles utilisées pour évaluer le modèle. Implémentez une fonction `void random_split_data(data_set_t* initial_data_set, float fraction, data_set_t* training_set, data_set_t* validation_set)` qui initialise `training_set` et `validation_set` avec les données de `initial_data_set` et en répartissant aléatoirement les points du data set initial entre les deux nouveaux data sets. `fraction` indiquera la proportion du data set initial qui sera allouée pour l'ensemble d'apprentissage. Il n'est pas nécessaire de réaliser des copies de tous les points puisqu'un data set contient un tableau de pointeurs vers ces points. `training_set` et `validation_set` auront été préalablement obtenus via un `malloc` réservant la bonne quantité de mémoire.
2. (Bonus) Implémentez une fonction de calcul approché du log en exploitant la relation $\ln(x) = \ln(\frac{x}{e}) + 1 = \ln(e \times x) - 1$ et le développement limité de $\ln(1+x)$ en 0. N'oubliez pas de tester cette fonction.
3. Implémentez une fonction `double compute_entropy(data_set_t* d)` qui calcule l'entropie de l'ensemble passé en argument. Vous pourrez utiliser la fonction `log` via un `#include <math.h>`. Attention, l'utilisation de `<math.h>` nécessite d'ajouter `-lm` à la fin de la commande de compilation.
4. Pour pouvoir calculer le gain d'information lié à un attribut, il est nécessaire de pouvoir séparer un ensemble en fonction des valeurs que prend cet attribut (feature). Implémentez une fonction `data_set_t** split_data_set(data_set_t* initial_data_set, int feature_id)` qui renvoie un tableau de pointeurs vers des datasets correspondant à une partition du data set initial selon les valeurs de la feature dont l'id est donné en argument. Là encore, pas besoin de copier pleinement les points.
5. Implémentez une fonction `compute_information_gain(data_set_t* d, int feature_id)` qui calcule le gain d'information obtenu avec une partition selon les valeurs prises par `feature_id`.
6. Implémentez une fonction `int best_information_gain(data_set_t* d, bool* unused_features)` qui cherche la feature qui maximise le gain d'information parmi celles qui n'ont pas déjà été utilisées. S'il n'est pas possible de trouver une feature qui maximise le gain d'information, cette fonction renverra `-1`.

2 Algorithme ID3

Pour construire un arbre de décision, on choisit d'utiliser la structure suivante :

```
1 struct decision_tree_s {struct decision_tree_s** children; char* children_feature_value; data_set_t* data;
  ↪  int feature_id; char* feature_name; bool is_leaf; int predicted_class; char class_name;};
2 typedef struct decision_tree_s decision_tree_t;
```

Cet enregistrement permet de représenter des arbres. Le champ `children` est un tableau de pointeurs vers d'autres arbres. Le champ `children_feature_value` établit une correspondance entre les enfants et la valeur de la feature sur laquelle ils ont été sélectionnés. Le champ `data_set` correspond à un data set contenant tous les points qui sont aux feuilles de cet arbre. Le champ `feature_id` est l'identifiant de la feature utilisée dans ce nœud pour répartir les points entre ses différents fils. `feature_name` est une chaîne de caractère qui est le nom de la feature dont l'id est `feature_id`. `is_leaf` indique si l'arbre courant est une feuille. `predicted_class` indique l'identifiant de la classe prédite pour l'arbre courant. C'est un champs qui n'est valide que si l'arbre courant est une feuille. `class_name` est un caractère correspondant à l'identifiant de la classe prédite. Quand le champ `is_leaf` est faux, `predicted_class`, `class_name` ne devront pas être accédés. Dans le cas contraire `children` devra valoir `NULL`, de même pour `children_feature_value`. Ainsi, `t->children[i]` contiendra tous les éléments de `t->data` pour lequel l'attribut d'identifiant `t->feature_id` vaut la $i-ième$ valeur parmi toutes celles possibles pour cet attribut.

7. Implémentez une fonction `decision_tree_t* build_id3_tree(data_set_t* data_set)` qui construit un arbre de décision pour le data set donné en argument via l'algorithme ID3.
8. Implémentez une fonction qui libère la mémoire réservée pour un tel arbre. Cette fonction ne libérera pas les points contenus dans les ensembles, uniquement les ensembles eux-mêmes, s'ils ne sont pas ceux de la racine.
9. Implémentez une fonction `int predict_class(decision_tree_t* id3_tree, data_value_t* input)` qui prédit la classe pour une entrée donnée à l'aide d'un arbre de décision.
10. Implémentez une fonction `double compute_error(decision_tree_t* id3_tree, data_set_t* testing_set)` qui calcule l'erreur commise par l'arbre de décision sur un data set. Comment se comporte l'arbre que vous avez obtenu plus tôt du point de vue de l'erreur ?

Attention !

N'utilisez pas directement ce modèle pour déterminer si vous pouvez manger de véritables champignons que vous auriez ramassés !

11. (Bonus) Implémentez une fonction `double* compute_confusion_matrix(decision_tree_t* id3_tree, data_set_t* testing_set)` qui calcule une matrice de confusion liée à un arbre de décision sur un data set passé en argument.
12. Estimez la fiabilité de l'arbre de décision appris pour la classification des champignons toxiques. Qu'en déduisez-vous ?
13. (Bonus) Affichez l'arbre de décision obtenu.