INPF12 : projet 2023-2024

Tom Gilgenkrantz

2 mai 2024

1 Les arbres de décisions

1.1 Introduction

Un arbre de décision est un outil de prise de décision sous forme d'arbre avec des nœuds, des feuilles (nœuds terminals) et une profondeur. Il est largement utilisé en intelligence artificielle et en apprentissage automatique pour résoudre des problèmes de classification et de régression. Chaque nœud interne de l'arbre correspond à un test sur une caractéristique, chaque branche représente le résultat du test et chaque feuille représente une prédiction de classe.

On s'interessera ici à la construction d'un arbre de décision pour la classification de points colorés (rouges ou bleus) d'un dataset en deux classes. On commence tout d'abord par définir un type point qui est un couple de flottants choisis ici dans $[\![0, 1]\!]^2$.

```
type pos = { x:float ; y:float; } ;;
```

On définit ensuite un type couleur qui est un triplet d'entiers de $[\![0, 255]\!]^3$.

```
type color = {r:int ; g:int ; b:int} ;;
```

On peut enfin définir le type dataset qui est un liste de coordonnées colorées.

```
type 'a dataset = (pos*color) list;;
```

1.2 Type tree

Avant de pouvoir construire un arbre de décision, il est nécessaire de déterminer comment les tests seront effectués afin de séparer le dataset initial en sous-ensembles. L'idée la plus simple est tout d'abord de couper au milieu du dataset alternativement verticalement et horizontalement afin de déterminer 2 sous-ensembles de ce dernier. On pourrait ainsi définir le tree comme un arbre binaire avec pour noeuds des conditions sous forme de flottants de $[\![0, 1]\!]$, et deux sous-arbres, l'un avec une coordonnée inférieure au noeud parent, l'autre avec une coordonée supérieure ou égale.

```
type tree =
  | Leaf of color
  | Node of float * tree * tree ;;
```

Ce n'est pourtant pas ce type que j'ai choisi de garder. En choisissant de séparer le dataset en 4 sous-ensembles, on peut définir un arbre de décision comme un arbre avec pour noeuds des conditions sous forme de couples de flottants de $[\![0, 1]\!]^2$, et 4 sous-arbres, chacun correspondant à un sous-ensemble du dataset initial. Cette implémentation permet de beaucoup simplifier l'implémentation puisqu'il n'est plus nécessaire d'alterner entre coupe horizontale et verticale. On peut donc définir le type tree comme suit :

```
type tree =
  | Leaf of color
  | Node of pos * tree * tree * tree * tree ;;
```

On définit enfin des couleurs de base pour les points rouges et bleus.

```
let red = {r=255 ; g=0 ; b=0} ;;
let blue = {r=0 ; g=0 ; b=255} ;;
```

2 Fonctions auxiliaires

Avant de commencer à implémenter la construction de l'arbre de décision, il est nécessaire de définir quelques fonctions auxiliaires qui seront utiles pour la suite. On commence par définir une fonction qui permet de compter le nombre de points rouges et bleus dans un dataset. Cela sera particulièrement utile pour déterminer la couleur moyenne d'un sous-ensemble de points, ou simplement vérifier qu'un dataset n'est pas vide ou d'une couleur unique.

```
(* ('a * color) list -> int -> int -> int * int *)
let rec compter_couleurs training rouge bleu =
    match training with
    |[] -> rouge , bleu
    |head::tail ->
        if (snd head) = red
            then compter_couleurs tail (rouge+1) bleu
        else
            compter_couleurs tail rouge (bleu+1) ;;
```

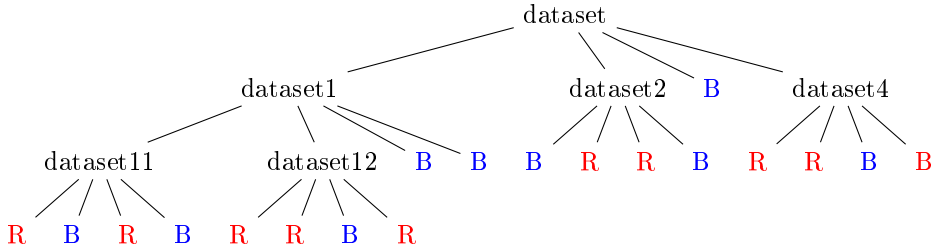
On définit ensuite une fonction qui permet de séparer un dataset en 4 sous-ensembles en fonction d'une pos donnée. On peut alors définir la fonction suivante :

```
(* (pos * 'a) list ->
   (pos * 'a) list ->
```

```
    (pos * 'a) list ->
    (pos * 'a) list ->
    (pos * 'a) list ->
    pos ->
    (pos * 'a) list * (pos * 'a) list * (pos * 'a)
       list * (pos * 'a) list *)
let rec separer_dataset d1 d2 d3 d4 training x_y =
  match training with
  |[] -> d1 , d2 , d3 , d4
  |head::tail ->
    if (fst head).x < x_y.x && (fst head).y >= x_y.y
      then separer_dataset (head::d1) d2 d3 d4 tail
         x_y
    else if (fst head).x >= x_y.x && (fst head).y >=
       x_y.y
      then separer_dataset d1 (head::d2) d3 d4 tail
         x_y
    else if (fst head).x < x_y.x && (fst head).y <
       x_y.y
      then separer_dataset d1 d2 (head::d3) d4 tail
         x_y
    else
      separer_dataset d1 d2 d3 (head::d4) tail x_y ;;
```

3 Construction de l'arbre de décision

On peut maintenant commencer à construire l'arbre de décision. On se sert d'une fonction récursive qui permet de construire un arbre de décision à partir d'un dataset. On commence par vérifier si le dataset a une taille inférieure à 1, d'une seule couleur ou bien si on a atteint la profondeur maximale choisie, ou si tous les points sont de la même couleur. Dans ce cas, on renvoie une feuille de la couleur moyenne des points. Sinon, on sépare le dataset en 4 sous-ensembles, et on construit récursivement les sous-arbres. Voici un exemple d'arbre de décision de profondeur 4 tel que défini ci-dessus :




```
(* int -> 'a dataset -> tree *)
let build_tree prof training =
```

```
let rec build_node profondeur dataset separations =
  let rouge , bleu = compter_couleurs dataset 0 0 in
  if profondeur >= prof + 1 || rouge = 0 || bleu = 0 || rouge + bleu < 2
   then if rouge > bleu then Leaf red else Leaf blue
  else
    let epsilon = 1.0 /. 2.0 ** (float_of_int profondeur) in
    let d1 , d2 , d3 , d4 = separer_dataset [] [] [] [] dataset separations in
    Node (separations,
          build_node (profondeur+1) d1 {x=(separations.x -. epsilon); y=(separations.y +. epsilon)},
          build_node (profondeur+1) d2 {x=(separations.x +. epsilon); y=(separations.y +. epsilon)},
          build_node (profondeur+1) d3 {x=(separations.x -. epsilon); y=(separations.y -. epsilon)},
          build_node (profondeur+1) d4 {x=(separations.x +. epsilon); y=(separations.y -. epsilon)}) in
build_node 0 training {x=0.5; y=0.5} ;;
```

Le calcul de la prochaine découpe est effectué en fonction de la profondeur de l'arbre, en prenant les séparations actuelles et en prenant les 4 nouveaux couples de flottants crées en ajoutant ou enlevant $2^{-profondeur}$ à la coordonnée x ou y, par exemple les coordonées de la découpe de profondeur 3 seront crée à partir de la découpe de profondeur 2 $(0.5, 0.5)$ de la manière suivante :

$$(0.5 - 0.25, 0.5 + 0.25) = (0.25, 0.75)$$
$$(0.5 + 0.25, 0.5 + 0.25) = (0.75, 0.75)$$
$$(0.5 - 0.25, 0.5 - 0.25) = (0.25, 0.25)$$
$$(0.5 + 0.25, 0.5 - 0.25) = (0.75, 0.25)$$

4 Prédiction à l'aide de l'abre de décision

On peut maintenant définir une fonction qui permet de prédire la couleur d'un point à l'aide de l'arbre de décision construit précédemment. On définit alors une fonction qui permet de naviguer dans l'arbre de décision en fonction des coordonnées du point à prédire. On renvoie la couleur de la feuille ainsi atteinte.

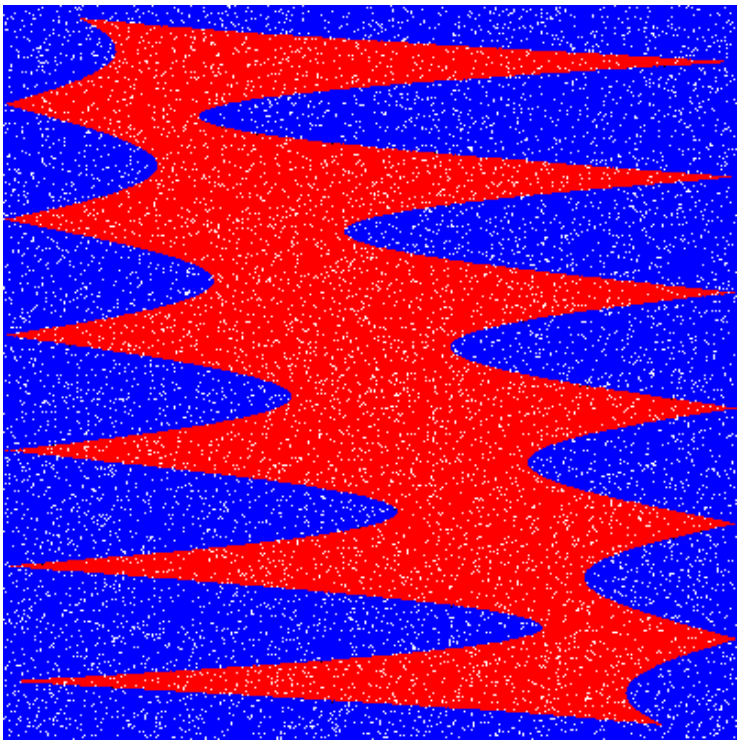
```
(* tree -> pos -> color *)
let rec predict_color tree pos =
  match tree with
  | Leaf color -> color
  | Node (separations,a,b,c,d) ->
      if pos.x < separations.x && pos.y >=
         separations.y
      then predict_color a pos
      else if pos.x >= separations.x && pos.y >=
         separations.y
      then predict_color b pos
      else if pos.x < separations.x && pos.y <
         separations.y
      then predict_color c pos
      else predict_color d pos ;;
```

5 Résultats

J'utiliserais pour la suite des tests trois fonctions de géneration de datasets. Une fonction qui génère un cercle, une autre qui génere le graphe de la fonction graphe et une fonction qui crée un dataset complexe, voici son code.

```
let generate p =
    if exp (-2.0 *. p.y) *. (abs_float (cos (20.0 *.
       p.y))) >= p.x || 1.0 -. exp (-2.0 *. (1.0 -.
       p.y)) *. (abs_float (cos (20.0 *. (1.0
       -.p.y)))) < p.x
        then blue else red ;;
```

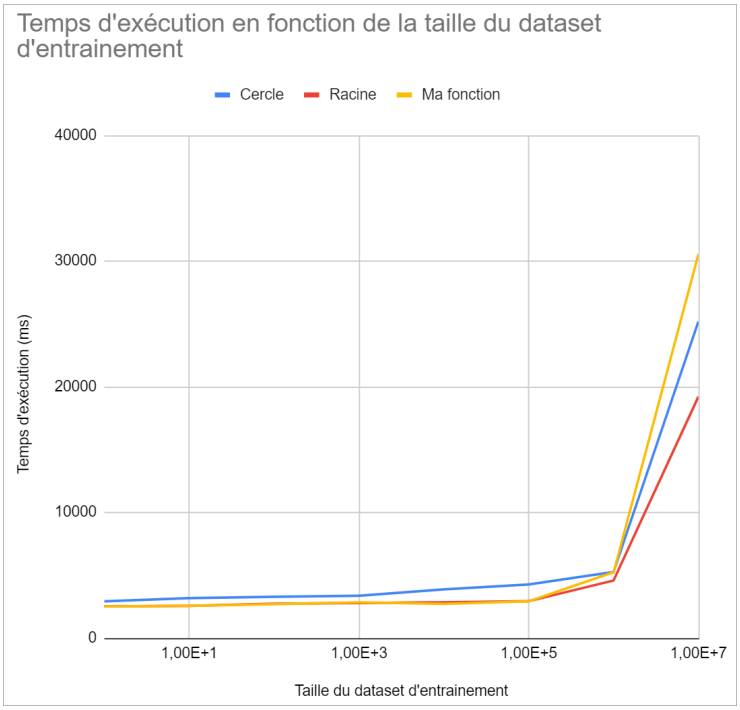
Cette fonction me semble interessante puisqu'elle genère un dataset avec une frontière de décision complexe.



On commence par mesurer complexité temporelle de notre programme en fonction de la taille du dataset. On génère un dataset de taille 10^n avec n allant de 0 à 6, et on mesure la durée d'exécution à l'aide de la commande Windows suivante :

```
Measure-Command {ocamlrun test}
```

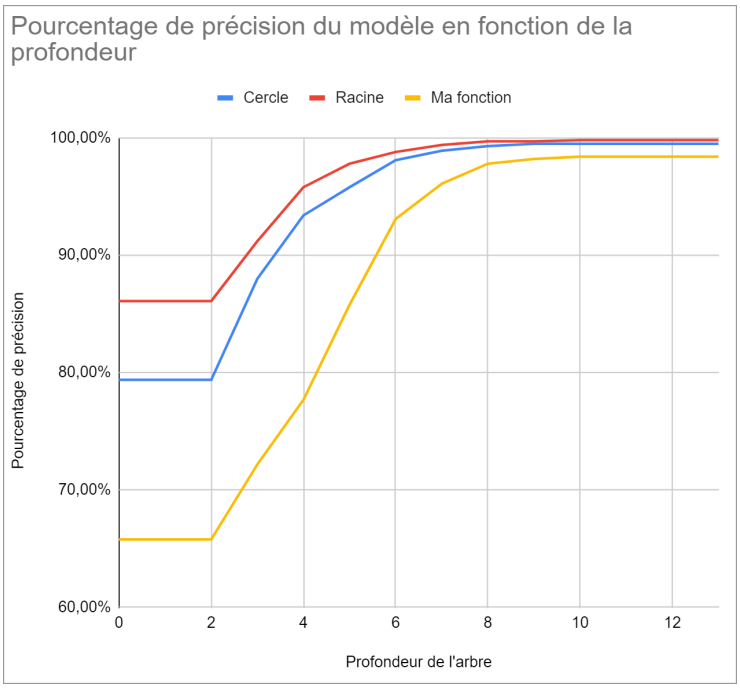
Ce qui nous donne le graphique suivant :



On constate que la complexité temporelle de notre programme est exponentielle en fonction de la taille du dataset. Cependant la durée d'execution reste acceptable (quelques secondes) pour des datasets d'une taille inférieur à 10^7 données.

On peut ensuite tester notre programme sur un dataset de taille fixe 100000, et faire varier la profondeur maximale de l'arbre de décision pour mesurer la précision de celui-ci. On fait cela à l'aide de la commande Windows suivante :

```
1 ocamlrun test -md profondeur
```



Ce graphique permet de montrer que notre arbre de décision est pertinent puisque la précision de celui-ci augmente en fonction de la profondeur de l'arbre pour tendre vers 100% de précision. On peut aussi voir que la précision stagne pour des profondeurs supérieures à 10, ce qui montre que l'arbre de décision est optimal pour une profondeur de 10 (d'où le choix de la pronfondeur maximale par défaut de 10).