Université Lille 1 – Master 1

RDF – TP n°09

Arbres de décision

BARCHID Sami – SLIMANI Anthony
24/03/2019

Introduction

L'objectif de ce TP est de comprendre le modèle des arbres de décision en utilisant l'algorithme ID3, une variante de l'algorithme général « CART ».

Ce rapport est divisé en 2 parties :

- **Question de bon sens :** questions d'introduction afin de mieux comprendre l'intuition derrière les arbres de décision
- **Variante du jeu du pendu :** implémentation d'une variante du célèbre jeu du pendu afin d'implémenter et de comprendre le fonctionnement d'un arbre de décision.

Question de bon sens

Problème posé

Deux joueurs (notés A et B) jouent au jeu suivant: A choisit un nombre entier x entre 1 et N (N étant un nombre connu de A et B), et B doit le deviner en lui proposant successivement des valeurs. Pour chaque valeur possible de x proposée par B, A répond “gagné !”, “x est plus petit”, ou “x est plus grand”.

a) Avant de jouer, B dit à A: “je n’ai pas besoin de plus de 4 propositions pour gagner”. A répond alors à B: “C’est vrai, mais avec mon choix de x tu auras besoin d’exactement 4 propositions”. Que vaut N ?

b) B répond alors: “Avec l’information que tu viens de me donner, je suis presque sûr de gagner avant.” Expliquer le raisonnement de B et calculer la quantité d’information que lui a donnée A par sa réponse.

a) Que vaut N ?

Il faut deviner un numéro entre 1 et N. La manière de trouver ce numéro en posant le moins de questions possible est de toujours choisir un numéro de telle sorte que l’ensemble des numéros plus grands et l’ensemble de numéros plus petits aient tous les deux la même taille. Nous voulons donc maximiser le gain d’information à chaque choix de numéro. En bref, il faut choisir toujours la médiane des numéros possibles.

Si nous utilisons cette méthode optimale en nombre de questions et que nous savons que nous n’avons besoin que de 4 choix maximum, on peut donc conclure que **N=16**.

En effet, en appliquant la stratégie optimale, on est sûr de tomber sur le bon numéro en 4 coups. Par exemple imaginons que A choisisse le numéro à trouver x = 1 :

- B choisi 8, A répond « x est plus petit »
- B choisi 4, A répond « x est plus petit »
- B choisi 2, A réponse « x est plus petit »
- B a trouvé x = 1

Ceci montre bien qu’il est obligatoire, avec cette stratégie, de trouver une valeur unique en 4 étapes maximum.

b) Expliquer le raisonnement de B et calculer la quantité d’information que lui a donné A par sa réponse.

Par sa réponse, A assure qu’il est impossible que B trouve le numéro en moins de 4 coups. Cela montre que le numéro que A a choisi est impair. En effet, la stratégie de B implique de devoir toujours choisir un numéro pair si nous pouvons séparer l’ensemble des numéros.

Le seul moment où B choisirait un numéro impair serait au moment où il n’y a plus que ce choix-là de disponibles (et donc que B trouve le bon numéro en 4 coups). A a donc donné l’information que le numéro choisi est **impair**.

B n’a plus qu’à travailler uniquement sur les nombres impairs et donc est presque sûr de pouvoir donner sa réponse en moins de 4 coups. Il n’a qu’à utiliser la même stratégie qu’auparavant mais sur l’ensemble des numéros impairs entre 1 et N.

Variante du jeu du pendu

L'exercice proposé pour ce TP que nous étudierons afin de comprendre les arbres de décision est une variante du jeu du pendu.

Le jeu se déroule de la manière suivante :

- Un ensemble de noms possibles est mis à disposition
- Le joueur en choisi un
- Tant que l'ordinateur n'a pas trouvé le nom choisi
    - Il pose une question du type : « Ce mot contient-il la lettre xxxxxx ? » (où xxxxxx est une des 26 lettres de l'alphabet).
    - Le joueur répond Oui ou non

Le but étant que l'ordinateur puisse trouver le mot le plus rapidement possible.

Lorsque nous faisons le lien avec la théorie des arbres de décisions, nous remarquons :

- Les variables à prédire (les classes) sont les mots eux-mêmes.
- Les variables prédictives sont les lettres de l'alphabet qui composent le mot.
- Pour chaque nœud, le choix de test est déjà défini puisque ce sera la question « Contient-il la lettre xxxxx » où xxxx est l'attribut à choisir (une lettre de l'alphabet).

La base de mots est une base de noms d'animaux contenus dans un vecteur « noms ». La taille de ce vecteur « noms » est :

$$n = 283$$

Question 2

« Si le programme est sûr de trouver la solution en au plus p questions, quelle est la relation entre n et p ? En déduire une valeur numérique minimale pour p. »

Nous pouvons déduire alors que p est la hauteur la plus grande de l'arbre de décision optimal pour la base de noms de taille n. Nous savons aussi que l'arbre de décision est binaire (car on ne peut répondre à la question d'un nœud que par « Oui » ou « Non »). De ce fait, on déduit que

$$p = \log_2 n$$

Question 3

Selon vous, à quoi sert la fonction suivante ?

```
maFonction <- function(x) { strtoi(charToRaw(x),16L)-96 }
```

Renommer cette fonction pour que le nom soit plus explicite.

« maFonction » prend en paramètre un caractère « x » et retourne la position de la lettre x dans l'alphabet.

On peut renommer cette fonction de la sorte :

```
numeroFromLettre = function(lettre) {
  strtoi(charToRaw(lettre),16L)-96
}

lettreFromNumero = function(numero) {
  rawToChar(as.raw(numero + 96));
}
```

Note : nous y avons également inclus l'opération inverse « lettreFromNumero » qui sera également utile pour la suite du TP.

Expliquer ce que contient la matrice "mat" après exécution des commandes R suivantes:

```
mat = matrix(rep(0,26*n),nrow=26, ncol=n);
for (i in 1:n)
{
    c = numeroFromLettre(noms[i]);
    mat[c,i] <- 1;
}
```

La matrice « mat » est une matrice $26 \times n$ (26 lignes pour les 26 lettres de l'alphabet et n colonnes pour les n mots).

Le but de « mat » est d'indiquer si un mot n_j contient une lettre de l'alphabet a_i. Donc :

- mat[i][j] = 1 quand le mot n_j contient la lettre a_i
- mat[i][j] = 0 sinon

Question 4

Calculer, sous la forme d'un vecteur à 26 colonnes h(i), la proportion de mots qui contiennent (au moins une fois) la lettre i. Quelle est la lettre la plus représentée ?

Dans le langage R, la fonction « *rowSums(matrix)* » permet d'obtenir un vecteur contenant, pour chaque ligne de « matrix », la somme des éléments appartenant à cette ligne dans la matrice. Le calcul de h est le suivant :

```
# vecteur des 26 lettres contenant le nombre de mots où la ième lettre de l'alhpabet se trouve
h = rowSums(mat)
```

Comme « mat » possède 26 lignes et que mat[i][j] = 1 quand un mot j contient la lettre i, le résultat de rowSums nous fournit bien le vecteur demandé.

La lettre la plus représentée s'obtient grâce aux instructions R suivantes :

```
# vecteur des 26 lettres contenant le nombre de mots où la ième lettre de l'alhpabet se trouve
h = rowSums(mat)
indiceLettre = which.max(h)
valOccurence = max(h)
lettre = lettreFromNumero(indiceLettre)
cat("La lettre ", lettre, " est la plus représentée avec ", valOccurence, " apparitions")
```

On observe que la lettre « **e** » est la lettre la plus représentée avec **184** apparitions.

Question 5

Calculer, pour tout i, l'entropie de la partition des mots en deux ensembles, les mots qui contiennent la lettre i et les autres.

Le but ici est de calculer le gain d'information selon l'attribut que l'on choisit pour le nœud (et donc de calculer la pureté pour les sous-ensembles créés suivant la lettre que nous choisissons pour séparer l'ensemble en un ensemble « Contient la lettre » et un ensemble « ne contient pas »). Ce calcul est donné par les instructions R suivantes :

```
# Entropie de la partition des mots en deux ensembles
probabilites = h/n # calcul des probabilités
entropies = - log2(probabilites^probabilites) - log2((1-probabilites) ^ (1 - probabilites))
```

La variable « entropies » ici contient un vecteur des 26 lettres de l'alphabet où la valeur est l'entropie (donc le gain d'information) obtenue pour chaque partition des mots en deux ensembles (un qui contient la lettre et l'autre non).

Question 6

En déduire la 1ère question que doit poser le programme pour maximiser l'information gagnée en moyenne.

Pour maximiser l'information gagnée en moyenne, on cherche la lettre qui possède la plus grande entropie comparée aux autres. Comme nous avons calculé dans la question précédente les entropies pour le partitionnement selon chaque lettre, nous sommes en mesure de retrouver la lettre qui maximise le gain d'information. L'instruction R suivant permet de retrouver la lettre qui maximise l'entropie grâce à :

```
# trouver la lettre qui sépare le mieux l'ensemble en deux sous-ensemble
# càd, trouver la lettre qui a la plus grande entropie, donc qui donne la plus grande incertitude
bestLetter = which.max(entropies)
cat("La lettre qui maximise l'entropie est ",
    lettreFromNumero(bestLetter),
    " avec une entropie de ",
    max(entropies))
```

Pour la première question, la lettre choisie sera '**o**' car son entropie est la plus élevée avec une valeur de **0.99991894**.

L'entropie pour la lettre **o** se rapproche fortement de 1, signifiant que partitionner l'ensemble de noms en deux sous-ensembles suivant 'o' permet de quasiment diviser l'ensemble en deux sous-ensembles équiprobables.

Question 7

Écrire une fonction qui renvoie l'indice i de la lettre associée à la question la plus informative pour le sous-ensemble de mots courant, ainsi que la partition de ce sous-ensemble.

```
#Fonction qui retourne l'indice de la lettre associée à la question la plus informative pour
# l'ensemble de mots courant, ainsi que la partition de l'ensemble
#
# ici, on recherche la lettre qui maximise l'entropie de sorte qu'on arrive à séparer l'ensemble de
# mots "e" en deux sous-ensembles les plus équitables possibles (le mieux serait de trouver des questions
# pour obtenir des ensembles equiprobables)
partage = function(e) {
  n = length(e)

  mat = matrix(rep(0,26*n),nrow=26, ncol=n);
  for (i in 1:n)
  {
    c = numeroFromLettre(e[i]);
    mat[c,i] <- 1;
  }

  # vecteur des 26 lettres contenant le nombre de mots où la ième lettre de l'alhpabet se trouve
  h = rowSums(mat)

  # Entropie de la partition des mots en deux ensembles
  probabilites = h/n # calcul des probabilités
  entropies = - log2(probabilites^probabilites) - log2((1-probabilites) ^ (1 - probabilites))

  # trouver la lettre qui sépare le mieux l'ensemble en deux sous-ensemble
  # càd, trouver la lettre qui a la plus grande entropie, donc qui donne la plus grande incertitude
  bestLetter = which.max(entropies)

  # trouver le sous-ensemble qui possède la lettre [bestLetter]
  contient = e[mat[bestLetter,] == 1]

  # trouver le sous-ensemble qui ne possède pas la lettre [bestLetter]
  neContientPas = setdiff(e, contient)

  list("indiceLettre" = bestLetter, "contient" = contient, "neContientPas" = neContientPas)
}
```

Note : des commentaires ont été ajoutés dans la macro pour aider à la compréhension de l'algorithme.

La fonction « partage » prend en paramètre un ensemble de mots « e » et retourne une liste contenant :

- L'indice de la lettre associée à la question qui maximise le gain d'information pour l'ensemble de mots en paramètre
- Le sous-ensemble des mots qui contiennent la lettre en question
- Le sous-ensemble des mots qui ne la contiennent pas

Pour ce faire, la fonction est composée des opérations suivantes :

- Calculer la matrice « mat » définie précédemment
- Calculer le vecteur « h » défini précédemment
- Calculer les entropies associées à chaque couple (question ; lettre alphabet)
- Trouver la lettre qui possède une entropie maximale
- Calculer les sous-ensembles à retourner
- Retourner la liste

Question 8

Ecrivez une fonction qui permet de partager itérativement l'ensemble initial en sous-ensembles de plus en plus petits.

```
partageIteratif = function(e) {
  sousEnsembles = list() # liste des sous-ensembles à retourner
  sousEnsembles[[1]] = e

  # index de parcourt des sous-ensembles à diviser
  i = 1

  # TANT QUE [j'ai un ensemble à diviser]
  while(i <= length(sousEnsembles)) {
    aPartager = sousEnsembles[[i]]
    # Si le sous-ensemble à partager n'est pas pur, on peut le partitionner
    if(length(aPartager)>1) {
      parts = partage(aPartager)

      e1 = parts$contient
      e2 = parts$neContientPas

      if(length(e1) > 0) {
        #Ajout du premier sous-ensemble (si celui-ci contient quelque chose)
        sousEnsembles[[length(sousEnsembles) + 1]] = e1
      }

      if(length(e2) > 0) {
        #Ajout du deuxième sous-ensemble (si celui-ci contient quelque chose)
        sousEnsembles[[length(sousEnsembles) + 1]] = e2
      }
    }
    i = i+1
  }

  sousEnsembles
}
```

Note : des commentaires ont été ajoutés pour aider à la compréhension

Question 9

Enfin, écrire une fonction qui permet de jouer interactivement avec l'ordinateur…

```
jeu = function() {
  # Chargement de la base de noms d'animaux
  source ("rdfAnimaux.txt") # crée un variable "noms"

  e = noms

  # tant que je ne suis pas sur un noeud pur
  while(length(e) > 1) {
    parts = partage(e)

    lettre = lettreFromNumero(parts$indiceLettre)

    cat("Est-ce que ton mot possède la lettre suivante : ", lettre, " ? 1 pour 'Oui', autre pour 'non'")
    answer = scan()

    if(answer == 1) {
      e = parts$contient
    }
    else {
      e = parts$neContientPas
    }
  }

  if(length(e) == 0) {
    cat("Aucun mot n'a été trouvé.")
  }
  else {
    cat("J'ai trouvé, ton mot est : ", e[1])
  }
}
```

La fonction de jeu interactive fonctionne de la même manière que celle décrite au début de la partie du TP lors de l'explication des règles de la variante du pendu.

Ci-dessous se trouve un exemple du déroulement du jeu où le mot à rechercher par l'ordinateur est « oie » :

```
> jeu()
Est-ce que ton mot possède la lettre suivante :  o  ? 1 pour 'Oui', autre pour 'non'
1: 1
2:
Read 1 item
Est-ce que ton mot possède la lettre suivante :  e  ? 1 pour 'Oui', autre pour 'non'
1: 1
2:
Read 1 item
Est-ce que ton mot possède la lettre suivante :  r  ? 1 pour 'Oui', autre pour 'non'
1: 0
2:
Read 1 item
Est-ce que ton mot possède la lettre suivante :  n  ? 1 pour 'Oui', autre pour 'non'
1: 0
2:
Read 1 item
Est-ce que ton mot possède la lettre suivante :  t  ? 1 pour 'Oui', autre pour 'non'
1: 0
2:
Read 1 item
Est-ce que ton mot possède la lettre suivante :  i  ? 1 pour 'Oui', autre pour 'non'
1: 1
2:
Read 1 item
Est-ce que ton mot possède la lettre suivante :  a  ? 1 pour 'Oui', autre pour 'non'
1: 0
2:
Read 1 item
Est-ce que ton mot possède la lettre suivante :  c  ? 1 pour 'Oui', autre pour 'non'
1: 0
2:
Read 1 item
J'ai trouvé, ton mot est :  oie
>
```

Question 10

Définir une fonction pour construire et afficher l'arbre de décision optimal associé au jeu

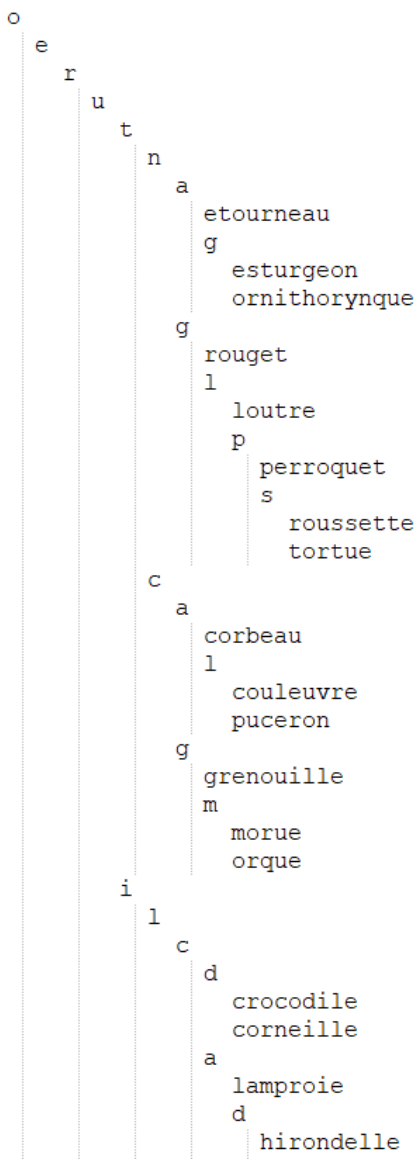
```
induction = function(e, espace) {
  # SI [ tous les éléments de e sont de la même catégorie]
  if(length(e) <= 1) {
    # Construire la feuille associée à la classe
    cat("\n ", espace, e)
  }
  # SINON
  else {
    # Séparer e en 2 parties selon qui possède la lettre qui divise correctement e
    parts = partage(e)
    e1 = parts$contient
    e2 = parts$neContientPas
    cat("\n ", espace, lettreFromNumero(parts$indiceLettre))


    # induction sur la partition
    espace = paste(espace, " ")
    induction(e1, espace)
    induction(e2, espace)
  }
}
```

La fonction « induction » est une fonction récursive qui permet de construire l'arbre de décision à partir d'un ensemble de mots « e ». Comme elle est chargée d'afficher l'arbre, la fonction garde en paramètres une chaine de caractères « espaces » servant à décaler les nœuds fils de leur parent.

Un extrait de l'arbre (les premières dizaines de lignes) se trouve ci-dessous :

```
> induction(noms, "")
```



Chaque nœud de l'arbre est la lettre qui a été choisie dans le test « Le mot contient-t-il la lettre xxxxx ? » et chaque feuille de l'arbre est le mot (donc la classe) qui découle des tests. Les feuilles de l'arbres sont donc d'office des nœuds purs puisqu'il n'y a qu'une seule classe possible.