Chapitre 18 : Intelligence artificielle et théorie des jeux

Table des matières

1 Introduction

L'expression *intelligence artificielle* est une manière floue de désigner des algorithmes chargés comme tous les autres de résoudre des problèmes. Certains d'entre eux doivent jouer à des jeux, d'autres doivent répartir des données en plusieurs catégories (on parle de *classification*) ou déterminer des valeurs numériques associées à des paramètres d'entrée (on parle de *problème de régression*). Quelques traits communs à ces algorithmes sont l'usage d'heuristiques afin d'essayer d'obtenir des réponses les meilleures possibles en temps raisonnable et l'exploitation d'une grande quantité de données afin d'en construire une représentation (on parle de l'*apprentissage d'un modèle de données*) qui sera exploité pour construire la réponse de l'algorithme. Dans ce chapitre, on se limite à l'étude des problèmes de classification et de la théorie des jeux.

2 Apprentissage supervisé

2.1 Algorithme des k plus proches voisins

2.1.1 Introduction

Pour résoudre un problème de classification, la méthode de l'apprentissage supervisé consiste à exploiter des données dont on connait déjà la classe afin de construire un algorithme de classification prenant les caractéristiques d'une donnée en entrée et renvoyant la classe à laquelle cette donnée appartient probablement.

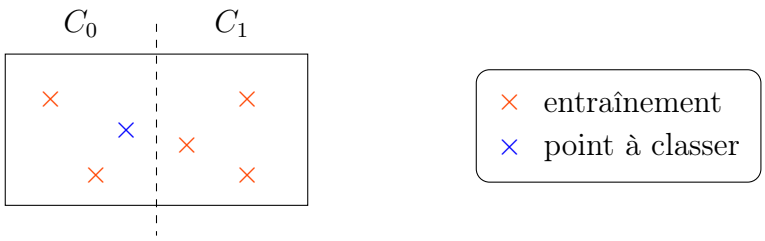
Les données manipulées sont en général représentées par des points de $\mathbb{R}^d$ ou d est souvent grand. Par exemple, si on veut reconnaître des caractères manuscrits, l'algorithme peut prendre en entrée une image de 28×28 pixels en 256 niveaux de gris contenant un scan du caractère manuscrit à reconnaître, donc la donnée est représentée par un point de $[\![0\ ;\ 255]\!]^d$, où $d = 28^2 = 784$.

Si on veut répartir dans C classes des données de $\mathbb{R}^d$, le problème consiste à construire un algorithme réalisant une fonction $\mathbb{R}^d \longrightarrow [\![0\ ;\ C-1]\!]$ en exploitant une heuristique s'appuyant sur un ensemble de couples $(x, y) \in \mathbb{R}^d \times [\![0\ ;\ C-1]\!]$, où y est la classe de x, appelées données d'entrainement.

2.1.2 Algorithme des k plus proches voisins

Idée : il est possible que des données proches appartiennent à la même classe, donc on pourrait pour un point donné, renvoyer la classe du point déjà étiqueté le plus proche.

Problème : la donnée d'entrainement la plus proche peut appartenir à une autre classe que celle du point, par example si les données d'entrainement sont mal réparties, ou bruitées, ou si le point considéré est proche de la frontière entre deux classes.



Pour éviter cet écueil, on considère plutôt la classe majoritaire parmi les classes des k données d'entrainement les plus proches du point d'entrée, pour un k fixé.

On parle alors de l'algorithme des k–plus proches voisins (kNN pour *k–nearest neighbors*).

Variables d'ajustement :

- En cas d'égalité, il faut choisir une classe, par exemple au hasard parmi les classes majoritaires.
- Le nombre k de voisins : si k est trop faible, l'algorithme sera trop sensible au bruit sur les données et si k est trop grand, l'algorithme renverra surtout la classe majoritaire parmi les données d'entraînement, donc effectue une mauvaise généralisation.
- La notion de distance : on utilise souvent la distance de Minkowski

$$d(x, x') = \left(\sum_{i=1}^{d} |x_i - x'_i|^p \right)^{\frac{1}{p}}$$

qui donne la distance de Manhattan pour $p = 1$, et la distance euclidienne pour $p = 2$. Le programme se limite à la distance euclidienne.

Pour déterminer les k plus proches voisins d'un point donné, on peut exploiter une file de priorité :

Algorithm 1: Algorithme des k plus proches voisins

Input: données d'entraînement $(x_i, y_i)_{i \in [\![1\ ;\ N]\!]}$
Input: point à classer x

```
F ← file de priorité max vide;
for i de 1 à k do
    Insérer i dans F avec la priorité d(x, x_i);
for i de k + 1 à N do
    if d(x, x_i) < d(x, x_{max F}) then
        Extraire le max de F;
        Insérer i dans F avec la priorité d(x, x_i);
C ← {C_i | i ∈ F};
return un élément le plus fréquent de C;
```

Complexité : $\mathcal{O}(N \log k)$ en temps, et $\mathcal{O}(k)$ en espace.

Remarque : si on a beaucoup de point à classer, cet algorithme est peu efficace car il nécessite de parcourir l'intégralité des données pour chaque point à classer. On pourrait

plutôt effectuer un prétraitement des données pour rendre plus efficace le calcul des k plus proches voisins d'un point donné.

2.1.3 Arbres k–dimensionnels

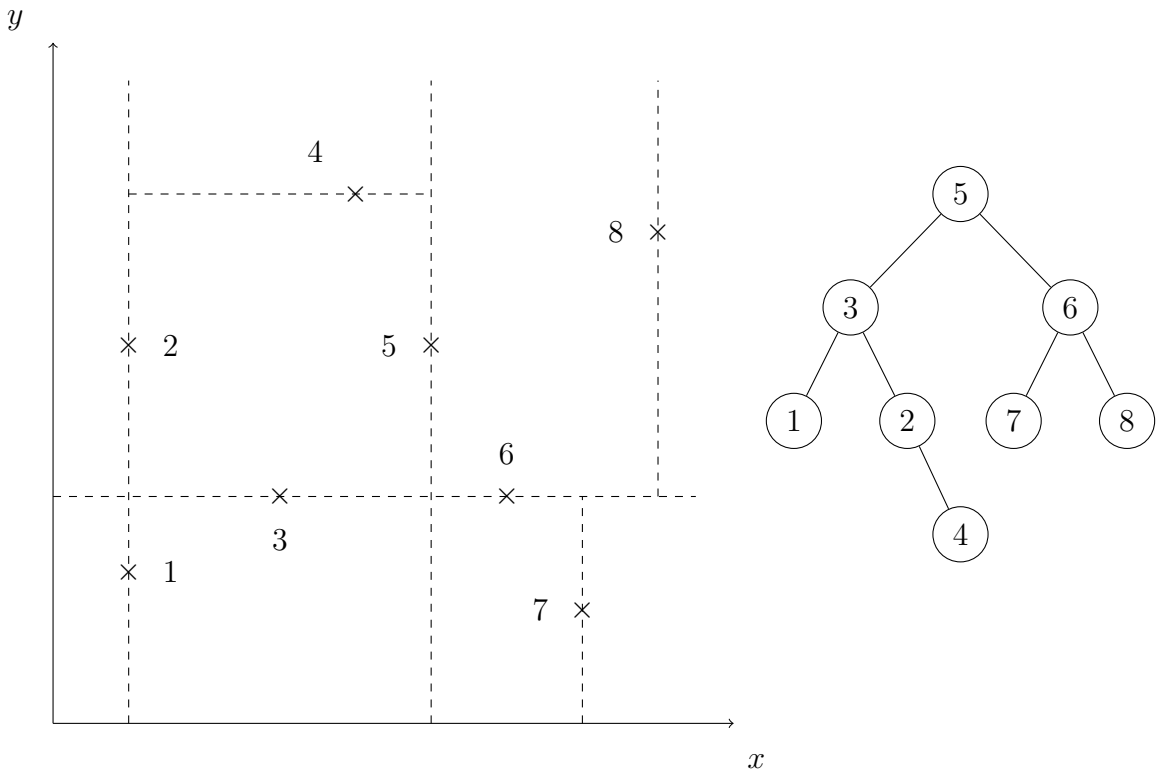
Attention : on ne parle pas du k de kNN, mais plutôt de la dimension de l'espace des données (d en 2.1.1, page 2), mais c'est la lettre k qui est utilisée dans la littérature.

• Définition : la structure d'arbre k–dimensionnel, ou arbre k-d, est une généralisation des la notion d'arbre binaire de recherche : un arbre binaire étiqueté par des éléments de $\mathbb{R}^k$ est un arbre k-d si et seulement si pour tout nœud d'étiquette $x = (x_0, \ldots, x_{k-1})$ de profondeur i,

$$\begin{array}{ll} \forall x' = (x'_0, \ldots, x'_{k-1}) \text{ étiquette du sous-arbre gauche}, & x'_j \leqslant x_j \\ \forall x' = (x'_0, \ldots, x'_{k-1}) \text{ étiquette du sous-arbre droit}, & x'_j > x_j \end{array}$$

où $j = i \mod k$.

• Exemple en dimension 2 :



• Remarque : dans cet exemple, on a fait en sorte de construire un arbre k-d équilibré en choisissant l'élément médian pour la coordonnée associée à la profondeur du nœud comme étiquette.

On écrit l'algorithme `créer_arbre` suivant :

Algorithm 2: `créer_arbre`

```
1 Function créer_arbre(k, i, l):
      Input: dimension k
      Input: profondeur i
      Input: liste de données l
2     if l = [] then
3         return l'arbre vide;
4     else
5         Extraire l'élément x de l médian pour la coordonnée i mod k;
6         l_<, l_> ← partition de l suivant le pivot x;
7         return Noeud(x, créer_arbre(k, i + 1, l_<), créer_arbre(k,
              (i + 1), l_>));
```

Complexité : à l'aide de l'algorithme de calcul de la médiane en temps linéaire (*cf* chapitre 7, 2.2), la complexité est celle du tri rapide dans le meilleur cas, *i.e* $\mathcal{O}(N \log N)$.

2.1.4 Recherche des k plus proches voisins dans un arbre k-d

Idée : pour trouver les k plus proches voisins d'un point $x \in \mathbb{R}^d$ dans un arbre k-d T de dimension d, on procède initialement comme pour la recherche de x dans un ABR (en comparant la bonne coordonnée à chaque profondeur) puis on remonte dans l'arbre en sélectionnant les k voisins, parfois en redescendant des un sous-arbre que l'on avait ignoré.

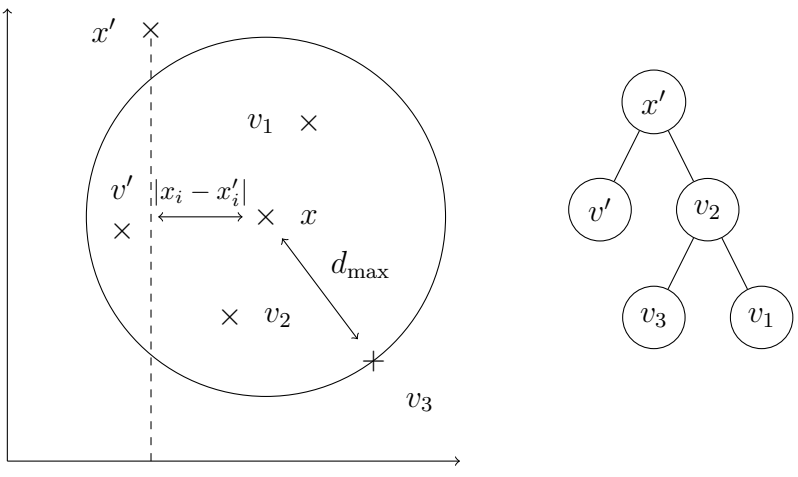
Exemple : on suppose être au niveau d'un nœud d'étiquette x' de profondeur i et tel que $x_i > x'_i$.

On cherche donc récursivement les k voisins dans le sous-arbre droit.

S'il y a moins de k nœuds dans ce sous-arbre, il faudra considérer x' comme voisin et peut-être aussi les nœuds du sous-arbre gauche.

Même si l'appel récursif sélectionne k voisins, on peut devoir considérer x' ou les nœuds du sous-arbre gauche si $|x_i - x'_i|$ est inférieure à la distance maximale entre x et les voisins sélectionnés.

Par exemple, si $k = 3, d = 2$, et i correspond aux abscisses,



D'où l'algorithme :

Algorithm 3: `recherche_voisins`

```
1  Function recherche_voisins(x, T, k):
2      F ← file de priorité max vide;
3      visite(F, x, T, 0, k);
4      return les éléments de F;
5  Function visite(F, x, T, i, k):
6      if T = Noeud(x', k, r) then
7          if x_i ⩽ x'_i then
8              t_1, t_2 ← l, r;
9          else
10             t_1, t_2 ← r, l;
11         visite(F, x, t_1, i + 1 mod d, k);
12         if |F| < k ou priorité max F ⩾ |x_i − x_j| then
13             if d(x, x') < priorité max F then
14                 Extraire max F;
15                 Insérer x' dans F avec la priorité d(x, x');
16             visite(F, x, t_2, i + 1 mod d, k);
```

Dans le pire cas, on visite les N nœuds de l'arbre (donc on n'a rien gagné par rapport au premier algorithme) mais le plus souvent on ne visite que de l'ordre de $\log N$ nœuds.