

1 Partie 1 OCaml : autour des k plus proches voisins

1.1 Contexte

Le principe de cette partie est de programmer, en OCaml¹, la méthode de classification des k plus proches voisins.

En intelligence artificielle, plus précisément en apprentissage automatique, la méthode des k plus proches voisins est une méthode d'apprentissage supervisé. En abrégé KPPV ou k-PPV en français, ou plus fréquemment k-NN ou KNN, de l'anglais k-nearest neighbors. Dans ce cadre, on dispose d'une base de données d'apprentissage constituée de N couples "entrée-sortie". Pour estimer la sortie associée à une nouvelle entrée x, la méthode des k plus proches voisins consiste à prendre en compte (de façon identique) les k échantillons d'apprentissage dont l'entrée est la plus proche de la nouvelle entrée x, selon une distance à définir.²

On suppose qu'on dispose ici d'une base de données d'apprentissage constituée de N couples "vecteur-classe" (le vecteur est sur un plan à 2 dimensions).

Dans cette partie, les fonctions OCaml demandées doivent être systématiquement accompagnées de tests unitaires sur des paramètres significatifs. De plus, les cas limites pouvant apparaître dans les différentes fonctions seront à traiter à la discrétion du candidat qui devra être prêt à les justifier.

L'utilisation des fonctions, d'une question à l'autre, lorsque c'est pertinent, est fortement recommandée.

On pourra vouloir tester les fonctions sur des données judicieusement générées aléatoirement (question 9).

1.2 Tests

Pour chaque fonction programmée, fournir des tests unitaires significatifs à l'aide de la fonction `assert`.

On propose le format de test unitaire suivant :

```
let test_nom_fonction_testée1 () =
  ....
  assert (nom_fonction_testée <jeu de paramètres 1> = résultat_attendu1) ;
  assert (nom_fonction_testée <jeu de paramètres 2> = résultat_attendu2) ;
  ...
  print_string(" test nom_fonction_testée1 ok\n") ;;
```

Où la quantité de tests, les paramètres donnés à la fonction testée sont à faire évoluer pour obtenir des tests significatifs. `résultat_attendu` est le résultat attendu à l'appel de la fonction avec le jeu de paramètres donnés.

En fin de programme, il faut ajouter ces lignes afin de réaliser l'ensemble des tests unitaires.

```
let tests () =
  test_nom_fonction_testée1 () ;
  test_nom_fonction_testée2 () ;
  ... ;;
tests ();;
```

Où la fonction `tests` doit contenir les appels aux fonctions de tests réalisées.

1.3 Programmation

L'objectif du programme est de pouvoir classer un nouveau vecteur parmi une liste de données contenant des vecteurs et leur classe en prenant en compte un nombre k de proches voisins. La

1. Le fonctionnel pur n'est pas exigé mais recommandé

2. Wikipedia

programmation devra permettre de faire varier le k et le vecteur à classer facilement.

On aura besoin de trier les données en fonction de leur distance au vecteur à classer (questions 1, 5, 6 et 7), c'est-à-dire définir une fonction de distance, pouvoir trier selon un ordre, définir cet ordre qui dépend de la distance au vecteur à classer et enfin trier.

A l'aide des k premiers éléments de cette liste triée (question 3), on pourra compter le nombre de classes qui sont présentes parmi ces k proches voisins (question 2) et ainsi déterminer la classe qui est majoritaire (question 4). Cette classe majoritaire est celle recherchée pour classer le nouveau vecteur (question 8).

Pour simplifier, les **coordonnées** des **vecteurs** seront de type `float`. Une **classe** sera représentée par un type `string`.

Une **donnée** sera un couple (vecteur,classe) de type `(float * float) * string`.

1. Ecrire une fonction `distance` qui calcule la distance euclidienne entre deux vecteurs.
2. Ecrire une fonction qui compte le nombre d'occurrences de chaque classe d'une liste de données, le résultat sera sous forme de couple de type `string * int`.
3. Ecrire une fonction qui extrait les k premiers éléments d'une liste quelconque.
4. Ecrire une fonction qui renvoie le couple maximum d'une liste de couples. Les couples sont de type `'a * int` et le maximum se calcule par comparaison entre les deuxièmes éléments des couples.
5. Ecrire une fonction de tri selon une fonction d'ordre. Pour rappel, une fonction d'ordre renvoie -1, 0 ou 1 selon si son premier paramètre est inférieur, égal ou supérieur au deuxième paramètre.
6. Ecrire une fonction d'ordre qui convient pour utiliser le tri selon la distance à un vecteur.
7. Ecrire une fonction de tri des données selon la distance de ces données à un vecteur donné en paramètre.
8. Ecrire une fonction `k_plus_proches_voisins` qui permet de classer un vecteur parmi une liste de données prenant en compte k voisins.
9. Ecrire une fonction qui génère aléatoirement une liste de n données. Indication : la librairie `Random` fournit une fonction `Random.float : float -> float` qui génère un `float` entre 0. et le paramètre inclus. Le candidat devra justifier de ce qui est aléatoire et de ce qui ne l'est pas. Il devra utiliser au maximum les primitives de liste OCaml.

2 Prétraitement des données pour optimiser la recherche (approchée) des k plus proches voisins.

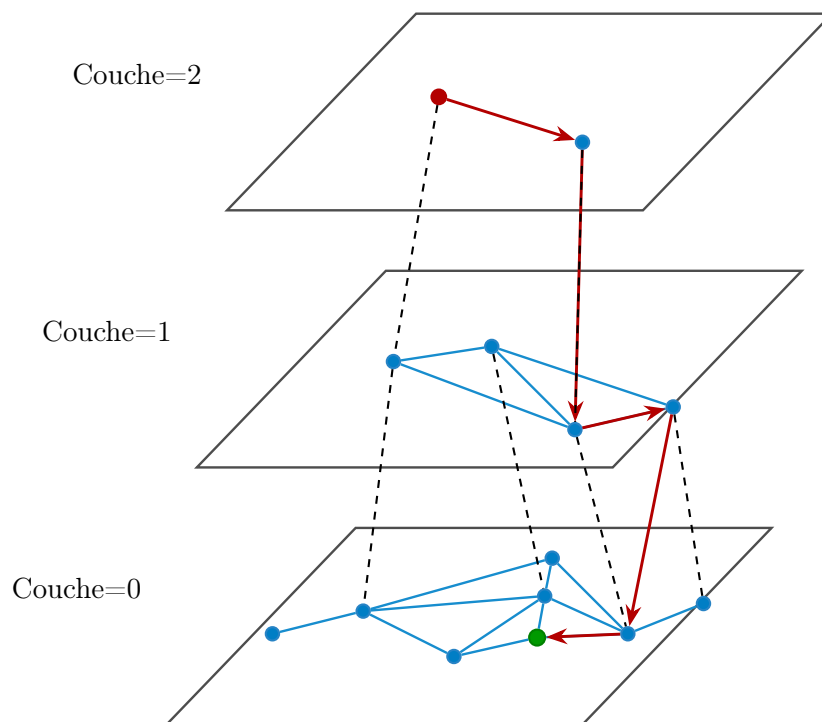
Le coût de la prédiction d'une classe par l'algorithme des k -plus proches voisins repose essentiellement sur la complexité de la fonction qui détermine, pour un vecteur fixé, les k plus proches voisins dans le jeu de données. Afin de rendre cette opération efficace, nous allons organiser nos données dans un graphe multicouche dans lequel on sera en mesure de faire une recherche efficace de " k plus proches voisins". Cette structure ne permettra pas de renvoyer de manière certaine les k plus proches voisins, elle permettra de fournir une réponse approchée à la question donnant lieu ici à une heuristique de recherche des k plus proches voisins.

Nous allons stocker nos vecteurs dans un graphe contenant plusieurs couches telles que toutes les données sont présentes dans la couche 0 et qu'un vecteur présent dans une couche $h \geq 0$ se retrouvera avec une certaine probabilité dans la couche du dessus. Ainsi, plus le graphe correspondra à une couche élevée, moins il contiendra de sommets et donc plus la recherche des voisins contenus dans cet étage sera rapide à effectuer. Chaque graphe correspondra à un graphe de proximité c'est-à-dire un graphe dans lequel un parcours glouton permet de trouver de manière approchée les plus proches voisins.

Dans un graphe de proximité, si on veut trouver un plus proche voisin d'un vecteur x , on entre dans le graphe par un sommet d'entrée et ensuite on avance depuis un sommet vers son voisin qui réalise la plus petite distance à x parmi tous les voisins envisagés et ceci jusqu'à trouver un minimum local.

Dans le cadre de nos graphes multicouches on procèdera de même en entrant par l'étage le plus haut par un point d'entrée qui sera défini dans la structure et à chaque fois que le parcours glouton nous conduira à un minimum local, on pourra descendre d'une couche afin de continuer la recherche dans un graphe plus dense situé à l'étage du dessous et en accédant à ce graphe depuis le sommet où on a arrêté la recherche précédente.

La figure suivante illustre une telle structure dans laquelle on accède par le point rouge dans la couche supérieure. Les flèches rouges illustrent un parcours glouton dans le graphe depuis ce point d'entrée jusqu'au point objectif vert en changeant de niveau à chaque fois que l'on a atteint un minimum local dans le niveau courant.



La construction d'une telle structure reposera sur une insertion successive des vecteurs du jeu de données. On se propose ici d'implémenter cette fonction d'insertion puis de voir comment trouver les k plus proches voisins d'un vecteur fixé dans la structure obtenue.

Pour représenter notre graphe multicouche, nous proposons la structure de donnée en C suivante :

```
struct vecteur{
    int dim;
    double* vect;
};

struct graphe_strat{
    int taille;
    /* le nombre de sommets du graphe*/
    int nb_couches;
    /*le nombre de couches*/
    int** nb_voisins;
    /* nb_voisins[s][k] contiendra le nombre de voisins du sommet s dans la couche k*/
    int*** voisins;
};
```

```

/*voisins[s][k] contiendra un tableau de taille M dont on considèrera
uniquement les nb_voisins[s][k] premières cases qui contiendront
les voisins du sommet s dans la couche k*/
int* hauteurs;
/* un tableau qui contient la hauteur de chaque donnée ie
le nombre de couches où elle apparaît*/

struct vecteur* sommets;
/*crée une correspondance entre les vecteurs données et les numéros des sommets
du graphe qui sont utilisés dans les tableaux ci-dessus*/
int M; /*nombre maximal de voisins par couche qui déterminera la taille(capacité)
de chaque tableau voisins[s][k]*/
int ef; /*paramètre permettant de moduler la qualité des recherches gloutonne*/
float ml; /*paramètre pour la loi de décroissance exponentielle qui
définit de manière probabiliste le niveau de chaque vecteur*/
int ep; /*le nom du sommet point d'entrée du graphe*/
int dim; /*dimension des vecteurs*/
int n; /*nombre de sommets maximal autorisé dans la structure*/
int k; /*nombre de couches maximal autorisé dans la structure*/
};

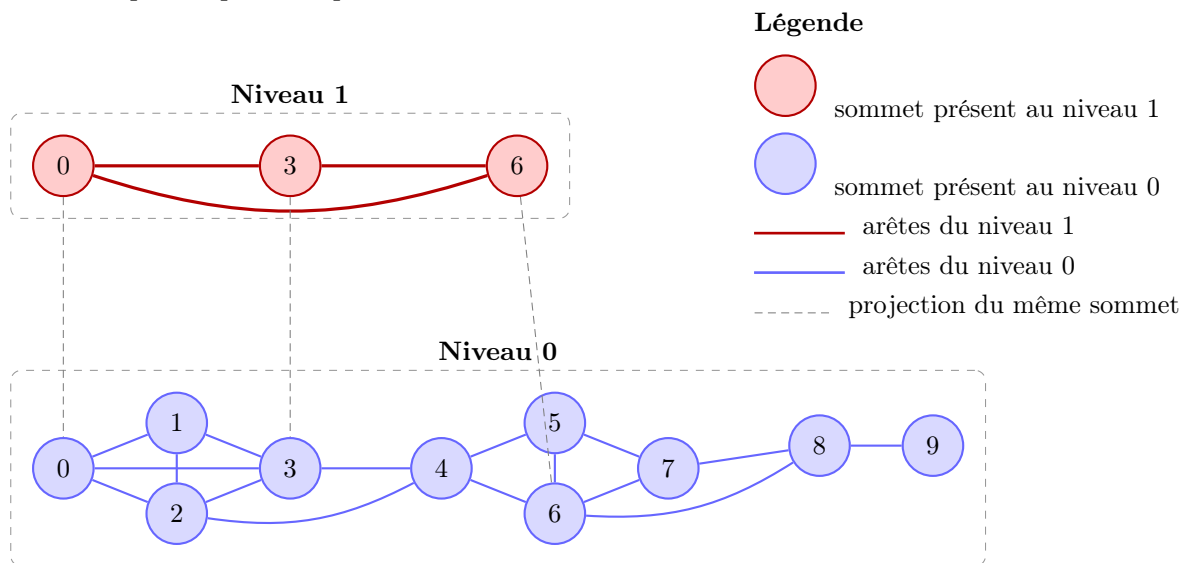
typedef struct graphe_strat graphe_strat;

```

1. Ecrire une fonction `graphe_strat* init(int M,int ef,int dim,float ml,int n,int k)` qui initialise un graphe multicouche vide permettant de stocker jusqu'à `n` vecteurs avec au plus `k` couches.
2. Ecrire une fonction `void libere_graphe(graphe_strat* g)` qui libère tout l'espace mémoire d'un graphe multicouche passé en entrée. Cette fonction aura la responsabilité de libérer les vecteurs stockés dans le graphe.

Dans ce sujet, on utilisera le graphe multicouche suivant à deux couches et dix sommets dont vous trouverez une implémentation dans le fichier de code fourni.

Cet exemple simple vous permettra de tester vos fonctions.



Afin d'insérer un nouvel élément dans un graphe multicouches, on utilisera des parcours gloutons de chacune des couches du graphe à la recherche d'un sommet dont tous les voisins sont

plus éloignés du point à insérer qu'il ne l'est. On se déplacera d'un sommet à un autre en prenant le voisin qui réalisera la plus courte distance au point à insérer.

3. Ecrire une fonction `int parcours(graphe_strat* g, struct vecteur v, int h, int ep)` qui effectue une recherche gloutonne dans la couche d'indice `h` de `g` à partir du sommet `ep` afin de déterminer un sommet qui réalise un minimum local vis-à-vis de ses voisins au sens de sa distance à `v`. La fonction commence au sommet `ep` et se déplace à chaque étape vers le voisin du sommet courant le plus proche de `v` à condition qu'il améliore cette distance par rapport à celle du sommet courant, le programme s'arrête et renvoie le sommet courant quand on a atteint un minimum local. Le fichier fourni vous propose quelques vecteurs avec lesquels tester cette fonction dans le graphe représenté ci-dessus.

Nous allons maintenant décrire de manière bien plus détaillée la procédure d'insertion d'un nouveau vecteur dans un graphe multicouche.

Afin d'insérer un vecteur x , on commence par déterminer la hauteur à laquelle il se trouvera dans le graphe en utilisant la fonction suivante :

```
static inline float frand01(void) {
    return (float)rand() / (float)RAND_MAX;
}

static int choix_hauteur(float ml) {
    float u = frand01();
    while (u <= 1e-9f) u = frand01();
    int lvl = (int)floorf(-logf(u) * ml);
    return lvl;
}
```

La procédure d'insertion se décomposera alors en deux étapes.

Une première étape de parcours des différentes couches du graphe qui sont supérieures à celle à partir de laquelle x sera réellement inséré qui permettra de passer d'étage en étage via un point d'accès le plus adapté en fonction de la distance de x aux données contenues dans chacune de ces couches. Pour cela, on effectuera un parcours glouton de chaque étage en choisissant à chaque étape le voisin le plus proche de x et en s'arrêtant dès que l'on trouve un minimum local qui sera alors le point d'accès pour la couche juste en dessous.

La deuxième étape de l'algorithme consistera aussi en un parcours glouton de chaque couche jusqu'à la couche de hauteur 0 mais on devra en plus insérer x dans ces couches et donc lui choisir des voisins dans chacune d'elles. La fonction prendra en entrée un argument $M_{connect}$ qui correspond à ce nombre de voisins à choisir dans chacune des couches. Pour cela, on adapte le parcours de l'étape précédente en un parcours glouton retenant à tout moment les ef plus proches voisins de x rencontrés dans le parcours. On arrêtera le parcours lorsqu'on trouve un vecteur qui est plus éloigné de x que ne l'est le plus éloigné de l'ensemble des sommets sélectionnés. A la fin de ce parcours, on sélectionne les $M_{connect}$ vecteurs les plus proches de x dans l'ensemble sélectionné pour être les voisins de x dans la couche courante (on aura $M_{connect} \leq ef$ mais la condition d'arrêt peut faire qu'on ait trouvé moins de $M_{connect}$ sommets lors du parcours de la couche et dans ce cas tous les sommets sélectionnés deviendront voisins de x). On doit ensuite s'assurer de ne pas dépasser le degré maximal M et donc on conservera les meilleurs voisins de chaque sommet parmi l'ensemble de tous les voisins obtenus (union des anciens voisins et de ceux ajoutés lors de l'insertion de x dans la couche). Pour entrer dans la couche du dessous, on utilisera le point le plus proche de x croisé dans la couche que l'on vient de parcourir. Enfin, si l'insertion a créé une nouvelle couche alors il ne faut pas oublier de mettre à jour le point d'entrée du graphe ainsi que son nombre de couches.

Vous trouverez ci-dessous le pseudo-code qui correspond à cette description (on supposera disposer d'une fonction `SELECT-NEIGHBORS(q, E, M)` qui renvoie l'ensemble des M éléments les plus proches de q dans l'ensemble E).

Algorithme 1 : INSERTION($hns w, q, M_{connect}, M, ef, m_L$)

Entrée : graphe multicouche $hns w$, nouvel élément q , nombre de connexions à établir $M_{connect}$, nombre maximal de connexions par couche M , taille de la liste dynamique ef , facteur de normalisation m_L

Sortie : mise à jour de $hns w$ par insertion de l'élément q

$W \leftarrow \emptyset;$ // liste des éléments proches trouvés

$ep \leftarrow$ obtenir le point d'entrée de $hns w$;

$L \leftarrow$ niveau de ep ; // couche supérieure du graphe

$l \leftarrow \lfloor -\ln(\text{unif}(0, 1)) \cdot m_L \rfloor;$ // niveau du nouvel élément

pour $l_c \leftarrow L$ **to** $l + 1$ **faire**

$W \leftarrow \text{SEARCH-LAYER}(q, ep, ef = 1, l_c);$

$ep \leftarrow$ obtenir dans W l'élément le plus proche de q ;

pour $l_c \leftarrow \min(L, l)$ **to** 0 **faire**

$W \leftarrow \text{SEARCH-LAYER}(q, ep, ef, l_c);$

$voisins \leftarrow \text{SELECT-NEIGHBORS}(q, W, M_{connect})$

 ajouter des connexions bidirectionnelles entre q et chaque élément de $voisins$ à la couche l_c ;

pour chaque $e \in voisins$ **faire**

$eConn \leftarrow$ voisinage(e) à la couche l_c ;

si $|eConn| > M$ **alors**

$eNewConn \leftarrow \text{SELECT-NEIGHBORS}(e, eConn, M)$

 définir le voisinage de e à la couche l_c comme $eNewConn$ et mettre à jour les voisinages des voisins modifiés;

$ep \leftarrow$ obtenir dans W l'élément le plus proche de q ;

si $l > L$ **alors**

 définir le point d'entrée de $hns w$ comme q et tous les nouveaux niveaux qui ne contiendront que q ;

Algorithme 2 : SearchLayer(q, ep, ef, lc)

Entrée : élément requête q , points d'entrée ep , nombre d'éléments proches à retourner ef , numéro de couche lc

Sortie : ef plus proches voisins de q

$v \leftarrow ep;$

$C \leftarrow ep;$ // ensemble des candidats

$W \leftarrow ep;$ // liste dynamique des plus proches voisins trouvés

tant que $|C| > 0$ **faire**

$c \leftarrow$ extraire de C l'élément le plus proche de q ;

$f \leftarrow$ obtenir dans W l'élément le plus éloigné de q ;

si $\text{distance}(c, q) > \text{distance}(f, q)$ **alors**

interrompre ; // tous les éléments de W ont été évalués

pour chaque $e \in \text{voisinage}(c)$ à la couche lc **faire**

si $e \notin v$ **alors**

$v \leftarrow v \cup \{e\};$

$f \leftarrow$ obtenir dans W l'élément le plus éloigné de q ;

si $\text{distance}(e, q) < \text{distance}(f, q)$ **ou** $|W| < ef$ **alors**

$C \leftarrow C \cup \{e\};$

$W \leftarrow W \cup \{e\};$

si $|W| > ef$ **alors**

 supprimer de W l'élément le plus éloigné de q ;

retourner W ;

4. En vous appuyant sur le pseudo code fourni, écrire une fonction `int* parcours_couche_pq(graphe_strat* g, struct vecteur q, int entry, int h, int* out)`. Cette fonction renvoie un tableau dont la taille sera stockée à l'adresse `out` qui contiendra les éléments trouvés dans la couche `h` de `g`. On parcourera cette couche depuis le point `entry` pour trouver les points "proches" de `q`. Vous pourrez utiliser la librairie de file de priorité fournie.
5. Écrire une fonction `void insere(graphe_strat* g, struct vecteur v, int Mc)` qui insère le vecteur `v` dans le graphe `g` en lui associant jusqu'à `Mc` voisins dans chaque couche où il sera inséré. On pourra supposer ici que le graphe n'a pas atteint son nombre maximum de sommets et que l'insertion est donc légale.
6. Adapter les idées de l'insertion à la recherche des `k`-plus proches voisins.

Source : Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs Yu. A. Malkov, D. A. Yashunin

3 Audit

Dans cette partie de l'épreuve, il est demandé au candidat d'étudier un fichier source qui comporte des erreurs ou des maladresses, de qualité de code ou de fonctionnement, et il est demandé d'auditer ce fichier, c'est-à-dire :

- de comprendre et d'être capable d'expliquer le fonctionnement du code à l'oral ;
- de proposer des corrections en réécrivant certaines parties afin de corriger les erreurs ou maladresses éventuelles et de rendre le code plus clair, notamment dans une optique pédagogique ;
- de proposer des améliorations de la complexité en temps ou en mémoire, ou de la sûreté du code.

Le temps indicatif de préparation de cette partie est d'une heure.

Partie OCaml Pour le programme OCaml ci-dessous, les consignes sont les suivantes :

- Définir un type pour représenter un arbre qui peut être soit une feuille entière, soit un nœud avec un arbre gauche, une valeur entière et un arbre droit.
- Définir une fonction `remplacer : int -> int -> arbre -> arbre` qui remplace toutes les occurrences d'un élément par une nouvelle valeur.

```
type arbre = Feuille of int
           | Noeud of arbre , int , arbre ;;

let rec remplacer elem nvelem =
  let aux_remp a = remplacer a elem nvelem in
  function
  | Feuille i -> if i==elem then Feuille nvelem
  | Noeud (ag,elem,ad) -> Noeud (aux_remp ag, nvelem, aux_remp ad)
  | Noeud (ag,i,ad) -> Noeud (aux_remp ag, i, aux_remp ad)
;;
```

Partie Python Pour le programme Python ci-dessous, la consigne est la suivante :

- Définir une fonction `min_somme` qui calcule la somme minimale des voisins d'une case dans une matrice deux dimensions d'entiers. La somme des voisins d'une case `c` est la somme des cases directement voisines (orthogonales et diagonales) sans compter l'entier de la case `c`.

```

mat = [[0]*10]*10

def init(mat) :
    for lig in range(10) :
        for col in range(10) :
            mat[lig][col] = randrange(100)

def somme_prox (mat, case) :
    l,c = case
    for lig in range (l-1,l+2) :
        for col in range(c-1,c+2) :
            somme += mat[lig][col]
    return somme

def min_somme(mat) :
    som_min = 0
    for l in range(10) :
        for c in range(10) :
            som_loc = somme_prox(mat,(l,c))
            if som_loc < som_min :
                som_min = som_loc

```