LYCÉE DU HAINAUT
Apprendre pour entreprendre

IA – TP07

2024-2025 – 2 heures

Algorithme des K-moyennes

Objet de l'étude

Parmi les méthodes d'apprentissage en intelligence artificielle (IA), l'étude proposée met en œuvre progressivement l'algorithme d'apprentissage non supervisé par la méthode des K-moyennes (fr), nommée *k-means* en anglais.

Outils utilisés

- Environnement de programmation EduPython.
- Ce quoi rédiger sur feuille pour préserver ses recherches personnelles. Un conseil, gardez les tentatives et les échecs pour mieux jalonner votre progression en annotant les essais avortés par les raisons de l'échec. Cette pratique permet de mieux comprendre son/ses erreur(s) dans le but de ne plus les commettre.
- Utiliser les documents de cours qui rappellent les définitions.

Organisation

Il est rappelé de conserver une copie des documents fournis pour ce TP, ainsi que ceux produits, en les enregistrant dans un dossier de son espace de stockage personnel Moodle associé à son compte utilisateur.

Et surtout, **dans un second tempse et par sécurité**, garder aussi une copie de ces fichiers sur une clé de stockage (et éventuellement aussi dans un dossier dans le nuage). Parce qu'une sauvegarde est toujours préférable afin d'éviter de perdre ses données en cas de problème.

Il est aussi conseillé de sauvegarder régulièrement son script en cours de travail (CTRL-S), même si l'éditeur le fait à chaque exécution.

Les deux heures attribuées à ces activités seront assurément insuffisantes pour acquérir toutes les connaissances abordées, c'est pourquoi un travail personnel en dehors des séances est nécessaire.

Compte rendu

Il n'est pas demandé de rendre compte rendu, mais il reste essentiel de noter ses remarques et certaines procédures pour « garder une trace » et se rappeler les résultats essentiels en cas de besoin.

I. Introduction

I.1. Rappels de quelques définitions (Cf. glossaire, annexe du cours introductif)

L'**apprentissage machine** (en, *machine learning*) est la phase durant laquelle un système d'intelligence artificielle (IA) construit ses bases de données et de connaissances en vue des usages visés.

La **classification** (*classification*) permet d'associer (ou prédire) un ensemble de données acquises à une classe d'apprentissage pour regrouper des données partageant des critères communs. Par exemple, une espèce pour les fleurs, une race pour les chiens ou les chats, une catégorie pour les panneaux de circulation (signalisation d'obstacles triangulaires, d'interdiction cerclées de rouge, de localisation sur fond blanc, indications sur fond bleu, etc.).

L'**apprentissage supervisé** (*supervised learning*) est un mode d'apprentissage machine qui consiste à associer des données d'apprentissage à une classe déjà définie. Ces données sont appelées « données étiquetées » car elles sont repérées comme associées à la classe connue (l'étiquette).

L'**apprentissage non supervisé** (*unsupervised learning*) est un autre mode d'apprentissage machine réalisé avec des données non étiquetées, c'est-à-dire non déjà associées à une classe. C'est le cas par exemple d'images d'animaux qui n'ont pas déjà été associées à une race par une légende ou autres informations données avec l'image.

La classification résulte de phases d'apprentissage qui sont basées sur des algorithmes d'apprentissage dont seuls deux types nous concernent :

1. Les **algorithmes** de **classification supervisée** qui permettent de prédire la classe d'une nouvelle donnée sur la base de classes des données d'entraînement étiquetées.

   Exemple : algorithme des k plus proches voisins ou k-nn, *k-nearest neighbors* (en).

2. La **classification non supervisée** où aucune classe n'est connue car aucune données d'entraînement n'a été fournie au préalable. Les classes sont donc créées ex-nihilo.

   Exemple : algorithme des k-moyennes, *k-means* (en).

Pour ce TP, nous allons mettre en œuvre progressivement l'algorithme des k-moyennes.

I.2. Présentation et précision concernant les données

En règle générale, pour les systèmes d'IA, les données sont les composantes de vecteurs d'un espace vectoriel (e.v.) munie d'une base de dimension finie : un vecteur = 1 donnée. Les composantes sont donc des paramètres ou des valeurs qui sont attachées ou propres à la donnée.

Ces composantes ne sont pas nécessairement homogènes : certaines sont des valeurs relatives à des grandeurs physiques (homogènes ou pas, d'unités différentes ou pas), d'autres des états de vérité (vrai/faux), des textes, etc. L'ensemble des données de l'e.v. forme des « nuages de points ». Pour pouvoir manipuler ces données de manière homogène, le plus souvent pour les comparer afin de les associer à une classe ou une autre, les composantes sont normalisées entre une valeur minimale et une maximale, par exemple entre –1 et 1 si elles sont signées ou 0 et 1 en non signé.

Pour pouvoir représenter les données, nous les supposerons normalisées et nous nous limiterons à une projection de ces nuages dans un plan, c'est-à-dire un s. e. v. de dimension 2 comme le montre l'exemple sur la Figure 1.

Les composantes des données y ont été normalisées approximativement dans l'intervalle $[\sim -4, \sim 4]$.

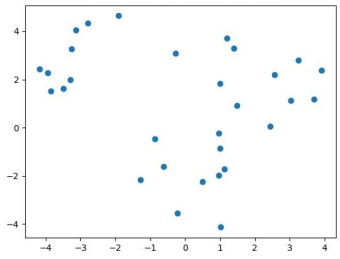

Figure 1

II. Préparation de données à manipuler

Pour notre étude, nous allons d'abord construire un nuage de points en dimension 2 reopresentant des données afin de leur appliquer l'algorithme. Les points seront choisis aléatoirement, mais pas de manière homogène pour laisser apparaître des regroupements « visibles ».

Pour générer les données, un premier script est donné sur la page suivante.

```
1  dim, nb = 2, 20 # Quelle dimension ? Combien de données ?
2  # Génération de paquets de points en agglomérant des données
3  X = np.vstack([np.array(p + np.random.randn(nb, dim))
4                 for p in [[3, 2], [0, -2], [-3, 3], [-3, -3]]]).tolist()
5  # Commenter en expliquant cette instruction utilisant vstack et .tolist
```

Q.1 Recopier ce script en lui ajoutant un entête pour **identifier** son contenu, puis adapter le commentaire de la ligne 1 précisant les réponses aux questions indiquées.

Q.2 En commentaire de la ligne 5, expliquer les lignes 3 et 4 en précisant où seront agglomérées les données « en paquets » (consulter `numpy.org`). Préciser le type de la variable `X`.

Q.3 Compléter le script pour éviter toute erreur lors de l'exécution.

Pour visualiser les données, nous complétons le script précédent avec les lignes suivantes.

```
6  # Laisser en l'état : cette ligne sera complétée plus tard
7  plt.scatter([x[0] for x in X], [x[1] for x in X]) # Expliquer
8  # Cette ligne sera complétée plus tard
9  # Cette ligne sera complétée plus tard
10 plt.title('Tirage de ' + str(nb) + ' points initiaux aléatoires')
11 plt.savefig('kmeans_init.pdf') # Pour garder le résultat (optionnel)
12 plt.show()
```

Q.4 Expliquer la ligne 7, puis adapter et exécuter le script pour juger du résultat (pas que « voir »).

III. Installation des centres, évaluation et caractérisation de leur position

Q.5 Nous allons étudier notre exemple en considérant 4 classes *a priori*. Compléter la ligne 1 en conséquence en initialisant le nombre de classes `K`.

Les centres sont initialement placés au hasard. Ils sont créés et affichés en complétant le script précédent pour obtenir la version finale ci-dessous.

```
1  dim, nb, K = 2, 20, 4 # Quelle dimension ? Combien de classes/données ?
2  # Génération de paquets de points en agglomérant des données
3  X = np.vstack([np.array(p + np.random.randn(nb, dim))
4                 for p in [[3, 2], [0, -2], [-3, 3], [-3, -3]]]).tolist()
5  # Commenter (expliquer)
6  centres = (np.random.rand(K, dim)*6 - 3).tolist() # Placement aléat. centres
7  plt.scatter([x[0] for x in X], [x[1] for x in X]) # Tracé des points
8  plt.scatter([x[0] for x in centres], [x[1] for x in centres],\
9              marker='x', s=100, c='b') # Tracé des centres initiaux
10 plt.title('Tirage de '+str(nb)+' points initiaux aléatoires')
11 plt.savefig('kmeans_init.pdf') # Pour garder le résultat (optionnel)
12 plt.show()
```

Q.6 Modifier puis exécuter le script et juger du résultat.

Modifier le marquage des centres en utilisant des marqueurs rouges (markers) « *diamond* » (♦).

Exécuter plusieurs fois le script (sans enregistrer la figure) pour constater visuellement que la position des centres change.

La position des K centres évolue en calculant leur position à partir de celle des données. Il faut donc évaluer l'isobarycentre (poids unitaires) d'un ensemble de n vecteurs x_i ($i \in [1,n]$) défini comme suit :

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Les coordonnées de $\overline{x}$ sont donc les moyennes des coordonnées des x_k.

Q.7 Écrire un script Python de la fonction dont le prototype est le suivant :

```
def centre(X:[list]) -> list:
```

qui renvoie le centre de la liste de vecteurs `X` (sachant que `X` contient des vecteurs représentés par la liste de leurs composantes).

Q.8 Une fois la fonction **`centre()`** créée, valider son fonctionnement avec ces vecteurs :

$$x_1(0,0)\ ;\ x_2(10,0)\ ;\ x_3(10,20)\ ;\ x_4(0,20)$$

À partir de la position aléatoire initiale, les centres évoluent au rythme de l'apprentissage puisque des vecteurs changent de regroupement en fonction d'un critère de minimisation des distances par rapport à chaque centre (Cf. cours, diapositive **D6**).

Ce critère se ramène à l'évaluation d'une quantité assimilée à la variance $V(X)$, ou moment d'inertie, d'un ensemble X de vecteurs x définie par :

$$V(X) = \sum_{x \in X} d(x, \overline{X})^2$$

La distance d exprime la norme dans la définition présentée dans **D6**, tandis que la variance mesure la variation par rapport à la moyenne : plus $V(X)$ est petit, plus les vecteurs x de X sont proches du barycentre $\overline{X}$.

> **Remarque** : la définition statistique définit la variance en divisant par le nombre d'échantillons. Dans notre étude, cette opération n'est pas nécessaire car $V(X)$ fournit juste un critère de comparaison pour l'algorithme : la division ne modifie en rien les comparaisons puisque le nombre de données est invariant. Cela économise une division dans le calcul.

Évaluation de la distance

Q.9 Écrire un script Python de la fonction dont le prototype est le suivant :

```
def d(x, y:list) -> float:
```

qui renvoie la distance euclidienne entre les vecteurs `x` et `y`.

Une fois la fonction créée, la valider avec quelques couples parmi ceux de **Q.8**.

Préciser la complexité de cette fonction.

Évaluation de la variance

Cette fonction est établie à titre indicatif car dans l'implantation de l'algorithme, la variance sera recalculée à partir de sa définition pour réduire le temps d'exécution du script en favorisant sa complexité.

Q.10 Écrire un script Python de la fonction dont le prototype est le suivant :

```
def V(X:[list]) -> float:
```

qui renvoie la variance des vecteurs de `X`.

Préciser la complexité de cette fonction et la comparer à celle de **`d()`**. Conclure.

IV. Algorithme des K-moyennes

IV.1. Mise en place

K étant fixé arbitrairement, pour l'instant, le but de la méthode est de construire les K classes partitionnant l'ensemble des données disponibles.

Mais la méthode (Cf. **D6**) nécessite d'établir un optimum de la somme de toutes les variances en la minimisant. Nous poursuivons donc l'objectif de trouver un partitionnement (en, *clustering*) de l'ensemble des données X en K sous-ensembles $X_1, \ldots, X_K$ appelés classes (en, *clusters*) minimisant la quantité I, appelé **inertie**, définie par :

$$I = \sum_{i=1}^{K} V(X_i)$$

Exprimé autrement, nous voulons associer à chaque donnée x une classe k (attention, minuscule, avec $k < K$) telle que l'inertie I soit minimale. De cette manière, plus l'inertie est petite, plus les données sont proches du centre de leur classe et donc plus le partitionnement est pertinent (considéré « bon »).

Pour y parvenir, nous suivons la démarche suivante :

1. À partir des vecteurs $c_1, \ldots, c_K$ choisis aléatoirement (**Q.5**).
2. Associer chaque donnée x à la classe X_i telle que $d(x, c_i)$ soit minimale.
3. Recalculer les centres des classes, $c_i = \overline{X_i}$.
4. Si les centres se sont déplacés, revenir à l'étape 2.

Sans le démontrer, nous affirmons que l'algorithme des K-moyennes se termine (pas de boucle infinie), car l'inertie I décroît strictement à chaque itération (passage 4→2). Et comme cette quantité ne peut prendre qu'un nombre fini de valeurs, alors le nombre d'itérations est fini.

Par conséquent :

- Il existe un nombre fini de partitions de X en K classes, donc l'inertie I ne peut prendre qu'un nombre fini de valeurs ;
- Il suffit alors de montrer que I décroît strictement (vers zéro) à chaque itération.

IV.2. Implantation progressive

Nous commençons par établir les K centres des classes, opération nécessaire à chaque étape.

Q.11 Écrire un script Python de la fonction dont le prototype est le suivant :

```
def calculer_centres(classes:[list]) -> list:
```

qui reçoit K classes et renvoie la liste des centres de chacune d'elle. Il faudra donc initialiser une liste vide des centres, puis la remplir avec les coordonnées du centre évaluées avec la fonction **`centre()`** (**Q.7**).

Préciser la complexité de cette fonction.

Ensuite il faut associer un vecteur à une classe en déterminant le centre le plus proche.

Q.12 Écrire un script Python de la fonction dont le prototype est le suivant :

```
def plus_proche(x, centres: list]) -> int:
```

qui reçoit un vecteur `x` et la liste des centres (à l'état actuel) pour renvoyer l'indice de la classe la plus proche de `x` parmi celles contenues dans `centres`.

Cette fonction met en œuvre une recherche de minimum des distances calculées avec la fonction **`d()`** (**Q.9**) pour tous les centres donnés en argument.

Cette fonction sera appelée à chaque étape pour à associer un vecteur à la bonne classe.

Préciser la complexité de cette fonction.

Q.13 La construction des classes de vecteurs les plus proches de chaque centre peut débuter en rédigeant un script Python de la fonction dont le prototype est le suivant :

```
def calculer_classes(X:[list], centres: list) -> list:
```

qui reçoit l'ensemble `X` de toutes les données (contenant tous les vecteurs `x`) et la liste des centres pour renvoyer une liste `classes` telle que `classes[i]` soit la liste des données de `X` dont le centre le plus proche est `centres[i]`.

Pour cela, il faut initialiser une liste `classes` de listes vides de taille le nombre de centres, puis sélectionner chaque vecteur `x` de `X` pour lui attribuer la classe la plus proche (**Q.12**).

Préciser la complexité de cette fonction.

Nous sommes parvenus au terme de la préparation. Par 4, il faut maintenant boucler les étapes pour évaluer le déplacement de chaque centre et s'arrêter quand ils ne bougent plus puisque la convergence est garantie.

Q.14 Écrire un script Python de la fonction dont le prototype est le suivant :

```
def kmeans(X, centres: [list]) -> [list]:
```

qui reçoit l'ensemble `X` des données et la liste des centres pour renvoyer la liste des sous-ensemble de `X` qui constituent les K classes résultant de l'apprentissage.

La structure globale de cette fonction s'appuie sur une boucle vérifiant une condition de non évolution des centres (comparaison de la position des centres entre deux étapes consécutives) après avoir initialisé un centre initial à `None` pour fait office de centre antérieur.

À titre final, déterminer le coût algorithmique de la méthode des K-moyennes.

V. Évaluation de l'efficacité de l'algorithme

Pour envisager l'efficacité de l'algorithme, il faut utiliser la décroissance de l'inertie I à chaque étape, et donc implanter une fonction permettant son calcul.

Q.15 Écrire un script Python de la fonction dont le prototype est le suivant :

```
def inertie(classes, centres: [list]) -> float:
```

qui renvoie l'inertie de l'ensemble des classes constituées par **`kmeans()`**.

Pour mettre en lien la valeur de l'inertie avec le résultat obtenu en termes de classification, nous allons reprendre le tracé du nuage de points en distinguant les points par des couleurs différentes suivant leur classe d'appartenance.

Q.16 Reprendre et adapter le script de **Q.5** pour tracer à nouveau le nuage de points en leur attribuant une couleur différente suivant leur classe d'appartenance et en affichant l'inertie obtenue dans le titre du graphe (après le texte `'Inertie ='`) comme le montre le script ci-dessous, à recopier et compléter dans son éditeur.

```
# Les paramètres dim, nb, K ont déjà été définis au début du script #...
# ... est un commentaire à remplacer par l'instruction demandée
classes = # ... Remplacer par l'instruction générant les classes
cmap = ListedColormap(['r', 'g', 'b', 'purple']) # carte des couleurs
centres = # ... Instruction de calcul les centres à partir des classes
plt.figure()
plt.scatter([x[0] for x in X], [x[1] for x in X], \
            c=[plus_proche(x, centres) for x in X], cmap=cmap)
plt.scatter([x[0] for x in centres], [x[1] for x in centres], \
            marker='x', s=100, c=range(K), cmap=cmap)
# ... Instruction pour générer et afficher le titre
plt.savefig('kmeans_resultat_classes.pdf') # Garder le tracé (optionnel)
plt.show()
```

Le nombre K de classes est un paramètre essentiel de l'algorithme des K-moyennes. Le tracé précédent a été effectué en l'imposant arbitrairement à 4, mais sans se soucier si cette valeur permettait une bonne classification, voire la meilleure. Celle obtenue est caractérisée par une certaine inertie, mais un autre choix de valeur pour K donnerait une autre valeur d'inertie qui serait plus faible puisque la fonction $I(K)$ est strictement décroissante (Cf. §IV.1). Mais, il n'est pas judicieux de trop morceler la classification (le pire étant de construire autant de classes que de données, même si ça c'est facile 🤔), alors il est bon de se demander « quelle est le nombre de classes optimal ? ».

La partie suivante a pour but d'y répondre.

VI. Recherche du nombre de classes K optimal

La méthode proposée consiste à déterminer un K, ni trop grand, ni trop faible, pour assurer une classification acceptable, optimale dira-t-on (attention à l'usage de ce terme souvent mal à propos). Elle est connue sous le nom de « méthode du coude » (*elbow method*). Elle consiste à fixer progressivement de manière croissante le nombre de classe K, donc de centres initiaux, puis à examiner la représentation graphique de la fonction $I(K)$ pour constater que deux domaines se distinguent (Figure 2) :

- dans le premier la décroissance est rapide ;
- dans le second, la décroissance devient de plus en plus faible en tendant vers zéro.

La courbe montre une frontière entre deux domaines : c'est là que le nombre de classes est choisi, en relevant la valeur de K (entière) au niveau du coude, par exemple par prolongement linéaire jusqu'à l'axe $I = 0$.

Q.17 Écrire un script Python permettant de tracer la représentation graphique de $I(K)$ pour des valeurs entières de K n'excédant pas 7 (au-delà, c'est inutile).

Il faudra pour cela structurer son script autour d'une boucle d'indice `K`, puis de calculer la valeur de l'inertie `I` du partitionnement obtenu (avec **`kmeans`**`()`, **`calculer_centres`**`()` et **`inertie`**`()`). Chaque valeur sera stockée dans la liste `I_lst` pour enfin afficher l'évolution de $I(K)$.

La valeur du meilleur `K` possible est déterminée par lecture du graphe.

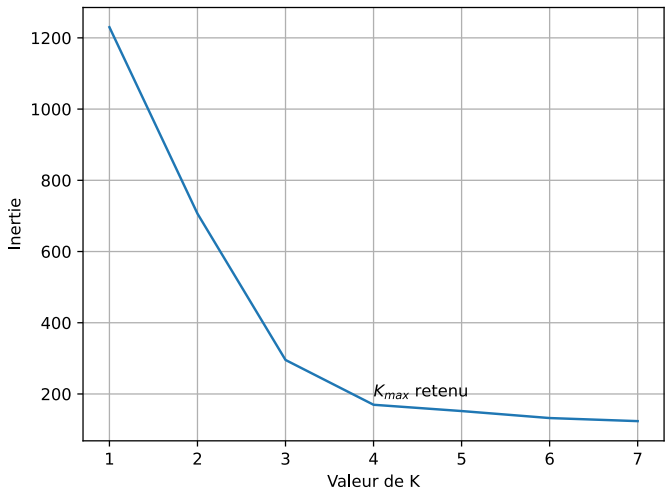

Figure 2 – Courbe du coude.

Prolongement possible de l'étude

Il pourrait être envisagé d'appliquer l'algorithme à une classification de fleurs, d'animaux, de caractères pour un usage dans un logiciel de reconnaissance de caractères (OCR, *optical character recognition*) ou de panneaux de la circulation, tous obtenus à partir de photographies.

Les données ne seraient alors pas des valeurs aléatoires, mais des paramètres issus d'analyses des fichiers photographiques : histogrammes suivant la hauteur et la largeur pour en déterminer la valeur moyenne de chaque couleur suivant les axes (R, V, B), nombres de pixels par rapport à un seuil, etc. c'est-à-dire des paramètres qui caractérisent la photographie, donc le sujet contenu, pour constituer les composantes des vecteurs de données.