Intelligence artificielle

March 21, 2024

1 Définition

L'intelligence artificielle (IA) désigne la capacité des machines à effectuer des tâches qui nécessitent normalement l'intelligence humaine. Cela inclut des domaines tels que la résolution de problèmes, la reconnaissance de formes, l'apprentissage, la compréhension du langage naturel et la prise de décision.

2 Techniques d'intelligence artificielle

Il existe plusieurs techniques d'IA, on en cite les 3 les plus connues :

- Apprentissage machine (Machine Learning) : Une méthode d'IA qui permet aux machines d'apprendre à partir de données sans être explicitement programmées. Cela inclut des techniques telles que les réseaux de neurones artificiels, les arbres de décision, les SVM (machines à vecteurs de support), etc;
- Réseaux de neurones artificiels (ANN) : Inspirés du fonctionnement du cerveau humain, ces réseaux sont composés de nombreuses unités de traitement interconnectées (neurones) qui apprennent à partir de données;
- Apprentissage profond (Deep Learning) : Une branche du Machine Learning qui utilise des réseaux de neurones artificiels à plusieurs couches pour apprendre des modèles de données complexes.

3 Apprentissage machine (Machine Learning)

3.1 Apprentissage supervisé (Méthode KNN)

KNN, ou k-plus proches voisins (k-nearest neighbors), est un algorithme d'apprentissage supervisé utilisé pour la classification et la régression. C'est l'un des algorithmes les plus simples et les plus utilisés en apprentissage machine.

L'idée de base derrière l'algorithme KNN est de trouver les k points les plus proches d'un point donné dans l'espace des caractéristiques (espace de données). Une fois que les k voisins les plus proches sont identifiés, la classe (dans le cas de la classification) ou la valeur (dans le cas de la régression) du point donné est déterminée en fonction des classes ou des valeurs des voisins les plus proches.

3.1.1 Exemple d'application

Supposons avoir une base de données des patients qui ont consulté les médecins d'un hôpital. Les patients effectuent 4 analyses et selon les résultats de ces quatres analyses le medecin décide si le patient souffre de la maladie A, ou de la maladie B ou si le patient est sain.

Les données sont organisées comme suit :

Patient	Analyse 1	Analyse 2	Analyse 3	Analyse 4	Résultat
Patient 1	66	77	52	73	Maladie A
Patient 2	42	16	71	34	Maladie B
Patient 3	72	53	59	62	Sain
Patient 4	53	16	36	24	Sain
Patient 5	37	24	68	65	Maladie B
Patient 6	88	47	16	29	Maladie A
Patient 7	76	85	79	70	Maladie A
...	...	...	...	...	...
...	...	...	...	...	...
Patient n	12	56	6	8	Maladie B

Supposons que nous disposons d'une matrice (une liste de listes) ou chaque ligne (liste) dans cette matrice contient les données enregistrés d'un patient sous la forme suivante [Analyse 1, Analyse 2, Analyse 3, Analyse 4, etat du patient (A, B, S)].

Nous devons implémenter la fonction d'algorithme KNN qui prendra en paramètre la matrice de données, les résultats d'analyses d'un nouveau patient, et la constante k. La fonction doit retourner une prédiction de l'état du nouveau patient.

[5]:
```
# Exemple de la matrice de données

# data = [[66, 77, 53, 73, 'A'], [42, 16, 71, 34, 'B'], [72, 53, 59, 62, 'S'], .
 ↪....., [12, 56, 6, 8, 'B']]

# Exemple des résultats des analyses du nouveau patient

# new = [67, 18, 5, 3]
```

Pour faire simple, nous commençons par écrire une fonction NN(data, new) qui prend en paramètre la matrice de données "data" et les résultats d'analyses du nouveau patient stockés dans la liste "new". Dans cette fonction on considere que k = 1. La fonction NN doit retourner une prédiction de l'état du nouveau patient ('A', 'B' ou 'S')

Nous définissons d'abord une fonction distance_euclidienne(V, W) qui prend deux vecteurs (listes) de même dimension V et W et qui retourne la distance euclidienne entre eux. Cette fonction nous permettera de calculer la distance entre le vecteur qui représente les résultats des analyses du nouveau patients et tous les vecteurs qui font parties de notre liste data (Les patients dont on connait l'état).

[6]:
```
# La distance euclidienne est la racine carrée de la somme des carrés des écarts

def distance_euclidienne(V, W):
    s = 0
    for i in range(len(V)):
        s = s + (V[i] - W[i]) ** 2
    return s ** 0.5
```

Aprés on écrit une fonction tri(L) qui prend en paramètre une fonction L qui contient des tuples. Chaque tuple représente la distance entre le vecteur du nouveau patient et un vecteur de la liste data, le tuple est sous la forme (distance, état 'A' ou 'B' ou 'S'). La fonction trie la liste L dans l'ordre croissant des distances (Des premiers éléments des tuples).

Nous allons utiliser les tuples pour mémoriser l'état du patient avec lequel on calcule la distance, supposons que la distance entre le patient 2 et new est d et que l'état du patient 2 est maladie B, le tuple qui représente la distance sera (d, 'B').

[7]:
```
# Nous avons utilisé le tri par sélection adapté à une liste de tuples

def tri(L):
    for i in range(len(L) - 1):
        k = i
        for j in range(i + 1, len(L)):
            if L[j][0] < L[k][0]:
                k = j
        L[i], L[k] = L[k], L[i]
```

Finalement nous définissons la fonction NN.

[8]:
```
def NN(data, new):
    # La liste qui contiendra les tuples des distances
    D = []

    for x in data:
        # Construire le tuple de la distance et l'ajouter dans D
        D.append((distance_euclidienne(x[:-1], new), x[-1]))

    # Trier D dans l'ordre croissant des distances
    tri(D)

    # Retourner l'état correspondant à la distance minimale
    return D[0][1]
```

Nous écrivons maintenant la fonction KNN(data, new, k) où k est un paramètre variable.

[9]:
```
def KNN(data, new, k):
    # La liste qui contiendra les tuples des distances
    D = []
```

```
    for x in data:
        # Construire le tuple de la distance et l'ajouter dans D
        D.append((distance_euclidienne(x[:-1], new), x[-1]))

    # Trier D dans l'ordre croissant des distances
    tri(D)

    # Prendre les k plus petites distances
    KD = D[:k]

    # Nous devons savoir l'état dominant dans la liste KD
    # C'est à dire l'état dont l'occurence est maximale
    # Les éléments de KD sont aussi des tuples de distances

    # Construire le dictionnaire dont les clés sont les états et les valeurs␣
↪sont leurs occurences
    classes = {}
    for x in KD:
        if x[1] not in classes:
            classes[x[1]] = 1
        else:
            classes[x[1]] = classes[x[1]] + 1

    # Chercher l'occurence maximale
    maxocc = max(classes.values())

    # Chercher et retourner l'état convanable à cette ocuurence
    for etat in classes:
        if classes[etat] == maxocc:
            return etat
```

Normalement, avant d'éxécuter la fonction KNN, nous divisons notre data en deux parties : - 80% de data sera réservé a l'entrainemet du modele, elle sera passé en paramètre de la fonction KNN; - 20% de data nous servira pour tester et évaluer notre modèle et etablir la matrice de confusion.

La matrice de confusion est un outil utilisé en apprentissage automatique pour évaluer les performances d'un modèle de classification. Elle permet de comparer les valeurs prédites par le modèle aux valeurs réelles. La matrice de confusion est généralement une matrice de dimensions n x n ou n est le nombre de classes dans le problème de classification.

Prenons par exemple une liste de données contenant 10 000 patients :

- 8 000 patients seront réservés à l'entraînement ;
- 2 000 patients seront réservés pour tester notre modèle.

Parmi ces 2 000 patients de test, nous disposons des résultats pour chacun. Nous allons évaluer si notre fonction KNN produit des résultats cohérents pour ces 2 000 patients.

Supposons que parmi ces 2 000 patients de test, 800 sont atteints de la maladie A, 700 de la maladie B et 500 sont en bonne santé.

- Sur les 800 patients atteints de la maladie A, la fonction KNN a prédit correctement 780 comme étant atteints de la maladie A, 15 comme atteints de la maladie B et 5 comme étant en bonne santé ;
- Parmi les 700 patients atteints de la maladie B, la fonction KNN a prédit 30 comme atteints de la maladie A, 650 correctement prédit comme atteints de la maladie B et 20 comme étant en bonne santé ;
- Sur les 500 patients en bonne santé, la fonction KNN a prédit 5 comme atteints de la maladie A, 5 comme atteints de la maladie B et a correctement prédit 490 comme étant en bonne santé.

La matrice de confusion sera comme suit :

	Maladie A	Maladie B	Sein	Total	Précision
Maladie A	780	15	5	800	98%
Maladie B	30	650	20	700	93%
Sein	5	5	490	500	98%

Essayons d'ecrire une fonction qui retourne la matrice de la confusion adaptee a notre situation : 3 classes (A, B et S).

La fonction prendra en paramètre les données d'entrainement "data" qui constituent 80% de nos données, les données de test "data_test" qui constituent 20% des données et le paramètre k.

La fonction effectuera un test sur toutes les données de la liste "data_test" pour construire la matrice de confusion.

[10]:
```python
def matrice_confusion(data, data_test, k):
    # Initialiser la matrice de confusion par des 0
    confusion = [[0] * 3 for i in range(3)]

    # La liste des classes
    classes = ['A', 'B', 'S']

    # Parcourir la liste des données de test
    for x in data_test:
        # Appliquer la fonction KNN sur x
        prediction = KNN(data, x[:-1], k)

        # Classe de x
        classe_r = x[-1]

        # Chercher la case correspondante dans la matrice de confusion
        i = classes.index(classe_r)
        j = classes.index(prediction)

        # Ajouter 1 dans la case correspondante
        confusion[i][j] += 1
```

```
    # Retourner la matrice de confusion
    return confusion
```

Essayons d'ecrire une fonction qui retourne la matrice de la confusion en general (Pour le cas d'un nombre inconnues de classes).

[11]:
```
def matrice_confusion(data, data_test, k):
    # Chercher les classes
    classes = []
    for x in data:
        if x[-1] not in classes:
            classes.append(x[-1])

    # Le nombre de classes
    n = len(classes)

    # Initialiser la matrice de confusion par des 0
    confusion = [[0] * n for i in range(n)]

    # Parcourir la liste des données de test
    for x in data_test:
        # Appliquer la fonction KNN sur x
        prediction = KNN(data, x[:-1], k)

        # Classe de x
        classe_r = x[-1]

        # Chercher la case correspondante dans la matrice de confusion
        i = classes.index(classe_r)
        j = classes.index(prediction)

        # Ajouter 1 dans la case correspondante
        confusion[i][j] += 1

    # Retourner la matrice de confusion
    return confusion
```

3.2 Apprentissage non supervisé (Méthode K-means)

La méthode K-means est un algorithme de clustering largement utilisé en apprentissage non supervisé. Son idée de base est de diviser un ensemble de données en k groupes (clusters) distincts et mutuellement exclusifs.

3.2.1 Algorithme

1. Initialisation des centroïdes : Sélectionnez k points initiaux comme centroïdes de manière aléatoire. Un centroïde est le centre de gravité des points dans un cluster ;

2. Assignation des points aux clusters : Pour chaque point dans l'ensemble de données, calculez la distance entre ce point et tous les centroïdes. Assignez ce point au cluster représenté par le centroïde le plus proche ;
3. Mise à jour des centroïdes : Une fois que tous les points ont été assignés à des clusters, recalculez les centroïdes de chaque cluster en prenant la moyenne des coordonnées de tous les points appartenant à ce cluster ;
4. Répétition des étapes 2 et 3 : Répétez les étapes d'assignation des points aux clusters et de mise à jour des centroïdes jusqu'à ce que les centroïdes ne changent plus de position ou que le critère d'arrêt soit atteint (par exemple, un nombre maximum d'itérations) ;
5. Convergence : L'algorithme converge lorsque les centroïdes ne changent plus de position entre deux itérations successives.

3.2.2 Exemple d'application

Nous nous plaçons dans le même exemple traité dans le paragraphe de la méthode KNN, mais cette fois-ci on suppose que nous nous connaissons pas l'état des patients. Les données seront organisées comme suit :

Patient	Analyse 1	Analyse 2	Analyse 3	Analyse 4
Patient 1	66	77	52	73
Patient 2	42	16	71	34
Patient 3	72	53	59	62
Patient 4	53	16	36	24
Patient 5	37	24	68	65
Patient 6	88	47	16	29
Patient 7	76	85	79	70
...	...	...	...	...
...	...	...	...	...
Patient n	12	56	6	8

Supposons que nous disposons d'une matrice (une liste de listes) ou chaque ligne (liste) dans cette matrice contient les données enregistrés d'un patient sous la forme suivante [Analyse 1, Analyse 2, Analyse 3, Analyse 4].

Nous devons implémenter la fonction d'algorithme k_means qui prendra en paramètre la matrice de données et le nombre de groupes k. La fonction divisera les patients en k groupes (clusters) et retournera les k centroïdes correspondant. La fonction k_means suivera l'algorithme ci-dessus.

[12]:
```
# Exemple de la matrice de données

# data = [[66, 77, 53, 73], [42, 16, 71, 34], [72, 53, 59, 62], ......., [12,␣
↪56, 6, 8]]
```

Nous définissons d'abord une fonction distance_euclidienne(V, W) qui prend deux vecteurs (listes) de même dimension V et W et qui retourne la distance euclidienne entre eux.

[13]:
```python
# La distance euclidienne est la racine carrée de la somme des carrés des écarts

def distance_euclidienne(V, W):
    s = 0
    for i in range(len(V)):
        s = s + (V[i] - W[i]) ** 2
    return s ** 0.5
```

Aprés nous définissions une fonction k_alea(data, k) qui choisit aléatoirement k élément de la liste des données, il s'agit de l'étape 1 de l'algorithme

[14]:
```python
# Une fonction qui choisit k centroïdes aleatoirement

def k_alea(data, k):
    # Importer numpy.random pour utiliser randint
    import numpy.random as nr

    # La liste des indices des centroïdes dans la liste data
    indices = []
    while len(indices) < k:
        # Choisir un indice aléatoire
        i = nr.randint(0, len(data))
        # Tester si cet indice n'est pas déjà dans la liste indices
        if not i in indices:
            # Ajouter l'indice dans la liste
            indices.append(i)
    # Retourner la liste des centroïdes
    return [data[i] for i in indices]
```

Puis, nous écrivons une fonction qui affecte une donnée (élément de la liste data) à un cluster (centroïde). La fonction prend la liste des centroïdes et affecte la donnée au centroïde convenable (le centroïde le plus proche à la donnée). On utulisera cette fonction pour la partie 2 de l'algorithme.

[15]:
```python
# Une fonction qui permet l'affectation d'une donnée au groupe (cluster)␣
 ↪convenable

def affectation(donnee, clusters):
    # La liste des distances
    D = []

    # Parcours des centroïdes
    for g in clusters:
        # Ajouter la distance entre le centroïde et donnee à la liste D
        D.append(distance_euclidienne(donnee, g))

    # Chercher la distance minimale
    dmin = min(D)
```

```
    # Chercher l'indice du centroïde correspondant à cette distance
    i_cluster = D.index(dmin)

    # Retourner cet indice
    return i_cluster
```

Aprés avoir affecter les données au clusters convenables, nous devons recalculer le nouveau centroïde de chaque cluster. Pour obtenir un centroïde d'un cluster nous calculons le vecteur moyen de tout les vecteurs appartenant à cet cluster puis nous cherchons la donnée la plus proche au vecteur moyen.

Attention : Le vecteur moyen calculée peut être non existant parmi nos données, c'est pour cette raison que nous devons chercher la donnée la plus proche au vecteur moyen.

Nous définissons une fonction qui prend en paramètre les données appartenant à un groupe (cluster) et qui retourne leur vecteur moyen (centre théorique).

[16]:
```
# Une fonction qui retourne le centre theorique d'un cluster

def centre_theorique(data_cluster):
    # Initialiser la liste du vecteur moyen par des 0
    moyen = [0] * len(data_cluster[0])

    # moyen[i] sera égal à la moyenne des data_cluster[i]
    for i in range(len(moyenne)):
        # On calcul moyen[i]
        for x in data_cluster:
            moyen[i] = moyen[i] + x[i]
        moyen[i] = moyen[i] / len(data_cluster)

    # Retourner moyen
    return moyen
```

Nous définissons maintenant la fonction centre_reel(data_cluster) qui retournera le centroïde des données de la liste data_cluster. Il s'agit de la donnée la plus proche du vecteur moyen. C'est l'étape 3 de notre algorithme.

[17]:
```
# Une fonction qui retourne le centre reel d'un cluster

def centre_reel(data_cluster):
    # Chercher le centre théorique (Vecteur Moyen)
    centre_t = centre_theorique(data_cluster)

    # La liste qui contiendra les distances entre le centre théorique et les␣
 ↪données du cluster
    D = []

    # Parcourir les données du cluster
    for x in data_cluster:
```

```
        # Calcul de la distance entre la donnée x et le centre théorique
        # Ajouter la distance dans la liste D
        D.append(distance_euclidienne(centre_t, x))

    # Chercher la distance minimale
    dmin = min(D)

    # Cherche l'indice de la donnée qui fait cette distance minimale
    i_centre_r = D.index(dmin)

    # Retourner le centroïde du cluster
    return data_cluster[i_centre_r]
```

Nous regroupons toutes ces fonctions pour définir la fonction k_means qui prend en paramètre la liste des données data, le nombre k et le nombre d'itérations n.

La fonction retournera deux listes : - clusters : qui contient les centroïdes de chaque cluster ; - attributions : qui contient l'affectation de chaque données (chaque donnée est affecté à un cluster)

[18]:
```
# La fonction k-means á n itérations

def k_means(data, k, n):

    # Choisir aléatoirement k centroïdes
    clusters = k_alea(data, k)

    # Faire n fois
    for i in range(n):
        # La liste qui contiendra les tuples (donnee, indice du centroide␣
 ↪correspondant)
        attributions = []

        # Parcourir nos données
        for x in data:
            # Ajouter le tuple d'affectation dans la liste attributions
            attributions.append((x, affectation(x, clusters)))

        # Parcourir les clusters
        for j in range(len(clusters)):
            # Liste qui contiendra la liste des données d'un cluster
            data_cluster = []

            # Remplir la liste data_cluster
            for t in attributions:
                if t[1] == j:
                    data_cluster.append(t[0])
            # Actualiser le centroïde du cluster d'indice j
            clusters[j] = centre_reel(data_cluster)
```

```
    # Retourner les deux listes clusters et attributions
    return clusters, attributions
```

L'objectif principal de l'algorithme K-means est de minimiser la variance intra-cluster, c'est-à-dire la distance entre les points d'un même cluster et son centroïde, tout en maximisant la distance entre les différents clusters. Cela permet de regrouper les points similaires ensemble tout en les séparant des autres groupes de points.

Une manière d'évaluer un modèle k_means est de calculer la somme des distances carrées entre les données et leurs centroïdes, plus cette somme est petite plus le modèle est précis.

Nous définissons une fonction qui retourne une liste de distances carrées entre les données d'un cluster et le centroïde de ce cluster

[19]:
```
def evaluation_cluster(data_cluster, centre):
    return [distance_euclidienne(x, centre) ** 2 for x in data_cluster]
```

Nous écrivons maintenant une fonction qui prend en paramètre les deux listes retournées par la fonction k_means et qui retourne la somme des distance carrées entre chaque donnée et son centroïde.

[20]:
```
def evaluation_modele(clusters, attributions):
    # Liste qui contiendra les distances
    D = []

    # On parcours les clusters
    for i in range(len(clusters)):
        # Extraire les données du cluster
        data_cluster = [x[0] for x in attributions if x[1] == i]

        # Calculer et ajouter les distances carrés dans la liste D
        D.extend(evaluation_cluster(data_cluster, clusters[i]))

    # Retourner la somme des distances carrées
    return sum(D)
```