III

Algorithme des k plus proches voisins (ou *k-NN algorithm* pour *k-Nearest Neighbors algorithm*) : utilisation en apprentissage automatique supervisé

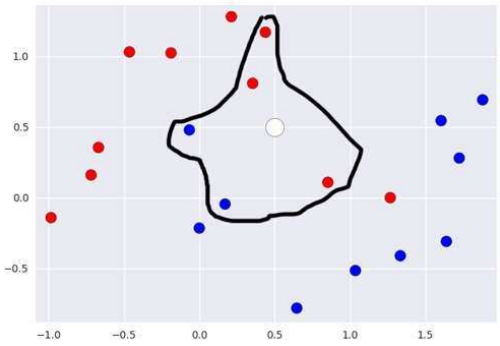


FIGURE III.1 – Peut-on prédire à quelle famille appartient cette nouvelle entrée (point blanc) : les rouges ou les bleus ? (Source : info.openclassrooms.com)

C'est à l'un de ses pionniers, Arthur Samuel (1901-1990), que l'on doit l'invention du terme « apprentissage automatique » grâce à ses recherches sur le jeu de dames. Robert Nealey, le maître autoproclamé du jeu de dames, a joué à ce jeu sur un ordinateur IBM 7094 en 1962, et il a perdu contre l'ordinateur. Comparé à ce qui peut être fait aujourd'hui, cet exploit semble presque futile, mais il est considéré comme une étape majeure dans le domaine du développement de l'intelligence artificielle. (Source : IBM*)*

PLAN DU CHAPITRE

I Position du problème et idée de l'algorithme

I.1 Vocabulaire élémentaire

Le Machine-Learning supervisé ou *Apprentissage Automatique supervisé* est une branche de l'intelligence artificielle qui permet de donner aux machines la capacité d'apprendre, à partir d'un **set de données**, à prédire l'appartenance d'une entrée à une **classe** ou **famille**, c'est à dire **l'étiqueter**, ou bien lui attribuer **la "meilleure" valeur lui correspondant** dans le cadre d'un modèle.

La problématique de l'apprentissage supervisé s'articule toujours en deux phases :

- la première, dite d'**apprentissage** ou d'**entrainement**, consiste à implémenter des données :
  - POUR UN PROBLÈME DIT DE `classification supervisée` :
    les données sont toutes **étiquetées**, i.e. chaque donnée possède une **étiquette** signalant à quelle classe/famille elle appartient.

    L'étiquette est donc **une information ou une valeur discrète** ; on parlera alors « d'étiquette de classe » pour cette donnée. On recense deux types de classification :
    - Classification dite **binaire** lorsque le problème ne comporte exactement que 2 classes (par exemple les réponses de type **Chien ou chat**, **vrai/faux**, **courriel légitime/indésirable** etc...)
    - classification dite **multiclasse** lorsque le problème comporte plus de deux classes (par exemple la détermination de la langue d'écriture d'un texte à partir d'une liste prédéfinie de n langues avec $n > 2$).

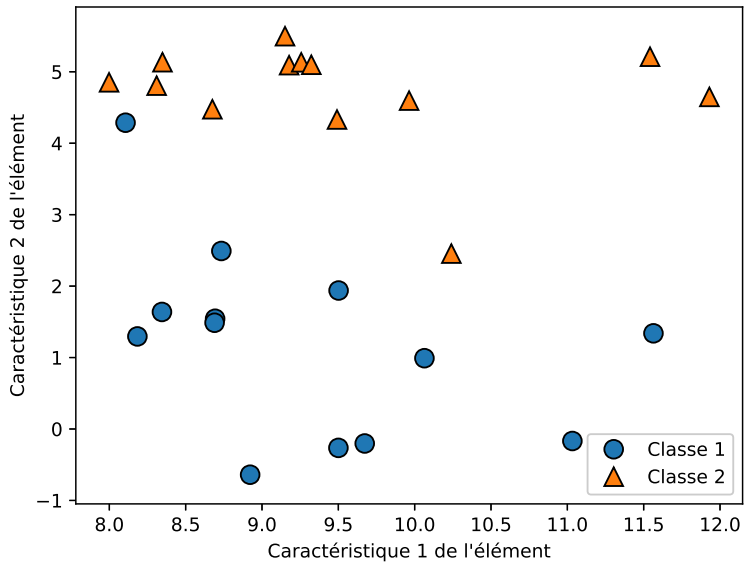


FIGURE III.2 – Exemple de classification binaire : classe 1 et classe 2 (les entrées sont arbitrairement à caractéristiques)
Cet ensemble de données est appelé **ensemble d'apprentissage**.

- POUR UN PROBLÈME DIT DE `régression` :
  les données se présentent sous la forme d'un **ensemble d'entrées-sorties**, ou **ensemble caractéristique-cible**, i.e. chaque entrée est attachée à une valeur de sortie.

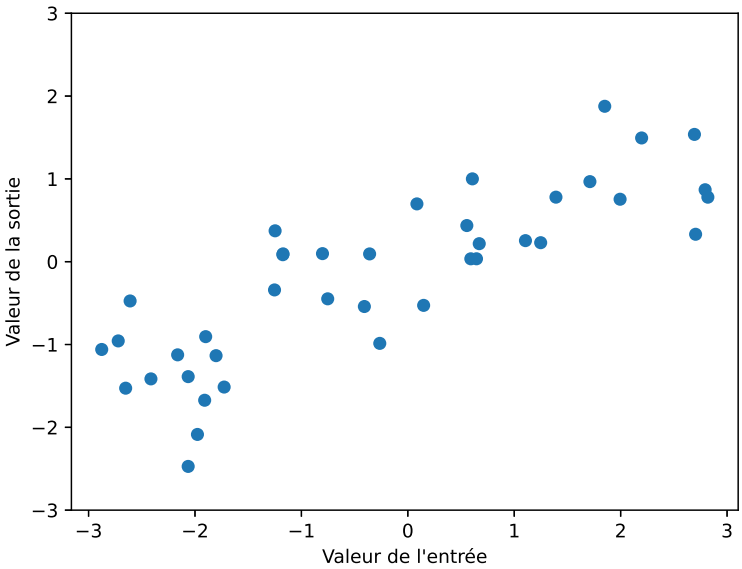


FIGURE III.3 – Exemple de régression : tracé d'un jeu de données, l'axe x donnant la valeur de la caractéristique et l'axe y la cible de la régression

- la seconde met en œuvre l'ensemble d'apprentissage, **à l'aide d'un algorithme** pour prédire l'appartenance de nouvelles entrées à l'une des classes/familles pour la classification (et donc leur attribuer une étiquette) ou bien une valeur de sortie dans le cas de la régression.

I.2 Première idée des k plus proches voisins

a - Principe

La prévision de l'appartenance d'une nouvelle entrée à une classe dans le cas d'une classification (ou la position de cette entrée dans le cas d'une régression) peut se faire à l'aide de l'algorithme des k plus proches voisins qui est le plus simple en apprentissage automatique.

Sa mise en œvre passe par les étapes suivantes :

- On enregistre l'ensemble d'apprentissage (jeu de données). Dans la mesure où ces données sont déjà toutes étiquetées en apprentissage supervisé, l'entrainement est simplement le fait de procéder à cette implémentation.
- Pour une nouvelle entrée dont on connait *les caractéristiques propres*, l'algorithme recherche et trouve alors les k éléments du jeu de données qui s'en rapprochent le plus : **les k plus proches voisins**. Cela sous entend de définir une notion de distance.

– L'algorithme détermine alors, dans ces k plus proches voisins, l'étiquette ou classe la plus fréquente, et décrètera **que c'est celle de la nouvelle entrée**.

Par exemple, sur le jeu de données de classification binaire ci-dessus, on peut faire une tentative de prédiction sur 3 nouvelles entrées avec $k=1$ (très maigre ! ! !) et $k=3$ (mieux) :

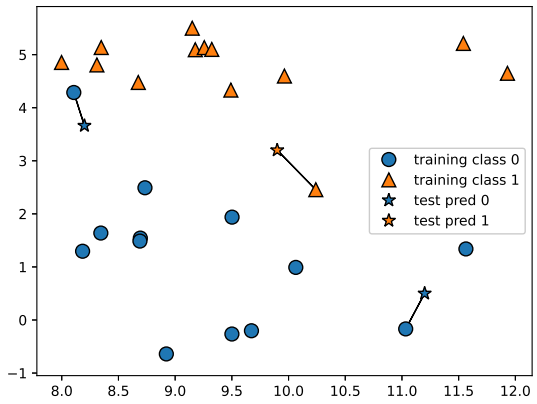


FIGURE III.4 – Prédiction pour 3 entrées avec $k=1$ ⇒ risqué !

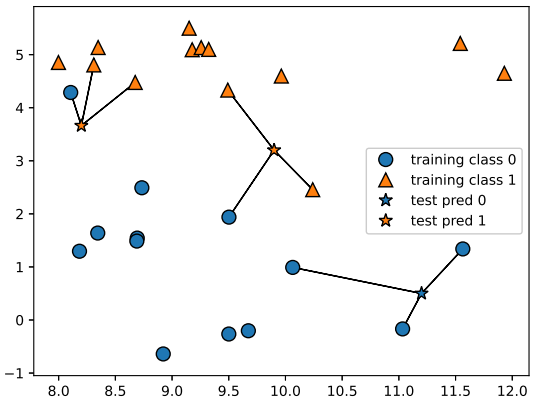


FIGURE III.5 – Prédiction pour 3 entrées avec $k=3$ ⇒ mieux !

Remarque I-1: CHOIX DE **k** ?

L'exemple ci-dessus montre que le choix de k, nombre de plus proches voisins choisis, influence la performance du modèle (observer l'étiquette déduite pour le point d'entrée "test" en haut à gauche suivant le choix de k !) On remarque tout de suite que k n'est pas un paramètre que le modèle va pouvoir déterminer à partir de l'ensemble d'apprentissage ; on le nomme *hyperparamètre* (par opposition aux paramètres classiques). Même si nous ne pouvons pas connaître à l'avance la valeur renvoyant le meilleur résultat, il est en revanche toujours possible de déterminer celle-ci en mesurant les performances du modèle suivant k, ce que nous ferons !

b - Les limites

Pour les jeux de données comportant 2 caractéristiques, comme c'est le cas ici, il est possible d'illustrer les prédictions de classe pour tous les points du plan *(caractéristique 1, caractéristique 2)* en donnant à chacun une certaine teinte en fonction de la classe que l'algorithme lui affecterait ; pour l'exemple précédent de classification binaire cela donne :

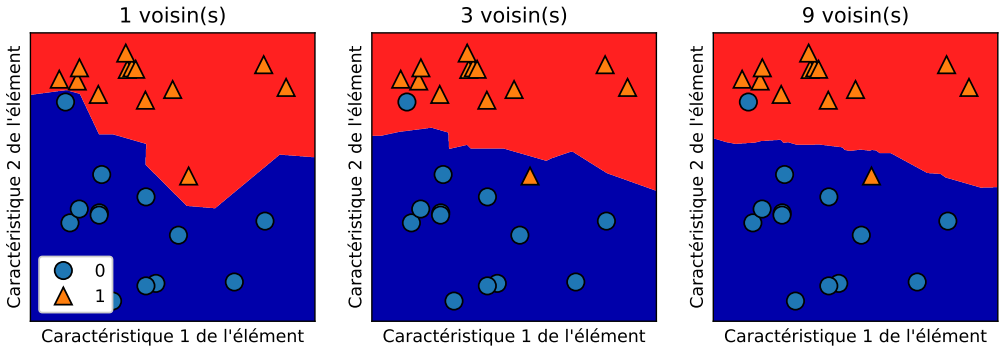


FIGURE III.6 – Prédiction pour l'ensemble des points du plan *(caractéristique 1, caractéristique 2)*

On peut constater que plus le nombre de voisins est élevé, et plus la frontière entre les deux classes est "lissée" facilitant l'écriture d'un modèle (on pourrait facilement tracer une droite de séparation entre les deux !).

En revanche, **une fausse bonne idée** serait de choisir une valeur trop élevé pour k ; prenons le cas extrême où le nombre de voisins k est choisi égal au nombre de données du jeu d'apprentissage, alors chaque point de test aurait exactement les mêmes voisins (ie tout le jeu d'apprentissage) et toutes les prédictions seraient identiques en donnant **la classe majoritaire dans le jeu de données**, ce qui bien évidemment est faux ! ! !

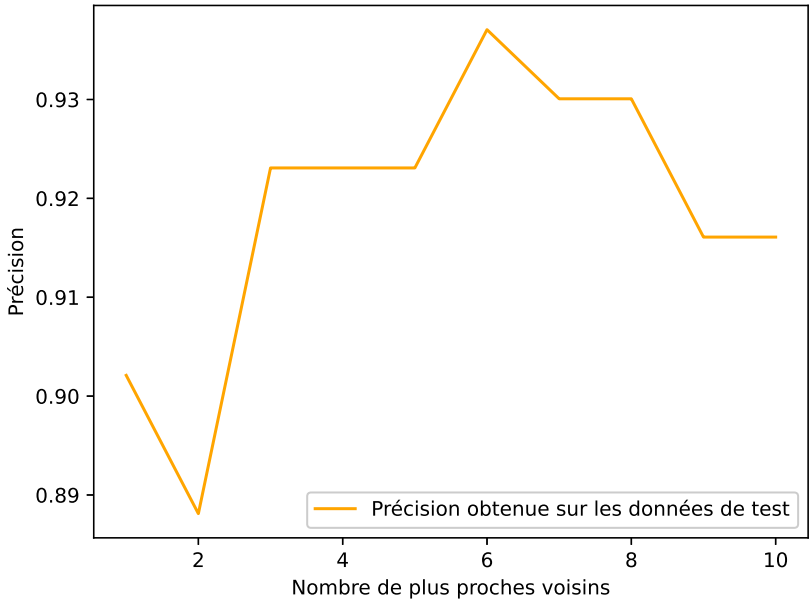


FIGURE III.7 – Exactitude du test en fonction de k

CONCLUSION : il y aura donc bien une valeur optimale à choisir pour k, réalisant le meilleur compromis.

I.3 Que peut-on faire avec l'apprentissage automatique supervisé ?

Les exemples concrets d'usage de l'apprentissage supervisé sont innombrables ; on peut citer un ensemble d'apprentissage constitué d'images de chiens et de chats, chacune d'elle étant attachée à des caractéristiques

référencées (taille des oreilles, couleurs des yeux, dimensions et proportions de la tête etc...) et également associée, suivant le cas, à l'étiquette *chien* ou *chat*.

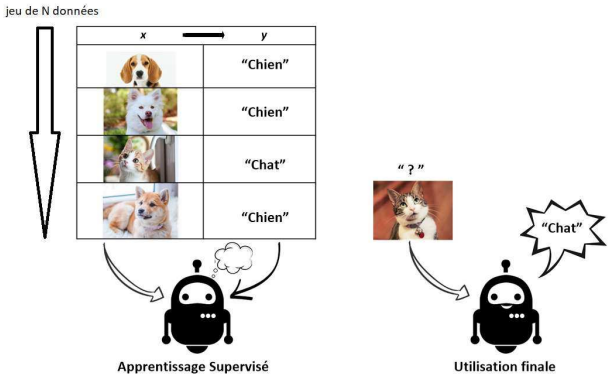


FIGURE III.8 – Apprentissage automatique supervisé appliqué à la reconnaissance d'animaux

Un autre exemple classique, que nous mettrons en œuvre un peu plus bas, est la reconnaissance des chiffres, très utilisée dans le tri postal pour déterminer les codes postaux apposés sur les enveloppes :

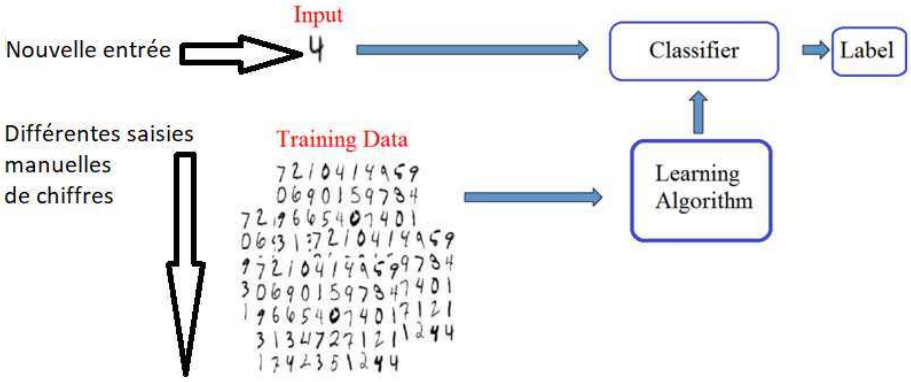


FIGURE III.9 – Apprentissage automatique supervisé appliqué à la reconnaissance des chiffres (pour le tri des codes postaux par exemple)

II Formalisation du problème et mise en œuvre

Un algorithme d'apprentissage automatique supervisé repose donc sur un jeu de N données d'entrée $(x_i)_{1\leq i\leq N}$ ou *dataset* (chaque x_i constitue les caractéristiques de l'élément i), **étiquetées** $(y_i)_{1\leq i\leq N}$, c'est à dire que chaque x_i du jeu est associée à une étiquette y_i d'identification ou *label*, ou encore **classe**.

L'algorithme devra, à partir du jeu $(\mathbf{x_i}, y_i)_{1\leq i\leq N}$ et des caractéristiques $x_{j\notin[1,N]}$ d'un nouveau candidat, renvoyer une "estimation" de l'étiquette y_j initialement inconnue de ce dernier.

II.1 Partitions et classes

Appelons E l'ensemble des éléments étiquetés (set de données).

Définition II-1: PARTITION

Une partition de E est une famille $(A_i)_i$ de sous-ensemble de E non vides constituée d'éléments telle que :

$$\begin{cases} \bigcup_i A_i = E \\ A_i \cap A_j = \emptyset \ \forall i \neq j \end{cases}$$

Définition II-2: CLASSE

Une classe est une fraction d'une partition de E, formée par des données homogènes, c'est à dire présentant des "ressemblances" à la faveur d'un critère de similarité ou d'une distance que l'on devra définir.

Classer ou étiqueter revient donc à associer à chaque élément une **étiquette** qui permet de le placer dans une classe, les éléments de cette classe possédant tous la même étiquette. Encore une fois, en apprentissage supervisé, tous les éléments du jeu de données sont déjà étiquetés.

Dans le contexte de la classification, un sous-ensemble A_i contiendra exactement les éléments d'une des classes, et la réunion de tous les sous-ensemble A_i sera l'ensemble E.

Exercice de cours: **(II.1) - n° 1.** *On considère P une liste de liste(s) Pk d'éléments et E une liste E d'éléments.*
On suppose que les éléments de E et des sous-listes Pk de P peuvent être des clés dans un dictionnaire (c'est à dire hashables, cf chapitre II).
Bâtir une fonction Python d'en tête `Est_partition(P:list(list),E:list) → boolean` *qui renvoie* `True` *si et seulement si P est une partition de E, False sinon.*
On exploitera le schéma suivant :

1. *Créer un dictionnaire D.*
2. *Pour chaque liste P_k de P :*
   - *Afficher un message d'erreur si P_k est vide et conclure*
   - *Sinon, pour tout élément $X \in P_k$:*
     - *si $X \notin E$ afficher un message d'erreur et conclure.*
     - *si $X \notin D$ alors faire $D[X] = k$....*
     - *....sinon si $X \in D$ alors envisager deux possibilités et renvoyer pour chacune le message d'erreur correspondant et conclure.*
   - *Vérifier enfin que $\bigcup_k P_k = E$ et conclure.*

   Calculer la complexité de cette fonction.

RÉPONSE :

II.2 Evaluation de la similarité/distances entre données : distance Euclidienne

La prévision de l'appartenance d'un élément inconnu à une classe sous-entend que l'on puisse mesurer s'il partage des "traits" communs avec d'autres éléments connus de cette classe, c'est à dire présentent des **ressemblances**, ou bien une certaine **proximité**.

Suivant la nature des données à classer, on conçoit facilement que cette ressemblance puisse être exprimée très différemment. Par exemple, si chacune des n informations de l'élément à classer est quantitative, donc mesurable sur une échelle comportant une norme et une origine (c'est par exemple le cas dans les problèmes de classement d'objets géométriques), on calculera simplement **la distance euclidienne** entre le candidat à évaluer et les autres données du set de données dans cet espace à n dimensions ; en revanche si ces informations sont de nature qualitatives ("petit", "moyen", "grand" par exemple), il devient nécessaire de définir ce que l'on appelle des "indices de similarité" (comme par exemple **l'indice de Jaccard**) pour juger de la proximité du candidat avec les données du set ; le programme d'ITC exclut ce dernier cas.

Définition II-3: DISTANCE EUCLIDIENNE

On considère toujours l'ensemble E consitué des éléments du set de données et ceux à tester **défini sur un espace normé**. Pour trois éléments de E comportant chacun n caractéristiques mesurables et de coordonnées respectives $X(x_1, x_2, \ldots, x_n)$, $Y(y_1, y_2, \ldots, y_n)$, et $Z(z_1, z_2, \ldots, z_n)$, la distance Euclidienne D_e dans un espace de dimension n est une application de E sur $E^2 = E \times E$ respectant les propriétés :

- de séparation : $D_e(X, X) = 0$ et $D_e(X, Y) = 0 \Rightarrow X = Y$
- de symétrie : $D_e(X, Y) = D_e(Y, X)$
- d'inégalité triangulaire : $D(X, Z) \leq D(X, Y) + D(Y, Z)$

La distance Euclidienne se calcule par la relation :

$$D_e(X, Y) = \sqrt{\sum_{i=1} n\left((x_i - y_i)^2\right)}$$

L'ensemble E muni de ces définitions est un espace dit *métrique*.

II.3 L'algorithme des k plus proches voisins (K-NN algorithm)

a - Implémentation

On appellera $\mathscr{X}$ le set de données. On supposera que chaque élément du set de données correspond à la ligne d'une matrice qui sera notée X. Ainsi, chaque ligne $X[i,:]$ comporte toutes les caractéristiques $x_{i,j\in[1,n]}$ de l'élément dont l'étiquette/classe d'appartenance est connue et implémentée dans un dictionnaire $DictX$ avec i comme clé ; ainsi $DictX[i]$ est la classe de l'élément enregistré à la ligne $X[i,:]$.

Elément 1 :	CARACTÉRISTIQUE 1	CARACTÉRISTIQUE 2	...	CARACTÉRISTIQUE n
Elément 2 :	...	...	...	...
...				
Elément i :	CARACTÉRISTIQUE 1	CARACTÉRISTIQUE 2	...	CARACTÉRISTIQUE n
...				
Elément N :	...	...	...	...

$$\Longrightarrow X = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & \dots & x_{2,n} \\ \dots & & & \\ x_{i,1} & x_{i,2} & \dots & x_{i,n} \\ \dots & & & \\ x_{N,1} & x_{N,2} & \dots & x_{N,n} \end{pmatrix}$$

$$DictX = \Big\{\text{Elément 1 :Etiqu.(élément 1)},\dots,\text{Elément i :Etiqu.(élément i)},\dots,\text{Elément N :Etiqu.(élément N)}\Big\}$$

On cherche à connaître l'étiquette (évidemment **inconnue**) d'un élément x dont les caractéristiques sont **connues**, à partir d'un sous-ensemble $\mathscr{A}$ du set de données qui sera notre `set d'apprentissage` ; ce sous-ensemble sera représenté par une matrice A qu'il faudra construire. On introduira également le dictionnaire $DictA$ selon le même schéma que $DictX$, à savoir que $DictA[i]$ est la classe de l'élément enregistré à la ligne $A[i,:]$.

Remarque II-1: PROPORTIONS SET DE DONNÉES/SET D'APPRENTISSAGE

On découpe généralement le set de données en deux sous-ensemble : l'un comportant 75% des données et qui servira de **set d'apprentissage**, et l'autre, comportant les 25% restants, dont tout ou partie constituera le **set de test**.

On va construire le schéma de fonctionnement en deux étapes :

- Une première fonction `kppv(A:array,DictA:dict,k:int,x:array)⟶ list` (k : nombre choisi de plus proches voisins à considérer) qui bâtira une liste L de dimension N dans laquelle chaque terme est un tuple contenant l'étiquette de la donnée i, et sa distance au candidat x, représenté par **un tableau 1D contenant les n caractéristiques du candidat** ; cette liste est **triée** dans l'ordre croissant des valeurs de distance au candidat x :

  `L=[(0,3.76),(0,2.54),(1,6.845),...]`

  NB : dans cet exemple les informations du set de données sont donc de type `float`.
- Une seconde fonction `Etiqu_maj(L:list,epsilon:float)` qui reçoit la liste `L` et le flottant `epsilon` et qui renverra :
  - l'étiquette du premier tuple si la distance associée est nulle (ou bien $\leq epsilon$)....
  - ...ou l'étiquette la plus fréquente sinon. Si plusieurs étiquettes possèdent la fréquence d'occurrence majoritaire, alors choisira l'une d'elles au hasard.

Exercice de cours: **(II.3) - n° 2.** *Rédiger les fonctions python* `kppv(A:array,D:dict,k:int,x:array)⟶ int` *et* `Etiqu_maj(L:list,epsilon:float)⟶ type(`*étiquette*`)`.

b - Test et caractérisation de performance : la matrice de confusion (ou tableau de contingence)

Dans la pratique, il est toujours nécessaire de tester un algorithme d'apprentissage supervisé afin de mesurer ses performances. On peut résumer ces dernières à l'aide d'un outil appelé **matrice de confusion**. Pour construire cette dernière, on compare les prédictions d'étiquettes que fournit l'algorithme à partir du set d'apprentissage $\mathscr{A}$ aux étiquettes connues pour des éléments du set $\mathscr{X}$ n'appartenant pas à $\mathscr{A}$ (dans le cas contraire, l'analyse de performance serait biaisée avec le risque de tenir compte d'étiquettes à distance nulle !).

On peut en donner un définition formelle :

Définition II-4: MATRICE DE CONFUSION

On considère un ensemble de données X, chaque donnée étant étiquetée par sa classe parmi $\mathscr{C} = \{Et.1, Et.2, \ldots\}$ et deux applications f_1 et f_2 avec $f_i : X \to \mathscr{C}$.
On appelle **matrice de confusion** des applications f_1, f_2 la matrice M de taille $|\mathscr{C}| \times |\mathscr{C}|$ telle que :

$$M_{ij} = |x \in X / \ f_1(x) = Et.i, f_2(x) = Et.j|$$

Plus concrètement, si l'application f_1 renvoie l'étiquette réelle de la donnée $x \in X$, et l'application f_2 l'étiquette prédites par l'algorithme, alors les lignes de la matrice représenteront les **étiquettes réelles**, et les colonnes les **étiquettes prédites** ; ainsi, le nombre M_{ij} consigné dans la cellule ligne i et colonne j est, pour un nombre important et connu d'estimations, le nombre d'éléments appartenant effectivement à la classe i et ayant été estimés comme appartenant à la classe j :

i (réel) \ j (estimé)	Et.1	Et.2	…
Et.1	M_{11}	M_{12}	…
Et.2	M_{21}	M_{22}	…
…	…	…	…

$$\Longrightarrow \quad M = \begin{pmatrix} M_{11} & M_{12} & \dots \\ M_{21} & M_{22} & \dots \\ \dots & \dots & \end{pmatrix}$$

Supposons par exemple que nous collections un ensemble de paramètres biologiques sur des femmes potentiellement enceintes, caractéristiques autres qu'un test de grossesse évidemment, l'état de chaque femme `enceinte / pas enceinte` étant connu et constituant l'étiquette. Ces données forment alors un `dataset` d'entrainement pour un algorithme d'apprentissage supervisé.

Comme deux étiquettes seules sont envisageables ici : `enceinte / pas enceinte`, il s'agit d'une classification **binaire**. La matrice de confusion sera donc une matrice 2×2, traduisant quatre situations concrètes :

- VRAI NEGATIF (TN) : l'algorithme prédit que la patiente n'est pas enceinte ce qui est la réalité : elle n'est pas enceinte.
- VRAI POSITIF (TP) : l'algorithme prédit que la patiente est enceinte ce qui est la réalité : elle est enceinte
- FAUX POSITIF (FP) : l'algorithme prédit que la patiente est enceinte ce qui n'est pas la réalité : elle n'est pas enceinte.
- FAUX NEGATIF (FN) : l'algorithme prédit que la patiente n'est pas enceinte ce qui n'est pas la réalité : elle est enceinte

La matrice de confusion prend donc l'allure suivante :

i (diagn.réel) \ j (diagn.estimé)	Négatif	Positif
Négatif	nb Vrai Négatif	nb Faux Positif
Positif	nb Faux Négatif	nb Vrai Positif

Ainsi, la matrice de confusion :

- fournit un résumé des résultats de prédiction pour un problème particulier de classification.
- permet de connaître les différentes erreurs commises par l'algorithme de classification, c'est à dire dans le détail quelle(s) classe(s) est (/ sont) la (les) mieux ou la (les) plus mal prédite(s).
- mieux ! : permet de connaître **les différents types d'erreurs commises par l'algorithme !**

On va chercher à construire une fonction `MatConf` renvoyant la matrice de confusion à partir : du `dataset` X représenté par une matrice (N, n), du dictionnaire $DictX$, d'une liste $SplA$ contenant les indices de ligne des éléments prélevés dans $\mathscr{X}$ pour construire le sous-ensemble d'apprentissage $\mathscr{A}$, de la matrice A, du dictionnaire $DictA$, et du nombre de plus proches voisins choisis k.

Exercice de cours: **(II.3) - n° 3.** *Construire la fonction :*
`MatConf(X:array,DictX:dict,SplA:list,A:array,DictA:dict,k:int)` ⟶ `array`

RÉPONSE :

Remarque II-2: PERFORMANCE "APPARENTE"

Compte tenu du principe de construction de la matrice de confusion, on comprend qu'un algorithme **"idéal"** doit normalement renvoyer **une matrice diagonale** dont la trace correspond aux nombres d'éléments traités pour le test. Il sera donc facile, en observant les matrices de confusion de différents tests, de comparer leurs performances respectives : plus la valeur des termes **extra-diagonaux** est importante et moins bon est l'algorithme.

II.4 Exemple complet : la reconnaissance des chiffres

a - Mise en oeuvre du dataset MNIST784

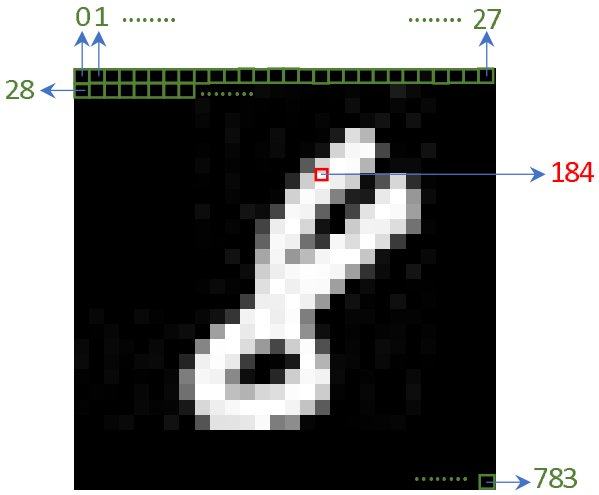
On va chercher ici à réaliser la reconnaissance d'un chiffre manuscrit à partir d'un *dataset* bien connu : le dataset *MNIST784* utilisé depuis longtemps dans le tri postal automatique. Il est constitué d'un ensemble de 70000 images de chiffres manuscrits (0..9) ,numérisés au format 28x28 pixels (soit 784 pixels par image) en noir (valeur 0) et blanc (valeur 255) et chacun attaché à l'étiquette correspondant au chiffre représenté ; par conséquent, **c'est un problème à 10 classes**.

Le *dataset* est implémenté sous forme d'un "Bunch de données", ensemble qui s'apparente à un simple dictionnaire `<clé>:<valeur>` , tout en supportant des opérations habituellement disponibles pour d'autres types (concaténation notamment).

Les **3 premières clés de ce dictionnaire** (les seules utiles pour nous) et les types de valeurs correspondantes sont :

- `data` : `Dataframe` de 70000 lignes et 784 colonnes, chaque ligne contenant le codage des 784 pixels d'un chiffre.
- `target` : `Series` de 70000 données correspondant aux étiquettes (chiffre 0..9) des 70000 chiffres.
- `frame` : `Dataframe` de 70000 lignes et 785 colonnes, chaque ligne comporte le codage des pixels du chiffre soit 784 colonnes puis une dernière colonne contenant donc un chiffre 0..9).

Compte tenu du codage des images dans le set (70000 images de 784 pixels chacune), la matrice A comporterait 70000 lignes dans l'hypothèse où l'on consignerait l'ensemble du dataset, ce qui n'est en rien une obligation, et 784 colonnes ; chaque élément de A est une valeur codée sur 8 bits ($\in [0,255]$) du pixel correspondant de l'image. On donne à titre d'exemple un aperçu du codage du nombre figurant sur la ligne 10001 du Dataframe `data` (c'est un 8 ! ! !)) :



0	1	...	27	28	...	184	...	783
(0.0	0.0	...	0.0	0.0	...	211.0	...	0.0)

On peut par exemple implémenter la matrice de données `X` et le dictionnaire `Dict X` à partir de seulement 800 données prises au hasard dans le `Dataframe data` (et la `Series target` pour leurs étiquettes respectives) (c'est largement suffisant pour obtenir d'excellents résultats). Ce nouveau set constituera **notre** `dataset` :

```
1 from sklearn.datasets import fetch_openml # importation de la base de données contenant mnist_784
2 import numpy as np # importation du module numpy
3 import random as rd
4
5 setdonnees=fetch_openml('mnist_784',version=1) # affectation du dataset mnist_784
6 nbdonneesX=800
7 SplX=rd.sample([i for i in range(70000)],nbdonneesX) # tirage aléatoire sans remise
8 X=np.zeros((nbdonneesX,784), dtype=int) # création de la matrice X remplie de zéros
9 DictX={} # Création du dictionnaire DictX
10 for i,j in enumerate(SplX):
11   X[i]=setdonnees.data.iloc[j].to_numpy()
12   DictX[i]=setdonnees.target[j]
```

On peut ensuite implémenter un set d'apprentissage A selon le même principe, c'est à dire en sélectionnant au hasard 75% des données du set $\mathscr{X}$ de 800 données, soit 600 données au total. On veille là-encore à ne pas créer de doublons qui biaiserait l'évaluation de performance de l'algorithme.

```
13 nbdonneesA=3*nbdonneesX//4 #75% des données du set X
14 SplA=rd.sample([i for i in range(nbdonneesX)],nbdonneesA) # tirage des données d'apprentissage
15 A=np.zeros((nbdonneesA,784), dtype=int)  # création de la matrice A remplie de zéros
16 DictA={} # création du dictionnaire DictA
17 for i,j in enumerate(SplA):
18   A[i]=X[j]
19   DictA[i]=DictX[j]
```

On va enfin faire un premier test sur l'algorithme, par exemple sur la donnée figurant sur la ligne 10001 du Dataframe `data` déjà évoqué plus haut. La ligne de code suivante implémente le candidat x inconnu en extrayant la donnée n°10001, et la convertit en un tableau comportant une ligne et 784 colonnes :

```
20 x=setdonnees.data.iloc[10001].to_numpy().reshape(1,784)
```

et les suivantes affichent l'étiquette de la donnée n°10001 et lance l'algorithme sur la donnée x "inconnue" :

```
print(setdonnees.target[10001])
k=5 # valeur faible du nombre de ppv, histoire que l'algorithme ne s'éternise pas
print(kppv(A,D,k,x))
```

avec comme sortie :

```
8
8
```

ce qui est bien le résultat espéré !

b - Matrice de confusion et mesure de performance - optimisation

On va caractériser la performance de cet algorithme en recherchant sa matrice de confusion M.

Dans le set `MNIST784`, les *étiquettes* correspondent aux 10 chiffres de 0 à 9. M est donc ici une matrice carré, de dimension (10×10).

ATTENTION : les *étiquettes* sont ici des chaînes de caractères qu'il faudra convertir en entier si l'on souhaite les utiliser comme indices de M. On travaillera pour la phase de test avec un extrait du set de données dont la taille correspond, comme de coutume, à 25% de celui-ci.

```
def MatConf_bis(X,DictX,SplA,A,DictA,k):
    N,n=X.shape
    nb_etiqu=len(set(DictX.values()))
    M=np.zeros((nb_etiqu,nb_etiqu),dtype=int)
    compt=0
    liste_i=[]
    while compt<N//4: # on realise les tests sur 25% du nb total de données
        i=rd.randint(0,N-1)
        if i not in SplA and i not in liste_i: #tiré pas dans le set d'apprentissage et non déjà tiré
            x=X[i,:].reshape(1,784) #extraction nouveau candidat, dimensionnement au bon format
            etiqu_vraie=DictX[i] # son étiquette vraie
            etiqu_estim= kppv(A,DictA,k,x) # son etiquette estimée
            M[int(etiqu_vraie),int(etiqu_estim)]+=1 #incrém. 1 unité de M[i,j] (NB: indices conv. en ent
            liste_i.append(i)
            compt+=1
    return M
```

En exécutant avec $N = 800$ et $k = 5$:

```
>>> print(MatConf_bis(X,DictX,SplA,A,DictA,k))
```

on obtient la matrice de confusion suivante :

```
[[18  1  0  0  0  5  1  1  0  0]
 [ 0 17  0  0  0  0  0  0  0  0]
 [ 1  3  7  1  0  2  1  2  1  0]
 [ 0  2  0 20  0  3  0  0  4  0]
 [ 0  1  0  0 12  0  0  0  0  6]
 [ 0  2  0  3  1  8  0  0  2  0]
 [ 0  0  0  0  1  0 19  0  0  0]
 [ 0  1  0  0  0  0  0 14  0  0]
 [ 1  0  0  1  1  0  1  0  8  1]
 [ 1  1  0  0  5  0  0  1  0 19]]
```

Une fois la matrice de confusion connue (on peut aussi faire une moyenne de matrice de confusion), on peut très facilement établir des statistiques de performance de l'algorithme. Par exemple, en lisant la première ligne de la matrice qui correspond à l'étiquette "vraie" "0", on constate que l'algorithme conclut 18 fois correctement et commet 8 fois une erreur ; ainsi la probabilité de succès de détection d'un "0" est, avec le choix $k = 5$, :

$$\mathscr{P}(\text{"0"}) = \frac{18}{18+1+5+1+1} = \frac{18}{26} \simeq 69,2\%$$

ce qui n'est clairement pas un taux de réussite satisfaisant.
En relançant l'algorithme avec $k = 10$, on obtient pour M :

```
[[29  0  1  0  0  0  2  0  2  0]
 [ 0 18  0  1  0  0  0  0  0  0]
 [ 0  8 12  1  0  0  1  0  2  0]
 [ 1  3  2  8  0  1  0  0  3  0]
 [ 0  0  0  0 10  0  1  4  0  3]
 [ 0  0  0  1  0  9  1  1  1  1]
 [ 0  0  0  0  3  0 15  1  2  0]
 [ 0  1  0  1  1  1  0 13  0  4]
 [ 0  2  0  3  1  2  0  1  4  1]
 [ 0  1  0  0  6  0  0  4  0  6]]
```

Le taux de réussite obtenu cette fois pour la détection du "0" est alors :

$$\mathscr{P}(\text{"0"}) = \frac{29}{29+1+2+2} = \frac{29}{34} \simeq 85,3\%$$

Exercice de cours: **(II.4) - n° 4.** *Proposer une fonction python* $Proba\_i(M : array, i : int) \longrightarrow float$ *prenant en argument la matrice de confusion* M *et un chiffre* i*, et renvoyant la valeur de la probabilité de succès de reconnaissance de ce chiffre* i*.*