Algorithmes de classification

- Un algorithme de **classification** consiste à associer à chaque **donnée** une **classe** (ou : étiquette).
  Exemple : à chaque image, on veut associer l'objet représenté par l'image.

- On distingue deux types d'algorithmes de classification :
  - **Supervisé** : on possède des **données d'entraînement** pour lesquelles on connaît les classes. On veut ensuite prédire les classes de nouvelles données (**données de test**).
    Exemple : algorithme des plus proches voisins.
  - **Non supervisé** : pas de données d'entraînement.
    Exemple : algorithme des k-moyennes.

- Les algorithmes d'apprentissage ont besoin d'une notion de distance entre les données. Pour cela, on se ramène à $\mathbb{R}^k$. Ainsi, si une donnée de voiture possède une vitesse maximum de 200 km/h, consomme 10 litres au 100 km et pèse 1500 kg, on peut la représenter par le vecteur $\begin{pmatrix} 200 \\ 10 \\ 1500 \end{pmatrix}$.
  Pour une image : on passe d'une matrice de pixels avec n lignes, p colonnes à un vecteur de taille np.

- On représente classiquement l'ensemble des données (donc de vecteurs de $\mathbb{R}^p$) par une matrice `X` dont les lignes sont les données et les colonnes sont les attributs (coordonnées des vecteurs).

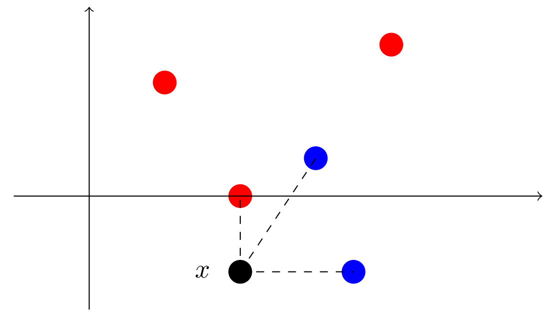
- On peut utiliser la distance euclidienne entre deux données : $d(x, y) = \sqrt{\sum_{i=1}^{p} (x_i - y_i)^2}$

```python
def d(x, y):
    s = 0
    for i in range(len(x)):
        s += (x[i] - y[i]) ** 2
    return s ** 0.5
```

- Pour éviter que les données ne soient trop influencées par les attributs qui ont des valeurs plus grandes, on peut standardiser (ou : normaliser) les données. On soustrait à chaque attribut sa moyenne et on divise par l'écart-type.

Algorithme des k plus proches voisins

- Soit $k \in \mathbb{N}$. L'**algorithme des k plus proches voisins** prédit la classe d'une nouvelle donnée x de la façon suivante:
  1. Calculer les distances de x à toutes les données d'entraînement.
  2. Trouver les k données d'entraînement les plus proches de x (en termes de distance).
  3. Trouver la classe majoritaire $c \in Y$ parmi ces k données les plus proches de x.
  4. Prédire que x est de classe c.



La classe de x est prédite comme étant bleue (ici, $k = 3$ et il y a deux classes : bleu et rouge)

- Exercice : Écrire une fonction `voisins(x, X, k)` renvoyant la liste des k plus proches voisins de `x` dans `X`.

```python
def voisins(x, X, k):
    I = [] # indices des k plus proches voisins dans X
    for i in range(k): # ajout du ième minimum
        jmin = 0
        for j in range(len(X)):
            if d(x, X[j]) < d(x, X[jmin]) and j not in I:
                jmin = j
        I.append(jmin)
    return I
```

- Exercice : Écrire une fonction `maj(L)` renvoyant l'élément le plus fréquent de la liste `L`, en complexité linéaire.

```python
def maj(L):
    C = {} # C[e] = nombre d'occurrences de e dans L
    for e in L:
        if e in C:
            C[e] += 1
        else:
            C[e] = 1
    kmax = L[0]
    for k in C:
        if C[k] > C[kmax]:
            kmax = k
    return kmax
```

  Complexité : $O(n)$, où n est la taille de `L`.

- Exercice : Écrire une fonction `knn(x, X, Y, k)` renvoyant la classe prédite par l'algorithme des k plus proches voisins pour la donnée `x` et les données d'entraînement `X` et `Y`.

```python
def knn(x, X, Y, k):
    """ Prédit la classe de x avec l'algorithme KNN
    x : nouvelle donnée
    X : données d'entraînement
    Y : étiquettes des données d'entraînement
    k : nombre de voisins à considérer
    """
    V = voisins(x, X, k)
    return maj([Y[i] for i in V])
```

- Supposons que l'on possède des données X avec des étiquettes Y et qu'on veuille savoir si KNN est un bon classifieur pour ces données.

Pour cela, on partitionne les données en deux ensembles :

- Ensemble d'entraînement X_{train} (de classes Y_{train}) : données parmi lesquelles on va chercher les k plus proches voisins.
- Ensemble de test X_{test} (de classes Y_{test}) : données utilisées pour évaluer le modèle, en comparant les classes prédites par KNN avec les classes réelles.

- La **précision** d'un modèle est la proportion de données de test bien classées par rapport au nombre total de données.

```python
def precision(k):
    n = len(X_test)
    p = 0
    for i in range(n):
        if predict(X_test[i], k) == Y_test[i]:
            p += 1
    return p/n
```

- La **matrice de confusion** est une matrice carrée dont les lignes et les colonnes sont les classes possibles. La case (i, j) contient le nombre de données de test de classe i qui ont été prédites comme appartenant à la classe j.
  Exemple : Dans la matrice suivante, on voit que 21 données de classe 0 ont été prédites comme appartenant à la classe 0, 2 données de classe 1 ont été prédites comme appartenant à la classe 2...

```
array([[21,  0,  0],
       [ 0, 29,  1],
       [ 0,  2, 23]])
```

- Pour choisir la valeur de k dans l'algorithme des k plus proches voisins, on peut afficher la précision en fonction de k pour choisir la valeur de k qui donne la meilleure précision.

Algorithme des k moyennes

- Le **centre** (ou : **isobarycentre**) d'un ensemble de vecteurs $x_1, \ldots, x_n$ est le vecteur $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$.
  Exercice : Écrire une fonction `centre(X)` renvoyant le centre de la liste de vecteurs `X`.

```python
def centre(X):
    n, p = len(X), len(X[0])
    x = [0] * p
    for i in range(n):
        for j in range(p):
            x[j] += X[i][j]
    for j in range(p):
        x[j] /= n
    return x
```

- La variance (ou : moment d'inertie) $V(X)$ d'un ensemble de vecteur X est définie par

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

La variance mesure la variation par rapport à la moyenne : plus $V(X)$ est petit, plus les vecteurs de X sont proches du barycentre $\overline{X}$.
Objectif : trouver un partitionnement (*clustering*) de X en k sous-ensembles $X_1, \ldots, X_k$ (classes ou *clusters*) minimisant l'inertie I : $I = \sum_{i=1}^{k} V(X_i)$
Dit autrement : on veut associer à chaque donnée x une classe k telle que l'inertie I soit minimum.
Plus l'inertie est petite, plus les données sont proches du centre de leur classe et plus le partitionnement est bon.

Algorithme des k-moyennes

Objectif : partitionner X en classes X_1, ..., X_k.

1. Soit $c_1, ..., c_k$ des vecteurs choisis aléatoirement.
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 ont changé, revenir à l'étape 2.

Attention : dans l'algorithme des k-moyennes, k est le nombre de classes alors que dans l'algorithme des plus proches voisins, k est le nombre de voisins.

```python
def calculer_centres(classes):
    centres = []
    for i in range(len(classes)):
        centres.append(centre(classes[i]))
    return centres

def plus_proche(x, centres):
    imin = 0
    for i in range(len(centres)):
        if d(x, centres[i]) < d(x, centres[imin]):
            imin = i
    return imin

def calculer_classes(X, centres):
    classes = [[] for i in range(len(centres))]
    for x in X:
        classes[plus_proche(x, centres)].append(x)
    return classes

def kmoyennes(X, centres):
    centres2 = None
    while centres != centres2:
        centres2 = centres
        classes = calculer_classes(X, centres2)
        centres = calculer_centres(classes)
    return classes
```