Science des données

Représentation des données

La science des données (data science) a pour objectif d'extraire de l'information à partir de données brutes.

- Les algorithmes de science des données 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 $\begin{pmatrix} 200 \\ 10 \\ 1500 \end{pmatrix}$. Chacune de ces valeurs numérique représente l'**attribut** qui lui correspond.

- 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.

- On travaille la plupart du temps à partir de données brutes (raw data). Ces données sont très rarement parfaites et contiennent des problèmes de validité ou d'uniformité et sont parfois incomplètes, incorrectes, inrrégulières ou inconsistantes. Il faut donc les nettoyer !

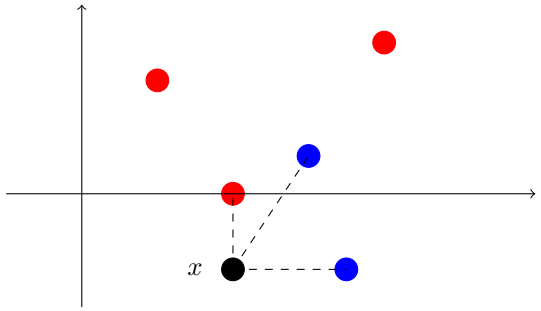
Algorithmes de classification et de régression

- Les algorithmes que nous étudions consistent à associer à chaque **donnée** une **classe** (ou étiquette). Si l'ensemble d'étiquettes est fini, on parle de **classification**, s'il est infini, on parle de **régression**.

- 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.
    Exemple : algorithme des plus proches voisins.
  - **Non supervisé** : pas de données d'entraînement, aucune classe n'est connue à l'avance.
    Exemple : algorithme des k-moyennes.

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 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)

- Fonction `d(x, y)` renvoyant la distance euclidienne entre les vecteurs `x` et `y`.

```
def d(x, y):
    s = 0
    for i in range(len(x)):
        s += (x[i] - y[i]) ** 2
    return s**.5
```

- Fonction `voisins(x, X, k)` renvoyant la liste des k plus proches voisins de `x` dans `X`.

```
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
```

- Fonction `maj(L)` renvoyant l'élément le plus fréquent de la liste `L`, en complexité linéaire.

```
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
```

- 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 leurs étiquettes `Y`.

```
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.

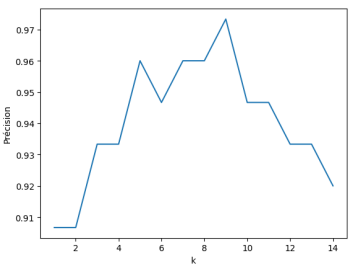
- La **précision** d'un modèle est la proportion de données de test bien classées par rapport au nombre total de données.

```
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 mais 2 données de classe 2 ont été prédites comme appartenant à la classe 1...

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

- La valeur de k, fixée à l'avance, va impacter la décision prise par l'algorithme : k est un **hyperparamètre**. Pour la choisir, 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 $\overline{x} = \dfrac{1}{n}\sum_{i=1}^{n} x_i$.
  Fonction `centre(X)` renvoyant le centre de la liste de vecteurs `X`.

```
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] = x[j]/n
    return x
```

- La variance $V(X)$ d'un ensemble de vecteurs X est définie par :

$$V(X) = \frac{1}{|X|}\sum_{x\in X} d(x, \overline{X})^2$$

  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 :

$$I = \sum_{i=1}^{k} |X_i| V(X_i) = \sum_{i=1}^{k}\sum_{x\in X_i} d(x, \overline{X_i})^2$$

  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 : partitionnner 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.

```
def centre(classe):
    centre = [0]*len(X[0])
    print(classe)
    for i in range(len(classe)):
        for j in range(len(centre)):
            centre[j] += classe[i][j]
    if len(classe) != 0:
        for j in range(len(centre)):
            centre[j] /= len(classe)
return centre

def calculer_centres(classes):
    centres = []
    for c in classes:
        centres.append(centre(c))
    return centres

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

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 kmeans(X, centres):
    classes = calculer_classes(X, centres)
    nouveaux_centres = calculer_centres(classes)
    for i in range(len(centres)):
        if not np.array_equal(nouveaux_centres[i], centres[i]):
            return kmeans(X, nouveaux_centres)
    return classes

def heuristique_centres(X,k):
    centres = [X[rd.randint(0,len(X)-1)]]
    for k in range(k-1):
        max_x=X[0]
        maxi=0
        for x in X:
            dist=0
            for c in centres:
                dist+= d(x,c)
            if dist>maxi:
                maxi=dist
                max_x=x
        centres.append(max_x)
    return centres
```