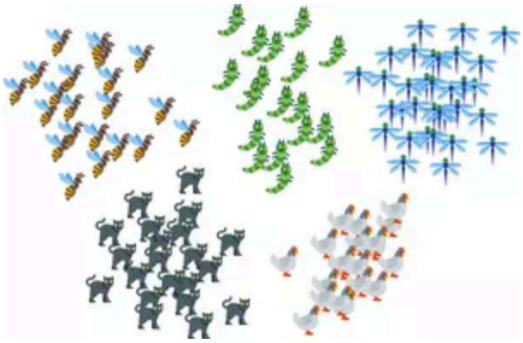
8. Classificateur k-Nearest Neighbor en Python



"Montre-moi qui sont tes amis et je te dirai qui tu es ?"

Le concept du classificateur k-plus proche voisin (k-Nearest Neighbor) peut difficilement être décrit plus simplement. C'est un vieux dicton, que l'on retrouve dans de nombreuses langues et de nombreuses cultures.

Cela signifie que le concept du classificateur `k-plus proche voisin` fait partie de notre vie quotidienne et de nos jugements : Imaginez que vous rencontriez un groupe de personnes, elles sont toutes très jeunes, élégantes et sportives. Ils parlent de leur ami Ben, qui n'est pas avec eux. Alors, comment imaginez-vous Ben ? Bien, vous l'imaginez jeune, élégant et sportif aussi.

Si vous apprenez que Ben vit dans un quartier où les gens votent conservateur et où le revenu moyen est supérieur à 200 000 dollars par an ? Ses deux voisins gagnent même plus de 300 000 dollars par an ? Que pensez-vous de Ben ? Très probablement, vous ne le considérez pas comme un outsider et vous le soupçonnez également d'être conservateur ?

Le principe de la classification par les plus proches voisins consiste à trouver un nombre prédéfini k , d'échantillons d'apprentissage les plus proches en distance d'un nouvel échantillon, qui doit être classé. L'étiquette du nouvel échantillon sera définie à partir de ces voisins. Les classificateurs à k voisins les plus proches ont une constante fixe définie par l'utilisateur pour le nombre de voisins qui doivent être déterminés. Il existe également des algorithmes d'apprentissage des voisins basés sur le rayon, dont le nombre de voisins varie en fonction de la densité locale des points, tous les échantillons étant situés dans un rayon fixe. La distance peut, en général, être une mesure métrique quelconque : la distance euclidienne standard est le choix le plus courant. Les méthodes basées sur les voisins sont connues comme des méthodes d'apprentissage automatique non généralisatrices, car elles se "souviennent" simplement de toutes leurs données d'apprentissage. La classification peut être calculée par un vote majoritaire des plus proches voisins de l'échantillon inconnu.

L'algorithme k-NN est l'un des plus simples de tous les algorithmes d'apprentissage automatique, mais malgré sa simplicité, il s'est avéré très efficace dans un grand nombre de problèmes de classification et de régression, par exemple la reconnaissance de caractères ou l'analyse d'images.

Maintenant, allons un peu plus loin sur le plan mathématique :

Comme expliqué dans le chapitre Préparation des données, nous avons besoin de données d'apprentissage et de test étiquetées. Contrairement aux autres classificateurs, les classificateurs purs de type "nearest-neighbor" n'effectuent aucun apprentissage, mais l'ensemble dit d'apprentissage LS est un composant de base du classificateur. Le classificateur `kNN` travaille directement sur les échantillons appris, au lieu de créer des règles, contrairement aux autres méthodes de classification.

Algorithme du plus proche voisin :

Étant donné un ensemble de catégories $C = \{c_1, c_2, \cdots, c_m\}$ également appelés classes, par exemple {"homme", "femme"}. Il existe également un ensemble d'apprentissage LS composé d'instances étiquetées :

$$LS = \{(o_1, c_{o_1}), (o_2, c_{o_2}), \cdots, (c_n, o_{o_n})\}$$

Comme cela n'a aucun sens d'avoir moins d'éléments étiquetés que de catégories, nous pouvons postuler que $n > m$ et dans la plupart des cas, même $n \gg m$. La tâche de classification consiste à attribuer une catégorie ou classe c à une instance arbitraire o.

Pour cela, il faut différencier deux cas :

- **Cas 1** : L'instance o est un élément de LS c'est-à-dire qu'il existe un tuple $(o, c) \in LS$. Dans ce cas, nous utiliserons la classe c comme résultat de la classification.
- **Cas 2** :Nous supposons maintenant que o n'est pas dans LS ou pour être précis :
$$\forall c \in C, (o, c) \notin LS$$

o est comparée à toutes les instances de LS. Une métrique de distance d est utilisée pour les comparaisons.

On détermine les k plus proches voisins de o c'est-à-dire les éléments présentant les distances les plus faibles.

k est une constante définie par l'utilisateur et un nombre entier positif, qui est généralement petit.

Le nombre k est généralement choisi comme la racine carrée de LS le nombre total de points dans l'ensemble de données d'apprentissage.

Pour déterminer la valeur de k plus proches voisins, nous réordonnons LS de la manière suivante :

$$(o_{i_1}, c_{o_{i_1}}), (o_{i_2}, c_{o_{i_2}}), \cdots, (o_{i_n}, c_{o_{i_n}})$$

de telle sorte que l'expression $d(o_{i_j}, o) < d(o_{i_{j+1}}, o)$ vraie $\forall j$ tq $1 \leq j \leq n-1$

L'ensemble des k plus proches voisins N_k est constitué des k éléments de cet ordre, c'est-à-dire:

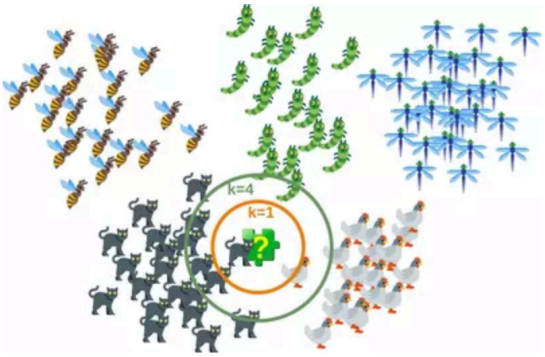
$$N_k = (o_{i_1}, c_{o_{i_1}}), (o_{i_2}, c_{o_{i_2}}), \cdots, (o_{i_n}, c_{o_{i_n}})$$

La classe la plus courante dans cet ensemble de plus proches voisins N_k sera attribuée à l'instance o. S'il n'existe pas de classe unique la plus courante, nous en prenons une arbitrairement.

Il n'existe pas de méthode générale pour définir une valeur optimale pour k. Cette valeur dépend des données. En règle générale, on peut dire que l'augmentation de k réduit le bruit mais rend les frontières moins distinctes.

L'algorithme du classificateur k-NN est l'un des plus simples de tous les algorithmes d'apprentissage automatique. k-NN est un type d'apprentissage basé sur les instances, ou apprentissage paresseux. Dans le domaine de l'apprentissage automatique, on entend par apprentissage paresseux une méthode d'apprentissage dans laquelle la généralisation des données d'apprentissage est retardée jusqu'à ce qu'une requête soit adressée au système. D'autre part, nous avons l'apprentissage rapide, où le système généralise généralement les données d'apprentissage avant de recevoir des requêtes. En d'autres termes : La fonction est seulement approximée localement et tous les calculs sont effectués, lorsque la classification réelle est réalisée.

L'image suivante montre de manière simple comment fonctionne le classificateur de plus proche voisin. La pièce du puzzle est inconnue. Pour savoir de quel animal il s'agit, nous devons trouver les voisins. Si $k = 1$, le seul voisin est un chat et nous supposons dans ce cas que la pièce du puzzle devrait également être un chat. Si $k = 4$, les plus proches voisins sont un poulet et trois chats. Dans ce cas également, il sera prudent de supposer que l'objet en question devrait être un chat.



Les k-plus-voisins à partir de zéro

Préparation du jeu de données

Avant de commencer à écrire un classificateur de plus proches voisins, nous devons réfléchir aux données, c'est-à-dire au jeu de données d'apprentissage et au jeu de données de test. Nous allons utiliser le jeu de données "iris" fourni par les jeux de données du module sklearn.

Le jeu de données consiste en 50 échantillons de chacune des trois espèces d'Iris suivantes

- Iris setosa,
- Iris virginica et
- Iris versicolor.

Quatre caractéristiques ont été mesurées pour chaque échantillon : la longueur et la largeur des sépales et des pétales, en centimètres.

Entrée [1]:
```
import numpy as np
from sklearn import datasets

iris = datasets.load_iris()
data = iris.data
labels = iris.target

for i in [0, 79, 99, 121]:
    print(f"index: {i:3}, features: {data[i]}, label: {labels[i]}")
```

```
index:   0, features: [5.1 3.5 1.4 0.2], label: 0
index:  79, features: [5.7 2.6 3.5 1. ], label: 1
index:  99, features: [5.7 2.8 4.1 1.3], label: 1
index: 121, features: [5.6 2.8 4.9 2. ], label: 2
```

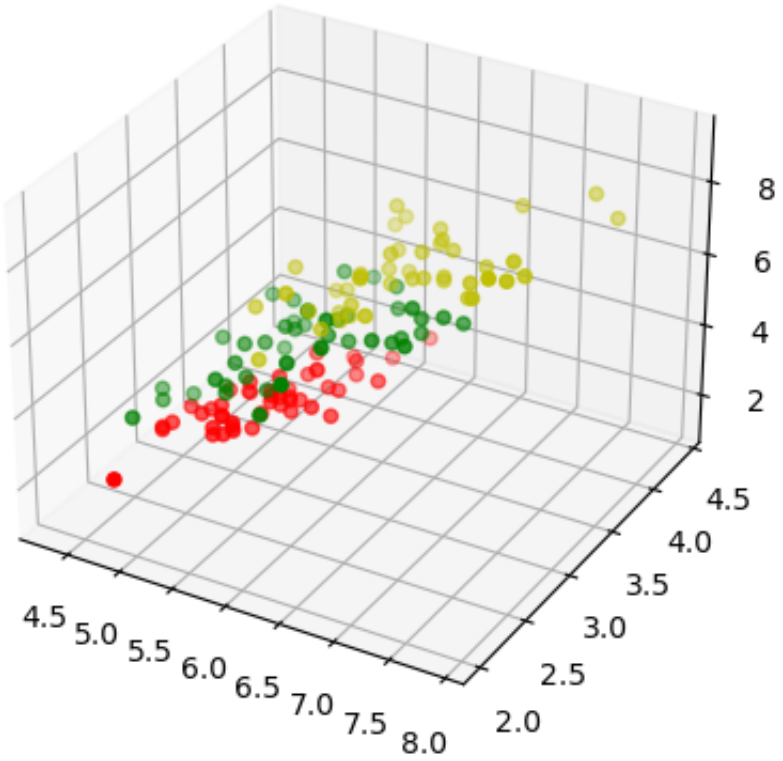
Nous créons un set d'apprentissage à partir des ensembles ci-dessus. Nous utilisons la permutation de np.random pour diviser les données de façon aléatoire.

Entrée [2]:
```
# seeding is only necessary for the website
#so that the values are always equal:
np.random.seed(42)
indices = np.random.permutation(len(data))

n_test_samples = 12     # number of test samples
learn_data = data[indices[:-n_test_samples]]
learn_labels = labels[indices[:-n_test_samples]]
test_data = data[indices[-n_test_samples:]]
test_labels = labels[indices[-n_test_samples:]]

print("The first samples of our learn set:")
print(f"{'index':7s}{'data':20s}{'label':3s}")
for i in range(5):
    print(f"{i:4d}   {learn_data[i]}   {learn_labels[i]:3}")

print("The first samples of our test set:")
print(f"{'index':7s}{'data':20s}{'label':3s}")
for i in range(5):
    print(f"{i:4d}   {learn_data[i]}   {learn_labels[i]:3}")
```

```
The first samples of our learn set:
index  data                 label
   0   [6.1 2.8 4.7 1.2]     1
   1   [5.7 3.8 1.7 0.3]     0
   2   [7.7 2.6 6.9 2.3]     2
   3   [6.  2.9 4.5 1.5]     1
   4   [6.8 2.8 4.8 1.4]     1
The first samples of our test set:
index  data                 label
   0   [6.1 2.8 4.7 1.2]     1
   1   [5.7 3.8 1.7 0.3]     0
   2   [7.7 2.6 6.9 2.3]     2
   3   [6.  2.9 4.5 1.5]     1
   4   [6.8 2.8 4.8 1.4]     1
```

Le code suivant est seulement nécessaire pour visualiser les données de notre set d'apprentissage. Nos données consistent en quatre valeurs par élément d'iris, nous allons donc réduire les données à trois valeurs en additionnant la troisième et la quatrième valeur. De cette façon, nous sommes capables de représenter les données dans un espace tridimensionnel :

Entrée [3]:
```
#%matplotlib widget

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

colours = ("r", "b")
X = []
for iclass in range(3):
    X.append([[], [], []])
    for i in range(len(learn_data)):
        if learn_labels[i] == iclass:
            X[iclass][0].append(learn_data[i][0])
            X[iclass][1].append(learn_data[i][1])
            X[iclass][2].append(sum(learn_data[i][2:]))

colours = ("r", "g", "y")

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for iclass in range(3):
       ax.scatter(X[iclass][0], X[iclass][1], X[iclass][2], c=colou
plt.show()
```



Métrique de la distance

Nous avons déjà mentionné en détail, que nous calculons les distances entre les points de l'échantillon et l'objet à classer. Pour calculer ces distances, nous avons besoin d'une fonction de distance.

Dans les problème de dimension n finie, on utilise généralement l'une des trois métriques de distance suivantes :

La distance euclidienne

La distance euclidienne entre deux points x et y dans le plan ou l'espace tridimensionnel mesure la longueur d'un segment de droite reliant ces deux points. Elle peut être calculée à partir des coordonnées cartésiennes des points à l'aide du théorème de Pythagore, c'est pourquoi on l'appelle aussi parfois la distance de Pythagore. La formule générale est la suivante:

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Distance de Manhattan

Elle est définie comme la somme des valeurs absolues des différences entre les coordonnées de x et y :

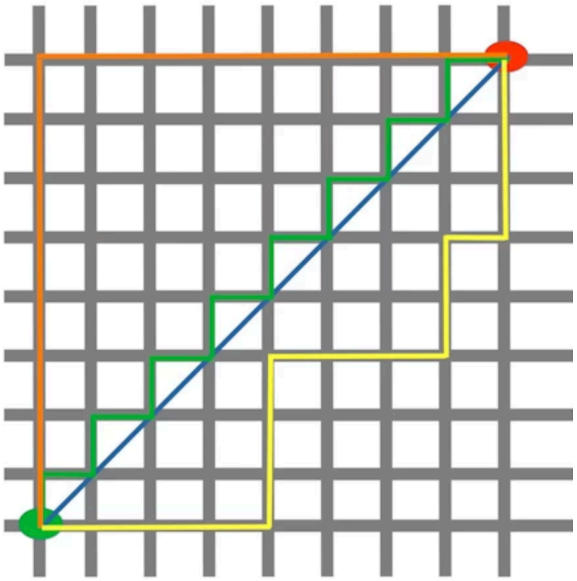
$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

Distance de Minkowski

La distance de Minkowski généralise la distance euclidienne et la distance de Manhattan en une seule métrique de distance. Si nous fixons le paramètre p de la formule suivante à 1, nous obtenons la distance de Manhattan et si nous utilisons la valeur 2, nous obtenons la distance euclidienne :

$$d(x, y) = \left(\sum_{i=1}^{n} (x_i - y_i)^p \right)^{\frac{1}{p}}$$

Le diagramme suivant visualise la distance euclidienne et la distance de Manhattan :



La ligne bleue illustre la distance d'Euclide entre le point vert et le point rouge. Sinon, vous pouvez aussi vous déplacer sur la ligne orange, verte ou jaune du point vert au point rouge. Les lignes correspondent à la distance de Manhattan. La longueur est égale.

Détermination des voisins

Pour déterminer la similarité entre deux instances, nous allons utiliser la distance euclidienne.

Nous pouvons calculer la distance euclidienne avec la fonction norm du module `np.linalg` :

Entrée [4]:
```
def distance(instance1, instance2):
    """ Calculates the Eucledian distance between two instances"""
    return np.linalg.norm(np.subtract(instance1, instance2))

print(distance([3, 5], [1, 1]))
print(distance(learn_data[3], learn_data[44]))
```

```
4.47213595499958
3.4190641994557516
```

La fonction `get_neighbors` renvoie une liste de k voisins, qui sont les plus proches de l'instance test_instance :

Entrée [5]:
```
def get_neighbors(training_set,
                  labels,
                  test_instance,
                  k,
                  distance):
    """
    get_neighors calculates a list of the k nearest neighbors
    of an instance 'test_instance'.
    The function returns a list of k 3-tuples.
    Each 3-tuples consists of (index, dist, label)
    where
    index    is the index from the training_set,
    dist     is the distance between the test_instance and the
             instance training_set[index]
    distance is a reference to a function used to calculate the
             distances
    """
    distances = []
    for index in range(len(training_set)):
        dist = distance(test_instance, training_set[index])
        distances.append((training_set[index], dist, labels[index]))
    distances.sort(key=lambda x: x[1])
    neighbors = distances[:k]
    return neighbors
```

Nous allons tester la fonction avec nos échantillons d'iris :

Entrée [6]:
```
for i in range(5):
    neighbors = get_neighbors(learn_data,
                              learn_labels,
                              test_data[i],
                              3,
                              distance=distance)
    print("Index:          ",i,'\n',
          "Testset Data:  ",test_data[i],'\n',
          "Testset Label: ",test_labels[i],'\n',
          "Neighbors:      ",neighbors,'\n')
```

```
Index:          0
 Testset Data:   [5.7 2.8 4.1 1.3]
 Testset Label:  1
 Neighbors:       [(array([5.7, 2.9, 4.2, 1.3]), 0.141421356237309
95, 1), (array([5.6, 2.7, 4.2, 1.3]), 0.17320508075688815, 1), (ar
ray([5.6, 3. , 4.1, 1.3]), 0.22360679774997935, 1)]

Index:          1
 Testset Data:   [6.5 3.  5.5 1.8]
 Testset Label:  2
 Neighbors:       [(array([6.4, 3.1, 5.5, 1.8]), 0.141421356237309
3, 2), (array([6.3, 2.9, 5.6, 1.8]), 0.24494897427831785, 2), (arr
ay([6.5, 3. , 5.2, 2. ]), 0.3605551275463988, 2)]

Index:          2
 Testset Data:   [6.3 2.3 4.4 1.3]
 Testset Label:  1
 Neighbors:       [(array([6.2, 2.2, 4.5, 1.5]), 0.264575131106458
6, 1), (array([6.3, 2.5, 4.9, 1.5]), 0.574456264653803, 1), (array
([6. , 2.2, 4. , 1. ]), 0.5916079783099617, 1)]

Index:          3
 Testset Data:   [6.4 2.9 4.3 1.3]
 Testset Label:  1
 Neighbors:       [(array([6.2, 2.9, 4.3, 1.3]), 0.200000000000000
18, 1), (array([6.6, 3. , 4.4, 1.4]), 0.2645751311064587, 1), (arr
ay([6.6, 2.9, 4.6, 1.3]), 0.3605551275463984, 1)]

Index:          4
 Testset Data:   [5.6 2.8 4.9 2. ]
 Testset Label:  2
 Neighbors:       [(array([5.8, 2.7, 5.1, 1.9]), 0.316227766016837
55, 2), (array([5.8, 2.7, 5.1, 1.9]), 0.31622776601683755, 2), (ar
ray([5.7, 2.5, 5. , 2. ]), 0.33166247903553986, 2)]
```

Voter pour obtenir un seul résultat

Nous allons maintenant écrire une fonction de vote. Cette fonction utilise la classe Counter des collections pour compter la quantité de classes à l'intérieur d'une liste d'instances. Cette liste d'instance sera bien sûr les voisins. La fonction vote retourne la classe la plus commune :

Entrée [7]:
```
from collections import Counter

def vote(neighbors):
    class_counter = Counter()
    for neighbor in neighbors:
        class_counter[neighbor[2]] += 1
    return class_counter.most_common(1)[0][0]
```

Nous allons tester 'vote' sur norte dataset:

Entrée [8]:
```
for i in range(n_test_samples):
    neighbors = get_neighbors(learn_data,
                              learn_labels,
                              test_data[i],
                              3,
                              distance=distance)
    print("index: ", i,
          ", result of vote: ", vote(neighbors),
          ", label: ", test_labels[i],
          ", data: ", test_data[i])
```

```
index:  0 , result of vote:  1 , label:  1 , data:  [5.7 2.8 4.1 1
.3]
index:  1 , result of vote:  2 , label:  2 , data:  [6.5 3.  5.5 1
.8]
index:  2 , result of vote:  1 , label:  1 , data:  [6.3 2.3 4.4 1
.3]
index:  3 , result of vote:  1 , label:  1 , data:  [6.4 2.9 4.3 1
.3]
index:  4 , result of vote:  2 , label:  2 , data:  [5.6 2.8 4.9 2
. ]
index:  5 , result of vote:  2 , label:  2 , data:  [5.9 3.  5.1 1
.8]
index:  6 , result of vote:  0 , label:  0 , data:  [5.4 3.4 1.7 0
.2]
index:  7 , result of vote:  1 , label:  1 , data:  [6.1 2.8 4.  1
.3]
index:  8 , result of vote:  1 , label:  2 , data:  [4.9 2.5 4.5 1
.7]
index:  9 , result of vote:  0 , label:  0 , data:  [5.8 4.  1.2 0
.2]
index:  10 , result of vote:  1 , label:  1 , data:  [5.8 2.6 4.
1.2]
index:  11 , result of vote:  2 , label:  2 , data:  [7.1 3.  5.9
2.1]
```

Nous pouvons voir que les prédictions correspondent aux résultats étiquetés, sauf dans le cas de l'élément avec l'indice 8.

`vote_prob` est une fonction comme `vote` mais retourne le nom de la classe et la probabilité pour cette classe :

Entrée [9]:
```
def vote_prob(neighbors):
    class_counter = Counter()
    for neighbor in neighbors:
        class_counter[neighbor[2]] += 1
    labels, votes = zip(*class_counter.most_common())
    winner = class_counter.most_common(1)[0][0]
    votes4winner = class_counter.most_common(1)[0][1]
    return winner, votes4winner/sum(votes)
for i in range(n_test_samples):
    neighbors = get_neighbors(learn_data,
                              learn_labels,
                              test_data[i],
                              5,
                              distance=distance)
    print("index: ", i,
          ", vote_prob: ", vote_prob(neighbors),
          ", label: ", test_labels[i],
          ", data: ", test_data[i])
```

```
index:  0 , vote_prob:  (1, 1.0) , label:  1 , data:  [5.7 2.8 4.1
1.3]
index:  1 , vote_prob:  (2, 1.0) , label:  2 , data:  [6.5 3.  5.5
1.8]
index:  2 , vote_prob:  (1, 1.0) , label:  1 , data:  [6.3 2.3 4.4
1.3]
index:  3 , vote_prob:  (1, 1.0) , label:  1 , data:  [6.4 2.9 4.3
1.3]
index:  4 , vote_prob:  (2, 1.0) , label:  2 , data:  [5.6 2.8 4.9
2. ]
index:  5 , vote_prob:  (2, 0.8) , label:  2 , data:  [5.9 3.  5.1
1.8]
index:  6 , vote_prob:  (0, 1.0) , label:  0 , data:  [5.4 3.4 1.7
0.2]
index:  7 , vote_prob:  (1, 1.0) , label:  1 , data:  [6.1 2.8 4.
1.3]
index:  8 , vote_prob:  (1, 1.0) , label:  2 , data:  [4.9 2.5 4.5
1.7]
index:  9 , vote_prob:  (0, 1.0) , label:  0 , data:  [5.8 4.  1.2
0.2]
index:  10 , vote_prob:  (1, 1.0) , label:  1 , data:  [5.8 2.6 4.
1.2]
index:  11 , vote_prob:  (2, 1.0) , label:  2 , data:  [7.1 3.  5.
9 2.1]
```

Le classificateur des plus proches voisins pondérés

Nous n'avons examiné que k éléments dans le voisinage d'un objet inconnu **UO**, et avons procédé à un vote majoritaire. L'utilisation du vote majoritaire s'est avérée très efficace dans notre exemple précédent, mais cela ne tenait pas compte du raisonnement suivant : Plus un voisin est éloigné, plus il **dévie** du résultat **réel**. En d'autres termes, nous pouvons faire plus confiance aux voisins les plus proches qu'aux plus éloignés. Supposons que nous ayons 11 voisins d'un élément inconnu **UO**. Les cinq voisins les plus proches appartiennent à une classe **A** et les six autres, qui sont plus éloignés, appartiennent à une classe **B**. Quelle classe doit-on attribuer à **UO**? L'approche précédente dit **B**, car nous avons un vote de 6 contre 5 en faveur de **B**. D'un autre côté, les 5 voisins les plus proches sont tous **A** et cela devrait compter davantage.

Pour poursuivre cette stratégie, nous pouvons attribuer des poids aux voisins de la manière suivante : Le voisin le plus proche d'une instance reçoit un poids 1/1 le deuxième plus proche obtient un poids de 1/2 et ensuite jusqu'à 1/k pour le voisin le plus éloigné.

Cela signifie que nous utilisons les séries harmoniques comme poids :

$$\sum_{i=1}^{n-1} \frac{1}{1+i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$$

Nous mettons cela en œuvre dans la fonction suivante :

Entrée [10]:
```
def vote_harmonic_weights(neighbors, all_results=True):
    class_counter = Counter()
    number_of_neighbors = len(neighbors)
    for index in range(number_of_neighbors):
        class_counter[neighbors[index][2]] += 1/(index+1)
    labels, votes = zip(*class_counter.most_common())
    #print(labels, votes)
    winner = class_counter.most_common(1)[0][0]
    votes4winner = class_counter.most_common(1)[0][1]
    if all_results:
        total = sum(class_counter.values(), 0.0)
        for key in class_counter:
             class_counter[key] /= total
        return winner, class_counter.most_common()
    else:
        return winner, votes4winner / sum(votes)
```

Entrée [11]:
```
for i in range(n_test_samples):
    neighbors = get_neighbors(learn_data,
                              learn_labels,
                              test_data[i],
                              6,
                              distance=distance)
    print("index: ", i,
          ", result of vote: ",
          vote_harmonic_weights(neighbors,
                                all_results=True))
```

```
index:  0 , result of vote:  (1, [(1, 1.0)])
index:  1 , result of vote:  (2, [(2, 1.0)])
index:  2 , result of vote:  (1, [(1, 1.0)])
index:  3 , result of vote:  (1, [(1, 1.0)])
index:  4 , result of vote:  (2, [(2, 0.9319727891156463), (1, 0.0
6802721088435375)])
index:  5 , result of vote:  (2, [(2, 0.8503401360544217), (1, 0.1
4965986394557826)])
index:  6 , result of vote:  (0, [(0, 1.0)])
index:  7 , result of vote:  (1, [(1, 1.0)])
index:  8 , result of vote:  (1, [(1, 1.0)])
index:  9 , result of vote:  (0, [(0, 1.0)])
index:  10 , result of vote:  (1, [(1, 1.0)])
index:  11 , result of vote:  (2, [(2, 1.0)])
```

L'approche précédente ne prenait en compte que le classement des voisins en fonction de leur distance. Nous pouvons améliorer le vote en utilisant la distance réelle. Dans ce but, nous allons écrire une nouvelle fonction de vote :

Entrée [12]:
```
def vote_distance_weights(neighbors, all_results=True):
    class_counter = Counter()
    number_of_neighbors = len(neighbors)
    for index in range(number_of_neighbors):
        dist = neighbors[index][1]
        label = neighbors[index][2]
        class_counter[label] += 1 / (dist**2 + 1)
    labels, votes = zip(*class_counter.most_common())
    #print(labels, votes)
    winner = class_counter.most_common(1)[0][0]
    votes4winner = class_counter.most_common(1)[0][1]
    if all_results:
        total = sum(class_counter.values(), 0.0)
        for key in class_counter:
            class_counter[key] /= total
        return winner, class_counter.most_common()
    else:
        return winner, votes4winner / sum(votes)
```

Entrée [13]:
```
for i in range(n_test_samples):
    neighbors = get_neighbors(learn_data,
                              learn_labels,
                              test_data[i],
                              6,
                              distance=distance)
    print("index: ", i,
          ", result of vote: ",
          vote_distance_weights(neighbors,
                                all_results=True))
```

```
index:  0 , result of vote:  (1, [(1, 1.0)])
index:  1 , result of vote:  (2, [(2, 1.0)])
index:  2 , result of vote:  (1, [(1, 1.0)])
index:  3 , result of vote:  (1, [(1, 1.0)])
index:  4 , result of vote:  (2, [(2, 0.8490154592118361), (1, 0.1
5098454078816387)])
index:  5 , result of vote:  (2, [(2, 0.6736137462184479), (1, 0.3
263862537815521)])
index:  6 , result of vote:  (0, [(0, 1.0)])
index:  7 , result of vote:  (1, [(1, 1.0)])
index:  8 , result of vote:  (1, [(1, 1.0)])
index:  9 , result of vote:  (0, [(0, 1.0)])
index:  10 , result of vote:  (1, [(1, 1.0)])
index:  11 , result of vote:  (2, [(2, 1.0)])
```

Un autre exemple de classification par les plus proches voisins

Nous voulons tester les fonctions précédentes avec un autre ensemble de données très simple :

Entrée [14]:
```
train_set = [(1, 2, 2),
             (-3, -2, 0),
             (1, 1, 3),
             (-3, -3, -1),
             (-3, -2, -0.5),
             (0, 0.3, 0.8),
             (-0.5, 0.6, 0.7),
             (0, 0, 0)
            ]

labels = ['apple',  'banana', 'apple',
          'banana', 'apple', "orange",
          'orange', 'orange']

k = 2
for test_instance in [(0, 0, 0), (2, 2, 2),
                      (-3, -1, 0), (0, 1, 0.9),
                      (1, 1.5, 1.8), (0.9, 0.8, 1.6)]:
    neighbors = get_neighbors(train_set,
                              labels,
                              test_instance,
                              k,
                              distance=distance)

    print("vote distance weights: ",
          vote_distance_weights(neighbors))
```

```
vote distance weights:  ('orange', [('orange', 1.0)])
vote distance weights:  ('apple', [('apple', 1.0)])
vote distance weights:  ('banana', [('banana', 0.5294117647058824)
, ('apple', 0.47058823529411764)])
vote distance weights:  ('orange', [('orange', 1.0)])
vote distance weights:  ('apple', [('apple', 1.0)])
vote distance weights:  ('apple', [('apple', 0.5084745762711865),
('orange', 0.4915254237288135)])
```

kNN en linguistique

L'exemple suivant provient de la linguistique informatique. Nous montrons comment nous pouvons utiliser un classificateur kNN pour reconnaître les mots mal orthographiés.

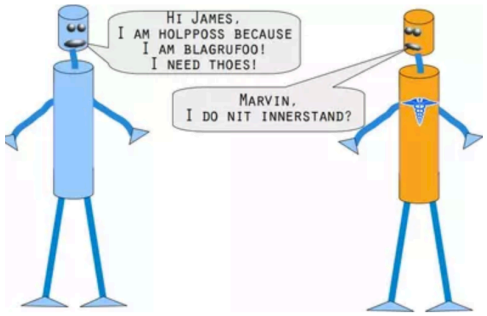
Nous utilisons un module appelé levenshtein, qui utilise la distance de Levenshtein.

Entrée [15]:
```
from levenshtein import levenshtein

cities = open("data/city_names.txt").readlines()
cities = [city.strip() for city in cities]

for city in ["Freiburg", "Frieburg", "Freiborg",
             "Hamborg", "Sahrluis"]:
    neighbors = get_neighbors(cities,
                              cities,
                              city,
                              2,
                              distance=levenshtein)

    print("vote_distance_weights: ", vote_distance_weights(neighbor
```

```
vote_distance_weights:  ('Freiberg', [('Freiberg', 0.8333333333333
334), ('Freising', 0.16666666666666669)])
vote_distance_weights:  ('Lüneburg', [('Lüneburg', 0.5), ('Duisbur
g', 0.5)])
vote_distance_weights:  ('Freiberg', [('Freiberg', 0.8333333333333
334), ('Freising', 0.16666666666666669)])
vote_distance_weights:  ('Hamburg', [('Hamburg', 0.714285714285714
3), ('Bamberg', 0.28571428571428575)])
vote_distance_weights:  ('Saarlouis', [('Saarlouis', 0.83870967741
93549), ('Bayreuth', 0.16129032258064516)])
```

Vous vous demandez peut-être pourquoi la ville de Freiburg n'a pas été reconnue. La raison en est que notre fichier de données avec les noms de ville ne contient que "Freiburg im Breisgau". Si vous ajoutez "Freiburg" comme entrée, elle sera également reconnue.

Marvin et James nous présentent notre prochain exemple :



Pouvez-vous aider Marvin et James ?

Vous aurez besoin d'un dictionnaire anglais et d'un classificateur de type k-nearest Neighbor pour résoudre ce problème. Il se situe dans le répertoire `data` de ce notebook.

Nous utilisons des mots extrêmement mal orthographiés dans l'exemple suivant. Nous constatons que notre simple fonction vote_prob ne donne de bons résultats que dans deux cas : En corrigeant "holpposs" en "helpless" et "blagrufoo" en "barefoot". Alors que notre vote à distance fonctionne bien dans tous les cas. Bon, nous devons admettre que nous avions "liberty" à l'esprit, lorsque nous avons écrit "liberdi", mais suggérer "liberal" est un bon choix.

Entrée [16]:
```
words = []
with open("data/british-english.txt") as fh:
    for line in fh:
        word = line.strip()
        words.append(word)

for word in ["holpful", "kundnoss", "holpposs", "thoes", "innerstan
             "blagrufoo", "liberdi"]:
    neighbors = get_neighbors(words,
                              words,
                              word,
                              3,
                              distance=levenshtein)

    print("vote_distance_weights: ", vote_distance_weights(neighbor
                                                          all_resu
    print("vote_prob: ", vote_prob(neighbors))
    print("vote_distance_weights: ", vote_distance_weights(neighbor
```

```
vote_distance_weights:  ('helpful', 0.5555555555555556)
vote_prob:  ('helpful', 0.3333333333333333)
vote_distance_weights:  ('helpful', [('helpful', 0.555555555555555
6), ('doleful', 0.2222222222222227), ('hopeful', 0.22222222222222
227)])
vote_distance_weights:  ('kindness', 0.5)
vote_prob:  ('kindness', 0.3333333333333333)
vote_distance_weights:  ('kindness', [('kindness', 0.5), ('fondnes
s', 0.25), ('kudos', 0.25)])
vote_distance_weights:  ('helpless', 0.3333333333333333)
vote_prob:  ('helpless', 0.3333333333333333)
vote_distance_weights:  ('helpless', [('helpless', 0.3333333333333
333), ("hippo's", 0.3333333333333333), ('hippos', 0.33333333333333
33)])
vote_distance_weights:  ('hoes', 0.3333333333333333)
vote_prob:  ('hoes', 0.3333333333333333)
vote_distance_weights:  ('hoes', [('hoes', 0.3333333333333333), ('
shoes', 0.3333333333333333), ('thees', 0.3333333333333333)])
vote_distance_weights:  ('understand', 0.5)
vote_prob:  ('understand', 0.3333333333333333)
vote_distance_weights:  ('understand', [('understand', 0.5), ('int
erstate', 0.25), ('understands', 0.25)])
vote_distance_weights:  ('barefoot', 0.4333333333333333)
vote_prob:  ('barefoot', 0.3333333333333333)
vote_distance_weights:  ('barefoot', [('barefoot', 0.4333333333333
333), ('Baguio', 0.2833333333333333), ('Blackfoot', 0.283333333333
3333)])
vote_distance_weights:  ('liberal', 0.4)
vote_prob:  ('liberal', 0.3333333333333333)
vote_distance_weights:  ('liberal', [('liberal', 0.4), ('liberty',
0.4), ('Hibernia', 0.2)])
```

Suivant (9_knn_sklearn.ipynb)