KNN et SQL

November 5, 2024

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
from numpy import array # on l'utilise très souvent, autant ne pas préfixer
```

1 Avertissement

Ce TP, qui mélange requêtes SQL et code Python, est destiné uniquement aux 5/2 qui ont terminé le TP plus classique de recherche des plus proches voisins en Python.

2 Algorithme des k plus proches voisins

L'Algorithme des k *plus proches voisins* appartient à la famille des algorithmes d'apprentissage automatique (machine learning). C' est un algorithme d'apprentissage supervisé : il est nécessaire d'avoir des données labellisées. À partir d'un ensemble E de données labellisées, on veut classer (i.e déterminer le label) d'une nouvelle donnée (n'appartenant pas à E). On peut aussi l'utiliser à des fins de régression (on cherche alors à déterminer une valeur à la place d'une classe)

Nous l'utilisons ici pour déterminer, à partir de la dimension des pétales, l'appartenance d'une fleur à une catégorie d'Iris.

On dispose d'une base de données écrite en SQLITE. Cette base contient une table `iris` ici

[2]:
```
import pandas
iris=pandas.read_csv("../iris.csv")
iris
```

[2]:

	sepal.length	sepal.width	petal.length	petal.width	variety
0	5.1	3.5	1.4	0.2	Setosa
1	4.9	3.0	1.4	0.2	Setosa
2	4.7	3.2	1.3	0.2	Setosa
3	4.6	3.1	1.5	0.2	Setosa
4	5.0	3.6	1.4	0.2	Setosa
..	…	…	…	…	…
145	6.7	3.0	5.2	2.3	Virginica
146	6.3	2.5	5.0	1.9	Virginica
147	6.5	3.0	5.2	2.0	Virginica
148	6.2	3.4	5.4	2.3	Virginica
149	5.9	3.0	5.1	1.8	Virginica

```
[150 rows x 5 columns]
```

[3]:
```
# Préparer la base de données
import sqlite3
conn = sqlite3.connect(":memory:")
#sqlite3.connect(":memory:") : la BDD n'existe que dans la RAM, pas écrite sur␣
 ↪le dique

def executer(req):
    cur = conn.cursor()
    cur.execute(req)
    t = cur.fetchall()
    cur.close()
    return t

df = pandas.read_csv("../iris.csv")
# transformer en BDD SQlite
df.to_sql("iris", conn, if_exists='append', index=False)

#req = """SELECT * FROM iris"""
#executer(cur,req)
```

[3]:
```
150
```

Pour comprendre le schéma de cette table `iris`, on donne ci-dessous la requête qui en a permis la création :

```
CREATE TABLE "iris" (
"sepal.length" REAL,
  "sepal.width" REAL,
  "petal.length" REAL,
  "petal.width" REAL,
  "variety" TEXT
)
```

Les attributs ont tous pour domaine `REAL` (flottant en SQLite) sauf `variety` : une chaîne de caractères qui indique la variété d'Iris à laquelle on a affaire. Cette table est le célèbre jeu de données "Iris" qui contient des longueurs et largeurs de pétales et de sépales selon variétés de fleurs. Enfin on rappelle que *width* signifie *largeur*, *length* :*longueur*, *sepal* : *sépale* et *petal* : *pétale*.

On dispose dans toute la suite de la fonction `executer(req)` qui retourne sous forme de liste de tuples `Python` la table calculée par la requête `req`. Cette fonction nous permet de faire communiquer `Python` et `SQLite`. Par exemple, on évalue en `SQL` toutes les lignes de la table `iris` mais on choisit en `Python` de n'afficher que les 3 premières :

[6]:
```
#projection sur l'ensemble des attributs :
req = """
SELECT * FROM iris
```

```
"""
t= executer(req)# t contient tous les enregstrements de la table
t[:3]# on n'en retient que 3 mais il y en a plus
```

[6]:
```
[(5.1, 3.5, 1.4, 0.2, 'Setosa'),
 (4.9, 3.0, 1.4, 0.2, 'Setosa'),
 (4.7, 3.2, 1.3, 0.2, 'Setosa')]
```

Précisons que Lorsqu'on projette sur une seule colonne, on obtient une liste de tuples à un seul élément comme

```
[(1.2,), (2.3,), (0.75,)]
```

1. L'exemple plus haut donne les les 3 premières lignes de `iris` en mêlant une requête `SELECT * FROM iris` et un peu de slicing Python `t[:3]`. Ecrire une requête qui retourne sans slicing ces 3 premières lignes mais en utilisant uniquement `SQL` et la fonction `executer`.

```
req=... # à vous de l'écrire
```

[8]:
```
t= executer(req)
t
```

[8]:
```
[(5.1, 3.5, 1.4, 0.2, 'Setosa'),
 (4.9, 3.0, 1.4, 0.2, 'Setosa'),
 (4.7, 3.2, 1.3, 0.2, 'Setosa')]
```

2. Ecrire une requête qui donne la liste sans doublon des variétés décrites dans la table `iris`.

```
req=... # à vous de l'écrire
```

[10]:
```
varieties = executer(req)
varieties
```

[10]:
```
[('Setosa',), ('Versicolor',), ('Virginica',)]
```

3. Ecrire une requête `reqx` qui permet de donner la liste `x` des longueurs de pétales et une seconde, `reqy`, qui donne la liste `y` des longueurs de pétales.

```
reqx = ...
reqy = ...
```

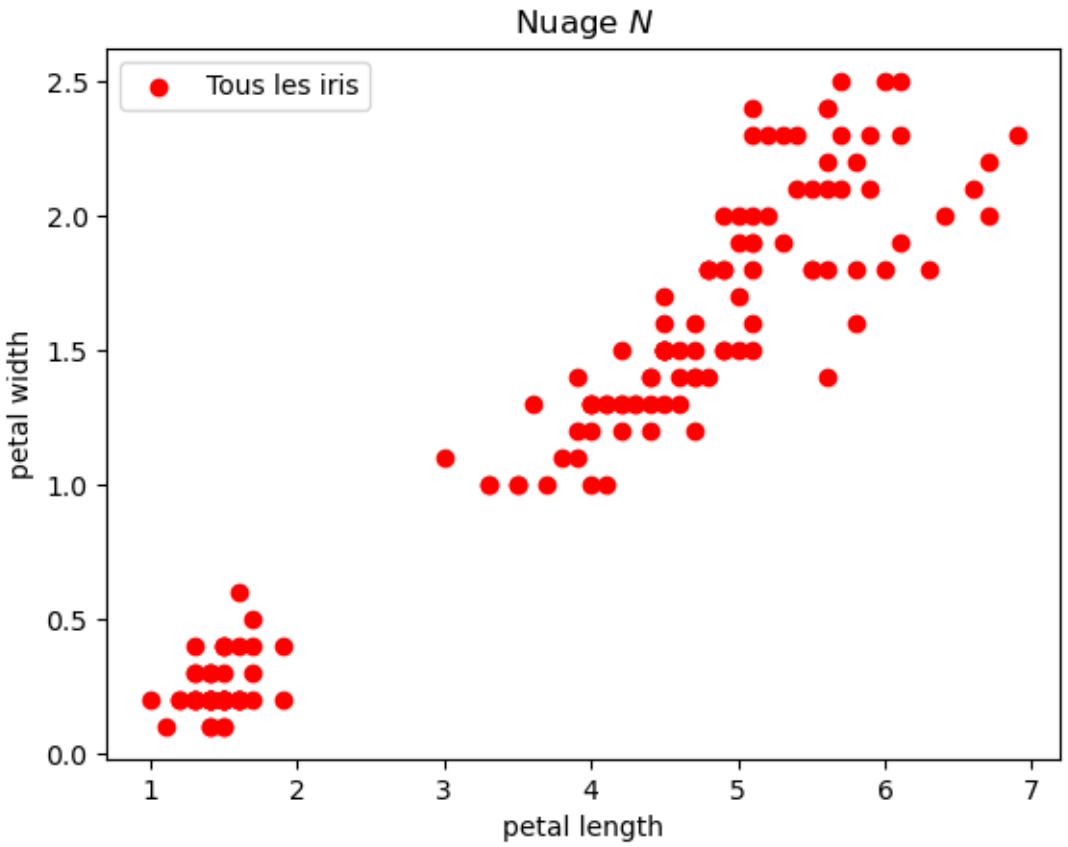
[12]:
```
x = executer(reqx)
y= executer(reqy)
x[:3], y[:3]
```

[12]:
```
([(1.4,), (1.4,), (1.3,)], [(0.2,), (0.2,), (0.2,)])
```

On représente pour tous les iris le nuage de points (en rouge) dont les coordonnées sont : en abscisse les longueurs de pétales et en ordonnées les largeurs. On indique une étiquette à mettre dans la légende.

Voici le code Python :

[13]:
```
_,ax = plt.subplots()
plt.scatter(x,y,color='r',label = 'Tous les iris')
ax.set_xlabel('petal length')
ax.set_ylabel('petal width')
plt.legend()
plt.title('Nuage $N$')
plt.show()
```

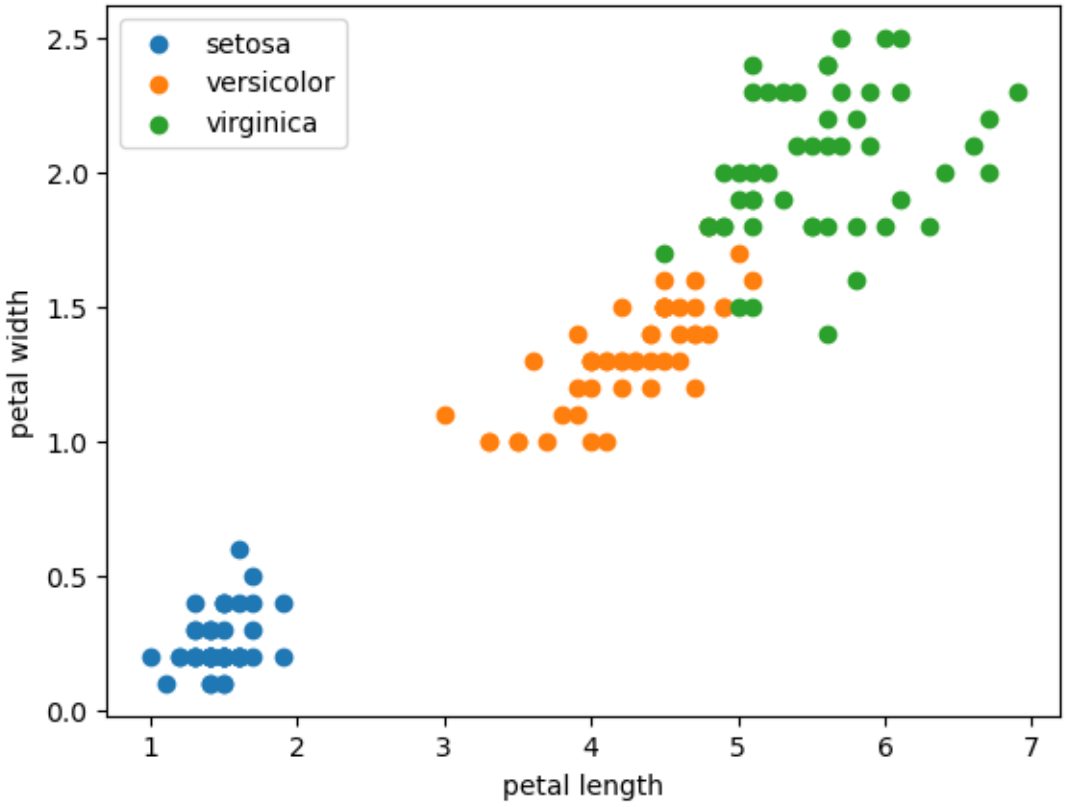


Pour la suite, nous notos N le nuage des longueurs et largeurs des pétales des iris ci-dessus.

4. On différencie maintenant les iris selon leur variété. On représente en vert les Setosa, en rouge les Virginica et en bleu les versicolor. Il faut donc obtenir les longueurs et largeurs de chacune des 3 catégories. Compléter le code ci-dessous pour obtenir un graphique différencié :

```
_,ax = plt.subplots()
xsetosa = executer(...)#compléter
ysetosa = executer(...)#compléter
xversicolor = executer(...)
yversicolor = executer(...)
```

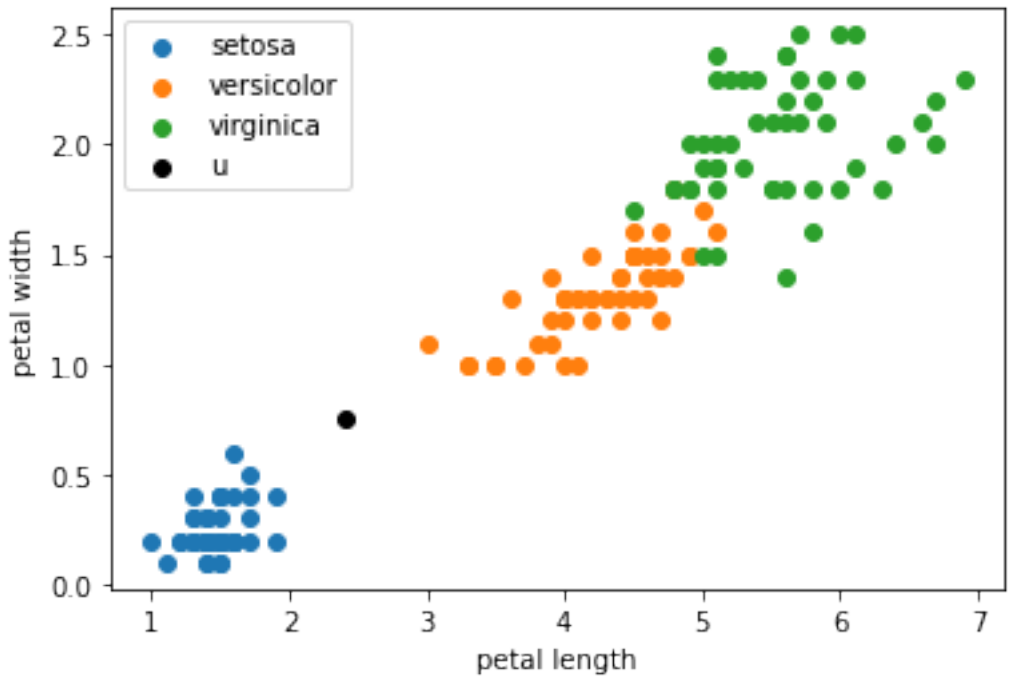
```
xvirginica = executer(...)
yvirginica = executer(...)
plt.scatter(...)#"setosa"
plt.scatter(...)#"versicolor"
plt.scatter(...#"virginica")
ax.set_xlabel('petal length')
ax.set_ylabel('petal width')
plt.legend()
plt.show()
```



Nous utilisons l'algorithme des k plus proches voisins pour déterminer l'appartenance d'une fleur à l'une des variétés d'Iris (méthode de *classification*) lorsqu'on ne connaît que la longueur et la largeur de ses pétales.

Considérons une donnée u (donc une fleur) dont on connaît juste le vecteur des longueurs et largeurs des pétales mais pas les autres informations, par exemple $\vec{u} = (2.4, 0.75)$ (on en déduit que la fleur a une longueur de pétale de 2.4 et une largeur de 0.75).

Il s'agit de trouver la variété d'Iris dont u est la plus proche. On représente u en noir sur le dessin :



Nous fixons un entier k. L'algorithme *des plus proches voisins* cherche les k plus proches voisins de u dans le nuage N au sens de la distance euclidienne, repère la variété majoritaire (ou l'une des variétés majoritaires en cas d'ex-aequo) et décide que u est de cette variété.

5. On pose $\vec{u} = (2.4, 0.75)$ dans cette question. Ecrire la requête SQL qui donne les cinq premiers résultats du tableau des carrés des distances à u des points du nuage rangés par ordre croissant.

```
req = ...
```

```
[(0.3725,), (0.48250000000000015,), (0.5525,), (0.5525,), (0.6124999999999999,)]
```

[18]:
```
[0.372, 0.483, 0.552, 0.552, 0.612]
```

Pour la suite, nous aurons besoin de combler en Python les *trous* dans des squelettes de requêtes. Cela se fait très facilement comme on le voit sur l'exemple suivant :

[17]:
```
req = """ SELECT {}, {} FROM {} WHERE {} > {}""" # squelette : requête␣
  ↪pré-remplie
print(req)
r = req.format("nom","prénom","élève","note",15.3) # requête complète
print(r)
```

```
SELECT {}, {} FROM {} WHERE {} > {}
SELECT nom, prénom FROM élève WHERE note > 15.3
```

6. On généralise un peu la requête de la question 5 pour la rendre adaptable à d'autres contextes (plus de coordonnées, autre nom de table, autres noms de colonnes). A l'aide notamment d'un

squelette de requête, écrire une fonction `proche(k,u,name,cols)` qui prend en paramètres un entier k, un point u modélisé par un tableau `numpy`, un nom de table `name` (comme par exemple "iris") et une liste d'attributs `cols`. Dans `cols` il y a autant d'attributs que la dimension de u et chacun des attributs admet les flottants pour domaine . La fonction donne les k points les plus proches de u dans le nuage de points défini par les colonnes listées dans `cols`. Par exemple, si `cols` est la liste `["petal.length", "petal.width"]`, les points forment le nuage N vu plus haut. Avec `["sepal.length", "sepal.width","petal.length", "petal.width"]`, le nuage est constitué de points à 4 dimensions. Le code doit donc pouvoir s'adapter à des listes de noms d'attributs de longueurs arbitraires. Par exemple, la commande `proches(5,np.array([2.4,0.75]),"iris",["petal.length", "petal.width"])` doit donner le même résultat que celui de la question 5.

[22]:
```
proches(5,np.array([2.4,0.75]),\
        "iris",["petal.length", "petal.width"])
```

[22]:
```
[(0.3725,),
 (0.48250000000000015,),
 (0.5525,),
 (0.5525,),
 (0.6124999999999999,)]
```

7. Il reste maintenant à déterminer la variété associée à la donnée u à partir du vecteur $\vec{u}$ selon la méthode des k plus proches voisins. Dans cette question, $k = 5$ et le vecteur $\vec{u}$ considéré a pour coordonnées $(2.4, 0.75)$. Parmi les 5 plus proches voisins, la requête détermine la variété majoritaire (et choisit n'importe laquelle si il y a 2 variétés majoritaires) : elle retourne le nom de cette variété et le nombre de représentants parmi les 5 plus proches voisins de u. Cette requête étant complexe, il faut bien l'expliquer.

[20]:
```
[('Setosa', 4)]
```

Quoiqu'il en soit, l'algorithme des k plus proches voisins nous donne bien la variété dont u est la plus proche.

2.1 Autres questions

Questions sans rapport avec l'algorithme des k plus proches voisins :

7. Donner les requêtes qui permettent de calculer :
   - le minimum des longueurs de sépales
   - les variétés et, pour chacune, la moyenne des longueurs de sépales de ses échantillons.
   - les largeurs de sépales qui sont aussi des longeurs de pétales.

```
[(4.3,)]
```

```
[('Setosa', 5.005999999999999), ('Versicolor', 5.936), ('Virginica',
6.587999999999998)]
```

```
[(3.0,), (3.3,), (3.5,), (3.6,), (3.7,), (3.8,), (3.9,), (4.0,), (4.1,), (4.2,),
(4.4,)]
```