td_note_2019_2

July 1, 2022

1 1A.e - Enoncé 23 octobre 2018 (2)

Correction du second énoncé de l'examen du 23 octobre 2018. L'énoncé propose une méthode pour renseigner les valeurs manquantes dans une base de deux variables.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

On sait d'après les dernières questions qu'il faudra tout répéter plusieurs fois. On prend le soin d'écrire chaque question dans une fonction.

1.1 Q1 - échantillon aléatoire

Générer un ensemble de $N = 1000$ couples aléatoires (X_i, Y_i) qui vérifient :

- X_i suit une loi normale de variance 1.
- $Y_i = 2X_i + \epsilon_i$ où ϵ_i suit une loi normale de variance 1.

[2]:
```
import numpy.random as rnd
import numpy

def random_mat(N):
    mat = numpy.zeros((N, 2))
    mat[:, 0] = rnd.normal(size=(N,))
    mat[:, 1] = mat[:, 0] * 2 + rnd.normal(size=(N,))
    return mat

N = 1000
mat = random_mat(N)
mat[:5]
```

[2]:
```
array([[-1.56987627, -0.87585938],
       [ 0.21230699,  1.85706677],
       [-1.32971056, -1.31614371],
       [ 0.99469359,  2.63550262],
       [-1.90844194, -3.84040783]])
```

Remarque : Un élève a retourné cette réponse, je vous laisse chercher pourquoi ce code produit deux variables tout-à-fait décorrélées.

```
def random_mat(N=1000):
    A = np.random.normal(0,1,(N,2))
    A[:,1] = 2*A[:,1] + np.random.normal(0,1,N)/10
    return A
```

Cela peut se vérifier en calculant la corrélation.
Remarque 2 : Un élève a généré le nuage $X + 2\epsilon$ ce qui produit un nuage de points dont les deux variable sont moins corrélées. Voir à la fin pour plus de détail.

1.2 Q2 - matrice m1

On définit la matrice $M \in \mathbb{M}_{N,2}(\mathbb{R})$ définie par les deux vecteurs colonnes (X_i) et (Y_i). Choisir aléatoirement 20 valeurs dans cette matrice et les remplacer par `numpy.nan`. On obtient la matrice M_1.

[3]:
```
import random

def build_m1(mat, n=20):
    mat = mat.copy()
    positions = []
    while len(positions) < n:
        h = random.randint(0, mat.shape[0] * mat.shape[1] - 1)
        pos = h % mat.shape[0], h // mat.shape[0]
        if pos in positions:
            # La position est déjà tirée.
            continue
        positions.append(pos)
        mat[pos] = numpy.nan
    return mat, positions

m1, positions = build_m1(mat)
p = positions[0][0]
m1[max(p-2, 0):min(p+3, mat.shape[0])]
```

[3]:
```
array([[-1.48750338, -4.92138266],
       [-0.59978536, -2.22258934],
       [ 1.72143302,         nan],
       [ 1.02229479,  1.52222862],
       [-0.1157862 ,  1.97598417]])
```

Remarque 1: l'énoncé ne précisait pas s'il fallait choisir les valeurs aléatoires sur une ou deux colonnes, le faire sur une seule colonne est sans doute plus rapide et ne change rien aux conclusions des dernières questions.
Remarque 2: il ne faut pas oublier de copier la matrice `mat.copy()`, dans le cas contraire, la fonction modifie la matrice originale. Ce n'est pas nécessairement un problème excepté pour les dernières questions qui requiert de garder cette matrice.
Remarque 3: l'énoncé ne précisait pas avec ou sans remise. L'implémentation précédente le fait sans remise.

1.3 Q3 - moyenne

Calculer $\mathbb{E}X = \frac{1}{N}\sum_i^N X_i$ et $\mathbb{E}Y = \frac{1}{N}\sum_i^N Y_i$. Comme on ne tient pas compte des valeurs manquantes, les moyennes calculées se font avec moins de N termes. Si on définit V_x et V_y l'ensemble des valeurs non manquantes, on veut calculer $\mathbb{E}X = \frac{\sum_{i \in V_x} X_i}{\sum_{i \in V_x} 1}$ et $\mathbb{E}Y = \frac{\sum_{i \in V_y} Y_i}{\sum_{i \in V_y} 1}$.

[4]:
```
def mean_no_nan(mat):
    res = []
    for i in range(mat.shape[1]):
        ex = numpy.mean(mat[~numpy.isnan(mat[:, i]), i])
        res.append(ex)
    return numpy.array(res)

mean_no_nan(m1)
```

[4]:
```
array([0.01928312, 0.09388639])
```

Remarque 1 : il était encore plus simple d'utiliser la fonction nanmean.
Remarque 2 : Il fallait diviser par le nombre de valeurs non nulles et non le nombre de lignes de la matrice.

1.4 Q4 - matrice m2

Remplacer les valeurs manquantes de la matrice M_1 par la moyenne de leurs colonnes respectives. On obtient la matrice M_2.

[5]:
```
def build_m2(mat):
    means = mean_no_nan(mat)
    m1 = mat.copy()
    for i in range(len(means)):
        m1[numpy.isnan(m1[:, i]), i] = means[i]
    return m1

m2 = build_m2(m1)
m2[max(p-2, 0):min(p+3, mat.shape[0])]
```

[5]:
```
array([[-1.48750338, -4.92138266],
       [-0.59978536, -2.22258934],
       [ 1.72143302,  0.09388639],
       [ 1.02229479,  1.52222862],
       [-0.1157862 ,  1.97598417]])
```

1.5 Q5 - x le plus proche

On considère le point de coordonnées (x, y), écrire une fonction qui retourne le point de la matrice M dont l'abscisse est la plus proche de x.

[6]:
```
def plus_proche(mat, x, col, colnan):
    mini = None
    for k in range(mat.shape[0]):
        if numpy.isnan(mat[k, col]) or numpy.isnan(mat[k, colnan]):
            continue
        d = abs(mat[k, col] - x)
        if mini is None or d < mini:
            mini = d
            best = k
    return best

plus_proche(m1, m1[10, 0], 0, 1)
```

[6]:
```
10
```

[7]:
```
def plus_proche_rapide(mat, x, col, colnan):
    mini = None
    na = numpy.arange(0, mat.shape[0])[~(numpy.isnan(mat[:, col]) | numpy.isnan(mat[:,␣
 ↪colnan]))]
    diff = numpy.abs(mat[na, col] - x)
    amin = numpy.argmin(diff)
    best = na[amin]
    return best

plus_proche_rapide(m1, m1[10, 0], 0, 1)
```

[7]:
```
10
```

[8]:
```
%timeit plus_proche(m1, m1[10, 0], 0, 1)
```

```
4.12 ms ± 399 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[9]:
```
%timeit plus_proche_rapide(m1, m1[10, 0], 0, 1)
```

```
33.1 µs ± 2.26 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

C'est beaucoup plus rapide car on utilise les fonctions *numpy*.

1.6 Q6 - matrice m3

Pour chaque y manquant, on utilise la fonction précédente pour retourner le point dont l'abscisse et la plus proche et on remplace l'ordonnée y par celle du point trouvé. On fait de même avec les x manquant. On construit la matrice ainsi M_3 à partir de M_1.

[10]:
```
def build_m3(mat):
    mat = mat.copy()
    for i in range(mat.shape[0]):
        for j in range(mat.shape[1]):
            if numpy.isnan(mat[i, j]):
                col = 1-j
                if numpy.isnan(mat[i, col]):
                    # deux valeurs nan, on utilise la moyenne
                    mat[i, j] = numpy.mean(mat[~numpy.isnan(mat[:,j]), j])
                else:
                    pos = plus_proche_rapide(mat, mat[i, col], col, j)
                    mat[i, j] = mat[pos, j]
    return mat

m3 = build_m3(m1)
m3[max(p-2, 0):min(p+3, mat.shape[0])]
```

[10]:
```
array([[-1.48750338, -4.92138266],
       [-0.59978536, -2.22258934],
       [ 1.72143302,  2.83806507],
       [ 1.02229479,  1.52222862],
       [-0.1157862 ,  1.97598417]])
```

1.7 Q7 - norme

On a deux méthodes pour compléter les valeurs manquantes, quelle est la meilleure ? Il faut vérifier numériquement en comparant $\| M - M_2 \|^2$ et $\| M - M_3 \|^2$.

[11]:
```
def distance(m1, m2):
    d = m1.ravel() - m2.ravel()
    return d @ d

d2 = distance(mat, m2)
d3 = distance(mat, m3)
d2, d3
```

[11]:
```
(93.88020645836853, 10.054794671768933)
```

Remarque : Un élève a répondu :

```
On obtient (norme(M-M2))^2 = 98.9707 et (norme(M-M3))^2 = 98.2287 : la meilleure méthode semble être la
```

La différence n'est significative et cela suggère une erreur de calcul. Cela doit mettre la puce à l'oreille.

1.8 Q8 - répétition

Une experience réussie ne veut pas dire que cela fonctionne. Recommencer 10 fois en changeant le nuages de points et les valeurs manquantes ajoutées.

[12]:
```
def repetition(N=1000, n=20, nb=10):
    res = []
    for i in range(nb):
        mat = random_mat(N)
        m1, _ = build_m1(mat, n)
        m2 = build_m2(m1)
        m3 = build_m3(m1)
        d2, d3 = distance(mat, m2), distance(mat, m3)
        res.append((d2, d3))
    return numpy.array(res)

repetition()
```

[12]:
```
array([[38.93113166, 13.65407502],
       [44.59161999, 31.20763444],
       [56.36123306, 39.49474066],
       [40.20767715, 15.72341549],
       [86.99591576, 36.28602503],
       [81.35006845, 12.18103292],
       [85.775306  , 37.15330721],
       [79.44248685, 22.80699951],
       [52.70774305, 21.74452936],
       [83.59144759, 15.22093401]])
```

1.9 Q9 - plus de valeurs manquantes

Et si on augmente le nombre de valeurs manquantes, l'écart se creuse-t-il ou se réduit -il ? Montrez-le numériquement.

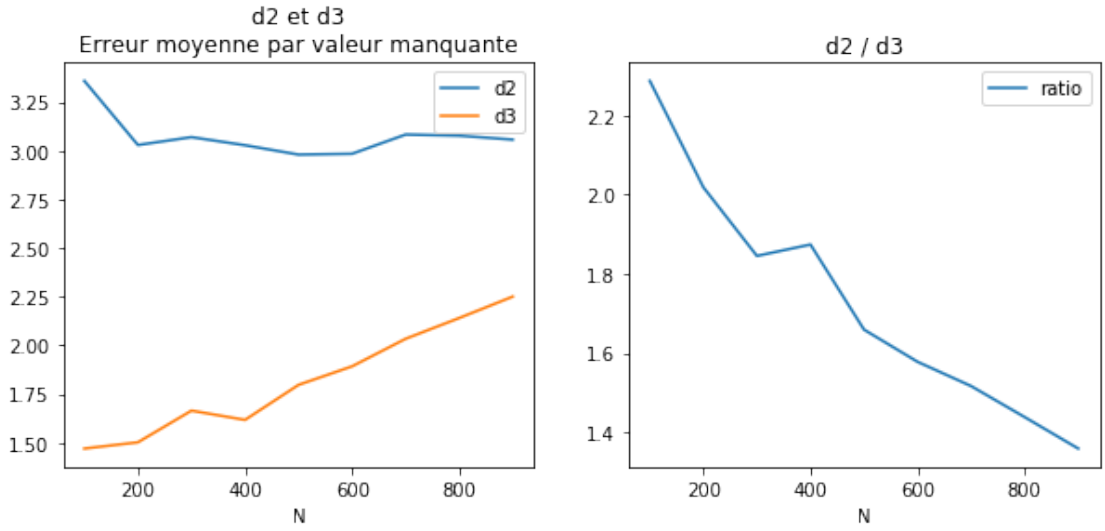
[13]:
```
diff = []
ns = []
for n in range(100, 1000, 100):
    print(n)
    res = repetition(n=n, nb=10)
    diff.append(res.mean(axis=0) / n)
    ns.append(n)
diff = numpy.array(diff)
diff[:5]
```

```
100
200
300
400
500
600
700
800
900
```

[13]:
```
array([[3.35913762, 1.46902292],
       [3.02940671, 1.50112628],
       [3.06988804, 1.66400287],
       [3.02826212, 1.6163169 ],
       [2.98007237, 1.7964768 ]])
```

[14]:
```
%matplotlib inline
```

[15]:
```
import pandas
df = pandas.DataFrame(diff, columns=["d2", "d3"])
df['N'] = ns
df = df.set_index('N')
df["ratio"] = df["d2"] / df["d3"]

import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 2, figsize=(10, 4))
df[["d2", "d3"]].plot(ax=ax[0])
df[["ratio"]].plot(ax=ax[1])
ax[0].set_title("d2 et d3\nErreur moyenne par valeur manquante")
ax[1].set_title("d2 / d3");
```



Plus il y a de valeurs manquantes, plus le ratio tend vers 1 car il y a moins d'informations pour compléter les valeurs manquantes autrement que par la moyenne. Il y a aussi plus souvent des couples de valeurs manquantes qui ne peuvent être remplacés que par la moyenne.

1.10 Q10

Votre fonction de la question 5 a probablement un coût linéaire. Il est probablement possible de faire mieux, si oui, il faut préciser comment et ce que cela implique sur les données. Il ne faut pas l'implémenter.
Il suffit de trier le tableau et d'utiliser une recherche dichotomique. Le coût du tri est négligeable par rapport au nombre de fois que la fonction `plus_proche` est utilisée.

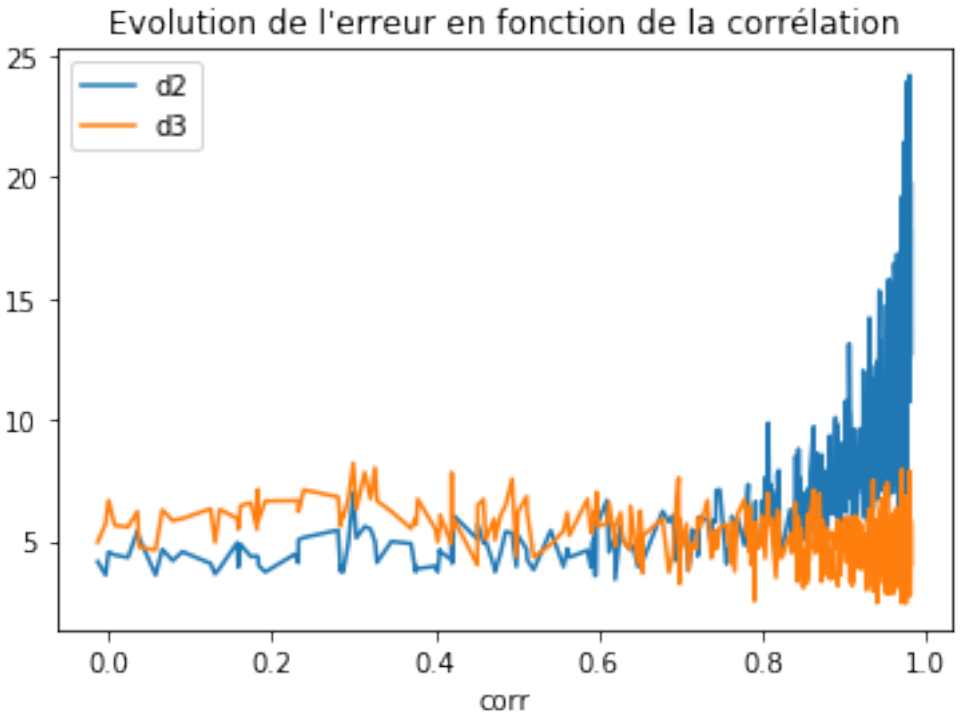
[16]:
```
def random_mat(N, alpha):
    mat = numpy.zeros((N, 2))
    mat[:, 0] = rnd.normal(size=(N,))
    mat[:, 1] = mat[:, 0] * alpha + rnd.normal(size=(N,))
    return mat

rows = []
for alpha in [0.01 * h for h in range(0, 500)]:
    m = random_mat(1000, alpha)
    m1, _ = build_m1(m, 20)
    m2 = build_m2(m1)
    m3 = build_m3(m1)
    d2, d3 = distance(m, m2), distance(m, m3)
    cc = numpy.corrcoef(m.T)[0, 1]
    rows.append(dict(corr=cc, d2=d2**0.5, d3=d3**0.5))

df = pandas.DataFrame(rows)
df.tail()
```

[16]:

	corr	d2	d3
495	0.978472	14.787724	5.693286
496	0.980944	17.399139	3.579552
497	0.979960	16.064428	7.893382
498	0.977117	15.492200	4.140280

```
499  0.981207  17.797778  2.785862
```

[17]:
```
ax = df.sort_values("corr").plot(x="corr", y=["d2", "d3"])
ax.set_title("Evolution de l'erreur en fonction de la corrélation");
```



On voit que la second méthode est meilleure si la corrélation est supérieur à 0.7. Plutôt moins bonne avant.

[18]:
```
```