Pesée des fraises

INFORMATIQUE COMMUNE - Devoir n° 3 - Olivier Reynet

Consignes :

1. utiliser une copie différente pour chaque partie du sujet,
2. écrire son nom sur chaque copie,
3. écrire de manière lisible et intelligible,
4. préparer une réponse au brouillon avant de la reporter sur la feuille.

A Pesée

On souhaite modéliser l'usage d'une balance de Roberval et d'une boîte de poids $B = (b_1, b_2, \ldots, b_n)$ composée de n poids. L'unité des poids est le gramme. Chaque poids de la boîte est unique.

On dispose d'une liste de 13 poids représentée par une liste Python `B = [1000, 500, 200, 100, 100, 50, 20, 10, 10, 5, 2, 1, 1]`.

A1. Existe-t-il une limite au poids des objets que l'on peut peser ? Pourquoi ? Si elle existe, donner des instructions en Python qui permettent de calculer cette limite.

> **Solution :** La boîte dispose d'un nombre fini de poids. Donc, on ne peut pas peser un objet plus lourd que la somme des poids, c'est-à-dire $1999g$. On ne peut pas non plus peser moins que le plus petit poids.

```python
limit = 0
for b in B:
    limit += b
```

A2. Proposer une approche gloutonne pour trouver une liste de poids permettant de peser un objet de P g.

> **Solution :** L'idée est de prendre les poids les plus lourds en premier. Comme la liste est triée ainsi, il suffit donc de parcourir la liste des poids et de prendre à chaque fois le poids si c'est possible, c'est-à-dire si le poids est inférieur l'objectif qui reste à atteindre.

A3. Écrire une fonction de prototype `glouton(B : list[int],P : int)` qui implémente cette approche gloutonne. La fonction renvoie la liste des poids utilisés pour atteindre le poids `P` et `None` si ce n'est pas possible.

Solution :

```python
def pglouton(B, P):
    solution = []
    nb_p = 0
    i = 0
    while P >= 0 and i < len(B):
        if B[i] <= P:
            P = P - B[i]
            nb_p += 1
            solution.append(B[i])
        i += 1
    if P == 0:
        return nb_p, solution
    else:
        return None, None
```

L'approche gloutonne est optimale, c'est-à-dire que lorsqu'elle fournit une solution, le nombre de poids utilisé est minimum. Mais on propose de le vérifier en pratique en calculant l'optimal avec une approche en programmation dynamique. Soit $S(i, j)$ le nombre de pièces minimum qu'il est nécessaire d'utiliser pour atteindre le poids j avec i poids de la boîte $B = (b_1, b_2, \dots, b_n)$. Le principe de Bellman permet d'écrire que :

$$S(i,j) = \begin{cases} 0 & \text{si } j = 0 \\ +\infty & \text{si } j > 0 \text{ et } i = 0 \\ S(i-1, j) & \text{si } b_i > j \\ \min\left(1 + S(i-1, j-b_i), S(i-1, j)\right) & \text{sinon} \end{cases} \tag{1}$$

A4. Justifier la valeur de $S(i, j)$ pour chaque cas.

Solution :

1. si $j = 0$, il existe une solution triviale pour atteindre 0 g qui est de ne prendre aucun poids.
2. si $i = 0$ et $j > 0$, alors il n'existe pas de solution pour atteindre j car on ne dispose d'aucun poids. L'infini représente cette impossibilité.
3. si le poids b_i est plus lourd que l'objet à peser, alors on ne peut pas le prendre. La solution optimale est donc la solution sans prendre cet objet.
4. si le poids b_i est plus petit que j, alors on peut le prendre. La solution optimale est le minimum de celle pour laquelle on a choisi de prendre b_i et de celle sans prendre ce poids. A priori, on ne sait pas laquelle, donc on prend le minimum des deux.

A5. Écrire une fonction de signature `nb_poids(B,P)` qui renvoie le nombre minimal de poids qu'il faut utiliser pour atteindre le poids `P`. Cette fonction procède de manière itérative, en complétant un tableau. On pourra utiliser l'objet `inf` de la bibliothèque `math`, pourvu que l'importation soit correcte.

Solution :

```
def dyn_kp(B, P):  # Programmation dynamique
    n = len(B)
    S = [[math.inf for _ in range(P + 1)] for _ in range(n + 1)]
    for i in range(n + 1):
        for j in range(P + 1):
            if i == 0 and j > 0:
                S[0][j] = math.inf
            elif j == 0:
                S[i][0] = 0
            elif B[i - 1] > P:
                S[i][j] = S[i - 1][j]
            else:
                S[i][j] = min(1 + S[i - 1][j - B[i - 1]], S[i - 1][j])
    return S[n][P]
```

A6. Quelle est la complexité de la fonction `nb_poids` ? Est-ce plus intéressant que l'approche gloutonne ?

Solution : La complexité de `nb_poids` est $O(nP)$ car les instructions du corps de la boucle interne sont toutes de complexité constante et qu'on opère deux boucles imbriquées à n (nombre de pois) et P itérations. L'approche gloutonne est linéaire par rapport au nombre de poids n. Elle est donc plus intéressante car optimale également.

A7. Expliquer pourquoi la mémoïsation est nécessaire pour écrire la fonction précédente récursivement.

Solution : Sans la mémoïsation, la fonction présenterait une complexité exponentielle du fait des appels récursifs multiples liés aux chevauchements des sous-problèmes. La mémoïsation permet de stocker les résultats déjà calculés et de garantir une complexité quasi-identique à celle de la version impérative.

A8. Écrire une fonction de signature `rec_nb_poids(B,P, memo)` qui renvoie le nombre minimal de poids qu'il faut utilisé pour atteindre le poids `P`. Cette fonction procède de manière récursive en utilisant la mémoïsation. Vous choisirez une structure de données adaptée à la mémoïsation pour le paramètre `memo` en expliquant pourquoi celle-ci est adaptée.

Solution :

```
def rec_nb_poids(B, P, memo):
    n = len(B)
    if (n, P) in memo:
        return memo[(n, P)]
    elif n == 0 and P > 0:
        return math.inf
    elif P == 0:
        return 0
    elif B[0] > P:
```

```python
            memo[(n, P)] = memo_kp(B[1:], P, memo)
            return memo[(n, P)]
        else:
            memo[(n, P)] = min(1 + memo_kp(B[1:], P - B[0], memo), memo_kp(B[1:], P, memo))
            return memo[(n, P)]
```

On dit qu'un système de poids est couvrant s'il permet d'atteindre tous les poids possibles de 0 à un entier maximum propre au système.

A9. Écrire une fonction de signature `couvrant(B)` qui renvoie `True` si le système de poids `B` est couvrant et `False sinon`.

Solution :

```python
def couvrant(B):
    Smax = 0
    for b in B:
        Smax += b
    for p in range(1, Smax + 1):
        solution = dyn_kp(B, p)
        if solution == math.inf :
            return False
    return True
```

A10. Quelle est la complexité de la fonction `couvrant` dans le pire et le meilleur des cas ?

Solution : Cette fonction est :

- dans le pire des cas, c'est-à-dire lorsque le système est couvrant, en $O(nPS_{max})$
- dans le meilleur des cas, c'est-à-dire lorsque le système est détecté non couvrant dès la première itération ($p = 1$), en $O(n)$.

A11. On dispose d'une boîte de poids B_2 constituée comme suit :

$$B_2 = (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024)$$

Ce système de poids est-il couvrant ? Y-a-t-il un avantage à utiliser cette boîte plutôt que B ?

Solution : La décomposition des nombres dans une base est unique. En base 2, cela signifie que quelque soit l'entier P considéré, il existe des $(a_0, a_1, \dots, a_n) \in [\![0, 1]\!]$ uniques tels que :

$$P = a_0 2^0 + a_1 2^1 + a_2 2^2 + \dots + a_n 2^n$$

Ce qui signifie qu'avec un seul exemplaire de chaque poids de B_2 de type 2^k grammes, il existe une solution pour peser P, tant que P est inférieur ou égal à la somme des poids de la boîte, soit dans le cas de B_2 : 2047 g.

B_2 est intéressant car il ne comporte que 11 poids au lieu de 13 poids pour B. Il sera probablement moins pratique pour les personnes ne sachant pas compter en base 2... Un autre défaut est que, même si elle comporte moins de poids, elle est légèrement plus lourde que B !

B Usinage, usure et algorithmes

Les circuits électroniques nécessitent une réalisation précise. On peut créer des pistes de cuivre en détourant avec une fraiseuse numérique. Cependant, il est impossible de réaliser une largeur de piste absolument conforme à son plan et le concepteur de la pièce doit préciser la tolérance, c'est-à-dire un écart aux dimensions nominales qui permet un fonctionnement correct du circuit. La tolérance nécessaire peut être de l'ordre du micron dans le domaine des circuits microondes. Une fraise qui présente une certaine usure ne sera plus capable de garantir une telle tolérance.

On dispose d'une machine outil capable de mesurer la perte de masse des fraises et de mesurer ainsi leur usure : plus une fraise est légère, plus elle est usée. Dans le but de faire de la maintenance prédictive, on souhaite classer les fraises en cours d'utilisation en trois groupes repérés par 0, 1 ou 2, selon la perte de masse.

B1. Expliquer, en détaillant son fonctionnement, en quoi l'algorithme des K-moyennes peut-être utile dans l'objectif de créer ces trois groupes ?

Solution : cf. cours

On dispose d'une liste des pertes de masse de 11 fraises :

`P = [3,50,7,17,21,72,43,64,35,9,20]`

Les fraises sont identifiées par leur position dans la liste. L'unité u est une unité de masse.

B2. En opérant à la main à partir de la partition initiale `[[6, 4, 8], [10, 5, 7, 3, 9], [0, 1, 2]]`, et en s'aidant de schémas, calculer la partition qui résulte de l'algorithme des 3-moyennes sur cette liste. La partition initiale est constituée par les indices des éléments dans la liste `P`.

Solution :

```
#0 [[6, 4, 8], [10, 5, 7, 3, 9], [0, 1, 2]]
#1 [[8], [1, 3, 5, 6, 7], [0, 2, 4, 9, 10]]
#2 [[6, 8], [1, 3, 5, 7], [0, 2, 4, 9, 10]]
#3 [[6, 8], [1, 3, 5, 7], [0, 2, 4, 9, 10]]
pertes = [[[43], [35]], [[50], [57], [72], [64]], [[3], [7], [21], [9], [20]]]
```

B3. Proposer un adjectif qui caractérise la principale différence entre l'approche de l'algorithme des K-moyennes et celle de l'algorithme des K plus proches voisins.

Solution : supervisé/ non supervisé

Pour utiliser l'algorithme des K plus proches voisins, il est nécessaire de disposer d'un jeu de données étiquetées, c'est-à-dire des données dont on connaît la classe. Cela n'est pas nécessaire pour K-moyennes.

B4. En appliquant l'algorithme des 3 plus proches voisins sur les données étiquetées :

```
E=[([3],0),([50], 2),([7],0),([57],2),([21],0),([72],2),([43],1),([64],2),([35],1),([9],0),([20],0)
,([13],0)]
```

déterminer le groupe auquel appartient la fraise dont la perte de masse est $29u$. On choisit un nombre de voisins $k = \sqrt{\frac{n}{N}}$, si n est le nombre d'échantillons étiquetés et N le nombre de classes. En cas d'égalité sur la classe majoritaire, on choisit de prendre classe dont l'indice est le plus faible.

Solution : On trouve que k vaut 2, les deux voisins de X sont `[4,8]` soit `[21], [35]` et que la classe de X est 0. On peut remarquer que X est cependant plus près de 35 que de 21 !

C Changement de dimension

Les fournisseurs du tour numérique parviennent à produire d'autres indicateurs du degré d'usure de la pièce. On dispose maintenant :

- d'une estimation de la largeur maximale de la fraise,
- du nombre de tours qu'elle a déjà effectués.

Au fur et à mesure, la production a réussi à produire un ensemble de données étiquetées en demandant aux techniciens d'évaluer eux-mêmes le groupe dans lequel devrait se trouver une fraise à l'aide d'autres contrôles. On dispose donc d'une liste `E` de n tuples `(p_1, p_2, p_3, c)` dans laquelle chaque tuple représente la perte de masse, la largeur maximale, le nombre de tours et le groupe de l'échantillon. On cherche à appliquer l'algorithme des K plus proches voisins à ces données.

C5. Écrire une fonction de signature `de(x: list[int],y: list[int])->float` qui renvoie la distance euclidienne entre deux points x et y d'un espace de dimension d. On s'assurera que les listes `x` et `y`, qui représentent les points de l'espace x et y ont bien la même dimension.

Solution :

```
def de(A: list[float], B: list[float]) -> float:
    assert len(A) == len(B)
    n = len(A)
    d = 0
    for i in range(n):
        d += (A[i] - B[i]) ** 2
    return d**(1/2)
```

On dispose du code à trous suivant qui implémente le tri fusion. Ce tri permet de trier des listes du type `E=[(i0,d0), (i1,d1),..., (in,dn)]` d'après le second élément du tuple `di`.

```
def partager_en_deux(t):
    n = len(t)
    t1, t2 = [], []
    for i in range(n // 2):
        t1.append(t[i])
    for i in range(n // 2, n):
        t2.append(t[i])
    return t1, t2

def fusion(t1, t2):
    n1, n2 = len(t1), len(t2)
    if n1 == 0:
        return ...   # 1 à compléter
    elif n2 == 0:
        return ...  # 2 à compléter
    elif t1[0][1] < t2[0][1]:
        return [t1[0]] + fusion(t1[1:], t2)
```

```
    else:
        return ...  # 3 à compléter

def tri_fusion(t):
    if len(t) < 2:
        return t
    else:
        t1, t2 = partager_en_deux(t)
        return ...  # 4 à compléter
```

C6. En faisant attention à utiliser une formulation récursive, compléter les instructions `return ...` de manière idoine.

Solution :

```
return t2
return t1
return [t2[0]] + fusion(t1, t2[1:])
fusion(tri_fusion(t1), tri_fusion(t2))
```

C7. Donner, sans la justifier, la complexité du tri fusion.

Solution : $O(n \log n)$

C8. Tel qu'il est implémenté, ce tri fusion est-il en place ?

Solution : Non, car la fonction fusion construit et renvoie un autre tableau trié.

C9. Écrire une fonction de signature `k_plus_proches(k : int, E : list[list[int]], X : list[int],)-> list[int]` qui renvoie la liste des indices dans `E` des k plus proches voisins de `X` au sens de la distance euclidienne.

Solution :

```
def k_plus_proches(k, E, X):
    assert len(E) > 0
    assert len(X) == len(E[0][:-1])
    n = len(E)
    D = [(i, de(X, E[i][0])) for i in range(n)]  # on enlève la classe
    # tri de D en conformité à distance croissante
    T = tri_fusion(D)
    V = [v for v,d in T] # on ne garde que l'indice du voisin
    return V[:k]
```

C10. Écrire une fonction de signature `classe_maj(V, E, N)` qui renvoie la classe majoritaire des éléments indexés par `V`, les voisins de `X`. Cette classe sera choisie à la majorité relative. En cas d'égalité on choisit la classe la plus petite. Le paramètre `N` est le nombre de classes.

Solution :

```
def classe_majoritaire(V: list[int], E: list[int], N: int) -> int:
    h = [0 for _ in range(N)]  # histogramme des classes
    for v in V:
        h[E[v][1]] += 1
    cmaj = 0
    vmax = h[0]
    for i in range(len(h)):  # calcul de la classe majoritaire
        if h[i] > vmax:
            vmax = h[i]
            cmaj = i
    return cmaj
```

La méthode de la validation croisée permet déterminer le k optimal de l'algorithme des k plus proches voisins, c'est-à-dire le nombre de voisins qui maximise le taux prédiction de l'algorithme. Elle consiste à tester tous les k sur un jeu de données divisé en b blocs que l'on considère alternativement comme des données de tests ou des données de vérification. Par exemple, pour $b = 4$, on a :

Valeur de k	Entrainement	Validation croisée	Précision
1	D1, D2, D3	D4	p14
1	D1, D2, D4	D3	p13
1	D1, D3, D4	D2	p12
1	D2, D3, D4	D1	p11
2	D1, D2, D3	D4	p24
2	D1, D2, D4	D3	p23
2	D1, D3, D4	D2	p22
2	D2, D3, D4	D1	p21
...	...	...	...

La moyenne des précisions obtenues pour chaque k permet par la suite de choisir la valeur optimale. Par exemple, dans le cas suivant, on choisira $k = 3$:

k	Précision moyenne
1	0,78
2	0,82
3	0,9
4	0,85
5	0,8
...	...

C11. Écrire une fonction de signature `blocks(E, b)` qui divise les échantillons en b blocs. Cette fonction renvoie donc une liste comportant b sous-listes. Les $b-1$ premières sous-listes sont de taille égale à $\lfloor n/b \rfloor$. La dernière sous liste comporte le reste des éléments de `E`.

Solution :

```
def blocs(E, b):
    n = len(E)
    B = [[] for _ in range(b)]
    c = 0
    for i in range(b - 1):
        for k in range(n // b + 1):
            B[i].append(E[k + i * (n // b)])
            c += 1
    while c < len(E):
        B[b - 1].append(E[c])
        c += 1
    assert c == len(E)
    return B
```

On dispose d'une fonction de signature `knn(train,test)` qui renvoie la précision de la prédiction des classes de `test` d'après les données d'entrainement `train` et les classes vraies de `test`.

C12. Écrire une fonction de signature `validation_croisee(E, b, k_max)` qui renvoie le nombre de voisins optimal inférieur ou égal à k_{max} pour les données d'entrées.

Solution :

```
def validation_croisee(E, b, k_max):
    B = blocs(E, b)
    k_opt = 0
    p_opt = 0
    for k in range(1, k_max + 1):
        p = 0
        for j in range(b):
            tests = B[j]
            tr = []
            for i in range(b):
                if i != j:
                    for h in range(len(B[i])):
                        tr.append(B[i][h])
            p += knn(tr, tests)
        p += p / b  # moyenne des précisions pour k
        if p_opt < p:
            p_opt = p
            k_opt = k
    return k_opt
```

D SQL forever

On dispose de la base de données décrite par la figure 1. Cette base de données comporte les éléments suivants :

1. la table fraise qui représente les fraises utilisées :
    - fraise_id : l'identifiant de la fraise, clef primaire
    - dur : la dureté de la fraise en N/m^2

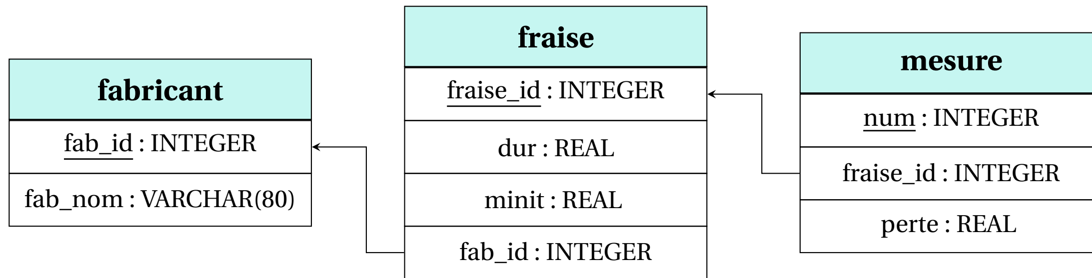

FIGURE 1 – Modèle physique de la base de données

- minit : la masse initiale de la fraise en u
- fab_id : l'identifiant du fabricant de la fraise, clé étrangère

2. la table fabricant :
   - fab_id : l'identifiant du fabricant, clef primaire,
   - fab_nom : le nom du fabricant
3. la table mesure :
   - num : le numéro de la mesure de perte de masse
   - fraise_id : l'identifiant de la fraise mesurée, clé étrangère
   - perte : la perte de masse de la fraise en %

D1. En utilisant des requêtes SQL, trouver les informations suivantes :

(a) Le nombre de fraises du fabricant dont l'identifiant est 42.

Solution :

```
SELECT COUNT(fraise)
FROM fraise
WHERE fab_id = 42
```

(b) Les noms des fabricants dont la dureté des fraises dépasse strictement les 250 N/m^2 dans l'ordre alphabétique. Il n'y aura pas de résultats redondants.

Solution :

```
SELECT DISTINCT(fab_nom)
FROM fabricant
JOIN fraise ON fabricant.fab_id = fraise.fab_id
WHERE dur > 250
ORDER BY fab_nom
```

(c) Les identifiants, la masse initiale des fraises et leur perte de masse si les fraises ont perdu strictement plus de 10% de leur masse. Les résultats sont présentés par ordre décroissant de perte de masse et on limitera le nombre de résultat à 5.

Solution :

```
SELECT fraise.fraise_id, minit, perte
FROM fraise
JOIN mesure ON mesure.fraise_id = fraise.fraise_id
WHERE perte > 10
ORDER perte DESC
LIMIT 3
```

(d) Le nombre de fraises, la moyenne de leur perte de masse et le nom du fabricant de ces fraises si la moyenne de la perte de masse est strictement inférieure à 5%.

Solution :

```
SELECT COUNT(fraise.fraise_id), AVG(perte), fab_nom
FROM fraise
JOIN fabricant ON fraise.fab_id = fabricant.fab_id
JOIN mesure ON fraise.fraise_id = mesure.fraise_id
GROUP BY fab_nom
HAVING AVG(perte) < 5
```