REPUBLIQUE TUNISIENNE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Concours Nationaux d'Entrée aux Cycles de Formation d'Ingénieurs
Session 2019

الجمهورية التونسية
وزارة التعليم العالي والبحث العلمي

المناظرات الوطنية للدخول
إلى مراحل تكوين المهندسين
دورة 2019

Concours Mathématiques et Physique, Physique et Chimie et Technologie
Epreuve d'Informatique

Date : Mardi 11 Juin 2019 **Heure** : 12 H **Durée** : 2 H **Nbr pages** : 10

Barème : **PROBLEME 1** : 14 points (**Partie 1** : 2 points ; **Partie 2** : 5 points ; **Partie 3** : 7 points)
PROBLEME 2 : 6 points (**Partie 1** : 1 point ; **Partie 2** : 2 points ; **Partie 3** : 3 points)

DOCUMENTS NON AUTORISES
L'USAGE DES CALCULATRICES EST INTERDIT
Il FAUT RESPECTER IMPERATIVEMENT LES NOTATIONS DE L'ENONCE
VOUS POUVEZ EVENTUELLEMENT UTILISER LES FONCTIONS PYTHON DECRITES A L'ANNEXE2 (PAGE 10)

Présentation Générale

L'apprentissage automatique supervisé permet d'élaborer des programmes capables d'apprendre automatiquement à partir d'un ensemble de données (**dataset**) comportant des valeurs d'observations et les décisions qui leur sont associées. Il a ainsi pour objectif de produire un modèle capable de prédire la décision à prendre pour des nouvelles valeurs d'observations.

Par exemple, on peut donner au programme d'apprentissage un ensemble de données contenant les observations relatives à des patients (**tension** artérielle, **âge** du patient et présence d'une **tachycardie** sinusoïdale) et expliquer lesquels ont un risque élevé de crise cardiaque (**décision** = 1) et lesquels ont un risque très faible (**décision** = 0). Comme illustré dans le tableau suivant, les lignes représentent les valeurs des observations relatives à un patient et la dernière colonne représente la décision finale.
Une fois l'apprentissage terminé, à partir des observations d'un nouveau patient, le programme, appelé aussi **classifieur**, devra déterminer automatiquement la décision à prendre (0 ou 1).

tension	âge	tachycardie	décision
110	50	1	**0**
119	36	0	**0**
82	72	0	**1**
81	70	0	**1**
56	50	1	**1**

Table 1 : Exemple de données d'apprentissage.

PROBLEME 1

L'objectif est d'implémenter un modèle qui permet de représenter des règles permettant de prédire la décision à partir des valeurs d'observations. Le modèle proposé est basé sur la structure d'arbre binaire.

Partie 1 : Représentation de la structure d'arbre binaire

Un **arbre binaire** est une structure de données formée par une hiérarchie d'éléments appelés **nœuds. Un nœud** est caractérisé par deux catégories d'informations :

- Les informations propres au nœud ;
- Les informations décrivant les liens avec ses nœuds descendants.

Un arbre binaire est toujours désigné par un **nœud :** son nœud initial appelé **racine**.
Chaque nœud possède **au plus** deux **nœuds** fils :

- Si le nœud possède exactement deux nœuds fils, ils sont appelés **fils gauche** et **fils droit**.
- Si le nœud possède un seul nœud fils, ce dernier est soit le fils gauche soit le fils droit.
- Si le nœud ne possède aucun nœud fils, il est appelé **feuille**.

Alors, un arbre binaire est une structure récursive, puisque le fils gauche et le fils droit sont eux-mêmes des nœuds (représentant des arbres à leur tour).
Une **branche** dans l'arbre est un chemin de la racine de l'arbre à une feuille.

Exemple

La figure 1 représente un arbre binaire dont le nœud **A** est la racine avec **B** son fils gauche et **C** son fils droit.

Le nœud **C** a un seul fils **F** (fils droit). **D**, **E** et **F** sont des nœuds feuilles.

[A,B,D], **[A,B,E]** et **[A,C,F]** sont les branches de l'arbre.

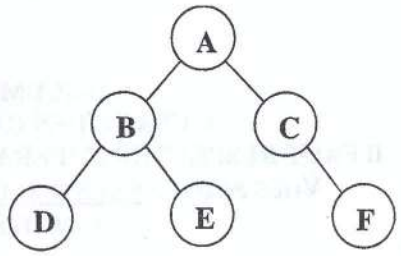

Figure 1 : Exemple d'un arbre binaire

Dans la suite, on propose de construire la classe **Node** dont le squelette est donné par :

```
class Node :

  def __init__ (self, val, leftNode = None, rightNode = None):
      self.label = val                          # chaine de caractère
      self.left = leftNode                      # instance de la classe Node ou None
      self.right= rightNode                     # instance de la classe Node ou None
  def isLeaf(self) :
     # à completer …
  def __repr__(self) :
        return self.label
  def  linearise(self) :   #méthode récursive retournant la liste des branches
      if self.isLeaf() :                        # traitement si le nœud est une feuille
         return [[self]]
      else :
         if self.left != None :
           L1 = self.left.linearise()  # traitement récursif du nœud fils gauche
         else :
           L1=[]
         if self.right != None :
           L2 = self.right.linearise() # traitement récursif du nœud fils droit
         else :
           L2=[]
      return [[self]+e for e in L1]+[[self]+e for e in L2]
   def __len__(self): # méthode récursive
     # à compléter …
   def __str__(self): # méthode récursive
     # à compléter …
```

Exemple

Nd est une instance de la classe **Node** correspondant à l'arbre de la figure 1.

```
Nd = Node('A', Node('B', Node('D'), Node('E') ) , Node('C', None, Node('F')))
```

Travail demandé

1. Ecrire la méthode **isLeaf** qui retourne `True` si le nœud est une feuille et `False` sinon.
2. Ecrire la méthode **__len__** qui permet de déterminer le nombre de nœuds d'un arbre.

   Exemple :
   ```
   >>>len(Nd)
   6
   ```
3. Ecrire la méthode **__str__** qui retourne la chaine représentant l'arbre conformément au format donné par l'exemple suivant.

   Exemple
   ```
   >>>print(Nd)
   Node('A',Node('B',Node('D'),Node('E')),Node('C',None,Node('F')))
   ```

Partie 2 : Représentation du modèle de décision

Les règles de décision peuvent être représentées par un arbre binaire, appelé arbre binaire de décision. Par exemple, pour les données d'apprentissage de la table 1, il est possible de construire l'arbre binaire de décision illustré par la figure 2.

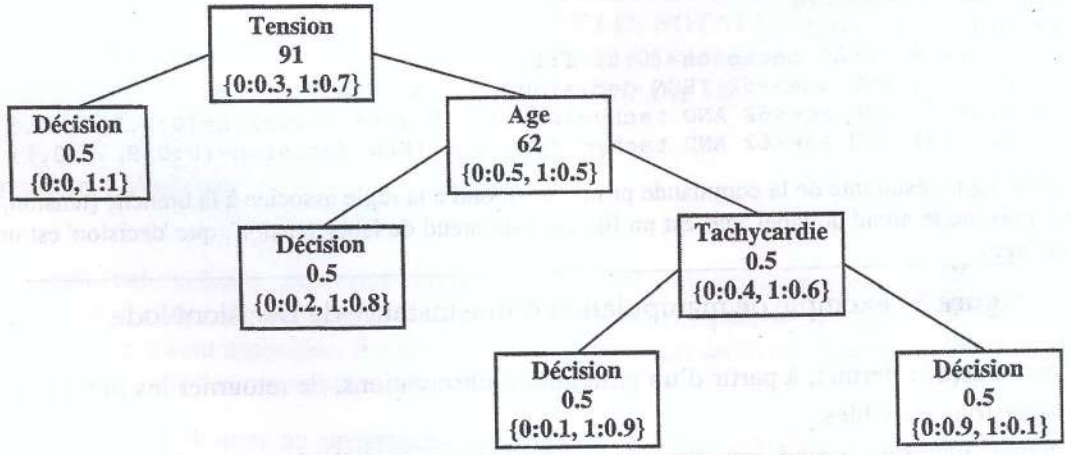

Figure 2 : Exemple d'un arbre de décision

Dans cette partie, on propose construire deux classes :

- **DecisionNode** : classe qui hérite de la classe **Node** permettant de représenter un arbre binaire de décision ;
- **DecisionForest** : classe qui représente un ensemble d'arbres, appelée forêt.

Description des classes

- Classe **DecisionNode** :
  - Attributs :
    - **label**, chaine de caractères, représentant l'observation, hérité de la classe **Node** ;
    - **distr**, un dictionnaire représentant la probabilité de chaque décision :
      - chaque clé représente une décision possible 0 ou 1 ;
      - chaque valeur est un réel représentant la probabilité de décision.
    - **seuil,** un réel, représentant le seuil de test utilisé pour déduire la branche à suivre ;
    - **left,** instance de la classe représentant le fils gauche, hérité de la classe **Node** ;
    - **right,** instance de la classe représentant le fils droit, hérité de la classe **Node** ;
  - Méthodes :
    - **__init__(...) :** permet l'initialisation des attributs de la classe **DecisionNode**, sachant que cette dernière hérite de la classe **Node**.

- **outcome(...) :** permet de retourner, à partir d'un réel **val** donné, le fils gauche si **val** est supérieur ou égal à la valeur du seuil et le fils droit sinon. Cette méthode doit afficher un message d'erreur dans le cas où le nœud courant est une feuille.
- **__str__(...) :** permet de retourner une chaine de caractères représentant les règles de décision extraites de l'arbre conformément au format donné dans la figure 3. Cette méthode doit appeler la méthode **linearise** héritée de la classe **Node**.

DecisionNd est une instance de la classe **DecisionNode** associée à la figure 2

```
>>>left = DecisionNode('decision', {0:0, 1:1})
>>>right= DecisionNode('age', {0:0.5, 1:0.5}, 62,
                DecisionNode('decision', {0:0.2, 1:0.8},0.5),
                DecisionNode('tachycardie',{0:0.4,1:0.6},0.5,
                      DecisionNode('decision', {0:0.1, 1:0.9},0.5),
                      DecisionNode('decision', {0:0.9, 1:0.1},0.5)) )

>>>DecisionNd = DecisionNode('tension', {0:0.3, 1:0.7}, 91, left, right)
```

La méthode linearise appliquée sur **DecisionNd** donne la liste des branches de l'arbre de la figure 2

```
>>>DecisionNd.linearise()
[[tension, decision],[tension, age, decision],[tension, age, tachycardie,
decision],[tension, age, tachycardie, decision]]
```

La fonction print (appel de la méthode **__str__**) permet d'afficher textuellement les règles associées aux branches de l'arbre **DecisionNd**

```
>>>print(DecisionNd)
IF tension>=91 THEN decision={0:0, 1:1}
IF tension<91 AND age>=62 THEN decision={0:0.2, 1:0.8}
IF tension<91 AND age<62 AND tachycardie>=0.5 THEN decision={0:0.1, 1:0.9}
IF tension<91 AND age<62 AND tachycardie<0.5 THEN decision={0:0.9, 1:0.1}
```

La deuxième ligne résultante de la commande print correspond à la règle associée à la branche [tension, age, decision], puisque le nœud de label 'age' est un fils droit du nœud de label 'tension', que 'decision' est un fils gauche de 'age'.

Figure 3 : Exemple de manipulation d'une instance de DecisionNode

- **predict(..) :** permet, à partir d'un ensemble d'observations, de retourner les prédictions des décisions possibles.

  Cette méthode prend en paramètre un dictionnaire **dicobs**, associé à un ensemble d'observations, où :
  - Chaque clé représente le nom d'une observation (chaine de caractères) ;
  - Chaque valeur représente la valeur d'une observation (entier).

  Cette méthode retourne un dictionnaire **distr**, en considérant la variable **CurrentNode** (nœud courant) initialisée à self et en appliquant le procédé suivant :
  - si le label de **CurrentNode** n'est pas une clé du dictionnaire **dicobs** où bien si **CurrentNode** est une feuille alors, retourner son attribut **distr**
  - sinon,
    - la variable **Curvalue** reçoit la valeur dont la clé est le label de **CurrentNode** dans **dicobs**;
    - **CurrentNode** reçoit le résultat de la méthode **outcome** à partir de l'instance **CurrentNode** en passant **Curvalue** comme paramètre.

- Classe **DecisionForest** :
  - Attribut : **listNodes** : une liste d'instances de la classe **DecisionNode**
  - Méthodes :
    - **__init__(...) :** permet d'initialiser l'attribut **listNodes** à une liste vide.
    - **add(...) :** permet d'ajouter une instance de la classe **DecisionNode** à l'attribut **listNodes**

- **predict(...) :** permet, à partir d'un ensemble d'observations représenté par un dictionnaire **dicobs**, de retourner un dictionnaire **distr** contenant la probabilité moyenne de chaque décision suivant les prédictions des éléments de **listNodes.**

Travail demandé

En se basant sur les descriptions ci-dessus répondre aux questions suivantes :
Construire la classe **DecisionNode** :

1. Donner l'entête qui permet la définition de la classe **DecisionNode**.
2. Ecrire la méthode **__init__**.
3. Ecrire la méthode **outcome**.
4. Ecrire la méthode **__str__**.
5. Ecrire la méthode **predict**.

Construire la classe **DecisionForest** :

6. Ecrire la méthode **__init__**.
7. Ecrire la méthode **add**.
8. Ecrire la méthode **predict**.

Partie 3 : Apprentissage (Les questions de 1 à 8 sont indépendantes des Parties 1 et 2)

L'objectif de cette partie est d'implémenter un algorithme pour la construction automatique de l'arbre de décision binaire à partir des données.
Les données d'apprentissage seront représentées par une matrice **DSET** de **n** lignes et **m** colonnes. Pour chaque ligne de la matrice, nous convenons d'associer les (**m**-1) premières colonnes pour les valeurs d'observations et la dernière pour la décision associée à ces observations.
Les noms des observations et le nom de la décision sont stockés dans une liste, notée **Lcol**, selon le même ordre dans la matrice.

Exemple : La matrice **DSET** et la liste **Lcol** associées aux données de la table 1 sont :

$$\mathbf{DSET} = \begin{pmatrix} 110 & 50 & 1 & 0 \\ 119 & 36 & 0 & 0 \\ 82 & 72 & 0 & 1 \\ 81 & 70 & 0 & 1 \\ 56 & 50 & 1 & 1 \end{pmatrix}$$

$$\mathbf{Lcol} = [\text{'tension'}, \text{'age'}, \text{'tachycardie'}, \text{'decision'}]$$

Travail demandé

Dans la suite :

- On suppose que **DSET** et **Lcol** sont déjà définies.
- On suppose que le module `numpy` a été importé ainsi : `import numpy as np`
- Les fonctions demandées seront écrites en langage Python en respectant la nomenclature présentée à l'Annexe 1 (page 9).

1. Ecrire une fonction nommée **CountValues** qui prend en paramètres **DSET** et un entier **ind** et retourne le nombre de valeurs distinctes de la colonne d'indice **ind**.
   Exemple : L'appel de CountValues(DSET,1) retourne 4.
2. Ecrire une fonction **EvalDistr** qui prend en paramètre **DSET** et retourne un dictionnaire **distr** où :
   - chaque clé, notée **d**, est une valeur de la décision (0 ou 1);
   - chaque valeur est la probabilité d'apparition de **d** dans **DSET** exprimée par :

$$p(decision = \mathbf{d} | \mathbf{DSET}) = \frac{nombre\ de\ lignes\ où\ la\ décision\ est\ égale\ à\ \mathbf{d}}{nombre\ de\ lignes\ de\ \mathbf{DSET}} \qquad (Eq.1)$$

   Si **DSET** est vide, **distr**= {0:0.5, 1:0.5}.

   Exemple : L'appel de **EvalDistr**(DSET) retourne {0: 0.4, 1: 0.6}.

3. Ecrire une fonction nommée **IsPure**, qui prend en paramètre **DSET** et retourne un booléen égal à `True` si **DSET** est pure et `False` sinon, sachant que **DSET** est pure, si et seulement si, toutes les valeurs des décisions sont égales.

4. Ecrire une fonction **IsQualitative** qui prend en entrée **DSET** et retourne une liste de booléens, **qual,** de taille égale au nombre de colonnes de **DSET**, contenant `True` pour les observations de valeurs qualitatives (0 ou 1) et `False` pour les observations de valeurs quantitatives (autres valeurs numériques).

   Exemple : L'appel **IsQualitative(DSET)** retourne la liste `[False, False, True, True]`.

5. Ecrire une fonction **Cut** qui permet de découper **DSET** en deux matrices. Elle prend en paramètres **DSET**, **Lcol**, **obs** (une chaine de caractères correspondant au nom d'une observation) et un réel **S** représentant le seuil de découpage et prenant la valeur 0.5 par défaut.

   Cette fonction retourne un tuple formé par les trois listes **L1**, **L2** et **L3** :
   - **L1** contient les noms des observations excepté **obs** ;
   - **L2** contient les matrices **DSET1** et **DSET2** telles que :
     - **DSET1** est formée par les lignes de **DSET** où la valeur de l'observation **obs**, notée **Vobs**, est supérieure ou égale à **S** ;
     - **DSET2** est formée par les autres lignes de **DSET** ;
     - La colonne **obs** de **DSET** ne doit pas figurér dans **DSET1** et **DSET2** ;
   - **L3** contient les probabilités p_1 et p_2 décrites par les équations (*Eq.*2) et (*Eq.*3) :

$$p_1 = p(\mathbf{Vobs} \geq \mathbf{S} \,|\, \mathbf{DSET}) = \frac{\textit{nombre de lignes de } \mathbf{DSET1}}{\textit{nombre de lignes de } \mathbf{DSET}} \qquad (Eq.2)$$

$$p_2 = p(\mathbf{Vobs} < \mathbf{S} \,|\, \mathbf{DSET}) = \frac{\textit{nombre de lignes de } \mathbf{DSET2}}{\textit{nombre de lignes de } \mathbf{DSET}} = 1 - p_1 \qquad (Eq.3)$$

   Exemple : L'appel **Cut**(DSET, Lcol, 'age', 70) retourne :

```
(['tension', 'tachycardie', 'decision'],
 [array([[82,0,1], [81,0,1]]), array ([[110,1,0],[119,0,0],[56,1,1]])],
 [0.4, 0.6])
```

6. Ecrire une fonction **Impurity** qui prend en paramètres **DSET**, **Lcol**, **obs** et un seuil **S** (ayant par défaut la valeur 0.5). Cette fonction retourne un réel mesurant la qualité du découpage de **DSET** en deux matrices **DSET1** et **DSET2**, calculé selon l'équation (*Eq.*4).

$$p_1 \times \min_{d \in \{0,1\}} (p(decision = \mathbf{d} \,|\, \mathbf{DSET1})) + p_2 \times \min_{d \in \{0,1\}} (p(decision = \mathbf{d} \,|\, \mathbf{DSET2})) \qquad (Eq.4)$$

   Où :
   - **DSET1** et **DSET2** résultent du découpage de la matrice **DSET** selon **obs** et **S** ;
   - p_1 et p_2 sont décrites par les équations (*Eq.*2) et (*Eq.*3) ;
   - $p(decision = \mathbf{d} | \mathbf{DSET})$ est donnée par l'équation (*Eq.*1).

   Exemple : L'impureté du découpage illustré dans l'exemple de la question 5 est donnée par :

$$\boldsymbol{Impurity(DSET, Lcol, 'age', 70) = 0.4 \times min(0,1) + 0.6 \times min\left(\frac{2}{3}, \frac{1}{3}\right) = 0.2}$$

7. Ecrire une fonction **SortObs** qui prend en paramètres **DSET**, **Lcol** et **obs**, puis retourne une liste **Lc** obtenue selon les étapes suivantes :
   - à partir des deux colonnes **obs** et 'decision', créer la liste **Lc** qui est une liste de tuples (**v**,**d**) où **v** est une valeur de l'observation **obs** et **d** la valeur de la décision associée,
   - puis, trier **Lc** par ordre croissant suivant les valeurs de **obs**.

   Exemple
   L'appel de SortObs(DSET, 'age') retourne : `[(36,0),(50,0),(50,1),(70,1),(72,1)]`

8. Ecrire une fonction **BestCut** qui prend en paramètres **DSET**, **Lcol** **obs** et une liste **qual** et retourne un tuple (**vBest**, **sBest**) où **vBest** est la meilleure impureté et **sBest** est le meilleur seuil de découpage associé. **vBest** et **sBest** sont déterminés comme suit :
   - Cas des observations qualitatives :
     - **sBest**= 0.5
     - **vBest= Impurity(DSET, Lcol, obs)**.
   - Cas des observations quantitatives :
     - Créer **Lc**, la liste formée par les tuples (v_i, d_i) triée par ordre croissant en utilisant la fonction **SortObs**.
     - Créer à partir de **Lc**, une liste **LSeuil** contenant les seuils possibles de découpage de la matrice **DSET** selon les conditions suivantes :
       - Si **DSET** est pure, alors **LSeuil** contient la valeur maximale des v_i de la liste **Lc**.
       - Si **DSET** est impure, alors **LSeuil** est formée par les valeurs $\frac{v_i + v_{i+1}}{2}$ pour tous les tuples adjacents (v_i, d_i) et (v_{i+1}, d_{i+1}) de **Lc** avec $\boldsymbol{d_i \neq d_{i+1}}$.
     - Déterminer le meilleur seuil **sBest** de la liste **LSeuil** associée à la valeur minimale de l'impureté, **vBest**.

9. Ecrire une fonction nommée **BuildTree** qui prend en paramètres **DSET**, une liste **Lcol** et une liste de booléens **qual** et retourne une instance de la classe **DecisionNode** (définie dans la partie 1) représentant l'arbre de décision, selon le procédé <u>récursif</u> suivant :

   - créer le dictionnaire **distr** en calculant la distribution des valeurs de décisions dans **DSET**.
   - **Traitement de base** : Si **DSET** est vide ou bien si **DSET** est pure ou bien si **Lcol** contient une seule valeur, alors retourner une instance de DecisionNode correspondant à un nœud feuille avec la distribution **distr** ;
   - **Traitement récursif (général) :**
     - pour chaque observation **obs** de **Lcol**, calculer (en utilisant la fonction **BestCut**) le meilleur seuil ainsi que l'impureté associée ;
     - trouver l'observation **oBest** qui a la meilleure impureté **vBest** (minimale) associée à son seuil **sBest** parmi toutes les observations **obs** dans **Lcol** ;
     - créer une instance de DecisionNode contenant le nom de l'observation **oBest** comme label et son seuil **sBest** et la distribution **distr** ;
     - appliquer le découpage de **DSET** en **DSET1** et **DSET2** selon **oBest** et **sBest** à l'aide de la fonction ***Cut*** ;
     - créer le nœud fils gauche (appel récursif avec **DSET1**, **Lcol** sauf **oBest**) ;
     - créer le nœud fils droit (appel récursif avec **DSET2**, **Lcol** sauf **oBest**).

PROBLEME 2

Soit la base de données relationnelle intitulée 'classifieurs.db' contenant la description des données d'apprentissage avec les classifieurs automatiques créés autour de ces données, représentée par le schéma relationnel suivant :

- **DataSet** (<u>ds_id</u>, ds_name, nb_instances, format, ds_description)
  La table DataSet décrit les données d'apprentissage, avec les colonnes :
  - ds_id : identifiant du dataset (entier), *clé primaire*.
  - ds_name : nom du dataset (chaîne de caractères).
  - nb_instances : nombre de lignes du dataset (entier).
  - format : le format des données du dataset (chaine de caractères).
  - ds_description : le sommaire du dataset (chaine de caractères).

- **Classifieur** (cls_id, cls_description, error_rate, language, ds_id)
  La table Classifieur décrit un classifieur automatique construit à partir d'un dataset, avec les colonnes :
  - cls_id : identifiant du classifieur (chaine de caractère), *clé primaire*.
  - cls_description : description du classifieur (chaine de caractères).
  - error_rate : un nombre réel entre 0 et 1 qui décrit le pourcentage des données où les décisions prédites par le classifieur sont différentes de la réalité.
  - language : nom du langage de programmation utilisé pour implémenter le classifieur (chaine de caractère).
  - ds_id : identifiant du dataset utilisé pour l'apprentissage du classifieur, *clé étrangère*.
- **Method** (m_name, category, m_description)
  La table Method décrit les méthodes utilisées dans le domaine de l'apprentissage automatique pour la construction des classifieurs, avec les colonnes :
  - m_name : le nom de la méthode (chaine de caractères), *clé primaire*.
  - category : la catégorie de la méthode (chaine de caractères).
  - m_description : la description de la méthode (chaine de caractères).
- **Combine** (cls_id, m_name, description)
  La table Combine décrit les méthodes de classification utilisées dans chaque classifieur, de clé primaire (cls_id, m_name), avec les colonnes :
  - cls_id : identifiant du classifieur (chaine de caractère), *clé étrangère*.
  - m_name : le nom de la méthode (chaine de caractère), *clé étrangère*.
  - description : stratégie d'intégration de la méthode dans le classifieur (chaine de caractères)

Partie 1 : algèbre relationnelle

Exprimer en algèbre relationnelle les requêtes suivantes :

1. Déterminer les identifiants, les noms et les descriptions des datasets au format 'csv'.
2. Déterminer les descriptions des classifieurs implémentés en 'Python' et qui utilisent la méthode de catégorie 'KNN'.

Partie 2 : SQL

Exprimer en SQL les requêtes suivantes :

3. Donner les identifiants des datasets pour lesquels il existe au moins un classifieur avec un taux d'erreur < 0.3 (error_rate).
4. Donner les identifiants et les noms des datasets où tous les classifieurs sont implémentés en 'Python'.
5. Donner pour chaque dataset le nombre de méthodes de classification utilisées.
6. Mettre à jour le nombre d'instances des datasets utilisés pour les classifieurs écrits en 'Python', en ajoutant 100 instances.

Partie 3 : sqlite3

On dispose d'un fichier texte nommé 'DataMeth.txt' contenant des informations relatives aux méthodes utilisées par les classifieurs, chaque ligne du fichier a la forme suivante :

Nom_méthode#catégorie#description

7. Ecrire les instructions python permettant de :
   - importer le module sqlite3 ;
   - se connecter à la base de données 'classifieurs.db';
   - créer le curseur **cur** d'exécution ;
   - créer la table Method ;
   - remplir la table Method à partir du fichier 'DataMeth.txt' ;
   - tracer la courbe dont les abscisses sont les identifiants des datasets et les ordonnées sont les taux d'erreur moyens des classifieurs associés.

ANNEXE 1 – Nomenclature associée à la Partie 3 du PROBLEME 1

Nom	Type	Description
DSET **DSET1** **DSET2**	*numpy.ndarray*	Matrices représentants des données d'apprentissage
n, m	*int*	Respectivement nombre de lignes et de colonnes de **DSET**
ind	*int*	Indice d'une colonne de la matrice **DSET**
distr	*dict*	Dictionnaire représentant la distribution des probabilités des décisions
qual	*list*	Liste indiquant les observations qualitatives et les observations quantitatives
Lcol	*list*	Liste de chaines des caractères représentant les noms des observations ainsi que le nom de la décision
obs	*str*	chaine de caractères représentant le nom d'une observation
S	*float*	Seuil de découpage
Lc	*list*	Liste de tuples où chaque tuple est formé par la valeur d'une observation et la valeur de la décision associée
vBest	*float*	Impureté donnant le meilleur seuil
sBest	*float*	Meilleur seuil
LSeuil	*list*	Liste des seuils possibles de découpage
oBest	*str*	Nom de l'observation qui a la meilleure impureté

ANNEXE 2 – Quelques fonctions Python

Opérations utiles sur les itérables (str, tuple, list, dict, etc.)

- **len**(it) retourne le nombre d'éléments de l'itérable it.
- **range**(d,f) retourne la séquence de valeurs entières successives comprises entre d et f exclu.
- **min**(it) (resp. **max**(it)) retourne la valeur minimale (resp. maximale) de l'itérable it.
- x **in** it (resp. x **not in** it) vérifie si x appartient à it (resp. n'appartient pas).
- **sorted**(it) retourne une liste contenant les éléments de it dans l'ordre croissant.
- lst.**sort**() trie la liste lst dans l'ordre croissant.
- lst.**count**(val) retourne le nombre d'occurrences de val dans la liste lst.
- lst.**append**(val) ajoute val à la fin de la liste lst.
- lst.**remove**(val) supprime la première occurrence de val dans la liste lst.
- lst.**index**(val) retourne l'indice de la première occurrence de val dans la liste lst.
- source.**split**(motif) retourne une liste formée par des chaines de caractères résultant du découpage de la chaine source autour de la chaine motif.
- motif.**join**(itérable de chaine de caractère) retourne une chaine de caractères résultant de la concaténation des éléments de l'itérable intercalés par le motif.
- motif.**format**(paramètres) retourne une chaine de caractères obtenue en substituant dans l'ordre chaque '{}' dans motif par un objet de paramètres.
- d.**values**() retourne un itérable formé par les valeurs du dictionnaire d.
- d.**items**() retourne un itérable de couples (k,v) ou k est une clé du dictionnaire d et v est la valeur associée.
- M.**shape** ou **np.shape**(M) retourne un tuple formé par le nombre de lignes et le nombre de colonnes d'une matrice M.

Opérations sur les fichiers

- f=**open** (nomF,m) permet d'ouvrir le fichier nomF en mode m où m='r' ou 'w'.
- f.**close**( ) permet de fermer un fichier.
- f.**read**() permet de lire et retourner le contenu d'un fichier dans une chaine de caractères.
- f.**readline**() permet de lire et retourner le contenu de la ligne courante d'un fichier dans une chaine de caractères.
- f.**readlines**() : permet de lire et retourner le contenu de toutes les lignes d'un fichier dans une liste.

Opérations sur le module matplotlib.pyplot

- **import matplotlib.pyplot as plt** permet le chargement du module
- **plt.plot**(x,y) crée la courbe où les abscisses sont décrites par les valeurs de l'itérable x et les ordonnées par ceux de l'itérable y.
- **plt.show**() affiche une fenêtre contenant le résultat du dessin.

REPUBLIQUE TUNISIENNE
Ministère de l'Enseignement Supérieur,
de la Recherche Scientifique

Concours Nationaux d'Entrée
aux Cycles de Formation d'Ingénieurs
Session 2019

الجمهورية التونسية
وزارة التعليم العالي والبحث العلمي

المناظرات الوطنية للدخول
إلى مراحل تكوين المهندسين
دورة 2019

Concours Mathématique et Physique, Physique et Chimie et Technologie

Corrigé épreuve d'informatique

PROBLEME 1

Partie 1 : Représentation de la structure d'arbre binaire

1. la méthode isLeaf

```
def isLeaf(self):
  return self.left == None and self.right == None
   #ou bien   return self.left is self.right is None
```

2. la méthode __len__

```
def __len__(self):
  if self.isLeaf():
       return 1
  elif self.left == None:
       return 1 + len(self.right)
  elif self.right == None:
       return 1 + len(self.left)
  else:
       return 1 + len(self.left) + len(self.right)
```

3. la méthode __str__

```
def __str__(self):
  if self.isLeaf():
       return "Node("+self.label+")"
  else:
       return "Node("+self.label+","+str(self.left)+
              ","+str(self.right)+ ")"
```

Partie 2 : Représentation du modèle de décision

1. l'entête de la classe DecisionNode

```
class DecisionNode(Node):
```

2. la méthode __init__

```
def __init__(self,label,distr,S,left,right):
  Node.__init__(self,label,left,right)
  self.seuil = S
  self.distr = distr
```

3. la méthode outcome

```
def outcome(self,val):
  try:
        assert not self.isLeaf()
        if val >= self.seuil:
              return self.left
        else:
              return self.right
  except:
        print("le noeud est une feuille")
```

4. la méthode __str__

```
def __str__(self):
  L = self.linearize()
  ch=""
  for e in L:
        ch+="IF"
        for i in range(1,len(e)):
              if e[i] == e[i-1].left:
                    ch+=e[i-1].label+">="+str(e[i-1].seuil)+ "AND"
              else:
                    ch+=e[i-1].label+"<"+str(e[i-1].seuil)+ "AND"
        ch = ch[:len(ch)-4]+"THEN"+
              e[len(e)-1].label+"="+str(e[len(e)-1].distr)+ "\n"
  return ch
```

5. la méthode predict

```
def predict (self,dicobs):
  currentNode = self
  while not(currentNode.isLeaf())and currentNode.label in dicobs:
        curvalue = dicobs[currentNode.label]
        currentNode = currentNode.outcome(curvalue)
  return currentNode.distr
```

6. la méthode __init__

```
def __init__(self):
  self. listNodes = []
```

7. la méthode add

```
def add(self,dt):
  self. listNodes.append(dt)
```

8. la méthode predict

```
def predict(self, dicobs):
  if len(self.listNodes) == 0:
        distr = {0:0.5, 1:0.5}
  else:
        p0 = 0
        for e in self.listNodes:
```

```
            d = e.predict(dicobs)
            p0+=d[0]
        p0 = p0/len(self.listNodes)
        distr = {0:p0, 1:1-p0}
    return distr
```

Partie 3 : Apprentissage

1. la fonction CountValues

```
def CountValues(DSET,ind):
    e = set()
    for i in range(DSET.shape[0]):
        e.add(DSET[i,ind])
    return len(e)
```

2. la fonction EvalDistr

```
def EvalDistr(DSET):
    n,m = np.shape(DSET)
    if (n,m) == (0,0):
        d = {0:0.5,1:0.5}
    else:
        d={0:0,1:0}
        for i in range(n):
            if DSET[i,m-1] == 0:
                d[0]+=1/n
            else:
                d[1]+=1/n
    return d
```

3. la fonction IsPure

```
def IsPure(DSET):
    d = EvalDistr(DSET)
    if d[0] == 1 or d[1] == 1:
        return True
    else:
        return False
```

4. la fonction IsQualitative

```
def IsQualitative(DSET):
    n,m = np.shape(DSET)
    L = [True]*m
    for j in range(m):
        i = 0
        while 1:
            if DSET[i,j] not in [0,1]:
                L[j] = False
                break
            i+=1
            if i = n:
                break
```

```
    return L
```

5. la fonction Cut

```
def Cut(DSET, Lcol, obs, S = 0.5):
    indice = Lcol.index(obs)
    n,m = np.shape(DSET)
    n1,n2 = 0,0
    for i in range(n):
        if DSET[i,indice] >= S:
            n1+=1
        else:
            n2+=1
    ds1=np.empty((n1,m-1))
    ds2=np.empty((n2,m-1))
    k11,k21 = 0,0
    for i in range (n):
        if DSET[i,indice] >= S:
            k2=0
            for j in range(indice):
                ds1[k11,k2]=DSET[i,j]
                k2+=1
            for j in range(indice+1,m):
                ds1[k11,k2]=DSET[i,j]
                k2+=1
        k11+=1
        else:
            k2=0
            for j in range(indice):
                ds2[k21,k2]=DSET[i,j]
                k2+=1
            for j in range(indice+1,m):
                ds2[k21,k2]=DSET[i,j]
                k2+=1
            k21+=1
    L1 = Lcol[:indice] + Lcol[indice+1:]
    L2 = [ds1,ds2]
    L3 = [n1/n,n2/n]
    return L1 , L2 , L3
```

6. la fonction Impurity

```
def Impurity(DSET, Lcol, obs, S = 0.5):
    t = Cut(DSET, col, obs, S)
    p1 = t[2][0]
    p2 = t[2][1]
    ds1 = t[1][0]
    ds2 = t[1][1]
    d1 = EvalDistr(ds1)
    d2 = EvalDistr(ds2)
    m1 = min(d1[0],d1[1])
    m2 = min(d2[0],d2[1])
    return(p1*m1+p2*m2)
```

7. la fonction SortObs

```
def SortObs(DSET,Lcol,obs):
  n,m = np.shape(DSET)
  indice = Lcol.index(obs)
  Lc = []
  for i in range(n):
       Lc.append((DSET[i,indice],DSET[i,m-1]))
  Lc.sort()
  return Lc
```

8. la fonction BestCut

```
def BestCut(DSET, Lcol, obs, qual):
     i = Lcol.index(obs)
     if qual[i]:
          return Impurity(DSET,Lcol,obs,0.5))
     else:
          Lc = SortObs(DSET,Lcol,obs)
          if IsPure(DSET):
               LSeuil = [max(Lc)]
          else:
               LSeuil = []
               for i in range(len(Lc)-1):
                    if Lc[i][1]!=Lc[i+1][1]:
                         LSeuil.append((Lc[i][0]+Lc[i+1][0])/2)

          sBest = LSeuil[0]
          vbest = Impurity(DSET,Lcol,obs,sBest)
          for s in LSeuil:
               imp = Impurity(DSET,Lcol,obs,s)
               if imp<vBest:
                    vBest = imp
                    sBest = s
          return vBest,sBest
```

9. la fonction BuildTree

```
def BuildTree(DSET, Lcol, qual):
     distr = EvalDistr(DSET)
     if DSET.shape == (0,0)  or IsPure(DSET) or len(Lcol) == 1:
        return DecisionNode(Lcol[0], distr)
     else:
          LvBest = []
          LsBest=[]
          for obs in Lcol[:len(Lcol)-1]:
             sBest, vBest = BestCut(DSET, colonnes, obs, qual)
             LvBest.append(vBest)
             LsBest.append(sBest)
          indice=LvBest(min(LvBest))
          sBest=LsBest[indice]
```

```
            oBest=Lcol[indice]
                    L=Cut(DSET, Lcol, oBest, sBest)
            DSET1=L[1][0]
            DSET2=L[1][1]
            Lcol= L[0]
            rqual = qual[:]
            rqual.pop(Lcol.index(oBest))

    root=DecisionNode(oBest,distr,BuildTree(DSET1,Lcol,rqual),
    BuildTree(DSET2, Lcol, rqual),sBest)
    return root
```

PROBLEME 2

Partie 1 : algèbre relationnelle

1. $\Pi_{ds\_id, ds\_name, ds\_description}\left(\sigma_{format = 'csv'}(DataSet)\right)$
2. $\Pi_{cls\_decription}\left(\sigma_{language = 'Python' \text{ et } category = 'KNN'}(Classifieur \bowtie_{cls\_id} Combine \bowtie_{m\_name} Method)\right)$

Partie 2 : SQL

3.
```
SELECT ds_id FROM Classifieur WHERE error_rate < 0.3
```
4.
```
SELECT D.ds_id , ds_name FROM DataSet AS D , Classifieur AS C
 WHERE (D.ds_id = C.ds_id)
EXCEPT
SELECT D.ds_id , ds_name FROM DataSet AS D, Classifieur AS
WHERE (D.ds_id = C.ds_id) AND (language <> 'Python')
```
5.
```
SELECT D.ds_id , COUNT(DISTINCT M.m_name) AS NB_M
FROM DataSet AS D , Classifieur AS C , Combine AS CM , Method AS M
WHERE (D.ds_id = C.ds_id)
 AND (C.cls_id = CM.cls_id)
 AND (CM.m_name = M.m_name)
GROUP BY D.ds_id
```
6.
```
UPDATE DataSet SET nb_instances = nb_instances + 100
WHERE ds_id IN (SELECT ds_id FROM Classifieur WHERE language ='Python')
```

Partie 3 : sqlite3

7.
```
import sqlite3
cnx = sqlite3.connect("classifieur.db")
cur = cnx.cursor()

cur.execute("CREATE TABLE Method (m_name TEXT PRIMARY KEY,
                                  category TEXT, m_description TEXT ) ")
```

```
f = open("DataMeth.txt",'r')
L = f.readlines()
Ldata = [ l.strip().split('#') for l in L]
```

```
# ou bien
ldata =
[l.strip().split('
#') for l in f]
```

```
cur.executemany("INSERT INTO Method VALUES(?,?,?) ", ldata)
```

```
cur.execute("SELECT ds_id , AVG(error_rate) AS M_erreur FROM Classifieur
                    GROUP BY ds_id ORDER BY ds_id")
```

```
L=cur.fetchall()
x=[i[0] for i in L]
y=[i[1] for i in L]
```

```
# ou bien
x, y =
zip(*cur.f
etchall())
```

```
import matplotlib.pyplot as plt
plt.plot(x,y)
plt.show()
```