Contrôle Final - CSC3101 - 18 Janvier 2023 - 2h

Tout document écrit est autorisé. Pour répondre à une question, vous donnez le code correspondant (évitez les répétitions). Les classes sont à placer dans le paquetage par défaut. La visibilité des méthodes et attributs est celle par défaut. Il n'y a pas à écrire les directives `import`

Les arbres sont utilisés dans de nombreux domaines de l'informatique. Pour rappel, un arbre contient des nœuds. Les fils d'un nœud (zéro à plusieurs) sont les nœuds situés juste au-dessous de lui. Le nœud en haut de l'arbre est la racine. À l'inverse, une feuille est située en bas de l'arbre.

En fouille de données et en apprentissage automatique, on utilise des arbres de classification. Dans un arbre de classification, les feuilles représentent les valeurs cibles, ou *label*, d'une donnée et les embranchements correspondent à des règles qui mènent à ces valeurs. Au cours de cet examen, nous allons coder un arbre de classification et l'utiliser pour prédire les labels d'un jeu de données.

1. Les données *(~30 minutes, 8pt)*

Une donnée contient des valeurs dont un label. Dans cet examen, le label sera toujours booléen. Un exemple de donnée vous est fourni ci-dessous.

```
"SUNNY,NO,69,70,TRUE"
```

L'information stockée ici est que par beau temps (`SUNNY`) et sans vent (`NO`), avec une température de 69 degrés Fahrenheit (`69`) et un taux d'humidité de 70% (`70`), les enfants vont jouer dehors (`TRUE`). Le label apparaît en dernier dans cette donnée.

Nous stockons chaque donnée dans une instance de la classe `Datum`. Cette classe possède deux attributs, un tableau d'entiers `features` et un booléen `label`.

[**Q1a**] *(1pt)* Donnez le code de la classe `Datum`. Le constructeur de cette classe reçoit en argument un tableau d'entiers (`int[]`) et un booléen (`boolean`) pour initialiser les deux attributs.

[**Q1b**] *(2pt)* Ajoutez ensuite les méthodes suivantes :
- `boolean getLabel()` retourne le label,
- `int get(int i)` retourne l'entier stocké à l'indice `i` de `features` (on assumera que `i < features.legnth`),
- `int dimension()` retourne la taille du tableau `features`.

Reprenons le cas de la donnée ci-dessus. Pour l'encoder sous forme de `Datum`, il nous faut trouver une valeur entière à `SUNNY` et `NO`. Par exemple, les valeurs respectives `2` et `0`. Cela correspond donc au code suivant :

```
int[] features = {2,0,69,70};
Datum d1 = new Datum(features, true);
```

[**Q1c**] *(1pt)* Que retournerait un appel à `d1.get(3)` dans le code ci-dessus ?

Passons maintenant à l'encodage d'un jeu de données. Pour ce faire, nous utilisons la classe `Dataset`. Les données sont enregistrées dans une liste nommée `data`. Une instance de cette classe offre les méthodes suivantes :
- `void add(Datum datum)` ajoute `datum` à la liste `data`,
- `List<Datum> getData()` retourne la liste `data`.

[**Q1d**] *(2pt)* Donnez le code de `Dataset`, en incluant un constructeur qui initialise `data` à une liste vide de type `ArrayList<Datum>`. Pour ce faire, on vous rappelle que `ArrayList<E>` possède un constructeur sans paramètre. De plus, cette classe a une méthode `E get(int index)` qui retourne l'élément à l'indice `i`, et `void add(E e)` pour ajouter l'élément `e` en fin de liste.

La dimension d'un jeu de données correspond à la dimension de ses données (toutes sont supposées de même dimension).

[Q1e] *(2pt)* Ajouter une méthode `int dimension()` à `Dataset` qui retourne sa dimension, par exemple en retournant la dimension de la première donnée dans `data`. Dans le cas où la liste est vide, on lèvera une exception de type `IllegalStateException`. Il est à noter que cette exception existe déjà dans la librairie fournie par Java.

2. Prédiction à l'aide d'un arbre *(~1h, 8pt)*

Un arbre de classification prédit le label d'une donnée. Dans ce but, on parcourt l'arbre de haut en bas à partir de sa racine. Chaque nœud de l'arbre définit une valeur ainsi qu'une règle de classification. La règle indique comment continuer le parcours vers un des fils. Quand on arrive à une feuille, sa valeur nous donne le label de la donnée. Dans notre cas, un arbre de classification est binaire car les labels sont booléens.

```
(3, <=80)
├── (1, <=0)
│   ├── (0, <=0)
│   └── (0, <=0)
└── (0, <=1)
    ├── (0, <=0)
    └── (0, <=0)
```

L'illustration ci-dessus présente un arbre de classification. Chaque nœud bleu a un label de valeur `true` et chaque nœud rouge un label de valeur `false`. Un nœud de l'arbre contient une règle de la forme `(x, <=y)` pour faire le parcours. Elle se traduit comme suit : si l'indice `x` de la donnée est inférieur ou égale à `y`, alors on passe par le fils `true`; sinon, on continue vers le fils `false`.

Si on considère la donnée `"SUNNY,NO,69,70,TRUE"` correspondant à `Datum d1` son parcours dans l'arbre ci-dessus est le suivant : À la racine, le test `(3, <=80)` est positif car l'indice `3` du tableau features de `d1` vaut `70`. Dans le fils bleu de la racine, le test `(1, <=0)` est aussi positif car `NO` vaut `0` dans notre encodage. Ce qui nous conduit à la feuille bleu `(0, <=0)`. Le résultat de cette classification est donc le label `true`. L'arbre a prédit avec succès le label car il s'agit bien de celui de la donnée (`TRUE`).

Dans ce qui suit, nous allons coder l'arbre de classification ainsi que l'algorithme de prédiction.

2.1 Les nœuds Chaque nœud est codé par une instance de la classe `Node`. Un nœud a deux fils `trueChild` et `falseChild`. Il stocke aussi deux variables `int feature` et `float cutoff` qui définissent la règle `(feature, <=cutoff)`. Enfin, chaque nœud a aussi une valeur `boolean value`.

[Q2a] *(1pt)* Donnez le code du constructeur de `Node`. Ce constructeur prend en paramètre un booléen pour initialiser `value`. Les fils sont mis à la valeur `null` et les valeurs de `feature` et `cutoff` sont à `0`.

[Q2b] *(2pt)* La méthode d'instance `boolean predict(Datum datum)` prédit le label de la donnée passée en paramètre. Pour ce faire, la méthode regarde d'abord si `trueChild` vaut `null`. Si c'est le cas, la méthode retourne la valeur stockée dans le nœud (dans ce cas, le nœud est donc une feuille). Sinon, la règle de classification est appliquée sur `datum` et son résultat retourné; à savoir si `feature, <=cutoff` est vrai on retourne `true`, et sinon `false`.

2.2 Arbre de classification Un arbre de classification appartient à la classe `ClassificationTree`. Initialement, il contient un nœud racine dont la valeur est fixée à `true`.

[Q2c] *(1pt)* Donnez le code de la classe `ClassificationTree`. Incluez un constructeur qui initialise la racine.

[Q2d] *(4pt)* La méthode `boolean predict(Datum datum)` de `ClassificationTree` permet de prédire le label de la donnée `datum`. Pour ce faire, vous appliquez l'algorithme ci-dessus de parcours de l'arbre.

Ce parcours est itératif, en passant d'un nœud à ses fils (et non récursif, comme dans les CIs 4 et 6). Pendant la traversée de l'arbre, on prendra garde à que tout nouveau nœud exploré ne soit pas `null`.

3. Apprentissage supervisé *(~30 minutes, 4+2pt)*

Dans cette troisième partie, nous allons effectuer la construction de l'arbre de classification. Cette construction se fait par apprentissage supervisé. Dans cet apprentissage, on utilise un jeu de données convenablement labellisées, dit jeu d'entraînement. Par exemple, le tableau suivant correspond à un jeu d'entraînement.

```
String[] file = {"RAINY,NO,70,96,TRUE",
         "RAINY,NO,68,80,TRUE",
         "RAINY,YES,65,70,FALSE",
         "OVERCAST,YES,64,65,TRUE",
         "SUNNY,NO,72,95,FALSE",
         "SUNNY,NO,69,70,TRUE",
         "SUNNY,YES,80,90,FALSE",
         "RAINY,NO,75,80,TRUE",
         "SUNNY,YES,75,70,TRUE",
         "OVERCAST,YES,72,90,TRUE",
         "RAINY,YES,71,91,FALSE",
         "OVERCAST,NO,83,86,TRUE",
         "SUNNY,NO,85,85,FALSE",
         "OVERCAST,NO,81,75,TRUE"};
```

En partant du jeu d'entraînement, on agrandit l'arbre en divisant successivement ses feuilles. Chaque nœud ainsi divisé donne lieu à deux feuilles (`trueChild` et `falseChild`), qui seront ses fils.

Pour diviser un nœud, on applique sa règle de classification au jeu d'entraînement. Les données validant la règle seront utilisées pour entraîner `trueChild`. Les autres seront utilisées pour l'entraînement de `falseChild`. Par exemple, si on applique la règle `(2, <=65)` au jeu d'entraînement `file`, les éléments suivants vont servir à entraîner `trueChild` : `"RAINY,YES,65,70,FALSE"` et `"OVERCAST,YES,64,65,TRUE"`.

Dans ce qui suit, on suppose que la classe `Node` possède un champ supplémentaire `Dataset training`. Ce champ est passé en argument du constructeur (dont la signature est désormais `Node(Dataset training, boolean value)`). Par ailleurs, on suppose que `feature` et `cutoff` ont des valeurs appropriées.

[Q3a] *(2pt)* Ajoutez une méthode `void split()` à `Node`. Cette méthode doit appliquer la règle de classification `(feature, <=cutoff)` à toutes les données d'entraînement, créant ainsi deux nouveaux `Dataset`. Les jeux de données ainsi obtenus sont ensuite utilisés pour créer les deux fils du nœud.

[Q3b] *(2pt)* Ajoutez une méthode `void train(int height)` à `ClassificationTree`. Cette méthode entraîne itérativement les feuilles de l'arbre en utilisant la méthode `split` ci-dessus jusqu'à atteindre la hauteur `height`. Vous supposerez ici que l'arbre est créé avec un jeu d'entraînement passé en argument du constructeur qui est utilisé pour initialiser convenablement la racine.

NB: Pour rappel, la hauteur d'un arbre est la longueur du plus long chemin de la racine à une feuille. Il est aussi rappelé que la méthode `Math.pow(x,y)` calcule `x` à la puissance `y`. Pour stocker les nœuds, vous pouvez employer une file (`Queue<Node>`). La méthode `void add(Node n)` ajoute `n` en fin de liste. `Node poll()` enlève le premier nœud de la file puis le retourne.

Attention, la question bonus qui suit est longue et difficile. Il vous est fortement conseillé de la faire une fois que vous avez fait le reste de l'examen.

[Q3c] *(2pt, bonus!)* Dans cette question, nous calculons les valeurs de `feature` et `cutoff`. Pour ce faire, on suppose que la classe `Dataset` possède les méthodes suivantes :
- `int mostCommon(int i)` retourne la valeur la plus fréquente à l'indice `i` du jeu de données,

- `float impurity()` calcule l'impureté du `Dataset`, à savoir une valeur entre `0` et `1` indiquant à quel point les labels du `Dataset` sont uniformes. Plus précisément, soit un ensemble D de données ayant au plus L labels, l'impureté de D est définie par : $Gini(D) = 1 - \sum_{l=1}^{L} p_l^2$, où p_l est la probabilité d'une donnée de D d'avoir le label l. L'impureté est donc maximale si la distribution des labels est uniforme dans D, et elle vaut `0` si un seul label est présent.

Ajoutez une méthode `boolean findBestSplit()` à `Node`. Cette méthode retourne `false` si le jeu d'entraînement n'a pas d'impureté. Dans le cas contraire, on itère sur chacune des dimensions du jeu d'entraînement. A chaque indice `i`, vous effectuerez les étapes suivantes :

1. Sauvegarder les valeurs de `feature` et `cutoff` dans deux variables temporaires,
2. Stocker `i` dans `feature`,
3. Stocker dans `cutoff` la valeur la plus commune à l'indice `i` des données labellisées `true` dans le jeu d'entraînement (utilisez ici la méthode `mostCommon` de `Dataset`),
4. Diviser le nœud en appelant `split` ,
5. Calculer l'impureté pour cette division selon la formule suivante :

$$\frac{n_1}{n} Gini(D_1) + \frac{n_2}{n} Gini(D_2)$$

   où n est la cardinalité du jeu d'entraînement du nœud, et D_1 (respectivement, D_2) est le jeu d'entraînement du fils `trueChild` (resp. `falseChild`) de cardinalité n_1 (resp. n_2)
6. Si l'impureté ainsi obtenue n'est pas inférieure à celle obtenue lors de l'itération précédente, on revient en arrière. A savoir, les valeurs de `feature` et `cutoff` sont restaurées et la division du nœud recalculée.

On peut désormais entraîner l'arbre convenablement en remplaçant `split` dans `ClassificationTree.train` par `findBestSplit`. Voici un exemple d'entraînement à partir des données dans `file` :

```
(0, <=0)

Feature: 0, cutoff: 1, impurity: 0.39365083, best impurity: 1.0
Feature: 1, cutoff: 0, impurity: 0.42857146, best impurity: 0.39365083
Feature: 2, cutoff: 75, impurity: 0.44285715, best impurity: 0.39365083
Feature: 3, cutoff: 80, impurity: 0.36734694, best impurity: 0.39365083

(3, <=80)
├── (0, <=0)
└── (0, <=0)

Feature: 0, cutoff: 0, impurity: 0.1904762, best impurity: 1.0
Feature: 1, cutoff: 0, impurity: 0.1904762, best impurity: 0.1904762
Feature: 2, cutoff: 75, impurity: 0.23809522, best impurity: 0.1904762
Feature: 3, cutoff: 80, impurity: 0.24489796, best impurity: 0.1904762

(3, <=80)
├── (1, <=0)
│   ├── (0, <=0)
│   └── (0, <=0)
└── (0, <=0)
```