INF2022
Algorithmique avancée
Enseignants
Arnaud Bannier, Nicolas Bodin, Matthieu Le Berre

Classification avec des arbres de décision

1 Introduction

Au cours de cette activité, vous allez implémenter une intelligence artificielle utilisant les concepts vus lors de ce semestre. Cependant, il ne s'agit pas d'un jeu : on va faire de la classification. Dans le domaine de l'apprentissage automatique et de l'analyse de données, la classification est une tâche fondamentale. Elle consiste à attribuer des étiquettes (des classes) à des données en fonction de leurs caractéristiques. La classification est omniprésente dans de nombreuses applications : détection de spam, diagnostic médical et encore recommandation de produits en ligne. Commençons par un peu de terminologie :

- **Dataset (Ensemble de données)** : Un dataset est une collection structurée de données qui est utilisée pour entraîner, tester et évaluer des modèles de classification. Il est composé d'instances et de features. Les datasets sont essentiels pour la création de modèles d'apprentissage automatique, car ils contiennent les informations nécessaires pour effectuer la classification.
- **Instance (Exemple)** : Une instance, également appelée exemple, est une observation individuelle dans un dataset. Elle représente un objet, un événement ou une entité spécifique que l'on souhaite classifier. Par exemple, dans un dataset de classification d'animaux, chaque animal serait une instance avec des caractéristiques spécifiques, auquel on attribuerait une classe.
- **Feature (Caractéristique)** : Une feature, parfois appelée attribut, est une propriété ou une caractéristique de chaque instance qui est utilisée pour effectuer la classification. Les features sont les variables que le modèle d'arbre de décision analyse pour prendre des décisions. Dans le cas de la classification d'animaux, les features pourraient inclure des éléments tels que la taille, le poids, la couleur, etc.

Il existe de nombreuses méthodes de classification : les réseaux de neurones, les machines à support vectoriel, ... Nous nous intéresserons ici aux arbres de décision. Un arbre de décision est un modèle simple et rapide à construire, qui sépare les instances en se basant sur les features et des seuils.

La Figure 1 donne un exemple d'arbre de décision avec quatre noeuds internes et 5 feuilles. Chaque noeud interne est capable d'effectuer un test : on se base sur une feature, que l'on compare à une valeur fixe. Si la feature est inférieure ou égale à ce seuil, alors on part à gauche, sinon on part à droite. Les feuilles contiennent quant à elles une valeur de classe.

Imaginons maintenant une instance définie par $f = [22, 6, 32, 0, 13]$. On part de la racine, dans laquelle on compare la valeur de $f[4]$ au seuil valant 12. Ici, $f[4]$ vaut 13, et est strictement supérieure au seuil, on part donc sur le fils droit de la racine. A nouveau, on doit faire un test, se basant cette fois-ci sur la feature $f[1]$ et un seuil valant 7. Dans notre exemple, $f[1]$ a pour valeur 6, et est donc inférieure ou égale au seuil, on se déplace

donc sur le fils gauche du noeud courant. Enfin, on arrive sur une feuille, qui contient la valeur entière 2. Notre arbre prédit alors que l'instance étudiée est de la classe 2.

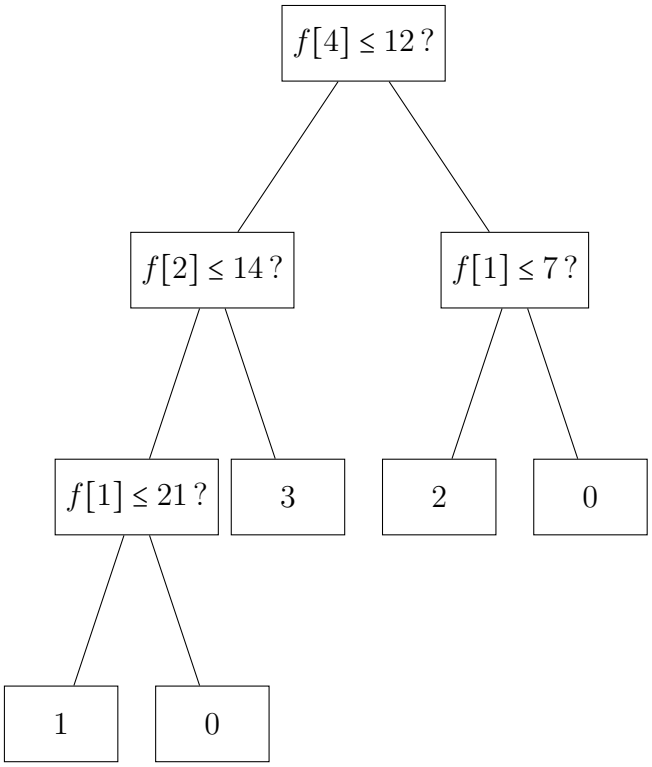

FIGURE 1 – Un arbre de décision

Ce sujet vous permettra d'implémenter votre propre générateur d'arbres de décisions, et est divisé en plusieurs parties :

- la section 2 vous guide dans l'implémentation du chargement des datasets ;
- la section 3 donne les différentes étapes de construction d'un arbre en fonction d'un dataset, ainsi que quelques détails sur la prédiction d'une classe par un arbre pour une instance donnée ;
- la section 4 introduit la notion de forêt, permettant d'exploiter de multiples arbres de décision sur un même dataset ;
- la section 5 donne des pistes d'amélioration ;
- la section 6 résume les conditions d'évaluation de votre travail.

2 Chargement d'un dataset

Dans le cadre de cette activité, nous n'utiliserons que des datasets d'images. Chaque instance sera donc une image (carrée), et chaque feature un pixel, en nuances de gris. Chaque dataset fourni respectera le format suivant :

1. la première ligne contient trois entiers N, C et M séparés par des espaces, et représentant respectivement le nombre d'instances composant le dataset, le nombre de classes différentes et le nombre de features (i.e. le nombre de pixels composant une image) ;

2. les N lignes suivantes décrivent les instances et respectent toutes le même format. Par exemple, la ligne $i+1$ contient c_i la classe de l'instance i (entre 0 et $C-1$), suivie de M entiers $f_{1,i}, f_{2,i}, ..., f_{M,i}$ donnant les valeurs des features pour l'instance i.

Sur Moodle, vous retrouverez trois datasets : PENDIGITS, MNIST et FASHION. Ces trois datasets sont divisés en deux fichiers `train` et `test`, découpage qui sera expliqué dans la section 3 de ce document. Le format des fichiers .txt respecte celui décrit ci-dessus.

Vous commencerez par créer deux fichiers `Dataset.h` et `Dataset.c` qui se chargeront, entre autres, de l'extraction des données des fichiers. Ces fichiers s'articulent autour de quatre structures.

```
typedef struct sInstance{
  int* values;
  int classID;
} Instance;
```

La première, `Instance`, est composée d'un tableau de valeurs (entières) représentant les features, et d'un entier correspondant à la classe de l'instance. Une variable de ce type contiendra donc les valeurs des différents pixels d'une image, ainsi que sa classe.

```
typedef struct sDataset{
  Instance *instances;
  int instanceCount;
  int featureCount;
  int classCount;
} Dataset;
```

La deuxième structure contient le dataset original chargé du fichier. On y retrouve les instances composant ce dataset, stockées à l'intérieur d'un tableau, le nombre d'instances, le nombre de features et le nombre de classes.

```
typedef struct sSubproblemClass{
  int instanceCount;
  Instance **instances;
} SubproblemClass;

typedef struct sSubproblem{
  Instance **instances;
  int instanceCount;
  int capacity;
  int featureCount;
  int classCount;
  SubproblemClass *classes;
} Subproblem;
```

Enfin, les structures `Subproblem` et `SubproblemClass` seront utilisés pour la création de sous-problèmes. Les sous-problèmes seront utiles pour la génération des noeuds de l'arbre : si la racine est générée en considérant l'ensemble des instances, cet ensemble sera

ensuite divisé en deux partie, une pour le fils gauche, et une pour le fils droit. Et ce processus de division se répétera à chaque création de noeud interne. La structure `Subproblem` contient le tableau de pointeurs vers les instances à considérer, celles-ci étant également organisées en sous-tableaux, un par classe, au sein des structures `SubproblemClass`. On retrouvera donc l'adresse d'une instance i particulière à la fois dans le tableau de la structure `Subproblem`, qui recense toutes les instances en considération, mais également dans le tableau de la structure `SubproblemClass` relative à c_i.

Vous commencerez par implémenter les fonctions suivantes :

- `Dataset* Dataset_readFromFile(char* filename);` lit le fichier dont le nom est donné, et extrait toutes les informations du dataset. Ces informations seront enregistrées dans un `Dataset` dont l'adresse sera renvoyée. Si la lecture du fichier échoue, la fonction renvoie `NULL`.
- `void Dataset_destroy(Dataset *data);` détruit proprement une allocation de la structure `Dataset` dont l'adresse est donnée en paramètre. On pensera également à désallouer le tableau d'instances présent à l'intérieur de la structure.
- `Subproblem *Dataset_getSubproblem(Dataset *data);` renvoie un sous-problème prenant en compte toutes les instances stockées dans le dataset dont l'adresse est donnée en paramètre. **Attention :** on ne demande pas de copier les instances contenues dans `Dataset`, mais de simplement stocker leur adresse dans un tableau. On en profitera également pour recopier les valeurs `instanceCount`, `featureCount` et `classCount`, qui seront identiques dans les deux structures.
- `Subproblem *Subproblem_create(int maximumCapacity, int featureCount, int classCount);` renvoie une allocation de la structure `Subproblem`, en instanciant les valeurs des attributs `featureCount` et `classCount`, et en allouant le tableau de pointeurs d'instance à sa capacité donnée en paramètre.
- `void Subproblem_destroy(Subproblem *subproblem);` détruit proprement une allocation de `Subproblem`, en détruisant les différents tableaux. **Attention :** ce n'est pas parce que l'on travaille avec des tableaux de pointeurs qu'il faut détruire ce qui se trouve dans les cases des dits-tableaux, les adresses stockées font références aux instances présentes dans la structure `Dataset`.
- `void Subproblem_insert(Subproblem *subproblem, Instance *instance);` insère l'adresse d'une instance dans le sous-problème, à la fois dans le tableau principal et le tableau dédié à la classe de l'instance, et met à jour le nombre d'instances considérées.
- `void Subproblem_print(Subproblem *subproblem);` affiche le nombre de features, de classes et d'instances référencées, ainsi que le nombre d'instances pour chacune des classes.

Dans le code de test ci-dessous, on commence par charger les données d'entraînement du dataset PENDIGITS. Ensuite, on créé une instanciation de `Subproblem` prenant en compte le problème global (i.e. toutes les instances), et on affiche les détails. Enfin, on détruit les instanciations de `Subproblem` et `Dataset`.

```c
int main(int argc, char** argv){
  char path[128] = "Datasets/PENDIGITS/PENDIGITS_train.txt";
  Dataset *trainData = Dataset_readFromFile(path);

  Subproblem *subproblem = Dataset_getSubproblem(trainData);
  Subproblem_print(subproblem);

  Subproblem_destroy(subproblem);
  Dataset_destroy(trainData);
  return 0;
}
```

```
Dataset with 10 classes of 16 features
Size = 7494, capacity = 7494
- class 0: 780 instances
- class 1: 779 instances
- class 2: 780 instances
- class 3: 719 instances
- class 4: 780 instances
- class 5: 720 instances
- class 6: 720 instances
- class 7: 778 instances
- class 8: 719 instances
- class 9: 719 instances
```

3 Arbre de décision

Un arbre de décision est un classifieur défini par un arbre binaire. Pour un problème donné, la construction de l'arbre correspond à la phase d'entraînement. Durant cette phase, l'arbre apprend à distinguer les classes les unes des autres. Une fois la construction de l'arbre terminée, l'apprentissage prend fin, et on peut utiliser celui-ci pour traiter des instances nouvelles : c'est la phase d'inférence. En testant l'arbre sur ces nouvelles instances, on estimera la capacité de celui-ci à utiliser son apprentissage pour généraliser. Ainsi, un arbre correctement formé sera capable d'identifier la classe de la plupart des nouvelles instances.

Le principe de l'arbre de décision est le suivant :

- chaque nœud interne représente un test logique, défini ici comme la comparaison de la valeur d'une feature à un seuil. On se contentera de fixer la forme de nos tests comme il suit : $f[i] \leq \varphi$, avec $f[i]$ la feature concernée par le test, et φ le seuil. A un nœud donné, si la valeur de la feature est inférieure ou égale au seuil, alors on passe au fils gauche, sinon on passe au fils droit ;
- les feuilles représentent des classes : lorsque l'on arrive sur une feuille, alors on obtient la décision de l'arbre concernant l'instance étudiée.

Avant de passer à la suite, créez quatre fichiers `Split.h`, `Split.c`, `DecisionTree.h` et `DecisionTree.c`, et déclarez les structures suivantes :

```c
//Dans Split.h
typedef struct sSplit{
  int featureID;
  float threshold;
} Split;

//Dans DecisionTree.h
typedef struct sDecisionTreeNode{
  struct sDecisionTreeNode* left;
  struct sDecisionTreeNode* right;
  Split split;
  int classID;
} DecisionTreeNode;
```

La première structure représente un test : `featureID` permet de savoir à quelle feature on se fie, et `threshold` contient la valeur du seuil du test. La deuxième structure représente un nœud d'arbre, avec `left` et `right` représentant les fils gauches et droits, et `split` le test permettant de choisir la route à emprunter. Si `left` et `right` sont à `NULL`, alors on est sur une feuille, et l'entier `classID` représente la réponse de l'arbre à l'instance.

3.1 Construction d'un arbre de décision

L'idée fondamentale derrière les arbres de décision est de diviser un ensemble de données en sous-ensembles plus petits, en fonction de caractéristiques spécifiques, jusqu'à ce que chaque sous-ensemble soit suffisamment homogène pour être associé à une classe ou une valeur de prédiction. En d'autres termes, ils permettent de prendre des décisions itératives en posant des questions successives sur les caractéristiques des données, en les divisant en groupes plus petits et enfin, en attribuant une étiquette ou une valeur à chaque groupe résultant. Pour la construction d'un arbre de décision, on suivra l'algorithme ci-dessous.

Algorithme 1 – Construction d'un nœud
Entrée. Un sous-problème sp et un seuil de pureté p
Sortie. Un nœud d'arbre

```
n ← création d'un nœud d'arbre
Si purete(sp)≥ p alors
    n.classID ← la classe majoritaire de sp
    Renvoyer n
Fin Si
n.split ← calculSplit(sp)
sp_l ← {i ∈ sp.instances | i.f[n.split.featureID] ≤ n.split.threshold}
sp_r ← {i ∈ sp.instances | i.f[n.split.featureID] > n.split.threshold}
n.left ← construction(sp_l, p)
n.right ← construction(sp_r, p)
Renvoyer n
```

On entre dans la fonction avec deux paramètres sp et p, le premier représentant le sous-problème à traiter (pour la racine, on considérera l'ensemble des instances), et le deuxième un réel entre 0 et 1 indiquant la pureté nécessaire pour arrêter la construction

récursive de la branche. On commence par créer un nœud n, puis on teste la pureté du sous-problème. Cela consiste simplement à compter le nombre d'instances de la classe majoritaire de sp, puis à diviser cette valeur par le nombre total d'instances de sp. Si la pureté vaut 1, cela signifie que toutes les instances de sp sont de la même classe. Dans le cas où la pureté de sp atteint le seuil désiré p, alors on assigne à n l'étiquette correspondant à la classe majoritaire de sp, et on renvoie directement le nœud. Dans ce cas, n sera une feuille. Sinon, on calcule un split en considérant sp, qui nous permettra de diviser le sous-problème en deux sous-problèmes exclusifs. L'un sera traité par le fils gauche de n, l'autre par le fils droit.

La difficulté principale est de savoir comment calculer un split. Il existe de nombreuses solutions, mais, pour l'instant, il est recommandé de suivre le processus défini ci-après. Soit un split $s = \{j, \varphi\}$ défini par une feature j et un seuil φ, et soit sp le sous-problème sur lequel on travaille. On commence par définir les notations suivantes :

- C est le nombre de classes ;
- sp_l est l'ensemble des instances de sp dont la valeur de la j-ème feature est inférieure ou égale à φ. Pour une classe $c \in \{0, .., C-1\}$, on définit également $I_{l,c}$ l'ensemble des instances de sp_l dont la classe est c.
- sp_r est l'ensemble des instances de sp dont la valeur de la j-ème feature est strictement supérieure à φ. Pour une classe $c \in \{0, .., C-1\}$, on définit également $I_{r,c}$ l'ensemble des instances de sp_r dont la classe est c.

Ensuite, on définit les impuretés des sous-problèmes sp_l et sp_r de la manière suivante :

$$g(sp_l) = 1 - \sum_{c=0}^{C-1} \left(\frac{|I_{l,c}|}{|sp_l|} \right)^2$$
$$g(sp_r) = 1 - \sum_{c=0}^{C-1} \left(\frac{|I_{r,c}|}{|sp_r|} \right)^2$$

Ici, $g(sp_l)$ et $g(sp_r)$ correspondent respectivement aux mesures d'impureté des sous-ensembles sp_l et sp_r. Considérant les classes présentes dans l'ensemble étudié, plus celui-ci sera hétérogène, plus sa mesure d'impureté sera proche de 1. A l'inverse, si un ensemble ne contient que des instances d'une même classe, sa mesure d'impureté sera nulle.

$$gini(sp, s) = \frac{|sp_l|}{|sp|} \times g(sp_l) + \frac{|sp_r|}{|sp|} \times g(sp_r)$$

Le mesure $gini(sp, s)$ donne l'impureté de sp en fonction du split s, et est définie en fonction des mesures d'impureté de sp_l et sp_r, tout en tenant compte de la proportionnalité : plus un des sous-problèmes considérés contiendra d'instances, plus sa mesure d'impureté aura d'impact sur la mesure finale. Enfin, à partir d'un sous-problème sp, on calculera le split à choisir de la manière suivante :

1. pour chaque feature $f[j]$, on détermine les valeurs min_j et max_j correspondant respectivement aux valeurs minimum et maximum de la feature $f[j]$ au sein du sous-problème sp ;
2. on calcule ensuite $\varphi_j = \frac{max_j + min_j}{2}$ qui correspondra au seuil du split ;
3. on détermine la pureté du split $s = (j, \varphi_j)$ via la formule de $gini(sp, s)$;
4. on retourne le meilleur split généré (i.e. celui avec l'impureté la plus faible).

Commençons l'implémentation dans les fichiers `Split.h` et `Split.c`, dans lesquels vous implémenterez les fonctions suivantes :

- `float Split_gini(Subproblem *sp, int featureID, float threshold);` réalise le calcul d'impureté tel qu'il est défini plus haut dans le texte, en tenant compte du sous-problème passé en paramètre, mais également de la feature et du seuil caractérisant le split étudié. Le résultat sera obligatoirement entre 0 et 1 inclus.
- `Split Split_compute(Subproblem *subproblem);` cette fonction va itérer sur les features et générer à chaque fois un split comme expliqué plus haut. Le split générant le moins d'impureté sera renvoyé.

Ensuite, on se place dans les fichiers `DecisionTree.h` et `DecisionTree.c`, et vous implémenterez :

- `DecisionTreeNode* DecisionTree_create(Subproblem* sp, int currentDepth, int maxDepth, float prunningThreshold);` implémente l'algorithme donné plus haut, tout en ajoutant une profondeur maximum qui sert de cas d'arrêt supplémentaire. On a également `prunningThreshold` qui représente le seuil de pureté à atteindre pour qu'un nœud soit une feuille ;
- `void DecisionTree_destroy(DecisionTreeNode *decisionTree);` détruit récursivement les nœuds de l'arbre ;
- `int Decision_nodeCount(DecisionTreeNode* node);` renvoie le nombre de nœuds présents dans l'arbre (y compris les feuilles), qui servira à vérifier le bon fonctionnement de votre procédure de construction, et également d'obtenir la taille de votre modèle.

Dans le code de test ci-dessous, on commence par charger les données d'entraînement du dataset PENDIGITS. Ensuite, on créé une instanciation de `Subproblem` prenant en compte le problème global (i.e. toutes les instances). Ensuite, on créé un arbre de décision de profondeur maximum 30, avec un seuil de pureté de 1 (un nœud doit être complètement pur, et donc ne contenir qu'une seule classe, pour être une feuille). Après cela, on affiche le nombre de nœuds de l'arbre. Enfin, on détruit l'arbre et l'instanciation de `Dataset` (`sp` ayant été détruit lors de la construction de l'arbre).

```
int main(int argc, char** argv){
  char path[128] = "Datasets/PENDIGITS/PENDIGITS_train.txt";
  Dataset *trainData = Dataset_readFromFile(path);

  Subproblem *sp = Dataset_getSubproblem(trainData);
  DecisionTreeNode *tree = DecisionTree_create(sp, 0, 30, 1.0);

  printf("Génération d'un arbre de %d nœuds\n",
  ↪  Decision_nodeCount(tree));

  DecisionTree_destroy(tree);
  Dataset_destroy(trainData);
  return 0;
}
```

Votre code devrait afficher ceci :

```
Génération d'un arbre de 895 nœuds
```

3.2 Prédiction d'un arbre de décision

Une fois un arbre de décision construit, celui-ci est capable de prendre des décisions. Le but est que lorsque l'on lui donne une instance, il soit capable de se baser sur les valeurs de features de celle-ci afin de déterminer sa classe. A chaque nœud parcouru, on vérifiera d'abord si celui-ci est une feuille. Dans ce cas, la fonction renverra directement la classe représentée par la dite-feuille. Sinon, on vérifie si le test du split du nœud est respecté par l'instance : si c'est le cas, on descend vers le fils gauche du nœud, sinon on part vers le fils droit du nœud.

Vous avez constaté que chaque problème est divisé en deux fichiers "train" et "test". Le premier fichier représente la partie dédiée à l'entraînement, c'est à dire à la construction de l'arbre. Le deuxième fichier contient des instances inédites (i.e. différentes des instances présentes dans le fichier de train), et servira à évaluer les performances de l'arbre. Dans un premier temps, vous implémenterez les deux fonctions suivantes dans `DecisionTree.h` et `DecisionTree.c` :

- `int DecisionTree_predict(DecisionTreeNode *tree, Instance* instance);` détermine la réponse de l'arbre à l'instance. On part de la racine et on descend dans l'arbre jusqu'à atteindre une feuille. La classe représentée par cette feuille correspondra à la prédiction de l'arbre.
- `float DecisionTree_evaluate(DecisionTreeNode *tree, Dataset *dataset);` permet de calculer la précision de l'arbre sur le dataset passé en paramètre. Le score renvoyé est un réel entre 0 (l'arbre s'est tout le temps trompé) et 1 (l'arbre a déterminé correctement la classe de chacune des instances du dataset). On le calculera facilement comme le nombre d'instances bien classées sur le nombre total d'instances.

Dans le code de test ci-dessous, on commence par charger les données d'entraînement du dataset PENDIGITS ainsi que les données de test. Ensuite, on créé une instanciation de `Subproblem` prenant en compte le problème global (i.e. toutes les instances). Après ça, on créé un arbre de décision de profondeur maximum 30, avec un seuil de pureté de 1 (un nœud doit être complètement pur, et donc ne contenir qu'une seule classe, pour être une feuille). L'arbre de décision est construit grâce au dataset d'entraînement, et ne connaît pas le dataset de test. On calcule ensuite les précisions de l'arbre sur les données d'entraînement et de test, que l'on affiche.

```
int main(int argc, char** argv){
  char pathTrain[128] = "Datasets/PENDIGITS/PENDIGITS_train.txt";
  Dataset *trainData = Dataset_readFromFile(pathTrain);
  char pathTest[128] = "Datasets/PENDIGITS/PENDIGITS_test.txt";
  Dataset *testData = Dataset_readFromFile(pathTest);

  Subproblem *sp = Dataset_getSubproblem(trainData);
  DecisionTreeNode *tree = DecisionTree_create(sp, 0, 30, 1.0);

  float scoreTrain = DecisionTree_evaluate(tree, trainData);
  float scoreTest = DecisionTree_evaluate(tree, testData);

  printf("train = %.3f, test = %.3f\n", trainScore, testScore);
```

```
    Subproblem_destroy(sp);
    DecisionTree_destroy(tree);
    Dataset_destroy(trainData);
    Dataset_destroy(testData);
    return 0;
}
```

Normalement, vous devriez obtenir les résultats suivants :

```
train = 1.000, test = 0.919
```

Le score de précision sur les données d'entraînement est parfait (100% de précision). C'est tout à fait logique, car on a construit notre arbre sur ces données, tout en exigeant une pureté totale des feuilles. En revanche, la précision de notre arbre sur les données de test n'est que de 91.9%, ce qui est améliorable. Lorsque l'écart entre les score d'entraînement et de test est trop important, on parle de surapprentissage, ou *overfit*. Dans notre cas, cela revient à dire que notre arbre a tendance à apprendre par cœur le dataset d'entraînement, et a du mal à généraliser, c'est-à-dire à extraire les motifs importants à la caractérisation des classes. Pour éviter cela, on peut abaisser le taux de pureté de création des feuilles, ce qui diminuera le score d'entrainement mais pourra potentiellement augmenter la précision en test. Par exemple, modifier le code de la manière suivante :

```
DecisionTreeNode *tree = DecisionTree_create(dataset, 0, 30, 0.99f,
 ↪  NULL);
```

permet d'obtenir les résultats :

```
train = 0.998, test = 0.921
```

On voit que la précision en entraînement a baissé, les feuilles pouvant posséder une pureté entre 0.99 et 1.0. En revanche, on constate une faible augmentation du score sur les données de tests.

4 Random forest

Si les arbres de décision sont rapides et faciles à construire, leur tendance au surapprentissage les rend peu fiable sur cette application. Pour pallier ce problème, on les organise souvent en forêt. Notre classifieur ne sera donc plus un seul arbre, mais un ensemble d'arbres qui prendront leur décision via un vote. Chaque arbre émet une prédiction, et la prédiction majoritaire est finalement retenue par la forêt. Le problème, c'est que la création d'un arbre est déterministe, c'est-à-dire que si vous lancez 10 fois la construction d'un arbre sur un même problème, vous aurez 10 arbres identiques, qui prendront exactement les mêmes décisions. Il faut donc trouver un moyen "logique" de randomiser la construction des arbres. Une solution est de procéder au *bagging* sur les instances. Cela consiste à entraîner les arbres sur des sous-parties de l'ensemble initial des instances.

```
Subproblem *Dataset_bagging(Dataset *data, float proportion);
```

La fonction `Dataset_bagging` permet de produire un sous-problème considérant seulement une partie des instances du dataset original. Le nombre d'instances à considérer est donné par `proportion`. Par exemple, si on a 1000 instances, avec une proportion de 0.7, alors le sous-problème contiendra 700 instances. Cependant, il s'agit d'un tirage aléatoire avec remise, une instance peut se voir être sélectionnée plusieurs fois.

Ensuite, on introduit la structure d'une forêt, qui sera implémentée dans les fichiers `RandomForest.h` et `RandomForest.c` :

```
typedef struct sRandomForest{
  DecisionTreeNode** trees;
  int treeCount;
  int classCount;
} RandomForest;
```

On voit qu'une instance de RandomForest se résume simplement comme un tableau d'arbres. Cette structure s'accompagne de plusieurs fonctions :

- `RandomForest *RandomForest_create(int numberOfTrees, Dataset *data, int maxDepth, float baggingProportion, float prunningThreshold)` : permet la création d'une forêt d'arbres ;
- `int RandomForest_predict(RandomForest *rf, Instance *instance)` : enregistre la prédiction de chacun des arbres de la forêt sur l'instance en paramètre, et renvoie la prédiction majoritaire ;
- `float RandomForest_evaluate(RandomForest *rf, Dataset *data)` : renvoie la précision de la forêt sur le dataset en paramètre ;
- `int RandomForest_nodeCount(RandomForest *rf)` : renvoie la somme des nombres de noeuds des arbres de la forêt ;
- `void RandomForest_destroy(RandomForest *rf)` : détruit la forêt.

Voici un dernier code de test, qui lance cette fois-ci la construction d'une forêt, puis mesure la précision de celle-ci. La forêt est composée de 20 arbres, chaque arbre étant entraîné sur la moitié des instances originales (choisies aléatoirement, parfois avec des doublons). Les arbres ont une profondeur maximum de 30, et une pureté de feuille à 1.

```
int main(int argc, char** argv){
  char pathTrain[128] = "Datasets/PENDIGITS/PENDIGITS_train.txt";
  Dataset *trainData = Dataset_readFromFile(pathTrain);
  char pathTest[128] = "Datasets/PENDIGITS/PENDIGITS_test.txt";
  Dataset *testData = Dataset_readFromFile(pathTest);

  Subproblem *sp = Dataset_getSubproblem(trainData);
  RandomForest *rf = RandomForest_create(20, trainData, 30, 0.5f,
  ↪  1.0f);

  float scoreTrain = RandomForest_evaluate(rf, trainData);
  float scoreTest = RandomForest_evaluate(rf, testData);

  printf("train = %.3f, test = %.3f\n", trainScore, testScore);
```

```
    RandomForest_destroy(rf);
    Subproblem_destroy(sp);
    Dataset_destroy(trainData);
    Dataset_destroy(testData);
    return 0;
}
```

permet d'obtenir les résultats :

```
train = 0.996, test = 0.953
```

On voit que la précision sur les données de test est bien plus élevée que lors de l'utilisation d'un seul arbre. Notre classifieur, ici une forêt, est donc capable d'une meilleure généralisation.

5 Améliorations

Dans cette section, vous retrouverez des pistes pour améliorer votre implémentation des Random Forests. D'autres idées sont possibles, n'hésitez pas à en parler avec votre chargé de TD.

5.1 Calcul du split

Actuellement, on calcule les splits de manière simpliste. Cela a l'avantage de minimiser le temps de construction des arbres, mais cela se fait au détriment du nombre de noeuds. En fonction de l'application ciblée, la taille du modèle peut être importante. Par exemple, si vous souhaitez porter votre forêt sur un micro-contrôleur, le nombre de noeuds composant les arbres, et donc la taille du modèle, devient critique. Maximiser la qualité des splits permet en général de réduire le nombre de noeuds nécessaire pour qu'un arbre atteigne la pureté demandée. Soit un split s défini par une feature j, plutôt que de définir son seuil φ par $\frac{max_j+min_j}{2}$, on pourrait tester les différentes valeurs comprises dans l'ensemble $\{min_j, .., max_j\}$. Comparons les résultats des deux implémentations sur PENDIGITS :

```
//Implémentation actuelle
train = 0.994, test = 0.918, nbNodes=737
//Recherche de la meilleure valeur de split
train = 0.991, test = 0.914, nbNodes=337
```

Les résultats en terme de précision sont très proches, et on voit que la taille du modèle a été divisée par deux. On obtient donc un classifieur bien plus compact. Cependant, cela a un coût : la génération de l'arbre est bien plus longue. A vous de voir s'il est possible de trouver un bon compromis entre compacité et temps d'apprentissage, tout en préservant les performances.

Ensuite, toujours sur la génération des splits, nous nous sommes basées sur la mesure d'impureté de Gini. Ce n'est pas la seule existante, à vous d'en chercher de nouvelles et de les tester pour observer (ou pas) des améliorations.

5.2 Bagging

On l'a vu, le *bagging* sur les instances permet de diversifier une forêt. Il existe également le *bagging* sur les features, qui présente les avantages suivants :

- chaque arbre est entrainé sur tout le problème, et sera par conséquent plus "compétent" que dans la version actuelle ;
- ce *bagging* s'implémente facilement, on génère un tableau de booléens indiquant si on a le droit, ou non, d'utiliser les features dans les splits ;
- cela force les arbres à utiliser des stratégies de classification différentes.

Mais cela présente également des inconvénients :

- chaque arbre est entrainé sur tout le problème, c'est donc plus long pour le même nombre d'arbre ;
- le fait d'interdire d'utiliser des features peut provoquer l'impossibilité à l'arbre de séparer toutes les classes : il faut donc vérifier si, pour un noeud possible, aucun split n'est possible (tout va à gauche, ou tout va à droite, et le problème est répercuté sur un fils). Dans ce cas, on arrête les appels récursifs, et on fait de ce noeud une feuille ;
- combien de features sont nécessaires ? la moitié ? plus ? moins ?

5.3 Enregistrement et chargement d'un modèle

Si on se place sur MNIST ou FASHION, l'apprentissage peut être long. Il serait intéressant de pouvoir enregistrer une forêt dans un fichier et de la recharger.

6 Modalités d'évaluation

Ce TP doit être effectué par binôme de niveau équivalent. Votre projet doit pouvoir être compilé à l'aide de GCC (norme c11 et antérieure, Makefile autorisé) pour un projet Linux ou du compilateur natif MSVC pour une version Visual Studio 2022 sur Windows. Aucun autre compilateur n'est autorisé (pas de cmake, clion ou autre). La `libsdl` est autorisée pour une version graphique du projet. Pour toute autre bibliothèque, demandez à votre enseignant. L'usage d'IA générative est interdit.

La date limite de rendu est fixée au **vendredi 22 décembre, 23h59**. Vous retrouverez sur Moodle un dépôt VPL vous permettant de tester votre code et de le soumettre à une vérification anti-plagiat. Votre code doit produire un exécutable prenant en arguments de la ligne de commandes deux fichiers : le chemin vers le dataset train et celui du test. Il doit produire sur la sortie standard la prédiction de classe de chaque instance du test. Si vous disposez d'un projet avec plus de fonctionnalités, déposez-le dans la zone de dépôt bonus avant la date limite de rendu.

Afin de vérifier que vous êtes bien l'auteur du code que vous avez rendu, une soutenance par binôme aura lieu le jeudi 21 ou vendredi 22 décembre. Lors de cette soutenance, vous serez emmené à détailler certaines partie de votre code. Selon vos réponses, un facteur multiplicatif entre 0 et 1 s'appliquera à votre note de projet. Ce facteur multiplicatif est individuel.