Benjamin Auder

IMN637 - TP 1 : Arbre de décision

J'ai choisi d'implémenter mon propre algorithme en C++, qui est présenté ici.

I) Fonctionnement global

Architecture des classes :

Nœud : décrit un nœud de l'arbre, pouvant aussi être une feuille. Ses champs les plus importants sont :

a. type : 'f' si feuille,
'd' si le nœud discrimine les données suivant un attribut symbolique,
'c' si le nœud discrimine les données suivant un attribut continu,
'o' si le nœud discrimine les données suivant un attribut entier.
Les deux derniers cas sont très similaires.

b. distribExemples : un vecteur dont chaque élément contient des informations sur les données d'une même classe; une entrée de distribExemples comprend :
   1) la chaîne représentative de la classe commune aux éléments.
   2) L'index de cette classe dans le tableau des classes existantes.
   3) Le nombre d'exemples de cette classes que l'on trouve à ce nœud.
   4) Le vecteur des numéros de lignes des exemples dans le fichier.

c. Fils, père, nbFils : respectivement un pointeur sur le nœud père, sur les nœuds fils ainsi que le nombre de fils engendrés par ce nœud.

On trouve d'autres informations relatives notamment à la façon dont le nœud sépare les données, ainsi que la classe d'appartenance des données dans le cas d'une feuille.

Arbre : structure très simple comprenant un nœud racine (qui pointera sur le reste de l'arbre) et un vecteur de nœuds à explorer, utile pour la création de l'arbre. Arbre.cpp contient une fonction d'affichage graphique utilisant la librairie SDL.

Classif : classe au cœur du programme, qui travaille en deux temps; tout d'abord elle lit le fichier de données et recopie tout dans des structures plus adéquates au traitement de l'information en C++, puis dans la fonction *buildTree()* elle crée l'arbre de décision brut (non élagué) à partir des données et de l'heuristique (Gini, entropie ou misclassification error) choisie par l'utilisateur. Plus de détails au II).

Prune : une fois que l'arbre est construit, on regarde si l'on peut regrouper des feuilles pour gagner en généralité en appliquant les méthodes présentées en cours.

Test : classe similaire à *Classif* : dans le cas où des exemples tests sont disponibles dans un fichier .test, elle commence par lire les données et les recopier dans des structures

facilitant leur traitement, puis la fonction *testTree()* prend en paramètre un arbre et teste sa qualité sur le nouveau jeu de données. Elle renvoit un entier égal au nombre d'erreurs. Remarque : on utilise toujours un parcours de l'arbre itératif en profondeur « de gauche à droite », ce qui évite de stocker trop de données sur la pile avec des appels récursifs.

Graphic : classe encapsulant les primitives graphiques (dont je ne suis pas l'auteur).

II) Heuristiques de développement d'un nœud

Fonctions **findBestSplit___()** dans Classif.cpp :

On calcule simplement le gain d'information (pour pénaliser les divisions en trop de noeuds fils), avec l'heuristique choisie par l'utilisateur, sur chaque attribut (suivant son type on ne fait pas le même calcul), et l'on garde la meilleure séparation trouvée.

Cas des attributs ordonnés (discrets ou continus) : on trie les valeurs avant traitement.
Cas des attributs symboliques : on regroupe les symboles définissant une même classe, dans la mesure où cela ne regroupe pas tous les symboles ensemble.

III) Simplification de l'arbre

Prune.cpp :

L'idée est simple : pour chaque feuille de l'arbre, on regarde l'erreur de généralisation estimée en regroupant cette feuille ainsi que ses sœurs avec le père, à condition qu'une classe domine les autres en nombre (coefficient PRUNING_FILTER dans Arbre.h); on gagne en généralisation mais on fait plus d'erreurs sur l'ensemble d'entraînement. Utilisant la formule e'(T) = e(T) + N × PRUNING_FACTOR (N: number of leaf nodes) avec PRUNING_FACTOR d'autant plus grand que l'on souhaite élaguer l'arbre, on regarde si l'erreur de généralisation diminue. Si oui, alors on crée une nouvelle feuille qui est l'ancien père de la feuille de départ, puis on itère le processus sur une nouvelle feuille jusqu'à ce que l'on ne puisse plus améliorer l'erreur.

Dans le cas où l'on dispose d'un ensemble de données test, c'est plus facile car il suffit de regarder la somme pondérée des taux d'erreur commis sur chaque jeu de données : WEIGHT_ERROR*generalisation_error_onData+generalisation_error_onTest. On cherche à faire diminuer cette valeur en regroupant les fuilles comme auparavant.

En pratique de meilleurs résultats ont été observés en choisissant PRUNING_FACTOR assez grand (>6) et WEIGHT_ERROR plutôt faible, entre 0.1 et 0.3. C'est peut-être particulier aux données testées, car cela me semble un peu extrême.

IV) Traitement des valeurs manquantes

Fonctions **findIndex___()** dans Classif.cpp :

Lorsque l'on a déterminé qu'il faut séparer les données suivant la valeur d'un certain attribut, il se peut que des exemples n'aient pas cet attribut. La méthode utilisée est alors la suivante : s'il n'y a pas assez d'exemples à ce niveau de l'arbre, on ignore simplement l'exemple à la valeur manquante (voir les constantes NBMIN_REPBIN et NBMIN_REPDISC dans Arbre.h qui définissent justement ces seuils). Sinon, on détermine le nœud fils dans lequel il va aller en comptant le nombre d'éléments de la même classe que l'exemple à l'attribut manquant dans chaque nœud fils, puis en tirant une variable aléatoire (dont la plage de valeurs acceptée pour chaque fils est d'autant plus grande que celui-ci contient d'exemples de la bonne classe) qui donnera la destination.

V) Résultats expérimentaux :

Les fichiers SmallTree.jpg, MTree.jpg/MTree_miscError.jpg et BigTree.jpg contiennent respectivement les images des arbres générés sur les jeux de données iris.data, breast-cancer-wisconsin.data et adult.data, avec un élaguage raisonnable de l'arbre dans les deux premiers cas (PRUNING_FACTOR=5, PRUNING_FILTER=0.6, pas d'exemples test) et beaucoup dans le dernier cas (WEIGHT_ERROR=0.1, PRUNING_FILTER=0.7, présence d'exemples test; ces réglages permettent de très bonnes performances comparées à celles indiquées dans le fichier adult.names, mais avec ces deux coefficients à 0.5, on a seulement environ 0.01% d'erreurs de plus sur l'ensemble de test. Le taux d'erreur ne semble donc pas très sensible à ces paramètres sur ce jeu de données). Les heuristiques utilisées sont respectivements entropie, Gini/misclassification_error, Gini.

L'heuristique misclassification_error semble moins performante que les deux autres (voir MTree.jpg VS MTree_miscError.jpg); c'est peut-être particulier aux données testées, car dans le cas général l'entropie et l'indice de Gini ne sont pas parfaits non plus. Sur l'exemple du tic-tac-toe, misclassification_error donne des résultats plus généraux. Donc la « bonne » heuristique doit beaucoup dépendre du type de problème.

Tableau des résultats : Tr/Te : training/test errors; PRUNING_FACTOR=5.0, PRUNING_FILTER=0.6, NBMIN_REPDISC|BIN=7|10, WEIGHT_ERROR=0.2.

Heuristique : / Données :	Gini	Entropie	Misc_Error
Iris	Tr=0.04=6/150 Te=___	Tr=0.04=6/50 Te=___	Tr=0.06=9/150 Te=___
Tic-tac-toe	Tr=0.085=81/958 Te=___	Tr=0.084=80/958 Te=___	Tr=0.152=146/958 Te=___
Breast-cancer	Tr=0.033=23/699 Te=___	Tr=0.033=23/699 Te=___	Tr=0.033=23/699 Te=___
Adult	Tr=0.144 Te=0.134	Tr=0.143 Te=0.135	Tr=0.240 Te=0.235