Homework #4

RELEASE DATE: 04/28/2020
DUE DATE: 05/19/2020, **noon** on Gradescope/CEIBA

As directed below, you need to submit your code to the designated place on the course website.

Any form of cheating, lying or plagiarism will not be tolerated. Students can get zero scores and/or get negative scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconducts.

Discussions on course materials and homework solutions are encouraged. But you should write the final solutions alone and understand them fully. Books, notes, and Internet resources can be consulted, but not copied from.

Since everyone needs to write the final solutions alone, there is absolutely no need to lend your homework solutions and/or source codes to your classmates at any time. In order to maximize the level of fairness in this class, lending and borrowing homework solutions are both regarded as dishonest behaviors and will be punished according to the honesty policy.

Both English and Traditional Chinese are allowed for writing any part of your homework (if the compiler recognizes Traditional Chinese, of course). We do not accept any other languages. As for coding, either C or C++ or a mixture of them is allowed.

This homework set comes with 200 points and 20 bonus points. In general, every homework set of ours would come with a full credit of 200 points.

4.1 Trees

(1) (20%) Write down the pseudo code of an algorithm that solves Exercise R-7.15 of the textbook for any given pair of valid (preorder, inorder) strings. You can assume that each character shows up no more than once in each string. Your algorithm needs to run within $O(N^2)$ time complexity, where N is the length of the strings.

(2) (20%) Do Exercise R-7.22 of the textbook. You can use Euler traveral in the textbook or any other traversals that we have taught in the class.

(3) (20%) Do Exercise C-7.5 of the textbook for only the `inOrderNext` algorithm as well as describing the worst-case running time of this algorithm.

(4) (20%) Do Exercise C-7.7 of the textbook.

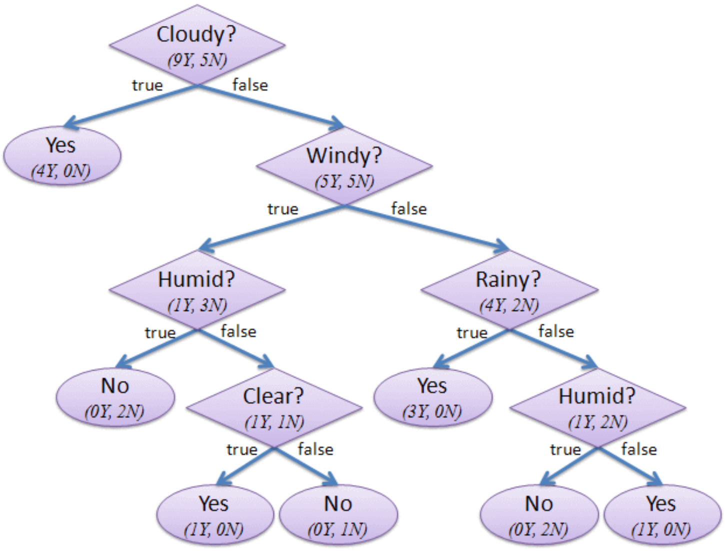
(5) (20%) Do Exercise C-7.33 of the textbook.

4.2 Decision Tree

In this problem, we will explore an application of trees in the area of Artificial Intelligence and Machine Learning. Decision Tree is one of the earliest tool for Machine Learning. The non-leaf nodes of a decision tree represent choices (factors) considered, while the leaf nodes represent final decisions under the choices. For simplicity, we will consider the trees with binary factors—i.e., binary decision trees. Such a tree is shown in Example 7.8 of the textbook.

For instance, the following tree[1] is a binary decision tree for deciding whether to play golf. If the sky is cloudy, we decide to play golf; if not, we check if it is windy—if so, we play golf only if it is not humid and the sky is clear. On the other hand, if it is not windy, we do not play golf only when the sky is not rainy but it is humid.

[1] Thanks to our TA emeritus Chun-Sung Ferng for drawing.



Such a decision tree is called a "classification tree." It classifies different $(sky, windy, humid)$ situations to decision categories $play? = \{yes, no\}$. The tree is not arbitrarily formed. In fact, it is automatically learned by a program from a bunch of given examples. In other words, you can "teach" the program with the examples. For instance, consider the following examples.

sky	windy	humid	play?
clear	true	true	no
clear	true	false	yes
rainy	true	false	no
rainy	false	true	yes
clear	false	true	no
clear	false	false	yes
cloudy	false	true	yes
clear	false	true	no
cloudy	true	false	yes
cloudy	true	true	yes
rainy	false	true	yes
cloudy	false	true	yes
rainy	true	true	no
rainy	false	true	yes

The decision tree can be taught with the examples in a top-down recursive way. First, we need to find the root branch. There are 9 yes and 5 no (9Y5N) in the examples above. If we consider a factor of "sky being clear or not", we can separate the 14 examples to two branches: 2Y3N for the clear branch and 7Y2N otherwise; if we consider a factor of "sky being cloudy or not", we can separate the 14 examples to two branches: 4Y0N for the cloudy branch and 5Y5N otherwise. We can continue checking possible branches.

One heuristic for making a good branching choice is to check the total confusion after branching. The confusion of a mixture of aYbN is defined as

$$\text{confusion}(a, b) = 2 \min(\frac{a}{a+b}, \frac{b}{a+b}).$$

and the total confusion after branching from $(c+e)$Y$(d+f)$N to cYdN and eYfN is

$$\text{total}(c, d, e, f) = \frac{c+d}{c+d+e+f}\text{confusion}(c, d) + \frac{e+f}{c+d+e+f}\text{confusion}(e, f).$$

For instance, the total confusion after branching by "sky is clear or not" is

$$\frac{5}{14} \cdot 2 \cdot \min(\frac{2}{5}, \frac{3}{5}) + \frac{9}{14} \cdot 2 \cdot \min(\frac{7}{9}, \frac{2}{9}).$$

The total confusion after branching by "sky is cloudy or not" is

$$\frac{4}{14} \cdot 2 \cdot \min(\frac{4}{4}, \frac{0}{4}) + \frac{10}{14} \cdot 2 \cdot \min(\frac{5}{10}, \frac{5}{10}).$$

The heuristic tries to find a branch such that the total confusion is the smallest, with ties arbitrarily broken.

Now, after finding a good branch for the root, we separate the examples to two subsets: one for the left-child of the root, one for the right-child of the root. The same branching strategy can be applied to the two subsets to form the two sub-trees and the tree building continues recursively.

Recursively? What is the termination condition, then? Well, you do not need to branch if all the examples are the same and you cannot branch anymore, or if the confusion without branching is readily less than or equal to a value ϵ. For instance, if $\epsilon = 0.3$ and there is 9Y1N in a subset, its

$$\text{confusion}(9, 1) = 2 \cdot \min(\frac{9}{10}, \frac{1}{10}) = \frac{2}{10} \leq 0.3$$

Then we do not need to branch anymore. In such a case we declare a leaf with the final decision with the majority of the examples (such as the Y in the 9Y1N case), with ties arbitrarily broken. The simple decision tree algorithm is listed as follows:

```
given ε as the parameter
Decision-Tree(examples)
if confusion in the examples ≤ ε or cannot branch anymore then
  build and return a leaf node with the associated final decision
else
  find a branch such that the total confusion is smallest, store the branch in the root of the tree
  separate examples to two subsets, one for the left-child and one for the right-child
  set the left-subtree to be DecisionTree(example subset for the left-child)
  set the right-subtree to be DecisionTree(example subset for the right-child)
  return the tree
end if
```

The branch we discussed are on discrete factors. We can also branch on numerical (continuous) factors by setting a proper threshold. For instance, a numerical factor may be the *temperature* and a branch may be "is *temperature* greater than t?" The best threshold t can be found by "cleverly" searching all possible thresholds.

For instance, consider branching on one numerical variable with M examples. Consider a simple case where the values of the numerical variable on those examples are $v_1, v_2, \ldots, v_M$ with $v_1 < v_2 < \ldots < v_M$— if the values are not sorted or not unique, you can always sort them and remove the duplicates to get sorted and unique values. Trivially, there are only $M + 1$ possible "regions" of thresholds t, where all the thresholds within the same region are equivalent—that is, they separate the M examples to two subsets in identical ways. Consider choosing the thresholds among $\{\frac{v_1+v_2}{2}, \frac{v_2+v_3}{2}, \ldots, \frac{v_{M-1}+v_M}{2}\}$. A naïve algorithm for making the best choice is to evaluate the total confusion (which is $O(M)$) with each threshold, needing a total time of $O(M^2)$. The naïve algorithm does not use the property that the v_m values are sorted, by the way.

In this problem, we ask you to implement such a program that can be taught with examples of numerical variables and produces the binary decision tree. One interesting thing about binary decision trees is that you can output the tree as some C code `if(...){...} else{...}`. That is, after you teach your program, it can automatically produce another program that can make future decisions.

(1) (Bonus 20%) Using the property that the v_m values are sorted, describe an $O(M)$ algorithm to calculate the best threshold. Note that if you combine the $O(M)$ algorithm with a sorting algorithm, which can be done in $O(M \log M)$ (we taught you heap sort to do so, but we will tell you some other sorting algorithms later as well), you get an $O(M \log M)$ time algorithm for picking the best branching threshold.

(2) (40%) Implement the decision tree algorithm. Your program should read the input examples that contain numerical values, and print out a piece of C code representing the decision tree. Your program should be named `tree`, take the data file as the first argument, and ϵ as the second argument. For instance, `./tree heart_scale 0.1` learns a decision tree from the data file `heart_scale` with $\epsilon = 0.1$.

The data file is assumed be of the famous LIBSVM sparse format.[2] Each line of the data file represents an example of the form

```
label sparse_array
```

where `label` is either `+1` (indicating YES) or `-1` (indicating NO), and `sparse_array` represents an C-style dense array {`0, 4, 0, 0, 3, 2`} by `1:4 4:3 5:2`

You can read the sparse array and store it in whatever way (sparse or dense) you choose in your program. A friendly hint is that it might be simpler to internally use the dense array.

Please output your tree as a function in C/C++ language. The function must follow this interface:

```
int tree_predict(double *attr);
```

The only argument is a double array which contains the factors of one example in the same format as input. This function should return the label prediction of the example (1 or -1 for `heart_scale`, for instance). After the `Makefile.ta` redirects your standard output to `tree_pred_func.cpp`, you can compile and run the provided `tree_predictor.cpp` to check how good your decision tree is (see README). For example, your `tree_pred_func.cpp` should look like:

```
int tree_predict(double *attr){
  if(attr[3] > 5){
    return 1;
  }
  else{
    return -1;
  }
}
```

(3) (20%) Illustrate (ideally with drawing) the internal data structure you use to represent the decision tree in your memory. Please be as precise as possible.

(4) (20%) Construct your own data set with at least 2 numerical factors and at least 8 examples. Teach your program to make a decision tree of at least 2 levels with this data set. List the examples as well as draw the tree found. Briefly explain the tree.

(5) (30%) Take $\epsilon = 0$. If, instead of branching by minimizing the total confusion, you do a "random branching" by randomly pick one numerical factor and then randomly choose one θ within $\{\frac{v_1+v_2}{2}, \frac{v_2+v_3}{2}, \ldots, \frac{v_{M-1}+v_M}{2}\}$ of the numerical factor. Due to randomness you may end up revisiting some factors you have considered in the top levels. You can also do "maximum total confusion branching" by branching with the maximum confusion factor instead of the minimum confusion one. Using the data set in

`https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/heart_scale` .

Compare the depth of the tree you get from "minimum total confusion branching" to the average depth of the trees you get from "random branching" (over 1000 random runs) to the depth of the tree you get from "maximum total confusion branching." Briefly state your findings.

[2]LIBSVM is a world-famous machine learning algorithm developed by Prof. Chih-Jen Lin's lab in our department.

Submission File

Please submit your written part of the homework to Gradescope before the deadline. Also, you need to upload your coding part as a single ZIP compressed file to CEIBA before the deadline. The zip file should contain no directories and only the following items:

- all the source code and your own `Makefile` (different from the TAs' `Makefile.ta`) such that `make tree` would generate a program named `tree`, which reads examples and outputs a piece of code to stdout for the decision tree
- an optional `README`, anything you want the TAs to read before grading your code

Please make sure that your code can be compiled and run with the Makefile on CSIE R217 linux machines. Otherwise your program "fails" its most basic test and can result in ZERO!