CS 414, Fall 2010
Project 0 (20% of final grade)
Due October 4th, 6am

Start Early

1. This assignment will probably take you longer than you think it will. I've given you over three weeks to do this – *do not* expect to bang it out in a weekend and still do well.
2. If you submit this project early, or parts of the project early, I will give you comments within 3 working days, which do not count towards your grade. Thus, if you finish a week early, I can look over the entire project and tell you what, if anything, needs to be changed in order to earn full marks.
3. I may assign additional homework and reading before October 4th – it will not be time intensive, but it will be an additional TODO on your stack.

Inducing Decision Trees - ID3

In this project you will investigate Quinlan's ID3 learning algorithm (described in Mitchell's Chapter 3).

Your assignment has multiple parts. First, you will implement ID3, including a stochastic pruning algorithm. Second, you will run some experiments with ID3. Third, you will plot results in ROC and recall-precision curves.

It is acceptable to look at the Java code available from WEKA. However, it is *not* acceptable to share code with anyone, including current students. You may talk with other students in the class, but discussions should be at a high level (i.e., not implementation details) and you must note the names of students with whom you discussed the project on your write-up.

Part 1: Implementing ID3

Implement the algorithm in Mitchell's Table 3.1, augmenting it as explained below. You need not deal with continuous features on this homework.
Your program should be able to handle data sets that have up to 10 discrete outputs (e.g., 0 or 1 for False/True problems, or 'A,' 'B,' 'C,' 'D,' 'E,' 'F' for a grade prediction problem). Inputs will consist of up to ten features, each of which will have up to en values. For instance, consider the following dataset.

1, 1, 1, 1, 1, 0
1, 1, 0, 1, 0, 1
1, 1, 0, 1, 1, 2
1, 1, 1, 1, 0, 0
1, 0, 1, 1, 1, 1
…
The data is in the format:

```
Output 1,0
f1 discrete 1,0
f2 discrete 1,0
f3 discrete 1,0
f4 discrete 1,0
f5 discrete 2,1,0
```

For instance, the third instance in the data set specifies that the input <1, 0, 1, 1, 2> is True (i.e., has class value 1).

Please use the car dataset (four classes: unacc, acc, good, vgood), found by clicking "Data Folder" here:
http://archive.ics.uci.edu/ml/datasets/Car+Evaluation
and the Monk dataset (ignore the final feature, as it is a unique identifier):
http://archive.ics.uci.edu/ml/datasets/MONK%27s+Problems
I reserve the right to test your algorithm on other UIC datasets that conform to the problem specifications (<=10 discrete outputs, <=10 features, each with <= 10 categorical values).

Show the performance of your algorithm when training on `monks-1.train` and testing on `monks-1.test` and when using 10-fold cross-validation on the `car.data` set (i.e., split the data set into 90% training and 10% test, and repeat the process 10 times so that each 10% "chunk" of data is used as a test set).

Part 2a: Handling an Additional Splitting Function

The crucial step in the ID3 algorithm involves choosing which feature to use as the next node in the decision tree. Besides using Quinlan's *info_gain* measure (Equation 3.4 in Mitchell), also implement the following (not as intelligent) alternative for use as an experimental control:

random

(uniformly) randomly choosing one of the remaining features

You should implement a function called *RUN_ID3(String namesFile, String trainsetFile, String splittingFunction)*, whose arguments are, in order, the file describing the examples, the trainset to use, and the splitting function to use. (Either *info_gain,* or *random*)

You should also write a function *DUMP_TREE()* that prints, in some reasonable fashion, the most recently learned decision tree and the function *REPORT_TREE_SIZE()* which reports the number of interior and leaf nodes (and their sum) in the most recently learned decision tree. To prevent wasting paper, limit the maximum depth of printed trees to THREE interior nodes (including the root); wherever the tree is deeper than that, simply print something like "there are *N* interior nodes below this one; *P* positive and *N* negative training examples reached this node."

Part 2b: Avoiding Overfitting

A decision tree that correctly classifies all the training examples might be too complicated; some subtree might generalize better to future examples. In this homework, you will implement a *stochastic* search for a simpler tree, according to the following (call this variant *pruned*):

1. Given a training set, create a training set and a *tuning* (or "pruning") set; place 20% of the examples in the tuning set.
2. Create a tree that fully fits the train' set; use info_gain as the scoring function for features. Call it *Tree*. Set *scoreOfBestTreeFound* to the score of *Tree* on the tuning set.
3. Number all the *interior* nodes from 1 to *N* (e.g., using a "preorder" traversal; however the precise method for numbering nodes doesn't matter, as long as each interior node is counted once and only once).

4. *L* times do the following
    - Make a *copy* of *Tree*. Call it *CopiedTree*.
    - Uniformly pick a random number, *R*, between 1 and *K*.
    - *R* times, uniformly pick a random number, *D*, between 1 and *N*. Mark node *D* in *CopiedTree* as "to be deleted."
    - Make a new copy of *CopiedTree*, this time pruning the tree whenever you encounter a "to be deleted" node. Replace the deleted node by the *majority* category for the subtree rooted at this node. *CopiedandPrunedTree* should point to this pruned tree.
    - At this point, you will have a random subtree of *Tree*. Score it on the tuning set, and keep track of the best tree found.
5. Return the best tree found (on the *tune* set).

Let *L*=100 and *K*=5 (arbitrarily chosen values; if your code runs too slowly, feel free to adjust *L* but be sure to document this in your HW writeup - also feel free to use *higher* values for *L*).

Part 2c: Making Decisions

You are to also create the function *CATEGORIZE(testsetFile)*. This function takes as input a string indicating which testset to use. For each example in this list, the code should traverse the most recently learned decision tree, using the feature values of the instance, until a leaf is reached. The decision at the leaf node should be compared to the correct answer and errors counted.

CATEGORIZE should print out the names (e.g., TEST_POS1, TEST_NEG1, etc) of the miscategorized examples and report the overall error rate.

During grading I will test *RUN_ID3*, *DUMP_TREE*, *REPORT_TREE_SIZE*, and *CATEGORIZE*. When called via

`HW1 task.names train_examples.data test_examples.data`

have your main function do the following sequence of training runs:

```
RUN_ID3(task.names, train_examples.data, "info_gain");
DUMP_TREE();
REPORT_TREE_SIZE();
CATEGORIZE(test_examples.data);

RUN_ID3(task.names, train_examples.data, "pruned");
DUMP_TREE();
REPORT_TREE_SIZE();
CATEGORIZE(test_examples.data);

RUN_ID3(task.names, train_examples.data, "random");
DUMP_TREE();
REPORT_TREE_SIZE();
CATEGORIZE(test_examples.data);
```

Part 3a: Measuring Generalization

Investigate how well the ID3 algorithm works on *new* (i.e., "testset") instances as a function of the splitting technique used. Provide a single figure, with error bars (the 95% confidence intervals), that reports testset error rates as a function of the splitting strategy used. For the *random* function, average your result over at least 30 runs. Compute these results using a single train/test split of 80%/20%. Discuss your results, including whether or not your tree outperforms a naïve classifier. Specifically, compare to a classifier that always outputs the most common output in the training set (e.g., for the above data set, this "naïve classifier" would always output True).

Part 3b: Investigating the Correlation between Tree Size and Tree Accuracy

Draw a single figure whose *X*-axis is the size of the tree learned on the *training* data and whose *Y*-axis is the error rate of the tree on the *testing* data. Plot all the results for the experiments you ran in Part 2a on your first train/test fold, clearly indicating the splitting function that produced each. Discuss your experimental results. How well do they support the "Occam's Razor" hypothesis?

Part 3c: ROC and Precision-Recall Curves

Plot ROC curves - false positives (*X*-axis) versus true positives (*Y*-axis) - for (a) your info_gain tree without pruning and (b) your info_gain tree with pruning. Plot BOTH data on the same ROC graph for ease of visual comparison.
Repeat the above, but this time draw recall-precision curves.
Discuss the results you plotted in your graphs. You may use any method you wish to draw the curves, including "by hand."

Requirements

Email, in a single .pdf:

1. A copy of all of the (commented) code you write.
2. Neatly plot the data requested in the experimental sections. Be sure to label all of your axes.
3. Type your answers to the questions posed.
4. Unless otherwise noted, include a printout of your code's output for the experiments you are asked to run; however, use your judgment - if this output gets too big, feel free to delete unimportant portions.
5. A link to any additional the data set(s) you used (if applicable).

Additionally please send:

1. A binary of your program.
2. A short explanation of how to run your program (if it does not confirm exactly to the guidelines above).
3. The name of the cs department computer I should run your binary on (if it matters).