EECS 349 Problem Set 2

Due 11:59PM Monday, May 9

Overview

In this assignment you will work in groups of 2 or 3 to implement a decision tree learning algorithm and apply it to a synthetic dataset. You will also implement a pruning strategy in your algorithm. You will be given labeled training data, from which you will generate a model. You will be given labeled validation data, for which you will report your model's accuracy. You will also be given unlabeled test data for which you will generate predictions.

Submission Instructions

Each student should turn in their own copy of the homework.   You can work together on the code, but you should write-up your answers to the questions independently.   Here is how James Bond would submit the homework. Please adjust for your own name:

1. Create a single PDF file with your answers to the questions below. Name this file `PS2.pdf`.
2. Create a directory (i.e. a folder) named `PS2.code` that contains your source code.
3. Run your code on the test file and output a file in the same format, but with your predicted labels in the last column. Name this file `PS2.csv`.
4. Create a ZIP file named `PS2.zip` containing:
    - `PS2.pdf`
    - `PS2.code` (directory)
    - `PS2.csv`
5. **Ensure that the zip file contains all of your source code.** You may have to tell the ZIP utility explicitly to include the contents of the subdirectory containing your code.
6. Turn in your code under Problem Set 2 in Canvas.

Download the Homework Zip

Start by downloading PS2.code.zip, which contains all the code and data you'll need for the assignment.

The primary data sets include:

- btrain.csv : the training set
- bvalidate.csv: the validation set
- btest.csv : the test set

We have also included smaller test_*.csv versions of these data for convenience in testing your code.

The dataset is from a synthesized (and therefore fictitious) database of 70,000 baseball games played between two rival teams. Each line of text contains the following information about the game:

- Winning percentage of one team : numeric
- Winning percentage of the opposing team : numeric
- Weather : nominal
- Temperature: numeric
- # of injured players on one team: numeric
- # of injured players on the other team: numeric
- Starting pitcher: nominal
- Opposing starting pitcher: nominal
- Days since one team's last game: numeric
- Days since the other team's last game: numeric
- Whether it's a home or away game (for the first team): nominal
- Run differential for first team: numeric
- Run differential for second team: numeric
- Winner : binary (0 or 1)

The class label is given by the *winner* attribute. This is a *binary classification* problem with *numeric* and *nominal* attributes. Some attribute values are missing (as might happen in a real-world scenario). These values are indicated by a "?" in the file. In the test files the class labels are missing, and these missing labels are also indicated by a "?". The test sets are all drawn from the same distribution as the training and validation sets.

If you want, you can imagine that the task is to predict the winner of a baseball game that will be played under the conditions described by each line. However, it's not advised to read too much into the meaning of each attribute, since the data is fictional.

Implementation

For this assignment you will *implement* a decision tree algorithm starting from Python starter code. You should not use Weka or any other existing framework for generating decision trees, instead, you should complete the individual Python methods in the starter code we have provided.

Note: your algorithm must handle missing attributes, and must use some kind of pruning strategy. The exact choice of pruning strategy is up to you.

Pruning

Add a pruning strategy to your decision tree algorithm. You are free to choose the pruning strategy, but you SHOULD use the validation set for pruning. Note that you don't, for example, necessarily need to iteratively greedily select the one *best* node to prune, as this might be computationally prohibitive. So feel free to choose an approximation (e.g. any node that improves accuracy on the validation set).

Be sure you can run your algorithm both with and without pruning.

Learning curve

As discussed in class, in general, the more training data your algorithm has available, the better it will perform. To illustrate this, you'll be creating a graph showing the performance of your decision trees on the validation set while restricting its training set to .1x, .2x, …, .9x, 1.0x the size of the training set, with finer gradations if desired. For each amount of training data (except the last, which uses all the data), take the average over multiple runs, using a different subset of the training data each time.

Getting Started with the Code

You want to use Python 2.7, and make sure you have the matplotlib library installed. Your first step should be to unpack the starter code in PS2.code.zip. Then, from the PS2.code directory, run autograder.py. You'll see that the tests fail, which is exactly as expected. Your goal is to go through the individual modules (in the "modules" folder) and implement each method that currently has a body consisting of **pass**. Implement those according to the specification given in the comments.

As you're writing methods, you can check many, but not all, of them for correctness using the script in auto_grader.py. There are a handful of methods (including the pruning and graphing methods) that you can write in any appropriate manner you choose, so you will need to write your own tests for these methods. Once your implementation is complete, you can run decision_tree_driver.py to complete the experiments needed to answer the questions below.

Where should I Start?

You can choose, but starting in the Node file and then moving to ID3 is one good option.

Data Format

The code represents the `data_set` as a list of examples. Each example is a list. In each example, the first entry represents the output (winner or not winner), and the other attribute values follow. The `parse` function has already been written for you to read the data files we've given you into this format.

What is ID3?

ID3 is your main training function. It takes in a `data_set` of examples, `attribute_metadata` giving information about the attributes (names, whether or not they are nominal), `numeric_split_counts` (discussed under "Numeric Attributes" below) and a depth limit. It returns a trained tree, which is a `Node` object (see node.py). The basic recursive algorithm to use here is given in the lecture notes, although your implementation of ID3 will include enhancements that go beyond the original ID3 algorithm: you must handle numeric attributes as well as missing data.

Choosing Attributes to Split on

For this assignment, for choosing attributes we'll use a slightly improved metric over the standard information gain metric discussed in class. Specifically, you should use the Information Gain Ratio. It normalizes the information gain (IG) expression we learned in class to adjust for the fact that attributes with more distinct values have an "unfair" advantage in IG.

Numeric Attributes

Then, for numeric attributes, we'll be using a simple discretization scheme. Instead of testing all possible numeric split points, we will only test every *i*th one, where *i* is an argument (`steps`) to the relevant function (`gain_ratio_numeric`).

What value should I use for `steps`? Make `steps`=1 by default. This should allow all tests in auto_grader.py to pass. You can temporarily increase `steps` when you execute your runs on the larger data sets, so that your code runs more quickly. But make sure that your turned-in code uses `steps` = 1 by default, that is what we will be expect when grading your assignment.

What is `numerical_split_counts`? That's an array (of length = number of attributes) that gives the maximum number of times you should split on each numeric attribute down any path in the tree. For nominal attributes, the array entries can be ignored. For numeric attributes, you should not return the attribute from `pick_best_attributes` if its corresponding entry in `numeric_split_counts` is zero. Likewise, in ID3, the tree you return must not contain more than `numeric_split_counts[i]` splits of attribute i on any given path from the root to a leaf.

Can I Add Functions?

Yes, and it may be helpful to do so. Just make sure that the interfaces to the functions that are tested in auto_grader.py remain unchanged – the tests in auto_grader.py should all pass for you to get full credit on the assignment.

Which Functions will I be Graded on?

We will automatically grade all the functions that are currently tested in auto_grader.py (we will test on some additional examples beyond those in auto_grader.py). We will manually examine the other key functions.

My code takes forever to train on the large data sets!

You have three knobs for ID3 that allow you to increase efficiency, potentially at the cost of accuracy. Try adjusting `steps`, `numeric_split_counts`, and `depth`.

What is Disjunctive Normal Form?

Say we have a tree that looks like:

```
      A
A=0/    \A=1
  /      \
 1        B
     B=0/ \B=1
       /   \
      1     0
```

Disjunctive Normal Form (DNF) expresses a boolean expression as an "or of ands".   So for this tree the DNF is: `A=0 v (A=1 ^ B=0)`.

Where v represents OR and ^ represents AND.

Questions

Put answers to the following questions in a PDF file, as described in the submission instructions.

Answer concisely. You may include pseudocode or short fragments of actual code if it helps to answer the question. However, please keep the answer document self-contained. It should not be necessary to look at your source files to understand your answers.

1. How did you handle missing attributes in examples?
2. Apply your algorithm to the training set, without pruning. Print out a Boolean formula in disjunctive normal form that corresponds to the *unpruned* tree learned from the training set. For the DNF assume that group label "1" refers to the positive examples.   NOTE: if you find your tree is cumbersome to print in full, you may restrict your print-out to only 16 leaf nodes.
3. Explain in English one of the rules in this (unpruned) tree.
4. How did you implement pruning?
5. Apply your algorithm to the training set, with pruning. Print out a Boolean formula in disjunctive normal form that corresponds to the *pruned* tree learned from the training set.
6. What is the difference in size (number of splits) between the pruned and unpruned trees?
7. Test the unpruned and pruned trees on the validation set. What are the accuracies of each tree? Explain the difference, if any.
8. Create learning curve graphs for both unpruned and pruned trees. Is there a difference between the two graphs?
9. Which tree do you think will perform better on the unlabeled test set? Why? Run this tree on the test file and submit your predictions as described in the submission instructions.
10. Which members of the group worked on which parts of the assignment?

11. BONUS: This assignment used Information Gain Ratio instead of Information Gain (IG) to pick attributes to split on, which is expected to boost accuracy over IG. We also used a limited step side for numeric attributes instead of testing all possible attributes as split points. Were these good model selections? Try using plain IG and see if this impacts validation set accuracy. Likewise, try testing all numeric split points (doing so efficiently will probably require writing new code, rather than just setting `steps` = 1), and evaluate whether this improves validation set accuracy.

Grading Breakdown

This assignment is worth 15 points, broken down as follows:

- Code completion and correctness
  - 5 points
- Disjunctive Normal Form (Questions 2, 5)
  - 2 points
- Pruning (Questions 4-5)
  - 3 points
- Learning Curve (Question 8)
  - 1 point
- Output of Algorithm (Question 9)
  - 3 points
- Who worked on what (Question 10)
  - 1 point
- Bonus (Question 11)
  - 2 points

It is possible to get up to 12 points of credit without implementing pruning.