15-681: Machine Learning

Homework 3: Decision Trees

Due: Thursday, September 26 1996

1. Introduction

This assignment asks you to understand, run, and modify the decision tree learning algorithm discussed in Chapter 3. You will be given an implementation of the decision tree learning algorithm and a set of training data describing German loan applicants. You will use the program to create and test decision trees that specify whether or not applicants are granted loans.

Direct any questions you may have to the class TA (Scott Davies, `scottd@cs.cmu.edu`). You should check the class web page periodically for updates.

All code and data for this assignment may be found in the directory `/afs/cs/project/learn-5/mlhw3`.

1.1. The Algorithm

The file `decision.tree.c` contains the ID3 decision tree learning algorithm, which is similar to the algorithm described in Table 3.1. You should review this code and attempt to gain a basic understanding of what it does. The ID3 function is the primary function of this assignment and uses many supporting functions found in other files. For a full listing of all the functions used by the ID3 function, look in `decision.tree.h`.

The algorithm contained in `decision.tree.c` is not identical to the one described by the book. Most importantly, it is not a recursive function. Rather, it uses a queue to perform a breadth-first expansion of the tree. However, the ID3 function does primarily follow the algorithm described by the book and the book can be used as a valuable reference in understanding the code.

1.2. The Data

The file `credit.ssv` contains 600 examples of loan applicants, half of whom are granted a loan, half of whom are not. The file is in the space-separated value format. Each element of a single example is separated by a space and each example is separated by a newline. The first line of the file describes the total number of examples in the file and the number of attributes in each example. The second line contains a listing of the names of all the attributes. The third line contains a single string (no spaces) describing whether or not the attributes are *c*ontinuous or *d*iscrete. The fourth and later lines describe the examples.

For further information on the data set, you should consult the file `credit.doc`, which describes all of the attributes found in the credit data set and contains other useful information about the data.

1.3. Utility Program

In addition to the main code (whose usage is described in section 3.1.1), you are provided with a main utility program (written in perl) which simplifies the process of turning raw data into usable information.

- `tree_script` is the main utility program you will use for this assignment. It will run the main executable, gather all the data from the executable and separate the data into a number of files which can easily be used for viewing the actual tree and viewing a plotting of the accuracy of the tree as it is built and pruned. See section 3.1.2 for a full description of usage.

1.4. The Decision Tree Code

C/C++ code is provided for the ID3 training algorithm and the reduced-error pruning algorithm. Additionally, the German credit data and the perl utility script are provided with this assignment.

The code is located in `/afs/cs/project/learn-5/mlhw3`. To compile this code, you should copy all of the code to your local directory and type `make re`. This will compile all necessary code to run the ID3 algorithm and perform reduced-error pruning on the created tree. It will produce an executable named `re`.

Most of the changes you will make to the program will be done through `tree_script`. When you do make changes to the actual C code, it will primarily be done through the files which contain the main function and the id3 function: `main.re.c` and `decision.tree.c`.

2. The Assignment

2.1. Part I

1. Edit `tree_script`, changing the size of the training set to 400, the size of the testing set to 200, and the size of the pruning set to 0. Also, ensure that the target attribute is 'Acceptable?' (number 20). With your changes complete, run `tree_script`.

2. Once the program has completed, view `train.gnu` with `gnuplot` by issuing the load command: `load "train.gnu"`. Examine the graph, recording and explaining any trends you discover. After doing this once, return to editing `tree_script` and change the random seed before rerunning the program. Again, report your findings and explain any general trends. Does training actually do anything for the test set? How much does it seem to help? Remember, the test set is kept completely exclusive of the training, so it is a pretty good gauge for how well the tree would classify additional data brought in from the outside.

   If you wish to look at the actual data which makes up the graph, it can be found in files with the extension `.data`. In addition, if you wish to keep a running record of the performance of certain algorithms, the perl script automatically writes the final accuracy value for the 'Test Set' in a file called `past.data` after each execution.

   After running `tree_script`, you should examine the file `tree.html`. This contains an HTML print out of the tree which would be useful for understanding what the algorithm achieves. It may be viewed with any web browser which supports tables. How sensitive is the structure of the tree to changes in the random number seed? Does the tree's top-level structure stay pretty much the same, or do even the topmost nodes change? Provide specific examples taken from the actual trees you wind up with.

3. Next, you should return to the `tree_script` file and add in a pruning set. Set 1/2 of the 600 examples for training, 1/4 for pruning and 1/4 for testing. Incorporate the training set into the pruning set by adding a '+b' after the size of the prune set (`$PRUNE_SET = '150+b';`). Once you have done this, run the script and record your observations.

4. Now, redo the previous test, but this time leave the training data out of the prune set (`$PRUNE_SET = '150';`). What differences does this seem to make?

5. Now, redo the previous test (keeping the training data out of the prune set), but vary the number of training examples (leave the prune set and test set untouched). Run the program with training sets of 200, 100, 50 and 25 examples. You should also change the random seed a few times to get a representative sample of the algorithm's performance. What happens as the training set size decreases? Save the graphs and/or data sets that seem to support your theories the best — along with those which refute them!

   The utility script will output a number of data files each time you run it. It uses the same output names on every run, so to be able to save graphs, you will need to assign the data files different names once they have been run. Additionally, you will need to change the contents of the gnuplot file to reflect the changes of the data names.

2.2. Part II

Now it's your turn to have some fun. It's time to kick off those restraining belts and dive into the meat of the assignment. Choose any one of the following suggestions, or create your own (if you decide to create your own, get the approval of your TA before starting work). Please turn in a short write up of what you decided to do, your results and any conclusions you draw (2-4 pages should be sufficient).

- Create your own pruning function that will accept as input a pointer to the root of a decision tree, a target_attribute and an example set. You should make sure to use the function `print_sets()` to allow the utility script to graph accuracy values as your function progresses. Look at `re.prune.c` for examples. Possibilities to think about include chi-squared tests, or — if you're feeling adventurous — a pruner that transforms the tree into a set of rules and then "prunes" the rules by eliminating variables from them.

- Try different functions for the selection of attributes while building the tree. Currently, `expected_entropy()` uses the information gain algorithm to determine the most appropriate attribute. You should consult the book for a listing of functions to try. You may even want to try a random function to see whether this function actually has any effect on performance. To implement this, you will need to create your own function, similar to the `expected_entropy()` function found in `decision.tree.c`.

3. Documentation

The code for this assignment is broken into several parts:

- `decision.tree.c`: the id3 core algorithm. All the functions you need to run id3 on a set of discrete examples.

- `utility.c`: support functions. These functions contain basic operations used by other higher functions. **You will probably not need to modify any code in this module to complete the assignment.**

- `output.c`: output functions. Functions that print out the decision trees and figure out the accuracy of these trees. **You will probably not need to modify any code in this module to complete the assignment.**

- `continuous.c`: all the support functions to run id3 on continuous example sets. **You will probably not need to modify any code in this module to complete the assignment.**

- `decision.tree.h`: the header file for all of the above C functions. It includes declarations for all the functions in addition to most of the structure type definitions used in this code.
- `args.c`: the command line parsing code. **You will probably not need to modify any code in this module to complete the assignment.**
- `stack.value.c`, `stack.att.c`: similar stack classes. **You will probably not need to modify any code in this module to complete the assignment.**
- `re.prune.c`, `re.prune.h`: the reduction-error pruning functions.
- `main.re.c`: the top level program which builds the decision tree and prunes it with reduction-error pruning.
- `tree_script`: a perl script that runs the id3 program and turns the output into a useful, easy to understand form. See Section 3.1.2 for information on running this program.

3.1. `re, rp, and tree_script`

3.1.1. Running `re`

The main executables, `re`, can be compiled with the command `make re`. Running `re` creates a decision tree using the id3 algorithm and then performs reduced-error pruning on it. `re` uses the following command line options:

-h <help> - in case you don't have this document handy, you can get a quick rundown of the available options using this switch.

-f <filename> - this specifies the data file you wish to use. Example: `-f credit.ssv`.

-b <training (or building) set> - this is a specification of which examples you want to put in the training set. The specification may either be a range or a set size. If it is a range, it may either be in the format 'sss-eee' or 'sss - eee' where 'sss' is the beginning example and 'eee' is the ending example. If it is a set size, the specified number of examples will be chosen at random from the pool of possible examples. Sets which are specified in this manner will not be allowed to overlap examples and thus will be mutually exclusive. However, if you wish have one set include the examples from another set, you may add a '+c', where 'c' references one of the sets (b, p, t or e). Thus -b 50+p would create a training set which includes all of the examples in the pruning set, in addition to 50 random examples which are not included in any other set.

-p <pruning set> - this option specifies the pruning set. See the '-b' option for details on how to specify the set. This is the set used to make accuracy comparisons while pruning the tree.

`-t <test set>` - this option specifies the test set. See the '-b' option for details on how to specify the set. The testing set is generally used as a baseline to measure the true accuracy of training and pruning the decision tree.

`-e <extra set>` - this option specifies the extra set. See the '-b' option for details on how to specify the set. The extra may be used for any purpose, as the current code has no reliance upon it.

`-a <attribute number>` - this will indicate which of the attributes you wish to have as the target attribute. This must be an integer and usually turns out to be the last attribute in the set. Note that the attributes are numbered from zero, so in the case of the `credit.ssv` data, which has 21 attributes, the one that indicates loan approval will be number 20.

`-r <random seed>` - an integer which will be used to initialize the random number generator. Using the same seed on unchanged code will give you the same results each time. This will allow you to repeat results if data is lost or misplaced.

3.1.2. Running `tree_script`

In addition to the C/C++ code, you are provided with a perl script which can be used to automate the collection and presentation of data. The program is very configurable and has a number of configurable options imbedded in the file. Before running the script, you should open the file with your favorite editor and read through the first few sections.

Upon execution, the program will run the main executable with the specified options. All of the output from the main executable will be stored in a temporary file. From this, the script will first extract any HTML-ized trees it finds in the data file (which is created by a call to `print_html_tree()`). Next, it will extract all the accuracy data and put all the information into individual data files. Once this is done, it will create gnuplot files which use the data files as their sources. The names of these files will be displayed as the script runs.

To be able to view the plots, you will need to run `gnuplot` and then issue the command `load "plot.file.gnu"` where `plot.file.gnu` is one of the file names displayed by the script.

To be able to view the HTML-ized trees, you will need a web browser which is capable of decoding HTML v2.0 (tables are used to display the tree). Netscape version 1.1 or later will do just fine.

Additionally, the script will also keep track of final accuracy values of past runs. It will store these values in a file named `past.data`.

3.2. Structures

3.2.1. Decision Tree Structure

The decision tree consists of two structures: `value_branch` and `attribute_node`. Below is a definition of these structres:

```
struct attribute_node {
  char * attribute;
  partition_s * att_info;
  partition_s * target_info;
  value_branch * head;
  value_branch * parent;
};

struct value_branch {
  char * value;
  value_branch * next;
  attribute_node * parent;
  attribute_node * attribute;
};
```

For this program, the decision tree is represented by a conglomeration of `attribute_node` structs and `value_branch` structs.

An `attribute_node` represents a node in the tree. It stores the name of the attributes which is used to classify examples, along with a pointer to the branches it connects to. `parent` is the branch immediately above the node. `head` is the beginning of a linked list of branches which extend below the node. `att_info` is a partitioning of the training examples according to the attribute specified by this node. `target_info` is a partitioning of a set of examples according to the target attribute. The `target_info` variable is only used by `apply_tree()` and tree printing functions.

Each `value_branch` represents one classification of the `attribute` stored in the preceeding `attribute_node`. The classification type is stored in `value`. `attribute` is a pointer to the `attribute_node` which is connected immeditately below. `parent` points to the `attribute_node` which is connected immediately above. `next` points to another hbox `value_branch` which is connected immediately below the `parent`.

There is one distinction that is made between internal nodes and leaf nodes. A leaf `attribute_node` uses the `attribute` variable to store the value of the target attribute as which it classifies examples. In addition, `att_info`, `target_info` and `head` are NULL at a leaf node. Specifically, `head` being equal to NULL is the condition which is used to determine a leaf node.

3.2.2. Partition Structure

There are several different types of partitions. Below is a list of all the structures that handle partitioning:

```
struct example_list_s {
  int example;
  example_list_s *next;
};

struct partition_s {
  char *value;
  example_list_s *example_list;
  partition_s *next;
  int example_list_size;
};

struct attribute_s {
  char *attribute;
  partition_s *parted_value;
};
```

A 'partition' is actually made up of a linked list of `partition_s` structs, connected with the `next` element. It is broken up into sections according to a certain attribute. Examples with a similar attribute `value` are placed in the same `partition_s`. Similar to the way conditions are stored, the `example_list` is a dynamically allocated array of `example_list_s` structs. A `next` element is included in the example list to allow for a linked list implementation, which is used in some cases.

The `attribute_s` struct is used as an intermediate data structure for data which binds an `attribute` with its cooresponding partition (`parted_value`).