Ryan Barker
ECE 4420
12/03/2015

Takehome 3 Final Report

Section I. Project Topic Summary

For Takehome 3, the software developer completed an implementation of ID3 in C. The code itself is in a modular and recursive format, and the ID3 functions are contained in a fully exportable C library. The software package consists of a makefile and four different C files: *ID3.c*, *ID3.h*, *ID3test.c*, and *ID3debug.c* (See section V of this report for descriptions of each file in the package).

Functions inside the package include a function to pretty print input file data (ID3_print_file_data), an overall ID3 function (ID3), a function to pretty print ID3 trees (ID3_print_tree), an ID3 training set validation function (ID3_tree_validate), a debug-specific function (ID3_set_globals), memory management functions for destroying ID3 trees and freeing memory at the close of the program (ID3_tree_destruct/ID3_destruct), and 15 ID3 auxiliary functions to assist the prior listed functions (various names).

ID3_tree_validate operates on the training set used construct the tree (which is also stored in the abstract data type [ADT] used to represent each tree), and prints the result of the validation process to the terminal. The main recursive portion of ID3 (called ID3_compute in the software library) is expanded to print warnings for conflicting data sets where ID3 runs out of attributes but multiple outcomes exist. Note that ID3_tree_validate will also fail for these data sets. Lastly, the overall algorithm was expanded to take inputs through formatted file data, creating a very user-friendly interface that allows user to run any data they wish through ID3 if they know the correct file formatting.

Section II. Pre-Development Project Research and the Development Process

See section VI for a formal list of citations of the research done leading into this project. The developer began his preparation by reviewing his class notes on ID3 and chapter 16 of *Intelligent Systems: Principles, Paradigms, and Pragmatics* (specifically, section four) [1]. He initially used the information he gathered to write very broad-level, mostly-text filled pseudo code and pseudo-data-structures. To fill in the holes in his pseudo-code and complete his theoretical understanding of ID3, the developer then turned to several internet databases and academic websites and found three addition academic write-ups on ID3 ([2], [3], and [4]). The completed pseudo-code is in an appendix in *takehome3_proposal.pdf*. Of the sources he pulled, the developer found Chapter 6, Section 2 of Steven Marsland's *Machine Learning: An Algorithmic Perspective* most useful for preparing for this project, because it did the best job of breaking down how ID3 works in each of the specific cases in the overall algorithm [2].

Once he filled the holes in the pseudo-code, the developer began turning his ideas into reality. He quickly put together *ID3.h* and set up the functions and function prototypes in *ID3.c* so each function compiled, but immediately returned a meaningless value. This allowed him to have a skeletal template of the overall algorithm, but write and test each of the 21 functions inside of the file modularly. Further, to supplement this building block approach, the developer worked on *ID3debug.c* for testing purposes as he developed each function in *ID3.c*. *ID3debug.c* was set up to run several script-like unit drivers to verify the functionality of every function in *ID3.c*. The overall development process was as follows: The developer would write a function in *ID3.c*, write a test script for it in *ID3debug.c*, test the function along with debugging tools like GDB and Valgrind, and then move onto the next function. He would only write a function with auxiliary functions (I.E. – ID3_compute) after all of the associated auxiliaries were written and tested. *ID3test.c* was written last in the package; after the developer knew *ID3.c* and *ID3.h* were functional. During testing for each function in the package, the developer was careful to test all input cases on each function and ensure to avoid memory leaks anywhere in the software.

Section III. Brief Description of Implementation

Each function in *ID3.c* is generously well-commented to explain each algorithm used, so this section will only describe the overall ID3 function's implementation at a basic level. The engineer's implementation of ID3 can be broken into four basic steps:

1. *Encode the input file data (ID3_encode_data)*: The software engineer chose to encode the input file data from strings to integers and only run ID3 on encoded data. This is done with standard file reading functions (fopen, fclose, etc.) and heavy usage of string manipulation functions (especially the string delimiting function strtok) to parse the file format described on the next page. Data is encoded for two main reasons (among others).
    a. Encoding data to strictly integers starting at zero and increasing allows for each possible value to line up with the number of possible values. For instance, if there are five output values possible, the output value is either 0, 1, 2, 3, or 4. This allows for faster and more efficient manipulation of data, and easy indexing into string "key" arrays (by ID3_print_tree) to print the original input and output names.
    b. Encoding data to a strict set allows for ID3_compute to use a value outside of the range to mark inputs as already parsed for entropy (in this case, -1). This is essential for ID3 functionality and also allows data tables to easily be checked for being out of attributes (If the input values are all -1, there are no attributes left to check). If this is the case, and all of the output values are not the same, ID3_compute warns the user it is out of attributes with multiple output values. This fully implements the check for conflicting data.
2. *Create the initial ID3 root node (ID3_create_node):* Because of the nature of ID3_compute, it is necessary to create the first ID3 node before calling ID3_compute and passing it in. ID3_create_node is also used by ID3_compute to create new children nodes where applicable.
3. *Recursively compute the remainder of the ID3 tree (ID3_compute):* This function is very complex and the main brain of the overall ID3 algorithm. It is broken into three parts:
    a. *Base Case 1 – All Output Values in the data set associated with this node are the same*: In this case, the node is marked as an output node and is simply equal to the output value that triggered the base case.
    b. *Base Case 2 – All Input Values in the data set associated with this node are -1:* Failing base case 1 but passing base case 2 means that the algorithm is out of attributes, but there are multiple output values and the overall data set has conflicting data. The program warns the user of this, and then takes a "best guess" approach by initializing the node to an output node with output value equal to the most common output value. This is done to avoid crashing and continue the algorithm.
    c. *Recursive Step – Node is an input node and requires entropy examination:* If neither base case is satisfied, the node being examined is an input node and needs to have the entropies in its data table examined. To do this, ID3 computes a row vector of entropies for each input (with -1 entries meaning the input has already been examined at a prior recursive step and should not be reconsidered), chooses the most ideal input to partition the table by (the node with minimum entropy not equal to -1), changes the input node's name to that node's value, partitions the data set at this stage by each value of the chosen input node, creates an initially blank node for each new data table, links the new nodes to the current node appropriately in the tree ADT, and recursively calls ID3 on each new node and new data table pair. This building-block approach repeats on the new nodes until either base case 1 or base case 2 is triggered.
4. *Return the completed ID3 tree to the user*.

Input File Format: The syntax of input files compatible with this ID3 implementation is shown below. See the data directory for example files in the correct syntax. Use of this syntax allows for user-friendly and intuitive entry new data sets into the algorithm.

Line 1: <Number of Inputs> <(String) Name Values of Inputs> <Number of Possible Input Values> <(String) Input Values*>*
Line 2: <(String) Name Value of Output> <Number of Possible Output Values> <(String) Output Values>*
Line 3: <Number of rows of data (n)>
Line 4: <Blank line>
Lines 5 through (n + 4): <Row of data>
Row of data: <Input Values> <Output Value>*

Note: A "" above denotes a value with multiple cardinality. The best way to understand the syntax of the input files is to view the examples in the data subdirectory. The format is not complex.*

Section IV. Overall Goals (from Proposal) and Outcomes

- Fully develop ID3. This means developing each function in the proposal alongside *ID3debug.c*. Develop and test modularly on a per function basis to minimize errors and bugs in the development process.
    - Outcome: Successful. The overall development process was as described in section II and each function in the file was verified usability in all input cases.
- Develop ID3_print_tree once the overall ID3 algorithm has been built to show trees as formatted output.
    - Outcome: Successful. ID3_print_tree is capable of printing any ID3 tree developed by the overall ID3 function and does not crash when passed a NULL pointer.
- Develop ID3_tree_validate to address the validation of the training set for each produced ID3 tree.
    - Outcome: Successful. ID3_tree_validate was fully developed to check each training set for each tree (which, for accessibility, was stored in the root node of each tree when a tree was created) and report whether the check succeeded or failed. This function was run on every data set in *ID3test.c* and correctly validated all data sets without conflicting data and failed all data sets without conflicting data.
- Address the issues in *Intelligent Systems: Principles, Paradigms, and Pragmatics [1]* chapter 16, section 4, page 633 related to conflicting data sets in *Takehome3_report.pdf.*
    - Discussion: After implementing the check for conflicting data sets in ID3_compute (its second base case), the engineer observed sets with conflicting data fail ID3_tree_validate because the trees they generate contain contradictions. This is due to the ID3 algorithm becoming out of attributes to test, but still observing multiple possible output values. This behavior does not happen with non-conflicting data sets, and causes ID3 to take a "best guess" at the data by picking the most common output value because an absolute output value does not exist. Other difficulties surrounding ID3 include its undesirably slow efficiency with large sets of data, inability to update ID3 trees for new data sets, having to rerun the algorithm when data is added to a set to get a complete tree, and its accuracy limitations by the data it is passed. ID3 cannot make full specification for all of an inputs values if it is not given data containing all possible input values.

Section V. Software Package Contents (Deliverables)

- takehome3_proposal.pdf: Original proposal PDF for Takehome 3.
- takehome3_report.pdf: Final report for Takehome 3.
- ID3.c: Contains all functions for ID3 in C. Together with ID3.h, it is a completely portable library.
- ID3.h: Contains all ADT and function prototypes for ID3.c. Together with ID3.c, it is a completely portable library.
- ID3debug.c: Debugging script heavily utilized in the development and testing of ID3.c. Heavily verbose output.
- ID3test.c: Test script used to show the functionality of ID3.c. Runs algorithm through 10 sets of sample data.
    - Note: All of the ID3 trees generated by the algorithm were compared to either trees in the book or trees drawn by the software developer to ensure ID3 algorithm accuracy.
- makefile: Makefile to build ID3 executables. Commands:
    - 'make': Builds ./ID3 executable; an executable version of ID3test.c.
    - 'make debug': Builds ./debug; an executable version of ID3debug.c.
    - 'make all': Builds ./ID3 and ./debug.
    - 'make clean': Deletes ./ID3 and ./debug.
- data directory: Contains formatted test files for ID3 algorithm. Most examples were pulled from Intelligent Systems: Principles, Paradigms, and Pragmatics (IS:PPP).
    - block.txt: IS:PPP pg. 631
    - chords.txt: IS:PPP pg. 622
    - chords_extended2.txt: IS:PPP pg. 623
    - chords_extended3.txt: IS:PPP pg. 623-624
    - chords_extended4.txt: IS:PPP pg. 624
    - conflicting_data.txt: Generated by developer to show a case where ID3_tree_validate() fails. The last four rows in the set conflict with each other.
    - four_class_three_attribute.txt: IS:PPP pg. 619
    - four_input_and.txt: Generated by developer in initial test phases of overall ID3() function.
    - three_input_or.txt: IS:PPP pg. 617
    - tsand_reduced.txt: IS:PPP pg. 635
- logs directory: Contains log files for each executable's output.
    - debuglog.txt: Log file for ./debug executable.
    - ID3log.txt: Log file for ./ID3 executable.

Section VI. Reference List

[1] R. Schalkoff. Intelligent Systems: Principles, Paradigms, and Pragmatics. ISBN: 978-0-7637-8017-3. Cited: 11/12/2015.

[2] S. Marsland. Machine Learning: An Algorithmic Perspective [Online]. Available: http://www.briolat.org/assets/R/classif/Machine%20learning%20an%20algorithmic%20perspective(2009).pdf. Cited: 11/12/2015.

[3] R. Bhardwaj & S. Vatta. Implementation of the ID3 Algorithm [Online]. Available: http://www.ijarcsse.com/docs/papers/Volume_3/6_June2013/V3I6-0454.pdf. Cited: 11/12/2015.

[4] University of Florida Department of Computer & Information Science & Engineering. The ID3 Algorithm. Available: http://www.cise.ufl.edu/~ddd/cap6635/Fall-97/Short-papers/2.htm. Cited: 11/12/2015.