Decision Tree Classifier with GA based feature selection

Mini Project Report

Submitted to

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

By

Samir Sheriff

Satvik N

In partial fulfilment of the requirements

for the award of the degree

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

R V College of Engineering

(Autonomous Institute, Affiliated to VTU)

BANGALORE - 560059

May 2012

DECLARATION

We, Samir Sheriff and Satvik N bearing USN number 1RV09CS093 and 1RV09CS095 respectively, hereby declare that the dissertation entitled "**Decision Tree Classifier with GA feature selection**" completed and written by us, has not been previously formed the basis for the award of any degree or diploma or certificate of any other University.

Bangalore

Samir Sheriff
USN:1RV09CS093

Satvik N
USN:1RV09CS095

R V COLLEGE OF ENGINEERING

(Autonomous Institute Affiliated to VTU)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled, "**Decision Tree Classifier with GA based feature selection**", which is being submitted herewith for the award of B.E is the result of the work completed by **Samir Sheriff and Satvik N** under my supervision and guidance.

Signature of Guide

(Name of the Guide)

Signature of Head of Department

(Dr. N K Srinath)

Signature of Principal

(Dr. B.S Sathyanarayana)

Name of Examiner

Signature of Examiner

1:

2:

ACKNOWLEDGEMENT

The euphoria and satisfaction of the completion of the project will be incomplete without thanking the persons responsible for this venture.

We acknowledge RVCE (Autonomous under VTU) for providing an opportunity to create a mini-project in the 5th semester. We express our gratitude towards **Prof. B.S. Satyanarayana**, principal, R.V.C.E for constant encouragement and facilitates extended in completion of this project. We would like to thank **Prof. N.K.Srinath**, HOD, CSE Dept. for providing excellent lab facilites for the completion of the project. We would personally like to thank our project guides **Ms. Shantha Rangaswamy and Dr.** and also the lab in charge, for providing timely assistance & guidance at the time.

We are indebted to the co-operation given by the lab administrators and lab assistants, who have played a major role in bringing out the mini-project in the present form. Bangalore

Samir Sheriff

6th semester, CSE

USN:1RV09CS093

Satvik N

6th semester, CSE

USN:1RV09CS095

ABSTRACT

Machine Learning techniques have been applied to the field of classification for more than a decade. Machine Learning techniques can learn normal and anomalous patterns from training data and generate classifiers, which can be used to capture characteristics of interest. In general, the input data to classifiers is an extremely large set of features, but not all of features are relevant to the classes to be classified. Hence, the learner must generalize from the given examples in order to produce a useful output in new cases.

A major focus of machine learning research is the design of algorithms that recognize complex patterns and make intelligent decisions based on input data. Our Project, titled "**Decision Tree Classifier with Genetic Algorithm-based Feature Selection** is aimed at developing a complete program that constructs an optimal decision tree, based on any kind of data set, divided into training and testing examples, by selecting only a subset of features to classify data.

Although our program works with generic data samples, it must be noted that when we started this project, our main intention was to classify ground water samples into two classes, namely Potable and Non-Potable Water. However, thanks to the miracle of Object-Oriented Programming Concepts, we were able to extend this project.

Contents

Chapter 1

INTRODUCTION

Machine learning, a branch of artificial intelligence, is about the construction and study of systems that can learn from data. The core of machine learning deals with representation and generalization. Representation of data instances and functions evaluated on these instances are part of all machine learning systems. There is a wide variety of machine learning tasks and successful applications.

1.1 SCOPE

The machine learning concepts we have used in our project are listed below,

- **Supervised learning** is the machine learning task of inferring a function from labeled training data. The training data consist of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which is called a classifier (if the output is discrete; see classification) or a regression function (if the output is continuous; see regression). The inferred function should predict the correct output value for any valid input object. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way.

- **Decision tree learning**, used in statistics, data mining and machine learning, uses a decision tree as a predictive model which maps observations about an item to conclusions about the item's target value. The goal is to create a model that predicts the value of a target variable based on several input variables.

- A **Genetic Algorithms** is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover.

Chapter 2

REQUIREMENT SPECIFICATION

Software Requirement Specification (SRS) is an important part of the software development process. We describe the overall description of the Mini-Project, the specific requirements of the Mini-Project, the software requirements and hardware requirements and the functionality of the system.

Software Requirements

- Front End: Java SWT Application.
- Back End: Java
- Operating System: Windows 7, Ubuntu 12.10.

Hardware Requirements

- Processor: Intel Core 2 Duo or higher version
- RAM: 4GB or more
- Hard disk: 5 GB or less

Chapter 3

Decision Tree Learning

3.1 A Naive Approach

3.2 Definition

Decision tree is the learning of decision tree from class labeled training tuples. A decision tree is a flow chart like structure, where each internal (non-leaf) node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf (or terminal) node holds a class label. The topmost node in tree is the root node.

There are many specific decision-tree algorithms. Notable ones include:

- **ID3** (Iterative Dichotomiser 3)
- **C4.5** algorithm, successor of ID3
- **CART** (Classification And Regression Tree)
- **CHi-squared Automatic Interaction Detector** (CHAID). Performs multi-level splits when computing classification trees.
- **MARS**: extends decision trees to better handle numerical data

3.3 The Basic Idea

Decision tree is a classifier in the form of a tree structure (as shown in Fig. 3.1, where each node is either:

1. A **leaf node** - indicates the value of the target attribute (class) of examples (*In Fig. 3.1, the nodes containing values K=x, K=y)*
2. A **decision node** - specifies some test to be carried out on a single attribute-value, with one branch and sub-tree for each possible outcome of the test.*In Fig. 3.1, the nodes containing attributes A, B and C)*

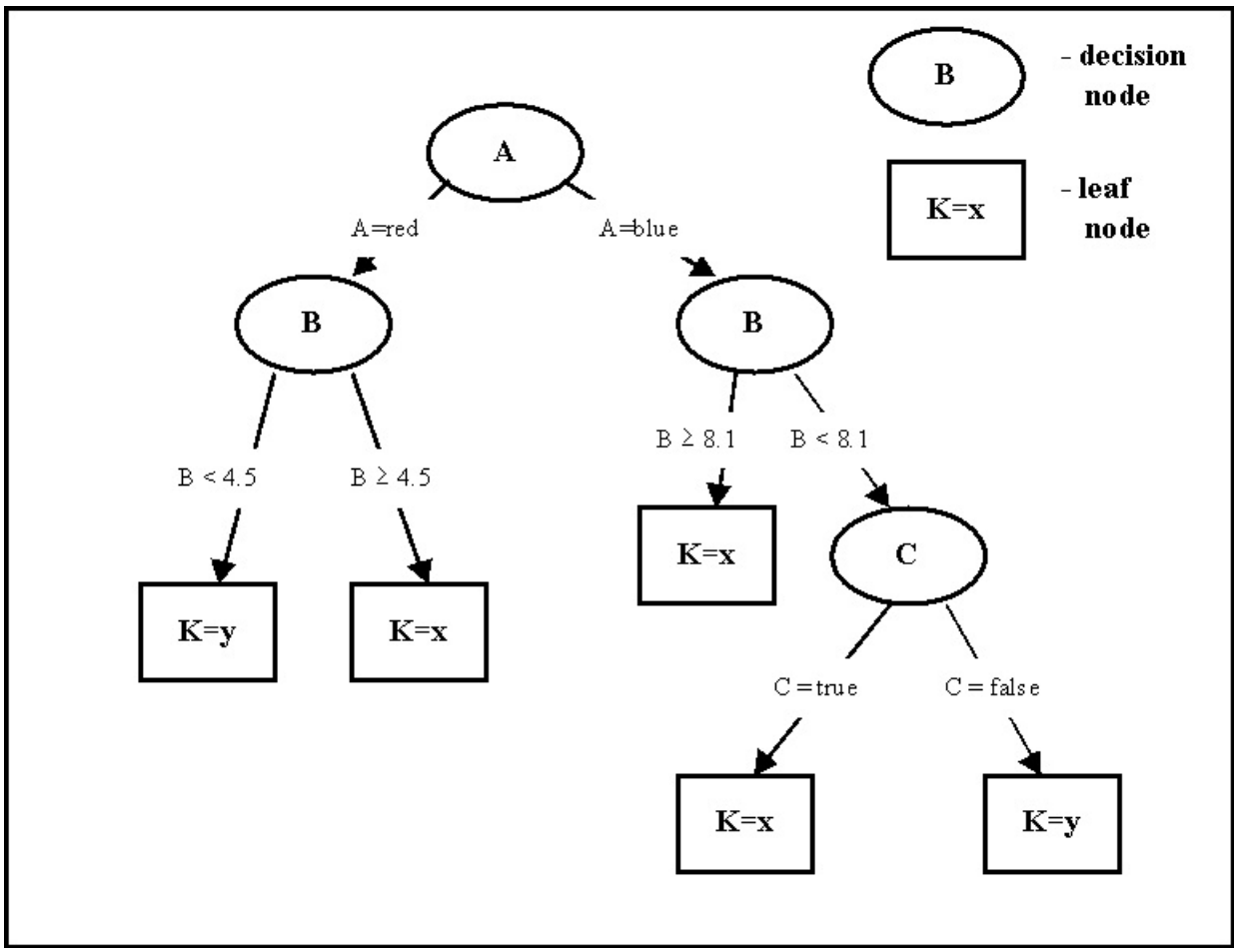

Figure 3.1: Sample Decision Tree

A decision tree can be used to classify an example by starting at the root of the tree and moving through it until a leaf node, which provides the classification of the instance.

Decision tree induction is a typical inductive approach to learn knowledge on classification. The key requirements to do mining with decision trees are:

- **Attribute-value description**: object or case must be expressible in terms of a fixed collection of properties or attributes. This means that we need to discretize continuous attributes, or this must have been provided in the algorithm.*(A, B and C, in Fig. 3.1)*
- **Predefined classes (target attribute values)**: The categories to which examples are to be assigned must have been established beforehand (supervised data) *(Classes X and Y in Fig. 3.1)*.
- **Discrete classes**: A case does or does not belong to a particular class, and there must be more cases than classes.
- **Sufficient data**: Usually hundreds or even thousands of training cases.

3.4 Building the Huffman Tree

- Begin with a forest of trees. All trees are one node, with the weight of the tree equal to the weight of the character in the node. Characters that occur most frequently have the highest weights. Characters that occur least frequently have the smallest weights.
- Repeat this step until there is only one tree:
- Choose two trees with the smallest weights, call these trees T1 and T2. Create a new tree whose root has a weight equal to the sum of the weights T1 + T2 and whose left subtree is T1 and whose right subtree is T2.
- The single tree left after the previous step is an optimal encoding tree..

3.5 An Example

3.5.1 An Example: "go go gophers"

3.5.2 Example Encoding Table

The character encoding induced by the last tree is shown below where again, 0 is used for left edges and 1 for right edges.

3.5.3 Encoded String

The string "go go gophers" would be encoded as shown (with spaces used for easier reading, the spaces wouldn't appear in the real encoding). **00 01 100 00 01 100 00 01 1110 1101 101 1111 1100**

Once again, 37 bits are used to encode "go go gophers". There are several trees that yield an optimal 37-bit encoding of "go go gophers". The tree that actually results from a programmed implementation of Huffman's algorithm will be the same each time the program is run for the same weights (assuming no randomness is used in creating the tree).

Chapter 4

Decompression

Generally speaking, the process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be somehow reconstructed.

4.1 Storing the Huffman Tree

- In the simplest case, where character frequencies are fairly predictable, the tree can be preconstructed (and even statistically adjusted on each compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency.
- Otherwise, the information to reconstruct the tree must be sent a priori.
- A naive approach might be to prepend the frequency count of each character to the compression stream. Unfortunately, the overhead in such a case could amount to several kilobytes, so this method has little practical use.
- Another method is to simply prepend the Huffman tree, bit by bit, to the output stream. For example, assuming that the value of 0 represents a parent node and 1 a leaf node, whenever the latter is encountered the tree building routine simply reads

the next 8 bits to determine the character value of that particular leaf. The process continues recursively until the last leaf node is reached; at that point, the Huffman tree will thus be faithfully reconstructed. The overhead using such a method ranges from roughly 2 to 320 bytes (assuming an 8-bit alphabet).

Many other techniques are possible as well. In any case, since the compressed data can include unused "trailing bits" the decompressor must be able to determine when to stop producing output. This can be accomplished by either transmitting the length of the decompressed data along with the compression model or by defining a special code symbol to signify the end of input (the latter method can adversely affect code length optimality, however).

4.2 Creating the Huffman Table

To create a table or map of coded bit values for each character you'll need to traverse the Huffman tree (e.g., inorder, preorder, etc.) making an entry in the table each time you reach a leaf. For example, if you reach a leaf that stores the character 'C', following a path left-left-right-right-left, then an entry in the 'C'-th location of the map should be set to 00110. You'll need to make a decision about how to store the bit patterns in the map. At least two methods are possible for implementing what could be a class/struct BitPattern:

- Use a string. This makes it easy to add a character (using +) to a string during tree traversal and makes it possible to use string as BitPattern. Your program may be slow because appending characters to a string (in creating the bit pattern) and accessing characters in a string (in writing 0's or 1's when compressing) is slower than the next approach.

- Alternatively you can store an integer for the bitwise coding of a character. You need to store the length of the code too to differentiate between 01001 and 00101. However, using an int restricts root-to-leaf paths to be at most 32 edges long since

an int holds 32 bits. In a pathological file, a Huffman tree could have a root-to-leaf path of over 100. Because of this problem, you should use strings to store paths rather than ints. A slow correct program is better than a fast incorrect program.

4.3 Storing Sizes

The operating system will buffer output, i.e., output to disk actually occurs when some internal buffer is full. In particular, it is not possible to write just one single bit to a file, all output is actually done in "chunks", e.g., it might be done in eight-bit chunks. In any case, when you write 3 bits, then 2 bits, then 10 bits, all the bits are eventually written, but you can not be sure precisely when they're written during the execution of your program. Also, because of buffering, if all output is done in eight-bit chunks and your program writes exactly 61 bits explicitly, then 3 extra bits will be written so that the number of bits written is a multiple of eight. Because of the potential for the existence of these "extra" bits when reading one bit at a time, you cannot simply read bits until there are no more left since your program might then read the extra bits written due to buffering. This means that when reading a compressed file, you CANNOT use code like this.

```
int bits;
while (input.readbits(1, bits))
{
// process bits
}
```

To avoid this problem, you can write the size of a data structure before writing the data structure to the file.

Chapter 5

CONCLUSION AND FUTURE WORKS

Summary

Limitations

1. Huffman code is optimal only if exact probability distribution of the source symbols is known.

2. Each symbol is encoded with integer number of bits.

3. Huffman coding is not efficient to adapt with the changing source statistics.

4. The length of the codes of the least probable symbol could be very large to store into a single word or basic storage unit in a computing system.

Further enhancements The huffman coding the we have considered is simple binary Huffman codingbut many variations of Huffman coding exist,

1. **n-ary Huffman coding:** The n-ary Huffman algorithm uses the {0, 1, ... , n 1} alphabet to encode message and build an n-ary tree. This approach was considered by Huffman in his original paper. The same algorithm applies as for binary (n equals 2)codes, except that the n least probable symbols are taken together, instead of just the 2 least probable. Note that for n greater than 2, not all sets of source words

can properly form an n-ary tree for Huffman coding. In this case, additional 0-probability place holders must be added. If the number of source words is congruent to 1 modulo n-1, then the set of source words will form a proper Huffman tree.

2. **Adaptive Huffman coding:** A variation called adaptive Huffman coding calculates the probabilities dynamically based on recent actual frequencies in the source string. This is some what related to the LZ family of algorithms.

3. **Huffman template algorithm:** Most often, the weights used in implementations of Huffman coding represent numeric probabilities, but the algorithm given above does not require this; it requires only a way to order weights and to add them. The Huffman template algorithm enables one to use any kind of weights (costs,frequencies etc)

4. **Length-limited Huffman coding:** Length-limited Huffman coding is a variant where the goal is still to achieve a minimum weighted path length, but there is an additional restriction that the length of each codeword must be less than a given constant. The package-merge algorithm solves this problem with a simple greedy approach very similar to that used by Huffman's algorithm. Its time complexity is O(nL),where L is the maximum length of a codeword. No algorithm is known to solve this problem in linear or linear logarithmic time, unlike the presorted and unsorted conventional Huffman problems, respectively.

Bibliography

[1] Genetic Algorithms -http://en.wikipedia.org/wiki/Genetic_algorithm

[2] Genetic Algorithm for constructing DT - http://www.jprr.org/index.php/jprr/article/viewFile/44/25

[3] Project brief for the DT using Horse data sets - https://cs.uwaterloo.ca/~ppoupart/teaching/cs486-spring06/assignments/asst4/asst4.pdf

[4] Supervised and Unsupervised Discretization of Continous Features - http://robotics.stanford.edu/users/sahami/papers-dir/disc.pdf

[5] Hybrid learning using Genetic Algorithms and Decision Trees for pattern sification - http://cs.gmu.edu/~eclab/papers/ijcai95.pdf

[6] Kardi Maams' Tutorials - http://people.revoledu.com/kardi/tutorial/DecisionTree/index.html

Appendices

Appendix A : Source Code

Appendix B : Screen Shots

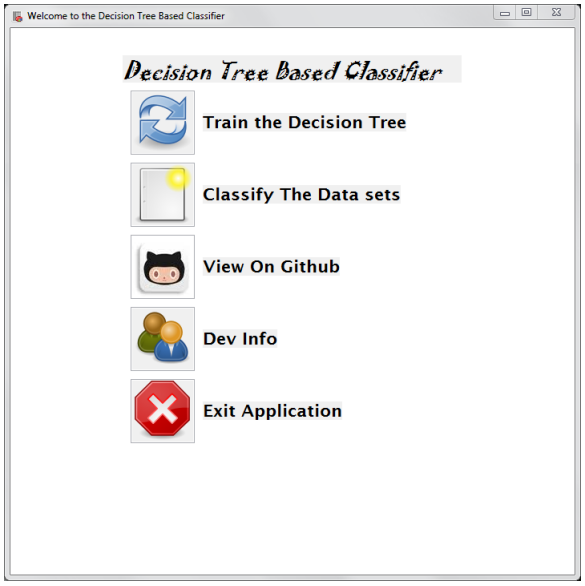

Figure 5.1: Application Window - Welcome Screen

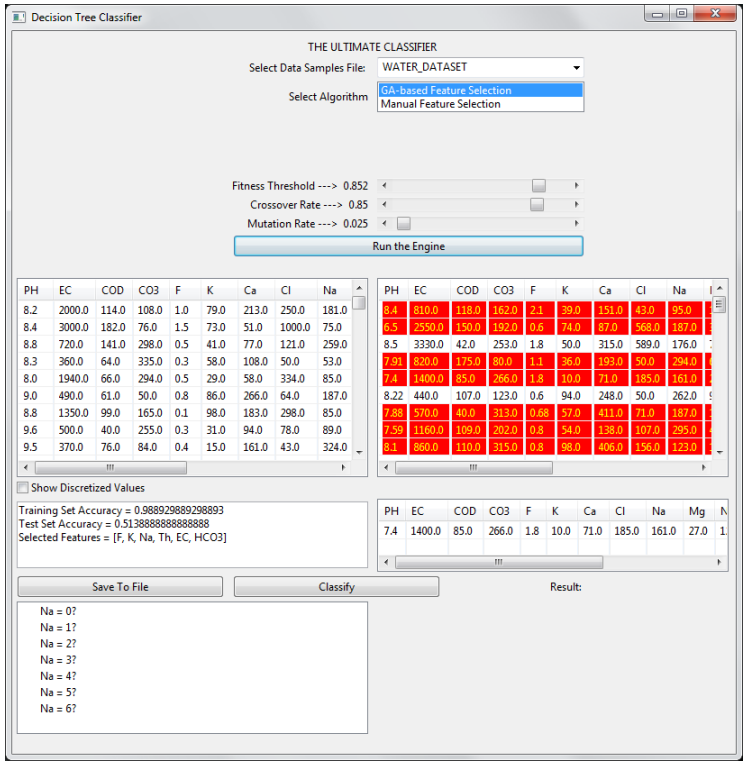

Figure 5.2: Decision Tree Constructor Window.

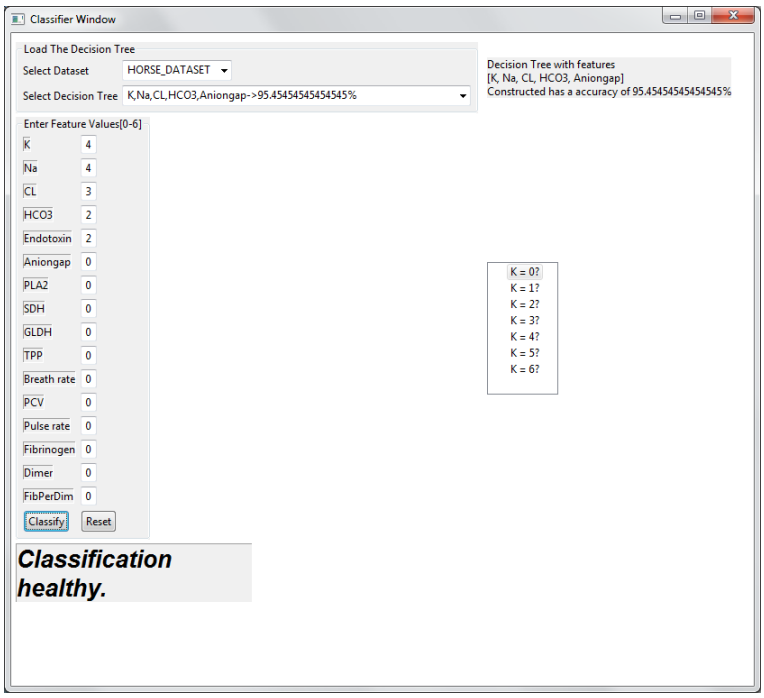

Figure 5.3: Decision Tree Classifier Window.

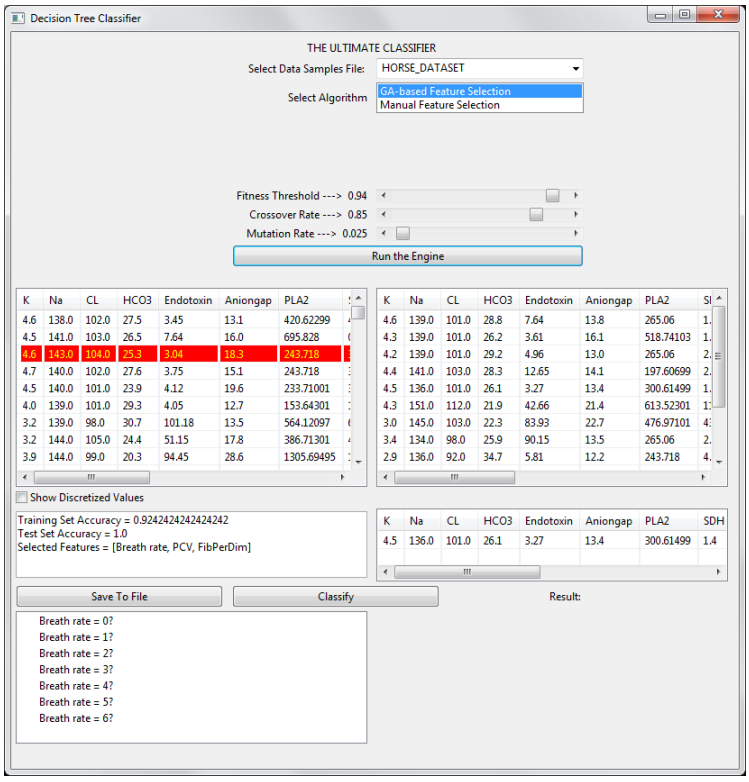

Figure 5.4: Decision Tree Construction with GA based Feature Selector.

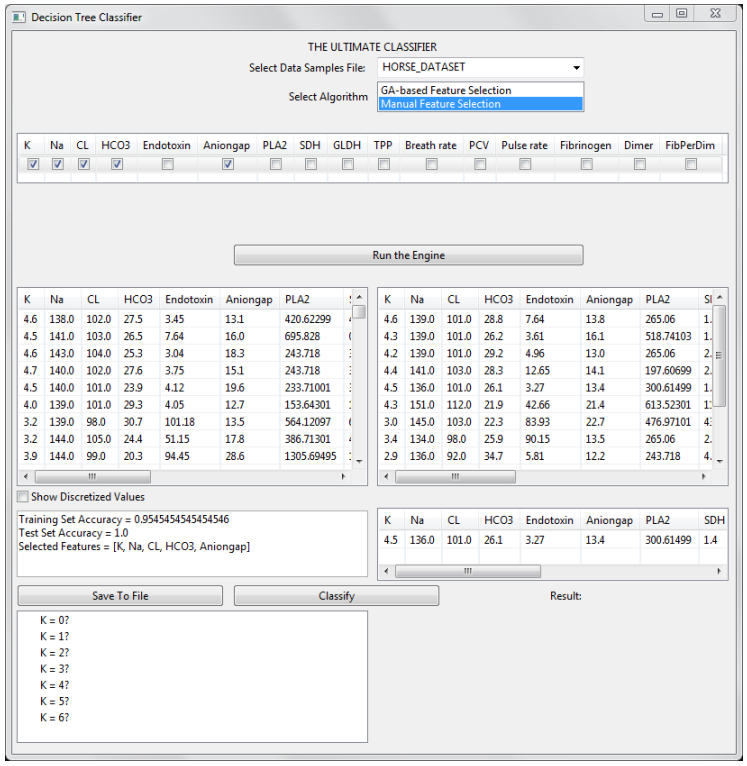

Figure 5.5: Decision Tree Construction with manual feature selection.

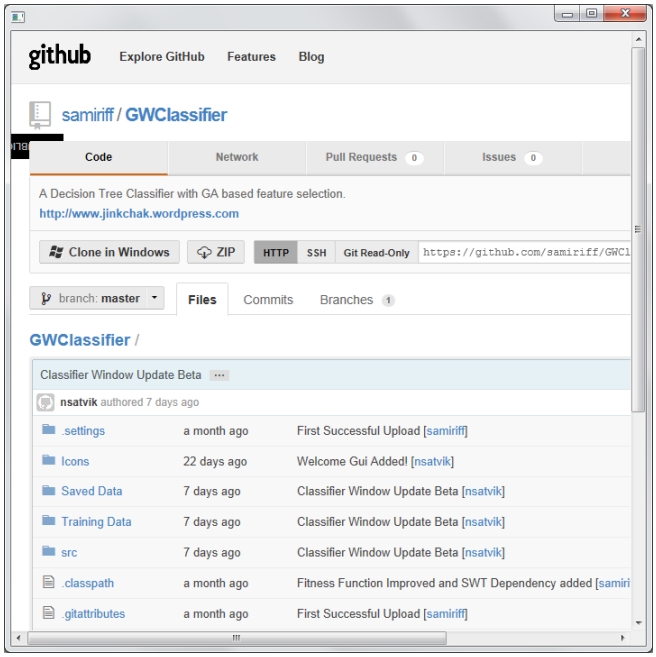

Figure 5.6: The project source code on github public repository.