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 **Chaitra B.H. and Suma B.** 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

The Project "**Data Compression Techniques** is aimed at developing programs that transform a string of characters in some representation (such as ASCII) into a new string (of bits, for example) which contains the same information but whose length is as small as possible. Compression is useful because it helps reduce the consumption of resources such as data space or transmission capacity. The design of data compression schemes involve trade-off s among various factors, including the degree of compression, the amount of distortion introduced (e.g., when using lossy data compression), and the computational resources required to compress and uncompress the data.

Many data processing applications require storage of large volumes of data, and the number of such applications is constantly increasing as the use of computers extends to new disciplines. Compressing data to be stored or transmitted reduces storage and/or communication costs. When the amount of data to be transmitted is reduced, the effect is that of increasing the capacity of the communication channel. Similarly, compressing a file to half of its original size is equivalent to doubling the capacity of the storage medium. It may then become feasible to store the data at a higher, thus faster, level of the storage hierarchy and reduce the load on the input/output channels of the computer system.

Contents

Chapter 1

INTRODUCTION

Trees are everywhere. We love trees.

1.1 SCOPE

The data compression techniques find applications in almost all fields. To list a few,

- **Audio data compression** reduces the transmission bandwidth and storage requirements of audio data. Audio compression algorithms are implemented in software as audio codecs. Lossy audio compression algorithms provide higher compression at the cost of fidelity, are used in numerous audio applications. These algorithms almost all rely on psychoacoustics to eliminate less audible or meaningful sounds, thereby reducing the space required to store or transmit them. Video

- **Video compression** uses modern coding techniques to reduce redundancy in video data. Most video compression algorithms and codecs combine spatial image compression and temporal motion compensation. Video compression is a practical implementation of source coding in information theory. In practice most video codecs also use audio compression techniques in parallel to compress the separate, but combined data streams.

- **Grammar-Based Codes**

They can extremely compress highly-repetitive text, for instance, biological data collection of same or related species, huge versioned document collection, internet archives, etc. The basic task of grammar-based codes is constructing a context-free grammar deriving a single string. Sequitur and Re-Pair are practical grammar compression algorithms which public codes are available.

Chapter 2

REQUIREMENT SPECIFICATION

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

Software Requirements

- Front End: Qt GUI Application.
- Back End: C++
- Operating System: Linux.

Hardware Requirements

- Processor: Intel Pentium 4 or higher version
- RAM: 512MB or more
- Hard disk: 5 GB or less

Chapter 3

Compression

3.1 A Naive Approach

3.2 The Basic Idea

3.3 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..

3.4 An Example

3.4.1 An Example: "go go gophers"

3.4.2 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.

3.4.3 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

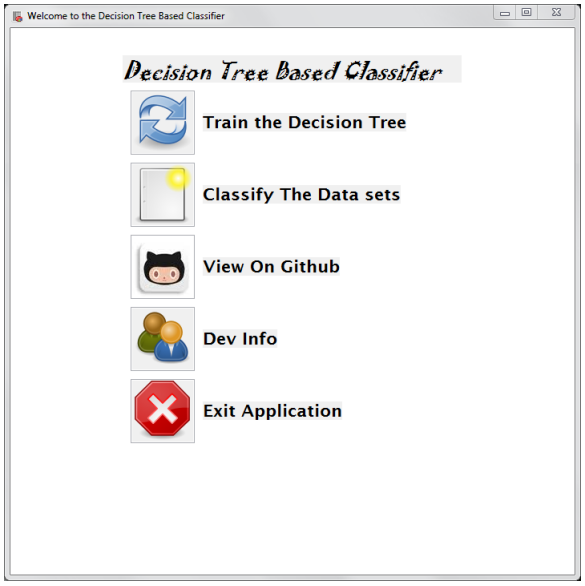

Figure 5.1: Application Window - Welcome Screen

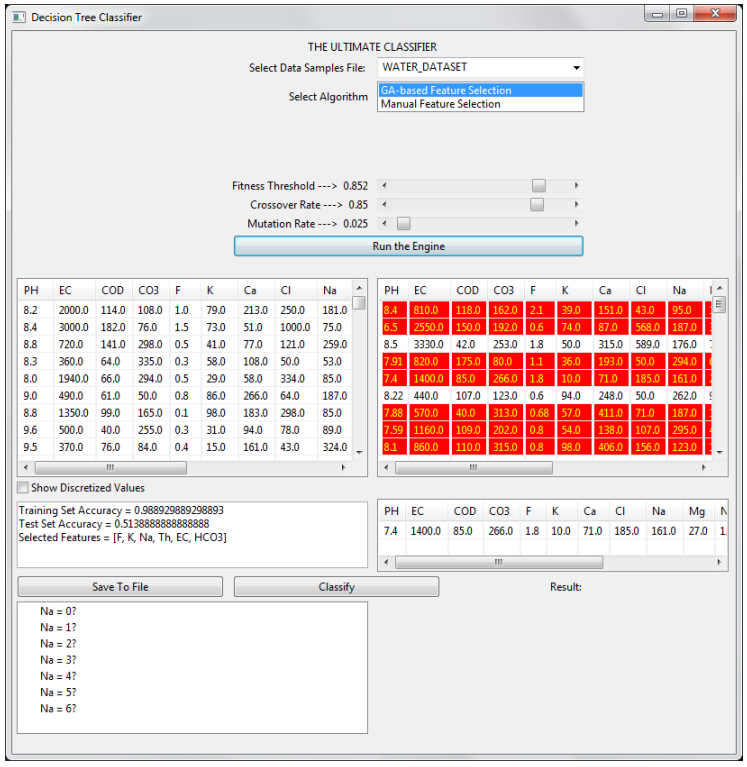

Figure 5.2: Decision Tree Constructor Window.

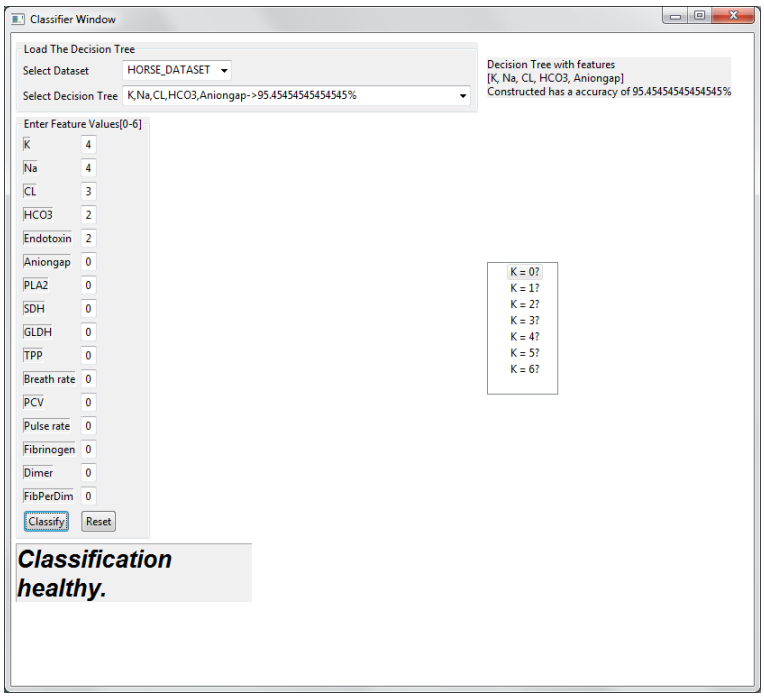

Figure 5.3: Decision Tree Classifier Window.

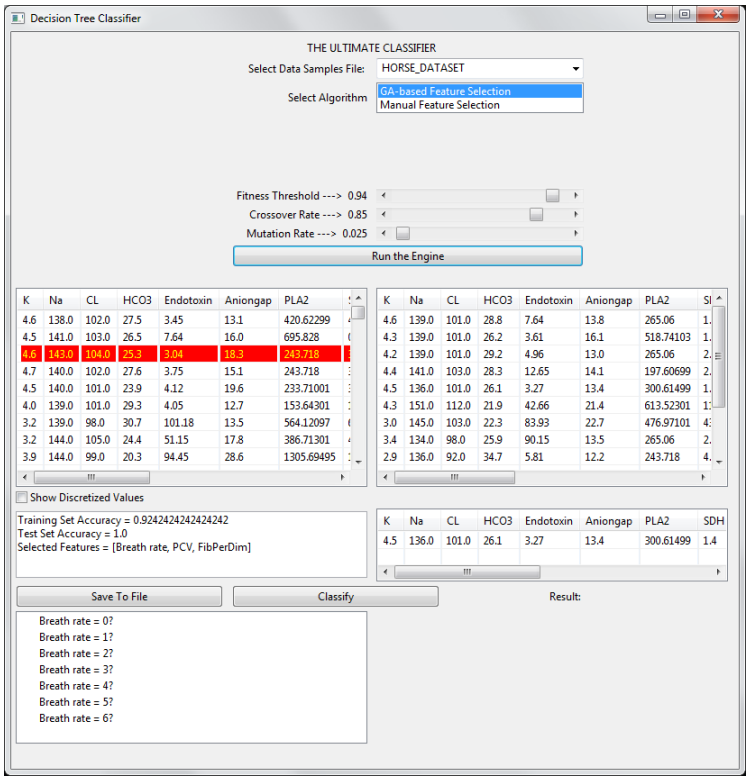

Figure 5.4: Decision Tree Construction with GA based Feature Selector.

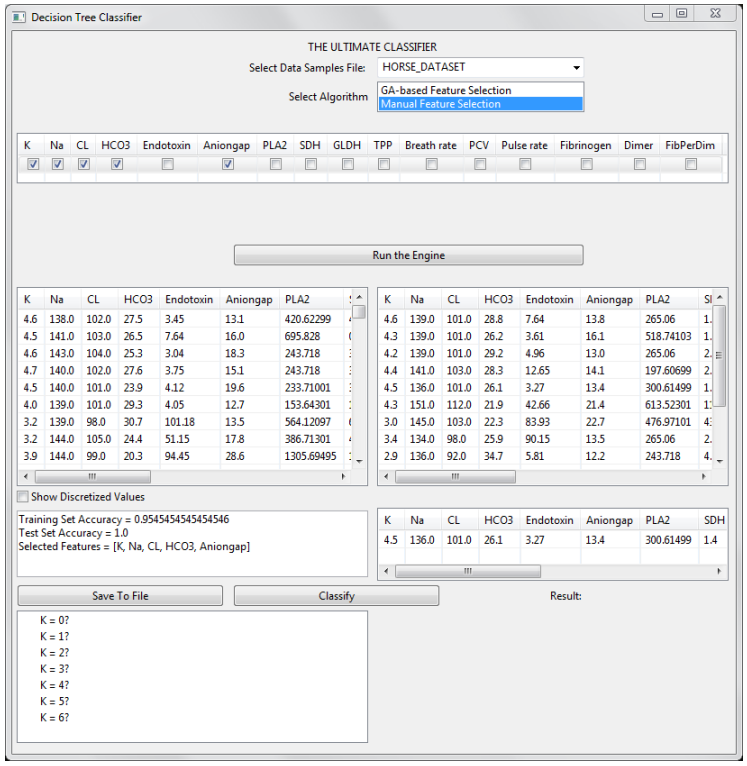

Figure 5.5: Decision Tree Construction with manual feature selection.

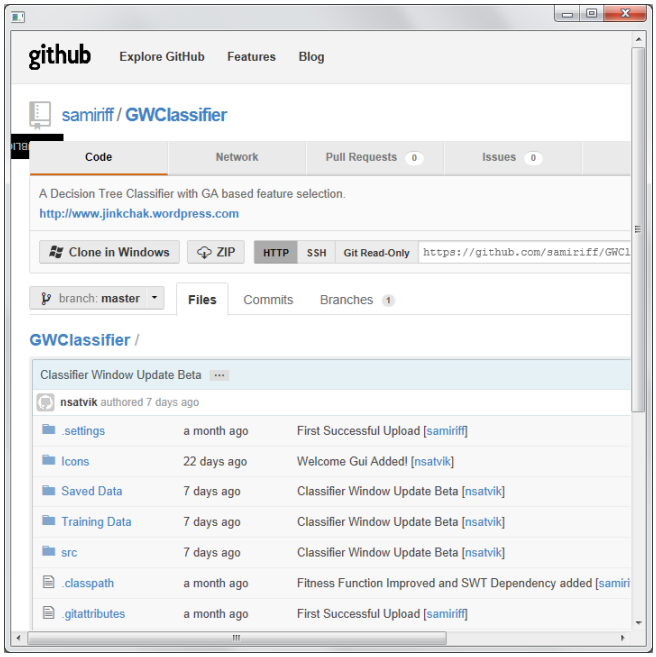

Figure 5.6: The project source code on github public repository.