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 based 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
(Mrs. Shanta R)

Signature of Head of Department
(Dr. N K Srinath)

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 **Mrs. Shantha Rangaswamy and Dr. G. Shobha** 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

7th semester, CSE

USN:1RV09CS093

Satvik N

7th 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 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

3.2 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)*

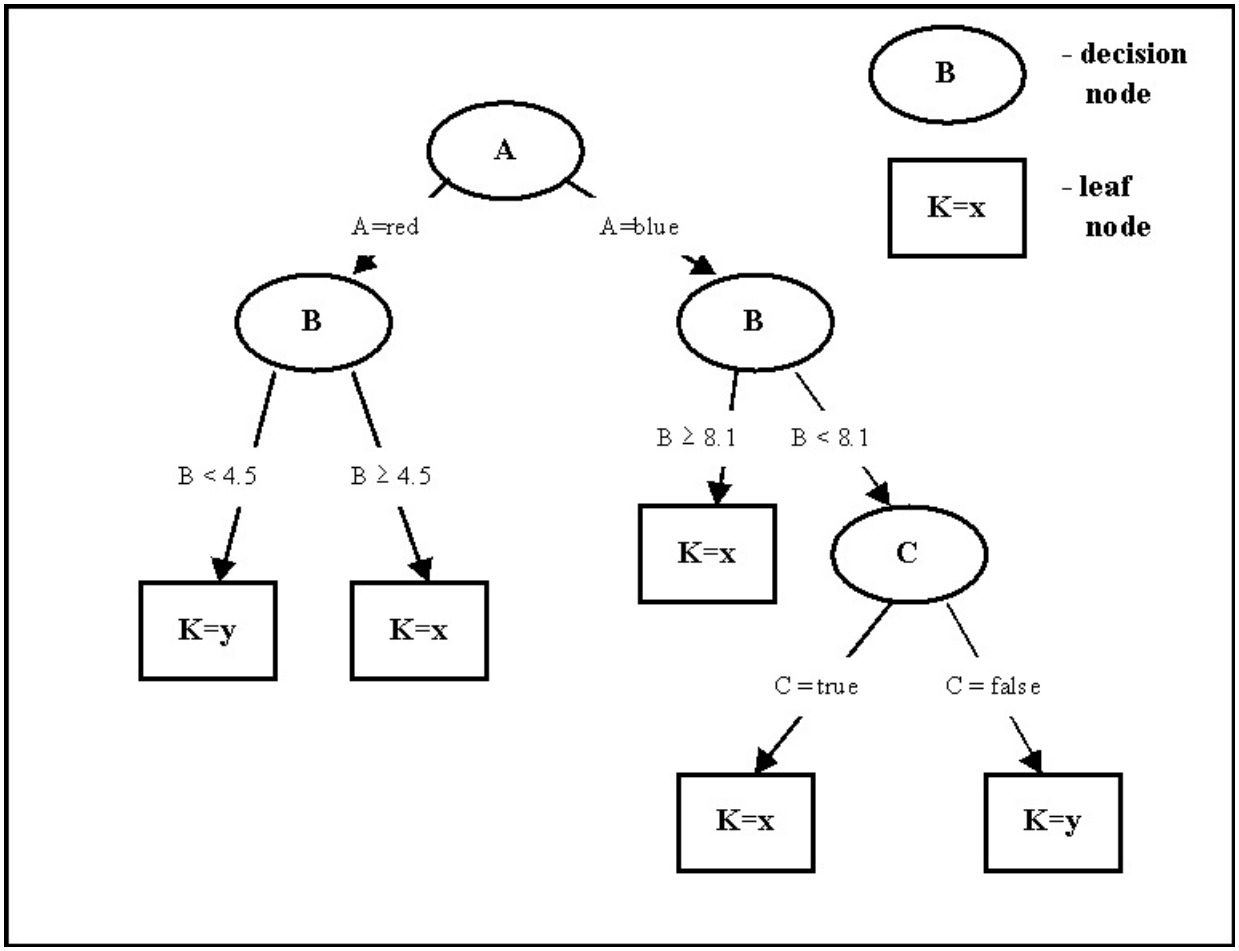

Figure 3.1: Sample Decision Tree

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.

3.3 Building the Decision Tree

Most algorithms that have been developed for learning decision trees are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees. Decision tree programs construct a decision tree T from a set of training cases.

3.3.1 ID3 Algorithm

J. Ross Quinlan originally developed ID3 at the University of Sydney. He first presented ID3 in 1975 in a book, Machine Learning, vol. 1, no. 1. ID3 is based on the Concept Learning System (CLS) algorithm. ID3 searches through the attributes of the training instances and extracts the attribute that best separates the given examples. If the attribute perfectly classifies the training sets then ID3 stops; otherwise it recursively operates on the m (where m = number of possible values of an attribute) partitioned subsets to get their "best" attribute. The algorithm uses a greedy search, that is, it picks the best attribute and never looks back to reconsider earlier choices. Note that ID3 may misclassify data.

```
function ID3
Input:   (R: a set of non-target attributes,
          C: the target attribute,
          S: a training set) returns a decision tree;
begin
   If S is empty, return a single node with
      value Failure;
   If S consists of records all with the same
      value for the target attribute,
      return a single leaf node with that value;
   If R is empty, then return a single node
      with the value of the most frequent of the
      values of the target attribute that are
      found in records of S; [in that case
      there may be be errors, examples
      that will be improperly classified];
   Let A be the attribute with largest
      Gain(A,S) among attributes in R;
   Let {aj| j=1,2, .., m} be the values of
      attribute A;
   Let {Sj| j=1,2, .., m} be the subsets of
      S consisting respectively of records
      with value aj for A;
   Return a tree with root labeled A and arcs
      labeled a1, a2, .., am going respectively
      to the trees (ID3(R-{A}, C, S1), ID3(R-{A}, C, S2),
      .....,ID3(R-{A}, C, Sm);
   Recursively apply ID3 to subsets {Sj| j=1,2, .., m}
      until they are empty
end
```

Figure 3.2: ID3 Algorithm

3.3.2 Choosing the best attribute for a given node

The estimation criterion in the decision tree algorithm is the selection of an attribute to test at each decision node in the tree. The goal is to select the attribute that is most useful for classifying examples. A good quantitative measure of the worth of an attribute is a statistical property called information gain that measures how well a given attribute separates the training examples according to their target classification. This measure is used to select among the candidate attributes at each step while growing the tree.

3.3.3 Entropy - a measure of homogeneity of the set of examples

In order to define information gain precisely, we need to define a measure commonly used in information theory, called entropy, that characterizes the (im)purity of an arbitrary collection of examples. Given a set S, containing only positive and negative examples of some target concept (a 2 class problem), the entropy of set S relative to this simple, binary classification is defined as:

$Entropy(S) = -p_p log_2 p_p - p_n log_2 p_n$

where p_p is the proportion of positive examples in S and p_n is the proportion of negative examples in S. In all calculations involving entropy we define 0 log 0 to be 0.

To illustrate, suppose S is a collection of 25 examples, including 15 positive and 10 negative examples [15+, 10-]. Then the entropy of S relative to this classification is

$Entropy(S) = -(15/25)log_2(15/25) - (10/25)log_2(10/25) = 0.970$

Notice that the entropy is 0 if all members of S belong to the same class. For example, if all members are positive (p_p= 1), then p_n is 0, and:

$Entropy(S) = -1log_2(1) - 0log_2 0 = -10 - 0log_2 0 = 0.$

Note the entropy is 1 (at its maximum!) when the collection contains an equal number of positive and negative examples. If the collection contains unequal numbers of positive and negative examples, the entropy is between 0 and 1. Figure 3.3 shows the form of the entropy function relative to a binary classification, as p+ varies between 0 and 1.

One interpretation of entropy from information theory is that it specifies the minimum number of bits of information needed to encode the classification of an arbitrary member of S (i.e., a member of S drawn at random with uniform probability). For example, if p_p is 1, the receiver knows the drawn example will be positive, so no message need be sent, and the entropy is 0. On the other hand, if p_p is 0.5, one bit is required to indicate whether the drawn example is positive or negative. If p_p is 0.8, then a collection of messages can be encoded using on average less than 1 bit per message by assigning shorter codes to collections of positive examples and longer codes to less likely negative examples.

Thus far we have discussed entropy in the special case where the target classification is binary. If the target attribute takes on c different values, then the entropy of S relative

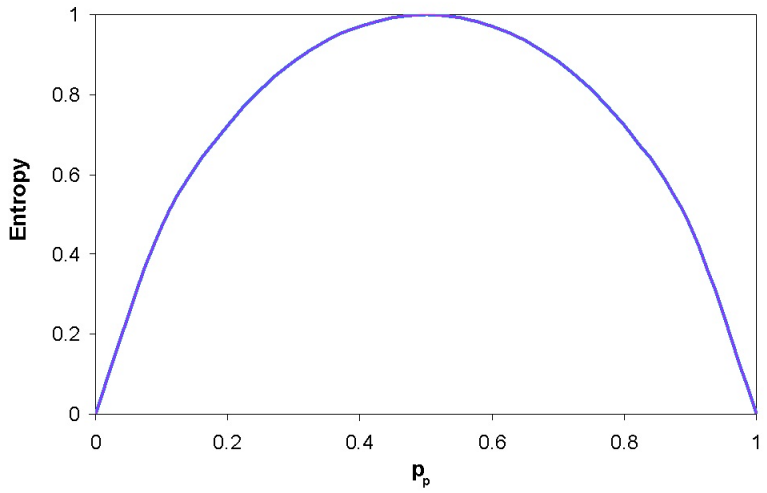

Figure 3.3: The entropy function relative to a binary classification, as the proportion of positive examples p_p varies between 0 and 1.

to this c-wise classification is defined as:

$Entropy(S) = \sum_1^c -p_i log_2 p_i$

where p_i is the proportion of S belonging to class i. Note the logarithm is still base 2 because entropy is a measure of the expected encoding length measured in bits. Note also that if the target attribute can take on c possible values, the maximum possible entropy is $\log_2 c$.

3.3.4 Information gain measures the expected reduction in entropy

Given entropy as a measure of the impurity in a collection of training examples, we can now define a measure of the effectiveness of an attribute in classifying the training data. The measure we will use, called information gain, is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, Gain (S, A) of an attribute A, relative to a collection of examples S, is defined as:

$Gain(S, A) = Entropy(S) - \sum_{v \in Value(A)} \frac{S_v}{S} Entropy(S_v)$

where Values(A) is the set of all possible values for attribute A, and S_v is the subset of S for which attribute A has value v (i.e., S_v = s S — A(s) = v). Note the first term in the equation for Gain is just the entropy of the original collection S and the second term is the expected value of the entropy after S is partitioned using attribute A. The expected entropy described by this second term is simply the sum of the entropies of each subset S_v, weighted by the fraction of examples $|S_v|/|S|$ that belong to S_v. Gain (S,A) is therefore the expected reduction in entropy caused by knowing the value of attribute A. Put another way, Gain(S,A) is the information provided about the target attribute value, given the value of some other attribute A. The value of Gain(S,A) is the number of bits saved when encoding the target value of an arbitrary member of S, by knowing the value of attribute A.

The process of selecting a new attribute and partitioning the training examples is now repeated for each non-terminal descendant node, this time using only the training examples associated with that node. Attributes that have been incorporated higher in the tree are excluded, so that any given attribute can appear at most once along any path through the tree. This process continues for each new leaf node until either of two conditions is met:

1. Every attribute has already been included along this path through the tree
2. The training examples associated with this leaf node all have the same target attribute value (i.e., their entropy is zero).

3.3.5 Discretization

Decision Tree Learning requires a discrete feature space. To handle continuous feature spaces, the process of discretization has to be carried out on the feature space to obtain a discrete feature space, which can act as an input to the ID3 algorithm. One of the most famous algorithms for discretization is, perhaps, equal width interval binning. It involves sorting the observed values of a continuous feature and dividing the range of observed values for a variable into k equally sized bins, where k is a parameter supplied by the user. If a variable x is observed to have values bounded by x_{min} and x_{max}, then

this method computes the bin width: $\delta = \frac{x_{max} - x_{min}}{k}$ and constructs bin boundaries, or thresholds, at x_{min} + iδ where *i = 1,...,k-1*. This method is applied to each continuous feature independently.

3.3.6 Limitation of Decision Tree Methods

The weaknesses of decision tree methods

- Decision trees are less appropriate for estimation tasks where the goal is to predict the value of a continuous attribute.
- Decision trees are prone to errors in classification problems with many class and relatively small number of training examples.
- Decision tree can be computationally expensive to train. The process of growing a decision tree is computationally expensive. At each node, each candidate splitting field must be sorted before its best split can be found. In some algorithms, combinations of fields are used and a search must be made for optimal combining weights. Pruning algorithms can also be expensive since many candidate sub-trees must be formed and compared.
- Decision trees do not treat well non-rectangular regions. Most decision-tree algorithms only examine a single field at a time. This leads to rectangular classification boxes that may not correspond well with the actual distribution of records in the decision space.

Chapter 4

Genetic Algorithms

Nature seems to have an uncanny knack for problem-solving. Life began as a handful of simple, single-celled organisms barely equipped to survive the harsh environment of planet Earth. However, in the short span of a few billion years, nature has adapted and evolved them into beings complex enough to ponder their own origins. While this is indeed amazing, the truly incredible part is that it all happened according to a simple plan–allow individuals with favorable traits to survive and reproduce, and let die all the rest. This, in short, is the basis for a genetic algorithm.

4.1 The Algorithm

- Create an initial population of random genomes.
- Loop through the genetic algorithm, which produces a new generation every iteration.
  - Assess the fitness of each genome, stopping if a solution is found.
  - Evolve the next generation through natural selection and reproduction.
    - Select two random genomes based on fitness.
    - Cross the genomes or leave them unchanged.
    - Mutate genes if necessary.

- Delete the old generation and set the new generation to the current population.

- When a solution is found or a generation limit is exceeded, the loop breaks and the genetic algorithm is complete.

4.2 Genetic Operators

The basic genetic algorithm attempts to evolve traits that are optimal for a given problem. It has a wide variety of common uses, notably for balancing weights in neural networks.

4.2.1 Generation Zero

The first step in the genetic algorithm is to create an initial population, generation zero, that contains a set of randomized strings of genes. Each string of genes, illustratively called a genome or chromosome, represents a series of traits that may or may not be useful for the problem at hand. These "genes" are usually represented by either binary digits or real numbers.

Random Genome

Bits (Genes)	0110	1100	1111	1011	0100	1010	0111	0101	1110
Values (Traits)	6	12	15	11	4	10	7	5	14

Figure 4.1: Random Genome

4.2.2 Survival of the Fittest

Every genome in the population must now be assigned a fitness score according to how well it solves the problem at hand. The process and approach to measuring a genomes fitness will be different for every problem. Determining the fitness measure is the most important and often most difficult part of developing a genetic algorithm.

4.2.3 The Next Generation

Once the fitness for every genome is determined, its time to start building the next generation of genomes based on probability and fitness. This is the main part of the genetic algorithm, where the strong survive and the weak perish. It usually consists of these three parts:

Selection

Two genomes are selected randomly from the current population (reselection allowed), with fitter genomes having a higher chance of selection. The selected genomes, which should have a relatively high fitness score, are guaranteed to pass some of their traits to the next generation. This means that the average fitness of each successive generation will tend to increase.

The best way to program the selection function is through a method creatively named roulette selection. First, a random number between zero and the sum of the populations fitness is generated. Imagine this value as a ball landing somewhere on a pie graph of the populations fitness. Then, each genomes fitness, or slice of the pie graph, is added one by one to a running total. If the ball ends up in that genomes slice, it is selected.

```
RouletteSelection()
{
    float ball  = rand_float_between(0.0, total_fitness);
    float slice = 0.0;

    for each genome in population
    {
        slice += genome.fitness;

        if ball < slice
            return genome;
    }
}
```

Figure 4.2: Roulette Selection Pseudo-Code

Crossover

The two genomes now have a good chance of crossing over with one another, meaning that they will each donate a portion of their genes to form two offspring that become part of the next generation. If they do not cross over, they simply go on to the next generation unchanged. The crossover rate determines how often the genomes will cross over, and should be in the vicinity of 65-85

A crossover operation on the binary genomes in our example would begin by choosing a random position at which to cross them. The first part of the fathers genes and the second part of the mother's genes combine to form the first child, with a similar effect for the second child. The following shows a crossover operation with the crossover point at 12.

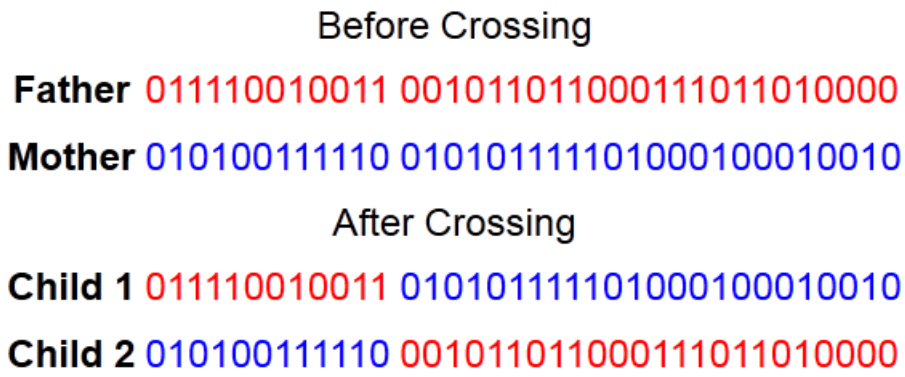

Figure 4.3: Crossover

Mutation

Just before the genomes are placed into the next generation, they have a slight chance of mutating. A mutation is simply a small, random change to one of the genes. With binary genes, mutation means flipping the bit from 1 to 0 or 0 to 1. With real number genes, a small, random perturbation is added to the gene.

The mutation rate determines the chances for each gene to undergo mutation, meaning that every individual gene should get a chance to mutate. The mutation rate should be roughly 1-5 percent for binary and 5-20 percent for real numbers.

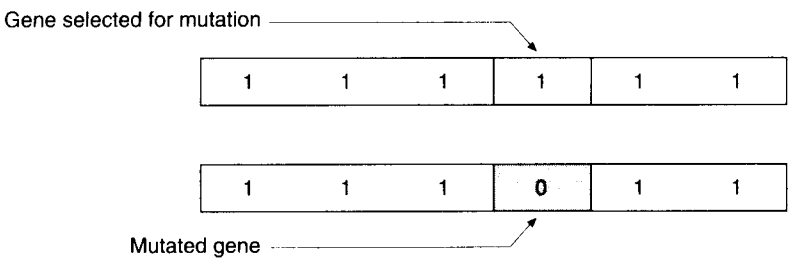

Figure 4.4: Mutation

The purpose of mutation in GAs is preserving and introducing diversity. Mutation should allow the algorithm to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping evolution. This reasoning also explains the fact that most GA systems avoid only taking the fittest of the population in generating the next but rather a random (or semi-random) selection with a weighting toward those that are fitter.

Chapter 5

Decision Trees and Genetic Algorithms

In GA based DT Classifier, the search component is a GA and the evaluation component is a decision tree. A detailed description of this algorithm is shown in Figure 5.1.

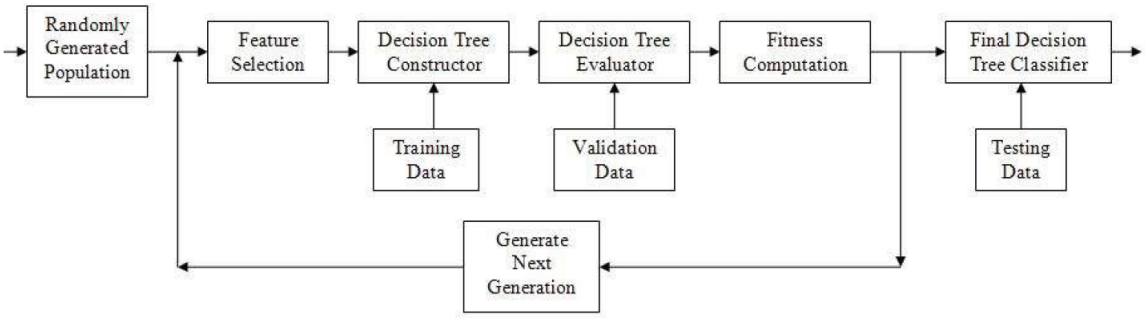

Figure 5.1: The data flow in DT/GA Hybrid Classifier Algorithm.

The basic idea of our hybrid system is to use GAs to efficiently explore the space of all possible subsets of a given feature set in order to find feature subsets which are of low order and high discriminatory power. In order to achieve this goal, fitness evaluation has to involve direct measures of size and classification performance, rather than measures such as the ranking methods such as information gain, etc.

An initial set of features is provided together with a training set of the measured feature vectors extracted from raw data corresponding to examples of concepts for which the decision tree is to be induced. The genetic algorithm (GA) is used to explore the

space of all subsets of the given feature set where preference is given to those features sets which achieve better classification performance using smaller dimensionality feature sets. Each of the selected feature subsets is evaluated (its fitness measured) by testing the decision tree produced by the ID3 algorithm. The above process is iterated along evolutionary lines and the best feature subset found is then recommended to be used in the actual design of the pattern classification system.

5.1 Representation of Chromosomes

Every individual of the population has N genes, each of which represents a feature of the input data and can be assigned to 1 or 0. 1 means the represented feature is used during constructing decision trees; 0 means it is not used. As a result, each individual in the population represents a choice of available features.

PH	Cu	HCo3	Co3	F	K	Ca	Cl	Na	Mg	No3	S04	Th	Si	EC
1	0	0	0	1	1	1	0	0	1	1	0	0	0	0

Figure 5.2: Sample Representation

In this figure, only the following features are being considered: PH, F, K, Ca, Mg, and No3, yielding the chromosome: 100011100110000

5.2 Population

The initial population is randomly generated. For each individual in the current population, a decision tree is built using the ID3 algorithm. This resulting decision tree is then tested over validation data sets, which generate classification error rates. In our application, we calculate fitness values using the weighted average of training and test classification errors. The lower the classification error rate, the better the fitness of the individual. In other words, the population consists of a number of probable decision trees.

5.3 Advantages

Such a hybrid learning system will identify significantly better feature subsets than those produced by existing methods for two reasons.

1. The power of Genetic algorithms is being exploiting to efficiently explore the non-linear interactions of a given set of features.

2. By using ID3 in the evaluation loop, an efficient mechanism for directly measuring classification accuracy is present

Chapter 6

Our State-of-the-art Object-Oriented System

Object-oriented programming (OOP) is a programming paradigm that represents concepts as "objects" that have data fields *(attributes that describe the object)* and associated procedures known as methods. Objects, which are instances of classes, are used to interact with one another to design applications and computer programs.
Had it not been for the presence of the OOP paradigm, our efforts in this project would have gone in vain, and we do not use that term lightly. Code management was a whole lot easier when compared to our past experience with procedural programming. In this chapter, we describe the different packages that were created by us to efficiently manage our code. Know first that our application consists of four diverse packages, namely:

1. **org.ck.sample**
2. **org.ck.dt**
3. **org.ck.ga**
4. **org.ck.gui**

We describe each package in detail, in the following sections. We follow a bottom-up methodology for explaining the layout of the classes.

6.1 org.ck.sample

This package allows us to efficiently manage and encapsulate the details of the data samples, provided by users, which are required for analysis. It is this class that allows our application to accept generic data sets. It consists of five classes:

1. **DataHolder** - This class keeps track of names of files that contain training and testing samples; lists of features; their corresponding classification values; and Probability values. It provides this information, when required, to the front-end or back-end of our application. To make a long story short, this class acts like a middleman between the back-end and front-end of our application.

2. **Feature** - This class stores a mapping between a feature name and a feature value.

3. **Sample** - This class stores all the features and the corresponding classification value for one training/test sample only.

4. **SampleCollection** - A SampleCollection, as the name suggests, is a collection of samples. In essence, this class reads the sample data from a file (using information provided by a DataHolder object) and initializes all the necessary data structures to store the data values for Classification analysis.

5. **SampleSplitter** - This class contains methods that operate on a given SampleCollection, in order to split it into two new SampleCollections, based on a given Feature. It also calculates the information gain *(as described in Chapter 3)* of the given split operation.

6.2 org.ck.dt

This package allows us to efficiently manage and encapsulate methods for all stages of Decision Tree Learning.

1. **DecisionTreeNode** - A DecisionTreeNode is a structure which may have either:

(a) a classification value, if it happens to be a leaf node

(b) a list of one or more children DecisionTreeNodes, if it happens to be a decision node, i.e., an internal node.

Know that the types of these nodes is defined by the decision tree that is constructed, and a node has at most one parent.

2. **DecisionTreeConstructor** - This class takes a SampleCollection *(containing training samples)* as input, builds a decision tree *(as described in Chapter 3)* and stores the root DecisionTreeNode of the decision tree. In essence, a DecisionTreeConstructor consists of a number of DecisionTreeNodes.

3. **DecisionTreeClassifier** - This class keeps track of the measurements of the DecisionTree constructed by a DecisionTreeConstructor object. It keeps track of the training as well as the test SampleCollection, and runs each sample through the Decision Tree that was constructed, to find out its classification accuracy, which it stores and retrieves, when required.

4. **Discretizer** - This class provides implementations of algorithms used for discretization. As mentioned earlier, decision trees work with discretized values, and if continuous-valued features are present, they have to be discretized. The Discretizer class contains two algorithms for discretization:

   (a) A naive discretizer that discretizes data based on the median, with those values below the median being set to 0 and those values above the median being set to 1.

   (b) An Equal-Binning Discretizer that discretizes the values of certain feature of a collection of samples, by putting each value into particular bins. After discretization, the values can be any integer between 0 and binSize (inclusive).

6.3 org.ck.ga

This package takes care of all operations of the Genetic algorithm that was mentioned earlier.

1. **Genome** - This class takes as input, a SampleCollection, and initializes a chromosome with random values for the presence/absence of features, as defined in Chapter 5. It keeps track of this chromosome, and provides methods to manipulate this chromosome; to calculate the fitness score of this chromosome; and to throw an exception when the fitness value threshold has been crossed or when the best solution has been discovered. It also provides facilities to switch between a chromosome and the corresponding optimal decision tree to which it is bound.

2. **Population** - As defined earlier, a population is a collection of genomes. And this is exactly what this class is. Initially, the Population class randomly initializes a large number of genomes, of which it keeps track. It provides methods such as roulette selection, reproduction, crossover, and mutation to operate on the population and discover the best genome, and hence, the best decision tree with the appropriate feature subset.

3. **OptimalScoreException** - This class is responsible for catching the best genome as soon as it is discovered, since the best genome should never be allowed to escape. It should be caught and nurtured for future use.

6.4 org.ck.gui

As you've probably guessed by now, this package handles the Graphical User Interface of our application, with all its bells and whistles. We made use of the Standard Widget Toolkit for the GUI of our application. This package consists of the following classes:

1. **WelcomeWindow** - This class takes care of drawing the window that appears when our application is first switched on, and obviously, its name should be Wel-

comeWindow, nothing more, nothing less. It displays a list of clickable options, namely

(a) Train Decision Tree

(b) Classify Data Sets

(c) View on Github

(d) Exit Application

We have organized the code in this package in such a way that all the options *(except for the last one - "Exit Application")*, correspond to a different class which handles the creation of the corresponding window.

2. **MainWindow** - This class manages the window that is opened when a user clicks the *Train Decision Tree* option in the Welcome Window. In this window, the user can select the appropriate options required to construct a decision tree using our Hybrid DT/GA Classifier. By the way, did we mention that a constructed decision tree can be saved for later usage?

3. **ClassifyWindow** - This class manages the window that is opened when a user clicks the *Classify Data Sets* option in the Welcome Window. A user is provided with an interface to select a saved decision tree, and classify new samples based on it. It really saves a lot of time in this fast-paced world of ours.

4. **BrowserWindow** - This class manages the window that is opened when a user clicks the *View on Github* option in the Welcome Window. In order to see and verify whether our code is original or not, users can see the online repository of our code (including its version history) on Github, in this window. Verification couldn't have been more easier.

5. **Constants** - This interface (mark my words, this is not a class) contains a list of constants used by all the classes in all the packages. This interface really makes updating our software and meddling with various values much easier, like never before.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Summary

In this mini project, we were able to successfully implement and test the performance of Decision Tree-based classifiers. The Decision Tree classifier was optimized using a Genetic Algorithm to select a subset of the features that were to be used in constructing an optimal decision tree.

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 our application, which was developed in Java and Java SWT. We were able to extend this application to work with any generic samples. Two other samples/ Classification problems were addressed:

- Diagnosing whether a Horse has colic or is healthy, based on its Blood Sample Data.
- Classifying/Determining the quality of a wine based on Data Samples containing itś quality parameters

The hybrid GA /decision tree algorithm needs to be tested further to realize its true

potential. Clearly more work needs to be done. The test results show that the Decision Trees constructed using the Genetic algorithm-based feature selector, were more efficient and accurate in classifying the data than the Decision Trees constructed by selecting features manually.

7.2 Limitations

These are a few limitations of this application.

1. The application uses about 800MB-1GB of RAM.
2. The GA feature selector takes a considerable amount time to optimize the Decision Tree depending on the size of the training and testing samples.
3. The GA optimizer may take a long time to converge or may not converge at all, which usually results in the application crashing, due to high memory usage.
4. The Decision Tree classifier with GA-based feature selection requires the use of accurate as well as a large number of training and testing samples. The efficiency of the Decision Tree constructed is solely based on the input training and testing samples.

7.3 Future enhancements

Some of the future enhancements are :

1. The application could be made more responsive by using Threads and Parallel/-Cloud Computing
2. The Decision Tree Classifier of this application could be optimized using Neural Networks which are more efficient than Genetic Algorithms.
3. An interesting extension to be explored is the possibility of additional feedback from ID3 concerning the evaluation of a feature set.

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] Decision Trees - http://web.cecs.pdx.edu/~mm/MachineLearningWinter2010/pdfslides/DecisionTrees.pdf

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

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

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

[7] Kardi Tutorials on Decision Trees- http://people.revoledu.com/kardi/tutorial/DecisionTree/index.html

Appendices

Appendix A : Source Code

Listing 7.1: DecisionTreeConstructor.java

```
1  package org.ck.dt;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.util.ArrayList;
7  import java.util.HashMap;
8
9  import org.ck.ga.OptimalScoreException;
10 import org.ck.sample.DataHolder;
11 import org.ck.sample.Sample;
12 import org.ck.sample.SampleCollection;
13 import org.ck.sample.SampleSplitter;
14
15 /**
16  * This class will take the training data as input and build a DT
17  * and return the RootNode
18  */
19 public class DecisionTreeConstructor
20 {
21         private DecisionTreeNode RootNode;
22         private static final double MAX_PROBABILITY_STOPPING_CONDITION = 0.98; //
              Required by isStoppingCondition()
23
24         /*
25          * This constructor takes as a parameter - a collection of samples and constructs a
              MULTIWAY decision tree
```

```
26          */
27          public DecisionTreeConstructor(SampleCollection samples)
28          {
29                  RootNode = buildDecisionTree(samples.getSampleAsArrayList(), samples.
                        getfeatureList(), samples.getNumDiscreteClassesList());
30          }
31
32          /*
33           * This constructor takes as a parameter - a collection of samples, a subset of features
                constructs a MULTIWAY decision tree
34           * considering only those parameters in features.
35           */
36          public DecisionTreeConstructor(SampleCollection samples, ArrayList<String> features)
37          {
38                  RootNode = buildDecisionTree(samples.getSampleAsArrayList(), features,
                        samples.getNumDiscreteClassesList());
39          }
40
41
42
43          /*
44           * Takes as parameters - an arraylist of samples and an arraylist of features
45           *   Constructs a multiway decision tree recursively, and returns the root of the decision
                tree.
46           *   Makes use of the SampleSplitter class methods
47           */
48          public DecisionTreeNode buildDecisionTree(ArrayList<Sample> samples, ArrayList<
                String> featureList, HashMap<String, Integer> numDiscreteClassesList)
49          {
50                  //System.out.println("buildDecisionTree - "+samples.size()+"\t"+featureList
                        +" "+featureList.size());
51
```

```
52              //Base Condition
53              if ((samples.size() > 0 && isStoppingCondition(samples)) || (featureList.size()
                    ==0))
54              {
55
56                  DecisionTreeNode newleaf = new DecisionTreeNode();
57                  newleaf.setAsLeaf();
58
59                  newleaf.setClassifiedResult(getMajorityClass(samples));
60                  //System.out.println("New leaf Node - The classification is "+ newleaf
                        .getClassification());
61                  return newleaf;
62              }
63
64              /*
65               * Find a node for feature(0) and it's optimum value for splitting and initialize
                     the node
66               * split into left and right sample array lists, then call recursively
                     buildDecisionTree for left and right
67               * return node
68               */
69              int bestFeatureIndex = findBestSplitFeatureIndex(samples, featureList,
                    numDiscreteClassesList);
70
71              DecisionTreeNode new_test_node = new DecisionTreeNode(featureList.get(
                    bestFeatureIndex), numDiscreteClassesList.get(featureList.get(
                    bestFeatureIndex)));
72
73              SampleSplitter sampleSplitter = new SampleSplitter(samples, featureList.get(
                    bestFeatureIndex), numDiscreteClassesList.get(featureList.get(
                    bestFeatureIndex)));
```

```
74              sampleSplitter.splitSamples(); //Find an optimum value of the feature and Split
                    the samples into left and right sample subsets
75
76              String featureName = featureList.get(bestFeatureIndex);
77              featureList.remove(bestFeatureIndex);
78
79              //Creating the children nodes
80              for(int i = 0; i < numDiscreteClassesList.get(featureName); i++)
81              {
82                      ArrayList<Sample> sampleSubset = sampleSplitter.getSampleSubset(i);
83                      new_test_node.setChildNode(i, buildDecisionTree(sampleSubset, (
                            ArrayList<String>) featureList.clone(), numDiscreteClassesList));
84                      //new_test_node.setChildNode(i, buildDecisionTree(sampleSubset,
                            featureList, numDiscreteClassesList));
85              }
86
87              return new_test_node;
88      }
89
90      /*
91       * This method tries to split the samples based on every feature in featureList.
92       *  It returns the index of the feature in featureList which has the highest information
             gain.
93       */
94      private int findBestSplitFeatureIndex(ArrayList<Sample> samples, ArrayList<String>
            featureList, HashMap<String, Integer> numDiscreteClassesList)
95      {
96              double maxInformationGain = Double.MIN_VALUE;
97              int bestFeatureIndex = 0;
98
99              int index = 0;
100             for(String feature : featureList)
```

```
101             {
102                     SampleSplitter sampleSplitter = new SampleSplitter(samples, feature,
                            numDiscreteClassesList.get(feature));
103                     sampleSplitter.splitSamples(); //Find an optimum value of the feature
                            and Split the samples into left and right sample subsets
104
105                     //System.out.println(sampleSplitter.getInformationGain());
106
107                     if(sampleSplitter.getInformationGain() > maxInformationGain)
108                     {
109                             maxInformationGain = sampleSplitter.getInformationGain();
110                             bestFeatureIndex = index;
111                     }
112
113                     index++;
114             }
115
116             //System.out.println("Best = " + bestFeatureIndex + " " + featureList.get(
                        bestFeatureIndex));
117             return bestFeatureIndex;
118     }
119
120     /*
121      * Returns the class to which a majority of the samples belong
122      */
123     private String getMajorityClass(ArrayList<Sample> samples) {
124             int positive_class = 0, negative_class = 0;
125             for (Sample sample : samples)
126             {
127                     if (sample.getClassification().equals(DataHolder.getPositiveClass()))
                            positive_class++; else negative_class++;
128             }
```

```
129                 return positive_class>negative_class ? DataHolder.getPositiveClass():DataHolder
                        .getNegativeClass();
130         }
131
132         /*
133          * Returns true if the majority class of the samples is greater than 0.9
134          */
135         private boolean isStoppingCondition(ArrayList<Sample> samples) {
136                 int positive = 0;
137                 for (Sample sample : samples)
138                 {
139                         if (sample.getClassification().equals(DataHolder.getPositiveClass()))
                                positive++;
140                 }
141                 double prob_positive = (double)positive/samples.size();
142                 double prob_negative = 1 - prob_positive;
143                 double max = (prob_positive > 0.5)? prob_positive :prob_negative;
144                 return (max > MAX_PROBABILITY_STOPPING_CONDITION);
145         }
146
147         /*
148          * This method returns the RootNode of the DT
149          */
150         public DecisionTreeNode getDecisionTreeRootNode()
151         {
152                 return RootNode;
153         }
154 }
```

Listing 7.2: DecisionTreeClassifier.java

```
1 package org.ck.dt;
2
```

```
3  import java.util.ArrayList;
4
5  import org.ck.sample.Sample;
6  import org.ck.sample.SampleCollection;
7  import org.eclipse.swt.widgets.Tree;
8  import org.eclipse.swt.widgets.TreeItem;
9
10
11 /**
12  * This class is used to construct a DT based Classifier that builds a DT by creating
13  * a object of DecisionTreeBuilder class
14  *
15  */
16 public class DecisionTreeClassifier {
17         private DecisionTreeConstructor dtConstructor;
18         private DecisionTreeNode RootNode;
19         private SampleCollection trainingSamples;
20         private SampleCollection testingSamples;
21         private double Accuracy;
22
23
24         /*
25          * This constructor takes an object of SampleCollection and initializes the DT
26          * using DTConstructor method
27          */
28         public DecisionTreeClassifier(SampleCollection samples)
29         {
30                 this.trainingSamples = samples;
31                 this.dtConstructor = new DecisionTreeConstructor(samples);
32                 this.RootNode = this.dtConstructor.getDecisionTreeRootNode();
33         }
34
```

```
35
36          /*
37           * This constructor takes an object of SampleCollection and initializes the DT
38           * using DTConstructor method
39           */
40          public DecisionTreeClassifier(SampleCollection samples, ArrayList<String> features)
41          {
42                  this.trainingSamples = samples;
43                  this.dtConstructor = new DecisionTreeConstructor(samples,features);
44                  this.RootNode = this.dtConstructor.getDecisionTreeRootNode();
45          }
46
47          /*
48           * This method initializes the testingSamples variable
49           */
50          public void setTestingSamples(SampleCollection test_Samples)
51          {
52                  this.testingSamples = test_Samples;
53          }
54
55          /*
56           * This method uses the testingSamples and tests the accuracy of the
57           * decisiontree and initializes the Accuracy variable.
58           *
59           * Returns an arraylist of indices of all the samples that have been misclassified - This
                 was added for the GUI
60           */
61          public ArrayList<Integer> TestAndFindAccuracy()
62          {
63                  ArrayList<Sample> samples = testingSamples.getSampleAsArrayList();
64                  int errors = 0;
65
```

```
66              int index = 0;
67              ArrayList<Integer> errorIndices = new ArrayList<Integer>();
68              for(Sample sample : samples)
69              {
70                      String classifiedValue = Classify(sample);
71                      if (!classifiedValue.equals(sample.getClassification()))
72                      {
73                              //System.out.println("Classification Failed : " + "Actual Class
                                    is "+sample.getClassification());
74                              errorIndices.add(index);
75                              ++errors;
76                      }
77                      index++;
78              }
79              Accuracy = 1 - (double)errors/samples.size();
80
81              return errorIndices;
82      }
83
84      /*
85       * This method traverses the DT and Classifies the sample
86       */
87      public String Classify(Sample sample)
88      {
89              DecisionTreeNode treeNode = RootNode;
90              while(true)
91              {
92                      if(treeNode.isLeaf())
93                      {
94                              return treeNode.getClassification();
95                      }
96
```

```
97                      String feature = treeNode.getfeatureName();
98                      treeNode = treeNode.getChildNode((int)sample.getFeature(feature).
                            getValue());
99                  }
100         }
101         /*
102          * Returns the accuracy of the DT constructed
103          */
104         public double getAccuracy()
105         {
106             System.out.println("The Accuracy of the DT is "+Accuracy*100+"%");
107             return Accuracy;
108         }
109
110         /*
111          * Returns the current training samples based on which this decision tree was
                constructed
112          */
113         public SampleCollection getTrainingSamples()
114         {
115             return trainingSamples;
116         }
117
118         /*
119          * Returns the current testing samples based on which this decision tree was constructed
120          */
121         public SampleCollection getTestingSamples()
122         {
123             return testingSamples;
124         }
125
126         /*
```

```
127         * Sets the samples based on which this decision tree will be constructed
128         */
129        public void setTrainingSamples(SampleCollection samples)
130        {
131                trainingSamples = samples;
132        }
133
134        /*
135         * Takes a Tree SWT object and creates a graphical representation of the decision tree.
               This is a wrapper class
136         */
137        public void getGraphicalDecisionTree(Tree tree)
138        {
139                getGraphicalDecisionTree(tree, RootNode);
140        }
141
142        /*
143         * To reduce the number of lines of code, this method was made generic. Due to this,
               there is an
144         *   instanceof check to find the type-cast required wherever necessary.
145         */
146        private <T> void getGraphicalDecisionTree(T treeItem, DecisionTreeNode root)
147        {
148                if(root.isLeaf())
149                {
150                        TreeItem item;
151                        if(treeItem instanceof Tree)
152                                item = new TreeItem((Tree) treeItem, 0);
153                        else
154                                item = new TreeItem((TreeItem)treeItem, 0);
155                        item.setText(root.getClassification());
156                }
```

```
157                 else
158                 {
159                         for(int child = 0; child < root.getNumChildren(); child++)
160                         {
161                                 TreeItem item;
162                                 if(treeItem instanceof Tree)
163                                         item = new TreeItem((Tree) treeItem, 0);
164                                 else
165                                         item = new TreeItem((TreeItem)treeItem, 0);
166                                 item.setText(root.getfeatureName() + " = " + child + "?");
167
168                                 getGraphicalDecisionTree(item, root.getChildNode(child));
169                         }
170                 }
171         }
172 }
```

Appendix B : Screen Shots

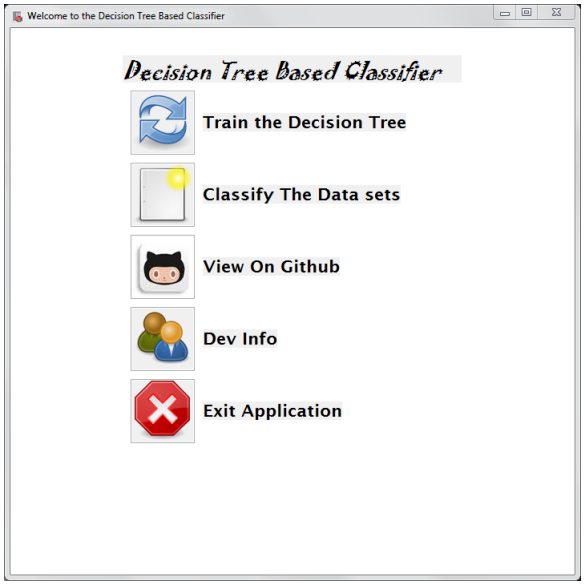

Figure 7.1: Application Window - Welcome Screen

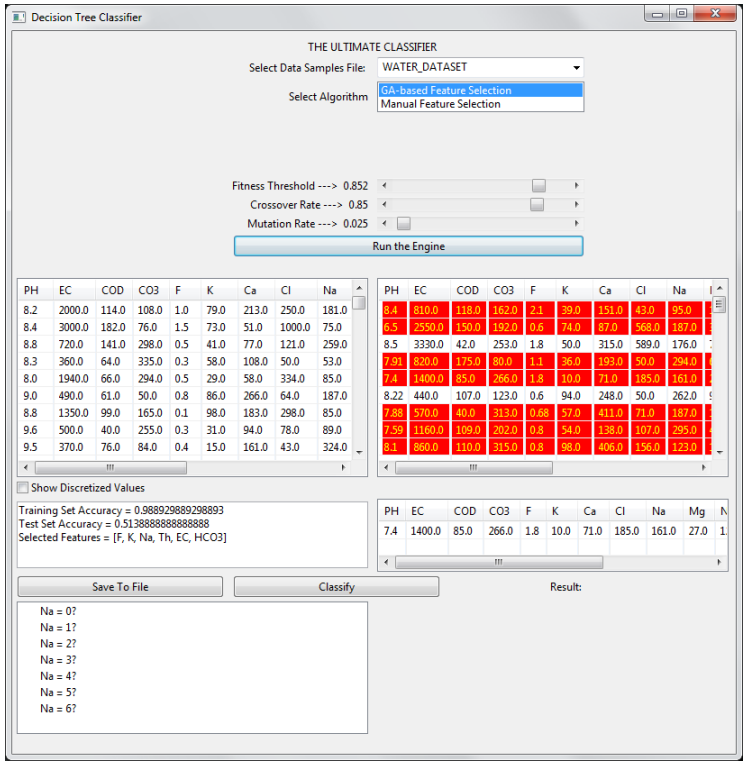

Figure 7.2: Decision Tree Constructor Window.

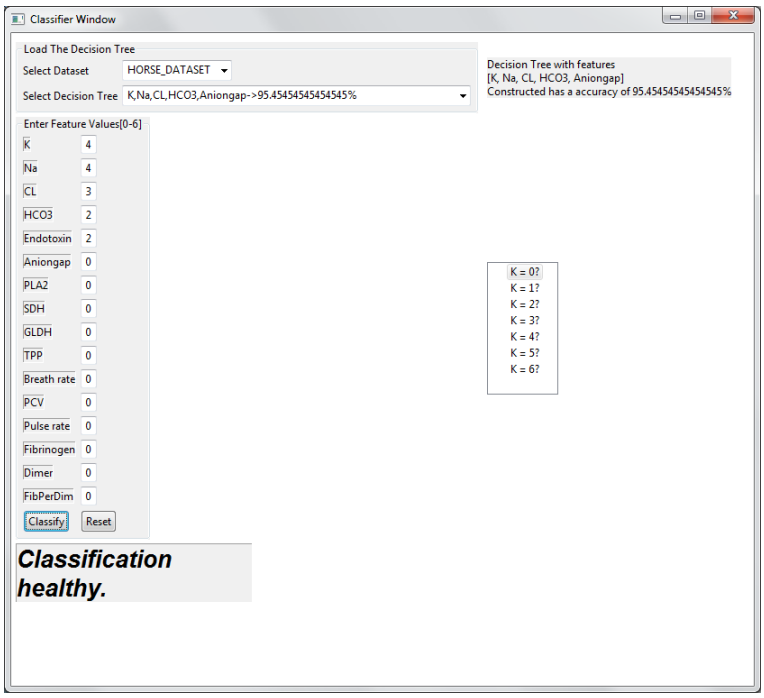

Figure 7.3: Decision Tree Classifier Window.

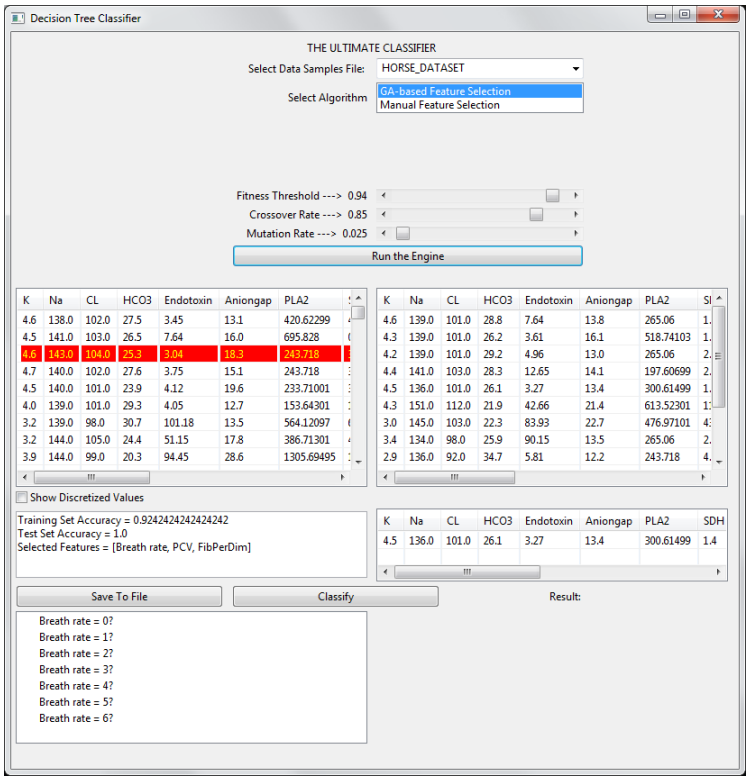

Figure 7.4: Decision Tree Construction with GA based Feature Selector.

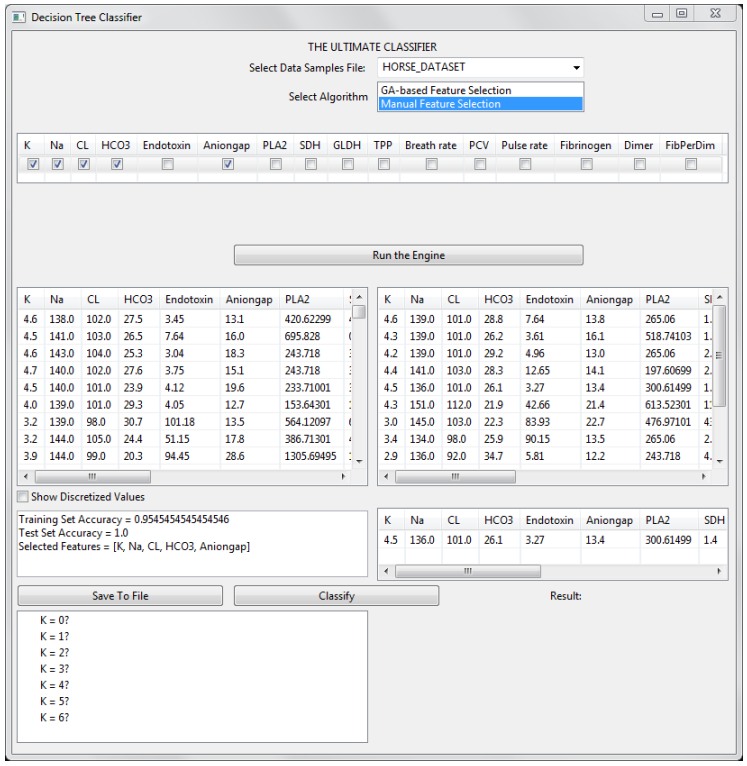

Figure 7.5: Decision Tree Construction with manual feature selection.

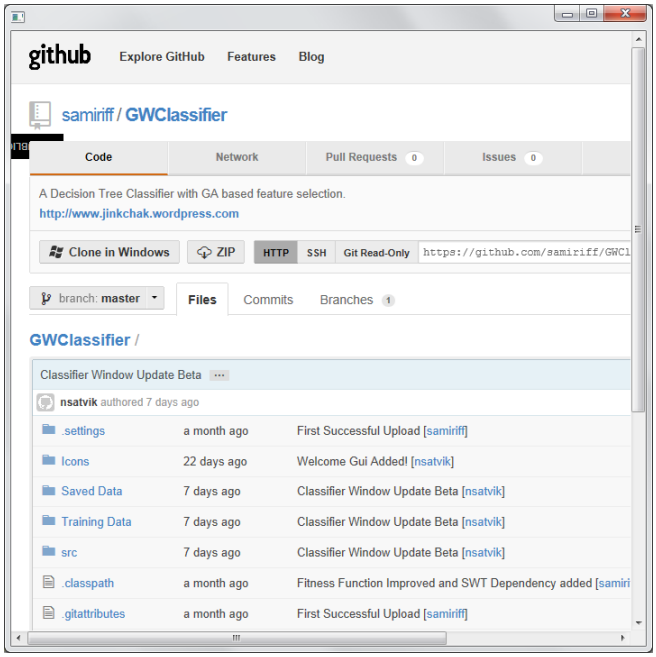

Figure 7.6: The project source code on github public repository.