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. 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)*

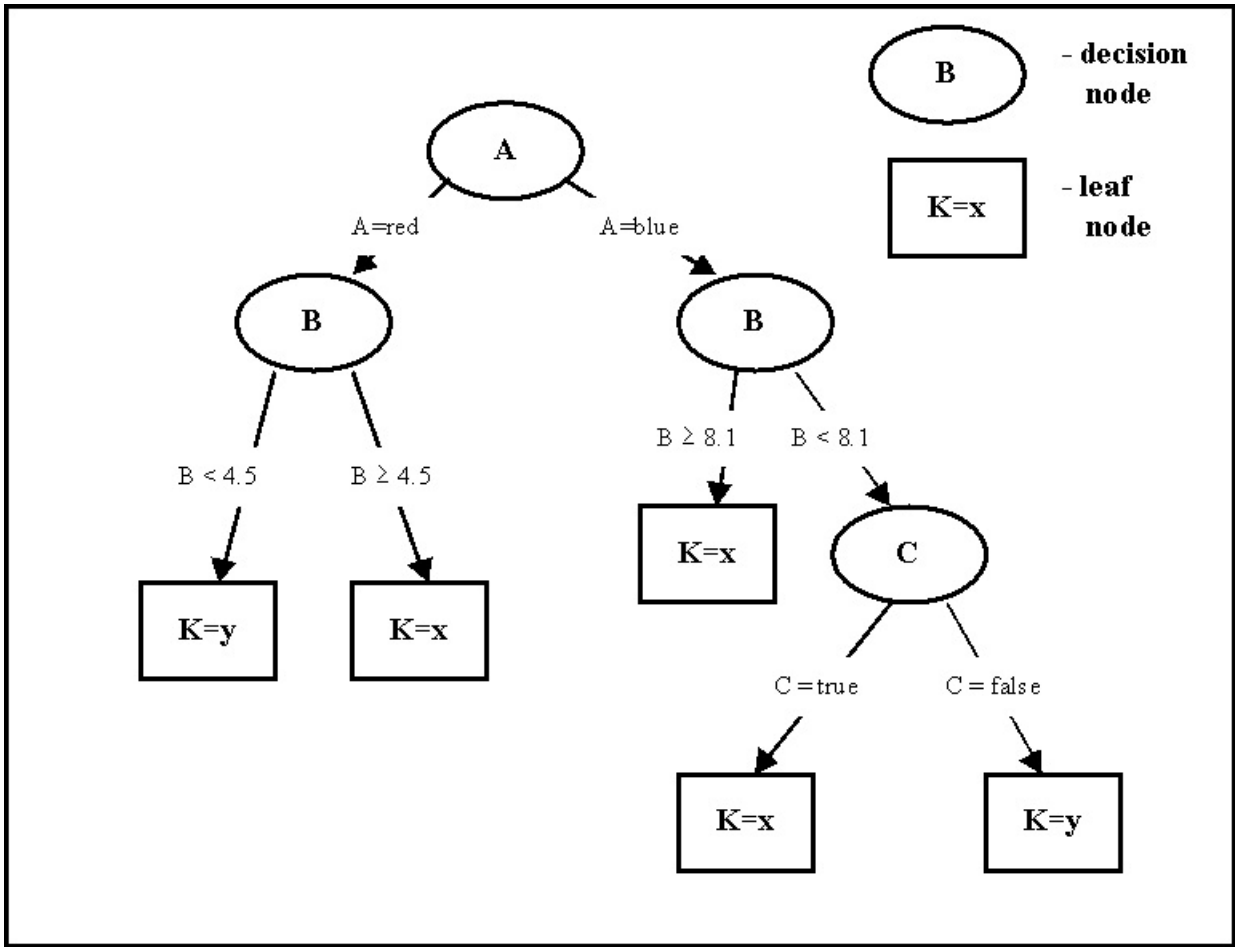

Figure 3.1: Sample Decision Tree

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

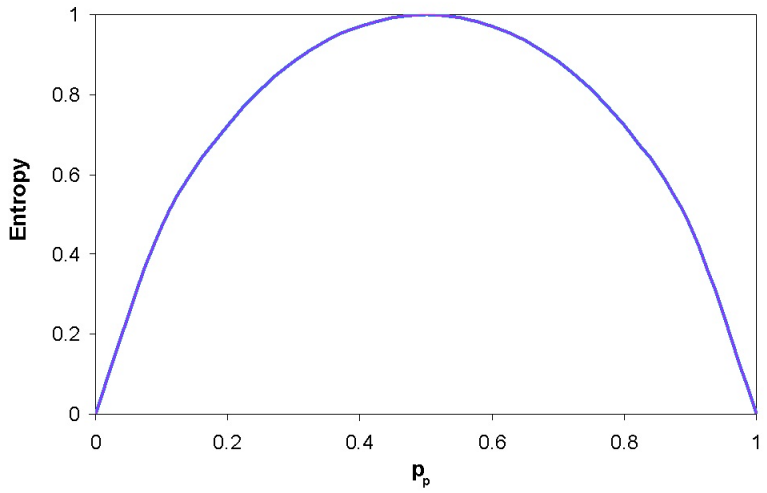

Figure 3.3: The entropy function relative to a binary classification, as the proportion of positive examples p_p varies between 0 and 1.

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.

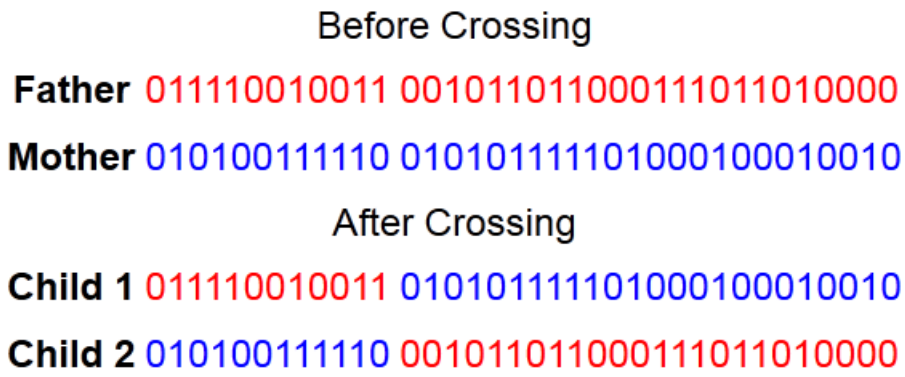

Figure 4.3: Crossover

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.

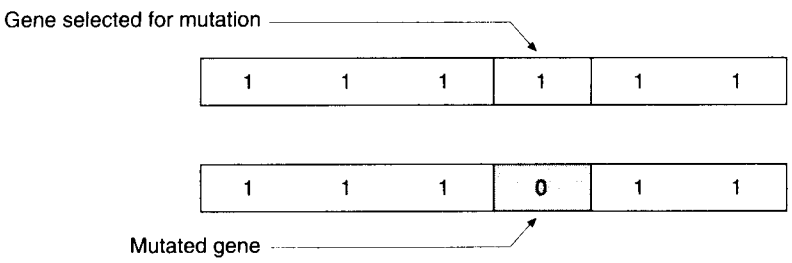

Figure 4.4: Mutation

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.

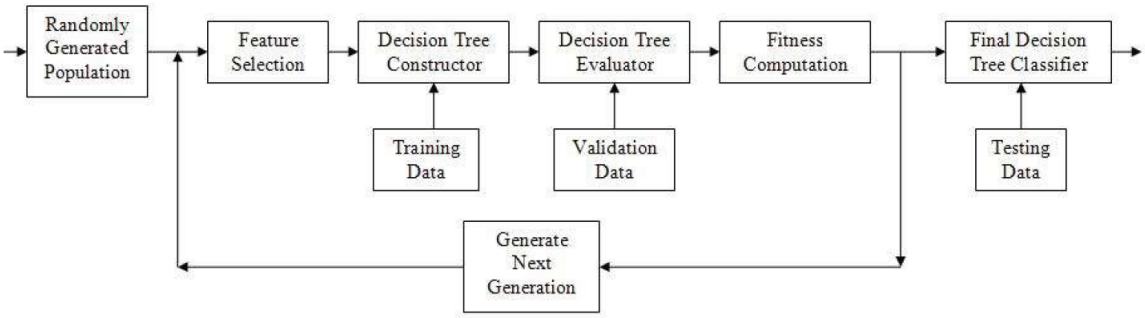

Figure 5.1: The data flow in DT/GA Hybrid Classifier Algorithm.

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.

6. **MainClass** - Before our application had a GUI, this class was used to test out the code in the other packages using the console. The SampleCaller2 method is still being used by the MainWindow class. We didn't have the heart to delete this class, which has been with us for so very long. We kept it for old times' sake.

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 Decision Trees.
3. An interesting extension to be explored is the possibility of additional feedback from ID3 concerning the evaluation of a feature set. Currently only classification

accuracy is returned. However, there is potentially exploitable information with respect to which features were actually used to build the decision tree and their relative positions in the tree.

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

Listing 7.1: DataHolder.java

```
1  package org.ck.sample;
2
3  import org.ck.gui.Constants;
4
5  public class DataHolder implements Constants{
6
7          private static String TRAINING_SAMPLES_FILE_NAME;
8          private static String TESTING_SAMPLES_FILE_NAME ;
9          private static String ATTRIBUTES_FILE_NAME ;
10         private static String SAVE_DATA_TO_FILE;
11         private static String POSITIVE_CLASS;
12         private static String NEGATIVE_CLASS;
13         private static String CURRENT_DATASET;
14         private static double FITNESS_SCORE_THRESHOLD;
15         private static double CROSSOVER_PROBABILITY_THRESHOLD;
16         private static double MUTATION_PROBABILITY_THRESHOLD;
17
18         static
19         {
20                 setDataset(DatasetOptions.HORSE_DATASET);
21                 setFitnessScoreThreshold(Constants.FITNESS_SCORE_THRESHOLD);
22                 setCrossoverProbabilityThreshold(Constants.
                       CROSSOVER_PROBABILITY_THRESHOLD);
23                 setMutationProbabilityThreshold(Constants.
                       MUTATION_PROBABILITY_THRESHOLD);
24         }
25
```

```
26          /**
27           * The option indicates the desired data sets to be used and
28           * depending on this option, the training sample and testing sample file
29           * are initialized.
30           */
31          public DataHolder(DatasetOptions option)
32          {
33                  setDataset(option);
34          }
35
36          /*
37           * This method sets the
38           */
39          public static void setDataset(DatasetOptions option)
40          {
41                  switch(option)
42                  {
43                  case HORSE_DATASET: TRAINING_SAMPLES_FILE_NAME = "Training Data
                        /Horse/horse.train";
44                                                  TESTING_SAMPLES_FILE_NAME =
                                                        "Training Data/Horse/horse.test"
                                                        ;
45                                                  ATTRIBUTES_FILE_NAME = "
                                                        Training Data/Horse/horse.
                                                        attribute";
46                                                  POSITIVE_CLASS = "healthy.";
47                                                  NEGATIVE_CLASS = "colic.";
48                                                  CURRENT_DATASET = "
                                                        HORSE_DATASET";
49                                                  SAVE_DATA_TO_FILE = "Saved
                                                        Data/HorseDT";
50                                                  break;
```

```
51
52              case WHINE_DATASET: TRAINING_SAMPLES_FILE_NAME = "Training Data
                    /Whine/whine.train";
53                                                  TESTING_SAMPLES_FILE_NAME =
                                                        "Training Data/Whine/whine.test
                                                        ";
54                                                  ATTRIBUTES_FILE_NAME = "
                                                        Training Data/Whine/whine.
                                                        attribute";
55                                                  POSITIVE_CLASS = "excellent.";
56                                                  NEGATIVE_CLASS = "poor.";
57                                                  CURRENT_DATASET = "
                                                        WHINE_DATASET";
58                                                  SAVE_DATA_TO_FILE = "Saved
                                                        Data/WhineDT";
59                                                  break;
60              default :
61              case WATER_DATASET: TRAINING_SAMPLES_FILE_NAME = "Training Data
                    /Water/water.train";
62                                                  TESTING_SAMPLES_FILE_NAME =
                                                        "Training Data/Water/water.test
                                                        ";
63                                                  ATTRIBUTES_FILE_NAME = "
                                                        Training Data/Water/water.
                                                        attribute";
64                                                  POSITIVE_CLASS = "potable.";
65                                                  NEGATIVE_CLASS = "not potable.";
66                                                  CURRENT_DATASET = "
                                                        WATER_DATASET";
67                                                  SAVE_DATA_TO_FILE = "Saved
                                                        Data/WaterDT";
68
```

```
69                  }
70          }
71          public static String getSaveDatoToFileName()
72          {
73                  return SAVE_DATA_TO_FILE;
74          }
75
76          public static String getTrainingSamplesFileName()
77          {
78                  return TRAINING_SAMPLES_FILE_NAME;
79          }
80          public static String getTestingSamplesFileName()
81          {
82                  return TESTING_SAMPLES_FILE_NAME;
83          }
84          public static String getAttributesFileName()
85          {
86                  return ATTRIBUTES_FILE_NAME;
87          }
88
89          public static String getPositiveClass()
90          {
91                  return POSITIVE_CLASS;
92          }
93
94          public static String getNegativeClass()
95          {
96                  return NEGATIVE_CLASS;
97          }
98
99          public static void setFitnessScoreThreshold(double value)
100         {
```

```
101             FITNESS_SCORE_THRESHOLD = value;
102         }
103
104         public static double getFitnessScoreThreshold()
105         {
106             return FITNESS_SCORE_THRESHOLD;
107         }
108
109         public static void setCrossoverProbabilityThreshold(double value)
110         {
111             CROSSOVER_PROBABILITY_THRESHOLD = value;
112         }
113
114         public static double getCrossoverProbabilityThreshold()
115         {
116             return CROSSOVER_PROBABILITY_THRESHOLD;
117         }
118
119         public static void setMutationProbabilityThreshold(double value)
120         {
121             MUTATION_PROBABILITY_THRESHOLD = value;
122         }
123
124         public static double getMutationProbabilityThreshold()
125         {
126             return MUTATION_PROBABILITY_THRESHOLD;
127         }
128
129         public static String getCurrentDataSet() {
130
131             return CURRENT_DATASET;
132         }
```

```
133 }
```

Listing 7.2: Feature.java

```
1  package org.ck.sample;
2
3  /**
4   * A Feature class to represent each feature of the sample like pH, CO3, NO3, NC, NH3 ...(
        indicated by the
5   * feature name), private variables to indicate it's lower and upper limit(I don't think these are
        needed for
6   * each instance, we'll redesign this),
7   *
8   *
9   */
10 public class Feature {
11         private String featureName;
12         private double featureValue;
13
14         /*
15          * Initializes the current feature with a Feature name and its corresponding value
16          */
17         public Feature(String name, double value)
18         {
19                 featureName = name;
20                 featureValue = value;
21         }
22
23         /*
24          * Displays the name and value of the feature
25          */
26         public void display()
27         {
```

```
28                  System.out.print(featureName + " = " + featureValue + "\t");
29          }
30
31          /*
32           * Returns the Name of the feature
33           */
34          public String getName()
35          {
36                  return featureName;
37          }
38
39          /*
40           * Returns the value of the feature
41           */
42          public double getValue()
43          {
44                  return featureValue;
45          }
46  }
```

Listing 7.3: Sample.java

```
1   package org.ck.sample;
2
3   import java.io.BufferedReader;
4   import java.io.File;
5   import java.io.FileNotFoundException;
6   import java.io.FileReader;
7   import java.io.IOException;
8   import java.util.ArrayList;
9   import java.util.HashMap;
10  import java.util.StringTokenizer;
11
```

```
12 import org.ck.dt.Discretizer;
13
14
15 /**
16  * Sample class indicates each horse sample.
17  *
18  *
19  *
20  */
21 public class Sample {
22
23         private HashMap<String, Feature> featureMap;  //Maps a feature name to its
               corresponding value
24         private ArrayList<String> featureList;   //List of features to be tracked. It helps in
               retrieving feature values from the Hashmap in the proper order.
25
26         private String classifiedResult;
27
28         /**
29          * Constructor that takes a string containing values for all the features, and a list of
               attributes for
30          * initialization
31          * @param featureString - A comma-separated string of values for all the features
32          * @param attributeList - An ArrayList of names of attributes (featureNames)
33          */
34         public Sample(String featureString, ArrayList<String> featureList)
35         {
36                 this.featureList = featureList;
37                 int currentFeature = 0;
38                 featureMap = new HashMap<String, Feature>();
39
40                 StringTokenizer tokens = new StringTokenizer(featureString, ",");
```

```
41
42              while(tokens.hasMoreTokens())
43              {
44                      String token = tokens.nextToken();
45                      if(currentFeature < featureList.size())
46                      {
47                              featureMap.put(featureList.get(currentFeature), new Feature(
                                    featureList.get(currentFeature), Double.parseDouble(token)
                                    ));
48                              currentFeature++;
49                      }
50                      else
51                      {
52
53                              if(token.equals(DataHolder.getPositiveClass()))
54                                      classifiedResult = DataHolder.getPositiveClass();
55                              else
56                                      classifiedResult = DataHolder.getNegativeClass();
57                              currentFeature = 0;
58                      }
59              }
60      }
61
62      /*
63       * This constructor just initializes the Feature Map and the attribute list from a parent
             Sample
64       *    with a subset of attributes
65       */
66      public Sample(Sample parentSample, ArrayList<String> subFeatureList)
67      {
68              featureMap = new HashMap<String, Feature>();
69              this.featureList = subFeatureList;
```

```
70
71              for(String feature : subFeatureList)
72              {
73                      Feature attribute = parentSample.featureMap.get(feature);
74                      featureMap.put(feature, attribute);
75              }
76
77              classifiedResult = parentSample.classifiedResult;
78      }
79
80      /*
81       * Returns a new Sample which contains values corresponding to the attributes in
             subAttributeList (subset)
82       */
83      public Sample getSampleSubset(ArrayList<String> subFeatureList)
84      {
85              return new Sample(this, subFeatureList);
86      }
87
88
89      /*
90       * Displays all features of the sample included the classified Result
91       */
92      public void display()
93      {
94              for(String feature : featureList)
95              {
96                      if(featureMap.containsKey(feature))
97                              featureMap.get(feature).display();
98              }
99
100             System.out.print("Classification = " + classifiedResult);
```

```
101         }
102
103         /*
104          * Returns the value of the feature that corresponds to the feature name stored in the
                attribute parameter
105          * @param attribute − contains the name of the feature whose value is to be returned
106          */
107         public Feature getFeature(String feature)
108         {
109                 if(featureMap.containsKey(feature))
110                         return featureMap.get(feature);
111                 return null;
112         }
113
114         /*
115          * If the name of the feature exists in the Feature Map, this method changes the
                corresponding entry to parameter "feature"
116          *  Returns true, if operation is successful, else returns false
117          */
118         public boolean setFeature(Feature feature)
119         {
120                 if(featureMap.containsKey(feature.getName()))
121                 {
122                         featureMap.put(feature.getName(), feature);
123                         return true;
124                 }
125
126                 return false;
127         }
128
129         /*
130          * Returns the class to which this sample belongs
```

```
131             */
132             public String getClassification()
133             {
134                     return classifiedResult;
135             }
136
137             /*
138              * To be used ONLY IN EMERGENCIES...The normal way is to use the discretizer
                    method of SampleCollection
139              */
140             public void discretize(String featureName, double delta, double min)
141             {
142                     Discretizer.discretizeSample(this, featureName, delta, min);
143             }
144     }
```

Listing 7.4: SampleCollection.java

```
1   package org.ck.sample;
2
3   import java.io.BufferedReader;
4   import java.io.FileNotFoundException;
5   import java.io.FileReader;
6   import java.io.IOException;
7   import java.util.ArrayList;
8   import java.util.HashMap;
9
10  import org.ck.dt.Discretizer;
11  import org.ck.gui.Constants;
12
13
14  /**
```

```
15  * This class reads the sample data from a file and initializes all the necessary data structures to
        store
16  * the data values for Classification analysis.
17  */
18
19  public class SampleCollection implements Constants
20  {
21          private ArrayList<Sample> samples;
22          private ArrayList<String> featureList;
23          private HashMap<String, Integer> featureNumDiscreteClasses;   //To keep track of the
                number of values of each attribute after discretization
24
25          private class BinningVars
26          {
27                  public double minValue;
28                  public double delta;
29
30                  public BinningVars(double d, double m)
31                  {
32                          delta = d;
33                          minValue = m;
34                  }
35          }
36          private ArrayList<BinningVars> binningVars;   //Used to keep track of the Equal
                Binning variables - delta and min of each feature
37
38
39  // private String trainingSamplesFilename;        //Name of the file that has training examples
40  // private String featuresFilename;         //Name of the file that has the list of attributes
41
42          /*
```

```
43          * Constructor that takes as parameters, the file name of the file containing all the data
                samples,
44          *   and the file name of the file that contains the list of features required to describe
                each
45          *   data sample
46          */
47          public SampleCollection(String samplesFileName, String featuresFileName)
48          {
49  //  trainingSamplesFilename = samplesFileName;
50  //  this.featuresFilename = featuresFileName;
51
52                  samples = new ArrayList<Sample>();
53
54                  try {
55                          this.featureList = getfeatureList(featuresFileName);
56
57                          BufferedReader sampleFile = new BufferedReader(new FileReader(
                                samplesFileName));
58                          while(true)
59                          {
60                                  String line = sampleFile.readLine();
61                                  if(line == null)
62                                          break;
63                                  samples.add(new Sample(line, featureList));
64                          }
65                          sampleFile.close();
66
67                  } catch (FileNotFoundException e) {
68                          e.printStackTrace();
69                  } catch (IOException e) {
70                          e.printStackTrace();
71                  }
```

```
72
73              //featureNumDiscreteClasses = new int[featureList.size()];
74              featureNumDiscreteClasses = new HashMap<String, Integer>();
75
76              binningVars = new ArrayList<SampleCollection.BinningVars>();
77              for(int i=0; i<featureList.size(); i++)
78                      binningVars.add(null);
79      }
80
81      /*
82       * This constructor initializes a SampleCollection with features specified in subfeatureList
             ,
83       *  from a parent sample collection that has already been defined. (subset)
84       */
85      public SampleCollection(SampleCollection parentSampleCollection, ArrayList<String>
           subfeatureList)
86      {
87              featureList = subfeatureList;
88              featureNumDiscreteClasses = parentSampleCollection.featureNumDiscreteClasses
                    ;
89
90              ArrayList<Sample> samplesSubset = new ArrayList<Sample>();
91              for(Sample sample : parentSampleCollection.samples)
92              {
93                      Sample subSample = sample.getSampleSubset(subfeatureList);
94                      samplesSubset.add(subSample);
95              }
96
97              samples = samplesSubset;
98      }
99
100     /*
```

```
101         * Returns a new SampleCollection with features specified in subfeatureList,
102         *  from a this sample collection as a parent table
103         */
104        public SampleCollection getSampleCollectionSubset(ArrayList<String> subfeatureList)
105        {
106                return new SampleCollection(this, subfeatureList);
107        }
108
109
110        /*
111         * This method is used to call the desired method in the Discretizer class that will
                convert
112         *  continuous valued features of the current sample collection to discrete-valued
                features.
113         * The algorithm for discretized is specified by the parameter, which can be any constant
                of the
114         *  enum - DiscretizerAlgorithms, defined in the Constants interface.
115         */
116        public void discretizeSamples(DiscretizerAlgorithms algorithmType)
117        {
118                switch(algorithmType)
119                {
120                default:
121
122                case MEDIAN:
123                        for(int i=0; i<featureList.size(); i++)
124                        {
125                                Discretizer.discretizeBasedOnMedian(this, i);
126                        }
127                        break;
128
129                case EQUAL_BINNING:
```

```
130                     int numDiscreteClasses = NUMBER_OF_BINS;
131                     for(int i=0; i<featureList.size(); i++)
132                     {
133                             Discretizer.discretizeEqualBinner(this, i, numDiscreteClasses);
134                     }
135                     break;
136             }
137     }
138
139     /*
140      * Displays all the samples that have been stored by the program
141      */
142     public void displaySamples()
143     {
144             for(Sample sample : samples)
145             {
146                     sample.display();
147                     System.out.println();
148             }
149     }
150
151     /*
152      * Returns the arraylist containing strings of features (features)
153      */
154     public ArrayList<String> getfeatureList()
155     {
156             return featureList;
157     }
158
159
160     /*
161      * Displays the contents of the featureList arraylist
```

```
162         */
163         public void displayfeatureList()
164         {
165                 int i = 0;
166                 System.out.println("features: ");
167                 for(String feature : featureList)
168                         System.out.println(i++ + " " + feature);
169         }
170         /*
171          * This method returns the ArrayList of samples
172          */
173         public ArrayList<Sample> getSampleAsArrayList()
174         {
175                 return samples;
176         }
177
178         /*
179          * Sets the number of discrete values for a given feature after discretization
180          */
181         public void setNumDiscreteClasses(int featureIndex, int numValues)
182         {
183                 featureNumDiscreteClasses.put(featureList.get(featureIndex), numValues);
184         }
185
186         /*
187          * Returns the number of discrete values for a given feature after discretization
188          */
189         public int getNumDiscreteClasses(String feature)
190         {
191                 return featureNumDiscreteClasses.get(feature);
192         }
193
```

```
194         /*
195          * Returns the array containing the number of discrete values for a given feature after
                 discretization
196          */
197         public HashMap<String, Integer> getNumDiscreteClassesList()
198         {
199                 //return Arrays.copyOf(featureNumDiscreteClasses, featureNumDiscreteClasses.
                        length);
200                 return (HashMap<String, Integer>) featureNumDiscreteClasses.clone();
201         }
202
203         /*
204          * Returns details of the filenames from which this collection read its samples.
205          */
206         public String getSamplesFilename(Filenames type)
207         {
208                 switch(type)
209                 {
210                 case TRAINING_SAMPLES_FILE: return DataHolder.
                        getTrainingSamplesFileName();
211                 case FEATURES_FILE: return DataHolder.getAttributesFileName();
212                 default: return null;
213                 }
214         }
215
216
217         /*
218          * Reads a file containing the list of features (feature names) necessary for describing
                 each sample
219          */
220         private static ArrayList<String> getfeatureList(String filename)throws IOException
221         {
```

```
222             BufferedReader br = new BufferedReader(new FileReader(filename));
223             ArrayList<String> featureList = new ArrayList<String>();
224
225             while(true)
226             {
227                     String line = br.readLine();
228
229                     if(line == null)
230                             break;
231
232                     featureList.add(line);
233             }
234
235             return featureList;
236     }
237
238     /*
239      * Adds a new entry (delta, minValue) to the arraylist of binningVars at index i
240      */
241     public void addBinningVar(int index, double delta, double minValue)
242     {
243             binningVars.set(index, new BinningVars(delta, minValue));
244     }
245
246     /*
247      * Returns the delta value of the bin at index
248      */
249     public double getBinningVarDelta(int index)
250     {
251             return binningVars.get(index).delta;
252     }
253
```

```
254         /*
255          * Returns the min value of the bin at index
256          */
257         public double getBinningVarMinvalue(int index)
258         {
259                 return binningVars.get(index).minValue;
260         }
261
262         /*
263          * Displays the bin values for each feature of the sample collection
264          */
265         public void displayBinning()
266         {
267                 for(int i=0; i<binningVars.size(); i++)
268                         System.out.println(binningVars.get(i).delta + "\t" + binningVars.get(i).
                                minValue);
269         }
270
271         /*
272          * Discretizes a sample based on the Binning values of this sample collection
273          */
274         public void discretizeSample(Sample sample)
275         {
276                 for(int i=0; i<featureList.size(); i++)
277                         sample.discretize(featureList.get(i), binningVars.get(i).delta, binningVars
                                .get(i).minValue);
278         }
279
280         public void discretizeSamplesBasedOnOtherSampleCollection(SampleCollection
              trainingSampleCollection)
281         {
282                 for(Sample sample : samples)
```

```
283                 {
284                         trainingSampleCollection.discretizeSample(sample);
285                 }
286         }
287 }
```

Listing 7.5: SampleSplitter.java

```
1  package org.ck.sample;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5
6  /*
7   * Used to find an optimal way to split samples based on a given feature
8   */
9  public class SampleSplitter {
10
11          private ArrayList<Sample> samples;
12          private String feature_name;
13
14          private double Optimum_feature_value;
15          private ArrayList<Sample> leftsampleSubset = new ArrayList<Sample>(); //Not
                needed now
16          private ArrayList<Sample> rightsampleSubset = new ArrayList<Sample>(); //Not
                needed now
17
18          private ArrayList<Sample> sampleSubsets[];  //An array of arraylists
19
20          /*
21           * This constructor initializes the class variables, and specifies that the list of samples
                should be
22           *  split based on the parameter - feature.
```

```
23          */
24          public SampleSplitter(ArrayList<Sample> samples, String feature, int
                numDiscreteClasses)
25          {
26                  this.samples = samples;
27                  this.feature_name = feature;
28
29                  //Initializing the Array of the arraylists of samples, that will contain the split
                        subsets for a multiway decision tree
30                  sampleSubsets = (ArrayList<Sample>[])new ArrayList[numDiscreteClasses];
31                  for(int i=0; i<sampleSubsets.length; i++)
32                          sampleSubsets[i] = new ArrayList<Sample>();
33          }
34
35          /*
36           * Splits the given sample set into left and right samples based on the median of all
                values of the given
37           *  feature. Duplicate feature values aren't considered. The median is stored in the
                variable
38           *   Optimum_feature_value.
39           */
40          public void splitSamples() {
41
42                  for (Sample sample : samples )
43                  {
44                          sampleSubsets[(int)sample.getFeature(feature_name).getValue()].add(
                                sample);
45                  }
46          }
47
48          /*
49           * Returns the value based on which the data is split into left and right subsets.
```

```
50          */
51          public double getOptimumValue()
52          {
53                  return Optimum_feature_value;
54          }
55
56          /*
57           * Returns the samples for which the given feature has values lesser than the Optimum
                value.
58           */
59          public ArrayList<Sample> getLeftSampleSubset()
60          {
61                  return leftsampleSubset;
62          }
63
64          /*
65           * Returns the samples for which the given feature has values greater than the Optimum
                value.
66           */
67          public ArrayList<Sample> getRightSampleSubset()
68          {
69                  return rightsampleSubset;
70          }
71
72          /*
73           * Returns the sample subset of the "index"th partition
74           */
75          public ArrayList<Sample> getSampleSubset(int index)
76          {
77                  return sampleSubsets[index];
78          }
79
```

```
80          /*
81           * Returns the Information Gain of the current split, calculated using the formula:
82           *   Entropy of Parent Table − Sum(k/n ∗ Entropy of subsetTable i)
83           */
84          public double getInformationGain()
85          {
86                  double informationGain = 0.0;
87
88                  for(int i=0; i<sampleSubsets.length; i++)
89                  {
90                          informationGain += ((double)sampleSubsets[i].size() / (double)
                                samples.size()) ∗ getEntropy(sampleSubsets[i]);
91                  }
92
93                  informationGain = getEntropy(samples) − informationGain;
94
95                  return informationGain;
96          }
97
98          /*
99           * Returns the entropy of the given sample list, calculated by the formula
100          *   sum ( − p ln(p) )
101          */
102         private double getEntropy(ArrayList<Sample> samples)
103         {
104                 HashMap<String, Double> groups = new HashMap<String, Double>();
105
106                 //Find number of samples for each classification
107                 for(Sample sample : samples)
108                 {
109                         String classification = sample.getClassification();
110                         if(groups.containsKey(classification))
```

```
111                                 groups.put(classification, groups.get(classification) + 1);
112                         else
113                                 groups.put(classification, 1.0);
114                 }
115
116                 double entropy = 0.0;
117                 for(String key : groups.keySet())
118                 {
119                         double probability = groups.get(key) / samples.size();
120                         entropy += - (probability * Math.log(probability) / Math.log(2));
121                 }
122
123                 return entropy;
124         }
125 }
```

Listing 7.6: DecisionTreeNode.java

```
1  package org.ck.dt;
2
3  /**
4   * This class is used to create objects that represent nodes in a BINARY decision tree
5   */
6  public class DecisionTreeNode
7  {
8          private String featureName; //Name of the feature this node indicates
9          private double lowerLimit;
10         private double upperLimit;
11
12         private boolean isLeaf; //Is this node a leaf
13         private String ClassifiedResult;//If this is the leaf node what is the classified result
14
15         //private DecisionTreeNode Left, Right;//Left and Right Nodes
```

```
16
17          private DecisionTreeNode children[];
18
19          /*
20           * This constructor Initializes the class variables with default values
21           *   By default, this node is not a leaf and contains no children.
22           */
23          public DecisionTreeNode()
24          {
25                  //Left = Right = null;
26                  lowerLimit = upperLimit = 0;
27                  isLeaf = false;
28                  children = null;
29          }
30
31
32          /*
33           * This constructor initializes the feature name of the (internal) node
34           */
35          public DecisionTreeNode(String feature_name)
36          {
37                  this.featureName = feature_name;
38          }
39
40          /*
41           * This constructor initializes the feature name and the number of children of the (
                internal) node.
42           */
43          public DecisionTreeNode(String feature_name, int numChildren)
44          {
45                  this(feature_name);
46                  children = new DecisionTreeNode[numChildren];
```

```
47          }
48
49          /*
50           * Converts the current node to a leaf node
51           */
52          public void setAsLeaf()
53          {
54                  isLeaf = true;
55          }
56
57          /*
58           * Returns true if this node is a leaf node
59           */
60          public boolean isLeaf()
61          {
62                  return isLeaf;
63          }
64
65          /*
66           * Sets lower limit of node feature value
67           */
68          public void setLowerLimit(double low_value)
69          {
70                  lowerLimit = low_value;
71          }
72
73          /*
74           * Sets upper limit of node feature value
75           */
76          public void setUpperLimit(double high_value)
77          {
78                  upperLimit = high_value;
```

```
79          }
80
81          /*
82           * Returns the value of UpperLImit variable
83           */
84          public double getUpperLimit()
85          {
86                  return upperLimit;
87          }
88
89          /*
90           * Can be used to modify the node's feature name
91           */
92          public void setfeatureName(String feature_name)
93          {
94                  featureName = feature_name;
95          }
96
97          /*
98           * Returns the node's feature name
99           */
100         public String getfeatureName()
101         {
102                 return featureName;
103         }
104
105         /*
106          * Initializes the node's left child
107          */
108         public void setLeftNode(DecisionTreeNode left)
109         {
110                 //this.Left = left;
```

```
111         }
112
113         /*
114          * Returns the left child of this node
115          */
116         public DecisionTreeNode getLeftNode()
117         {
118                 return null; //Left;
119         }
120
121         /*
122          * Returns the right child of this node
123          */
124         public DecisionTreeNode getRightNode()
125         {
126                 return null; //Right;
127         }
128
129
130         /*
131          * Initializes the node's right child
132          */
133         public void setRightNode(DecisionTreeNode right)
134         {
135                 //this.Right = right;
136         }
137
138
139         /*
140          * Returns the child node at index
141          */
142         public DecisionTreeNode getChildNode(int index)
```

```
143         {
144                 return children[index];
145         }
146
147         /*
148          * Initializes the current node's child at index
149          */
150         public void setChildNode(int index, DecisionTreeNode node)
151         {
152                 children[index] = node;
153         }
154
155         /*
156          * Sets the classification of a leaf node
157          */
158         public void setClassifiedResult(String Class)
159         {
160                 this.ClassifiedResult = Class;
161         }
162
163         /*
164          * Returns the classification of a leaf node
165          */
166         public String getClassification()
167         {
168                 return ClassifiedResult;
169         }
170
171         /*
172          * Returns the number of children of the given node
173          */
174         public int getNumChildren()
```

```
175         {
176                 if(children != null)
177                         return children.length;
178                 return 0;
179         }
180 }
```

Listing 7.7: 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 7.8: 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 }
```

Listing 7.9: Discretizer.java

```
1  package org.ck.dt;
2
3  import java.util.ArrayList;
4
5  import org.ck.gui.Constants;
6  import org.ck.sample.Feature;
7  import org.ck.sample.Sample;
8  import org.ck.sample.SampleCollection;
9
10
```

```
11 public class Discretizer implements Constants
12 {
13         /* **********************************ALGORITHM 1
               ****************************************/
14         /*
15          * A naive discretizer that discretizes data based on the median, with those values
16          *  below the median being set to 0 and those values above the median being set to 1.
17          */
18         public static void discretizeBasedOnMedian(SampleCollection samples, int featureIndex)
19         {
20                 //Median Calculation (without considering duplicates)
21                 ArrayList<Double> FeatureValueList = new ArrayList<Double>();
22                 for (Sample sample : samples.getSampleAsArrayList()) {
23                         double val = sample.getFeature(samples.getfeatureList().get(
                               featureIndex)).getValue();
24                         //if (!FeatureValueList.contains(val))
25                         {
26                                 FeatureValueList.add(val);
27                         }
28                 }
29
30                 //Converting the continuous values to discrete values
31                 ArrayList<Sample> samplesList = samples.getSampleAsArrayList();
32                 double median = FeatureValueList.get(FeatureValueList.size()/2);
33                 for(Sample sample : samplesList)
34                 {
35                         double newValue = sample.getFeature(samples.getfeatureList().get(
                               featureIndex)).getValue();
36                         if(newValue < median)
37                                 newValue = 0.0;
38                         else
39                                 newValue = 1.0;
```

```
40
41                      sample.setFeature(new Feature(samples.getfeatureList().get(
                            featureIndex), newValue));
42              }
43
44              //Setting the number of discrete classes for easy access during decision tree
                    induction.
45              samples.setNumDiscreteClasses(featureIndex, 2);
46          }
47
48          /* ************************************ALGORITHM 2
                ***************************************/
49
50          private static double minValue = Double.MAX_VALUE;
51
52          /*
53           * This is a static method that discretizes the values of a certain feature of a collection
                of samples.
54           *  After discretization, the values can be any integer between 0 and binSize (inclusive)
55           *  featureIndex specifies the index of the feature in the featureList array of the samples
                collection
56           */
57          public static void discretizeEqualBinner(SampleCollection samples, int featureIndex, int
                binSize)
58          {
59              ArrayList<String> featureList = samples.getfeatureList();
60
61              double delta = computeBinWidth(samples, featureList.get(featureIndex),
                    binSize);
62              //System.out.println("Delta = " + delta);
63              samples.addBinningVar(featureIndex, delta, minValue);
64
```

```
65              discretizeSamples(samples, featureList.get(featureIndex), delta);
66              samples.setNumDiscreteClasses(featureIndex, binSize + 1);
67          }
68
69          /*
70           * Using the "Equal Width Interval Binning" algorithm for discretization.
71           *
72           * See the paper for more information - http://robotics.stanford.edu/users/sahami/
                 papers-dir/disc.pdf
73           */
74          private static void discretizeSamples(SampleCollection samples, String featureName,
                double delta)
75          {
76              ArrayList<Sample> samplesList = samples.getSampleAsArrayList();
77
78              for(Sample sample : samplesList)
79              {
80                  /*double newValue = (int)((sample.getFeature(featureName).getValue
                        () - minValue) / delta);
81                  sample.setFeature(new Feature(featureName, newValue));*/
82                  discretizeSample(sample, featureName, delta, minValue);
83              }
84          }
85
86          /*
87           * A public method to discretize the feature - featureName of sample based on delta
                 and min
88           */
89          public static void discretizeSample(Sample sample, String featureName, double delta,
                double min)
90          {
```

```
91              double newValue = (int)((sample.getFeature(featureName).getValue() - min)
                    / delta);
92
93              //Check for extraneous values
94              if(newValue < 0)
95                      newValue = 0;
96              if(newValue > NUMBER_OF_BINS)
97                      newValue = NUMBER_OF_BINS;
98
99              sample.setFeature(new Feature(featureName, newValue));
100         }
101
102         /*
103          * Computes delta = (xmax - xmin) / k
104          */
105         private static double computeBinWidth(SampleCollection samples, String featureName,
                int binSize)
106         {
107             ArrayList<Sample> samplesList = samples.getSampleAsArrayList();
108
109             minValue = Double.MAX_VALUE;
110             double max = Double.MIN_VALUE;
111
112             for(Sample sample : samplesList)
113             {
114                     Feature feature = sample.getFeature(featureName);
115                     if(feature.getValue() < minValue)
116                             minValue = feature.getValue();
117                     if(feature.getValue() > max)
118                             max = feature.getValue();
119             }
120
```

```
121                 double delta = (max − minValue) / binSize;
122                 return delta;
123         }
124 }
```

Listing 7.10: Genome.java

```
1  package org.ck.ga;
2  import java.util.ArrayList;
3  import java.util.Random;
4
5  import javax.swing.plaf.basic.BasicInternalFrameTitlePane.MaximizeAction;
6
7  import org.ck.dt.DecisionTreeClassifier;
8  import org.ck.gui.Constants;
9  import org.ck.gui.Constants.Filenames;
10 import org.ck.sample.DataHolder;
11 import org.ck.sample.SampleCollection;
12
13
14 public class Genome implements Constants
15 {
16         static private SampleCollection samples;  //Sample Collection
17         private static ArrayList<String> FeatureSuperSet; //The complete set of features from
               which smaller subsets are derived for the GA
18         private Random rgen = new Random();
19         private String chromosome;     //A bit string that shows which features are present or
               absent
20         private double fitnessScore;    //Here, the fitness function is a function of the
               classification accuracy of the decision tree
21
22         static
23         {
```

```
24              samples = new SampleCollection(DataHolder.getTrainingSamplesFileName(),
                    DataHolder.getAttributesFileName());
25              FeatureSuperSet = samples.getfeatureList();
26              samples.discretizeSamples(Constants.DiscretizerAlgorithms.EQUAL_BINNING);
27          }
28
29          /*
30           * Used to reinitialize the static variables of this class, when DataHolder is updated,
                since static variables
31           * aren't updated automatically.
32           */
33          public static void reInitializeStaticVariables()
34          {
35              samples = new SampleCollection(DataHolder.getTrainingSamplesFileName(),
                    DataHolder.getAttributesFileName());
36              FeatureSuperSet = samples.getfeatureList();
37              samples.discretizeSamples(Constants.DiscretizerAlgorithms.EQUAL_BINNING);
38          }
39
40          /*
41           * This constructor takes a bit string representing features as a parameter and
42           *  initializes a decision tree from it.
43           */
44          public Genome(String chromosome) throws OptimalScoreException
45          {
46              initDTfromChromosome(chromosome);
47          }
48
49
50          /*
51           * Initializes a decision tree that uses only the features present in the chromosome.
52           *   It also calculates the fitness score of this chromosome.
```

```
53          */
54          private void initDTfromChromosome(String chromosome) throws
                OptimalScoreException
55          {
56                  this.chromosome = chromosome;
57
58                  calculateFitnessScore(TRAINING_SET_WEIGHT, TEST_SET_WEIGHT);
59
60                  if(fitnessScore >= DataHolder.getFitnessScoreThreshold())
61                          throw new OptimalScoreException(this);
62          }
63
64          /*
65           * A redesigned Fitness Function calculator
66           * It takes into account the accuracy of the decision tree while classifying both, training
                and test examples
67           * The fitness score is a function of the weighted average of the two accuracies.
68           */
69          private void calculateFitnessScore(double trainingWeight, double testingWeight)
                throws OptimalScoreException
70          {
71                  DecisionTreeClassifier dtClassifier = getDecisionTree();
72                  dtClassifier.TestAndFindAccuracy();
73
74                  //Part 1 - Get training set accuracy
75                  double trainingSetAccuracy = dtClassifier.getAccuracy();
76
77                  //Part 2 - Get test set accuracy
78                  SampleCollection test_samples = new SampleCollection(DataHolder.
                        getTestingSamplesFileName(), DataHolder.getAttributesFileName());
79                  //test_samples.discretizeSamples(Constants.DiscretizerAlgorithms.
                        EQUAL_BINNING);
```

```
80                  test_samples.discretizeSamplesBasedOnOtherSampleCollection(dtClassifier.
                        getTrainingSamples());
81                  dtClassifier.setTestingSamples(test_samples);
82                  dtClassifier.TestAndFindAccuracy();
83                  double testSetAccuracy = dtClassifier.getAccuracy();
84
85                  fitnessScore = (trainingWeight * trainingSetAccuracy + testingWeight *
                        testSetAccuracy) / (trainingWeight + testingWeight);
86                  //fitnessScore = trainingSetAccuracy; It was running very slowly that's why all
                        this circus. We'll find a solution.
87                  //fitnessScore = trainingSetAccuracy > testSetAccuracy ? trainingSetAccuracy :
                         testSetAccuracy;
88          }
89
90          /*
91           * Returns the Fitness score of this genome
92           */
93          public double getFitnessScore()
94          {
95                  return fitnessScore;
96          }
97
98          /*
99           * Every bit of the chromosome string has a probability equal to mutationProbability of
                mutating.
100          *   After mutation, the decision tree of this genome is reinitialized
101          */
102         public void mutate(double mutationProbability) throws OptimalScoreException
103         {
104                 StringBuffer chromosomeBuffer = new StringBuffer(getChromosome());
105                 for(int i=0; i<chromosomeBuffer.length(); i++)
106                 {
```

```
107                     if(getProbabilisticOutcome(mutationProbability))
108                     {
109                             chromosomeBuffer.setCharAt(i, (chromosomeBuffer.charAt(i)
                                    == '0') ? '1':'0');
110                     }
111             }
112
113
114             initDTfromChromosome(chromosomeBuffer.toString());
115     }
116
117     /*
118      * Returns the chromosome
119      */
120     public String getChromosome()
121     {
122             return chromosome;
123     }
124
125     /*
126      * Displays the chromosome as well as the Fitness score
127      */
128     public void displayGenes()
129     {
130             System.out.println("Chromosome - "+chromosome+" FitnessValue - "+
                    fitnessScore);
131     }
132
133     /*
134      * Generates an outcome for a random event.
135      */
136     private boolean getProbabilisticOutcome(double probability)
```

```
137         {
138                 Random rgen = new Random();
139                 return (rgen.nextInt((int)Math.pow(10, 6)) + 1 <= probability * Math.pow(10,
                        6));
140         }
141
142         /*
143          * Returns the number of features in the Feature Super set
144          */
145         public static int getFeatureSuperSetSize()
146         {
147                 return FeatureSuperSet.size();
148         }
149
150         /*
151          * Returns a new decision tree that was created by using only the features present in the
                chromosome.
152          */
153         public DecisionTreeClassifier getDecisionTree()
154         {
155                 ArrayList<String> features = new ArrayList<String>();
156
157                 for(int i=0; i<chromosome.length(); ++i)
158                         if(chromosome.charAt(i)=='1')
159                                 features.add(FeatureSuperSet.get(i));
160                 //randomizeFeatures(features);
161                 DecisionTreeClassifier dtClassifier = new DecisionTreeClassifier(samples, features
                        );
162                 SampleCollection training_samples = new SampleCollection(samples.
                        getSamplesFilename(Filenames.TRAINING_SAMPLES_FILE), samples.
                        getSamplesFilename(Filenames.FEATURES_FILE));
```

```
163                 training_samples.discretizeSamples(Constants.DiscretizerAlgorithms.
                        EQUAL_BINNING);
164                 dtClassifier.setTestingSamples(training_samples);
165                 dtClassifier.setTrainingSamples(training_samples);
166
167                 return dtClassifier;
168         }
169         /*
170          * Radomize the features. (Just experimenting)
171          * By randomizing the feature, the order in which the DT
172          * is different and may result in better accuracy.
173          */
174         private void randomizeFeatures(ArrayList<String> features) {
175                 for(int i=0; i<features.size(); ++i)
176                 {
177                         int j = rgen.nextInt(features.size());
178                         String temp = features.get(i);
179                         features.set(i, features.get(j));
180                         features.set(j, temp);
181                 }
182
183         }
184
185         /*
186          * Returns the statically initialized Sample Collection
187          */
188         public static SampleCollection getSamples()
189         {
190                 return samples;
191         }
192 }
```

Listing 7.11: Population.java

```
1  package org.ck.ga;
2  import java.util.ArrayList;
3  import java.util.Random;
4
5  import org.ck.gui.Constants;
6  import org.ck.sample.DataHolder;
7
8
9  public class Population implements Constants
10 {
11         private ArrayList<Genome> genomes;
12         private Random rgen = new Random();
13
14         /*
15          * Initializes the genomes list with a random population
16          */
17         public Population() throws OptimalScoreException
18         {
19                 genomes = new ArrayList<Genome>();
20
21                 randomPopulationInit();
22         }
23
24         /*
25          * Darwin's Survival of the Fittest algorithm
26          */
27         public void runGeneticAlgorithm() throws OptimalScoreException
28         {
29                 for(int i=0; i<NUM_OF_GENERATIONS; ++i)
30                         {
```

```
31                              double totalFitnessScore = assessFitness(genomes);
32                              System.out.println("Total Fitness Score = " +
                                    totalFitnessScore);
33                              naturalSelection(totalFitnessScore);
34                              //displayBestGenome();
35                      }
36              displayBestGenome();
37      }
38
39      private void displayBestGenome() throws OptimalScoreException
40      {
41              double bestFitnessScore = 0;
42              Genome bestGenome = null;
43              for(int i=0; i<genomes.size(); i++)
44              {
45                      if(genomes.get(i).getFitnessScore() > bestFitnessScore)
46                      {
47                              bestFitnessScore = genomes.get(i).getFitnessScore();
48                              bestGenome = genomes.get(i);
49                      }
50              }
51
52              System.out.println("Best Genome: ");
53              //bestGenome.displayGenes();
54              //throw new OptimalScoreException(bestGenome);
55      }
56
57      /*
58       * Creates a new population from the old population by selecting two genomes randomly
             at a time, and
59       *  performing crossover and mutation operations.
60       */
```

```
61      private void naturalSelection(double totalFitnessScore) throws OptimalScoreException
62      {
63              ArrayList<Genome> newPopulation = new ArrayList<Genome>();
64
65              while(newPopulation.size() < genomes.size())
66              {
67                      Genome randGenome1 = rouletteSelection(totalFitnessScore);
68                      Genome randGenome2 = rouletteSelection(totalFitnessScore);
69
70                      //System.out.println("Selected 1 " + randGenome1.getFitnessScore());
71                      //System.out.println("Selected 2 " + randGenome2.getFitnessScore());
72
73                      crossoverGenomes(randGenome1, randGenome2, newPopulation);
74                      mutateGenomes(newPopulation);
75
76                      //displayPopulation();
77              }
78
79              genomes = newPopulation;
80      }
81
82      /*
83       * Performs genetic mutation on the two most recent offspring
84       */
85      private void mutateGenomes(ArrayList<Genome> newPopulation) throws
            OptimalScoreException
86      {
87              newPopulation.get(newPopulation.size() - 1).mutate(DataHolder.
                    getMutationProbabilityThreshold());
88              newPopulation.get(newPopulation.size() - 2).mutate(DataHolder.
                    getMutationProbabilityThreshold());
89      }
```

```
90
91          /*
92           * With a probability equal to CROSSOVER_PROBABILITY, two new children are
                 created by mixing the traits of
93           *   two genomes based on a crossover point. These new children are added to the new
                 population. The parents aren't.
94           * With a probability equal to (1 - CROSSOVER_PROBABILITY), the father and
                 mother are added to the new population.
95           */
96          private void crossoverGenomes(Genome father, Genome mother, ArrayList<Genome>
                newPopulation) throws OptimalScoreException
97          {
98                  if(getProbabilisticOutcome(DataHolder.getCrossoverProbabilityThreshold()))
99                  {
100                         int crossoverPoint = rgen.nextInt(father.getChromosome().length());
101
102                         Genome child1 = new Genome(father.getChromosome().substring(0,
                               crossoverPoint)
103                                                         + mother.getChromosome().
                                                               substring(crossoverPoint)
                                                               );
104                         Genome child2 = new Genome(mother.getChromosome().substring(0,
                               crossoverPoint)
105                                         + father.getChromosome().substring(crossoverPoint));
106
107                         /*System.out.println("Child 1");
108                         child1.displayGenes();
109                         System.out.println("Child 2");
110                         child2.displayGenes();*/
111
112                         //New Generation
113                         newPopulation.add(child1);
```

```
114                     newPopulation.add(child2);
115
116                     return;
117             }
118             else
119             {
120                     newPopulation.add(father);
121                     newPopulation.add(mother);
122             }
123     }
124
125     /*
126      * Converts a number i of any length to a bit string of length n
127      */
128     private String toNBitBinaryString(int i, int n)
129     {
130             String str = Integer.toBinaryString(i);
131
132             if(str.length() == n)
133                     return str;
134
135             String zeroes = new String(new char[n - str.length()]).replace("\0", "0");
136             return zeroes + str;
137
138     }
139
140     /*
141      * This kind of selection ensures that genomes having a higher fitness score than others
            have a better
142      *  chance of being seleted for reproduction.
143      */
144     private Genome rouletteSelection(double totalFitnessScore)
```

```
145         {
146             double ball = rgen.nextDouble() * totalFitnessScore;
147             double slice = 0.0;
148 
149             for(int i=0; i<genomes.size(); i++)
150             {
151                 slice += genomes.get(i).getFitnessScore();
152 
153                 if(ball < slice)
154                     return genomes.get(i);
155             }
156 
157             return genomes.get(0);
158         }
159 
160         /*
161          * Returns the sum of all fitness scores of all the genomes in the current population.
162          */
163         private double assessFitness(ArrayList<Genome> genomes)
164         {
165                 double totalFitnessScore = 0.0;
166                 for(int i=0; i<genomes.size(); i++)
167                 {
168                         double fitnessScore = genomes.get(i).getFitnessScore();
169                         totalFitnessScore += fitnessScore;
170                 }
171 
172                 return totalFitnessScore;
173         }
174 
175         /*
176          * Displays the population
```

```
177          */
178         public void displayPopulation()
179         {
180                 System.out.println("\nPopulation: ");
181                 for(int i=0; i<genomes.size(); i++)
182                         genomes.get(i).displayGenes();
183                 System.out.println();
184         }
185
186         /*
187          * Initializes a random population
188          */
189         private void randomPopulationInit() throws OptimalScoreException
190         {
191                 int numOfFeatures = Genome.getFeatureSuperSetSize();
192                 double upperLimit = Math.pow(2, numOfFeatures)-1;
193                 int[] FeatureSubsetValues = new int[(int)upperLimit];
194                 for(int i=0; i<upperLimit; ++i)
195                         FeatureSubsetValues[i] = i;
196                 //randomShuffle(FeatureSubsetValues);
197                 for(int i=0; i< POPULATION_SIZE; i++)
198                         genomes.add(new Genome(toNBitBinaryString(FeatureSubsetValues[i],
                            numOfFeatures)));
199         }
200
201         private void randomShuffle(int[] featureSubsetValues) {
202                 for(int i=0; i<featureSubsetValues.length; ++i)
203                 {
204                         int j = rgen.nextInt(featureSubsetValues.length);
205                         int temp = featureSubsetValues[0];
206                         featureSubsetValues[0] = featureSubsetValues[j];
207                         featureSubsetValues[j] = temp;
```

```
208                     }
209             }
210
211
212             private boolean getProbabilisticOutcome(double probability)
213             {
214                     return (rgen.nextInt((int)Math.pow(10, 6)) + 1 <= probability * Math.pow(10,
                            6));
215             }
216
217     }
```

Listing 7.12: OptimalScoreException.java

```
1   package org.ck.ga;
2
3   import java.util.ArrayList;
4
5   import javax.print.attribute.standard.Chromaticity;
6
7   import org.ck.dt.DecisionTreeClassifier;
8   import org.ck.gui.Constants;
9   import org.ck.sample.DataHolder;
10  import org.ck.sample.SampleCollection;
11
12  public class OptimalScoreException extends Exception implements Constants
13  {
14          private Genome genome_solution;
15
16          private double trainingSetAccuracy;
17          private double testSetAccuracy;
18
19          private ArrayList<Integer> trainingErrorIndices;
```

```
20          private ArrayList<Integer> testErrorIndices;
21
22          private DecisionTreeClassifier dtClassifier;
23
24          public OptimalScoreException()
25          {}
26
27          public OptimalScoreException(String msg)
28          {
29              super(msg);
30          }
31
32          /*
33           * Since this exception is thrown when the Genetic algorithm has found a genome that
                has a high fitness
34           * score, this constructor finds out the accuracy of the chosen genome's decision tree on
                the test set.
35           */
36          public OptimalScoreException(Genome genome) {
37                  this.genome_solution = genome;
38                  genome_solution.displayGenes();
39
40                  System.out.println("\n\nEXCEPTION CAUGHT - SOLUTION FOUND");
41                  //System.out.println(genome.samples.getSamplesFilename(Filenames.
                        TRAINING_SAMPLES_FILE));
42
43                  DecisionTreeClassifier dtClassifier = genome_solution.getDecisionTree();
44                  trainingErrorIndices = dtClassifier.TestAndFindAccuracy();
45                  System.out.println("Training Set Accuracy = " + (trainingSetAccuracy =
                        dtClassifier.getAccuracy()));
46
```

```
47              SampleCollection test_samples = new SampleCollection(DataHolder.
                    getTestingSamplesFileName(), DataHolder.getAttributesFileName());
48              //test_samples.discretizeSamples(Constants.DiscretizerAlgorithms.
                    EQUAL_BINNING);
49              test_samples.discretizeSamplesBasedOnOtherSampleCollection(dtClassifier.
                    getTrainingSamples());
50              dtClassifier.setTestingSamples(test_samples);
51              testErrorIndices = dtClassifier.TestAndFindAccuracy();
52
53              System.out.println("Test set accuracy = " + (testSetAccuracy = dtClassifier.
                    getAccuracy()));
54
55              //test_samples.displayBinning();
56              this.dtClassifier = dtClassifier;
57      }
58
59      public double getTrainingSetAccuracy()
60      {
61              return trainingSetAccuracy;
62      }
63
64      public double getTestSetAccuracy()
65      {
66              return testSetAccuracy;
67      }
68
69      public ArrayList<Integer> getTrainingErrorIndices()
70      {
71              return trainingErrorIndices;
72      }
73
74      public ArrayList<Integer> getTestErrorIndices()
```

```
75          {
76                  return testErrorIndices;
77          }
78
79          public ArrayList<String> getSelectedFeatures()
80          {
81                  ArrayList<String> selectedFeatures = new ArrayList<String>();
82
83                  String chromosome = genome_solution.getChromosome();
84                  ArrayList<String> featureList = genome_solution.getSamples().getfeatureList();
85                  System.out.println(chromosome);
86                  for(int i=0; i<chromosome.length(); i++)
87                          if(chromosome.charAt(i) == '1')
88                                  selectedFeatures.add(featureList.get(i));
89
90                  System.out.println(selectedFeatures);
91                  return selectedFeatures;
92          }
93
94          public DecisionTreeClassifier getCurrentDTClassifier()
95          {
96                  return dtClassifier;
97          }
98  }
```

Listing 7.13: WelcomeWindow.java

```
1  package org.ck.gui;
2
3  import java.awt.Dialog;
4
5  import org.eclipse.swt.SWT;
6  import org.eclipse.swt.events.PaintEvent;
```

```
7  import org.eclipse.swt.events.PaintListener;
8  import org.eclipse.swt.graphics.Font;
9  import org.eclipse.swt.graphics.Image;
10 import org.eclipse.swt.layout.FormAttachment;
11 import org.eclipse.swt.layout.FormData;
12 import org.eclipse.swt.layout.FormLayout;
13 import org.eclipse.swt.widgets.Canvas;
14 import org.eclipse.swt.widgets.Display;
15 import org.eclipse.swt.widgets.Event;
16 import org.eclipse.swt.widgets.Label;
17 import org.eclipse.swt.widgets.Listener;
18 import org.eclipse.swt.widgets.Shell;
19
20 public class WelcomeWindow {
21         private Shell shell;
22         // private Canvas DTCanvas;
23         // private Canvas GACanvas;
24         // private Canvas GWCanvas;
25
26         private Canvas iconTrain;
27         private Canvas iconClassify;
28         private Canvas iconGit;
29         private Canvas iconSettings;
30         private Canvas iconExit;
31
32         public WelcomeWindow(Display display)
33         {
34                 shell = new Shell(display);
35                 initUI();
36                 initListeners();
37                 shell.setText("Decision Tree Based Classifier");
38                 shell.setImage(new Image(display, "Icons/statistics.png"));
```

```
39              shell.setSize(720,720);
40              shell.setLocation(50, 50);
41              shell.setBackgroundImage(new Image(display, "Icons/white_background.png"));
42              shell.open();
43              while(!shell.isDisposed())
44              {
45                      if(!display.readAndDispatch())
46                              display.sleep();
47              }
48      }
49
50      private void initListeners() {
51              iconGit.addListener(SWT.MouseDown, new Listener() {
52
53                      @Override
54                      public void handleEvent(Event event) {
55                              System.out.println("https://www.github.com/samiriff/
                                    GWClassifier");
56                              new BrowserWindow(shell.getDisplay());
57                      }
58              });
59
60              iconExit.addListener(SWT.MouseDown, new Listener() {
61
62                      @Override
63                      public void handleEvent(Event event) {
64                              System.out.println("Exit!");
65                              System.exit(0);
66                      }
67              });
68              iconTrain.addListener(SWT.MouseDown, new Listener() {
69
```

```
70                      @Override
71                      public void handleEvent(Event event) {
72                              new MainWindow(shell.getDisplay());
73
74
75                      }
76              });
77              iconClassify.addListener(SWT.MouseDown, new Listener() {
78
79                      @Override
80                      public void handleEvent(Event event) {
81                              new ClassifyWindow(shell.getDisplay());
82
83                      }
84              });
85
86
87      }
88
89      private void initUI() {
90              shell.setLayout(new FormLayout());
91              Label welcomeLabel = new Label(shell,SWT.LEFT);
92              welcomeLabel.setFont(new Font(shell.getDisplay(),"Jokerman",20,SWT.ITALIC)
                    );
93              FormData formData = new FormData(20,20);
94              formData.left = new FormAttachment(20);
95              formData.right = new FormAttachment(90);
96              formData.top = new FormAttachment(5);
97              formData.bottom = new FormAttachment(15);
98              welcomeLabel.setText("Decision Tree Based Classifier");
99              welcomeLabel.setLayoutData(formData);
100
```

```
101
102
103             iconTrain = new Canvas(shell, SWT.BORDER);
104             formData = new FormData();
105             formData.left = new FormAttachment(welcomeLabel, 10, SWT.LEFT);
106             formData.right = new FormAttachment(iconTrain, 80, SWT.LEFT);
107             formData.top = new FormAttachment(welcomeLabel, 10, SWT.BOTTOM);
108             formData.bottom = new FormAttachment(iconTrain, 80, SWT.TOP);
109             iconTrain.setLayoutData(formData);
110             iconTrain.addPaintListener(new PaintListener() {
111                     public void paintControl(final PaintEvent event) {
112                             Image imageSrc = new Image(shell.getDisplay(), "Icons/
                                    update.png");
113                             if (imageSrc != null) {
114                                     event.gc.drawImage(imageSrc, 0, 0);
115                             }
116                     }
117             });
118
119             iconClassify = new Canvas(shell, SWT.BORDER);
120             formData = new FormData();
121             formData.left = new FormAttachment(iconTrain, 0, SWT.LEFT);
122             formData.right = new FormAttachment(iconClassify, 80, SWT.LEFT);
123             formData.top = new FormAttachment(iconTrain, 10, SWT.BOTTOM);
124             formData.bottom = new FormAttachment(iconClassify, 80, SWT.TOP);
125             iconClassify.setLayoutData(formData);
126             iconClassify.addPaintListener(new PaintListener() {
127                     public void paintControl(final PaintEvent event) {
128                             Image imageSrc = new Image(shell.getDisplay(), "Icons/new.
                                    png");
129                             if (imageSrc != null) {
130                                     event.gc.drawImage(imageSrc, 0, 0);
```

```
131                         }
132                     }
133                 });
134
135                 iconGit = new Canvas(shell, SWT.BORDER);
136                 formData = new FormData();
137                 formData.left = new FormAttachment(iconClassify, 0, SWT.LEFT);
138                 formData.right = new FormAttachment(iconGit, 80, SWT.LEFT);
139                 formData.top = new FormAttachment(iconClassify, 10, SWT.BOTTOM);
140                 formData.bottom = new FormAttachment(iconGit, 80, SWT.TOP);
141                 iconGit.setLayoutData(formData);
142                 iconGit.addPaintListener(new PaintListener() {
143                     public void paintControl(final PaintEvent event) {
144                         Image imageSrc = new Image(shell.getDisplay(), "Icons/github
                                .jpg");
145                         if (imageSrc != null) {
146                             event.gc.drawImage(imageSrc, 0, 0);
147                         }
148                     }
149                 });
150
151                 iconSettings = new Canvas(shell, SWT.BORDER);
152                 formData = new FormData();
153                 formData.left = new FormAttachment(iconGit, 0, SWT.LEFT);
154                 formData.right = new FormAttachment(iconSettings, 80, SWT.LEFT);
155                 formData.top = new FormAttachment(iconGit, 10, SWT.BOTTOM);
156                 formData.bottom = new FormAttachment(iconSettings, 80, SWT.TOP);
157                 iconSettings.setLayoutData(formData);
158                 iconSettings.addPaintListener(new PaintListener() {
159                     public void paintControl(final PaintEvent event) {
160                         Image imageSrc = new Image(shell.getDisplay(), "Icons/users.
                                png");
```

```
161                         if (imageSrc != null) {
162                                 event.gc.drawImage(imageSrc, 0, 0);
163                         }
164                 }
165         });
166         iconSettings.setVisible(false);
167
168         iconExit = new Canvas(shell, SWT.BORDER);
169         formData = new FormData();
170         formData.left = new FormAttachment(iconSettings, 0, SWT.LEFT);
171         formData.right = new FormAttachment(iconExit, 80, SWT.LEFT);
172         formData.top = new FormAttachment(iconSettings, 10, SWT.BOTTOM);
173         formData.bottom = new FormAttachment(iconExit, 80, SWT.TOP);
174         iconExit.setLayoutData(formData);
175         iconExit.addPaintListener(new PaintListener() {
176                 public void paintControl(final PaintEvent event) {
177                         Image imageSrc = new Image(shell.getDisplay(), "Icons/delete.
                            png");
178                         if (imageSrc != null) {
179                                 event.gc.drawImage(imageSrc, 0, 0);
180                         }
181                 }
182         });
183
184         Label exitLabel = new Label(shell,SWT.WRAP);
185         formData = new FormData();
186         formData.top = new FormAttachment(iconExit, 5 ,SWT.CENTER);
187         formData.left = new FormAttachment(iconExit, 10, SWT.RIGHT);
188         exitLabel.setLayoutData(formData);
189         Font f = new Font(shell.getDisplay(), "Lucida Sans", 16, SWT.BOLD);
190         exitLabel.setFont(f);
191         exitLabel.setText("Exit Application");
```

```
192
193
194             Label settingsLabel = new Label(shell,SWT.WRAP);
195             formData = new FormData();
196             formData.top = new FormAttachment(iconSettings, 5 ,SWT.CENTER);
197             formData.left = new FormAttachment(iconSettings, 10, SWT.RIGHT);
198             settingsLabel.setLayoutData(formData);
199             settingsLabel.setFont(f);
200             settingsLabel.setText("Dev Info"); settingsLabel.setVisible(false);
201
202             Label viewLabel = new Label(shell,SWT.WRAP);
203             formData = new FormData();
204             formData.top = new FormAttachment(iconGit, 5 ,SWT.CENTER);
205             formData.left = new FormAttachment(iconGit, 10, SWT.RIGHT);
206             viewLabel.setLayoutData(formData);
207             viewLabel.setFont(f);
208             viewLabel.setText("View On Github");
209
210
211             Label classifyLabel = new Label(shell,SWT.WRAP);
212             formData = new FormData();
213             formData.top = new FormAttachment(iconClassify, 5 ,SWT.CENTER);
214             formData.left = new FormAttachment(iconClassify, 10, SWT.RIGHT);
215             classifyLabel.setLayoutData(formData);
216             classifyLabel.setFont(f);
217             classifyLabel.setText("Classify The Data sets");
218
219             Label trainLabel = new Label(shell,SWT.WRAP);
220             trainLabel.setFont(f);
221             formData = new FormData();
222             formData.top = new FormAttachment(iconTrain, 5 ,SWT.CENTER);
223             formData.left = new FormAttachment(iconTrain, 10, SWT.RIGHT);
```

```
224                 trainLabel.setLayoutData(formData);
225                 trainLabel.setText("Train the Decision Tree");
226
227         }
228 }
```

Listing 7.14: MainWindow.java

```
1  package org.ck.gui;
2
3  import java.io.BufferedWriter;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.io.OutputStreamWriter;
7  import java.util.ArrayList;
8
9  import org.ck.dt.DecisionTreeClassifier;
10 import org.ck.ga.Genome;
11 import org.ck.ga.OptimalScoreException;
12 import org.ck.gui.Constants.DatasetOptions;
13 import org.ck.sample.DataHolder;
14 import org.ck.sample.Sample;
15 import org.ck.sample.SampleCollection;
16 import org.eclipse.swt.SWT;
17 import org.eclipse.swt.custom.StyledText;
18 import org.eclipse.swt.custom.TableEditor;
19 import org.eclipse.swt.events.SelectionAdapter;
20 import org.eclipse.swt.events.SelectionEvent;
21 import org.eclipse.swt.graphics.Color;
22 import org.eclipse.swt.graphics.Point;
23 import org.eclipse.swt.graphics.Rectangle;
24 import org.eclipse.swt.layout.GridData;
25 import org.eclipse.swt.layout.GridLayout;
```

```
26 import org.eclipse.swt.widgets.Button;
27 import org.eclipse.swt.widgets.Combo;
28 import org.eclipse.swt.widgets.Control;
29 import org.eclipse.swt.widgets.Display;
30 import org.eclipse.swt.widgets.Event;
31 import org.eclipse.swt.widgets.Label;
32 import org.eclipse.swt.widgets.List;
33 import org.eclipse.swt.widgets.Listener;
34 import org.eclipse.swt.widgets.ProgressBar;
35 import org.eclipse.swt.widgets.Shell;
36 import org.eclipse.swt.widgets.Slider;
37 import org.eclipse.swt.widgets.Table;
38 import org.eclipse.swt.widgets.TableColumn;
39 import org.eclipse.swt.widgets.TableItem;
40 import org.eclipse.swt.widgets.Text;
41 import org.eclipse.swt.widgets.Tree;
42
43 public class MainWindow implements Constants
44 {
45         private Shell shell;
46         private Display display;
47
48         private int gridHorizontalSpacing = 10;
49         private int gridVerticalSpacing = 4;
50         private int gridMarginBottom = 5;
51         private int gridMarginTop = 5;
52         private int gridPadding = 2;
53
54         private Combo comboDatasetBox;
55
56         private List algorithmList;
57
```

```
58          private Table featureSelectorTable;
59          private Button featureSelectorButtons[];
60
61          private Slider fitnessSlider;
62          private Label fitnessSliderLabel;
63          private Button runButton;
64
65          private Slider crossoverSlider;
66          private Label crossoverSliderLabel;
67
68          private Slider mutationSlider;
69          private Label mutationSliderLabel;
70
71          private Table trainingSamplesTable = null;
72          private Button discretizeCheckBox;
73          private Table testingSamplesTable = null;
74
75          private StyledText accuracyTextArea;
76
77          private Table userSamplesTable = null;
78          private Button classifyButton;
79          private Label classifyResultLabel;
80
81          private Button saveDTButton;
82
83          private Tree graphicalDecisionTree = null;
84
85          private OptimalScoreException currentException = null;
86
87          /*
88           * A constructor that takes in a display parameter and initializes the shell and other
                components of the UI
```

```
89          */
90          public MainWindow(Display display)
91          {
92                  this.display = display;
93
94                  shell = new Shell(display);
95                  shell.setText("Decision Tree Classifier");
96
97                  centerShell();
98                  initUI();
99
100                 shell.setSize(1024, 1000);
101                 shell.setLocation(480, 0);
102
103                 shell.open();
104                 while(!shell.isDisposed())
105                 {
106                         if(!display.readAndDispatch())
107                                 display.sleep();
108                 }
109         }
110
111         /*
112          * The Main method
113          */
114         public static void main(String args[])
115         {
116                 Display display = new Display();
117                 new WelcomeWindow(display);
118                 //new MainWindow(display);
119                 display.dispose();
120         }
```

```
121
122         /*
123          * Centers the shell on the screen.... Doesn't work
124          */
125         private void centerShell()
126         {
127                 Rectangle bds = shell.getDisplay().getBounds();
128
129                 Point p = shell.getSize();
130
131                 int nLeft = (bds.width - p.x) / 2;
132                 int nTop = (bds.height - p.y) / 2;
133
134                 shell.setBounds(nLeft, nTop, p.x, p.y);
135         }
136
137
138         /*
139          * Initializes the UI
140          */
141         private void initUI()
142         {
143                 //Initialize Grid Layout parameters
144                 GridLayout gridLayout = new GridLayout(gridHorizontalSpacing, true);
145                 gridLayout.horizontalSpacing = gridHorizontalSpacing;
146                 gridLayout.verticalSpacing = gridVerticalSpacing;
147                 gridLayout.marginBottom = gridMarginBottom;
148                 gridLayout.marginTop = gridMarginTop;
149                 shell.setLayout(gridLayout);
150
151                 //Adding Widgets
```

```
152             addLabel("Decision Tree Based Classifier", gridHorizontalSpacing, SWT.
                    CENTER);
153             addDataSamplesComboBox();
154             //addBreak(gridHorizontalSpacing / 2);
155
156             addListBox();
157             addBreak(gridHorizontalSpacing);
158
159             addFeatureSelectorTable();
160
161             fitnessSlider = addFitnessThresholdSlider();
162             fitnessSlider.setVisible(false);
163
164             crossoverSlider = addCrossoverRateSlider();
165             crossoverSlider.setVisible(false);
166
167             mutationSlider = addMutationRateSlider();
168             mutationSlider.setVisible(false);
169
170             addBreak(gridHorizontalSpacing / 4 + 1);
171             runButton = addRunButton();
172             runButton.setVisible(false);
173
174             addBreak(gridHorizontalSpacing);
175             addTrainingSamplesTable(false, false);
176             addTestingSamplesTable(false, false);
177             addDiscretizeCheckbox();
178
179             addResultDisplay();
180
181             addEditableSamplesTable();
182
```

```
183                 saveDTButton = addSaveDTButton();
184                 addBreak(2);
185                 classifyButton = addClassifyButton();
186                 classifyButton.setVisible(false);
187
188                 classifyResultLabel = addLabel("Result: ", gridHorizontalSpacing / 4, SWT.
                        RIGHT);
189                 classifyResultLabel.setVisible(false);
190
191                 addBreak(gridHorizontalSpacing / 4);
192                 addGraphicalDecisionTree();
193                 graphicalDecisionTree.setVisible(false);
194         }
195
196         /*
197          * Adds a run button, to start Machine Learning
198          */
199         private Button addRunButton()
200         {
201                 Button button = new Button(shell, SWT.PUSH | SWT.CENTER);
202                 button.setText("Run the Engine");
203
204                 addToGrid(button, gridHorizontalSpacing / 2);
205
206                 button.addSelectionListener(new SelectionAdapter()
207                 {
208                         @Override
209                         public void widgetSelected(SelectionEvent e)
210                         {
211                                 //Handle the selection event
212                                 try
213                                 {
```

```
214                         switch(algorithmList.getSelectionIndex())
215                         {
216                         case 0:
217                                 MainClass.sampleCaller2();
218                                 break;
219                         case 1:
220                                 //String allFeaturesChromosome =
                                        "0001111000101001";
221                                 String allFeaturesChromosome =
                                        constructChromosomeFromFeatureSelectorButtons
                                        ();
222                                 Genome allFeaturesGenome = new Genome(
                                        allFeaturesChromosome);
223                                 System.out.println(allFeaturesChromosome);
224                                 throw new OptimalScoreException(
                                        allFeaturesGenome);
225                         }
226                 }
227                 catch(OptimalScoreException exception)
228                 {
229                         displayResult(exception);
230                 }
231         }
232
233         private String constructChromosomeFromFeatureSelectorButtons()
234         {
235                 String chromosome = "";
236                 for(int i=0; i<featureSelectorButtons.length; i++)
237                 {
238                         if(featureSelectorButtons[i].getSelection())
239                                 chromosome += '1';
240                         else
```

```
241                                     chromosome += '0';
242                             }
243                             return chromosome;
244                     }
245
246                     private void displayResult(OptimalScoreException exception)
247                     {
248                             currentException = exception;
249
250                             accuracyTextArea.setVisible(true);
251
252                             String result = "Training Set Accuracy = " + exception.
                                    getTrainingSetAccuracy()*100 + "%\n";
253                             result += "Test Set Accuracy = " + exception.
                                    getTestSetAccuracy()*100 + "%\n";
254                             result += "Selected Features = " + exception.
                                    getSelectedFeatures();
255                             accuracyTextArea.setText(result);
256
257                             //Clear previous selection, if any
258                             TableItem items[] = trainingSamplesTable.getItems();
259                             for(int i=0; i < items.length; i++)
260                             {
261                                     items[i].setBackground(display.getSystemColor(SWT.
                                            COLOR_WHITE));
262                                     items[i].setForeground(display.getSystemColor(SWT.
                                            COLOR_BLACK));
263                             }
264
265                             //highlightIncorrectlyClassifiedSamples();
266                             addTrainingSamplesTable(false, true);
267                             addTestingSamplesTable(false, true);
```

```
268
269
270                             toggleIllegalWidgetsForStep2(true);
271                             addGraphicalDecisionTree();
272                     }
273
274             });
275
276             return button;
277     }
278
279     /*
280      * Creates a button, which, when clicked, will save the decision tree that has just been
            generated
281      */
282     private Button addSaveDTButton()
283     {
284             Button button = new Button(shell, SWT.PUSH);
285             button.setText("Save To File");
286             button.setVisible(false);
287
288             addToGrid(button, gridHorizontalSpacing / 3);
289
290
291             button.addSelectionListener(new SelectionAdapter() {
292                     @Override
293                     public void widgetSelected(SelectionEvent e)
294                     {
295                             System.out.println("Save to File Button Clicked");
296                             //System.out.println(currentException.getChromosomes()+"
                                    "+currentException.getTestSetAccuracy());
297                             BufferedWriter bw;
```

```
298                             //OutputStreamWriter osw = new OutputStreamWriter(new
                                    File)
299
300                             try {
301                                     bw = new BufferedWriter(new FileWriter(DataHolder.
                                            getSaveDatoToFileName(),true));
302                                     ArrayList<String> SelectedFeatures =
                                            currentException.getSelectedFeatures();
303                                     String data = "";
304                                     for (int i=0; i<SelectedFeatures.size(); ++i)
305                                     {
306                                             data = data + SelectedFeatures.get(i)+",";
307                                     }
308                                     data = data.substring(0, data.length()-1);
309                                     System.out.println("Data "+data);
310                                     bw.append(data+"->"+currentException.
                                            getTrainingSetAccuracy()*100+"%\n");
311                                     bw.close();
312                             } catch (IOException e1) {
313                                     // TODO Auto-generated catch block
314                                     e1.printStackTrace();
315                             }
316                             catch (StringIndexOutOfBoundsException sioe)
317                             {
318                                     sioe.printStackTrace();
319                             }
320
321                     }
322             });
323
324             return button;
325     }
```

```
326
327
328         /*
329          * This method can be used to insert 'num' blank labels to create spaces in the Grid
                 Layout
330          */
331         private void addBreak(int num)
332         {
333                 final Label label = new Label(shell, SWT.LEFT);
334                 label.setText("");
335
336                 GridData gridData = new GridData();
337                 gridData.horizontalSpan = num;
338                 gridData.horizontalAlignment = GridData.FILL;
339                 label.setLayoutData(gridData);
340         }
341
342         /*
343          * Adds a combo box to select the name of the data samples file
344          */
345         private void addDataSamplesComboBox()
346         {
347                 addLabel("Select Data Samples File: ", gridHorizontalSpacing / 2, SWT.RIGHT
                        );
348
349                 comboDatasetBox = new Combo(shell, SWT.DROP_DOWN);
350                 for(int i = 0; i < DatasetOptions.values().length; i++)
351                         comboDatasetBox.add("" + DatasetOptions.values()[i]);
352
353                 comboDatasetBox.select(0);
354                 comboDatasetBox.addSelectionListener(new SelectionAdapter() {
355
```

```
356                     @Override
357                     public void widgetSelected(SelectionEvent e) {
358                             DataHolder.setDataset(DatasetOptions.valueOf(
                                    comboDatasetBox.getText()));
359                             Genome.reInitializeStaticVariables();
360                             addTrainingSamplesTable(discretizeCheckBox.getSelection(),
                                    false);
361                             addTestingSamplesTable(discretizeCheckBox.getSelection(),
                                    false);
362                             addEditableSamplesTable();
363                             addFeatureSelectorTable();
364
365                             if(algorithmList.getSelectionIndex() == 1)
366                                     featureSelectorTable.setVisible(true);
367
368                             accuracyTextArea.setText("");
369                             toggleIllegalWidgetsForStep2(false);
370                     };
371             });
372
373             addToGrid(comboDatasetBox, gridHorizontalSpacing / 2 - gridPadding);
374             addBreak(gridPadding);
375     }
376
377     /*
378      * Adds a slider to vary the fitness threshold for the fitness function of the genetic
            algorithm
379      *  Returns the initialized slider
380      */
381     private Slider addFitnessThresholdSlider()
382     {
383             final double sliderRange = 1000;
```

```
384
385             fitnessSliderLabel = addLabel("Fitness Threshold --->  ",
                    gridHorizontalSpacing / 2, SWT.RIGHT);
386             fitnessSliderLabel.setVisible(false);
387
388             final Slider slider = new Slider(shell, SWT.HORIZONTAL);
389             slider.setMaximum((int)sliderRange);
390             slider.setSelection((int) (DataHolder.getFitnessScoreThreshold() * sliderRange));
391             addToGrid(slider, gridHorizontalSpacing / 2 - gridPadding);
392             addBreak(gridPadding);
393
394             fitnessSliderLabel.setText(fitnessSliderLabel.getText() + slider.getSelection() /
                    sliderRange);
395
396             slider.addListener (SWT.Selection, new Listener () {
397                 public void handleEvent (Event e) {
398                     double value = slider.getSelection() / sliderRange;
399                     fitnessSliderLabel.setText("Fitness Threshold --->  " + value
                            );
400                     DataHolder.setFitnessScoreThreshold(value);
401
402                     accuracyTextArea.setText("");
403                     toggleIllegalWidgetsForStep2(false);
404                 }
405             });
406
407             slider.setVisible(false);
408
409             return slider;
410         }
411
412         /*
```

```
413         * Adds a slider to vary the Crossover Rate for the natural selection process of the
                genetic algorithm
414         * Returns the initialized slider
415         */
416        private Slider addCrossoverRateSlider()
417        {
418                final double sliderRange = 10000;
419
420                crossoverSliderLabel = addLabel("Crossover Rate ---> ",
                        gridHorizontalSpacing / 2, SWT.RIGHT);
421                crossoverSliderLabel.setVisible(false);
422
423                final Slider slider = new Slider(shell, SWT.HORIZONTAL);
424                slider.setMaximum((int)sliderRange);
425                slider.setSelection((int) (DataHolder.getCrossoverProbabilityThreshold() *
                        sliderRange));
426                addToGrid(slider, gridHorizontalSpacing / 2 - gridPadding);
427                addBreak(gridPadding);
428
429                crossoverSliderLabel.setText(crossoverSliderLabel.getText() + slider.getSelection
                        () / sliderRange);
430
431                slider.addListener (SWT.Selection, new Listener () {
432                        public void handleEvent (Event e) {
433                                double value = slider.getSelection() / sliderRange;
434                                crossoverSliderLabel.setText("Crossove Rate ---> " + value)
                                        ;
435                                DataHolder.setCrossoverProbabilityThreshold(value);
436
437                                accuracyTextArea.setText("");
438                                toggleIllegalWidgetsForStep2(false);
439                        }
```

```
440                 });
441
442                 slider.setVisible(false);
443
444                 return slider;
445         }
446
447         /*
448          * Adds a slider to vary the Mutation Rate for Genome Mutation of the genetic
                algorithm
449          *  Returns the initialized slider
450          */
451         private Slider addMutationRateSlider()
452         {
453                 final double sliderRange = 10000;
454
455                 mutationSliderLabel = addLabel("Mutation Rate --->",
                        gridHorizontalSpacing / 2, SWT.RIGHT);
456                 mutationSliderLabel.setVisible(false);
457
458                 final Slider slider = new Slider(shell, SWT.HORIZONTAL);
459                 slider.setMaximum((int)sliderRange);
460                 slider.setSelection((int) (DataHolder.getMutationProbabilityThreshold() *
                        sliderRange));
461                 addToGrid(slider, gridHorizontalSpacing / 2 - gridPadding);
462                 addBreak(gridPadding);
463
464                 mutationSliderLabel.setText(mutationSliderLabel.getText() + slider.getSelection
                        () / sliderRange);
465
466                 slider.addListener (SWT.Selection, new Listener () {
467                         public void handleEvent (Event e) {
```

```
468                             double value = slider.getSelection() / sliderRange;
469                             mutationSliderLabel.setText("Mutation Rate --->" + value
                                    );
470                             DataHolder.setMutationProbabilityThreshold(value);
471
472                             accuracyTextArea.setText("");
473                             toggleIllegalWidgetsForStep2(false);
474                     }
475             });
476
477             slider.setVisible(false);
478
479             return slider;
480     }
481
482     /*
483      * Adds a list box to select an appropriate algorithm for Machine Learning.
484      *   The UI changes based on the algorithm selected
485      */
486     private void addListBox()
487     {
488             addLabel("Select Algorithm", gridHorizontalSpacing / 2, SWT.RIGHT);
489
490             algorithmList = new List(shell, SWT.BORDER);
491             algorithmList.add("GA-based Feature Selection");
492             algorithmList.add("Manual Feature Selection");
493
494             algorithmList.addListener(SWT.Selection, new Listener () {
495                     public void handleEvent (Event e) {
496
497                             if(algorithmList.getSelectionIndex() == 0)
498                             {
```

```
499                                     toggleIllegalWidgetsForStep1(true);
500
501                                     accuracyTextArea.setText("");
502                                     toggleIllegalWidgetsForStep2(false);
503
504                                     featureSelectorTable.setVisible(false);
505                             }
506                             else
507                             {
508                                     toggleIllegalWidgetsForStep1(false);
509
510                                     accuracyTextArea.setText("");
511                                     toggleIllegalWidgetsForStep2(false);
512
513                                     addFeatureSelectorTable();
514                                     featureSelectorTable.setVisible(true);
515                             }
516
517                             runButton.setVisible(true);
518                     }
519             });
520
521             addToGrid(algorithmList, gridHorizontalSpacing / 2 - gridPadding);
522             addBreak(gridPadding);
523     }
524
525     private void addDiscretizeCheckbox()
526     {
527             discretizeCheckBox = new Button(shell, SWT.CHECK);
528             discretizeCheckBox.setText("Show Discretized Values");
529             discretizeCheckBox.setSelection(false);
530             discretizeCheckBox.setVisible(false);
```

```
531
532                 addBreak(5);
533                 addToGrid(discretizeCheckBox, 2);
534
535                 discretizeCheckBox.addSelectionListener(new SelectionAdapter()
536                 {
537                         @Override
538                         public void widgetSelected(SelectionEvent e) {
539                                 if (discretizeCheckBox.getSelection()) {
540                                         addTrainingSamplesTable(true, true);
541                                         addTestingSamplesTable(true, true);
542                                 } else {
543                                         addTrainingSamplesTable(false, true);
544                                         addTestingSamplesTable(false, true);
545                                 }
546                         }
547                 });
548         }
549
550         /*
551          * Initializes the trainingSamplesTable and selects the appropriate SampleCollection,
                based on the value of
552          *  isDiscretize
553          */
554         private void addTrainingSamplesTable(boolean isDiscretize, boolean
              highlightIncorrectItems)
555         {
556                 Table previousTable = trainingSamplesTable;     //Required for replacement in
                      Grid Layout
557                 trainingSamplesTable = new Table (shell, SWT.MULTI | SWT.BORDER | SWT
                      .FULL_SELECTION);
558
```

```
559             SampleCollection samplesCollection = null;
560             if(isDiscretize)
561                     samplesCollection = Genome.getSamples();
562             else
563                     samplesCollection = new SampleCollection(DataHolder.
                            getTrainingSamplesFileName(), DataHolder.getAttributesFileName
                            ());
564
565             addTable(trainingSamplesTable, samplesCollection, previousTable);
566
567             if(highlightIncorrectItems)
568                     highlightIncorrectlyClassifiedSamples();
569     }
570
571     /*
572      * Initializes the testingSamplesTable and selects the appropriate SampleCollection, based
                on the value of
573      *   isDiscretize
574      */
575     private void addTestingSamplesTable(boolean isDiscretize, boolean
            highlightIncorrectItems)
576     {
577             Table previousTable = testingSamplesTable;     //Required for replacement in
                    Grid Layout
578             testingSamplesTable = new Table (shell, SWT.MULTI | SWT.BORDER | SWT.
                    FULL_SELECTION);
579
580             SampleCollection samplesCollection = null;
581             if(isDiscretize)
582                     samplesCollection = currentException.getCurrentDTClassifier().
                            getTestingSamples();
583             else
```

```
584                         samplesCollection = new SampleCollection(DataHolder.
                                getTestingSamplesFileName(), DataHolder.getAttributesFileName()
                                );
585
586                 addTable(testingSamplesTable, samplesCollection, previousTable);
587
588                 if(highlightIncorrectItems)
589                         highlightIncorrectlyClassifiedSamples();
590         }
591
592         /*
593          * This method creates an Excel-type table to display all the samples of the selected
                SampleCollection
594          *  in a samplesTable, replacing any previousSamplesTable that was drawn previously, if
                any.
595          */
596         private void addTable(Table samplesTable, SampleCollection samplesCollection, Table
              previousSamplesTable)
597         {
598                 samplesTable.setLinesVisible (true);
599                 samplesTable.setHeaderVisible (true);
600
601                 GridData data = new GridData(); //SWT.FILL, SWT.FILL, false, false);
602                 data.heightHint = 200;
603                 data.widthHint = 200;
604                 data.horizontalSpan = gridHorizontalSpacing / 2;
605                 data.horizontalAlignment = GridData.FILL;
606                 samplesTable.setLayoutData(data);
607
608                 //This block is executed if a previous table has to be overwritten with a new
                        table
609                 if(previousSamplesTable != null)
```

```
610             {
611                     samplesTable.moveAbove(previousSamplesTable);
612                     previousSamplesTable.dispose();
613                     samplesTable.getParent().layout();
614             }
615
616             ArrayList<String> featureList = samplesCollection.getfeatureList();
617             for (String feature : featureList)
618             {
619                     TableColumn column = new TableColumn(samplesTable, SWT.NONE);
620                     column.setText(feature);
621             }
622
623             //For the last column - classification
624             TableColumn column = new TableColumn(samplesTable, SWT.NONE);
625             column.setText("Class");
626
627             ArrayList<Sample> samplesList = samplesCollection.getSampleAsArrayList();
628
629             for(Sample sample : samplesList)
630             {
631                     //sample.display();
632                     //System.out.println();
633                     TableItem item = new TableItem (samplesTable, SWT.NONE);
634
635                     int featureIndex = 0;
636                     for(String feature : featureList)
637                             item.setText(featureIndex++, "" + sample.getFeature(feature).
                                getValue());
638                                 item.setText(featureIndex, sample.getClassification());
639             }
640
```

```
641             for (int i=0; i < featureList.size() + 1; i++)
642             {
643                     samplesTable.getColumn(i).pack();
644             }
645     }
646
647     /*
648      * This method is used to highlight all the samples in the graphical tables that have been
            classified
649      *  incorrectly by the chosen decision tree.
650      */
651     private void highlightIncorrectlyClassifiedSamples()
652     {
653             if(currentException == null)
654                     return;
655
656             ArrayList<Integer> trainingErrorIndices = currentException.
                getTrainingErrorIndices();
657             for(int index : trainingErrorIndices)
658             {
659                     trainingSamplesTable.getItem(index).setBackground(display.
                        getSystemColor(SWT.COLOR_RED));
660                     trainingSamplesTable.getItem(index).setForeground(display.
                        getSystemColor(SWT.COLOR_YELLOW));
661             }
662
663             ArrayList<Integer> testErrorIndices = currentException.getTestErrorIndices();
664             for(int index : testErrorIndices)
665             {
666                     testingSamplesTable.getItem(index).setBackground(display.
                        getSystemColor(SWT.COLOR_RED));
```

```
667                         testingSamplesTable.getItem(index).setForeground(display.
                                getSystemColor(SWT.COLOR_YELLOW));
668                 }
669         }
670
671         /*
672          * Adds a table for manual feature selection by the user.
673          */
674         private void addFeatureSelectorTable()
675         {
676                 Table previousTable = featureSelectorTable;
677
678                 featureSelectorTable = new Table(shell, SWT.MULTI| SWT.BORDER | SWT.
                        FULL_SELECTION | SWT.V_SCROLL);
679                 featureSelectorTable.setLinesVisible (true);
680                 featureSelectorTable.setHeaderVisible (true);
681                 featureSelectorTable.setVisible(false);
682
683                 GridData data = new GridData(); //SWT.FILL, SWT.FILL, false, false);
684
685                 data.heightHint = 25;
686                 if(comboDatasetBox.getText().startsWith("WHINE"))   //Special case...can't
                        resize row height later, due to bug https://bugs.eclipse.org/bugs/show_bug.
                        cgi?id=154341
687                         data.heightHint = 50;
688                 if(comboDatasetBox.getText().startsWith("HORSE"))
689                         data.heightHint = 35;
690
691                 data.widthHint = 200;
692                 data.horizontalSpan = gridHorizontalSpacing;
693                 data.horizontalAlignment = GridData.FILL;
694                 featureSelectorTable.setLayoutData(data);
```

```
695
696             //This block is executed if a previous table has to be overwritten with a new
                    table
697             if(previousTable != null)
698             {
699                     featureSelectorTable.moveAbove(previousTable);
700                     previousTable.dispose();
701                     featureSelectorTable.getParent().layout();
702             }
703
704             ArrayList<String> featureList = Genome.getSamples().getfeatureList();
705             for (String feature : featureList)
706             {
707                     TableColumn column = new TableColumn(featureSelectorTable, SWT.
                            NONE);
708                     column.setMoveable(true);
709                     column.setText(feature);
710             }
711
712             double minWidth = 0;
713             TableItem item = new TableItem(featureSelectorTable, SWT.NONE);
714
715             featureSelectorButtons = new Button[featureList.size()];
716
717             for(int i=0; i<featureList.size(); i++)
718             {
719                     featureSelectorButtons[i] = new Button(featureSelectorTable, SWT.
                            CHECK);
720                     featureSelectorButtons[i].pack();
721                     TableEditor editor = new TableEditor(featureSelectorTable);
722                     Point size = featureSelectorButtons[i].computeSize(SWT.DEFAULT,
                            SWT.DEFAULT);
```

```
723                     editor.minimumWidth = size.x;
724                     minWidth = Math.max(size.x, minWidth);
725                     editor.minimumHeight = size.y;
726                     editor.horizontalAlignment = SWT.CENTER;
727                     editor.verticalAlignment = SWT.CENTER;
728                     editor.setEditor(featureSelectorButtons[i], item , i);
729             }
730
731             for (int i=0; i < featureList.size(); i++)
732             {
733                     featureSelectorTable.getColumn(i).pack();
734             }
735
736             TableItem item1 = featureSelectorTable.getItem(0);
737             System.out.println(item1);
738     }
739
740     /*
741      * Creates a text area where the results of the classification process can be displayed
742      */
743     private void addResultDisplay()
744     {
745             accuracyTextArea = new StyledText (shell, SWT.BORDER);
746             accuracyTextArea.setVisible(false);
747
748             GridData gridData = new GridData();
749             gridData.horizontalSpan = gridHorizontalSpacing / 2;
750             gridData.heightHint = 70;
751             gridData.horizontalAlignment = GridData.FILL;
752             (accuracyTextArea).setLayoutData(gridData);
753     }
754
```

```
755         /*
756          * This creates a table of 1 row, that accepts user-input for classification
757          */
758         private void addEditableSamplesTable()
759         {
760                 Table tempTable = userSamplesTable;     //Required for replacement in Grid
                        Layout
761
762                 userSamplesTable = new Table (shell, SWT.MULTI | SWT.BORDER | SWT.
                        FULL_SELECTION);
763                 userSamplesTable .setLinesVisible (true);
764                 userSamplesTable .setHeaderVisible (true);
765                 userSamplesTable.setVisible(false);
766
767                 GridData data = new GridData(); //SWT.FILL, SWT.FILL, false, false);
768                 data.heightHint = 60;
769                 data.widthHint = 100;
770                 data.horizontalSpan = gridHorizontalSpacing / 2;
771                 data.horizontalAlignment = GridData.FILL;
772                 userSamplesTable.setLayoutData(data);
773
774                 //This block is executed if a previous table has to be overwritten with a new
                        table
775                 if(tempTable != null)
776                 {
777                         userSamplesTable.moveAbove(tempTable);
778                         tempTable.dispose();
779                         userSamplesTable.getParent().layout();
780                 }
781
782                 SampleCollection samplesCollection = new SampleCollection(DataHolder.
                        getTestingSamplesFileName(), DataHolder.getAttributesFileName());
```

```
783
784             ArrayList<String> featureList = samplesCollection.getfeatureList();
785             for (String feature : featureList)
786             {
787                     TableColumn column = new TableColumn(userSamplesTable, SWT.
                            NONE);
788                     column.setText(feature);
789             }
790
791             //Fill initial table with dummy values that can be modified
792             TableItem item = new TableItem (userSamplesTable, SWT.NONE);
793             ArrayList<Sample> samplesList = samplesCollection.getSampleAsArrayList();
794             int featureIndex = 0;
795             for(String feature : featureList)
796                     item.setText(featureIndex++, "" + samplesList.get(4).getFeature(
                            feature).getValue());
797
798
799                         for (int i=0; i < featureList.size(); i++)
800                         {
801                                 userSamplesTable.getColumn(i).pack();
802                         }
803
804
805                         //Editor
806                         final TableEditor editor = new TableEditor (userSamplesTable)
                                ;
807                         editor.horizontalAlignment = SWT.LEFT;
808                         editor.grabHorizontal = true;
809                         userSamplesTable.addListener (SWT.MouseDown, new Listener
                                () {
810                             public void handleEvent (Event event) {
```

```
811                 Rectangle clientArea = userSamplesTable.
                        getClientArea ();
812                 Point pt = new Point (event.x, event.y);
813                 int index = userSamplesTable.getTopIndex ();
814                 while (index < userSamplesTable.
                        getItemCount ()) {
815                     boolean visible = false;
816                     final TableItem item =
                            userSamplesTable.getItem (index)
                            ;
817                     for (int i=0; i<userSamplesTable.
                            getColumnCount (); i++) {
818                         Rectangle rect = item.
                                getBounds (i);
819                         if (rect.contains (pt)) {
820                             final int column = i;
821                             final Text text =
                                    new Text (
                                    userSamplesTable
                                    , SWT.NONE);
822                             Listener textListener
                                    = new Listener
                                    () {
823                                 public void
                                        handleEvent
                                        (final
                                        Event e
                                        ) {
824                                     switch
                                            (
                                            e
```

```
.
type
)

{

825 case

SWT
.
FocusOut
:

826 item
.
setText

(
column
,

text
.
getTex

()
)
;

827 text
.
dispose
```

```
();
break;
case SWT.Traverse:
switch (e.detail) {
case SWT.TRAVE
:
```

```
832                                                                    ite

833                                                                    //

834                                                                    case

                                                                         SWT

                                                                         .

                                                                         TRAVE

                                                                         :

835                                                                      tex
```

```
836                                                                 e
837                                                             }
838                                                         break
                                                                ;
839                                                     }
840                                                 }
841                                             };
842                                             text.addListener (
                                                    SWT.FocusOut,
                                                     textListener);
843                                             text.addListener (
                                                    SWT.Traverse,
                                                    textListener);
844                                             editor.setEditor (text
                                                    , item, i);
845                                             text.setText (item.
                                                    getText (i));
846                                             text.selectAll ();
847                                             text.setFocus ();
```

```
848                                                     return;
849                                                 }
850                                                 if (!visible && rect.intersects
                                                        (clientArea)) {
851                                                     visible = true;
852                                                 }
853                                             }
854                                             if (!visible) return;
855                                             index++;
856                                         }
857                                     }
858                                 });
859         }
860
861         /*
862          * Adds a "classify" button to classify the sample entered by the user in the Editable
                Table.
863          */
864         private Button addClassifyButton()
865         {
866                 Button button = new Button(shell, SWT.PUSH);
867                 button.setText("Classify");
868                 button.setVisible(false);
869
870                 addToGrid(button, gridHorizontalSpacing / 3);
871
872
873
874                 button.addSelectionListener(new SelectionAdapter() {
875                         @Override
876                         public void widgetSelected(SelectionEvent e)
877                         {
```

```
878                             String line = "";
879                             TableItem item = userSamplesTable.getItem(0);
880                             for(int i = 0; i < userSamplesTable.getColumnCount(); i++)
881                                     line += item.getText(i) + ",";
882                             line += "null";
883                             System.out.println(line);
884
885                             Sample sample = new Sample(line, Genome.getSamples().
                                    getfeatureList());
886                             //sample.display();
887
888                             SampleCollection trainingSamples = currentException.
                                    getCurrentDTClassifier().getTrainingSamples();
889                             trainingSamples.discretizeSample(sample);
890                             sample.display();
891
892                             String classification = currentException.getCurrentDTClassifier
                                    ().Classify(sample);
893                             classifyResultLabel.setText("Result: " + classification);
894                     }
895             });
896
897             return button;
898     }
899
900     /*
901      * Draws the generated optimal decision tree in the GUI, replacing any instance of an
            older decision tree
902      */
903     private void addGraphicalDecisionTree()
904     {
905             Tree previousTree = graphicalDecisionTree;
```

```
906
907                 graphicalDecisionTree = new Tree(shell, SWT.BORDER);
908
909                 if(currentException != null)
910                         currentException.getCurrentDTClassifier().getGraphicalDecisionTree(
                            graphicalDecisionTree);
911
912                 addToGrid(graphicalDecisionTree, gridHorizontalSpacing / 2);
913
914                 //This block is executed if a previous table has to be overwritten with a new
                        table
915                 if(previousTree != null)
916                 {
917                         graphicalDecisionTree.moveAbove(previousTree);
918                         previousTree.dispose();
919                         graphicalDecisionTree.getParent().layout();
920                 }
921         }
922
923         /*
924          * This method creates a label widget with the lyrics as the text parameter, and size =
                 horizontalSpacing
925          *   and style as parameter.
926          *   Returns the Initialized label
927          */
928         private Label addLabel(String lyrics, int horizontalSpacing, int style)
929         {
930                 Label label = new Label(shell, style);
931                 label.setText(lyrics);
932                 addToGrid(label, horizontalSpacing);
933                 return label;
934         }
```

```
935
936         /*
937          * A Generic Method to attach a generic widget to the grid, initialized with horizontal
                 span
938          */
939         private <T> void addToGrid(T widget, int horizantalSpan)
940         {
941                 GridData gridData = new GridData();
942                 gridData.horizontalSpan = horizantalSpan;
943                 gridData.horizontalAlignment = GridData.FILL;
944                 ((Control) widget).setLayoutData(gridData);
945         }
946
947         /*
948          * Shows/Hides all widgets that shouldn't be displayed in the GUI because algorithm
                 hasn't been selected (Step 1)
949          */
950         private void toggleIllegalWidgetsForStep1(boolean flag)
951         {
952                 fitnessSliderLabel.setVisible(flag);
953                 fitnessSlider.setVisible(flag);
954                 crossoverSliderLabel.setVisible(flag);
955                 crossoverSlider.setVisible(flag);
956                 mutationSliderLabel.setVisible(flag);
957                 mutationSlider.setVisible(flag);
958                 runButton.setVisible(flag);
959         }
960
961         /*
962          * Shows/Hides all widgets that shouldn't be displayed before the Genetic Algorithm is
                 run
963          */
```

```
964         private void toggleIllegalWidgetsForStep2(boolean flag)
965         {
966                 userSamplesTable.setVisible(flag);
967                 classifyButton.setVisible(flag);
968                 saveDTButton.setVisible(flag);
969                 classifyResultLabel.setVisible(flag);
970                 graphicalDecisionTree.setVisible(flag);
971                 discretizeCheckBox.setVisible(flag);
972         }
973 }
```

Listing 7.15: ClassifyWindow.java

```
1  package org.ck.gui;
2
3  import java.io.BufferedReader;
4  import java.io.FileNotFoundException;
5  import java.io.FileReader;
6  import java.io.IOException;
7  import java.util.ArrayList;
8
9  import org.ck.dt.DecisionTreeClassifier;
10 import org.ck.gui.Constants.DatasetOptions;
11 import org.ck.sample.DataHolder;
12 import org.ck.sample.Sample;
13 import org.ck.sample.SampleCollection;
14 import org.eclipse.swt.SWT;
15 import org.eclipse.swt.events.SelectionAdapter;
16 import org.eclipse.swt.events.SelectionEvent;
17 import org.eclipse.swt.graphics.Font;
18 import org.eclipse.swt.graphics.Image;
19 import org.eclipse.swt.layout.GridData;
20 import org.eclipse.swt.layout.GridLayout;
```

```
21 import org.eclipse.swt.widgets.Button;
22 import org.eclipse.swt.widgets.Combo;
23 import org.eclipse.swt.widgets.Control;
24 import org.eclipse.swt.widgets.Display;
25 import org.eclipse.swt.widgets.Event;
26 import org.eclipse.swt.widgets.Group;
27 import org.eclipse.swt.widgets.Label;
28 import org.eclipse.swt.widgets.Listener;
29 import org.eclipse.swt.widgets.Shell;
30 import org.eclipse.swt.widgets.Text;
31 import org.eclipse.swt.widgets.Tree;
32
33 public class ClassifyWindow {
34         private Shell shell;
35
36         private int gridHorizontalSpacing = 10;
37         private int gridVerticalSpacing = 4;
38         private int gridMarginBottom = 5;
39         private int gridMarginTop = 5;
40
41         private Label []featureLabels;
42         private Text []featureTextBox;
43         private ArrayList<String> featureList;
44
45         private SampleCollection samples;
46
47         private DecisionTreeClassifier dtClassifier;
48         private Tree graphicalDecisionTree = null;
49
50         private Combo dataSetSelectorCombo;
51         private Combo treeSelectorCombo;
52         private Label ClassificationLabel;
```

```
53
54          private Label infoLabel;
55
56          public ClassifyWindow(Display display)
57          {
58                  shell = new Shell(display);
59                  shell.setSize(1032, 500);
60                  shell.setBackgroundImage(new Image(display, "Icons/white_background.png"));
61                  initUI();
62                  shell.open ();
63                  while (!shell.isDisposed()) {
64                          if (!display.readAndDispatch ()) display.sleep ();
65                  }
66                  display.dispose ();
67
68          }
69
70          private void initUI() {
71                  GridLayout gridLayout = new GridLayout(2, true);
72                  gridLayout.horizontalSpacing = gridHorizontalSpacing;
73                  gridLayout.verticalSpacing = gridVerticalSpacing;
74                  gridLayout.marginBottom = gridMarginBottom;
75                  gridLayout.marginTop = gridMarginTop;
76                  shell.setLayout(gridLayout);
77                  shell.setText("Classifier Window");
78
79                  initDataLoaderPart();
80                  initFeatureReaderPart();
81                  initDTSelector();
82                  initGraphicalDecisionTree();
83                  ClassificationLabel = new Label(shell,SWT.BORDER);
```

```
84                  ClassificationLabel.setFont(new Font(shell.getDisplay(), "Helvectica", 20, SWT.
                        ITALIC | SWT.BOLD));
85                  ClassificationLabel.setText("Classification Result\n Appears Here");
86
87          }
88
89          private void initButtons(Group featureReader) {
90                  Button ClassifyButton = new Button(featureReader,SWT.PUSH | SWT.
                        CENTER);
91                  ClassifyButton.setText("Classify");
92                  Button ResetButton = new Button(featureReader,SWT.PUSH | SWT.CENTER
                        );
93                  ResetButton.setText("Reset");
94                  ClassifyButton.addListener(SWT.MouseDown, new Listener() {
95
96                          @Override
97                          public void handleEvent(Event event) {
98                                  String featureLine = "";
99                                  for(int i=0; i<featureList.size(); ++i)
100                                 {
101                                         featureLine += featureTextBox[i].getText()+",";
102                                 }
103                                 featureLine += DataHolder.getPositiveClass();//Ignore this,
                                        only for the format (Check the Sample constructor).
104                                 Sample currentSample= new Sample(featureLine, featureList);
105                                 samples.discretizeSample(currentSample);
106                                 String Classification = dtClassifier.Classify(currentSample);
107                                 ClassificationLabel.setText("Classification: \n" + Classification)
                                        ;
108 //      if(Classification.equals(DataHolder.getPositiveClass()))
109 //       ClassificationLabel.setBackground(new Color(shell.getDisplay(), 0, 1, 0));
110 //      else
```

```
111 //    ClassificationLabel.setBackground(new Color(shell.getDisplay(), 1, 0, 0));
112
113
114                     }
115             });
116             ResetButton.addListener(SWT.MouseDown, new Listener() {
117
118                     @Override
119                     public void handleEvent(Event event) {
120                             for(int i=0; i<16; i++)
121                             {
122                                     featureTextBox[i].setText(" 0");
123                             }
124                             ClassificationLabel.setText("Classification Result\n Appears
                                    Here");
125
126                     }
127             });
128     }
129
130     private void initGraphicalDecisionTree() {
131             Tree previousTree = graphicalDecisionTree;
132
133             graphicalDecisionTree = new Tree(shell, SWT.BORDER);
134             dtClassifier.getGraphicalDecisionTree(graphicalDecisionTree);
135
136             //graphicalDecisionTree.setLayoutData(gridData);
137             //This block is executed if a previous table has to be overwritten with a new
                    table
138             if(previousTree != null)
139             {
140                     graphicalDecisionTree.moveAbove(previousTree);
```

```
141                     previousTree.dispose();
142                     graphicalDecisionTree.getParent().layout();
143             }
144
145     }
146
147     private void initFeatureReaderPart() {
148             Group featureReader = new Group(shell, SWT.NONE);
149             featureReader.setText("Enter Feature Values"); //[0-"+Constants.
                    NUMBER_OF_BINS+"]");
150             GridLayout featureReaderLayout = new GridLayout(6, false);
151             featureReaderLayout.marginWidth = 5;
152             featureReaderLayout.marginHeight = 5;
153             featureReaderLayout.horizontalSpacing = 5;
154             featureReader.setLayout(featureReaderLayout);
155
156             featureLabels = new Label[16];
157             featureTextBox = new Text[16];
158
159             for(int i=0; i<16; ++i)
160             {
161                     featureLabels[i] = new Label(featureReader, SWT.WRAP|SWT.
                            BORDER);
162                     featureTextBox[i] = new Text(featureReader,SWT.WRAP);
163             }
164             initFeatureLabels();
165             initButtons(featureReader);
166     }
167
168     private void initDataLoaderPart() {
169             Group dataLoader = new Group(shell, SWT.NONE);
170             dataLoader.setText("Load The Decision Tree");
```

```
171                 GridLayout dataLoaderLayout = new GridLayout(2,false);
172                 dataLoaderLayout.marginWidth = 5;
173                 dataLoaderLayout.marginHeight = 5;
174                 dataLoader.setLayout(dataLoaderLayout);
175
176                 Label dogName = new Label(dataLoader, SWT.NONE|SWT.CENTER);
177                 dogName.setText("Select Dataset");
178                 dataSetSelectorCombo = new Combo(dataLoader, SWT.DROP_DOWN);
179         for(int i = 0; i < DatasetOptions.values().length; i++)
180                 dataSetSelectorCombo.add("" + DatasetOptions.values()[i]);
181
182         dataSetSelectorCombo.select(0);
183         DataHolder.setDataset(DatasetOptions.valueOf(dataSetSelectorCombo.getText()));
184         samples = new SampleCollection(DataHolder.getTrainingSamplesFileName(),
                DataHolder.getAttributesFileName());
185         samples.discretizeSamples(Constants.DiscretizerAlgorithms.EQUAL_BINNING);
186         dataSetSelectorCombo.addSelectionListener(new SelectionAdapter() {
187
188             @Override
189             public void widgetSelected(SelectionEvent e) {
190                 DataHolder.setDataset(DatasetOptions.valueOf(dataSetSelectorCombo.getText()
                        ));
191                 initFeatureLabels();
192                 initDTSelector();
193                 samples = new SampleCollection(DataHolder.getTrainingSamplesFileName(),
                        DataHolder.getAttributesFileName());
194                 samples.discretizeSamples(Constants.DiscretizerAlgorithms.EQUAL_BINNING);
195                 //Genome.reInitializeStaticVariables();
196             }
197
198
199         });
```

```
200         infoLabel = new Label(shell,SWT.None | SWT.BORDER_SOLID);
201         infoLabel.setText("General Information Label");
202         Label savedValues = new Label(dataLoader, SWT.NONE|SWT.CENTER);
203                 savedValues.setText("Select Decision Tree");
204                 treeSelectorCombo = new Combo(dataLoader, SWT.DROP_DOWN);
205
206                 treeSelectorCombo.select(0);
207                 treeSelectorCombo.addSelectionListener(new SelectionAdapter() {
208
209             @Override
210             public void widgetSelected(SelectionEvent e) {
211                 initDecisionTree();
212             };
213         });
214
215
216         }
217         private void initDecisionTree() {
218                 String selection = treeSelectorCombo.getText();
219         String featureLine = selection.split("->")[0];
220         //System.out.println(featres[0]);
221         String features[] = featureLine.split(",");
222         ArrayList<String> featrList = new ArrayList<String>();
223         for(int i=0; i<features.length; ++i)
224                 featrList.add(features[i]);
225         infoLabel.setText("Decision Tree with features\n"+featrList+"\nConstructed has a
              accuracy of "+selection.split("->")[1]);
226
227         dtClassifier = new DecisionTreeClassifier(samples,featrList);
228                 initGraphicalDecisionTree();
229         }
230         private void initDTSelector()
```

```
231         {
232                 treeSelectorCombo.removeAll();
233                 try {
234                         BufferedReader br = new BufferedReader(new FileReader(DataHolder.
                            getSaveDatoToFileName()));
235                         while(true)
236                         {
237                                 String line = br.readLine();
238                                 if(line == null)
239                                         break;
240                                 treeSelectorCombo.add(line);
241                         }
242                         treeSelectorCombo.select(0);
243
244                 } catch (FileNotFoundException e1) {
245                         e1.printStackTrace();
246                 } catch (IOException e) {
247                         e.printStackTrace();
248                 }
249                 initDecisionTree();
250         }
251         private void initFeatureLabels() {
252                 featureList = new ArrayList<String>();
253
254                 try {
255                         BufferedReader br = new BufferedReader(new FileReader(DataHolder.
                            getAttributesFileName()));
256                         int i = 0;
257                         while(true)
258                         {
259                                 String line = br.readLine();
260                                 if(line == null)
```

```
261                                     break;
262                             featureList.add(line);
263                     }
264                     for(i=0;i<featureList.size();i++)
265                     {
266                             //if(i>=11)
267                             {
268                                     //featureLabels[i].setVisible(true); //Some Bug here!
269                                     //featureTextBox[i].setVisible(true);
270                             }
271                             featureLabels[i].setText(featureList.get(i));
272                             featureTextBox[i].setText(" 0 ");
273                             System.out.println(""+featureList.get(i));
274
275                     }
276                     for(;i<16;i++)
277                     {
278                             //featureLabels[i].setVisible(false);
279                             //featureTextBox[i].setVisible(false);
280                     }
281             } catch (FileNotFoundException e) {
282                     e.printStackTrace();
283             } catch (IOException e) {
284                     e.printStackTrace();
285             }
286     }
287     public static void main(String args[])
288     {
289             new ClassifyWindow(new Display());
290
291     }
292
```

```
293 }
```

Listing 7.16: BrowserWindow.java

```
1  package org.ck.gui;
2
3  import org.eclipse.swt.SWT;
4  import org.eclipse.swt.browser.Browser;
5  import org.eclipse.swt.layout.FillLayout;
6  import org.eclipse.swt.widgets.Display;
7  import org.eclipse.swt.widgets.Shell;
8
9  public class BrowserWindow {
10         private Browser browser;
11         private Shell shell;
12         public BrowserWindow(Display display)
13         {
14                 shell = new Shell(display);
15                 initUI();
16
17                 shell.setSize(720, 720);
18                 shell.open();
19                 while(!shell.isDisposed())
20                 {
21                         if(!display.readAndDispatch())
22                                 display.sleep();
23                 }
24         }
25         private void initUI() {
26                 FillLayout fillLayout = new FillLayout();
27                 shell.setLayout(fillLayout);
28                 browser = new Browser(shell,SWT.NONE);
29                 browser.setUrl("https://www.github.com/samiriff/GWClassifier");
```

```
30
31
32          }
33
34  }
```

Listing 7.17: MainClass.java

```
1   package org.ck.gui;
2
3   import java.io.IOException;
4   import java.util.ArrayList;
5
6   import org.ck.dt.DecisionTreeClassifier;
7   import org.ck.ga.DTOptimizer;
8   import org.ck.ga.OptimalScoreException;
9   import org.ck.ga.Population;
10  import org.ck.sample.DataHolder;
11  import org.ck.sample.SampleCollection;
12
13  public class MainClass implements Constants{
14
15          public static void main(String args[]) throws IOException, OptimalScoreException
16          {
17                  System.out.println("Hello, Welcome to the Decision Tree Based Classifier");
18
19                  //sampleCaller(); // This is for nsatvik
20                  //sampleCaller2(); //This is for samiriff
21          }
22
23
24          public static void sampleCaller2()throws OptimalScoreException
25          {
```

```
26              Population population = null;
27              try {
28                      population = new Population();
29                      population.displayPopulation();
30
31                      System.out.println("Starting Genetic Algorithm Engine...");
32                      System.out.println(DataHolder.getPositiveClass());
33                      System.out.println(DataHolder.getFitnessScoreThreshold());
34                      Thread.sleep(0);
35                      population.runGeneticAlgorithm();
36
37              }
38              catch (InterruptedException e)
39              {
40                      e.printStackTrace();
41              }
42
43              System.out.println(DataHolder.getFitnessScoreThreshold());
44              System.out.println(DataHolder.getCrossoverProbabilityThreshold());
45              System.out.println(DataHolder.getMutationProbabilityThreshold());
46      }
47
48      /*
49       * I call this method from the Classifier Window.
50       */
51      public static DecisionTreeClassifier sampleCaller(ArrayList<String> featureList)
52      {
53
54              SampleCollection samples = new SampleCollection(DataHolder.
                    getTrainingSamplesFileName(), DataHolder.getAttributesFileName());
55
```

```
56                  SampleCollection testing_samples = new SampleCollection(DataHolder.
                        getTestingSamplesFileName(), DataHolder.getAttributesFileName());
57                  SampleCollection new_samples = new SampleCollection(samples, featureList);
58                  new_samples.discretizeSamples(Constants.DiscretizerAlgorithms.
                        EQUAL_BINNING);
59                  //Discretizing
60                  samples.discretizeSamples(Constants.DiscretizerAlgorithms.EQUAL_BINNING);
61                  testing_samples.discretizeSamples(Constants.DiscretizerAlgorithms.
                        EQUAL_BINNING);
62
63                  //new_samples.displaySamples();
64
65                  //DecisionTreeClassifier dtClassifier = new DecisionTreeClassifier(samples.
                        getSampleCollectionSubset(featureList));
66                  DecisionTreeClassifier dtClassifier = new DecisionTreeClassifier(new_samples);
67
68                  System.out.println("\n\nTest Set Accuracy : ");
69                  dtClassifier.setTestingSamples(testing_samples);
70                  dtClassifier.TestAndFindAccuracy();
71                  dtClassifier.getAccuracy();
72
73                  System.out.println("Training Set Accuracy : ");
74                  dtClassifier.setTestingSamples(samples);
75                  dtClassifier.TestAndFindAccuracy();
76                  dtClassifier.getAccuracy();
77                  return dtClassifier;
78
79          }
80  }
```

Listing 7.18: Constants.java

```
1   package org.ck.gui;
```

```
2
3  public interface Constants
4  {
5  // public static final String TRAINING_SAMPLES_FILE_NAME = "Training Data/Horse/horse.
       train";
6  // public static final String TESTING_SAMPLES_FILE_NAME = "Training Data/Horse/horse.
       test";
7  // public static final String ATTRIBUTES_FILE_NAME = "Training Data/Horse/horse.attribute";
8
9          enum Category
10         {
11                 VERY_LOW,
12                 LOW,
13                 MEDIUM,
14                 HIGH,
15                 VERY_HIGH
16         }
17
18         enum DiscretizerAlgorithms
19         {
20                 MEDIAN,
21                 EQUAL_BINNING
22         }
23
24         enum DatasetOptions
25         {
26                 HORSE_DATASET,
27                 WATER_DATASET,
28                 WHINE_DATASET
29         }
30         enum Filenames
31         {
```

```
32                  TRAINING_SAMPLES_FILE,
33                  FEATURES_FILE
34          }
35
36          public static final int NUMBER_OF_BINS = 6;
37
38          public static final int POPULATION_SIZE = 75;
39          public static final int NUM_OF_GENERATIONS = 150;
40          public static final double FITNESS_SCORE_THRESHOLD = 0.87;
41          public static final double CROSSOVER_PROBABILITY_THRESHOLD = 0.85;
42          public static final double MUTATION_PROBABILITY_THRESHOLD = 0.025;
43
44          public static final double TRAINING_SET_WEIGHT = 0.75;
45          public static final double TEST_SET_WEIGHT = 0.25;
46
47
48  }
```

Appendix B : Screen Shots

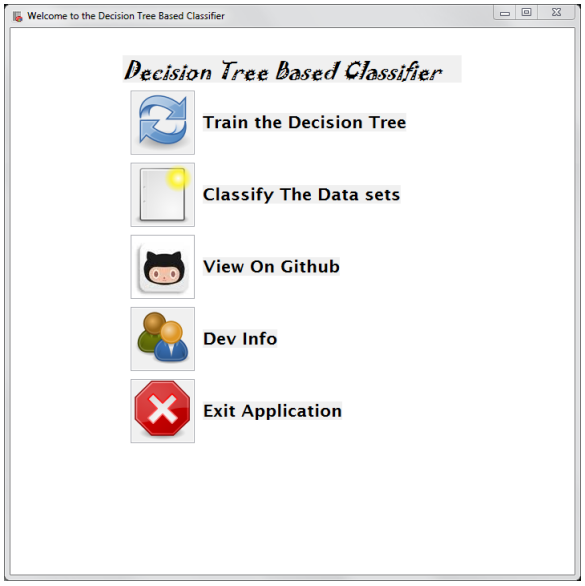

Figure 7.1: Application Window - Welcome Screen

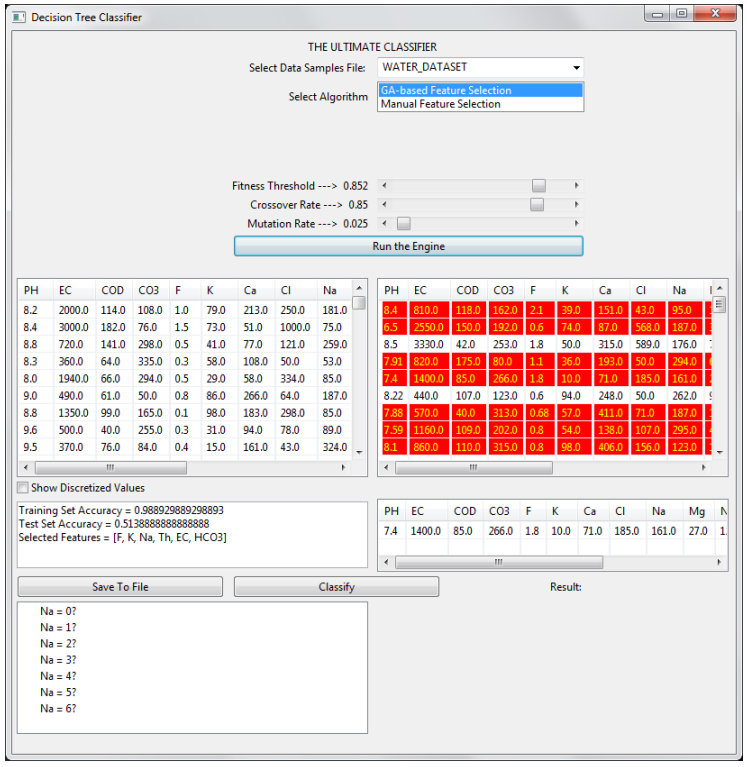

Figure 7.2: Decision Tree Constructor Window.

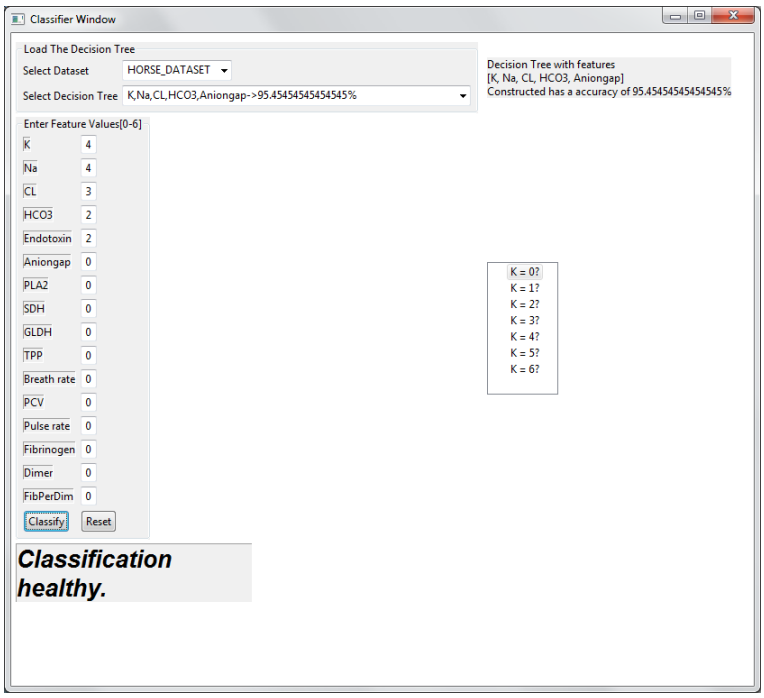

Figure 7.3: Decision Tree Classifier Window.

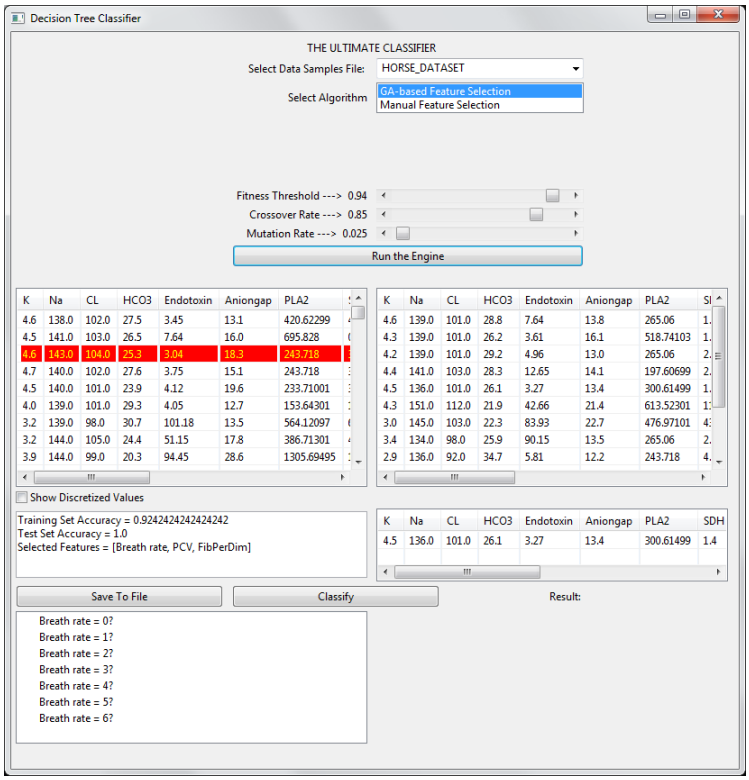

Figure 7.4: Decision Tree Construction with GA based Feature Selector.

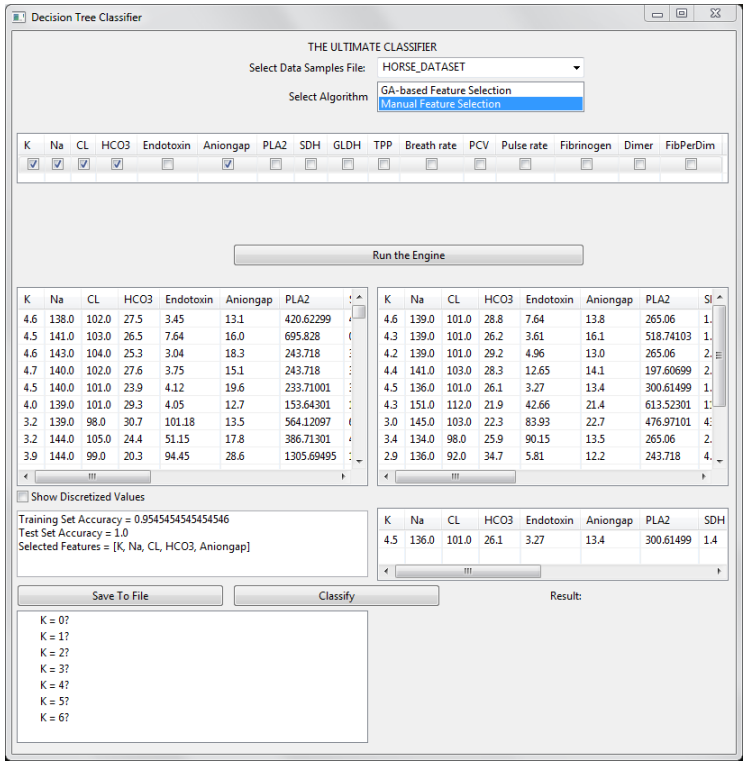

Figure 7.5: Decision Tree Construction with manual feature selection.

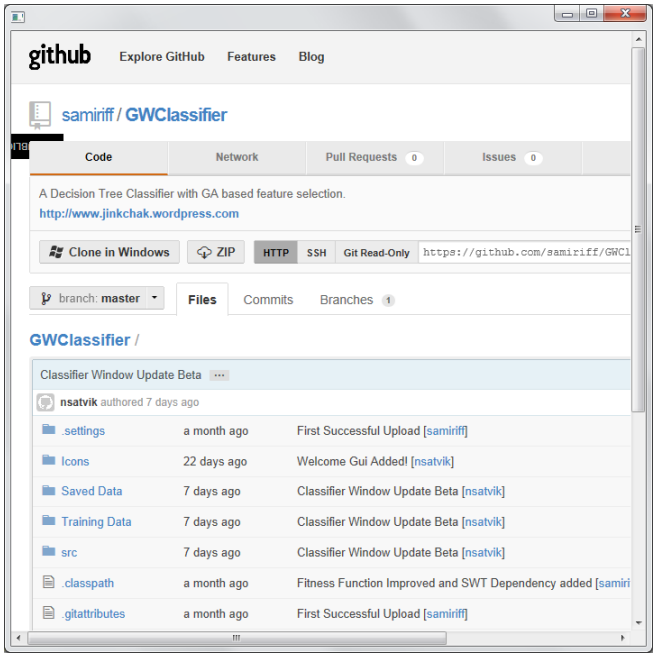

Figure 7.6: The project source code on github public repository.