Decision Trees

CIS3187 – Business Intelligence

B.Sc. Information Technology (Artificial Intelligence)

247696M

Edward Fleri Soler

Table of Contents

Statement of Completion

To the best of my knowledge, all parts of the implemented decision tree algorithm work and all program requirements have been met.

Introduction

In this assignment, a decision tree algorithm (namely ID3) is designed, constructed and tested with the provided dataset. This algorithm has been implemented in Java and, through this documentation, shall be demonstrated and its functionality reported. We shall start with a brief introduction to decision trees before we discuss the design and structure of the algorithm. Finally, we shall test the algorithm and demonstrate its functionality.

Background Knowledge

Decision Trees

A decision tree is a classification data structure implemented to classify input examples based on pre-learned data. Decision trees are first fed training data, consisting of a number of examples of discrete valued attributes together with the final classification label. An example of a common dataset used to test decision trees in literature is the 'PlayTennis' dataset. This dataset consists of a number of discrete attributes relating to weather conditions: Outlook, Humidity, Wind, etc. Training data consists of a combination of discrete values for each of the attributes, together with a final target attribute called 'PlayTennis'. This data structure may then be trained upon a set of examples, to predict the target attribute for future unseen cases. This relates to taking a decision based upon available data, thus duly naming the data structure a decision tree.

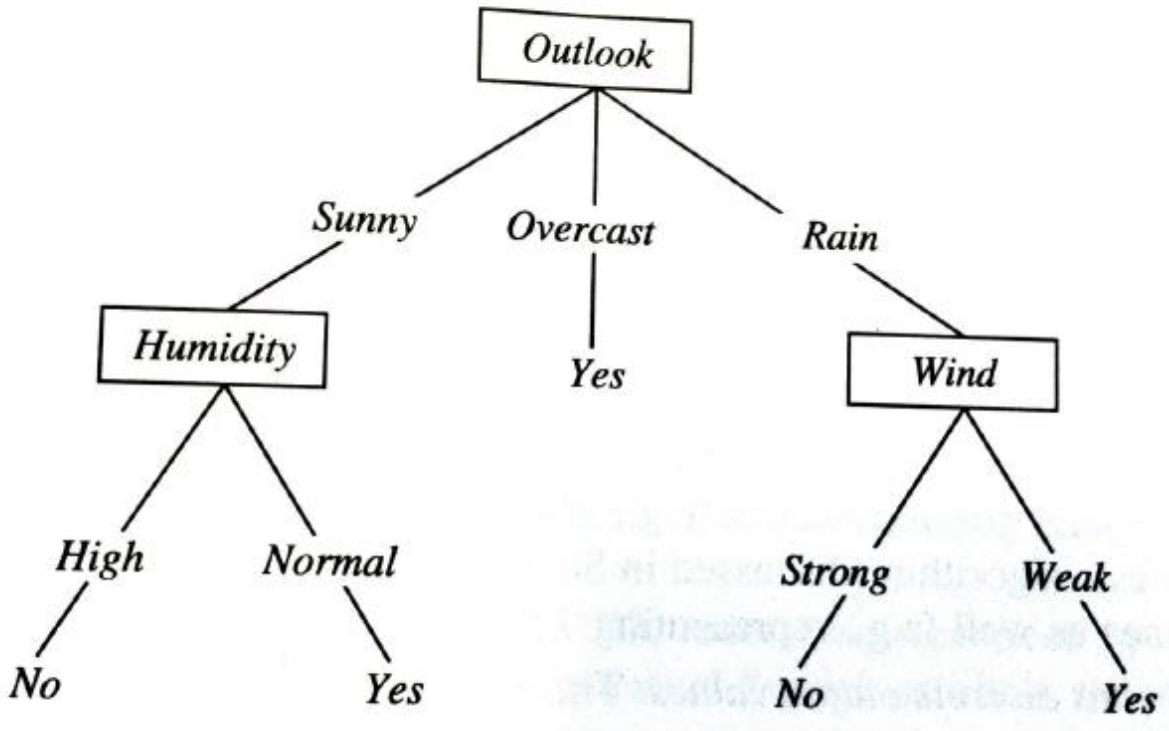

FIGURE 1 EXAMPLE OF DECISION TREE DATA STRUCTURE – ADAPTED FROM NOTES BY KRISTIAN GUILLAUMIER

Observing figure 1 above, one may easily grasp the concept of decision trees. Starting from the root node, the graph is traversed in a depth-first manner. At each node, the value of the attribute in question is observed, and the respective branch is followed downwards. This traversal continues until a leaf node is reached, which holds the outcome value for the target attribute. Decision trees represent disjunctions of conjunctions, whereby each possible path relates to a disjunction, and each successive edge in the path from the root node to a leaf node relates to a conjunction.

The data structure may seem simple in structure, however, the efficiency and effectiveness of this data structure is brought out by the order in which attribute nodes are structured within the tree. For efficiency's sake, the data structure is constructed in such a way as to conclude on the predicted outcome in the least possible number of steps down the tree. This is where entropy and information gain come in, two key words which shall be repeatedly mentioned in the remainder of this documentation.

Entropy

In the context of computer science and information theory, entropy is a measure of randomness, or predictability, of any given event. An event with an entropy of 0 is absolutely predictable, such as the prediction that flipping a coin with both sides as heads will always yield an outcome of heads. An entropy value of 1 relates to a 50:50 outcome, and a higher entropy value relates to a more chaotic and unpredictable outcome.

Information Gain

Information gain is the expected decrease in entropy by considering a new attribute outcome in conjunction to a previous outcome. Therefore, information gain is measured by comparing the entropy of a given event with x attributes considered, with the entropy of a given event with x+1 attributes considered. A positive information gain would mean that the newly considered result makes the outcome more predictable, while a negative information gain would mean that the new attribute makes the outcome more unpredictable.

Decision trees are constructed by iteratively considering all possible attributes, together with the training examples provided, to compute which of the attributes result in the greatest information gain at the current point in the construction process. This is repeated until all example outcomes are accounted for in the tree. By the end of the construction process, the attributes positioned at the upper levels of the tree will be the ones which provide the greatest information towards the final classification, while the attributes further down will relate to attributes which have very little effect on the predictability of the outcome.

Implementation

The Java program produced is based upon two classes: Node and DecisionTree. The Node class defines a new data structure, providing us with the parts required to build the decision tree, while the DecisionTree class implements the actual construction of the ID3 data structure.

Node

The Node class constructs the Node data type, which will be used to create the nodes for the decision tree. Each node holds three variables: *name* – a string value which shall hold the attribute name of the current node, *children* – A linked list of type Node, which shall point towards the children of the current node, and finally *branchLabels* – which is a linked list of type string which shall hold the labels for each branch to each child node. On construction, the name of the node is initialised to the attribute being incorporated into the tree, while the two linked lists are declared and left empty.

The lengths of the *branchLabels* and *children* linked lists shall always be the same, as the number of branches from the current node to its children is always equal to the number of children of the current node. The following is a list of methods implemented in this class for the interaction with Node variables:

String getName()

This function simply returns the name of the current node.

Void setName(String)

This function takes a string as an argument and sets the name of the current node to store this string.

LinkedList<Node> getChildren()

This function returns a linked list of type node holding the children of the current node.

LinkedList<String>getBranches()

This function returns a linked list of type string being the branches of the current node to its children.

Void addChild(Node, String)

This function allows a child node to be added to the current node. The arguments passed to this function are the child node itself as well as the string value for the branch label between the current node and its new child.

Void addBranch(String)

This function allows a branch to be added to the current node without having to immediately set a child. This is a slight modification to the *addChild* function for programmatic efficiency.

Boolean deleteChildren()

This function empties the linked list of children for the current node, returning a Boolean value relating to the success of the operation.

Boolean deleteBranches()

This function empties the linked list of branch labels of the current node, returning a Boolean value relating to the success of the operation.

DecisionTree

The DecisionTree class holds the main method as well as other methods required for the construction and testing of the decision tree. We shall now discuss the functionality and purpose of each method within this class.

String readFile(String)

This function takes as argument a file name and returns a string variable with the contents of the text file. This function reads the contents of the data set text file provided by creating a file stream and storing the contents in a string variable. This variable is then returned by the function.

LinkedList<LinkedList<String>> getData(String)

getData is a function which segments and organises the string data returned by the *readFile* function, returning a table like structure with the organised contents. The string variable holding the contents of the data set is passed to this function, which processes it. The string data is traversed character by character, separating values whenever a whitespace or new line is encountered. A nested linked list variable of type string is declared. This data structure will act as a table, with the inner linked list storing the attribute values for each example, and the outer linked list storing each example.

The first list of the input string is ignored, as this relates to the attribute titles and not example values. Next, each value is extracted from the string and input into the nested list data structure. Once all the string has been traversed, the table-like structure is returned.

Double entropy(Double, Double)

The function *entropy* takes as input two double values relating to the total outcome values from the training data. The entropy of the current attribute based on these two values is then computed through the following equation:

$$\sum_{i=1}^{c} - p_i log_2 p_i$$

In our case, *c* is 2 as we are only considering two double values and p_i is equal to the first, or second, value divided by the total (first + second value). The resulting entropy value is then returned.

Node ID3(LinkedList<LinkedList<String>>, LinkedList<LinkedList<String>>, String, LinkedList<String>)

This function is responsible for generating the actual decision tree, and takes in 4 arguments. The first is the organised data, computed in *getData*, the second is another table like structure which holds all examples conforming to the attribute values required to be in the current position of the tree, the third argument is the target attribute name, which in this case is 'PlayTennis', and the final argument is a list holding the set of all attributes: Outlook, Humidity, Wind, etc. This function returns a node, which shall be the root node of the current level. This algorithm shall be implemented recursively, until the tree as a whole has been constructed.

The algorithm starts by considering the total number of 'yes' and 'no' outcomes of all examples. In the case that all outcomes are the same, a node is created and its name set to the unanimous outcome and this node is returned: we are done.

If this is not the case, we take note of the entropy of the decision tree in its current form. We proceed to iterate through the attributes in the list passed as the 4th parameter. For each attribute, we then consider the possible values; such as Sunny, Cloudy, Rainy being the possible values for Outlook. We compute a tally of the quantity of 'yes'/'no' example outcomes for each attribute value - 'yes'/'no' quantity for Sunny, 'yes'/'no' quantity for Cloudy, etc. These quantities are then passed to the *entropy* function, which returns an entropy rating for predictability of the outcome based on the current attribute (example Outlook). This entropy value is compared to the previously computed entropy value to achieve the information gain.

This cycle is completed for all other attributes, with the information gains being compared. The attribute which achieves the greatest information gain at the current point in the decision tree, is the one to be included into the tree next. Therefore, a new node is computed, bearing the name of the chosen attribute. The selected attribute is then removed from the list of attributes (4th parameter) as it shouldn't be considered again in the current path from the root. A recursive call is then made to ID3, passing the respective parameters. This call will return the next 'root' node (which may be a leaf node), and so this 'root' node is set to be the child of the current node in this recursive level.

By means of tail recursion, this function will recursively call itself until the tree is exhausted and all examples are accounted for, producing a decision tree.

Boolean DFS(LinkedList<String>, Node, LinkedList<String>)

The function *DFS* is designed to perform a depth-first search of the tree, taking as arguments an example, the root node of the tree and a list of attributes. This function is designed to take a test example, traverse the decision tree and return the result of the target attribute as a Boolean value.

The base case for this recursive function is hit when a node with no children is reached. This is a leaf node and must hold a value of 'yes' or 'no'. If the base case is not hit, then we compare the name of the current node with the list of attributes, to find out what attribute the current node is testing. We then observe the respective attribute value of the current example, and follow the branch relating to this value. The function is then recursively called, passing the respective child node to continue the traversal downwards.

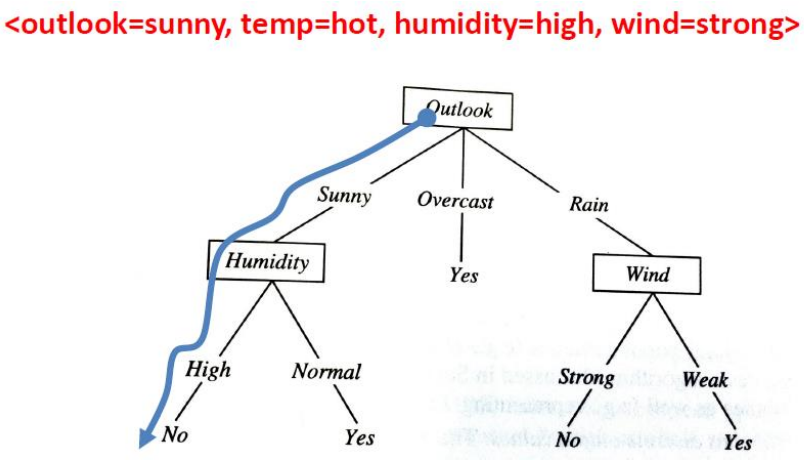

FIGURE 2 DEPTH-FIRST TRAVERSAL OF DECISION TREE - ADAPTED FROM NOTES BY KRISTIAN GUILLAUMIER

Void outputResutls(LinkedList<LinkedList<String>>, Node, LinkedList<String>, String)

This function takes the test data, root node, attributes and target attribute as arguments and outputs the predicted outcome of the decision tree for each test example. The test examples are traversed with their attribute values output on screen. The *DFS* function is then called upon the current example, and the resulting value returned is output as the predicted target attribute value.

Main

The main function simply acts as an environment to bring all the respective parts of the implementation together. This is possible due to the modular design of the code, making it easy to modify separate parts of the implementation separately. The functions discussed above are called and executed to perform the desired function of the system.

Testing

In this section we shall execute the algorithm with test data, observing the output and comparing it to the expected result. The training data used to train the algorithm is stored in a text file 'Dataset.txt' and holds the following examples:

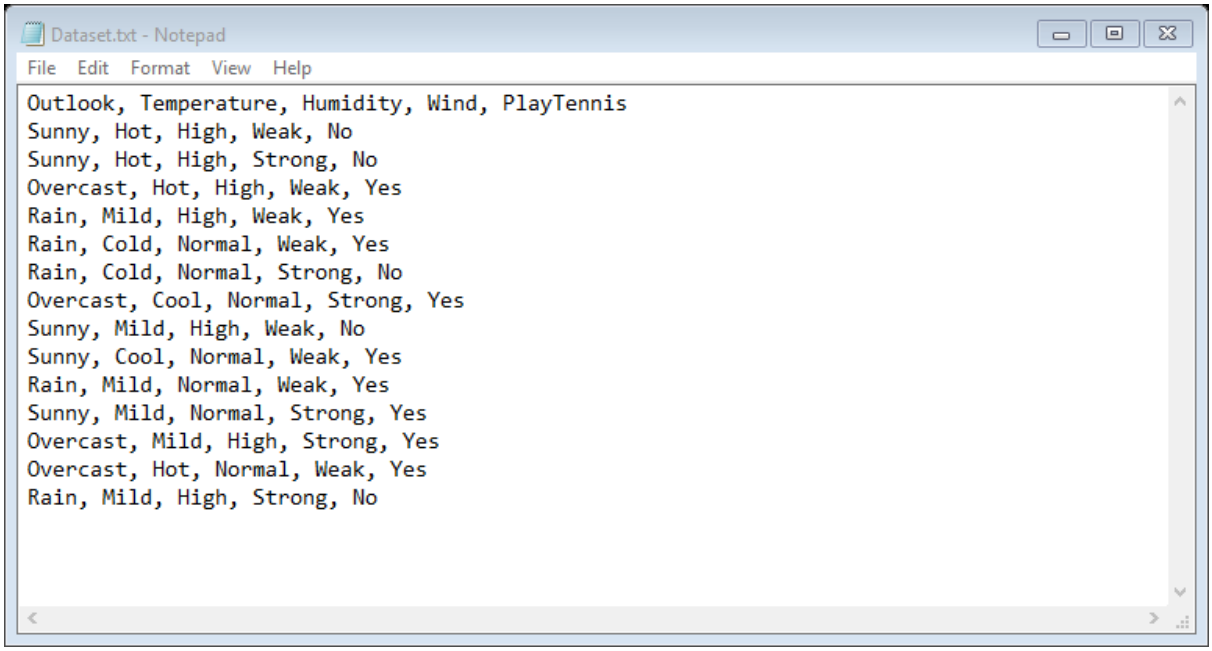

FIGURE 3 TRAINING DATA FOR ALGORITHM

Three test examples have been considered for the testing process, with one taken from the training dataset so as to trivially confirm that the system works, and the other two being random combinations of attribute values. The testing dataset may be found on the next page.

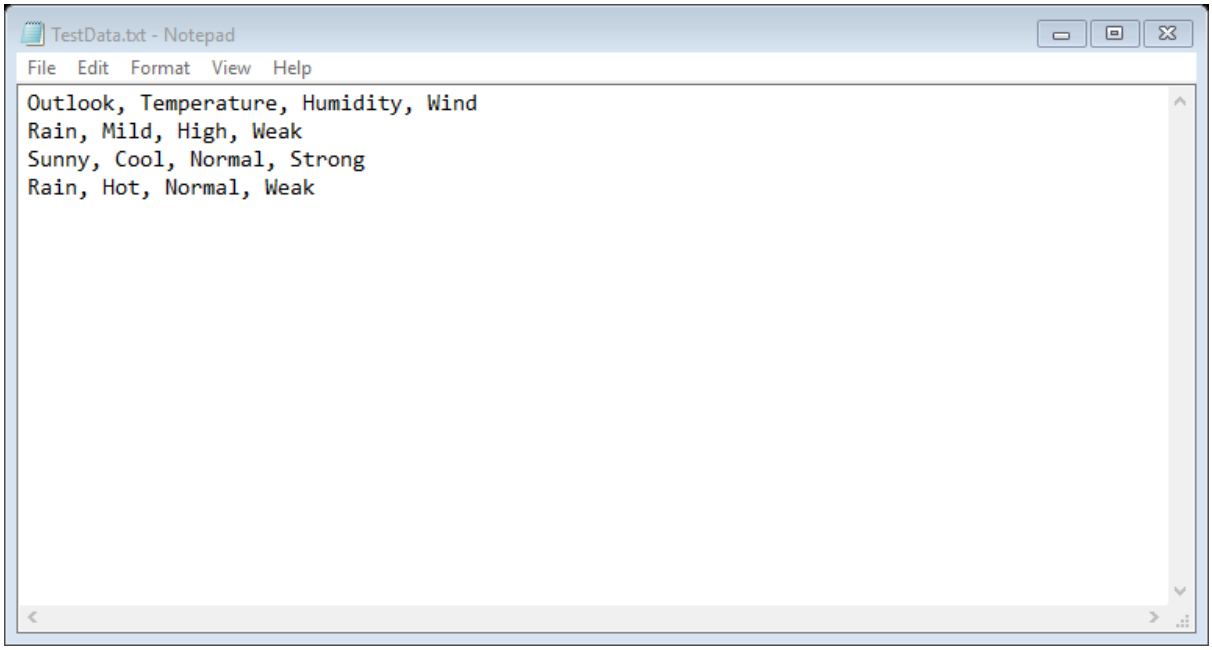

FIGURE 4 TEST DATA FOR ALGORITHM

When executed, the algorithm outputs a verbose description of the calculations being computed while constructing the decision tree. The following is a copy of said description:

```
Information Gain for Outlook is -0.03896446593984659
Information Gain for Temperature is -0.024697970209864528
Information Gain for Humidity is 0.15183550136234136
Information Gain for Wind is 0.04812703040826932
Humidity selected as sub-tree root.
Humidity node created.
Example set reduced to those with Humidity set to High
Information Gain for Outlook is -0.014771863965748366
Information Gain for Temperature is 0.02024420715375619
Information Gain for Wind is 0.02024420715375619
Temperature selected as sub-tree root.
Temperature node created.
Example set reduced to those with Temperature set to Hot
Information Gain for Outlook is -0.08170416594551039
Information Gain for Wind is -0.08170416594551039
Outlook selected as sub-tree root.
Outlook node created.
Example set reduced to those with Outlook set to Sunny
Child of Outlook node's Sunny branch set to No

Example set reduced to those with Outlook set to Overcast
Child of Outlook node's Overcast branch set to Yes

Example set reduced to those with Outlook set to Rain
Child of Outlook node's Rain branch set to No

Child of Temperature node's Hot branch set to Outlook

Example set reduced to those with Temperature set to Mild
Information Gain for Outlook is 0.0
Information Gain for Wind is 0.0
Outlook selected as sub-tree root.
```

```
Outlook node created.
Example set reduced to those with Outlook set to Sunny
Child of Outlook node's Sunny branch set to No

Example set reduced to those with Outlook set to Overcast
Child of Outlook node's Overcast branch set to Yes

Example set reduced to those with Outlook set to Rain
Information Gain for Wind is 0.0
Wind selected as sub-tree root.
Wind node created.
Example set reduced to those with Wind set to Weak
Child of Wind node's Weak branch set to Yes

Example set reduced to those with Wind set to Strong
Child of Wind node's Strong branch set to No

Child of Outlook node's Rain branch set to Wind

Child of Temperature node's Mild branch set to Outlook

Example set reduced to those with Temperature set to Cold
Child of Temperature node's Cold branch set to No

Example set reduced to those with Temperature set to Cool
Child of Temperature node's Cool branch set to No

Child of Humidity node's High branch set to Temperature

Example set reduced to those with Humidity set to Normal
Information Gain for Outlook is -0.37331115029816797
Information Gain for Temperature is -0.40832722141767247
Information Gain for Wind is -0.37331115029816797
Outlook selected as sub-tree root.
Outlook node created.
Example set reduced to those with Outlook set to Sunny
Child of Outlook node's Sunny branch set to Yes

Example set reduced to those with Outlook set to Overcast
Child of Outlook node's Overcast branch set to Yes

Example set reduced to those with Outlook set to Rain
Information Gain for Temperature is -0.08170416594551033
Information Gain for Wind is -0.08170416594551039
Temperature selected as sub-tree root.
Temperature node created.
Example set reduced to those with Temperature set to Hot
Child of Temperature node's Hot branch set to Yes

Example set reduced to those with Temperature set to Mild
Child of Temperature node's Mild branch set to Yes

Example set reduced to those with Temperature set to Cold
Information Gain for Wind is 0.0
Wind selected as sub-tree root.
Wind node created.
Example set reduced to those with Wind set to Weak
Child of Wind node's Weak branch set to Yes
```

```
Example set reduced to those with Wind set to Strong
Child of Wind node's Strong branch set to No

Child of Temperature node's Cold branch set to Wind

Example set reduced to those with Temperature set to Cool
Child of Temperature node's Cool branch set to Yes

Child of Outlook node's Rain branch set to Temperature

Child of Humidity node's Normal branch set to Outlook
```

As may be observed, the design flow of the decision tree is the same as that described earlier, with the attribute at any point in the tree providing the greatest information gain being added as the next child. The final structure of the decision tree computed based upon the training data is depicted through the following if-then construction:

```
if Humidity = High
       if Temperature = Hot
              if Outlook = Sunny
                     NO
              if Outlook = Overcast
                     YES
              if Outlook = Rain
                     NO
       if Temperature = Mild
              if Outlook = Sunny
                     NO
              if Outlook = Overcast
                     YES
              if Outlook = Rain
                     if Wind = Weak
                            YES
                     if Wind = Strong
                            NO
       if Temperature = Cold
              NO
       if Temperature = Cool
              NO
if Humidity = Normal
       if Outlook = Sunny
              YES
       if Outlook = Overcast
              YES
       if Outlook = Rain
              if Temperature = Hot
                     YES
              if Temperature = Mild
                     YES
              if Temperature = Cold
                     if Wind = Weak
                            YES
                     if Wind = Strong
                            NO
              if Temperature = Cool
                     YES
```

As may be observed, the structure of the tree also follows that described above, with each leaf node being set to ‘yes’ or ‘no’, and each path from the root to the leaf nodes relating to a set of conjunctions.

Finally, we observe the predicted results for the three test examples provided. The following is the program output:

```
|                          Example                                    Result

1) Rain             Mild         High         Weak                    true
2) Sunny            Cool         Normal       Strong                  true
3) Rain             Hot          Normal       Weak                    true
```

Looking at the first test example, the attribute values mirror an example present in the training set. Therefore, as expected, the same outcome as was defined in the training set was depicted. This example is clearly trivial and does not test the functionality of the algorithm, but serves as a simple measure of ensuring that all the basic features of the algorithm are working correctly.

Moving on to the next two test examples, we may see that the outcome is ‘true’ for both. Turning towards the if-then structure describing the decision tree, we may see that for both of these cases, the correct prediction is made. For the first of the two, we see that a combination of normal humidity and sunny outlook result in a positive outcome. Similarly, traversing the if-then structure with the final test case, a combination of normal humidity, rainy outlook and hot temperatures result in a positive outcome too.

Conclusion

Following a detailed discussion of the design, development and testing of the ID3 Decision Tree algorithm, we may conclude that the produced implementation meets all of the specified requirements and performs its expected tasks successfully. Throughout the development of this system, no specific problems were encountered, with the approach taken to this problem yielding an effective and successful solution.