# A Machine Learning Algorithm Based on C4.5

## 1.1 A Description of the Algorithm

The algorithm is based on the C4.5 Decision Tree algorithm. The C4.5 is an extension of the ID3 decision tree classifier that also deals with pruning of high-branching attributes to avoid overfitting. The algorithm used in this assignment only deals with classification of data sets with numeric predictor attributes.

The program can handle any data-set with up to nine numeric predictor attributes and one nominal target attribute specifying the classes. There is no specified upper-limit on the number of classes but the program is optimised for two to three. The program has an upper limit of one thousand cases which can be easily increased as well as the number of predictor attributes but any data set upwards of four hundred cases would cause a considerable time increase.  The data-sets this algorithm was tested against include the owls' data set and Illness data set.

The algorithm begins by randomising and splitting the dataset into two thirds for training and one third for testing and then proceeds creating the rules for each node of a decision tree. Rules are created based on the maximum information gain of each attribute, whereby the information gain is calculated at a number of thresholds for each attribute and the maximum is stored. The attribute with the highest information gain over the current "working-set" is chosen and the rule or "node" is created splitting the working set (initially the training cases) into the left-working set (lower than chosen threshold) and right-working set(greater than the chosen threshold). This process is then called on each subsequent working set until a working set is declared a "leaf" – a working set of only one class.

In addition to a node being declared a leaf when it is representative of only one class, a node could also be declared a leaf due to the pruning mechanism implemented. The pruning mechanism was implemented within the "isLeaf" method whereby if a subsequent working set consists of more than one class but consisted of over X% of a particular class it could be flagged as a leaf and no more operations would be carried out on the working set at that node. X is inputted by the user.

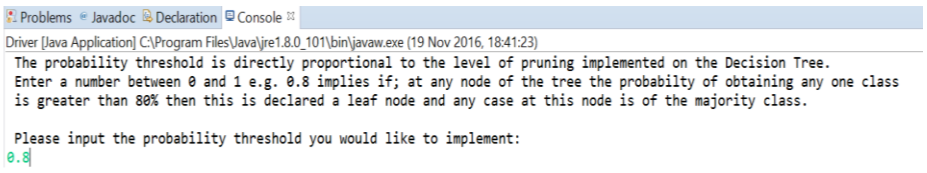
The output of the algorithm is essentially a tree that is stored as a list of rules. On completion of the algorithm (when each working set is declared a leaf) the program immediately tests the test cases (the remaining third of the data-set). It does this by comparing each case to the set of rules until the case reaches a leaf node/rule and then stores the predicted class- the leaf that the case has landed on, and the actual class in an array. When all test cases have been tested the program calls a "Performace" method that compares each predicted and actual pair and creates and prints a confusion matrix based on the results as well as a Classification Accuracy calculation.

## 1.2 Design Decisions

This program is a completely unique implementation of the C4.5 algorithm in the Java programming language and no method or functionality was used from any source other than information on the basic functionality of Quinlan's C4.5 algorithm[1] which yielded the general flow of the program; calculating information gain of attributes, splitting the data accordingly and implementing a pruning technique to avoid over fitting.

The first design decision made was to create a dynamic model of the algorithm that could work for any dataset with any number of numeric attributes, classes and cases up to a reasonable limit and requiring minimal user input. The only user input required is to define the pruning threshold, referred to as the probability threshold for clarity.

[1] Quinlan, R.J. (1992) *C4.5: Programs for machine learning*. 5th edn. San Mateo, CA: Morgan Kaufmann Publishers In.

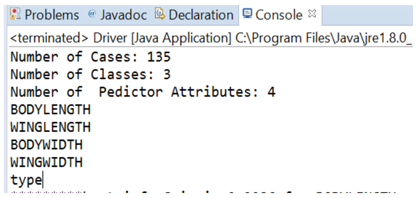


```
Driver [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (19 Nov 2016, 18:41:23)
 The probability threshold is directly proportional to the level of pruning implemented on the Decision Tree.
 Enter a number between 0 and 1 e.g. 0.8 implies if; at any node of the tree the probabilty of obtaining any one class
 is greater than 80% then this is declared a leaf node and any case at this node is of the majority class.

 Please input the probability threshold you would like to implement:
0.8
```

*Figure 1 User Input to Console.*

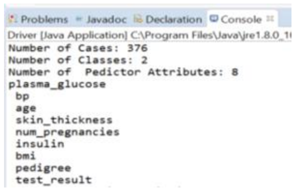
As the tree grows level by level the pruneThreshold is decreased by a small set value (0.025).The reduction of the pruneThreshold at each level of the tree  essentially means that the probability of obtaining a correct class as the tree grows becomes lower, this method therefore highlights further a particularly irrelevant or uncorrelated dataset but doesn't hugely effect the performance on a useful data-set.

The program depends on a particular formatting of the data set before operation. It is required to be in csv format whereby the attributes- both predictor and target are written horizontally on the first line and each of the cases are listed in corresponding attribute order; line by line. The program can then count and store the cases, attribute titles and classes in matrices and array lists for future referencing and not by hardcoded String variables. A quick print to the console confirms the data was received correctly.



```
<terminated> Driver [Java Application] C:\Program Files\Java\jre1.8.0_
Number of Cases: 135
Number of Classes: 3
Number of  Pedictor Attributes: 4
BODYLENGTH
WINGLENGTH
BODYWIDTH
WINGWIDTH
type
```

*Figure 3 Owls15.csv data set.*

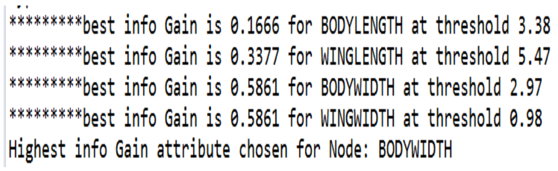


```
Driver [Java Application] C:\Program Files\Java\jre1.8.0_1
Number of Cases: 376
Number of Classes: 2
Number of  Pedictor Attributes: 8
plasma_glucose
 bp
 age
 skin_thickness
 num_pregnancies
 insulin
 bmi
 pedigree
 test_result
```

*Figure 2 Illness Data set.*

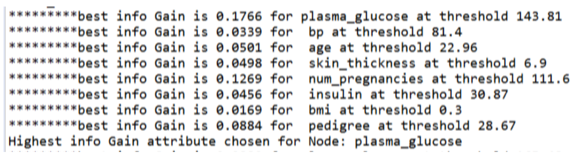
The program then completely randomizes and splits the cases into two thirds training and one third testing String-matrices.

To achieve each attribute's best information gain the information gain is calculated at every possible threshold value between 0 and the maximum value of that attribute among the training data. Starting at the maximum value it decreases the threshold by 1% and recalculates information gain each iteration. After this process is completed for each attribute the overall maximum is chosen as the nodes ruling attribute.



```
*********best info Gain is 0.1666 for BODYLENGTH at threshold 3.38
*********best info Gain is 0.3377 for WINGLENGTH at threshold 5.47
*********best info Gain is 0.5861 for BODYWIDTH at threshold 2.97
*********best info Gain is 0.5861 for WINGWIDTH at threshold 0.98
Highest info Gain attribute chosen for Node: BODYWIDTH
```

*Figure 5 Information Gain at root node: Owls15.csv*



```
*********best info Gain is 0.1766 for plasma_glucose at threshold 143.81
*********best info Gain is 0.0339 for  bp at threshold 81.4
*********best info Gain is 0.0501 for  age at threshold 22.96
*********best info Gain is 0.0498 for  skin_thickness at threshold 6.9
*********best info Gain is 0.1269 for  num_pregnancies at threshold 111.6
*********best info Gain is 0.0456 for  insulin at threshold 30.87
*********best info Gain is 0.0169 for  bmi at threshold 0.3
*********best info Gain is 0.0884 for  pedigree at threshold 28.67
Highest info Gain attribute chosen for Node: plasma_glucose
```

*Figure 4 Information Gain at root node: IllnessDataSet.csv*

In the *Figure 5* above, both body width and wing width have equally the best information gain. In these cases the program simply chooses the first in the list, and so body width was chosen.

It should also be noted that the calculation of information gain required the calculation of entropy in terms of total systemEntropy, entropyt -referring to cases below the threshold value and entropyf - referring to cases above the threshold value.

```
double ent =systemEntropy();
double var1 = ((numCasesLess/wc)*entropyt);
double var2 =((numCasesGr8r/wc)*entropyf);
double IG = (ent-var1-var2);
```

Entropy was calculated as:

```
ent=ent+(numCasesThresh/total)*(logb2(numCasesThresh/total));
```

In the case of entropyT: ‘numCasesThresh’ referred to the number of cases of a certain class below the threshold and ‘total’ referred to the total number of cases below the threshold. Total entropyt was the above formula summed for each class and the returned entropy was divided by the ‘maxEntropy’ value which was calculated based on the number of classes in the data-set. This ensured entropy never exceeded its maximum possible value of 1.

When the attribute with best information gain over the current working set was calculated it was saved as a Node object with the following parameters:

```
Node n = new Node(nodePos,att, thresh, detailsl,detailsr, leftleaf, rightleaf,
LeftworkingSet, RightworkingSet);
```

Its tree position, related attribute, optimal splitting threshold, the printable details of the left branch and right branch, Boolean values of whether the left or right branch leads to a leaf, and the training cases that now remain on the end of the left or right branch as new working sets to be operated upon. As stated above this procedure is iterative until all nodes lead to leaf.

When the algorithm has reached all leaves its output is a list of nodes that make up a set of rules.

```
Rule #1 Highest infoGain attribute: plasma_glucose split at 154.84 This rule has a left leaf: true This rule has a right leaf: false.Position: 1
Rule #2 Highest infoGain attribute: plasma_glucose split at 166.32 This rule has a left leaf: false This rule has a right leaf: true.Position: 3
Rule #3 Highest infoGain attribute:  skin_thickness split at 1.96 This rule has a left leaf: true This rule has a right leaf: false.Position: 6
Rule #4 Highest infoGain attribute:  age split at 17.63 This rule has a left leaf: true This rule has a right leaf: true.Position: 13
```

*Figure 6 Rules associated to the Illness data-set after completion of colassifier Training*

The position indicates where on a binary tree the node would occur. These positions then act as a path for the classification of training data whereby if a case had value below the threshold it’s next position would be its current-position*2, or if it had value above the threshold it’s next position would be its (current-position*2)+1.

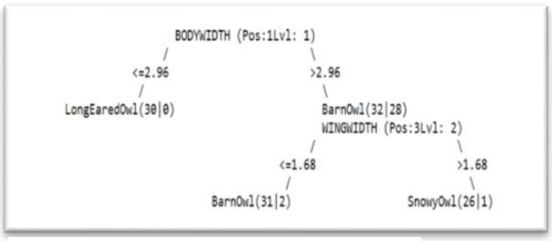
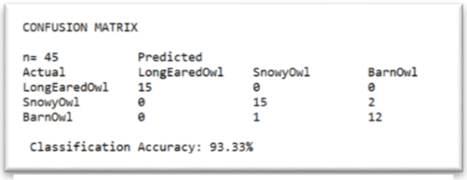


*Figure 7 Example Tree of Owls Data set*

For visualisation of the rules a tree is printed as in *Figure 7*. At the end of each branch the majority class is printed with details indicating: (number of cases of majority class at this node | number not of majority class.

The confusion matrix printed is also a dynamic model for visualisation of performance.



```
CONFUSION MATRIX

n= 45           Predicted
Actual          LongEaredOwl    SnowyOwl        BarnOwl
LongEaredOwl    15              0               0
SnowyOwl        0               15              2
BarnOwl         0               1               12

 Classification Accuracy: 93.33%
```

*Figure 8 Confusion Matrix*

## 1.3 Tests & Results

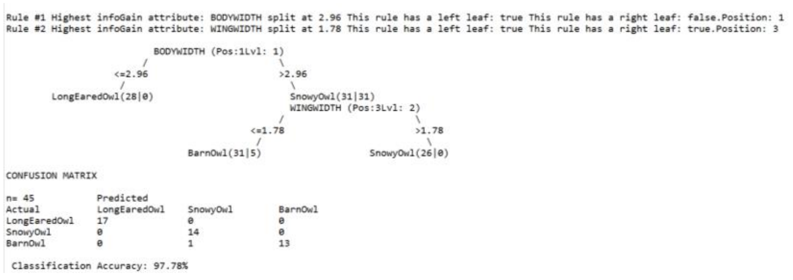
As the program randomises and splits the data automatically, to test the data ten times I simply added a for loop to the driver class and printed the average classification accuracy. I tested my program on both the owls data-set; a total of 135 cases with four numeric predictor attributes and three classes, and on the illness data set; a total of 376 cases with eight numeric predictor attributes and two classes. For each data set the only two variables were the probability/pruning threshold and the path location of the csv file.

The results were as follows for the Owls data-set.

```
The Average Classification Accuracy of ten runs with a probability threshold of 0.8 is 93.33%
```

*Figure 9 The Output to Console After 10 runs*

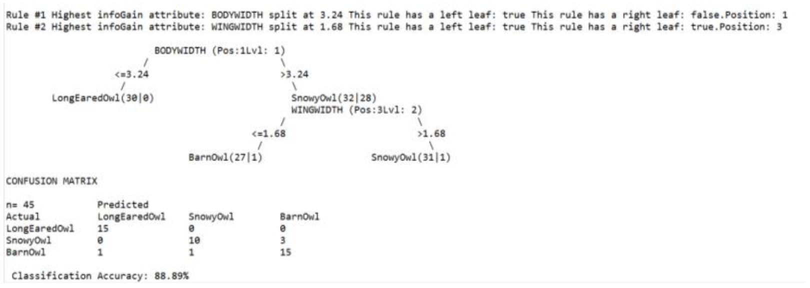
The best run outputted the below details whereby only one instance was incorrectly classified.



```
Rule #1 Highest infoGain attribute: BODYWIDTH split at 2.96 This rule has a left leaf: true This rule has a right leaf: false.Position: 1
Rule #2 Highest infoGain attribute: WINGWIDTH split at 1.78 This rule has a left leaf: true This rule has a right leaf: true.Position: 3

                    BODYWIDTH (Pos:1Lvl: 1)
                   /                     \
              <=2.96                      >2.96
              /                             \
      LongEaredOwl(28|0)                     SnowyOwl(31|31)
                                             WINGWIDTH (Pos:3Lvl: 2)
                                            /                      \
                                       <=1.78                       >1.78
                                       /                              \
                             BarnOwl(31|5)                  SnowyOwl(26|0)

CONFUSION MATRIX

n= 45           Predicted
Actual          LongEaredOwl    SnowyOwl        BarnOwl
LongEaredOwl    17              0               0
SnowyOwl        0               14              0
BarnOwl         0               1               13

 Classification Accuracy: 97.78%
```

*Figure 10 Owls15.csv Best result of ten runs.*

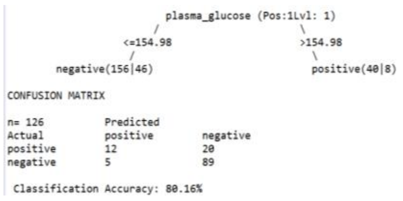
The worst of ten runs with a probability threshold of 0.8 still achieved a classification accuracy of 88.89%.



```
Rule #1 Highest infoGain attribute: BODYWIDTH split at 3.24 This rule has a left leaf: true This rule has a right leaf: false.Position: 1
Rule #2 Highest infoGain attribute: WINGWIDTH split at 1.68 This rule has a left leaf: true This rule has a right leaf: true.Position: 3

                    BODYWIDTH (Pos:1Lvl: 1)
                   /                     \
              <=3.24                      >3.24
              /                             \
      LongEaredOwl(30|0)                     SnowyOwl(32|28)
                                             WINGWIDTH (Pos:3Lvl: 2)
                                            /                      \
                                       <=1.68                       >1.68
                                       /                              \
                             BarnOwl(27|1)                  SnowyOwl(31|1)

CONFUSION MATRIX

n= 45           Predicted
Actual          LongEaredOwl    SnowyOwl        BarnOwl
LongEaredOwl    15              0               0
SnowyOwl        0               10              3
BarnOwl         1               1               15

 Classification Accuracy: 88.89%
```

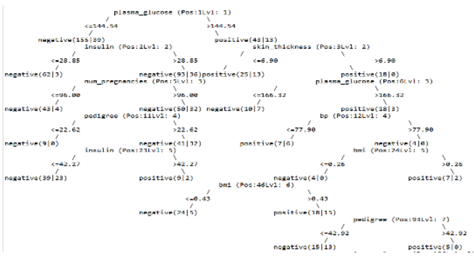
*Figure 11 Owls.csv Worst result of ten runs.*

The program was tested on the illness data set also. The results of this varied greatly, ranging from a set of only one rule to a set of twelve rules whereby an increase in rules did not imply a greater classification accuracy.



```
                    plasma_glucose (Pos:1Lvl: 1)
                   /                          \
             <=154.98                          >154.98
             /                                   \
     negative(156|46)                             positive(40|8)

CONFUSION MATRIX

n= 126          Predicted
Actual          positive        negative
positive        12              20
negative        5               89

 Classification Accuracy: 80.16%
```

*Figure 12 Best case for Illness data set. Pruning: 0.75*



```
                  plasma_glucose (Pos:1Lvl: 1)
                 /                         \
          <=144.54                          >144.54
          /                                   \
  negative(155|39)                             positive(43|13)
          insulin (Pos:2Lvl: 2)                      skin_thickness (Pos:3Lvl: 2)
       /                    \                    /                  \
  <=28.85                    >28.85          <=6.90                   >6.90
  /                            \            /                           \
negative(62|3)              negative(93|36)positive(25|13)           positive(18|0)
          num_pregnancies (Pos:5Lvl: 3)                     plasma_glucose (Pos:6Lvl: 3)
       /                    \                         /                \
  <=96.00                    >96.00              <=166.32               >166.32
  /                            \                /                         \
negative(43|4)              negative(58|32) negative(10|7)             positive(18|3)
          pedigree (Pos:11Lvl: 4)                              bp (Pos:12Lvl: 4)
       /                    \                            /                      \
  <=22.62                    >22.62                 <=77.90                      >77.90
  /                            \                    /                              \
negative(9|0)               negative(41|32)     positive(7|6)                 negative(4|0)
          insulin (Pos:23Lvl: 5)                                 bmi (Pos:24Lvl: 5)
       /                    \                               /                        \
  <=42.27                    >42.27                    <=0.26                         >0.26
  /                            \                       /                                \
negative(39|23)             positive(9|2)        negative(4|0)                    positive(7|2)
                                 bmi (Pos:46Lvl: 6)
                               /                    \
                          <=0.43                     >0.43
                          /                            \
                    negative(24|5)               positive(18|15)
                                                           pedigree (Pos:93Lvl: 7)
                                                      /                       \
                                                 <=42.92                       >42.92
                                                 /                               \
                                           negative(15|13)                 positive(5|0)
```

*Figure 13 Twelve rules with Classification Accuracy of 76.86%*

## 1.4 Conclusions and Observations

It is noted that the pruning mechanism often directly effects the classification accuracy of the algorithm for some data sets, but for a well correlated data set such as the owls data set, the classification accuracy was always higher than the pruning/probability threshold specified.

Increasing the pruning threshold did not necessarily achieve better results in terms of efficiency. If the pruning threshold was inputted as 0.9 for the Illness data set for example, this could result in a tree of over twelve levels with fifteen different rules and only marginally increase the classification accuracy but significantly increase time taken.

Time taken to calculate and choose optimal information gain increased significantly for larger data sets. This was observable when testing both the owls and illness data set.

Often times the success of the algorithm depended on the classifiers “luck” of randomly choosing the training set. Because the owls data set is relatively small and had a training set of ninety cases, one case with a higher than average body-width, for example, could greatly affect the threshold chosen and in-turn the classification accuracy.

## 1.5 Appendices

### 1.5.1 *Driver Class*

```java
package Classifier;

import java.io.IOException;
import java.util.Scanner;


public class Driver {

      /*******
       * Main method simply requires user to input the desired pruning threshold and passes it to
       * the C45 class
       * @param args
       */
      public static void main(String[] args)
      {
                  Scanner in = new Scanner(System.in);
                  System.out.println(" The probability threshold is directly proportional to the level of pruning implemented on the
Decision Tree.");
                  System.out.println(" Enter a number between 0 and 1 e.g. 0.8 implies if; at any node of the tree"
                                          + " the probabilty of obtaining any one class \n is greater than 80% then this is declared a
leaf node and any case at this node "
                                          + "is of the majority class. \n");
                  System.out.println(" Please input the probability threshold you would like to implement: ");
                  double pt = Double.parseDouble(in.nextLine());

                  String csv = "C:\\Users\\user\\Desktop\\book3.csv";

                  for(int i =0; i<10; i++){                                                                           //
for loop to run the program 10 times, not neccessary
                              try {
                                          C45 Testing = new C45(csv,pt);
                              } catch (IOException e) {
                                          e.printStackTrace();}
                              }


                                          //simple print out after running test 10 times
                  System.out.println("The Average Classification Accuracy of ten runs with a probability threshold of "
                                                                        + pt+ " is "+ String.format( "%.2f",C45.overallAccuracy/10)
+"%");
      }

}
```

### 1.5.2 *Node Class*

```java
package Classifier;
/************************************************
 *
 * Constructor class for the creation of Node/Rule objects
 *
 */
public class Node {

      private int attribute;
      private double threshold;
      private boolean leftleaf;
      private boolean rightleaf;
      private double infoGain;
      private String[][] LWS = new String[10][1000];
      private String[][] RWS = new String[10][1000];
      private String detailsL;
      private String detailsR;

      private int pos;
      private int lvl;
      /*****************
       * Constructor for temporary rules yet to be compared to other rules for best information gain
       *
       * @param att, attribute involved in the rule
       * @param thresh, the threshold is a numeric value that acheives the best information gain when splitting the dataset
       * @param info, the ionformation gain acheived as result of the thresh value
       */
      public Node(int att, double thresh, double info)
      {
                  this.attribute = att;
                  this.threshold = thresh;
                  this.infoGain = info;
      }

      /*************
       * Constructor for permanent rules/nodes that will be added to a list and used for test cases
       *
       * @param i, Node position on a binary tree
       * @param att, attribute associated with the rule
       * @param thresh, the threshold chosen that acheived best information gain
       * @param detailsl, the details to be printed at the end of left branch
       * @param detailsr, the details to be printed at the end of right branch
       * @param leftLeaf, boolean value of whether the left branch acheives a leaf
       * @param rightLeaf, boolean value of whether the left branch acheives a leaf
       * @param LeftworkingSet, a string array of all training cases with values less than/equal to specified threshold
       * @param RightworkingSet, a string array of all training cases with values greater than specified threshold
       */
      public Node(int i,int att, double thresh, String detailsl,String detailsr, boolean leftLeaf, boolean rightLeaf,String[][] LeftworkingSet,String[][]
RightworkingSet)
      {
                  this.attribute = att;
                  this.threshold = thresh;
                  this.detailsL = detailsl;
                  this.detailsR = detailsr;
                  this.leftleaf = leftLeaf;
                  this.rightleaf = rightLeaf;
                  this.LWS = LeftworkingSet;
                  this.RWS = RightworkingSet;
```

```
			this.pos =i;
			this.lvl=calcLvl();

	}

	/******************
	 *  Calculates the level the node on a standard binary tree depending on the position
	 *  (Primitive way of calculating this, I'm aware)
	 * @return integer, tree level
	 */
	public int calcLvl()
	{
			if(pos==1){lvl =1;}
			else if(pos<=3){lvl =2;}
			else if(pos<=7){lvl =3;}
			else if(pos<=15){lvl=4;}
			else if(pos<=31){lvl =5;}
			else if(pos<=63){lvl =6;}
			else if(pos<=127){lvl=7;}
			else if(pos<=255){lvl=8;}
			else {lvl=0;}
			return lvl;

	}

	/****************
	 *  To print a singular node for testing purposes, this will not take the parent node position into account
	 * @param tab, specifies how centered to the console the details will print
	 */
	public void printNode(String tab)
	{

			System.out.println(tab+"\t\t  " + C45.attributes.get(this.getAttribute())+ " (Pos:" +this.getPos()+ "Lvl: "+this.getLvl() + ")");
			System.out.println(tab+"\t/\t\t        \\");
			System.out.println(tab+"  \t   <=" + String.format( "%.2f",this.getThreshold()) +"          \t\t>"       + String.format(
	"%.2f",this.getThreshold()));
			System.out.println(tab+"    \t    /                 \t\t  \\");
			System.out.println(tab+this.getDetailsL() + "  \t\t\t  " + this.getDetailsR());

	}
    *  All general getters and setters
	 * @return
	 */
	public String Treepos()
	{
			String position = ".Position: " + pos;
			return position;
	}
	public int getPos()
	{
			return pos;
	}
	public int getLvl()
	{
			return lvl;
	}
	public String getDetailsL() {
			return detailsL;
	}
	public boolean isLeftleaf() {
			return leftleaf;
	}
	public boolean isRightleaf() {
			return rightleaf;
	}
	public String[][] getLWS() {
			return LWS;
	}
	public String[][] getRWS() {
			return RWS;
	}
	public void setLeftleaf(boolean leftleaf) {
			this.leftleaf = leftleaf;
	}
	public void setRightleaf(boolean rightleaf) {
			this.rightleaf = rightleaf;
	}
	public String getDetailsR() {
			return detailsR;
	}
	public int getAttribute() {
			return attribute;
	}
	public double getThreshold() {
			return threshold;
	}
	public double getInfoGain() {
			return infoGain;
	}
    }
```

### *1.5.3 C45 Class*

```
package Classifier;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;


//this implementation of C4.5 is dynamically modelled to work with
//any data set in a csv format with up to 9 numeric predictor attributes, 1000 cases
// and one target attribute
public class C45 {
	/*******
	 *  variables used for parsing and storing data set
	 */
	private String csv;
	private String delim = ",";
	private String headerRow;
	private int numatt;
	private int linecount=0;
	private int numCases;
	// list used to shuffle/randomize the inputted data
	private ArrayList<Integer> ListShuffle = new ArrayList<Integer>();
	// arrays for storing cases
	private String[][] csvParse = new String[10][1000];
	private String[][] Training;
	private String[][] Testing;

	private ArrayList<String> classes = new ArrayList<String>();
	private int numClasses;
	public static ArrayList<String> attributes = new ArrayList<String>();
	/*******
	 *  Variables used for training set manipulation and operations
	 */
	private String[][] workingSet;			// continuously cleared and updated at each iteration
	private String[][] LeftworkingSet;		// continuously cleared and updated at each iteration
	private String[][] RightworkingSet;		// continuously cleared and updated at each iteration
	private int pointerR=0;
	private int pointerL=0;
	private boolean nodeFlag =false;		// flag set when it's neccesary to gather data for Node creation
	private int workingCases=0;						// count variables
	private int LeftworkingCases=0;
	private int RightworkingCases=0;
	private int trainingCases;
	private int testingCases;

	private double maxEnt;							// this value is calculated depending on the
number of classes in data set

	private boolean t_flag=false;			// threshold variables relating to numeric attribute values
	double tolerance = 0.0;
	private double tempthresh =0.0;

	/*******
	 * Variables used for creation and storing of nodes
	 */
	private ArrayList<Node> tempNodes= new ArrayList<Node>();
	private ArrayList<Node> nodes= new ArrayList<Node>();
	private boolean leftleaf = false;
	private boolean rightleaf =false;
	private int nodePos=1;

	private double pruningThresh = 0.8;	//level of desired probability of a certain class at a leaf node, user input
	private double pruneDec =0.025;

	private String testCompare[][];			// array used to store predicted and actual classes for test cases

	public static double overallAccuracy=0.0;

	/*****
	 *  Constructor for the creation of a C4.5 based Algorithm
	 * @param csvLocation
	 * @param pt, pruning threshold inputted by user
	 * @throws IOException
	 */
	public C45(String csvLocation, double pt) throws IOException
	{
		this.pruningThresh =pt;
		this.csv = csvLocation;

		DataIn();									// read data in form csv
file, stores cases/ classes/ attributes
		DataSplit();								// split and randomize data
accordingly
		createNode();								// creates the initial root
node
		travelTree();								// iterative method
begining at the branches produced by the root node

		System.out.print("\n");
		for(int i =0; i< nodes.size(); i++)			// prints the set of rules calculated by the classifier
		{
			System.out.println("Rule #" + (i+1) + " Highest infoGain attribute: " +
attributes.get(nodes.get(i).getAttribute())
			+ " split at " +round(nodes.get(i).getThreshold())
			+ " This rule has a left leaf: " + nodes.get(i).isLeftleaf()
			+ " This rule has a right leaf: " + nodes.get(i).isRightleaf()
			+ nodes.get(i).Treepos());
		}
		System.out.print("\n");

		createTree(nodes);						// prints the final tree to console
		TestData();									// Tests the
test cases on the classifier
		Performance();								// evaluates and prints
performance
	}

	/*******
	 * Parses the csv file
	 * Counts and stores all attributes, cases classes etc
	 * @throws IOException
	 */
	private void DataIn() throws IOException
	{
```

```
                        //count the number of cases and attributes in total
                BufferedReader Countreader =
                                                new BufferedReader(new FileReader(csv)); // read file
                String temp;

                while((temp = Countreader.readLine()) != null)                          // number of lines= number of cases
                {
                        char[] countatt = temp.toCharArray();                                   // create character array
of first line
                        char c;
                        if(linecount ==0)
                        {
                                headerRow=temp;
                        for(int i =0; i<countatt.length; i++)
                        {
                                c=countatt[i];
                                if(c == ',')
                // count number of commas in line1 .csv
                                {
                                        numatt++;
                // let this be num attributes
                                }
                        }
                        }
                        linecount++;
                // count number of lines= num cases
                }
                numCases = linecount-1;
                Countreader.close();

                System.out.println("Number of Cases: " + numCases);

                //read each line/case into the string array
                BufferedReader reader =                                                                 //
new reader
                                new BufferedReader(new FileReader(csv));
                String addme = null;
                int j=0;
                while( (addme = reader.readLine()) != null)                             // read each line
                {
                        String[] splitline = addme.split(delim);                        //      read to each comma
                        for (int i =0; i<numatt+1;i++)                                          // for every
attribute +1(target)
                        {
                                csvParse[i][j] = splitline[i];                                  // add each
value to a table matrix
                        }
                  j++;
                }
                reader.close();

                // gather and count the classes
                for(int targetcol=1; targetcol<numCases; targetcol++)  // classes listed in last column line 2
                {
                        String target = csvParse[numatt][targetcol];
                        if(!classes.contains(target))                                           // if it hasn't
already been added
                                        {
                                                classes.add(target);                            //
add to list
                                        }
                }
                numClasses = classes.size();
                System.out.println("Number of Classes: " + numClasses); // print

                // gather and count attributes
                System.out.println("Number of  Pedictor Attributes: " +numatt);   // number of atts already counted
                String[] headings = headerRow.split(delim);                             // collect from header row(0)
                for(int i=0; i< numatt+1; i++)
                {
                        attributes.add(headings[i]);                                            // add and print
each heading
                        System.out.println(attributes.get(i));
                }
        }

        /********
         * Randomly splits the data into training and testing data
         */
        public void DataSplit()
        {
                trainingCases = (numCases/3)*2;                                         // training cases is two thirds
                testingCases = numCases-trainingCases;                          // remaining is test

                Training = new String[numatt+1][trainingCases];  // create new table matrices
                Testing = new String[numatt+1][testingCases];


                // this is to randomly add from csvParse to Training and Testing matrices
                for(int shuf=1; shuf <numCases+1; shuf++)                       // begin at 1 to discount the header row added to
csvParse
                {
                        ListShuffle.add(shuf);                                                  // list of
integers 0 to total number cases
                }

                Collections.shuffle(ListShuffle);                                       // shuffle these ints
```

```
		// randomly assign cases as training data
		for(int split = 0; split < trainingCases; split++)
		{
			for(int i = 0; i<numatt+1; i++)
			{
				Training[i][split] = csvParse[i][ListShuffle.get(split)];
			}
		}

		workingSet = Training;
		//working set changes at each node
		workingCases = trainingCases;

		// randomly assign cases to testing data
		for(int split = trainingCases; split <numCases; split++)		//+1 header row
		{
			for(int i = 0; i<numatt+1; i++)
			{
				Testing[i][split-trainingCases] = csvParse[i][ListShuffle.get(split)];
			}
		}
		testCompare = new String[2][testingCases];
		}

	/*********
	 * The main method iterated upon to determine the rules.
	 * runs through optimization method to find best information gain for each attribute
	 * saves the best info gain for each attribute
	 * and chooses best overall determines the cases
	 */
	public void createNode()
	{
		LeftworkingSet=new String[10][1000];
		RightworkingSet =new String[10][1000];
		double bestinfo =0.0;
		double tempinfo =0.0;
		int index=0;
		double thresh =0.0;
		String detailsl=null;
		String detailsr=null;

			for(int i =0; i<numatt; i++)
			{
				Optimizer(i);		// for each attribute chooses new threshold and recalculated 
information gain until max is found.
			}
			for(int i =0; i<tempNodes.size(); i++)		// the highest info gain of each attribute is stored as a 
temp Node
			{
				tempinfo =tempNodes.get(i).getInfoGain();

				System.out.println("*********best info Gain is " + String.format( "%.4f",tempinfo)
						+ " for " + attributes.get(i)+" at threshold " + 
round(tempNodes.get(i).getThreshold()));

				if(tempinfo> bestinfo)		// the attribute with the best info gain is chosen
				{
					index =i;
					thresh = tempNodes.get(index).getThreshold();
					bestinfo = tempinfo;
				}
			}

			System.out.println("Highest info Gain attribute chosen for Node: " + attributes.get(index));

			RightworkingCases =0; // its permanent Node constructor parameters are generated
			LeftworkingCases =0;

			for(int i=0; i< numClasses; i++)		// for each class
			{
				nodeFlag=true;								//assign each 
case to either left or right working set
				LeftworkingCases = LeftworkingCases+countBelowthreshold(i, index, round(thresh));

				RightworkingCases = RightworkingCases+countAbovethreshold(i, index, round(thresh));

			}

			detailsl = checkClasses(LeftworkingSet, LeftworkingCases);			// generate the details to 
be printed at the end of each branch
			detailsr = checkClasses(RightworkingSet, RightworkingCases);	// checkClasses method
			leftleaf = isLeaf(LeftworkingSet, LeftworkingCases);				// determine whether either 
branch of node is a leaf or not
			rightleaf = isLeaf(RightworkingSet, RightworkingCases);

								// create the permanent node
			Node n = new Node(nodePos,index, thresh, detailsl,detailsr, leftleaf, rightleaf, LeftworkingSet, 
RightworkingSet);
			if(tempNodes.get(index).getInfoGain() != 0.0)						// 
ensure info gain is not 0 (occurs if working set is 0)
			{
				nodes.add(n);
					// add to list of nodes/rules
			}

			setClear(workingSet);
			// clear the working set

	}
```

```
    /********
     * Iteratively calls the createNode method on each branch of a node
     *
     */
    public void travelTree()
    {
        int prevlevel=0;
        for(int i=0; i< nodes.size(); i++)          // nodes.size keeps growing as each time this method calls createNode()
        {
            Node n = nodes.get(i);
            int level = n.getLvl();
            if(level>prevlevel)
            {
                pruningThresh-=pruneDec;        // as the tree level grows, algorithm becomes more likely to
declare a leaf node
                prevlevel=level;
            }
            if(!n.isLeftleaf()){
                nodePos= n.getPos()*2;                      // next node position calculated based on parent
node position
                workingSet= n.getLWS();                     // working set is now nodes left working set
                workingCases = setSize(workingSet);
                pointerR=0;                                                     // all variables set to
defaults
                pointerL=0;
                nodeFlag=false;
                tempNodes.clear();
                if(workingCases!=0){            // as long as working cases exist
                    createNode();                               // create the next node
                    }
                }
            if(!n.isRightleaf()){
                nodePos= (n.getPos()*2)+1;
                workingSet= n.getRWS();
                workingCases = setSize(workingSet);
                pointerR=0;
                pointerL=0;
                nodeFlag=false;
                tempNodes.clear();
                if(workingCases!=0){
                    createNode();
                }
                }
        }
    }

    public void createTree(ArrayList<Node> nodes)
    {
        int levels =0;
        String centre = "\t\t\t\t";
        String tab = "\t";
        String tabs = "";
        String offset="";
        int parentPos =0;

        levels = nodes.get(nodes.size()-1).getLvl();        // get the total amount of levels
        if(nodes.get(0).getLvl()==1)
        {
            nodes.get(0).printNode(tab);                                // root node printed
            parentPos = nodes.get(0).getPos();
        }
            for(int j=2; j<=levels; j++)                                // for each level of the tree
            {
                tempNodes.clear();                                              // re-use of
tempNodes List
                tabs ="";
                for(int i =1; i<nodes.size(); i++)                      // go through each node
                {
                    int level=nodes.get(i).getLvl();
                    if(level==j)                                                        // if
the node belongs to this level
                    {
                        tempNodes.add(nodes.get(i));    // add it to list of nodes at this
level
                    }
                }
                for(int l = j; l>2 ;l--)                                        // closer to level 1 the
more tabs/spaced out
                {
                    tabs +=tab;
                }
                if(tempNodes.size()==1)                                         // if there is
only one node at this level
                {
                    if(tempNodes.get(0).getPos()> parentPos*2)  // and the node is a right branch
                    {
                        offset+=centre;                                                 //
offset the print of this node to the right
                    }
                    parentPos=tempNodes.get(0).getPos(); // update parent position
                }
                String att="";                                                          //
variables used to concatenate multiple nodes at same level
                String branch1 ="";
                String threshs ="";
                String branch2 ="";
                String deets ="";
                for(int k=0; k<tempNodes.size(); k++)       // concatenate node details
```

```
                                            {
                                                        att += "\t\t  " +attributes.get(tempNodes.get(k).getAttribute())+ " (Pos:" 
+tempNodes.get(k).getPos()+ "Lvl: "+tempNodes.get(k).getLvl() + ")" + tabs;
                                                        branch1+="\t         /\t\t      \\"+tabs;
                                                        threshs+="  \t   <=" + String.format( "%.2f",tempNodes.get(k).getThreshold()) +" 
           \t\t>"     + String.format( "%.2f",tempNodes.get(k).getThreshold())+tabs;
                                                        branch2 +="    \t    /                      \t\t  \\"+tabs;
                                                        deets +=tempNodes.get(k).getDetailsL() + "\t\t\t" + 
tempNodes.get(k).getDetailsR()+tabs;
                                            }

                      // print the nodes to console
                                            System.out.println(offset+att+"\n" +offset+ branch1+ "\n"+offset+ threshs +"\n"+ offset+ 
branch2 + "\n"+offset+ deets);
                      }
           }


           /*****
            * When classifier has constructed the decision tree model
            * This method tests each case of the Test cases with the rules that were constructed
            * It then stores the actual class of the test case and the predicted class in an array as a pair
            */
           public void TestData()
           {
                      int att=0;
                      double thresh =0.0;
                      int j=0;
                      int nextr=0;
                                            for(int i=0; i< testingCases; i++)
                                 // for each of the test cases
                                            {
                                                        Node rule = nodes.get(j);
                                            // get the current rule (initially root)
                                                        while(rule!=null)
                                                        {
                                                                    att = rule.getAttribute();
                                            // get the relevant attribute
                                                                    thresh = rule.getThreshold();
                                 // get the threshold
                                                                    if(Double.parseDouble(Testing[att][i])<=thresh)
           // if it is left of this threshold
                                                                    {
                                                                               if(rule.isLeftleaf())
                                            // and if the left of the node is a leaf
                                                                               {
                                                                                          testCompare[0][i]=rule.getDetailsL();
                      //extract the left details of the rule
                                                                                          testCompare[1][i]=Testing[numatt][i];
                      // store the predicted and actual as a pair
                                                                               }
                                                                               nextr=rule.getPos()*2;
                                                        // if it is not a leaf, the left means next position is current*2
                                                                               rule =searchRules(nextr);
                                            // call searchRules method for the node at this position

                                                                    }
                                                                    else if(Double.parseDouble(Testing[att][i])>thresh)              // 
same functionality for right branch
                                                                    {
                                                                               if(rule.isRightleaf())
                                                                               {
                                                                                          testCompare[0][i]=rule.getDetailsR();
                                                                                          testCompare[1][i]=Testing[numatt][i];
                                                                               }
                                                                               nextr=(rule.getPos()*2)+1;
                                            // if the right branch is not a leaf;next position is current*2+1
                                                                               rule = searchRules(nextr);
                                            // search the rules for this positon
                                                                    }
                                                                    else
                                                                    {
                                                                               System.out.println("Testing issue");
                      // default to signify error
                                                                    }
                                                        }
                                            }
           }
           /*******
            * Called by the TestData method to search for a next rule
            * @param p, the position of the next rule it is searching for
            * @return The node at this position
            */
           private Node searchRules(int p)
           {
                      for(int i=0; i< nodes.size(); i++)
                      {
                                 if(p==nodes.get(i).getPos())
                                 {
                                            return nodes.get(i);
                                 }
                      }
                      return null;
           }

           /*******
            * After the data has been tested and
            * a testCompare array with predicted and actual values has been created
            * This method creates a confusion matrix and calculates classification accuracy
```

```
	 */
	private void Performance()
	{
		String[][] confusionMatrix = new String[numClasses+1][numClasses+1]; // matrix is dependant on number of classes
		int col=0;
		int row=0;
		int truepos=0;
		int[][] matrix = new int[numClasses][numClasses];							// matrix for 
arithmetic purposes
		confusionMatrix[0][0] ="Actual";
		
			for(int a=1; a<numClasses+1; a++)									
		//add headings horizontally
			{
				confusionMatrix[0][a]= classes.get(a-1);
			}
			
			
			for(int b=1; b<numClasses+1; b++)									
		// add headings vertically
			{
				confusionMatrix[b][0]= classes.get(b-1);
			}
			
		
		for(int i =0; i< testingCases; i++)									
		//remove unwanted part of details string
		{
			String[] s=testCompare[0][i].split("\\(");
			testCompare[0][i] = s[0];
		}
		
		for(int j=0; j<testingCases; j++)									
		//compare predicted and actual for each test case
		{
			String predicted = testCompare[0][j];
			String actual = testCompare[1][j];
			
			for(int x=0; x<numClasses; x++)								
		// to find which matrix position, iterate through classes
			{
							// if predicted matches a class
				if(predicted.equals(classes.get(x)))				
	// predicted is across the columns
				{
					col =x;
						// added to this column
				}
							// if actual matches a class
				if(actual.equals(classes.get(x)))					
		// actual is down the rows
				{
							// added to this row
					row =x;
				}
			}
			if(col==row)
					// if column and row are the same
			{
				truepos++;
					// this case is a true positive: classified correctly
			}
			matrix[row][col] +=1;
			//increment the integer matrix at this position
		}
		
		for(int r=0; r< numClasses; r++)
		{
			for(int c =0; c< numClasses; c++)
			{
				int sum = matrix[r][c];
				confusionMatrix[r+1][c+1] = Integer.toString(sum)+"\t";			// format the 
confustion mattrix
			}
		}
		System.out.print("\n");
		System.out.println("CONFUSION MATRIX \n");
		System.out.print("n= "+testingCases+ "\t\tPredicted \n");
		for(int r=0; r< numClasses+1; r++)
		{
			for(int c =0; c< numClasses+1; c++)
			{
				System.out.print(confusionMatrix[r][c]+ "  \t");				// print the 
confusion matrix
			}
			System.out.print("\n");
		}
		double ca = ((double)truepos/testingCases)*100;								// 
classification accuracy calculates
		overallAccuracy+=ca;
			// if average of ten runs is enabled
		System.out.println("\n Classification Accuracy: " + round(ca) + "%");// print the classification accuracy
	}
	
	
	/******
	 * Finds the maximum information gain for an attribute by repeatedly calling the threshold method
	 * @param att, the attribute to calculate information gain for
	 */
	private void Optimizer(int att)
	{
		t_flag = false;
		double entropyt = 0.0;
		double entropyf =0.0;
		double tempinfo=0.0;
		double bestinfo =0.0;
		double bestthresh =0.0;
		double t = 0.0;
		int timeout=0;
			while(timeout<200)
			{
				t= threshold(att);
	// calculate the next threshold
				entropyf = entropyRight(t,att);							//
calculate the entropy of cases that would be at the right branch given the new threshold
				entropyt = entropyLeft(t, att);							//
calculate the entropy of cases that would be at the left branch given the new threshold
				bestinfo =infoGain(att, t, entropyt, entropyf); // calculate the information gain
				if(bestinfo> tempinfo)
				{
					tempinfo= bestinfo;
	// store the highest information gain and its relevant threshold
					bestthresh = t;
				}
				timeout++;
			}
			Node n = new Node(att,bestthresh, tempinfo);			// create a tempNode based on the
highest info gain of this attribute
			tempNodes.add(n);
	// add to temp nodes

	}

	/***************
	 * This method keeps decreasing the threshold value so the threshold with the best info gain can be chosen
	 * @param att, for a given attribute
	 * @return the next threshold
	 */
	private double threshold(int att)
	{
		double initmax = 0.0;
		double threshmax=0.0;
		double threshold=0.0;

		if(!t_flag)									// this only code only runs once for
each attribute
		{
		for(int i=0; i<numClasses; i++)
		{
			threshold = max(i,att);			// the first threshold is the maximum value of an attribute
present among the set
			if( threshold>initmax)
			{
				threshmax = threshold;		// find absolute max value for given attribute
				initmax=threshold;
			}

		}
			tempthresh= threshmax;
			tolerance = threshmax/100;	// reduction of 0.01max each iteration
		}
		t_flag = true;
											// keep reducing threshold
		tempthresh = tempthresh - tolerance;
		return tempthresh;
	}

	// calculates the max value of each
	// attribute for a given class
	/*******
	 *
	 * @param clas, to find the maximum value among the specified class
	 * @param att, the attribute
	 * @return The maximum value of a specific class of an attribute present among the working set
	 */
	private double max(int clas, int att)
	{
		double initmax = 0.0;
		double tempmax =0.0;
		double max = 0.0;
			for(int i=0; i<workingCases; i++)
			{
			  if(workingSet[numatt][i].equals(classes.get(clas)))
			  {
				if((tempmax = Double.parseDouble(workingSet[att][i]))> initmax)
				{
					max = tempmax;
					initmax = max;
				}
			  }
			}
			return max;
	}

	/*****
	 * Calculation of information gain
	 * @param att, the attribute this is calculated for
	 * @param threshold, the relevant threshold
	 * @param entropyt, entropy to the left of the node
	 * @param entropyf, entropy to the right of the node
	 * @return double value information gain of specified attribute
	 */
	private double infoGain(int att, double threshold, double entropyt, double entropyf)
	{
		double numCasesLess=0.0;
		double numCasesG8r =0.0;
		double wc = (double) workingCases;
		for( int clas =0; clas<numClasses; clas++)
		{
			numCasesLess += (double)countBelowthreshold(clas, att, threshold);
			numCasesG8r += (double)countAbovethreshold(clas, att, threshold);
		}
		double ent =systemEntropy();
		double var1 = ((numCasesLess/wc)*entropyt);
		double var2 =((numCasesG8r/wc)*entropyf);
		double IG = (ent-var1-var2);
		return IG;
	}

	/*******
	 * At a certain node calculate the total entropy of the remaining cases
	 * @return Ent(S), the system entropy of the current working set.
	 */
	private double systemEntropy()
	{
		double ent=0.0;
		for(int i =0; i<numClasses;i++)
	// for each class
		{
			double x = count(i, workingSet, workingCases);					// count the
number of each class in the working set
			if(x!=0)
			{
				ent = ent+ (-x/workingCases)*(logb2(x/workingCases));  // standard entropy formula
			}
		}
		maxEnt = -logb2(((double)workingCases/numClasses)/workingCases);// maximum entropy calculated depending on the
number of classes
		ent = ent/maxEnt;
		return ent;
	}

	/********
	 * Calculation of entropy of cases that would be less than the threshold
	 * @param threshold, the threshold to split the data
	 * @param att, the attribute the entropy is calculated for
	 * @return, the entropy
	 * For the  owls data set this formula would look like:
	 * -(LEbelowThresh/totalBelowThresh)*logb2(LEbelowThresh/totalBelowThresh)
	 * -(BObelowTresh/totalBelowThresh)*logb2(BObelowThresh/totalBelowThresh)
	 * -(SObelowThresh/totalBelowThresh)*logb2(SObelowThresh/totalBelowThresh)-
	 */
	private double entropyLeft(double threshold, int att)
	{
		double ent =0.0;
		int i = att;
			for(int j=0; j<numClasses; j++)
			{
			  double total =0.0;
				for(int tot=0; tot<numClasses; tot++)
				{
					total+= countBelowthreshold(tot,i,threshold); //all instances below threshold
				}
				int numCasesThresh = countBelowthreshold(j,i,threshold);
				if(numCasesThresh!= 0)
				{
					ent = ent+ (-numCasesThresh/total)*(logb2(numCasesThresh/total));
				}

			}
				ent = ent/maxEnt;
				return ent;
	}

	/*************
	 * This is the inverse of entropyLeft
	 * @param threshold
	 * @param att
	 * @return
	 */
	private double entropyRight(double threshold, int att)
	{
		double ent =0.0;
		int i = att;
			for(int j=0; j<numClasses; j++)
			{
			  double total =0.0;
				for(int tot=0; tot<numClasses; tot++)
				{
					total+= countAbovethreshold(tot,i,threshold);
				}
				int numCasesThresh = countAbovethreshold(j,i,threshold);
				if(numCasesThresh> 0)
				{
					ent = ent+ (-numCasesThresh/total)*(logb2(numCasesThresh/total));
				}
			}
```

```
                                    ent = ent/maxEnt;
                                    return ent;
            }

            /*****
             *
             * @param clas, the class to count
             * @param set, the set of cases to be counted
             * @param length, the total number of cases in this set
             * @return the numeber of cases of a certain class within a set
             */
            private int count(int clas, String[][] set, int length)
            {
                        int counter =0;
                        //numatt by this programs formatting requirements should always refer to the position of the target class in a set
                        int att = numatt;
                                    for(int i = 0; i< length; i++)
                                    {
                                                if(!(set[att][i].equals(null))){
                                                            if(  set[att][i].equals(classes.get(clas))){
                                                                        counter++;
                                                            }
                                                }
                                    }
                        return counter;
            }

            /********
             * given a threshold (val) count the number of cases of a certain class that is less than/equal to the threshold
             * @param clas, the class to be counted
             * @param att, the relevant attribute
             * @param val, the value of the threshold
             * @return the number of cases of a certain class BELOW the threshold
             */
            private int countBelowthreshold(int clas, int att, double val)
            {
                        int counter =0;
                                    for(int i = 0; i< workingCases; i++)
                                    {
                                                if (workingSet[numatt][i].equals(classes.get(clas)))                                    //if 
case is of specified class
                                                {
                                                            if(round(Double.parseDouble(workingSet[att][i])) <= round(val))   //and below or 
equal to threshold
                                                            {
                                                                        counter++;
                                                                                    // count it
                                                                        if(nodeFlag)                        // this flag is set when the cases 
must be collected for left working sets of node
                                                                        {
                                                                                    for(int a =0; a< numatt+1; a++)
                                                                                    {
                                                                                                LeftworkingSet[a][pointerL] = workingSet[a][i];
            //assign these cases to the left working set
                                                                                    }
                                                                                    pointerL++;
                                                                        }
                                                            }
                                                }
                                    }
                        return counter;
            }

            /*******
             *  "Greater than" equivalent method of countBelow threshold
             * @param clas, the class to be counted
             * @param att, the relevant attribute
             * @param val, the threshold value
             * @return the number of cases above the specified threshold
             */
            private int countAbovethreshold(int clas, int att, double val)
            {
                        int counter =0;
                                    for(int i = 0; i< workingCases; i++)
                                    {
                                                if (workingSet[numatt][i].equals(classes.get(clas)))                                   // if 
case if of specified class
                                                {
                                                            if(round(Double.parseDouble(workingSet[att][i])) > round(val))    // and greater 
than threshold
                                                            {
                                                                        counter++;
                                                                                    // count it
                                                                        if(nodeFlag)          // this flag is set when the cases must be 
collected for left working sets of node
                                                                        {
                                                                                    for(int a =0; a< numatt+1; a++)
                                                                                    {
                                                                                                RightworkingSet[a][pointerR] = workingSet[a][i];
            // assign these cases to the right working set
                                                                                    }
                                                                                    pointerR++;
                                                                        }

                                                            }
                                                }
                                    }
                        return counter;
            }
```

```
	/*******
	 * 
	 * @param set, the set to be analysed
	 * @param i, the length of the set
	 * @return a String of the details of the majority class residing a branch
	 *  String details, is of the form: "MajorityClass(number of cases of the majority class|number of cases not of the majority class)
	 */
	private String checkClasses(String[][] set, int i)
	{
		ArrayList<String> clas = new ArrayList<String>();
		Set<String> hs = new HashSet<>();		// hashset removes dublicates
		String mainClas = null;
		int main=0;
		int max =0;
		String details="";
		for(int j =0; j<i; j++)
		{
				clas.add(set[numatt][j]);	// collect all classes present in a set
		}
		hs.addAll(clas);					// remove duplicates
		clas.clear();							// clear the original list
		clas.addAll(hs);					// re-add to class list
		for(int k =0; k< clas.size(); k++)		// iterate through the list to find the majrity class
		{
			main =count(classes.indexOf(clas.get(k)), set,i);
			if(main>max)						// store the max
			{
				mainClas =(String) clas.get(k);
				max=main;
			}
		}
		details = mainClas +"(" + max +"|" + (setSize(set)-max) + ")";
		return details;
	}

	/************
	 * 
	 * @param set, the set to be analyzed
	 * @param i, set length
	 * @return true/false if a set is a leaf depending on number of classes and pruning threshold
	 */
	private boolean isLeaf(String[][] set, int i)
	{
		ArrayList<String> clas = new ArrayList<String>();
		Set<String> hs = new HashSet<>();
		double prune=1;
		double num=0.0;
		double maxnum=0.0;
		for(int j =0; j<i; j++)
		{
				clas.add(set[numatt][j]);

		}
		hs.addAll(clas);				//remove duplicates by using hashset
		clas.clear();					// clear the class list
		clas.addAll(hs);				// re-add to list without dublicates
		if(clas.size() ==1) {
			return true;				// if there is only one class present: true
			}
		else if( clas.size()>1){	// if there are more than  1 class

			double denom = (double)setSize(set);	// calculate the probability of the majority class

			for(int q=0; q<clas.size(); q++)		// count the number of cases of each of the classes present in the set
			{
				num =(double)count(classes.indexOf(clas.get(q)), set,i);
				if(num> maxnum)
				{
					maxnum=num;								// store the number of cases of the majority class
				}
			}
				prune = maxnum/denom;				// this is to be compared to the user inputted pruning threshold

				if(prune > pruningThresh) // e.g. if the chances of obtaining either class (majority class) is greater than pruning thresh
				{
					return true;				// return true
				}

			return false;								// other wise this is not a leaf
			}
		else {												// default
			return true;
			}
	}


	/***
	 * Basic arithmetic function
	 * @param n value to calculate log to the base 2 of
	 * @return the log base 2
	 */
	private double logb2(double n)
	{
		double x =(Math.log(n)/ Math.log(2));
		return x;
	}
	/****
	 * Basic arithmetic function
	 * @param value, double value to round to two decimal places
	 * @return double value rounded to 2 decimal places
	 */
	private double round(double value) {
	    BigDecimal dec= new BigDecimal(value);
	    dec = dec.setScale(2, RoundingMode.HALF_UP);
	    return dec.doubleValue();
	}
	/*****
	 *
	 * @param set
	 * @return, the number of cases within a set
	 */
	public int setSize(String[][] set)
	{
		int i=0;
		while(set[1][i]!=null)
		{
			i++;
		}
		return i;
	}
	/*****
	 * Clears a set completely, neccessary on each iteration of node creation
	 * @param set, the set to be cleared
	 */
	private void setClear(String[][] set)
	{
		int i=0;
		int j =0;
		for(i=0; i<numatt;i++)
		{
			for(j=0;j<workingCases;j++)
			{
				set[i][j] =null;
			}
		}
	}

}
```