5/18/2016

Decision Tree Based Induction Algorithm

Name: Varunsrivathsa Venkatesha
SBU Id: 110457727

Table of Contents

Abstract:

Classification of data objects based on a predefined knowledge of the objects is a data mining and knowledge management technique used in grouping similar data objects together. It can be defined as supervised learning algorithms as it assigns class labels to data objects based on the relationship between the data items with a pre-defined class label. A decision tree is a powerful method for classification and prediction and for facilitating decision making in sequential decision problems. Decision tree algorithms are a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. These kinds of algorithms are famous in inductive learning and have been successfully applied to a broad range of tasks. In this project, I examine the decision tree algorithm and analyzed the performance and accuracy of the algorithm.

Problem Description:

Implementation of the Decision Tree Induction algorithm for classification by experiment all the three metrics or Splitting rules. Small illustrating examples that were used for validation of implementation should be provided.

Introduction:

With the passage of time, large amount of data is collected in almost every field. The useful data need to be extracted or mined for fulfilling the future needs. Although there are various techniques through which the data can be extracted but the classification technique used for classifying the data for mining is the most useful technique. The classification technique is implemented using different algorithms like decision tree, neural network, genetic algorithm and K-nearest neighbor. The decision trees are constructed using the data whose values are known and precise. These are implemented using ID3,C4.5 or CART algorithms. But during the construction of decision tree major problem of uncertainty arises in the datasets that is responsible for the performance degradation and inaccurate results. The averaging approach and distributed approach are used to handle the problem of uncertainty. These approaches are compared on the same datasets so as to predict which approach reduces the error rate and depict more accurate results.

Data Mining and its Techniques: Different data mining techniques are used to extract the appropriate data from database. This goal of extraction is divided into two categories: prediction and description. In prediction process, the existing database and the variables in the database are interpreted in order to predict unknown or future values. On the other hand, description focuses on finding the patterns describing the data and subsequent presentation for user interpretation. Every data mining technique based either on the concept of prediction or description. Different Data Mining Techniques are association, classification, regression and clustering. Association rule mining is one of the most and well researched techniques of data mining. It aims to extract interesting correlations, frequent patterns, associations or casual structures among sets of items in the transaction databases or other data repositories. Classification is a process which builds a model based on previous datasets so as to predict future trends with new data sets. The association and classification data mining techniques deals with the categorical attribute. The technique used to predict the numeric or continuous values is known as numeric prediction or regression. It was developed by Sir Frances Galton. It is based on the concept of prediction and is used to predict a model based on the observed data over a period of time. Sometimes, the large amount of data is available but few variables in the datasets do not have class labels. The process of assigning class labels to the variables is known as clustering. In this process, the variables in the data sets are grouped together according to the similarity between each other and dissimilarity from other variables.

Classification: Classification is one of the fundamental tasks in data mining and has also been studied extensively in statistics, machine learning, neural networks and expert system over decades. It is a process which builds a model based on previous datasets so as to predict future trends with new data sets. It is a two-step process in which the first step is the learning step which involves building of a classifier or a model based on analyzation of training dataset so as to predict categorical labels. As the class label is provided in the data set, so it is also known as supervised learning. In the second step, the accuracy of the model is predicted using test data set. If the accuracy is acceptable, then the model is used to classify new data tuples. The different classification techniques are decision trees, neural networks, K-nearest neighbor and genetic algorithms. Among these techniques, the decision trees induction algorithms present several advantages over other learning algorithms, such as robustness to noise, low computational cost for generating the model and ability to deal with redundant attributes.

Decision Trees: Decision tree is a predictive modelling based technique developed by Rose Quinlan. It is a sequential classifier in the form of recursive tree structure. There are three kinds of nodes in the decision tree. The node from which the tree is directed and has no incoming edge is called the root node. A node with outgoing edge is called internal or test node. All the other nodes are called leaves (also known as terminal or decision node). The data set in decision tree is analyzed by developing a branch like structure with appropriate decision tree algorithm. Each internal node of tree splits into branches based on the splitting criteria. Each test node denotes a class. Each terminal node represents the decision. They can work on both continuous and categorical attributes.

Decision tree Algorithms: The different decision tree algorithms are ID3, C4.5 and CART

a) ID3 (Iterative Dichotomiser): ID3 is a greedy learning decision tree algorithm introduced in 1986 by Quinlan Ross. It is based on Hunts algorithm. This algorithm recursively selects the best attribute as the current node using top down induction. Then the child nodes are generated for the selected attribute. It uses an information gain as entropy based measure to select the best splitting attribute and the attribute with the highest information gain is selected as best splitting attribute. The accuracy level is not maintained by this algorithm when there is too much noise in the training data sets. The main disadvantage of this algorithm is that it accepts only categorical attributes and only one attribute is tested at a time for making decision. The concept of pruning is not present in ID3 algorithm.

b) C4.5 algorithm: C4.5 is an extension of ID3 algorithm developed by Quinlan Ross. This algorithm overcomes the disadvantage of overfitting of ID3 algorithm by process of pruning. It handles both the categorical and the continuous valued attributes. Entropy or information gain is used to evaluate the best split. The attribute with the lowest entropy and highest information gain is chosen as the best split attribute. The pessimistic pruning is used in it to remove unnecessary branches indecision tree.

c) CART Algorithm: In Classification and Regression Trees by Breiman et al, a CART tree which is a binary decision tree is constructed by splitting a node into two child nodes repeatedly, beginning with the root node. The basic idea of tree growing is to choose a split among all the possible splits at each node so that the resulting child nodes are the “purest”. In this algorithm, only univariate splits are considered. That is, each split depends on the value of only one predictor variable. All possible splits consist of possible splits of each predictor. If X is a nominal categorical variable of i categories, there are 2i-1 – 1 possible splits for this predictor. If X is an ordinal categorical or continuous variable with K different values, there are K - 1 different splits on X. A tree is grown starting from the root node.

Related Work:

1. *Naïve Bayesian Classifier:* The most straightforward and widely tested method for probabilistic induction is known as the naive Bayesian classifier. This scheme represents each class with a single probabilistic summary. In particular, each description has an associated class probability or base rate, p(Ci), which specifies the prior probability that one will observe a member of class Ci. Each description also has an associated set of conditional probabilities, specifying a probability distribution for each attribute. In nominal domains, one typically stores a discrete distribution for each attribute in a description. Each p(Vj/Ci) term specifies the probability of value Vj, given an instance of class Ci. In numeric domains, one must represent a continuous probability distribution for each attribute. This requires that one assume some general form or model, with a common choice being the normal distribution, which can be conveniently represented entirely in terms of its mean and variance.
   To classify a new instance A a naive Bayesian classifier applies Bayes' theorem to determine the probability of each description given the instance,

$$p\left(\frac{Ci}{A}\right) = \frac{p(Ci) * p(A/Ci)}{p(A)}$$

   Given data sets with many attributes, it would be extremely computationally expensive to compute *P*(*A*/Ci). To reduce computation in evaluating *P*(*A*/*Ci*), the naïve assumption of class-conditional independence is made. This presumes that the attribute values are conditionally independent of one another, given the class label of the tuple. Thus,

$$P(A/Ci) = \prod_{k=1}^{n} P(a_k/Ci)$$

$$= P(a1/Ci)\times P(a2/Ci)\times\cdots\times P(an/\mathrm{Ci}).$$

   We can easily estimate the probabilities *P*(*a*1/*Ci*), *P*(*a*2/*Ci*),..., *P*(*an/Ci*) from the training tuples.

   Advantages of using Naïve Bayesian Classifier: Naïve Bayes classifier is a simple classifier. However, although it is simple, Naive Bayes can outperform more sophisticated classification methods. Besides that, exhibits high accuracy and speed when applied to large database. Moreover, it is very fast for both learning and predicting. Its learning time is linear in the number of examples and its prediction time is independent of the number of examples. Naïve Bayes classifier is also fast, consistent, easy to maintain and accurate in the classification of attribute data. And from computation point of view, Naïve Bayes is more efficient both in the learning and in the classification task than Decision Tree.

   Disadvantages of using Naïve Bayesian Classifier: Naive Bayes classifier has strong feature independence assumptions. Also if a class label has no occurrence and a certain attribute value together then the frequency-based probability estimate will be zero. Given Naive-Bayes' conditional independence assumption, when all the probabilities are multiplied will yield zero and this will affect the posterior probability estimate.

2. *Rule Based Classifiers:*
    a. OneR: Is a simple algorithm which builds one rule for each attribute in the training data and then selects the rule with the smallest error rate as its one rule. The algorithm is based on ranking all the attributes based on the error rate. To create a rule for an attribute, the most frequent class for each attribute value must be determined. The most frequent class is simply the class that appears most often for that attribute value. A rule is simply a set of attribute values bound to their majority class. OneR selects the rule with the lowest error rate. In the event that two or more rules have the same error rate, the rule is chosen at random.
    b. PART: PART is a separate-and-conquer rule learning algorithm which produces sets of rules called decision lists which are ordered set of rules. A new data is compared to each rule in the list in turn, and the item is assigned the category of the first matching rule. PART builds a partial C4.5 decision tree in its each iteration makes the best leaf into a rule.
    c. Decision Table algorithm: It summarizes the dataset with a decision table which contains the same number of attributes as the original dataset. Then, a new data item is assigned a category by finding the line in the decision table that matches the non-class values of the data item. Decision Table employs the wrapper method to find a good subset of attributes for inclusion in the table. By eliminating attributes that contribute little or nothing to a model of the dataset, the algorithm reduces the likelihood of over-fitting and creates a smaller and condensed decision table.
    d. DTNB: This algorithm is for building and using a decision table/naive Bayes hybrid classifier. At each point in the search, the algorithm evaluates the merit of dividing the attributes into two disjoint subsets: one for the decision table, the other for naive Bayes. A forward selection search is used, where at each step, selected attributes are modelled by naive Bayes and the remainder by the decision table and all attributes are modelled by the decision table initially. At each step, the algorithm also considers dropping an attribute entirely from the model.
    e. Ridor algorithm: It generates a default rule first and then the exceptions for the default rule with the least weighted error rate. Then it generates the best exceptions for each exception and iterates until pure. Thus it performs a tree-like expansion of exceptions. The exceptions are a set of rules that predict classes other than the default. IREP is used to generate the exceptions.

Advantages of Rule Based Classifiers:

- Modularity: Each rule can be seen like a unit of knowledge.
- Uniformity: All the knowledge is expressed in the same format.
- Naturalness: the rules are a natural format to express knowledge in a dominion.

Disadvantages of Rule Based Classifiers:

- Infinite chaining: Rule-based systems need to be specially crafted to avoid infinite loops
- Possibility of contradictions: Sometimes, when introducing new knowledge to solve some specific problem (for example adding a new rule), we might introduce contradictions with the previous rules
- Inefficiency: Because rules require pattern matching, computational cost of rule-based systems can be very high.
- Complex Domains: Some domains are so complex that tens of thousands of rules would be needed to represent the large number of possible situations.

Proposed Solution:

➢ *Splitting Rules:*

1. Entropy: It is a measure in the information theory, which characterizes the impurity of an arbitrary collection of examples. If the target attribute takes on c different values, then the entropy S relative to this c-wise classification is defined as

$$\text{Entropy(S)} = \sum_{i=0}^{c} -\text{Pi}\log(\text{Pi})$$

where p_i is the proportion/probability of S belonging to class i. Logarithm is base 2 because entropy is a measure of the expected encoding length measured in bits.

Information gain: It measures the expected reduction in entropy by partitioning the examples according to this attribute. The information gain, Gain (S, A) of an attribute A, relative to the collection of examples S, is defined as

$$\text{Gain(S,A) = Entropy(S) - } \Sigma \frac{|SV|}{S}\text{Entropry(Sv)}$$

where Values (A) is the set of all possible values for attribute A, and v S is the subset of S for which the attribute A has value v. We can use this measure to rank attributes and build the decision tree where at each node is located the attribute with the highest information gain among the attributes not yet considered in the path from the root.

2. Gain Ratio: The information gain measure is biased toward tests with many outcomes. That is, it prefers to select attributes with larger number of attribute values. Hence this is an extension for to information gain.

$$\text{GainRatio(A)} = \frac{Gain(A)}{SplitInfoA\ (D)}$$

Where SplitInfo known as 'Split Information' value is defined analogously with Info(D) as

$$\text{SplitInfo(D)} = \ \sum \frac{|Dj|}{|D|} * \log 2(\frac{|Dj|}{|D|})$$

This values represents the potential information generated by splitting the training data set into n number of partitions.

3. Gini Index: Gini Index measures the impurity of *D*, a data partition or set of training tuples. Gini of a data set is defined as,

$$\text{Gini(D)} = 1 - \Sigma p_i^2$$

Gini index considers only binary split for each attribute. Hence if an attribute has more than 2 values then to calculate Gini all the possible combinations of the values should be considered and the Gini is calculated for each combination excluding power set and null set. Hence for a binary split Gini index can be written as,

$$\text{Gini}_A\text{(D)} = \frac{|D1|}{|D|}\text{Gini(D1)} + \frac{|D2|}{|D|}\text{Gini(D2)}$$

➢ *Pseudo Code:*

```
parse the input file and store it in a HashMap;

decisionTree(D,attribute_list){
node = DecesionTreeNode(Tuples in D);
if (all the tuples belongs to same class) {
    node.label = class C;
    return node;
}
else if (attribute_list is empty or number of examples < minimum allowed per branch){
     node.label = most common value in D;
     return node;
}
else {
    bestA = attributeselection (D, attribute_list, splitting_rule);
}
node.decision = bestA;
for(each possible value v of bestA){
     subset = the subset of examples that have value v for bestA;
      If (subset is not empty)
        node.addBranch(decisionTree(subset, targetAttribute, attributes-bestA));
}
return node;
}
```

```
attributeselection (D, attribute_list, splitting_rule) {
log2(x) = log(x)/log(2);
switch(splitting_rule) {
   case(info_gain): for value in attributeValues(D, attribute_list){
                                   sub = subset(D, attribute_list);
                                   gain -=  (number in sub)/(num of tuples_D) *  entropy(sub);
                     }
                     return attribute with max gain;
                     break;
   case(gain_ratio): for value in attributeValues (D, attribute_list) {
                                     Sub = subset(D,attribute_list);
                                     Split_info -= (number in sub)/(total number of D
                                                *log2 ((number in sub)/(number of tuples_D));
                                     gainRatio = gain of that attribute/split_info;
                      }
                      return attribute with max gainRatio;
                      break;
   case(gini_index): for value in attributeValues(D,attribute_list){
                                      If an attribute has more than two attributes
                                             find all the combination of attributes excluding
                                             super set and null;
                                      for each combination{
                                              gini_index = (D1)/(D)*gini(D1)+(D2)/(D)*gini(D2);
                                      }
                      }
                      return attribute with max gini_index;
                      break;
}
}
```

➢ *Detailed description of pseudo code with an example*:

Example Data-set D:

OUTLOOK	TEMPERATURE	HUMIDITY	PLAY GOLF
Rainy	Hot	High	No
Rainy	Mild	High	Yes
Overcast	Mild	Normal	Yes
Sunny	Cool	Normal	Yes
Sunny	Hot	High	No
Sunny	Cool	Normal	Yes

- Read the data from benchmark file and store it in a HashMap<Integer, HashMap<String,String>>, where the each row of outer hashmap corresponds to a row in data set and each row in inner hashmap is to store the value of attributes in that row. As shown below,

1	Outlook	Rainy
	Temperature	Hot
	Humidity	High
	Play Golf	No
2	Outlook	Rainy
	Temperature	Mild
	Humidity	High
	Play Golf	Yes

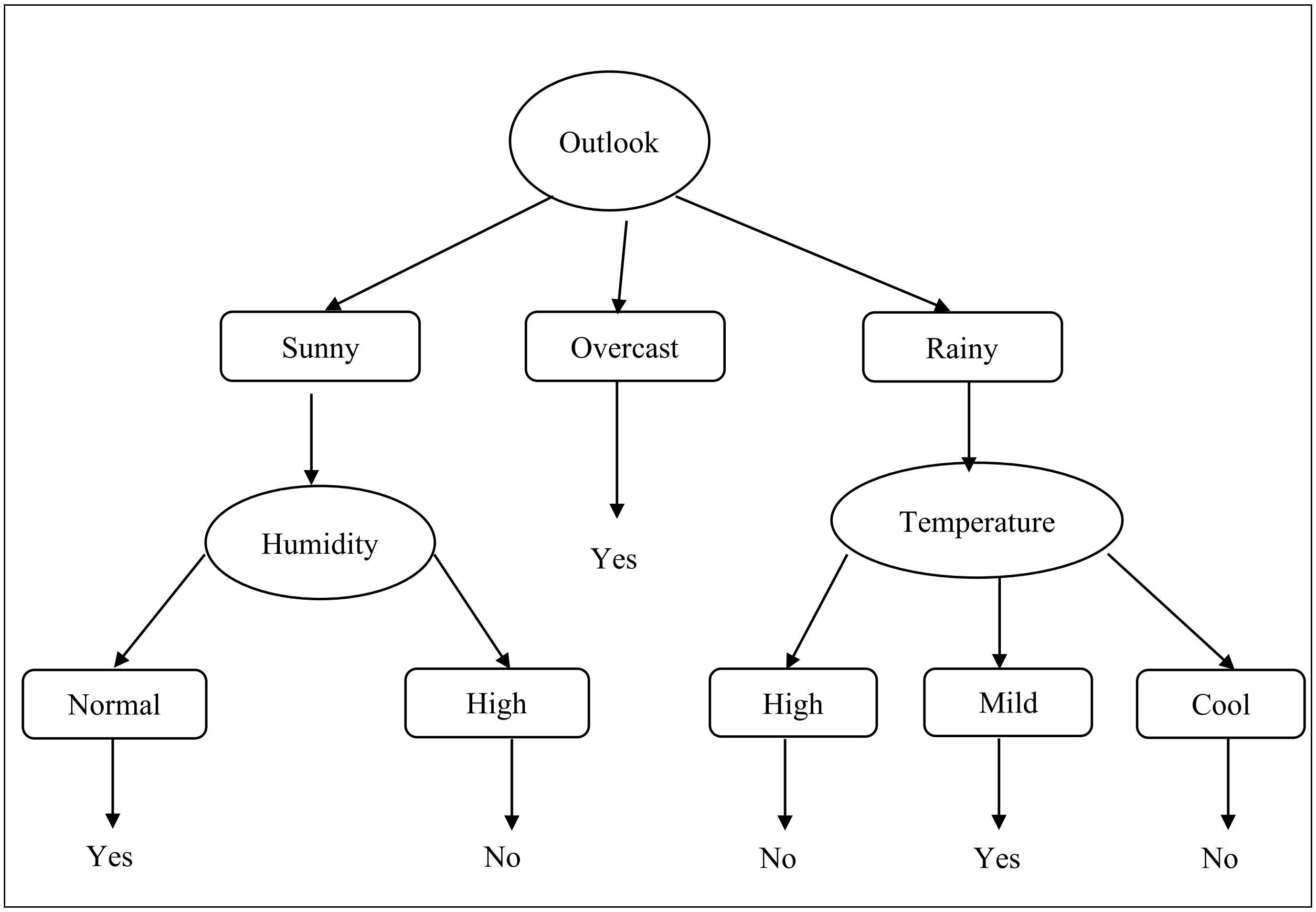
  Similarly store the whole dataset
- Create a node of type decision tree and if all the tuples in D belong to same class then label that node with the same class. The node becomes leaf node.
- If the tuples in the D are different then program calls attributeselection method which takes Dataset D, attribute_list and metric type or Splitting rule as arguments.
- In attribute selection method based on the splitting rule mentioned, Information gain, Gain Ratio or Gini Index is calculated as explained in splitting rules section.
- While calculating the splitting rule for every attribute the corresponding splitting information is stores in an HashMap<String,Double>

Outlook	0.446
Temperature	0.259
Humidity	0.132

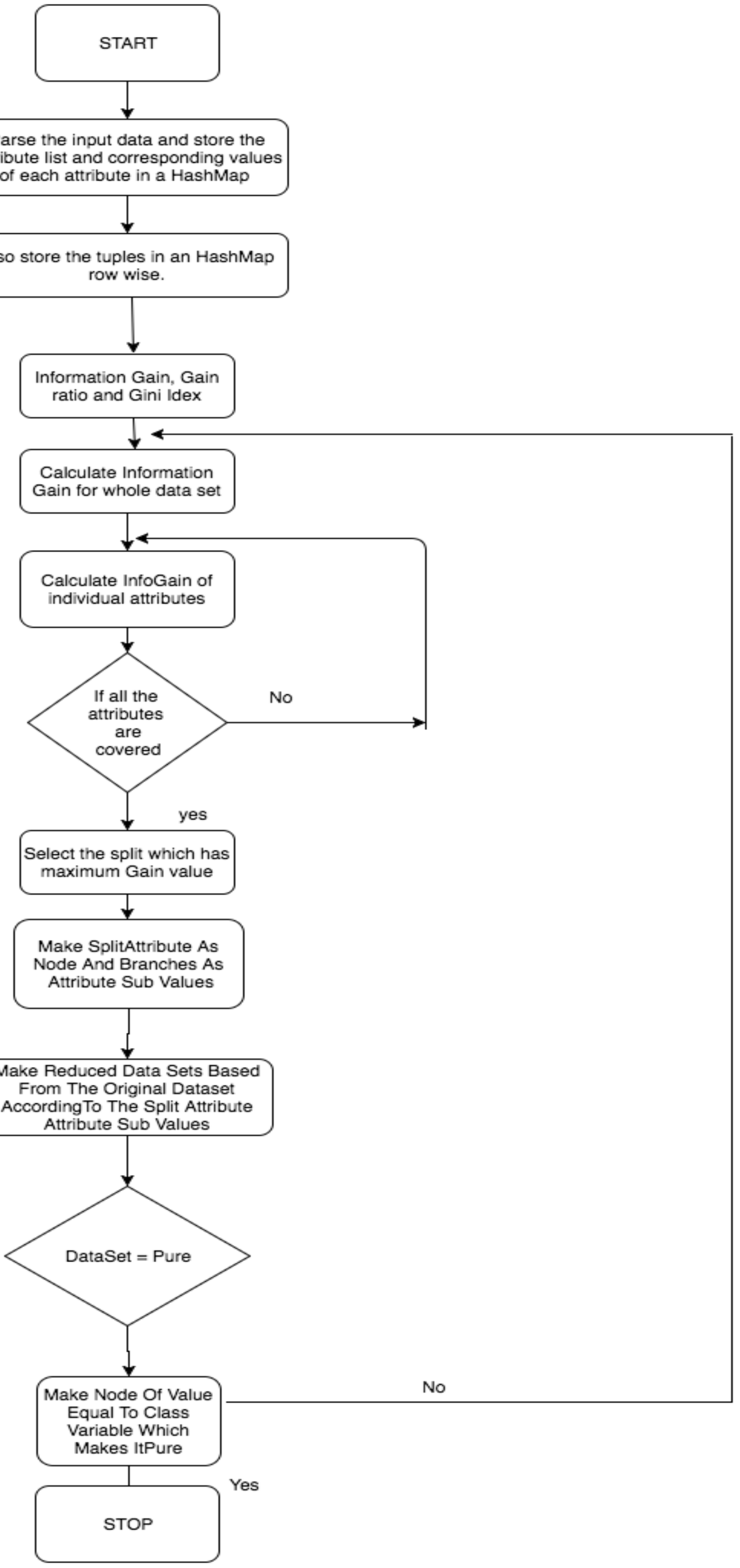
  After calculating the required Splitting rule, attribute with highest splitting rule is returned.
- BasetA stores the attribute to be split and further branches are added to node which are attribute values of the BestA.

- Recursively the partition is done until one of the stopping condition is encountered,
  All the tuples belong to same class.
  No remaining attributes on which tuples may be further partitioned.
  No tuples remaining in a given branch.

Finally, the decision tree constructed look like this,



➢ *Flow Chart:*


START
Parse the input data and store the Attribute list and corresponding values of each attribute in a HashMap
Also store the tuples in an HashMap row wise.
Information Gain, Gain ratio and Gini Idex
Calculate Information Gain for whole data set
Calculate InfoGain of individual attributes
If all the attributes are covered
No
yes
Select the split which has maximum Gain value
Make SplitAttribute As Node And Branches As Attribute Sub Values
Make Reduced Data Sets Based From The Original Dataset AccordingTo The Split Attribute Attribute Sub Values
DataSet = Pure
Make Node Of Value Equal To Class Variable Which Makes ItPure
No
Yes
STOP

Code Implementation:

1. DecisionTreeNode.java

   Attributes of this class are
   - String Label – This is to name the node with attribute value.
   - String decision – This is to store on which attribute to split.
   - Boolean hasSibling – This is to indicate if the node has sibling.
   - LinkedList<DecisionTreeNode> child – A list to store the list of children of the node.

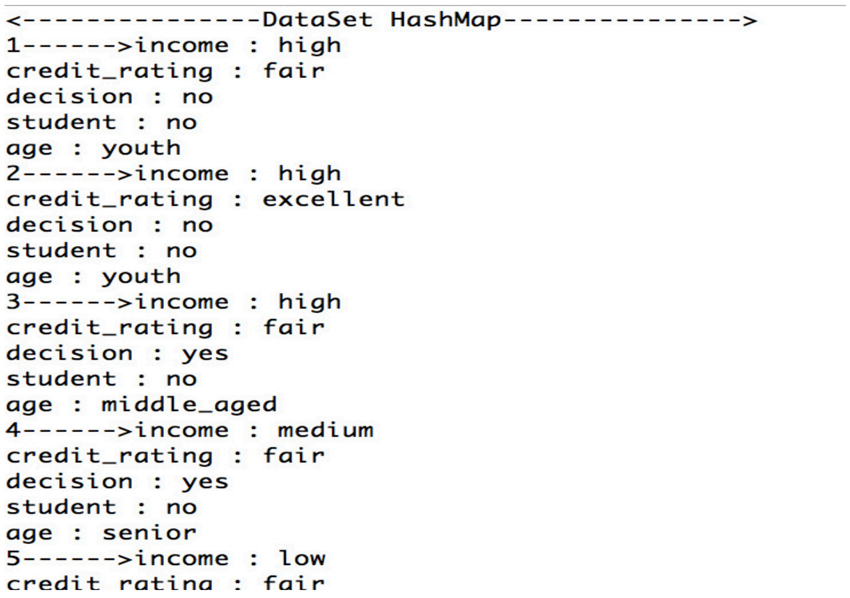
2. Main.java

   Attributes of the class are :-

   - HashMap<Integer,HashMap<String,String>> dataset : To store the values of each attribute row wise.
   - HashMap<String,HashMap<String,Integer>> attributreCount : To have count of the values of each attribute.
   - HashMap<String,HashMap<String,Integer>> decisionCount : To have count of decision of each attribute value.
   - HashMap<String,Double> InfoGain : This is to store Information Gain values of every attribute.
   - HashMap<String,Double> GainRatio : This is to store GainRatio values of every attribute.
   - HashMap<String,Double> Gini_Index: This is to store Gini Index values of every attribute.

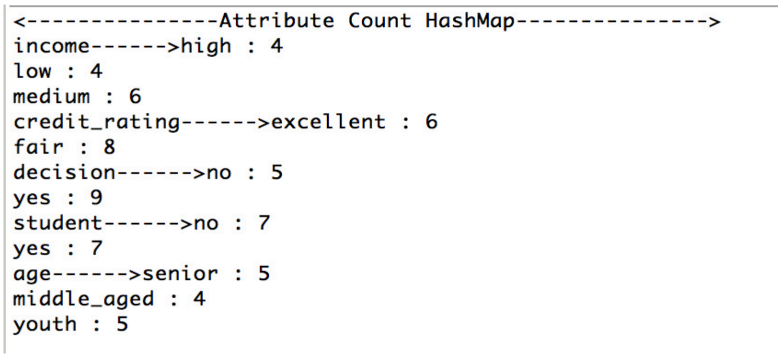
   Functions present in the class:

   1. void rdFile() : This function reads the input data from the file and stores it HashMap<Integer,HashMap<String,String>> dataset.

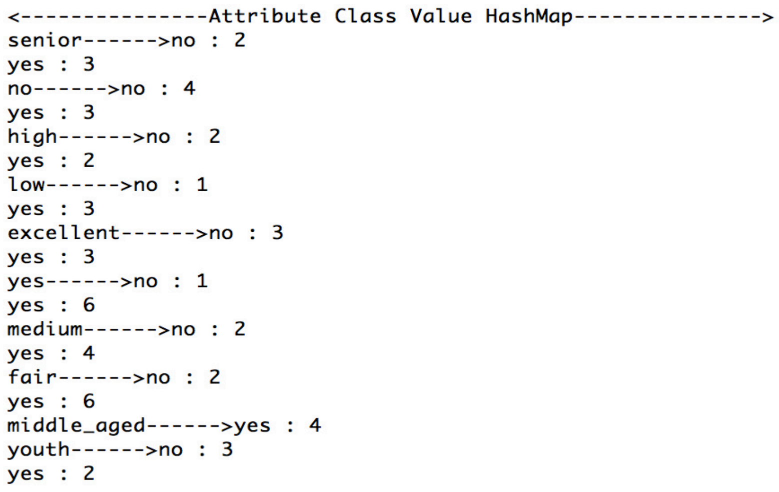


```
<---------------DataSet HashMap--------------->
1------>income : high
credit_rating : fair
decision : no
student : no
age : youth
2------>income : high
credit_rating : excellent
decision : no
student : no
age : youth
3------>income : high
credit_rating : fair
decision : yes
student : no
age : middle_aged
4------>income : medium
credit_rating : fair
decision : yes
student : no
age : senior
5------>income : low
credit rating : fair
```

2. createFreqTable() :- A function that calculates number of times the attribute value belong to particular class variable. It stores the value in attributeCount.



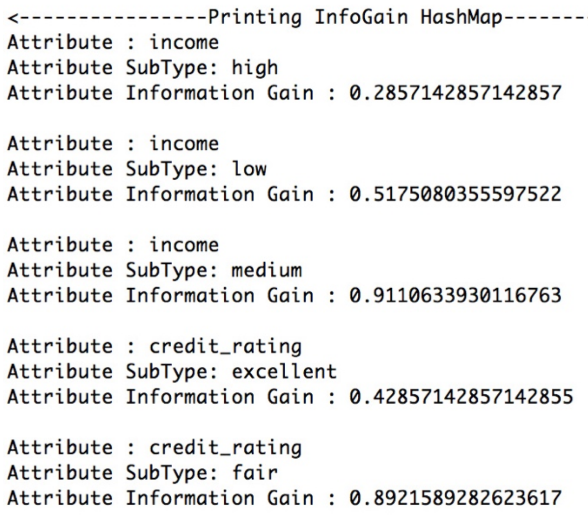
```
<---------------Attribute Count HashMap--------------->
income------>high : 4
low : 4
medium : 6
credit_rating------>excellent : 6
fair : 8
decision------>no : 5
yes : 9
student------>no : 7
yes : 7
age------>senior : 5
middle_aged : 4
youth : 5
```



```
<---------------Attribute Class Value HashMap--------------->
senior------>no : 2
yes : 3
no------>no : 4
yes : 3
high------>no : 2
yes : 2
low------>no : 1
yes : 3
excellent------>no : 3
yes : 3
yes------>no : 1
yes : 6
medium------>no : 2
yes : 4
fair------>no : 2
yes : 6
middle_aged------>yes : 4
youth------>no : 3
yes : 2
```

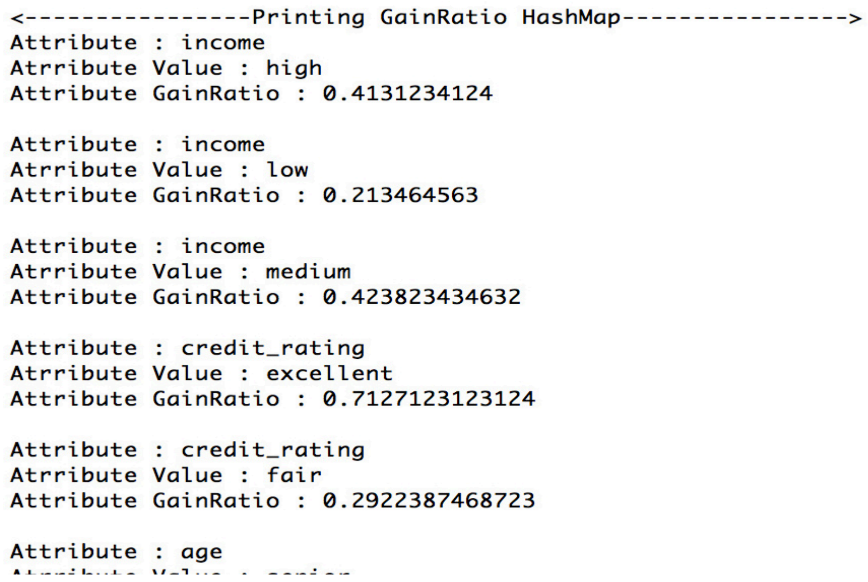
3. extractAttribute() :- this function extracts individual attribute from dataset HashMap and sends it to calcInfoGain(), calcGainRatio() and calGiniIndex() functions where the values calculated. Values calculated for every attribute value is stored in HashMap of<String,Double> InfoGain, HashMap of<String,Double> GainRatio and HashMap of<String,Double> Gini_Index correspondingly.

4. calculateInfoGain(String attributeValue): A function to calculate information gain of every Attribute and the calculated InfoGain for every attribute value is stored in HashMap of<String,Double> InfoGain



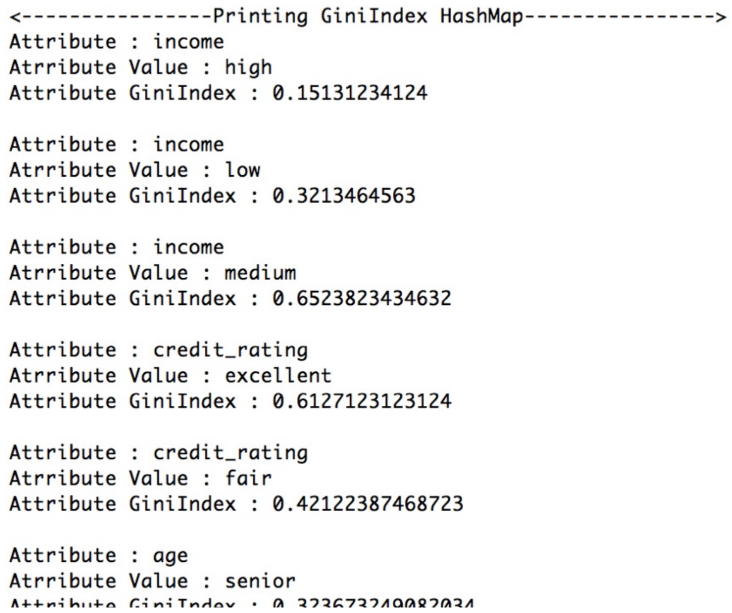
```
<---------------Printing InfoGain HashMap--------------->
Attribute : income
Attribute SubType: high
Attribute Information Gain : 0.2857142857142857

Attribute : income
Attribute SubType: low
Attribute Information Gain : 0.5175080355597522

Attribute : income
Attribute SubType: medium
Attribute Information Gain : 0.9110633930116763

Attribute : credit_rating
Attribute SubType: excellent
Attribute Information Gain : 0.42857142857142855

Attribute : credit_rating
Attribute SubType: fair
Attribute Information Gain : 0.8921589282623617
```

5. calculateGainRatio(String attributeValue): A function to calculate SplitInfo and inturn find GainRatio of every Attribute and the calculated GainRatio for every attribute value is stored in HashMap of<String,Double> GainRatio.



```
<----------------Printing GainRatio HashMap---------------->
Attribute : income
Atrribute Value : high
Attribute GainRatio : 0.4131234124

Attribute : income
Atrribute Value : low
Attribute GainRatio : 0.213464563

Attribute : income
Atrribute Value : medium
Attribute GainRatio : 0.423823434632

Attribute : credit_rating
Atrribute Value : excellent
Attribute GainRatio : 0.7127123123124

Attribute : credit_rating
Atrribute Value : fair
Attribute GainRatio : 0.2922387468723

Attribute : age
```

6. calculateGiniIndex(String attributeValue): A function to calculate Gini Index of every Attribute and the calculated I Gini Index for every attribute value is stored in HashMap of<String,Double> Gini Index.

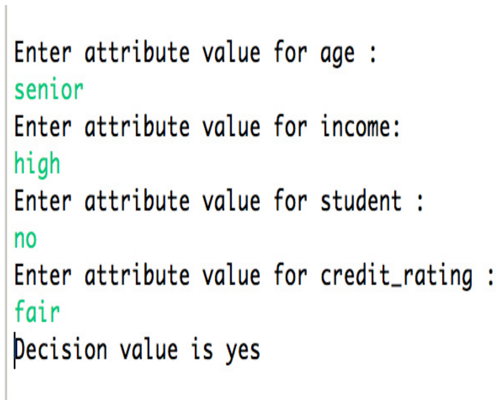


```
<----------------Printing GiniIndex HashMap---------------->
Attribute : income
Atrribute Value : high
Attribute GiniIndex : 0.15131234124

Attribute : income
Atrribute Value : low
Attribute GiniIndex : 0.3213464563

Attribute : income
Atrribute Value : medium
Attribute GiniIndex : 0.6523823434632

Attribute : credit_rating
Atrribute Value : excellent
Attribute GiniIndex : 0.6127123123124

Attribute : credit_rating
Atrribute Value : fair
Attribute GiniIndex : 0.42122387468723

Attribute : age
Atrribute Value : senior
Attribute GiniIndex : 0.323673249082034
```

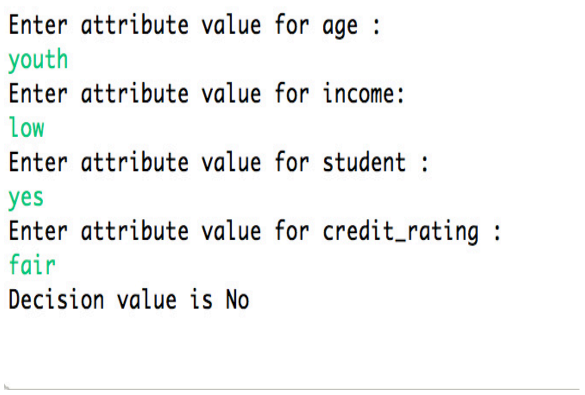
7. combinationAttributes(): this function is used to find all the combination of attribute values for all the attributes.

8. createDecisionTree(): This function is to get the attribute with highest attribute measure and form braches to the root node. This function is called recursively until termination condition is reached i.e until all the tuples are pure or all the attributes are covered.

9. checkCorrectness(): This function takes random test tuples and gives output based on the decision tree formed.

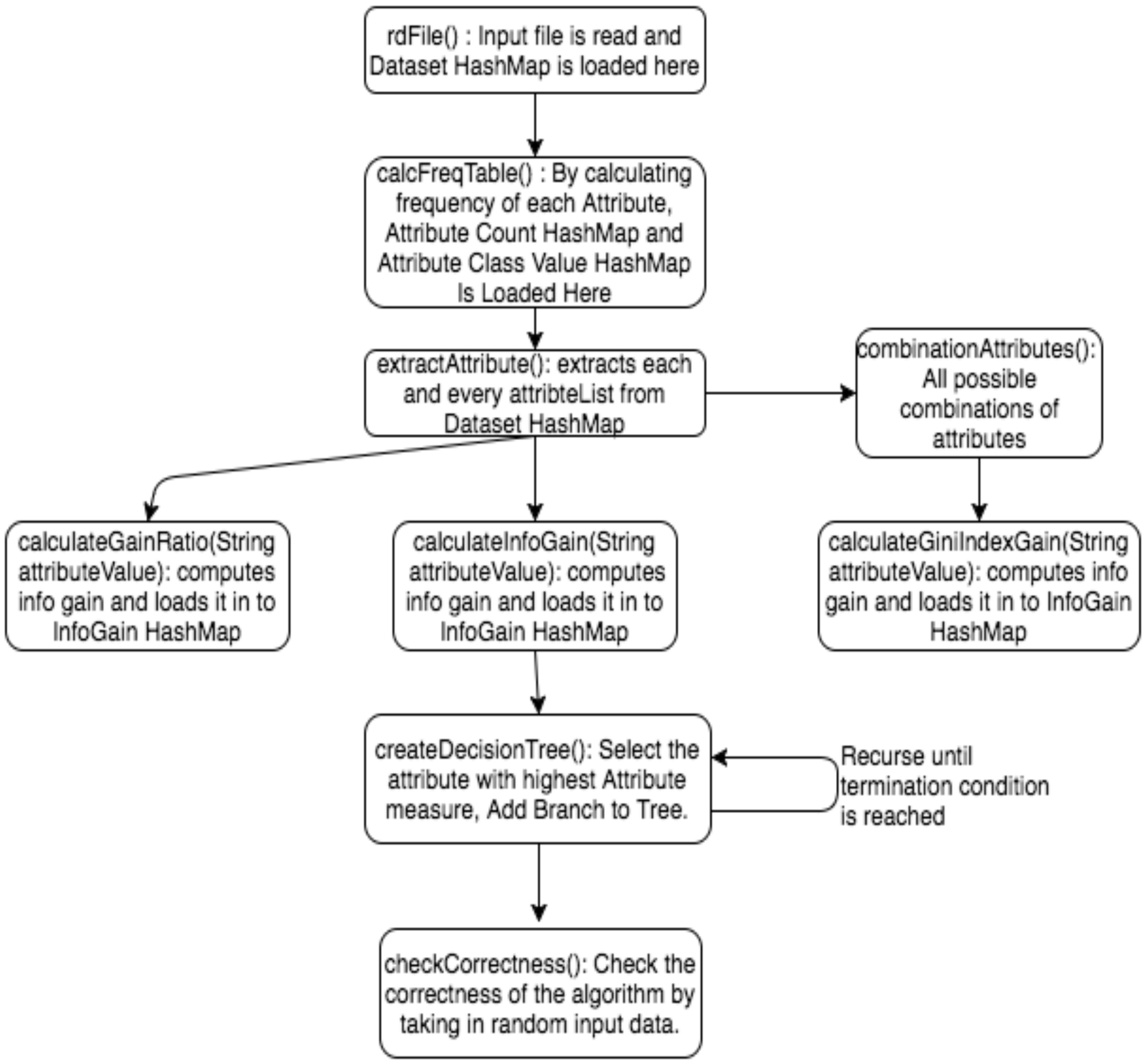


```
Enter attribute value for age :
senior
Enter attribute value for income:
high
Enter attribute value for student :
no
Enter attribute value for credit_rating :
fair
Decision value is yes
```



```
Enter attribute value for age :
youth
Enter attribute value for income:
low
Enter attribute value for student :
yes
Enter attribute value for credit_rating :
fair
Decision value is No
```

Data structure and Method Flow:


rdFile() : Input file is read and Dataset HashMap is loaded here
calcFreqTable() : By calculating frequency of each Attribute, Attribute Count HashMap and Attribute Class Value HashMap Is Loaded Here
extractAttribute(): extracts each and every attribteList from Dataset HashMap
combinationAttributes(): All possible combinations of attributes
calculateGainRatio(String attributeValue): computes info gain and loads it in to InfoGain HashMap
calculateInfoGain(String attributeValue): computes info gain and loads it in to InfoGain HashMap
calculateGiniIndexGain(String attributeValue): computes info gain and loads it in to InfoGain HashMap
createDecisionTree(): Select the attribute with highest Attribute measure, Add Branch to Tree.
Recurse until termination condition is reached
checkCorrectness(): Check the correctness of the algorithm by taking in random input data.

Results:

1. Initial Data Obtained from input file.

Sr. No.	Data Set Name	Number of Attributes	Number of Training Tuples	Number of Testing Tuples
1	Cars	6	1600	128
2	Pittsburg Highway	13	80	26
3	Iris	5	134	20
4	Textbook Example	5	10	5
5	Mushroom	23	7000	1124
6	Chronic Kidney Disease	25	350	50
7	Bank	17	4000	521

2. Output when Information gain is chosen as Splitting Rule.

Sr. No.	Data Set Name	Accuracy percentage	Execution Time (ms)	Memory used (MB)
1	Cars	65.03876	748	8
2	Pittsburg Highway	11.538462	470	2
3	Iris	100	383	2
4	Textbook Example	83.33	77	1
5	Mushroom	70.59	2137	22
6	Chronic Kidney Disease	60.78431	1035	5
7	Bank	84.09962	6287	7

3. Output when Gain Ration is chosen as Splitting Rule.

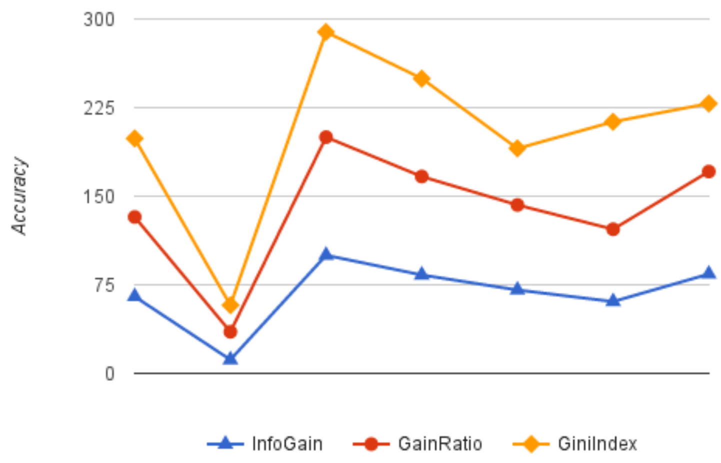
Sr. No.	Data Set Name	Accuracy percentage	Execution Time (ms)	Memory used (MB)
1	Cars	67.358	713	8
2	Pittsburg Highway	23.669	417	2
3	Iris	100	248	2
4	Textbook Example	83.33	56	1
5	Mushroom	71.98	2345	22
6	Chronic Kidney Disease	61.235	710	5
7	Bank	86.78161	4411	14

4. Output when Gini Index is chosen as Splitting Rule.

Sr. No.	Data Set Name	Accuracy percentage	Execution Time (ms)	Memory used (MB)
1	Cars	66.358	786	8
2	Pittsburg Highway	22.5	570	2
3	Iris	89	439	2
4	Textbook Example	83	76	1
5	Mushroom	47.8	1278	26
6	Chronic Kidney Disease	91	209	2
7	Bank	57.6	1652	46

Analysis of Result:

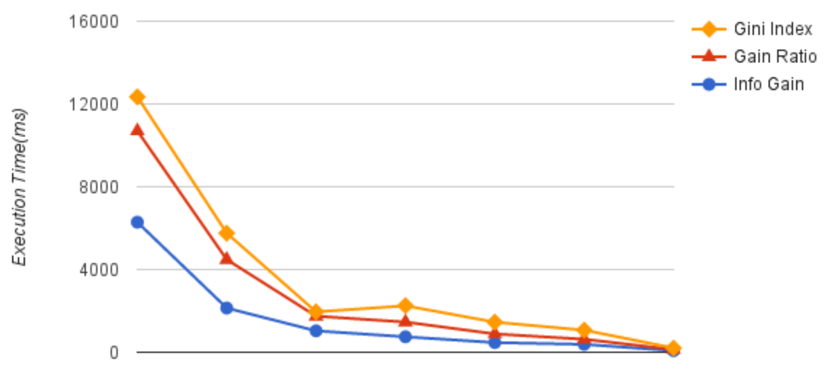
1. Accuracy Comparison of Decision Tree Algorithm for different Splitting Rules.



As we can see from the chart,

- Accuracy of the output is least when Splitting Rule Information Gain is used and Accuracy is most when Gain Ratio is used.
- This pattern is observed because in the benchmark files used there is an attribute in every file which has more distinct values hence while using Information gain as Splitting Rule, algorithm picks that attribute to split leading to less accuracy.
- There is a sudden dip in the plot because Pittsburg Highway data set has transaction Id as once of the attribute hence all the algorithms are choosing that attribute to split which leads to less accuracy.

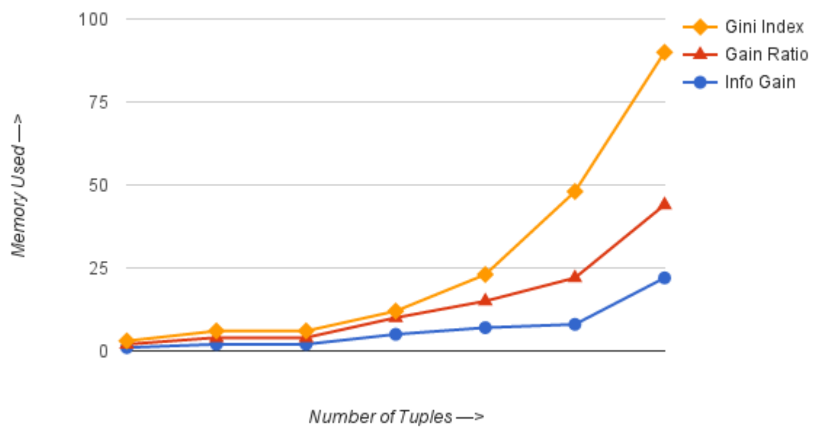
2. Execution Time comparison of Decision Tree Algorithm for different Splitting Rules.



As we can see from the chart,

- Execution time when Gini Index is used is always higher than other two splitting rules.
- This is because to calculate Gini Index the program calls one extra function combinationAttributes() which finds all the combinations of attribute values for every attribute.
- This graph shows continuous downward trend because the benchmark file used to plot the graph are arranged in decreasing order of Number of tuples.
- There is a sudden dip from 1st point to 3rd point because number of tuples in the benchmark files considered decreases significantly from 6000 to 700.

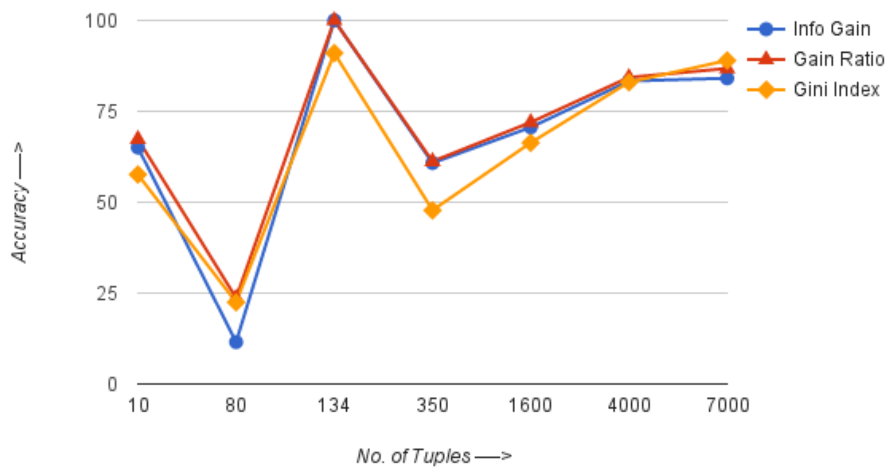
3. Memory comparison of Decision Tree Algorithm for different Splitting Rules.



As we can see from the chart,

- Memory Used by Algorithm when Gini Index is used as Splitting Ratio is more compared to other two because of the extra information such as attribute combinations and Gini Index to all the combination is to be stored.
- Memory Used has upward trend with number of tuples in all the cases.
- As the number of tuples increases the gap between the curves increase because the number of combinations produced to find the Gini Index increases exponentially. Hence memory required to store Gini Index of all the combinations also increases.

4. Variation of Accuracy with Number of Tuples.



As we can see from the chart,

- Variation of Accuracy with number of tuples is not as expected. Accuracy should increase with number of tuples because larger the number of tuples more training is done to build the classifier hence the output should be more accurate.
- Here the Anomaly is observed because, for the dataset corresponding to Number of Tuples 80, it has transaction Id as one of the attribute hence the algorithm in all the three cases has picked the same attribute to split. This led to significant drop in accuracy.
- Also for the dataset corresponding to Number of Tuples 134, that data set has all the attributes as nominal values hence the accuracy of the algorithm shoots up as the classifier gives expected value all the time.

5. Decision tree obtained for Pittsburg Highway Dataset:

```
Material = Wood: Wood (16.0)
Material = Iron
  Erected <= 1859: Suspen (3.0)
  Erected > 1859: Simple-T (8.0)
Material = Steel
  Purpose = Highway
      T-Or-D = Through
        Rel-L = S: Arch (5.13/1.13)
        Rel-L = S-F
          Lanes <= 2: Simple-T (2.13/1.13)
          Lanes > 2: Suspen (3.0)
        Rel-L = F
          Erected <= 1918: Simple-T (15.52/5.74)
          Erected > 1918
            Location <= 45: Arch (12.0/5.0)
            Location > 45: Cantilev (2.0/1.0)
      T-Or-D = Deck
        Erected <= 1939: Cantilev (5.22/2.22)
        Erected > 1939: Cont-T (6.0)
   Purpose = Aqueduct: Simple-T (0.0)
   Purpose = Rr: Simple-T (27.0/4.0)
   Purpose = Walk: Suspen (1.0)

Number Of Leaves  :       14  Size Of The Tree :      23
```

Conclusion:

As observed from all the graphs, Accuracy is not only a function of number of training tuples but also on the characteristic of the tuples. That is when the tuples are all nominal value higher accuracy is achieved and accuracy reduces when one of the tuple has all different attribute values. Execution time and Memory used behaved as expected with number of tuples in the dataset. Higher the number of Tuples higher the execution time and memory used. Finally, Decision tree algorithm with all the Splitting rules is analyzed and the output behaved as expected.

Q. Discuss how the algorithm must be modified to run on a grid network of embedded sensors as the one discussed in the paper.

A. In a grid based network, the embedding cells are divided into cells independent of the distribution of input objects. The object cells are divided into a finite number of cells that form a grid structure on which all the operations are performed. We know that the timestamp of each sensor reading is locally dumped in the microcontrollers cache.

Decision Tree Algorithm can be applied as follows: The Decision Tree algorithm can be applied to the sensor data to get to know the working of sensor at different conditions. A classifier can be built to predict the sensor value based on all the attributes which influence the sensor values. Accuracy of the classifier will be considerable low as all the attributes are numerical.

Result:

Decision tree algorithm with splitting rule as Information gain, Gain ratio and Gini Index is implemented. Analysis on the accuracy and performance of the algorithm is done for all kinds of splitting rules.

Bibliography:

1. Fayyad, U., Piatetsky-Shapiro, G., & Smyth, P. (2006, March). "From Data Mining to Knowledge Discovery in Databases". The Knowledge Engineering Review,Vol 21, No.1, pp. 1-24.
2. Han, J., & Kamber, M. (2006). "Data Mining: Concepts and Techniques" (2nd ed.). Morgan Kaufmann Publishers.
3. Pujari, A. K. (2001). "Data Mining Techniques". Universites Press India Private Limited.
4. Agarwal, R., Imieliński, T., & Swami, A. (1993, June). "Mining association rules between sets of items in large databases". ACM SIGMOD Record, Vol 22,No.2, pp. 207-216.
5. An Introduction to Data Mining,Review: http://www.thearling.com/text/dmwhite/dmwhite.html
6. A Tutorial on Clustering Algorithms, Review http://home.dei.polimi.it/matteucc/Clustering/tutorial_html
7. Naive Bayes Classifier Review: http://www.statsoft.com/textbook/naive-bayes-classifier/
8. Lecture notes by Professor Alexa Doboli.

Appendix:

1. Main.java

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;


public class Main{

        HashMap<Integer,DecisionNode> attributeMap = new HashMap<Integer,DecisionNode>();
        HashMap<String,ArrayList<String>> atttributeValue = new HashMap<String,ArrayList<String>>();
        ArrayList<String> attributes = new ArrayList<String>();

        HashMap<Integer,HashMap<String,String>> dataSet = new
HashMap<Integer,HashMap<String,String>>();

        HashMap<String, HashMap<String,Integer>> countRelativeAttr = new HashMap<String,
HashMap<String,Integer>>();

        HashMap<String,HashMap<String,Integer>> countofclassvalues = new
HashMap<String,HashMap<String,Integer>>();

        HashMap<String,Integer> countRelativeGain = new HashMap<String,Integer>();

        int numYes=0,numNo=0,numYouth=0,numMiddle_age=0,numSenior=0;
        double infoGain;

public void rdFile(){
                String fileName = "/Users/Varun/Documents/workspace/DecisionTree/src/input.txt";
     String line = null;


     try {
        FileReader fileReader = new FileReader(fileName);
        BufferedReader bufferedReader = new BufferedReader(fileReader);

        int index = 0,elementNum=5;

        while((line = bufferedReader.readLine()) != null) {
                index++;

                String[] arr = line.split(",");
                DecisionNode dn = new DecisionNode();
                dn.age = arr[0];
```

```
                    dn.income = arr[1];
                    dn.student = arr[2];
                    dn.creditRanking = arr[3];
                    dn.decision = arr[4];
                    attributeMap.put(index, dn);

                    //System.out.println(attributes.get(0)+ "last" + attributes.get(4));


                    HashMap<String,String> valueOfeachObject = new HashMap<String,String>();
                    for(int i=0; i<elementNum; i++){
                            valueOfeachObject.put(attributes.get(i),arr[i]);
                    }
                    dataSet.put(index, valueOfeachObject);

            }
            bufferedReader.close();
        }
        catch(FileNotFoundException ex) {
            System.out.println(
                "Unable to open file '" +
                fileName + "'");
        }
        catch(IOException ex) {
            System.out.println(
                "Error reading file '"
                + fileName + "'");
        }
            }

public void printMap(){

//          for(Map.Entry<String,Integer> j : countRelativeGain.entrySet()){
//                  System.out.println(j.getKey() +":");
//                  System.out.println(j.getValue());
//          }

            System.out.println("<---------------Attribute Class Value HashMap--------------->");

            for(Map.Entry<String, HashMap<String,Integer>> i : countofclassvalues.entrySet()){
                    System.out.print(i.getKey() + "------>");
                    for(Map.Entry<String,Integer> j : i.getValue().entrySet()){
                            System.out.print(j.getKey() + " : ");
                            System.out.println(j.getValue());
                    }
            }

}

public void addAttributeValues(){
            ArrayList<String> values1 = new ArrayList<String>();
            values1.add("youth");
            values1.add("middle_aged");
            values1.add("senior");
            ArrayList<String> values2 = new ArrayList<String>();
            values2.add("high");
            values2.add("medium");
            values2.add("low");
            ArrayList<String> values3 = new ArrayList<String>();
            values3.add("yes");
            values3.add("no");
            ArrayList<String> values4 = new ArrayList<String>();
            values4.add("fair");
            values4.add("excellent");
            ArrayList<String> values5 = new ArrayList<String>();
            values5.add("yes");
            values5.add("no");

            atttributeValue.put("age",values1);
            atttributeValue.put("income",values2);
            atttributeValue.put("student",values3);
            atttributeValue.put("credit_rating",values4);
            atttributeValue.put( "decision",values5);

            attributes.add("age");
            attributes.add("income");
            attributes.add("student");
            attributes.add("credit_rating");
            attributes.add("decision");
}

public void getCountAge(){

            for(Map.Entry<Integer, DecisionNode> i : attributeMap.entrySet()){
                    if(i.getValue().decision.equals("no")){
                            numNo++;
                    }
                    else{
                            numYes++;
                    }
            }
}

public void findInfoGain(){
            int numTotal = numYes+numNo;

            double ratio1 = (double)numYes/numTotal;
            double ratio2 = (double)numNo/numTotal;

            double term1 = -ratio1 * ((Math.log(ratio1))/(Math.log(2)));
            double term2 = -ratio2 * ((Math.log(ratio2))/(Math.log(2)));

            infoGain = term1+term2;
//      System.out.println();
//        System.out.println("<----------------Printing InfoGain HashMap---------------->");
}

public void findRelativeGain(){
            for(Map.Entry<String,ArrayList<String>> j: atttributeValue.entrySet()){
                    HashMap<String,Integer> t = new HashMap<String,Integer>();
                    informationgainattribute(j.getKey());
                    ArrayList<String> temp = j.getValue();
                    for(int k=0; k<temp.size();k++){
                            int x = countAttributes(temp.get(k), j.getKey());
                            t.put(temp.get(k), x);
                            countRelativeGain.put(temp.get(k), x);
                    }
                    countRelativeAttr.put(j.getKey(), t);
            }
}


public int countAttributes(String value,String mainAttr){
            int count=0;
            for(Map.Entry<Integer, HashMap<String,String>> i : dataSet.entrySet()){
                    HashMap<String,String> map = i.getValue();
                    for(Map.Entry<String,String> j : map.entrySet()){
                            if(j.getValue().equals(value) && j.getKey().equals(mainAttr)){
                                    count++;
                            }

                    }
            }
            return count;
}

public void findDecisionCount(){
            for(Map.Entry<String,ArrayList<String>> j: atttributeValue.entrySet()){
                    HashMap<String,Integer> t = new HashMap<String,Integer>();
                    ArrayList<String> temp = j.getValue();
                    for(int k=0; k<temp.size();k++){

                    }
            }
}


public int countDecisions(String AttrName,String value){
            int count=0;
            for(Map.Entry<Integer, HashMap<String,String>> i : dataSet.entrySet()){
                    HashMap<String,String> map = i.getValue();
                    for(Map.Entry<String,String> j : map.entrySet()){
                            if(j.getValue().equals(value) && j.getValue().equals(AttrName)){
                                    count++;
                            }
                    }
            }

            return count;
}


public double informationgainattribute(String attriname) {
            HashMap<String,Integer> attributetypefreq = new HashMap<String,Integer>();
            ArrayList<String> attributesubvalues = atttributeValue.get(attriname);

            for(String eachvalue:attributesubvalues){
                    HashMap<String,Integer> tempstorage = new HashMap<String,Integer>();
                    for(int rownumber:dataSet.keySet()){
                            HashMap<String,String> selecrow = (HashMap<String,String>)
dataSet.get(rownumber);

                                    if(selecrow.get(attriname).equals(eachvalue)){
                                    if(tempstorage.containsKey(selecrow.get("decision"))){

            tempstorage.put(selecrow.get("decision"),tempstorage.get(selecrow.get("decision"))+1);
                                    }else{
                                                    tempstorage.put(selecrow.get("decision"),1);
                                            }
                            }

                    countofclassvalues.put(eachvalue, tempstorage);

                    }
            }

            for(int rownumber:dataSet.keySet()){
                    HashMap<String,String> selecrow = (HashMap<String,String>) dataSet.get(rownumber);
                    String attritype = selecrow.get(attriname);
                    if(!attributetypefreq.containsKey(attritype)){
                            attributetypefreq.put(attritype, 1);
                    }else{
                            attributetypefreq.put(attritype,attributetypefreq.get(attritype)+1);
                    }

            }
            double attriinfo = 0;
            double pi = 0;
            double attrieachsubtypeinfo = 0;


            for(String aname : attributetypefreq.keySet()){
                    int countname = attributetypefreq.get(aname);
                    ArrayList<Integer> classvariableval = new ArrayList<Integer>();
                    if(countofclassvalues.containsKey(aname)){
                            HashMap<String,Integer> classvariablemap =new
HashMap<String,Integer>(countofclassvalues.get(aname));
```

```
                                for(String eachclassvariable: classvariablemap.keySet() ){
                                        classvariableval.add(classvariablemap.get(eachclassvariable));
                                }
                        }


                pi = (double)countname/dataSet.keySet().size();


//              System.out.println("Attribute : " + attriname);
//                      System.out.println("Attribute SubType: " + aname);
                        attrieachsubtypeinfo = infoD(classvariableval,countname);
                        attriinfo +=(double)attrieachsubtypeinfo*pi;

//                      System.out.println("Attribute Information Gain : " + attriinfo);
//                      System.out.println();

                }
                return attriinfo;
        }

        public double infoD(ArrayList<Integer>cvariablescountset,int tuplecount){
                double info = 0;

                for(int i = 0 ; i < cvariablescountset.size() ; i++){
                        double var1 = ((double)cvariablescountset.get(i) / tuplecount);
                        info += - (var1 * (Math.log(var1) / Math.log(2)));
                }

                return info;
        }

  public void buildTree() throws FileNotFoundException {

                        if(remainingattributes.size() == 0 ){
                                setLeaf(1);

                        }

                        else if(pos_classcount == data.size() || neg_classcount == data.size()){
                                setLeaf(2);

                        }
                        else{

                                bestattributeindex = findBestAttribute();

                                if(bestattributeindex != -1){
                                        className =
  mainClass.getAttributeName(remainingattributes.get(bestattributeindex));
                                        int temp = remainingattributes.get(bestattributeindex);
                                        remainingattributes.remove(bestattributeindex);
                                        for(int j = 0; j< 2;j++){

                                                ArrayList<Record> subrecordschild = getSubrecord(data, temp,
  Integer.toString(j));
                                                if(subrecordschild.size() != 0){


                                                        DecisonTreeCons_heu child_temp = new
  DecisonTreeCons_heu(subrecordschild, remainingattributes);
                                                        child.add(new children(child_temp, j));
                                                        child_temp.buildTree();

                                                }


                                        }
                                }
                                else{
                                        setLeaf(1);
                                }


                        }
  }


  private void setLeaf(int i) {
                        if(i == 2){
                                isLeaf = true;
                                classvalue = data.get(0).class_label;


                        }
                        else if( i == 1){
                                isLeaf = true;
                                if(pos_classcount> neg_classcount){
                                        classvalue = "1";
                                }
                                else{ classvalue ="0";
                                }
                        }


  }


  public int findBestAttribute() throws FileNotFoundException {


                        double max_variance = 0;
```

```
		int index =  0;


		for(int i = 0; i< remainingattributes.size();i++){


			double sub_pos_neg_variance = 0;
			for(int j=0; j<2; j++){


				ArrayList<Record> subdata = getSubrecord(data, remainingattributes.get(i),
Integer.toString(j));
				double  curvalue_variance  =  findCurrentVariance(subdata);
				//System.out.println(curvalue_variance+"---<");
				//sub_pos_neg_variance  += ( (double)(double)subdata.size()/
(double)data.size() )  * curvalue_variance;
				sub_pos_neg_variance  += (subdata.size()/ (double)data.size() ) *
curvalue_variance;
			}

			//System.out.println("Sub variance for attribute : "+ mainClass.getAttributeName(i) +
" : "+ sub_pos_neg_variance);
			if(max_variance < (variance - sub_pos_neg_variance)){


				max_variance = variance - sub_pos_neg_variance;
				index = i;



			}
		}
		if(max_variance == 0){
			return -1;
		}


		return index;
	}





	private double findCurrentVariance(ArrayList<Record> subdata) {
		int p_count = 0; int n_count = 0;

		for(int i = 0; i< subdata.size();i++){
			Record record = subdata.get(i);


			if	(record.getClass_label().equals("1")){
```

```
                    p_count += 1;
                }
            }
            n_count = subdata.size() - p_count;
            double a = (double)((double)p_count/ subdata.size());
            double b = (double)((double)n_count/ subdata.size());
            if( !Double.isNaN((a*b)) ){
                return (a*b);
            }
            else{
                return 0;
            }


        }


        public ArrayList<Record> getSubrecord(ArrayList<Record> datalist, int attributeposition, String
attributevalue) {


            ArrayList<Record> subset = new ArrayList<Record>();
            for(int i = 0; i < datalist.size(); i++) {
                Record record = datalist.get(i);
                if(record.getRecord_row()[attributeposition].equals(attributevalue)) {
                    subset.add(record);
                }
            }


            return subset;
        }




        private double calcVariance(ArrayList<Record> records) {


            int totalrecords = pos_classcount+neg_classcount;
            double a = (double)((double)pos_classcount/(double)totalrecords);
            double b = (double)((double)neg_classcount/(double)totalrecords);
            double c = a*b;
            if(!Double.isNaN(c)){
                return c;
            }
            else{
                return 0;
            }

        }
```

```
private int calc_poscount(ArrayList<Record> records) {
        int count = 0;
        for(int j = 0; j < records.size(); j++) {
                Record record = records.get(j);
                int a = Integer.parseInt(record.getClass_label());
                if(a == 1) {
                        count += 1;
                }


        }
        return count;
}


public String printDtree(int tabCount) {
        tabCount++;



        if(isLeaf ){


                return classvalue;
        }
        else{
                for(int counter = 0; counter < child.size(); counter++ ){
                        System.out.println();


                        for(int count = 0; count < tabCount; count++){
                                System.out.print("| ");
                        }
                        System.out.print("|" + className + "= " + child.get(counter).value + ": ");
                        String formatPrint = child.get(counter).childPointer.printDtree(tabCount);
                        if(formatPrint.equals("0") || formatPrint.equals("1")){
                                System.out.print(formatPrint);
                        }
                }
                return "null";
        }
}


public String traverseTree(Record testRecord){
```

```
                    DecisonTreeCons_heu node = this;
                    while( node.isLeaf != true ){


                              String best_attribute_name = node.className;
                              int best_attribute_index = HW1.printattributes.indexOf(best_attribute_name);
                              int testrec_value  =
  Integer.parseInt(testRecord.getRecord_row()[best_attribute_index]);
                              node = node.child.get(testrec_value).childPointer;
                    }
                    String obtained_classlabel = node.classvalue;
                    return obtained_classlabel;
          }

  }

public static void main(String args[]){
          Scanner sc = new Scanner(System.in);
          Main m = new Main();
          m.addAttributeValues();
          m.rdFile();
          m.getCountAge();
          m.findInfoGain();
          m.findRelativeGain();
          m.printMap();
          m.printGainRatio();
          m.printGini();
          int X = m.countDecisions("youth", "yes");
          m.informationgainattribute();
          }
}
```

2. DecisionTreeNode.java

```
public class DecisonTreeNode {

          ArrayList<children> child;
          ArrayList<Integer> remainingattributes;
          int bestattributeindex;
          int pos_classcount;
          int neg_classcount;
          boolean isLeaf;
          String classvalue, className;
          HW1 mainClass = new HW1();

          public class children{
                    DecisonTreeCons childPointer;
                    int value;
                    children(DecisonTreeNode address, int v){
                              value = v;
                              childPointer = address;
                    }
}
```