ESE 589 Decision Tree Induction Algorithm

Fedrik Durao 103302962, Nehul Oswal 112820228

Abstract:

The Decision Tree Induction algorithm has been implemented and tested in this project. The goal was to not only successfully implement the intended behavior of the algorithm but to do so efficiently, as to achieve low latency and high resourcefulness, while also being able to observe how the algorithm's results change depending on the data it is being used on, and the metric being used on the data. The algorithm behavior and design of this implementation of the algorithm will be discussed in detail. Involving the data structures chosen and structure of the code itself. The code had been validated on simpler data tables, prior to being used on the experimental databases.

Introduction:

The Decision Tree Induction algorithm aims to scan a database, split it into a training set and testing set, and then "predict" the result of the testing set by "learning" from the training set. The algorithm has 2 distinct parts. First the implementation is trained on the training set, developing a decision tree, and then the testing set is put through the decision tree and the resulting decision/prediction is given as output.

The decision tree is created recursively, using a specific metric to evaluate the database, partition it into relatively pure partitions, and then repeating this step for each partition. For every partition made, an attribute has been chosen by a certain metric, and the partitions are based off of the values available for the chosen attribute. When partitions are all of the same resultant class, or all attributes have already been considered, then the class for that partition is concluded and returned. The algorithm is finished once every partition's class has been returned.

Each data entry in the testing set is passed through the decision tree. Every node in the tree presents an attribute that is in question, and the children of every node represent the possible values that the attribute in question can be (and also the next attribute to be questioned). So at every node, the given data entry will have a distinct attribute be evaluated, and based off the value of the data entry's attribute, the tree will be traversed and then another attribute will be evaluated. The output is a leaf in the tree path that is traversed, which is the resulting prediction of the data entry's class. For every data entry's result, it is compared to the actual class given in the data entry. The accuracy of the algorithm is recorded, and is given as output once the whole testing set has been tested.

Design of Software:

Three data structures were used to implement this algorithm. As expected, Data Trees were used to represent the decision tree (created recursively from the training set). Dictionaries or Hash Tables were used to store the database, and the children nodes within the decision tree. Dictionaries were optimal to refer to a specific item quickly, rather than searching for a given item in an array or list. And by sorting children dictionaries based on their parent's attribute

partitions, traversing the decision tree with the testing set will be faster (choosing a child takes O(1) complexity). Lists or arrays were also used for intermediate logic throughout the program.

This implementation follows a simple and intuitive program flow beginning with reading and splitting the database given. The database is split into the training set and testing set by using getLearningSetAndTestingSet( ). Then the training set is passed through gen_Tree( ) to recursively make the decision tree. Here, a new node is created for the tree. After evaluating the data, it is determined if the current node is a leaf/classifier or if it will be another question in the decision tree. If the current node is determined to be a question node, then an attribute selection method will be used. Which attribute selection method to use will be chosen at runtime. The node is renamed to be the selected attribute. Partitions of the current data set will be made from using get_partitions( ) and the results from the attribute selection method. Then if partitions are non-empty, then the children of the current node will be made from calling gen_Tree( ) again but on the partitioned data sets. Below is the corresponding pseudo-code.

-Split code into training set and testing set
-generate decision tree from training set:
    -create new node N
    -return node with a classifier if all data is of the same class
    -return node with a classifier of the majority classifier of data set if there are no more attributes in list
    -apply the chosen attribute selection metric
    -update N with the selected attribute
    -create partitions in data set based on the selected attribute
    -for every partition:
        -return node with a classifier of the majority classifier of the data set if partition is empty
        -create children for node by recursively generating a tree for the partition
    -return node N
-pass testing data entries through decision tree

1. **readData( ):** This function reads every row from the database file as a list and stores every list in a dictionary with row number as key value. As dictionary are inherently unordered therefore ordered dictionary is used so that the list is stored in the same order as they are read.
2. **getLearningSetAndTrainingSet( ):** This function splits the database from readData( ) into a training set and testing set, using a splitting factor. For this project a splitting factor of 0.7 was used. Meaning that 70% of the database is used for training, and 30% used for testing.
3. **Class Tree_Node:** is a class defined with its member variables as item name, parent_decision, split, iscontinuous, and a list of its children nodes.
4. **gen_Tree( ):** Starts off by making copies of the passed dataset and attribute list (as to pass by value later). A node is created, filling the parent_decision field of the node. Then the program checks to see if every data entry is of the same class, if so then node N is returned as a leaf with the common classifier found in the data set. Also if the attribute list passed to the function is empty, then node N is returned as a leaf with the majority

class found in the dataset from getMajorityClass( ). If neither cases are true, then the attribute selection method is called, updating the contents of node N and reducing the current attribute list. Using the selected attribute, get_partitions( ) is called to make partitions for every unique value for the given attribute. Then for every partition a new child is added to N by calling gen_Tree again, passing the current partition data, attribute list, and the unique attribute value of the partition. If any partition is empty, then a leaf child node with a classifier representing the majority class of the current data set is added to N. When done with the partitions, N is returned.

5. **getMajorityClass( ):** Receives a data set as an input. Checks the last element of every entry (which is the classifier), and adds the classifier as a key to a dictionary. For every repeat a count is stored and updated in the corresponding dictionary key. The classifier key with the largest count is returned as output.
6. **Get_partition( ):** This function is used only if the node which is going to be created in the decision tree is for an categorical attribute. This function returns a dictionary containing all the tuples which contains the attributes values passed to it as an argument. Basically, this function is used to partition the data for categorical attributes at each level of the decision tree with the attribute been partitioned removed from the dictionary of list.
7. **Get_partition_continuous( ):** This function is used only if the node which is going to be created in the decision tree is for an continuous attribute. The split midpoint is been passed as an argument to this function and depending on it two dictionaries are been created one dictionary containing all the values for which the attribute value is less than or equal to the midpoint and the other dictionary contains all the value with attribute value greater than midpoint. This function is used to partition the data for continuous attributes.

Once the decision Tree has been created, the TestWithTree( ) function is called. This is a recursion function. A testing set data entry, decision tree root, and the database's majority classifier are passed as input. The name of the current node (an attribute) is used to get the unique valued attribute of the data entry. If the data entry's unique valued attribute is found amongst the node's children, then TestWithTree( ) is called again, passing the data entry, corresponding child, and majority classifier. This is done recursively until a leaf is found returning the node name (classifier), or if the unique valued attribute cannot be found which the majority classifier is retuned.

There are 3 metrics for selecting an attribute that is to be added to the decision tree. Information gain, Gain ratio, and Gini index. Each use different formulas to determine which attribute will result in the "purest" branches.

1. **Entropy( ):** This function calculates the entropy for the partitioned dictionary of list passed to it as the argument.
2. **Continuous_info_gain( ):** This function is called whenever any attributes in the database contains continuous attributes values. This function first sorts all the attributes values corresponding to the attributes and then a list of all the unique values in the attribute is created. For the purpose of finding only unique elements "Set" data structure was used. Finally, midpoint is calculated for each adjacent unique value and two dictionaries are created one containing all the values for which the attribute value is

less than or equal to the midpoint and the other dictionary contains all the value with attribute value greater than midpoint. Information gain for each midpoint is calculated and the functions returns the midpoint with maximum information gain. This function also calculates the gain ratio for every midpoint and return the midpoint with maximum gain ratio.

3. **Info_gain( ):** This function is used whenever the attribute values are categorical. This function checks whether the given is continuous or categorical and if it is continuous then Contiuous_info_gain() function is called. This function creates a dictionary of list with dictionary key as attribute value and count for each class label corresponding to that attribute value. Using this dictionary, the function calculates the information gain for each attribute and returns the attribute and index of the attribute with maximum information gain. This function also calculates gain ratio for each attribute and returns the attribute index with maximum gain ratio. This function also returns an identifier "Iscontinuous" which is used to indicate whether the split point is continuous or categorical.
4. **splitting_index_values( ):** This function returns a list of all the unique values of the attribute for which a node in the decision tree is going to be created.
5. **Gini_index( ):** This function is used to calculate the Gini Index for categorical attributes. This function is called for every combination of unique attribute value for categorical attributes. This function iterates over each of the class label and a count of each class label for each unique values of the attributes is been created. Using this data Gini Index for each combination of attribute values is calculated.
6. **Gini_index_continuous( ):** This function is used to calculate the Gini Index for attributes with continuous values. This function first sorts all the attributes values corresponding to the attributes and then a list of all the unique values in the attribute is created. For the purpose of finding only unique elements "Set" data structure was used. Finally, midpoint is calculated for each adjacent unique value and two dictionaries are created one containing all the values for which the attribute value is less than or equal to the midpoint and the other dictionary contains all the value with attribute value greater than midpoint. Gini Index for each midpoint is calculated and the functions returns the midpoint with minimum Gini Index.
7. **get_subset( ):** This function calls the "gini_index()" and "gini_index_continuous()" function for each of the attributes in the dataset. The functions are called depending upon whether the attribute is discrete or continuous. If the attribute is categorical the function generates all the combinations for all the unique attribute value and for each combination the "gini_index" function is called. For generating the combination of the attributes "combinations" functions in "itertools" library is used. This function returns the attribute value combination with the minimum Gini Index in case of categorical data and split midpoint if the attribute with minimum Gini Index is continuous. This function also returns an identifier "Iscontinuous_gini" which is used to indicate whether the split point is continuous or categorical.

Other small portions of the code have not been discussed. These portions are mostly just band-aid logic to cover for conditions that these functions missed.

Validation:

Before taking measurements of the algorithm for different databases, the algorithm was tested on smaller more readable databases made by hand. Similar to the one used in the textbook and class examples. As changes were made to the simple database, the Decision Tree Induction was done by hand and then compared to the results for the algorithm for each three parts. Many potential errors were eliminated while changing the small database and debugging the code. But there were indeed a few more errors afterwards when working with the large legitimate databases. These errors were eliminated after using the larger databases. Most of those late errors were related to the entries being sorted by classifier, resulting with testing sets that had different classifiers than the expected ones in the decision tree. This was mitigated by randomizing the data base, so that it is unlikely that the two data sets have heterogeneous classifiers. Without a large database, it was difficult to visualize all potential problems since the algorithm could not be verified exhaustively when testing the algorithm on the small database.

Below is our results of Decision Tree Induction based on a simple database, modeled after class examples.

```
{0: ['2', 'youth', 'high', 'no', 'fair', 'no'], 1: ['1', 'youth', 'high', 'no', 'excellent', 'no'], 2: ['1', 'middle
aged', 'high', 'no', 'fair', 'yes'], 3: ['2', 'senior', 'medium', 'no', 'fair', 'yes'], 4: ['2', 'senior', 'low', 'ye
s', 'fair', 'yes'], 5: ['1', 'senior', 'low', 'yes', 'excellent', 'no'], 6: ['1', 'middle aged', 'low', 'yes', 'excel
lent', 'yes'], 7: ['1', 'youth', 'medium', 'no', 'fair', 'no'], 8: ['2', 'youth', 'low', 'yes', 'fair', 'yes'], 9: ['
2', 'senior', 'medium', 'yes', 'fair', 'yes']}
```

Figure 1: Training set

```
---
Current question is  1
Previous decision is  None
---
Current question is  2
Previous decision is  youth
---
Current question is  no
Previous decision is  high
this is the leaf result
---
Current question is  no
Previous decision is  medium
this is the leaf result
---
Current question is  yes
Previous decision is  low
this is the leaf result
---
Current question is  yes
Previous decision is  middle aged
this is the leaf result
---
Current question is  0
Previous decision is  senior
---
Current question is  no
Previous decision is  <= 1.5
this is the leaf result
---
```

Figure 2: “Depth first” printed Decision Tree

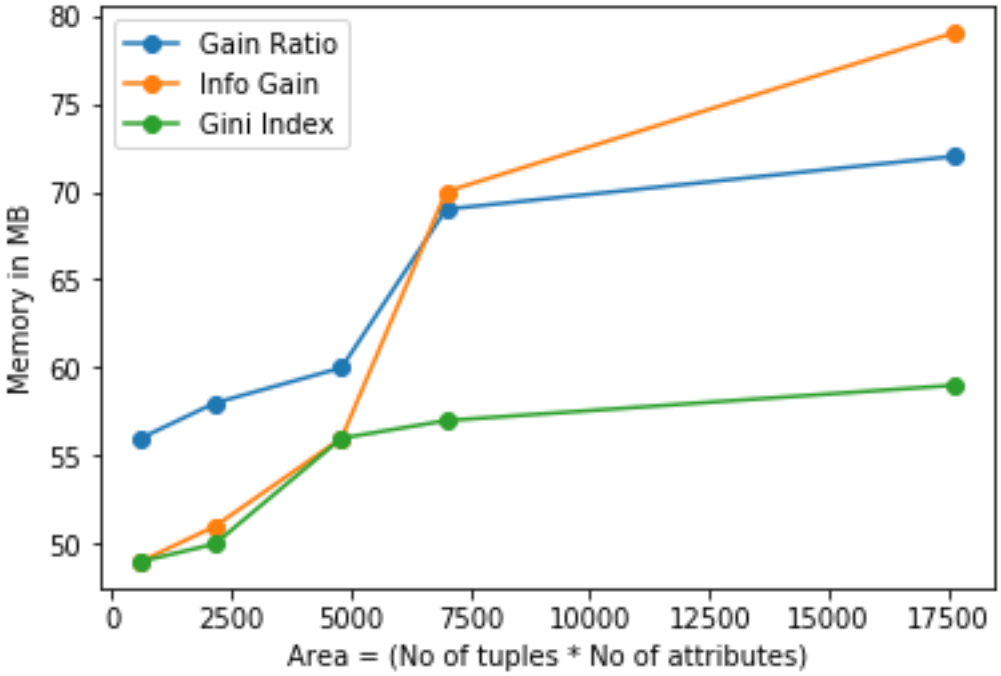
```
{0: ['1', 'youth', 'medium', 'yes', 'excellent', 'yes'], 1: ['1', 'middle aged', 'medium', 'no', 'excellent', 'yes'],
2: ['2', 'middle aged', 'high', 'yes', 'fair', 'yes'], 3: ['1', 'senior', 'medium', 'no', 'excellent', 'no']}
accuracy is  0.75
```

Figure 3: Testing set and accuracy of decision tree

Results and Analysis:

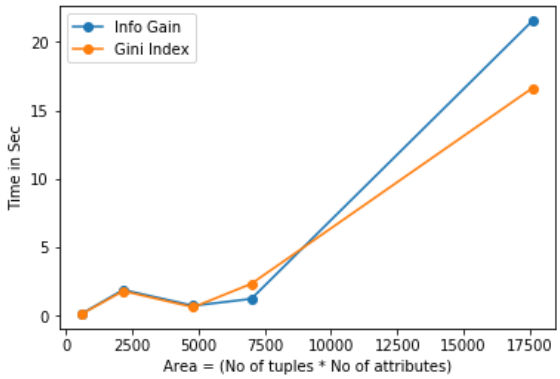
Three relationships were depicted from graphing the results of Decision Tree Induction on the seven databases: Winequality-red, Breast Cancer, Iris, Poker-hand testing, Divorce Predictor, Bank Note Authentication, and Glass. The relationships observed are execution time v. support, time v. database size, and patterns v. support. These results were produced on a laptop with 12GB of RAM and an i7 processor.

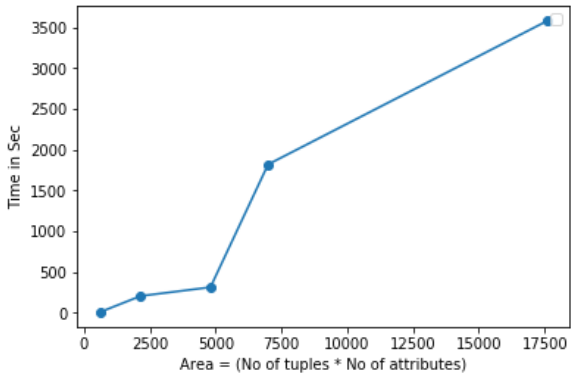
1) Memory v. Database size



The algorithm was expected to use up more memory as the size of the database increases. This is accurately described from the resulting memory usage of this implementation shown above. Although not quite linear, the memory usage is certainly proportional to the size of the data base being used. Notably, implementing either of the different attribute selection metrics has no significant effect on memory usage.

2) Time v. Database size





As expected, the runtime increases for increasing data base size. The relationship seems to be a square relation when using info gain and gini index metrics. Although the time of this implementation skyrockets when using the gain ratio metric for attribute selection. It was expected that gain ratio would take more time than info gain, considering that info gain must be calculated to progress with the gain ratio calculation. But this execution time is still too large, suggesting that many optimizations can be made to the gain ratio metric that has been implemented.

3) Accuracy

Database	Accuracy	No of Class Labels	Database Area = (No of tuples * No of Attributes)
Iris	83.41	3	600
GlassData	57.14	6	2150
Winequality-red	62.73	6	17600
Poker-hand testing	61.33	6	4800
Breast Cancer	93.25	2	6900
Divorce Predictor	90.25	2	9234
Bank Note Authentication	84.54	2	5488

Figure 4: Info Gain

Database	Accuracy	No of Class Labels	Database Area = (No of tuples * No of Attributes)
Iris	90.41	3	600
GlassData	51	6	2150
Winequality-red	56.78	6	17600

Poker-hand testing	52.44	6	4800
Breast Cancer	92.82	2	6900
Divorce Predictor	94.11	2	9234
Bank Note Authentication	82.32	2	5488

Figure 5: Gini Index

Database	Accuracy	No of Class Labels	Database Area = (No of tuples * No of Attributes)
Iris	92.3	3	600
GlassData	51.56	6	2150
Winequality-red	46.43	6	17600
Poker-hand testing	43.35	6	4800
Breast Cancer	96.65	2	6900
Divorce Predictor	96.07	2	9234
Bank Note Authentication	82.32	2	5488

Figure 6: Gain Ratio

These charts illustrate how databases with more classifier categories yield lower accuracies. This was to be expected since having more classifier categories means that there are more incorrect decisions the tree can potentially make. The differences in accuracies between different attribute selection metrics were not too large, although it was noticeable that the gain ratio had lower accuracies for larger databases, as opposed to the info gain metric. This is also expected because info gain is bias towards larger databases.

Conclusion:

Most challenges with the algorithm came from having to account for continuous data. The complexity for each of the attribute selection metrics increased when having to account for continuous data. Also, it was a difficult to decide what would be considered continuous data for our algorithm. To make it simple, all numerical data was chosen to be considered as continuous data, regardless of the data range of the numerical data. Also, to make things simpler when handling continuous data, attributes evaluated as continuous data were removed from the attribute list while making the decision tree. Having to keep the attribute, but consider ranges in which the attribute was evaluated at previously, was too difficult. So every attribute was treated as a discrete attribute when modifying the attribute list.

Before implementing the Decision Tree Induction algorithm, the value of the algorithm was not apparent to the group. Other machine learning algorithms were discussed in the course such as Neural Networks, SVM, and Bayesian Network, which seemed much more robust and sophisticated. But after implementing the Decision Tree Induction algorithm, it was clear that the algorithm can still produce good results (under certain conditions), with much lower complexity and resources.

Few aspects can be heavily improved with this implementation of the Decision Tree Induction algorithm. Certainly the effectiveness of the gain ratio needs to be improved through optimizations, so that the execution time may be reduced to satisfactory values. The way continuous data is handled can be improved, by improving how data is determined to be continuous, and handling attribute lists when certain continuous ranges have been considered. Besides these few blunders, this implementation follows the Decision Tree Induction algorithm concisely, and most of the time provides results one may expect from the Decision Tree Induction algorithm.