Project 3 Decision Tree

Name Raymond Li
ID 111009253
Due Date Dec. 9, 2020

1. Introduction

Everyday billions upon billions of decisions are made by people across the globe. These decisions could be trivial, such as deciding what to have for breakfast, or completely world-changing such as global leaders deciding to sign intercountry pacts. Some of these decisions may not be the best options, but human foresight is error prone and may lead to a decision-making process that is not optimal, or even in the best interest of the person deciding. Regardless of the magnitude or "correctness" of the choice, all decisions have a common ground in that there requires a series of conditions that somehow influence that choice. For example, if a person was deciding on what to eat for breakfast, they might consider their current health status and choose something with fruits and vegetables rather than bacon. Since some decisions have a very clear outcome based on a defined series of courses, there is no doubt engineers in this day and age would want to find a way to automate this process, which is exactly what has been done.

In 1983, Ross Quinlan developed a decision-making algorithm that was able to create decision trees to come up with relatively accurate results based on a given dataset. This algorithm made decisions based on how much information is gained the more specific the choices became. This gain is calculated by determining the amount of information in the entire dataset against the amount of information acquired from a certain class of attributes (e.g., Color could be a class with attributes blue, green, red, etc.) in the dataset. Information is calculated using the following formulas:

$$Entropy(D) = -\sum_{i=1}^{m} p_i \times log_2(p_i) \tag{1.1}$$

The "Entropy" function is essentially another information function that calculates for the information of a single attribute. p_i is the ratio of the number of a particular outcome over the total number of outcomes produced by that single attribute.

$$Info = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Entropy(D_j) \tag{1.2}$$

The information is calculated using the "entropy" of every attribute in that particular class and essentially a weighted sum is formed. The ratio calculated here is similar to the concept of p_i. It is the ratio of the total count of a certain outcome over the entire set of outcomes. The gain is finally calculated in the final formula, combining the first and second.

$$Gain = Entropy(S) - Info(D) \tag{1.3}$$

In this equation S represents the entire data set while D represents a single class of attributes. By finding the entropy of the overall dataset and subtracting the info from one of the attributes, we can find the information gain from that singular attribute. The data will then split on which ever attribute has the

highest gain. After these calculations are performed, that class essentially becomes a question where the options for the questions are the attributes (e.g., “What color?” is the question, while the attributes blue, green, red, etc. are the options the dataset has to chose from to get to an outcome). This class is then removed from the list of attributes and this list is passed down to a level lower to perform the process above. This algorithm will only stop in the following conditions:

1) All attributes in the attribute list are of the same class.
2) There are no more classes to continue refining the “questions”.
3) There are no more attributes in each class.

At each of the stopping conditions, a leaf is created with the particular outcome that is the majority of the final set of outcomes. This process is continuously repeated until the entire dataset has been processed and a fully functioning decision tree is created.

There are other methods that can be used to implement a decision tree, the ID3 algorithm is just a single method, however the other methods also follow a similar process, but with different calculations to determine what our “question” will be. The C4.5 algorithm is a derivative of the ID3 algorithm and is similar in that rather than dividing up the data set based on just the Gain function it uses a Gain Ratio and instead of “entropy” it uses “SplitInfo”. The formula could be seen in the following:

$$SplitInfo(D) = -\sum_{i=1}^{v} \frac{|D_j|}{|D|} \times log_2(\frac{|D_j|}{|D|}) \tag{1.4}$$

After calculating gain like previously, this value is then used to divide the gain.

$$GainRatio = \frac{Gain}{SplitInfo(D)} \tag{1.5}$$

This method of creating a decision tree is the benchmark for decision tree creation algorithms and is regarded as better than the ID3 algorithm in that it is able to remove some bias from the calculations. The decision to split the dataset is the same in that the algorithm will pick the class with the largest GainRatio.

The other method of creating decision trees is called the Classification and Regression Trees (CART) algorithm. Compared to the other methods, this algorithm is slightly different in that it forces that dataset to become a binary tree, rather than a tree with multiple splits. However, similar to the previous algorithms it also works with a classification method by using the Gini index. The Gini index essentially describes how “pure” data is. It uses the probability of how likely an attribute belongs to a class to determine this and similar to the previous algorithms, works in a way where rather than choosing the class with the highest attribute, we select the one with the highest purity.

This paper will discuss the implementation of a decision tree programmed using Java. The algorithm that was successfully completed in this is the ID3 version. Following the implementation, the paper will go over the results and the verification of the results of the tree. Finally, the results will be discussed in the conclusion also with areas of improvement.

2. Implementation

2.1. Data Preprocessing and Organization

One of the main pieces in any algorithm is the data that will be operated on. The organization of data is not exactly going to be the same throughout and will need to be rearranged to a certain standard. For this implementation, it was decided that the data used will only be discrete. This is so that the algorithm would not have to spend time organizing the data into bins to make it "discrete". The implementation would have been the same either way, just the continuous version would have to be processed into bins.

Another requirement for the data passed through this implementation is that the first column of the dataset must be for the outcomes (the final decisions) for the data. The other columns will all be used exclusively for different classes; this can be seen in *figure 1*. There is no limit to how many columns or rows since this implementation allows for that type of flexibility.

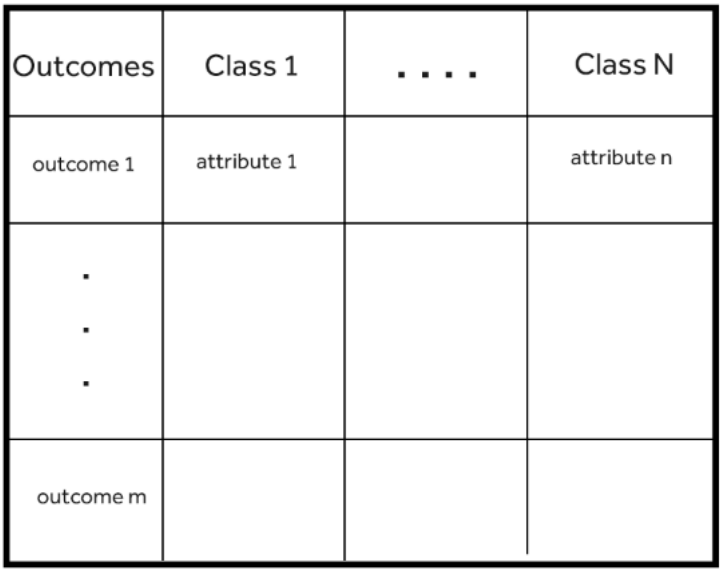

Figure 1 Visual representation of data organization

Similar to other algorithms involving large datasets, there will always be noise in data that may influence the overall final quality of the implementation. An approach to solving this issue is by implementing a minimum support function in the algorithm. This approach could be as it is done in the star-cubing algorithm, by simply replacing attributes that do not meet the minimum support with an asterisk. However, in this algorithm, I believe it may influence the final outcome of the tree, so simply ignoring these attributes may prove to be more effective and accurate when generating decisions. In addition to this, it is highly possible that attributes used in one unique class to be used in another class. In order to ensure that each attribute, depending on the class, is unique each attribute is appended with its column number.

For future testing, the dataset was divided into two unique datasets: one for training and one for testing. The decision tree will be generated using the training data, while the test data will be saved for later to compare how well the decision tree operates. *Figure 2* is the code for the method that will determine what will be training data and what will be test data. This is simply done by using the Math.random() function to decide. This method will take a random 30% of the given dataset and save that to a testData 2D ArrayList and will be removed from the original dataset. This way the tree will have no information on the remaining 30% of data.

```
/** Splits the raw data into training data and testing data. 30% of the raw data will be used for training ...*/
public static ArrayList<List<String>> splitData(ArrayList<List<String>> data){
    ArrayList<List<String>> testData = new ArrayList<>();

    for(int i = 0; i < data.size(); i++){
        if(Math.random() < 0.3){
            testData.add(data.get(i));
            data.remove(i);
        }
    }
    return testData;
}
```

Figure 2 splitData () method

Once the dataset is properly organized, the implementation of the algorithm requires a few pieces of information before it is ready to begin generating the decision tree. The first piece of information is the data itself, which has already been prepared. The second piece of information is the attribute list, which contains both the attribute along with its "Class ID" in the current set of data. The Class ID of an attribute is simply its column index of the dataset, this will be used later on to identify the class that each attribute belongs to. This is performed by the method named *getAttributeList ()*. This method takes in a 2D ArrayList, which would be the dataset, and iterates through each column of the given dataset. This method will only add to the attribute list if it finds a unique attribute, and if it does not find a unique attribute, it will ignore it and move on. The output of this method is not exactly a simple list, but it is a 2D (usually jagged) ArrayList of Pairs. The figure below is a visual representation of how the attribute list is organized, *figure 4* is the method itself.

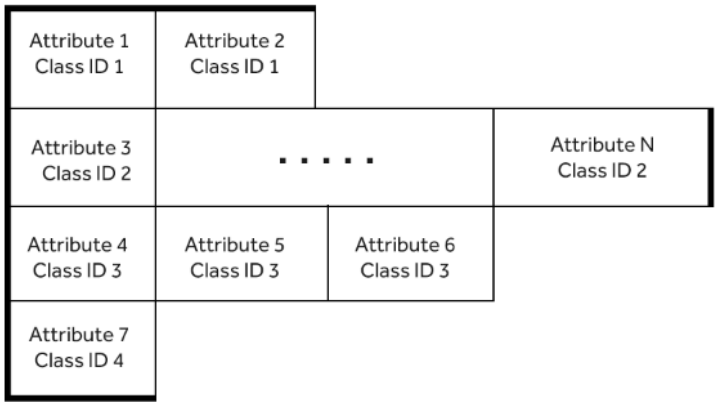

Figure 3 Visual Representation of Attribute List

```java
/** Creates a 2D ArrayList that holds all the unique attributes in a particular dimension (Class) ...*/
public static ArrayList<List<Pair>> getAttributeList(ArrayList<List<String>> data){
    ArrayList<List<String>> elements = new ArrayList<>();

    for(int attribute = 0; attribute < data.get(attribute).size(); attribute++){...}

    ArrayList<List<Pair>> temp = new ArrayList<>();
    for(int i = 0; i < elements.size(); i++){
        List<Pair> tempPair = new ArrayList<>();
        for(int j = 0; j < elements.get(i).size(); j++){
            tempPair.add(new Pair(elements.get(i).get(j), i));
        }
        temp.add(tempPair);
    }

    System.out.println("GENERATING ATTRIBUTE LIST:");

    for(int i = 0; i < temp.size(); i++){
        for(int j = 0; j < temp.get(i).size()-1; j++){
            System.out.print(temp.get(i).get(j).getKey() + ", ");
        }
        System.out.println(temp.get(i).get(temp.get(i).size()-1).getKey());
    }

    return temp;
}
```

Figure 4 getAttributeList () Method

2.2. ID3 Calculations

As mentioned before, the ID3 algorithm requires calculating for the information gain of each class to determine which class will be used to split the data on. A series of methods were created in order to perform this function. The first method in this series of methods is calculating the entropy of each individual Class in the dataset. This calculation is performed by the *getClassEntropy()* method. This method takes in a 2D ArrayList that contains information on each individual attribute. The organization and formatting of this data will be explained later. The method will loop through this ArrayList and find the entropy for each attribute and then sum it together.

```
/** Gets the entropy of an individual attribute. The information is placed back into the 2D ArrayList that is passed to it. ...*/
public void getClassEntropy(ArrayList<List<String>> attrOutCount){

    for(int i = 0; i < attrOutCount.size(); i++){
        int totalOutcomes = 0;
        ArrayList<Integer> individualOutcomeTotals = new ArrayList<>();
        double entropy = 0.0;
        //Data is formatted in the following fashion:
        //      [Attribute Name] | ...Outcomes... | Class ID
        //The result of attrOutCount will become as follows once this method is complete;
        //      [Attribute Name] | ...Outcomes... | Class ID | entropy
        //We're only interested in the outcomes.
        for(int j = 1; j < attrOutCount.get(i).size()-1; j++){
            totalOutcomes += Integer.parseInt(attrOutCount.get(i).get(j));
            individualOutcomeTotals.add(Integer.parseInt(attrOutCount.get(i).get(j)));
        }

        for(int j = 0; j < individualOutcomeTotals.size(); j++){
            String temp = String.valueOf(individualOutcomeTotals.get(j));
            double tempDouble = Double.parseDouble(temp);

            if(tempDouble == 0.0){      //if tempDouble (the numerator) is 0, this will cause issues as log2 will return -infinity
                entropy -= 0;
            }
            else{
                entropy -= (tempDouble/ (double) totalOutcomes) * log2((tempDouble/((double)totalOutcomes)));
            }
        }

        attrOutCount.get(i).add(String.valueOf(entropy));
    }
}
```

Figure 5 getClassEntropy() method

Once that is complete, the entropy is appended to each class, the information of each class must now be calculated. This is performed by the getInfo() method. Taking in the 2D ArrayList as mentioned previously, now with the entropy of each class appended, the information will now have to be calculated. This method follows the equation 1.2 as mentioned earlier. This method takes in an ArrayList that contains the total count of each of the outcomes in the given dataset. This will be used along with the 2D ArrayList for each class to determine the coefficient that each class' entropy will be multiplied by.

```
//Finds the total number of outcomes that occur with a single attribute and then divides that by the overall
//number of outcomes. This creates the coefficient to be used when calculating info.
for(int i = 0; i < attrOutCount.size(); i++){
    int totAttrOutcomes = 0;
    for(int j = 1; j < attrOutCount.get(i).size()-2; j++){
        totAttrOutcomes += Integer.parseInt(attrOutCount.get(i).get(j));
    }

    coeffs.add((double)totAttrOutcomes/(double)totalOutcomes);
}

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

    //The data is organized such that class ID is in the 2nd to last pos.
    //In addition to this, the class ID starts from 1
    int index = Integer.parseInt(attrOutCount.get(i).get(attrOutCount.get(i).size()-2));       //Attribute class ID repurposed as index value

    double temp = Double.parseDouble(attrOutCount.get(i).get(attrOutCount.get(i).size()-1));   //Save entropy value to temp variable
    attrOutCount.get(i).remove( index: attrOutCount.get(i).size()-1);                          //Remove entropy from last position, this will be recalculated
    if(info.size() < index){        //The Attribute Class ID will always be > the size of the info ArrayList if that class ID hasn't been computed yet

        //If we're on a new Attribute class, add to info.
        info.add(new Pair(String.valueOf(index), coeffs.get(i)*temp));
    }

    //If info has already been on the same Attribute Class ID, then simply add on the weight adjusted entropy.
    else{
        double infoTemp = info.get(index-1).getDoubleVal() + (coeffs.get(i)*temp);
        info.set(index-1, new Pair(String.valueOf(index), infoTemp));
    }
}

return info;
```

Figure 6 Portion of the getInfo() method

Figure 6 contains a portion of the getInfo() method that performs the calculations. The temporary ArrayList *coeffs* is used to store the total outcome count of each class. This is later used to complete the calculation. The second for-loop performs the actual calculations. Here the entropy is taken out of the inserted 2D ArrayList and saved to a temporary variable as it needs to be removed for the next iteration of the algorithm. This temporary variable is then multiplied with its respective coefficient contained in the *coeffs* ArrayList. Once it is finished computing, the value is saved to the *info* ArrayList and once the information value for all classes have been calculated then it is returned. This ArrayList will contain two pieces of information, the Class ID and the information value itself.

Before the Gain can be calculated for each class, the algorithm needs to figure out the overall information in the dataset. This method is essentially the same as the getClassEntropy() method, however, it computes it over the whole data set. The value will be taken with the information from each class and the difference of these two values will be compared for each class. This is done in the following figure which shows the code for calculating Gain. The getTotalEntropy() method is the method that will calculate the overall information in the dataset. After the calculations are complete, it is stored in an ArrayList of Pairs which will hold the Class ID and Gain value.

```java
/** Gets the gain of the data. This is done by: ...*/
public ArrayList<Pair> getGain(ArrayList<List<String>> attrOutCount, ArrayList<Pair> outcomeList){
    getClassEntropy(attrOutCount);
    ArrayList<Pair> gain = getInfo(attrOutCount, outcomeList);
    double trainingEntropy = getTotalEntropy(outcomeList);

    for(int i = 0; i < gain.size(); i++){
        gain.set(i, new Pair(gain.get(i).getKey(), trainingEntropy - gain.get(i).getDoubleVal()));
    }

    return gain;
}
```

Figure 7 getGain() method

2.3. Decision Tree Generation

The calculations done in the previous section is essentially the basis of the ID3 decision tree algorithm. In this implementation of the ID3 Decision Tree generation method, the method will take in three parameters. The first parameter is an ArrayList of *DecisionNode objects* named *current*. A *DecisionNode* is defined in the following figure.

```java
public class DecisionNode {
    private String attribute;
    private int classID;
    private ArrayList<DecisionNode> children;
```

Figure 8 Data fields of the DecisionNode class

This class contains a String data field meant to hold the attribute name, an int data field which would hold the class ID of the attribute, and finally a data field that is an ArrayList of DecisionNodes, which is meant the hold the children of the node. Once the tree is constructed, the structure, visually, will look something like the following.

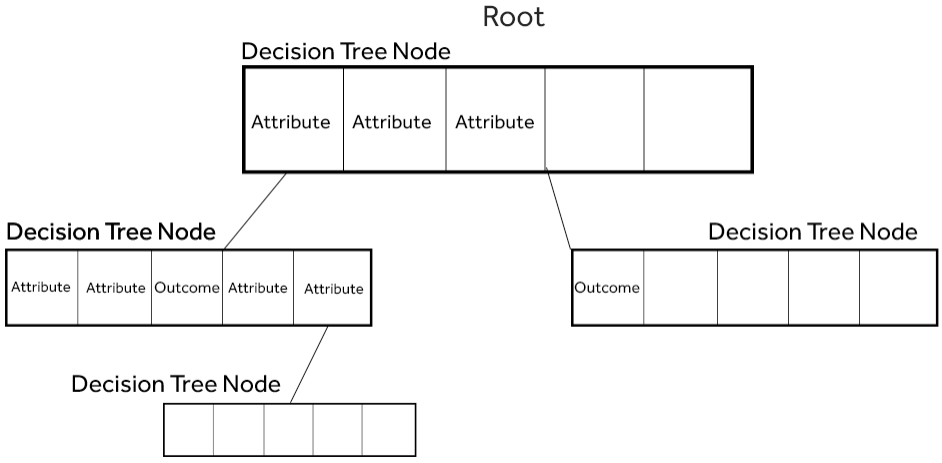

Figure 9 General Structure of Decision Tree

Each of the “attributes” and “outcomes” labeled in the figure are themselves *DecisionNodes*. The overall structure that holds each of the *DecisionNodes* labeled “Decision Tree Node” could be thought of as the actual node of the tree. The rest of the *DecisionNode* class are simple constructors, and getters and setters that will not be discussed.

The other two parameters that are passed are a 2D ArrayList of Strings which contain the training data, named *data*, to be used to create the tree and a 2D ArrayList of Pairs named *attribute*List. The *attributeList* parameter holds all the attributes that are unused at a certain decision tree node. If an attribute has been used to split the data in nodes above, those attributes will not show up in this list.

The decision tree generation method will use the *current* parameter to keep track of where it is in the tree and to manipulate the node it is on. The *data* parameter will be used to generate other pieces of data that will be necessary for the creation of the tree. Lastly, the *attributeList* parameter will help determine when to end the method. The method declaration and parameter list can be seen in the figure below.

```
public ArrayList<DecisionNode> generateTree(
        ArrayList<DecisionNode> current,
        ArrayList<List<String>> data,
        ArrayList<List<Pair>> attributeList){
```

Figure 10 generateTree() method

The method begins by first checking if the given *attributeList* is empty. If it is empty nothing will be done, and the method will be ended there. However, if this condition is passed, this means that there is more to be done.

```
if(!attributeList.isEmpty()){
    if(attributeList.size() == 1){
        ArrayList<Pair> outcomeList = getOutcomeList(data);
        int index = getLargestIndex(outcomeList);

        current.add(new DecisionNode(outcomeList.get(index).getKey(),  classID: 0,  children: null));

        return current;
    }
```

Figure 11 Base condition where all attributes are of the same class

The second check performed will check if the *attributeList* size is 1. This will be the stopping condition where all attributes in the attribute list are of the same class. Since this is a 2D ArrayList, this check is actually checking for if the number of classes in the list are 1. If this is the case, then this entire class automatically gets set as a leaf in the current node. This is performed by

first acquiring the list of outcomes that are in the current set of data. The method *getOutcomeList()* will take in the current iterations data and output the outcomes along with its count.

```
/** Finds all the possible outcomes in the data. This assumes that the outcomes the user is looking for is placed ...*/
public ArrayList<Pair> getOutcomeList(ArrayList<List<String>> data){
    ArrayList<Pair> outcomeList = new ArrayList<>();

    for(int i = 0; i < data.size(); i++) {
        boolean foundMatching = false;
        for(int j = 0; j < outcomeList.size(); j++){
            if(outcomeList.get(j).getKey().equals(data.get(i).get(0))){
                foundMatching = true;
                int temp = outcomeList.get(j).getValue() + 1;
                outcomeList.get(j).setValue(temp);
                break;
            }
        }

        if(!foundMatching){
            outcomeList.add(new Pair(data.get(i).get(0), 1));
        }
    }
    return outcomeList;
}
```

Figure 12 getOutcomeList() method

Figure 12 details the code describing the getOutcomeList() method. The outer for loop in this method will loop through *data*'s rows while the inner loop will loop through the *outcomeList* to check for a unique outcome. If it is not unique, then that particular outcome's index will be incremented, but if it is unique a new pair is added to the data and is given the attribute name and a value of 1. Once this is complete, the *outcomeList* is returned.

When the outcome list is obtained, it is necessary to figure out which of the outcomes are higher in value. It is not always the case that the final leaf in the tree will be simply one outcome or another, this will decide based off majority. The method getLargestIndex() will find the outcome pair that contains the largest value and return its index. This method is rather simple and will not be discussed as all it does is loop and compare. Once this is all complete, the current node is added with the outcome of the largest size after being provided the index of it.

```
if(current == null){            //if current node is null
    ArrayList<Pair> outcomeList = getOutcomeList(data);

    if(outcomeList.size() == 1){        //if there is only one outcome left in the list, it becomes a leaf
        current = new ArrayList<>();
        current.add(new DecisionNode(outcomeList.get(0).getKey(),  classID: 0,  children: null));

        return current;
    }
```

Figure 13 Base condition where a single outcome remains

After those checks are performed, comes another series of checks. The first overall check ensures that the current node that the method is on is null. This is done because each pass in this algorithm will fill an ArrayList, if the current node is not null, this means we have to skip the node. Once again the outcome list is prepared for this iteration so that the method is able to perform calculations. The outcome list is first checked for its size, this is because the base condition of this algorithm is that if only a single outcome remains, then that node automatically becomes a leaf of that outcome.

```
//Performing calculations on given data
ArrayList<List<String>> attrOutCount = getAttrOutCount(data, outcomeList);
ArrayList<Pair> gain = getGain(attrOutCount, outcomeList);          //Gets Gain of all Attribute Classes
Pair newNode = findLargestPair(gain);                               //Finds highest gain
int classID = Integer.parseInt(newNode.getKey());                   //Gets Class ID of highest gain
int index = -1;

//Searches in Attribute List for Class with Highest gain (may not be the same after first iteration)
for(int i = 0; i < attributeList.size(); i++){
    if(attributeList.get(i).get(0).getValue() == classID){
        index = i;
        break;
    }
}
```

Figure 14 Gain Calculations

If that check is passed, the algorithm can continue as normal. The first thing the algorithm performs is a rearrangement of the *data* 2D ArrayList. The method *getAttrOutCount()* is called, and counts all of the outcomes for each individual attribute.

```
for(int i = 0; i < outcomeList.size(); i++){
    outcomes.add(outcomeList.get(i).getKey());
}

//Loop through data arraylist
for(int i = 1; i < data.get(0).size(); i++){            //iterate through columns
    for(int j = 0; j < data.size(); j++){               //iterate through rows
        boolean foundMatching = false;

        for(int k = 0; k < outcomeCount.size(); k++){
            if(outcomeCount.get(k).get(0).equals(data.get(j).get(i))){
                foundMatching = true;
                int outcomeIndex = outcomes.indexOf(data.get(j).get(0));
                int countTemp = Integer.parseInt(outcomeCount.get(k).get(outcomeIndex+1)) + 1;

                outcomeCount.get(k).set(outcomeIndex+1, String.valueOf(countTemp));
            }
        }
```

Figure 15 First portion of getAttrOutCount()

This method will first take all the outcomes and place it into a more basic ArrayList so it is easier to find the index that it is in. Following this is the method iterating through every attribute in the 2D ArrayList. The third nested for-loop in this method checks if the attribute already exists. If it does, then it simply increments the count of that outcome for the attribute, if it does not the method will add in a new List for the given attribute, the new List is filled with the name of the attribute and zeros unless the outcome index matches the looping index.

```
if(!foundMatching){
    int outcomeIndex = outcomes.indexOf(data.get(j).get(0));  //Figure out which outcome corresponds to the attribute
    List<String> temp = new ArrayList<>();
    temp.add(data.get(j).get(i));                             //Add attribute that did not exist to a temp list
    for(int k = 0; k < outcomes.size(); k++){                 //Create the formatted attribute list data
        if(k == outcomeIndex){
            temp.add("1");                                   //Set count of the outcome index to 1
        }
        else{
            temp.add("0");                                   //outcome of the outcomes will be set to 0.
        }
    }
    temp.add(String.valueOf(i));
    outcomeCount.add(temp);                                   //Add formatted data to outcomeCount list
}
```

Figure 16 Second portion of getAttrOutCount()

After this method is complete it should generate a new 2D ArrayList to be used by the getGain() method. The 2D ArrayList is formatted in a specific way where the attribute name is the leading element. Following the attribute name are outcomes of arbitrary numbers, and lastly the class ID. *Figure 17* visually displays this layout of the *attrOutCount* 2D ArrayList.

attrOutCount Format

[Attribute Name]	[Outcome 1]	[Outcome 2]	· · · ·	[Outcome N]	[ID]

Figure 17 attrOutCount Formatting

This format is only modified temporarily when it is passed through the getClassEntropy() method. When *attrOutCount* is passed to it, the entropy of the class is appended to the end of this format. This was done for simplicity and recycling of the ArrayList.

After this is completed, and the *attrOutCount* variable is obtained, the method begins to find the Gain for each class. The method *getGain* is called and is performed as described previously. In order for this algorithm to function, it needs to search for the class with the largest gain and set it as the current node. This is performed using the findLargestPair() method. This method is exactly the same as the getLargestIndex() method, however, returns a pair and takes in

a different parameter. Once again this method will not be discussed due to its simplicity. Once this is done, the algorithm needs to figure out the index of the class of attributes in the attribute list is the largest. This is done by simply looping through the attribute list after obtaining the class ID performed in the prior step.

```
current = new ArrayList<>();
//The Class with the highest gain is set as a node in the tree
for(int i = 0; i < attributeList.get(index).size(); i++){
    current.add(new DecisionNode(attributeList.get(index).get(i).getKey(), classID,  children: null));
}

//Remove the attribute with highest gain from attribute list
attributeList.remove(index);

//Clone ArrayList to be used further down the tree
ArrayList<List<Pair>> newAttributeList = cloneArrayPair(attributeList);
```

Figure 18 Node creation and attribute list removal

Once the class with the largest gain value is obtained, is it subsequently placed into the current node. After this is done the attributes that have been placed into the node have to be removed, this is done immediately after. This is now a new attribute list that must be pass down and modified by other nodes down the tree. This attribute list is then cloned using the method *cloneArrayPair*() and placed in the 2D ArrayList of pairs named *newAttributeList*. This ArrayList will be used as the parameter for *attributeList* in the recursive call. The method *cloneArrayPair()* is a simple loop that creates new objects and places them in a new ArrayList and is not worth covering.

```
for(int i = 0; i < current.size(); i++){
    //Remove the class that is begin used to split data
    ArrayList<List<String>> extractedData = extractAttribute(data, current.get(i).getAttribute(), current.get(i).getID());

    if(extractedData.size() == 0){
        index = getLargestIndex(outcomeList);
        current.set(i, new DecisionNode(outcomeList.get(index).getKey(),  classID: 0,  children: null));
        continue;   //Skip current node (since its a leaf)
    }

    //Recursively adding nodes
    current.get(i).setChildren(generateTree(current.get(i).getChildren(), extractedData, newAttributeList));
}
```

Figure 19 Final Base Condition and Recursive Call

In *Figure 19*, the final portion of the code is displayed. This part of the code will loop through the current ArrayList and perform a recursive call while setting the children of each node. Before this, the data needs to be modified to accurately represent the splitting of data at each attribute in the node. The method *extractAttribute*(), which may be more appropriately named *removeAttribute()*, removes the current attribute that it is iterating on from the given training data.

This will create a new 2D ArrayList object and pass it down for similar use in its children. In the case that the *extractedData* is empty, meaning that the current attribute is the entirety of the remaining data, then that attribute will be converted to a leaf node. Once that is complete, that is the end of the entire generateTree() method, which gives us the decision tree. The following image details a very rough idea of the generateTree() method.

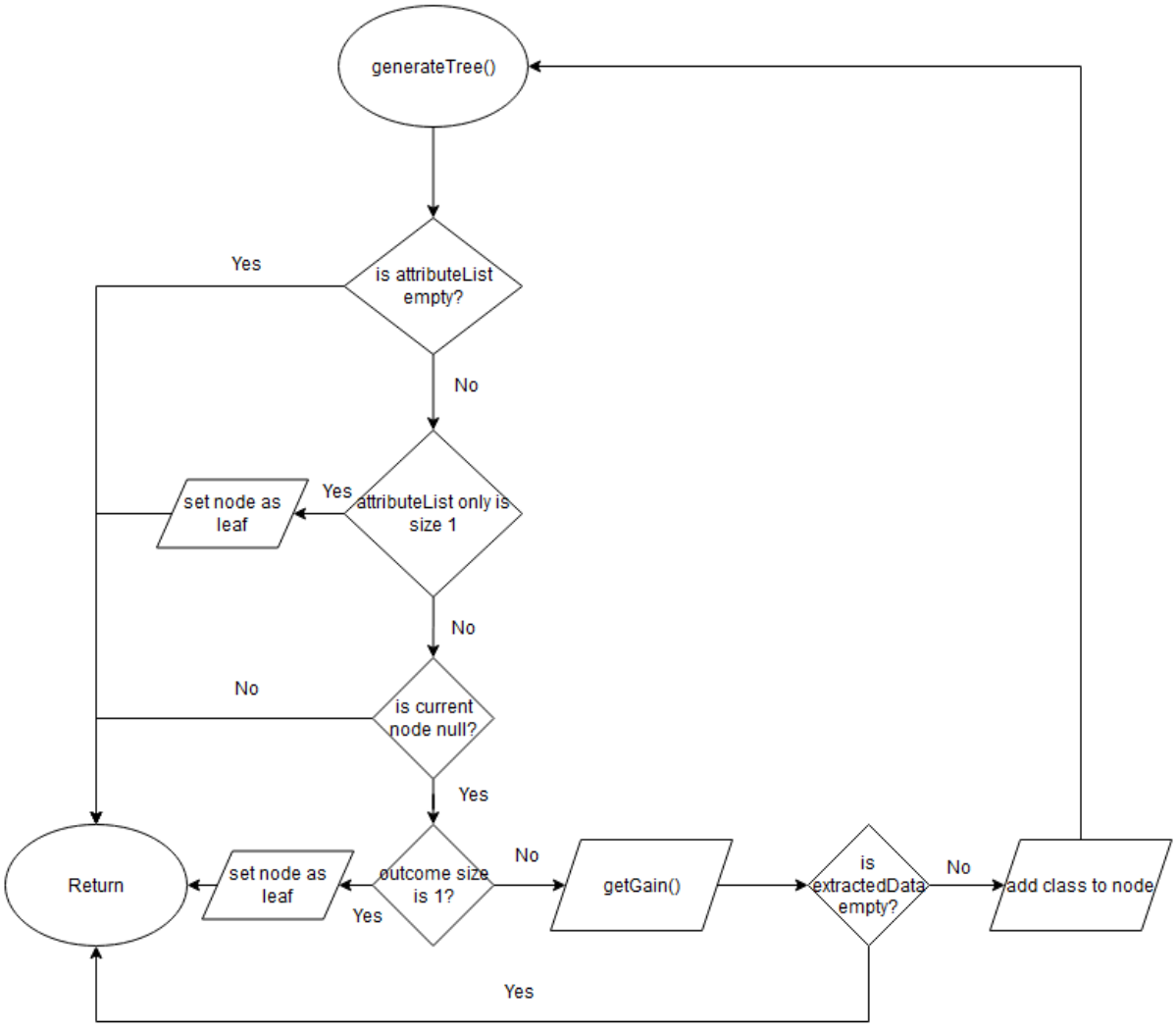

Figure 20 General Structure of the generateTree() method

3. Verification and Results

To verify if this implementation is correct, a simple dataset will be used to manual create a decision tree and be compared against the automated version. This dataset consists of weather conditions for a particular day, with the outcome being if tennis should be played. If the weather conditions are favorable, then the person will play tennis. If not, then the person will not play tennis. The dataset can be seen below in *figure 21*.

Play Tennis?	Outlook	Temperature	Humidity	Windy
no	sunny	hot	high	weak
no	sunny	hot	high	strong
yes	overcast	hot	high	weak
yes	rainy	mild	high	weak
yes	rainy	cool	normal	weak
no	rainy	cool	normal	strong
yes	overcast	cool	normal	strong
no	sunny	mild	high	weak
yes	sunny	cool	normal	weak
yes	rainy	mild	normal	weak
yes	sunny	mild	normal	strong
yes	overcast	mild	high	strong
yes	overcast	hot	normal	weak
no	rainy	mild	high	strong

Figure 21 Simple Testing Dataset

The first thing that could be done with this dataset is to find the entropy of it all. This could be done using equation 1.1 in the introduction. We note that there is a total of 14 entries in this dataset, along with 9 yes' and 5 no's. With the information we have, we can now calculate the overall entropy as follows.

$$Entropy(data) = -\frac{9}{14} \times log_2\left(\frac{9}{14}\right) - \frac{4}{14} \times log_2\left(\frac{4}{14}\right) = 0.94$$

The next thing to be done is to find the entropy for each individual attribute. First, we can take a look at Outlook. There is a total of 5 Sunny's with 2 yes' and 3 no's. Once again using the same equation as for the entire dataset we get the following.

$$Entropy(sunny) = -\frac{2}{5} \times log_2\left(\frac{2}{5}\right) - \frac{3}{5} \times log_2\left(\frac{3}{5}\right) = 0.971$$

Once the same is done for the other attributes, the entropy of overcast evaluates to 0, and the entropy of rainy evaluates to 0.971. Using this information, we can find the information of the entire class. We know that the total number of Sunny's is 5, the total number of Overcast's is 4,

and the total number of Rainy's is 5. Using this information, we now can begin calculating information using equation 1.2.

$$Info(Outlook) = \frac{5}{14} \times Entropy(Sunny) + \left(\frac{4}{14}\right) \times Entropy(Overcast) + \left(\frac{5}{14}\right) \times Entropy(Rainy)$$

Once this equation is calculated out, the information of the Outlook class evaluates to 0.693. After subtracting this value with the overall entropy of the data, we obtain an information gain of 0.247. This calculation must be done for every attribute in every class. *Figure 22* is a table of the gain of all classes in this simple dataset.

Class	Gain
Outlook	0.247
Temperature	0.02894
Humidity	0.15155
Windy	0.047841

Figure 22 Gain of All Classes in Simple Dataset

At this point, we can choose which node to be our first node in the decision tree by selecting the class with the max gain. In this case, the class with the max gain is Outlook. We would now expect the decision tree to look like the following.

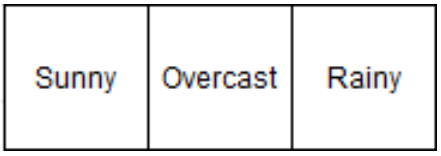

Figure 23 Initial Node of the Decision Tree

Once that is complete, we can divide the dataset into three. One dataset that only has the sunny attribute, another that has the rainy attribute and another that has the overcast attribute. In *figure 24*, we can see that every day with the Overcast attribute, the person plays tennis. In this case we can immediately set the child to overcast to yes. However, the other attributes must be divided further to finish the algorithm.

	Play Tennis?	Outlook	Temperature	Humidity	Windy
Only Sunny	no	sunny	hot	high	weak
	no	sunny	hot	high	strong
	no	sunny	mild	high	weak
	yes	sunny	cool	normal	weak
	yes	sunny	mild	normal	strong
Only Rainy	yes	rainy	mild	high	weak
	yes	rainy	cool	normal	weak
	no	rainy	cool	normal	strong
	yes	rainy	mild	normal	weak
	no	rainy	mild	high	strong
Only Overcast	yes	overcast	hot	high	weak
	yes	overcast	cool	normal	strong
	yes	overcast	mild	high	strong
	yes	overcast	hot	normal	weak

Figure 24 Simple Dataset Divided on Outlook

At this point the algorithm must continue and perform the exact same procedures for the Only Sunny section and Only Rainy sections. In order to skip the mundane calculations for the gains for each section and each attribute can be seen below.

Only Sunny	
Class	**Gain**
Temperature	0.571
Humidity	0.94
Windy	0.020022

Only Rainy	
Class	**Gain**
Temperature	0.420022
Humidity	0.020022
Windy	0.971

Figure 25 Information Gain for Only Sunny and Only Rainy Subsets

Humidity and Windiness are the two classes that has the most information gain in the Only Sunny and Only Rainy tables, respectively. Referring back to *figure 24*, we can see if we look at only humidity, for the Only Sunny table, it matches the Play Tennis? column perfectly. This is the same with the Rainy table, but for the Windiness Class. Since the base conditions have all been satisfied, the algorithm is complete and the final manually created decision table will look like the following.

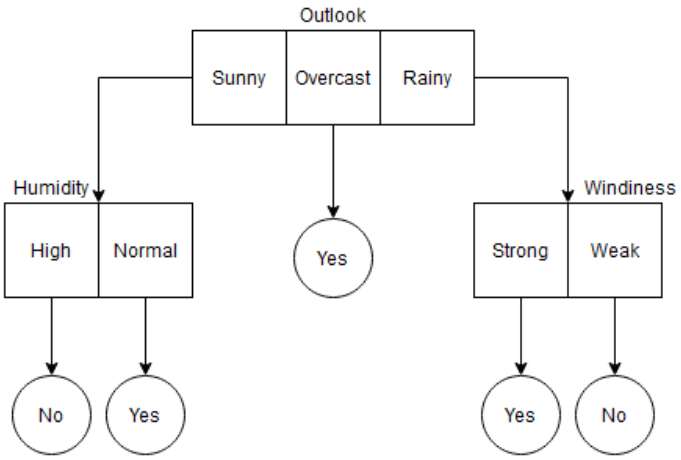

Figure 26 Manual Decision Tree

The decision tree generated by the Decision Tree algorithm implementation does not look as visually clean but can be seen in *figure 27*. In the top-most level that corresponds with the Outlook class in *figure 27*, we can see that the top-most node contains sunny, overcast, and rainy, which is exactly as in the manual tree. Under Sunny, we have the attributes High and Normal, which once again corresponds to the manual decision tree, in addition the outcomes are identical. Looking over at the Rainy attribute, we can see that its child is Strong and Weak, which both correspond to the Windiness class. Once again, all outcomes and attributes match. Although they are visually different, they are the exact same trees.

```
sunny1
    ├──high3
    │   └──no0
    └──normal3
        └──yes0
overcast1
    └──yes0
└──rainy1
    ├──weak4
    │   └──yes0
    └──strong4
        └──no0
```

Figure 27 Automatically Generated Decision Tree

To test this algorithm further, a larger dataset could be tested. Once again, like the Star Cubing algorithm and the Frequent Pattern Tree generation, the mushrooms data will be used again. The outcomes that each mushroom can be is either edible or poisonous. The tree generated for this can be seen below.

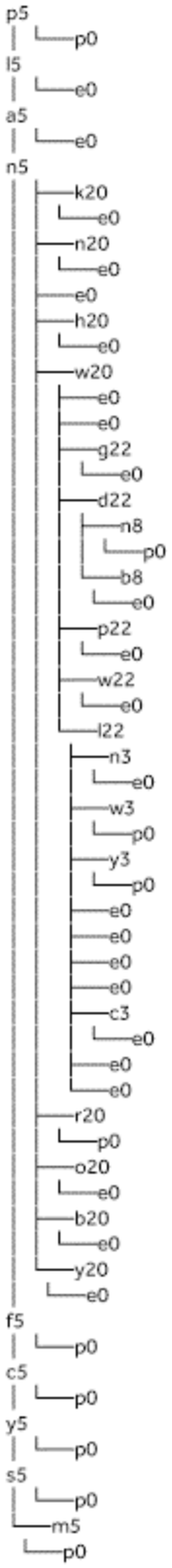

Figure 28 Decision Tree Generated by Mushrooms Dataset

As mentioned before, the implementation will pass in a set of test data extracted from the original dataset. The method that performs this operation is called testTree(). This method is essentially a recursive traversal of the tree given the root node and a single data entry, which in this context, is a single mushroom. The method that does this can be seen below.

```
public void testTree(ArrayList<DecisionNode> current, List<String> testData){
    if(current == null){
        return;
    }

    if(current.get(0).getAttribute().equals(testData.get(0))){
        System.out.println("Found Match!!!");
        System.out.println("Test Data: " +testData);
        System.out.println("Expected: " +testData.get(0));
        System.out.println("Result: " +current.get(0).getAttribute());
        counter++;
        return;
    }

    //Get the id of the current node, this will double as the index for the attribute in the test data
    int classID = current.get(0).getID();
    String attr = testData.get(classID);
    int index = -1;

    //Find the index of the attribute in the current node
    for(int i = 0; i < current.size(); i++){
        if(current.get(i).getAttribute().equals(attr)){        //If the attribute was found in the current node
            index = i;
            break;
        }
    }

    if(index == -1){
        return;
    }

    testTree(current.get(index).getChildren(), testData);
}
```

Figure 29 testTree() method

This method simply checks the first column of the data since this index is reserved for the outcome. In the case that it does find a match, then a counter is incremented, however if it is not, then the function returns without doing anything. The results of this method can be seen in the following figure.

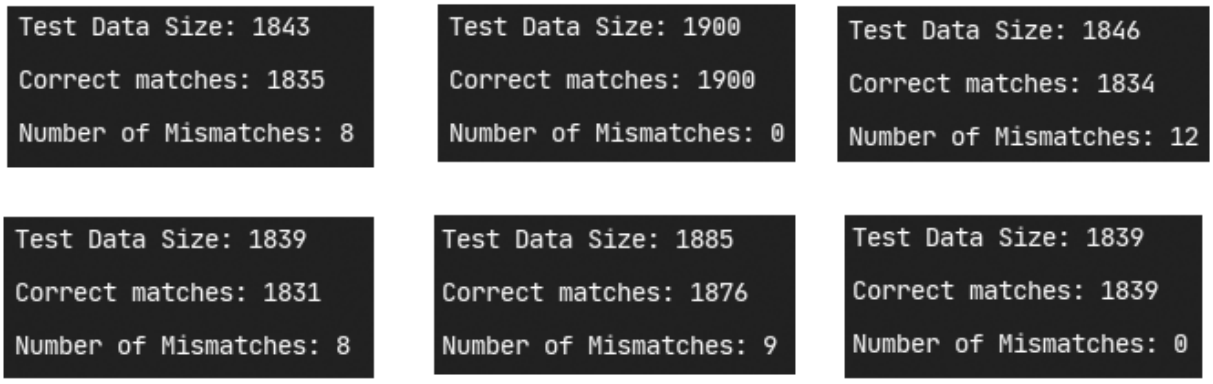

Figure 30 Six Test Data Results

A total of six executions were performed using this dataset. In *figure 30* it could be seen that the decision tree generated in each execution were not the same and that sometimes it could not create a perfect tree that would predict the correct results. This is anticipated as the ID3 algorithm does no pruning or adjusts for any biases.

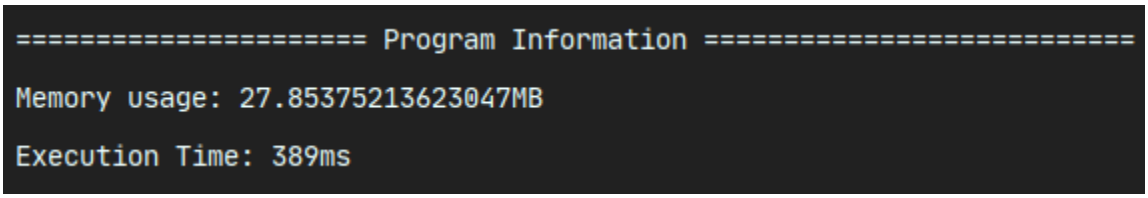

Figure 31 Addition Program Information

4. Discussion

Decision tree algorithms are extremely useful in that it has the ability to predict outcomes from unknown sets of transactions after being trained with a sample size of data. The benefits of the decision tree algorithm could be seen in the most recent technological trend of artificial intelligence and machine learning. These algorithms could assist self-learning machines on deciding what to do next depending on the inputs it gathers from either human sources or sensors. As technology continues to improve, decision algorithms could only get more accurate in predicting outcomes and its uses and implementations are essentially boundless. In this very basic implementation of the Decision Tree generation algorithm, we can see that this implementation is rather accurate, even considering it only uses the ID3 algorithm. After multiple runs, the highest error percentage of mismatched is less than 0.01%. Although this is quite a low percentage, it might still be pretty alarming considering how the mushroom dataset is attempting to determine if a mushroom is edible or not.

The issue with this implementation of the algorithm is that it uses the ID3 algorithm, which is known to be biased and performs no pruning to the tree. Something that could be done to mitigate this is to install a minimum support for the attributes. It is possible for a very small set of attributes to completely sway the tree and thus result in more inaccurate results. In addition to this, it would most likely be a lot more beneficial to use more modern decision tree generation algorithms like C4.5 and CART. The main difference between these algorithms is the way the splitting data is chosen. The C4.5 and CART algorithms both are able to adjust their values in order to reflect the data more accurately in comparison to the ID3 algorithm. However, this is essentially the only difference and the remainder of the algorithm is the same. The CART algorithm is notably different in that it forces the data to be a binary tree whereas both ID3 and C4.5 do not.

Although I consider the implementation of the ID3 algorithm, for the most part, complete, there are definitely areas in the code that could be improve upon. For instance, there is a part of the code that requires three nested for-loops in order to process data. This would definitely cause the code to be a lot slower than it could be, especially when processing large amounts of data. In addition to this, as previously mentioned, the input data into the implementation could definitely be improved upon. The mushrooms data is completely unprocessed and for a possibly better result, could be normalized to reflect the data better. There are some attributes in the data set that have less than 1000 entries and maybe even 100 entries. These could either be normalized or pruned out in order for a better decision tree. Along with this, this project could have been fully completed if there was more time, however, due to some circumstances it was only completed to this extent. Regardless of the completion of the project, I believe this is a decent implementation of generating a decision tree.

5. Notes

All equations were referenced from the following:

- Han, J., Kamber, M., & Pei, J. (2012). *Data mining: Concepts and techniques, third edition* (3rd ed.). Waltham, Mass.: Morgan Kaufmann Publishers.