Decision and Classification Trees

By: Adam Hill, Brian Ewanyk, Anton Medvedev (CSE4081 Team E)

# Introduction

We will investigate the application of decision tree induction algorithms as they are used to evaluate and classify individuals for bank loan qualification. Known classification data with multiple discrete and continuous attributes will be presented to the tree. Unknown targeted data will then be evaluated to make a prediction of an individual’s qualification for a bank loan based on the known case data. The accuracy of the predictions and efficiency of the algorithms employed will then be evaluated.

# Overview

Decision Tree Classification (DTC) algorithms, using a tree-like graph model, have been developed to make a prediction based on known data. By analyzing the Iterative Dichotomiser 3 (ID3) decision tree induction algorithm we will evaluate the efficiency of a DTC algorithm given different set conditions. The algorithm will be able to calculate a reference index (Gini Index), indicating information gain, and determine best split information based on a Gain Ratio index. The performance will be evaluated on accuracy, based on randomly generated learning data, and test sets composed of randomly generated and edited test data. The efficiency of the individual algorithms will be evaluated based on average run-time and memory usage data gathered through program profiling.

# Background

Several decision tree induction algorithms, based on Hunt’s algorithm (Kumar, Steinbach and Tan 152), have been developed to improve the efficiency and accuracy of decision tree construction. Hunt’s 1966 work to replace human problem solvers with computer programs utilizing inductive reasoning (Hunt, Marin and Stone) was investigated, refined, and built upon by Quinlan, resulting in the development of the ID3 algorithm in 1986. Quinlan went on to revise the ID3 algorithm, attempting to optimize it for classification, which eventually resulted in the development of the C4.5 algorithm in 1993. The ID3 algorithm can be utilized employing information gain or entropy evaluation during its’ tree development phase. Test data is then applied to the tree to determine a probable outcome.

# GINI Index

The GINI Index and its’ generated GINI mean values are used to estimate the information gain which would be achieved when making a decision based on a particular variable in a system. By calculating the probability of a result given a decision based on a variable, and comparing it to the probabilities available through the usage of other variables, the variable which will provide the most information is determined. For this evaluation, based on the process used to decide loan eligibility, Marital Status (M), Homeowner Status (H), Annual Household Income (I), and Loan Default Status (D) were used in training and testing. The homeowner status and default status variables are boolean values, having only True and False as possible outcomes. Marital status has a domain of {Single, Married, Divorced} and annual income is a integer value representing $1000 increments.

Figure : Information gain, mean, and index calculation

To calculate the GINI index and mean values at any point in the decision tree construction first the index of the system is determined by calculating the product of the probable outcomes of all objects at that point in the system. This index value is then used in the individual index values for each variable. The GINI mean for each attribute is calculated through the summation of the probability of any given outcome for that variable and its’ probability of default. The index for the attribute is defined as the difference between the system GINI index and the attribute’s GINI mean. The attribute index with the highest value indicates the highest possible information gain when that attribute is used to make a decision within the system.

For this evaluation, any splits based on annual income were treated as a binary divergence within the tree. In order to accomplish this values for the annual income attribute were grouped in $10,000 increments. The mean for each interval was calculated, and the interval with the lowest mean was used as a pivot point for the index calculation. The mean was then calculated for all values so that includes all values from zero to the upper limit of the selected interval, and includes all values above the selected increment.

# ID3 Algorithms

The ID3 algorithm takes a set of data and iterates through every unused attribute in the set on each algorithm execution. At each unused attribute the entropy or information gain of that attribute is calculated. Finally the attribute with the lowest entropy or largest information gain is selected by the algorithm and used to split the data set, acting recursively on the unused attributes of the two created subsets. This recursion continues until every element in the created subset belongs to the same class, there are no more unused attributes to be selected, or an empty subset is created.

Divide and conquer method is the main idea behind Decision Classification Tree. Such algorithm calls itself in a greedy manner until one of three base cases is achieved:

* If all remaining elements in a subset belong to the same class, the node is marked as a leaf and labeled with the attribute of the remaining elements.
* If there are no more unused attributes, the node is marked as a leaf and labeled with the most common attribute in the subset.
* If an empty subset is generated then a leaf is created which is labeled with the most common attribute found in members of the parent set.

Figure 1: ID3 generation pseudo-code

Figure : ID3 Tree Development Pseudocode

TreeGrowth(E,F)

if stopping\_cond(E,F) = true then

lead = createNode();

leaf.label = Classify(E);

return leaf;

else

root createNode();

root.test\_cond = find\_best\_split(E,F);

let V = {v|v is a possible outcome of root.test\_cond};

for each v that is an element of V do

E(v) = { (e | root.test\_cond(e) = v) and (e that is an element of E)}

child = TreeGrowth(E(v), F);

add child as descendent of root and label the edge (root -> child) as v;

end for;

end if;

return root;

# Train and Test Data

Separate Data Generation Tool was designed to obtain randomized data sets to train and test the algorithm. The following data, when applied, creates a tree classification structure.

Figure : Testing Data

Tid HomeOwner MaritalStatus AnnualIncome DefaultedBorrower

1 false Divorced 133 ?

2 false Single 67 ?

3 false Married 18 ?

4 false Single 147 ?

5 false Single 154 ?

6 true Divorced 0 ?

7 true Divorced 180 ?

8 false Divorced 117 ?

9 true Married 66 ?

10 true Married 11 ?

11 false Divorced 136 ?

12 false Married 108 ?

13 false Married 154 ?

14 false Married 41 ?

15 true Married 97 ?

16 false Single 98 ?

17 false Married 60 ?

18 true Single 18 ?

19 true Single 117 ?

20 true Married 33 ?

21 true Divorced 63 ?

22 true Single 124 ?

23 true Divorced 127 ?

24 true Single 173 ?

25 true Single 126 ?

26 true Single 24 ?

27 false Single 76 ?

28 false Single 92 ?

29 true Single 18 ?

30 false Divorced 122 ?

31 false Married 153 ?

32 true Single 145 ?

33 false Married 94 ?

34 false Divorced 65 ?

35 false Married 158 ?

36 false Single 69 ?

37 true Married 182 ?

38 true Married 133 ?

39 true Single 185 ?

40 false Divorced 188 ?

41 false Married 116 ?

42 true Married 16 ?

43 false Single 97 ?

44 false Single 120 ?

45 false Single 23 ?

46 true Divorced 37 ?

47 true Single 173 ?

48 true Divorced 199 ?

49 true Single 113 ?

Figure : Training Data

Tid HomeOwner MaritalStatus AnnualIncome DefaultedBorrower

1 false Single 197 true

2 true Married 32 true

3 true Divorced 13 true

4 true Married 78 true

5 false Single 47 true

6 true Married 112 false

7 false Divorced 80 true

8 false Single 108 true

9 false Single 80 false

10 true Divorced 132 true

11 true Divorced 184 false

12 true Single 130 true

13 false Divorced 12 false

14 true Divorced 99 false

15 true Divorced 169 true

16 true Divorced 164 false

17 false Single 160 true

18 true Single 151 false

19 false Single 177 false

20 false Single 87 false

21 false Divorced 88 true

22 true Divorced 22 true

23 true Married 27 false

24 true Married 62 true

25 true Married 186 false

26 false Divorced 77 false

27 true Single 69 true

28 true Single 30 false

29 true Married 185 false

30 false Single 47 true

31 true Divorced 123 true

32 false Single 37 true

33 false Married 50 true

34 true Divorced 81 true

35 false Divorced 81 false

36 false Single 70 false

37 true Divorced 42 false

38 false Married 31 false

39 false Divorced 187 true

40 true Married 45 true

41 true Divorced 12 false

42 true Married 127 true

43 false Single 95 true

44 false Divorced 128 true

45 true Single 30 false

46 false Married 197 true

47 true Married 1 false

48 true Single 187 true

49 true Single 109 false

Figure : Exemplary decision classification structure



# Complexity

Learning complexity is determined by possible variables of each attribute. Thus:



where:







After testing algorithmic performance, following data was conducted that shows linear learning curve.

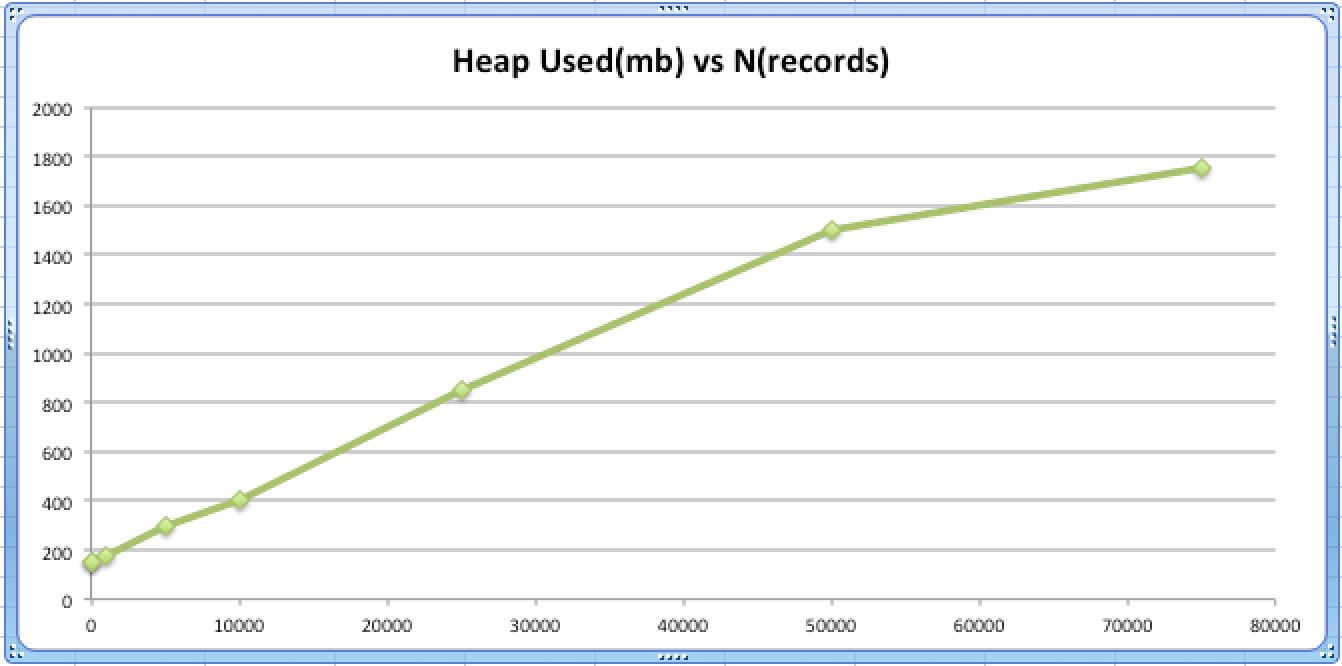


Figure : Heap size vs Number of records for the training phase

Testing complexity is determined by processing each test record and height of the tree. Thus:



where:





# Profiling

Profiling was an interesting aspect of this project because IDE’s like “Eclipse” and “IntelliJ” do not give you much stack space to work with. Thus in order to pass an issue of stack overflow, space was manually allocated. Figures 7, 8, 9, and 10 all show graphs of the heap memory used vs time for different problem sizes of size n. You can see that that space complexity for all these large problem sizes stays linear through the entire run-time. The profiler used was “YourKit.”

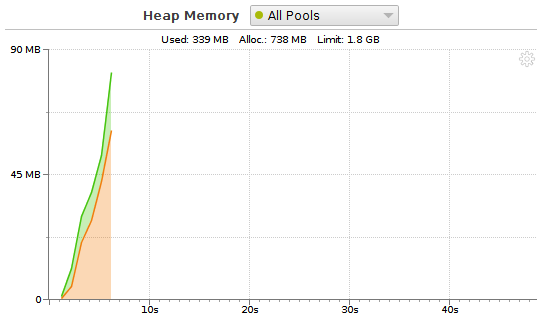


Figure 7: Problem size n = 1,000,000

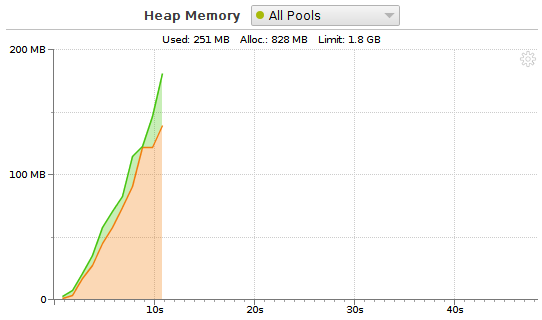


Figure 8: Problem size n = 2,000,000

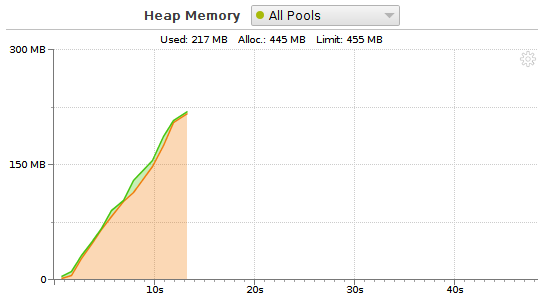


Figure 9: Problem size n = 3,000,000

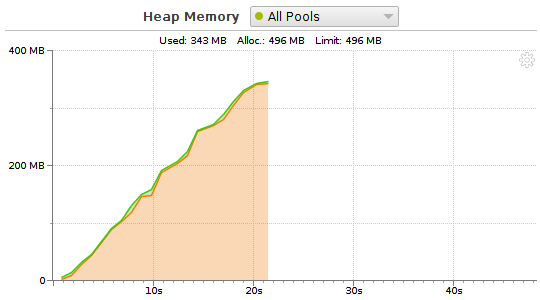


Figure 10: Problem size n = 4,000,000

# Deliverables

* Folder with different train sets.
* Folder with different test sets.
* Source code for Data Generation Tool
* Source code for ID3
* Pseudo Code
* A detailed report of the project with statistical analysis
* A presentation summarizing the investigation and results

# Task Matrix

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task | Anton | Adam | Brian | |
| Algorithm Implementation | 40% | 40% | 20% |
| Train/Test Data Set | 50% | 30% | 20% | |
| Design Document | 20% | 40% | 40% | |
| Detailed Report | 20% | 20% | 60% | |
| Presentation | 25% | 25% | 50% | |

Figure : Current Task Completion Matrix

### Appendix A: Decision Tree Algorithm Java code

package finalVer;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.UnsupportedEncodingException;

import java.util.ArrayList;

import java.util.Scanner;

public class Algorithm {

public static void main(String[] args) throws FileNotFoundException,

UnsupportedEncodingException {

//import test set

ArrayList<DataEntry> trainSet = importTrainSet();

//Create Root and input test set

Node root = new Node();

root.setAsRootNode();

root.setData(trainSet);

//Build tree

root = TreeGrowth.treeGrowth(root);

System.out.println();

System.out.println("Tree was built");

System.out.println();

//test tree

ArrayList<DataEntry> testSet = importTestSet();

for (DataEntry entry : testSet) {

decideDefault(root, entry);

}

}

private static ArrayList<DataEntry> importTrainSet() throws FileNotFoundException {

File file = new File("trainSet.txt");

Scanner sc = new Scanner(file);

sc.nextLine();

ArrayList<DataEntry> trainSet = new ArrayList<>();

//Loop through the Set file and add each case into the data entry

//then add each data entry into the Set

while(sc.hasNextLine()){

DataEntry entry = new DataEntry();

entry.setTid(Integer.parseInt(sc.next()));

entry.setHomeowner(Boolean.parseBoolean(sc.next()));

entry.setMaritalStatus(sc.next());

entry.setAnnualIncome(Integer.parseInt(sc.next()));

entry.setDefaultedBorrower(Boolean.parseBoolean(sc.next()));

trainSet.add(entry);

}

sc.close();

return trainSet;

}

private static ArrayList<DataEntry> importTestSet() throws FileNotFoundException {

File file = new File("testSet.txt");

Scanner sc = new Scanner(file);

sc.nextLine();

ArrayList<DataEntry> testSet = new ArrayList<>();

//Loop through the Set file and add each case into the data entry

//then add each data entry into the Set

while(sc.hasNextLine()) {

DataEntry entry = new DataEntry();

entry.setTid(Integer.parseInt(sc.next()));

entry.setHomeowner(Boolean.parseBoolean(sc.next()));

entry.setMaritalStatus(sc.next());

entry.setAnnualIncome(Integer.parseInt(sc.next()));

sc.next();

testSet.add(entry);

}

sc.close();

return testSet;

}

private static void decideDefault(Node node, DataEntry entry)

throws FileNotFoundException, UnsupportedEncodingException {

if (!node.isTerminal) {

if (node.atrib == 'H') {

if (entry.getHomeowner())

decideDefault(node.A, entry);

if (!entry.getHomeowner())

decideDefault(node.B, entry);

}

if (node.atrib == 'I') {

if (entry.getAnnualIncome() <= node.iSplit)

decideDefault(node.A, entry);

if (entry.getAnnualIncome() > node.iSplit)

decideDefault(node.B, entry);

}

if (node.atrib == 'M') {

if (entry.getMaritalStatus().equals("Married"))

decideDefault(node.B, entry);

if (entry.getMaritalStatus().equals("Single"))

decideDefault(node.A, entry);

if (entry.getMaritalStatus().equals("Divorced"))

decideDefault(node.C, entry);

}

}

if (node.isTerminal) {

entry.setDefaultedBorrower(node.decision);

// Print output

//System.out.println(entry.getTid() + " " + entry.getHomeowner() + " " +

entry.getMaritalStatus() + " " + entry.getAnnualIncome()

+ " " + entry.getDefaultedBorrower());

}

}

}

### Appendix B: Best Split Gini Java code

//Determining best split and returning list of children based on the best attribute with subset data already inside.

**public** **static** ArrayList<Node> gini(Node rootNode){

//Data

ArrayList<dataEntry> data = rootNode.getData();

**double** giniOfSystem,giniOfHomeOwner,giniOfMaritalStatus,giniOfAnnualIncome;;

**int** classOneOccurence = 0;

**int** classTwoOccurence = 0;

**int** numberOfTotalEntrys = data.size();

//count main class occurences

**for** (dataEntry aData : data) {

**boolean** x = aData.getDefaultedBorrower();

**if** (x) classOneOccurence++;

**else** **if** (!x) classTwoOccurence++;

}

//Calculating gini of the entire system

giniOfSystem = ((**double**)classOneOccurence/(**double**)numberOfTotalEntrys)\*((**double**)classTwoOccurence/(**double**)numberOfTotalEntrys);

//homeOwner total occurence

**int** hO1 = 0;

**int** hO2 = 0;

//homeOwner occurence per class

**int** hO1Class1 = 0;

**int** hO1Class2 = 0;

**int** hO2Class1 = 0;

**int** hO2Class2 = 0;

//MaritalStatus total occurence

**int** mS1 = 0;

**int** mS2 = 0;

**int** mS3 = 0;

//MaritalStatus occurence per Class

**int** mS1Class1 = 0;

**int** mS1Class2 = 0;

**int** mS2Class1 = 0;

**int** mS2Class2 = 0;

**int** mS3Class1 = 0;

**int** mS3Class2 = 0;

//AnnualIncome total occurence before and after the split

**int** aI1 = 0;

**int** aI2 = 0;

//AnnualIncome occurence per class

**int** aI1Class1 = 0;

**int** aI1Class2 = 0;

**int** aI2Class1 = 0;

**int** aI2Class2 = 0;

//Calculating best split point for continuous attribute

**int** aISplitOn = *findIpivot*(data);

//Calculating class occurences

**for** (dataEntry aData : data) {

//Class occurences for home Owner split

**if** (aData.getHomeowner()) {

hO1++;

**if** (aData.getDefaultedBorrower()) hO1Class1++;

**else** **if** (!aData.getDefaultedBorrower()) hO1Class2++;

} **else** **if** (!aData.getHomeowner()) {

hO2++;

**if** (aData.getDefaultedBorrower()) hO2Class1++;

**else** **if** (!aData.getDefaultedBorrower()) hO2Class2++;

}

//Class occurences for Matital Status split

**switch** (aData.getMaritalStatus()) {

**case** "Single":

mS1++;

**if** (aData.getDefaultedBorrower()) mS1Class1++;

**else** **if** (!aData.getDefaultedBorrower()) mS1Class2++;

**break**;

**case** "Married":

mS2++;

**if** (aData.getDefaultedBorrower()) mS2Class1++;

**else** **if** (!aData.getDefaultedBorrower()) mS2Class2++;

**break**;

**case** "Divorced":

mS3++;

**if** (aData.getDefaultedBorrower()) mS3Class1++;

**else** **if** (!aData.getDefaultedBorrower()) mS3Class2++;

**break**;

}

//Class occurences for Annual Income split based on the best possible split within the range of values

**if** (aData.getAnnualIncome() <= aISplitOn) {

aI1++;

**if** (aData.getDefaultedBorrower()) aI1Class1++;

**else** **if** (!aData.getDefaultedBorrower()) aI1Class2++;

} **else** **if** (aData.getAnnualIncome() > aISplitOn) {

aI2++;

**if** (aData.getDefaultedBorrower()) aI2Class1++;

**else** **if** (!aData.getDefaultedBorrower()) aI2Class2++;

}

}

//Calculating Gini Index For each attribute

//Gini Of HomeOwner

giniOfHomeOwner = ((**double**)hO1/(**double**)numberOfTotalEntrys) \*

(((**double**)hO1Class1/(**double**)hO1)\*((**double**)hO1Class2/(**double**)hO1))

+ ((**double**)hO2/(**double**)numberOfTotalEntrys) \* (((**double**)hO2Class1/(**double**)hO2)\*((**double**)hO2Class2/(**double**)hO2));

//Gini of Marital Status

giniOfMaritalStatus = ((**double**)mS1/(**double**)numberOfTotalEntrys)\*(((**double**)mS1Class1/(**double**)mS1)\*((**double**)mS1Class2/(**double**)mS1))

+((**double**)mS2/(**double**)numberOfTotalEntrys)\*(((**double**)mS2Class1/(**double**)mS2)\*((**double**)mS2Class2/(**double**)mS2))

+((**double**)mS3/(**double**)numberOfTotalEntrys)\*(((**double**)mS3Class1/(**double**)mS3)\*((**double**)mS3Class2/(**double**)mS3));

//Gini of Annual Income

giniOfAnnualIncome = ((**double**)aI1/(**double**)numberOfTotalEntrys)\*(((**double**)aI1Class1/(**double**)aI1)\*((**double**)aI1Class2/(**double**)aI1))

+((**double**)aI2/(**double**)numberOfTotalEntrys)\*(((**double**)aI2Class1/(**double**)aI2)\*((**double**)aI2Class2/(**double**)aI2));

**if**(Double.*isNaN*(giniOfHomeOwner)) giniOfHomeOwner = 1;

**if**(Double.*isNaN*(giniOfMaritalStatus)) giniOfMaritalStatus = 1;

**if**(Double.*isNaN*(giniOfAnnualIncome)) giniOfAnnualIncome = 1;

//Calculating information gain for each attribute

**double**[] InfoGain = **new** **double**[3];

InfoGain[0] = (data.get(0).getHused()) ? -1 : giniOfSystem - giniOfHomeOwner;

InfoGain[1] = (data.get(0).getMused()) ? -1 : giniOfSystem - giniOfMaritalStatus;

InfoGain[2] = giniOfSystem - giniOfAnnualIncome;

//Choosing best information gain aka what attribute to split on

**double** finalInfoGain = InfoGain[0];

**for**(**int** i = 1; i<InfoGain.length; i++)

**if** (InfoGain[i] > finalInfoGain) finalInfoGain = InfoGain[i];

/\*

System.out.println("");

System.out.println("Highest Gain " + finalInfoGain);

System.out.println("");

\*/

//what to return (a list of children with split data)

String attributeSplit;

ArrayList<Node> children = **null**;

//make a split based on final information gain

//if splitting on Home Owner

**if**(finalInfoGain == InfoGain[0]){

attributeSplit = "Home Owner";

System.***out***.println("Split on => " + attributeSplit);

System.***out***.println("");

//Make Children

Node isHomeOwner = **new** Node();

isHomeOwner.setHomeOwnerAttributeAsUsed();

isHomeOwner.setMaritalStatusAttributeFromParent(rootNode.getMaritalStatusAttributeIfUsed());

isHomeOwner.setAnnualIncomeAttributeFromParent(rootNode.getAnnualIncomeAttributeIfUsed());

Node inNotHomeOwner = **new** Node();

inNotHomeOwner.setHomeOwnerAttributeAsUsed();

inNotHomeOwner.setMaritalStatusAttributeFromParent(rootNode.getMaritalStatusAttributeIfUsed());

inNotHomeOwner.setAnnualIncomeAttributeFromParent(rootNode.getAnnualIncomeAttributeIfUsed());

rootNode.setHnode(isHomeOwner, inNotHomeOwner);

//Split data into list of children based on Home Owner attribute

children = *homeOwnerSplit*(rootNode,isHomeOwner,inNotHomeOwner, data);

//if splitting on Marital Status

}**else** **if**(finalInfoGain == InfoGain[1]){

attributeSplit = "Marital Status";

System.***out***.println("Split on => " + attributeSplit);

System.***out***.println("");

//Make Children

Node isSingle = **new** Node();

isSingle.setHomeOwnerAttributeFromParent(rootNode.getHomeOwnerAttributeIfUsed());

isSingle.setMaritalStatusAttributeAsUsed();

isSingle.setAnnualIncomeAttributeFromParent(rootNode.getAnnualIncomeAttributeIfUsed());

Node isMarried = **new** Node();

isMarried.setHomeOwnerAttributeFromParent(rootNode.getHomeOwnerAttributeIfUsed());

isMarried.setMaritalStatusAttributeAsUsed();

isMarried.setAnnualIncomeAttributeFromParent(rootNode.getAnnualIncomeAttributeIfUsed());

Node isDivorced = **new** Node();

isMarried.setHomeOwnerAttributeFromParent(rootNode.getHomeOwnerAttributeIfUsed());

isMarried.setMaritalStatusAttributeAsUsed();

isMarried.setAnnualIncomeAttributeFromParent(rootNode.getAnnualIncomeAttributeIfUsed());

rootNode.setMnode(isSingle, isMarried, isDivorced);

//Split data into list of children based on Marital Status attribute

children = *maritalStatusSplit*(rootNode,isSingle,isMarried,isDivorced,data);

//if splitting on Annual Income

}**else** **if**(finalInfoGain == InfoGain[2]){

attributeSplit = "Annual Income";

System.***out***.println("Split on => " + attributeSplit);

System.***out***.println("");

//Make children

Node beforeSplit = **new** Node();

beforeSplit.setHomeOwnerAttributeFromParent(rootNode.getHomeOwnerAttributeIfUsed());

beforeSplit.setMaritalStatusAttributeFromParent(rootNode.getMaritalStatusAttributeIfUsed());

beforeSplit.setAnnualIncomeAttributeAsUsed();

Node afterSplit = **new** Node();

afterSplit.setHomeOwnerAttributeFromParent(rootNode.getHomeOwnerAttributeIfUsed());

afterSplit.setMaritalStatusAttributeFromParent(rootNode.getMaritalStatusAttributeIfUsed());

afterSplit.setAnnualIncomeAttributeAsUsed();

rootNode.setInode(beforeSplit, afterSplit, aISplitOn);

//Split data into list of children based on Annual Income attribute

children = *annualIncomeSplit*(rootNode,beforeSplit,afterSplit, aISplitOn,data);

}

//return list of children with subset data in each

**return** children;

### }

# Bibliography

Acoggins38. "File: ID3 Decision Tree- splicing.png." 14 June 2016. *Wikipedia.* png file. 20 October 2016.

Breiman, L., et al. *Classification and Regression Trees*. New York: Chapman & Hall, 1984. Book.

"CART (Classification and Regression Trees)." *The SAGE Encyclopedia of Social Science Research Methods* (n.d.). Web. 21 October 2016.

Hssina, Badr, et al. "A comparative study of decision tree ID3 and C4.5." *International Journal of Advanced Computer Science and Applications* Special Issue Number 10 (n.d.): 13-19. Web Document. 7 October 2016. <http://saiconference.com/Downloads/SpecialIssueNo10/Paper\_3-A\_comparative\_study\_of\_decision\_tree\_ID3\_and\_C4.5.pdf>.

Hunt, Earl B., Janet Marin and Philip J. Stone. *Experiments in Induction*. New York: Academic Press, 1966. Book.

Kumar, Vipin, Michael Steinbach and Pang-Ning Tan. *Introduction to Data Mining*. Boston: Pearson Addison Wesley, 2006.

Powell, Laurel. "Decision Trees and the GINI Index." 2016. *You Tube.* video. 7 Nov. 2016.

Quinlan, J. R. and E. B. Hunt. "A Formal Deductive Problem-Solving System." *Journal of the ACM (JACM)* 15.4 (1968): 625-646. Web Document. 15 October 2016. <http://delivery.acm.org.portal.lib.fit.edu/10.1145/330000/321487/p625-quinlan.pdf?ip=163.118.172.206&id=321487&acc=ACTIVE%20SERVICE&key=A79D83B43E50B5B8%2E5EBE3BCD5AB68F1F%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&CFID=855223776&CFTOKEN=28620280&\_\_acm\_\_=14770>.

Quinlan, J. R. *C4.5 : programs for machine learning*. San Mateo: Morgan Kaufmann Publishers, 1993. Book.

—. "Induction of Decision Trees." *Scopus* (1986): 81-106. Journal.

Suknovic, M., Delibasic, B., Jovanovic, M., Vukicevic, M., Becejski-vujaklija, D., & Obradovic, Z. "Reusable components in decision tree induction algorithms." *Computational Statistics* 27.1 (2012): 127-148. Web Document. 20 October 2016.

Venkatadri, M. and Reddy C. Lokanatha. "A Comparative Study onDecision Tree Classification Algorithms in Data Mining." *International Journal of Computer Applications in Engineering, Technology, and Sciences* 2.2 (n.d.): 24-29. Web. October 2016.