



Stony Brook
University

ESE 589: LEARNING SYSTEMS

PROJECT 3:
DECISION TREE CLASSIFIER
IMPLEMENTATION & ANALYSIS

GUIDED BY:
Dr. ALEX DOBOLI

PREPARED BY:

VARUN JAIN (111685202)
AMOD GANDHE (111678938)
SHASHANK RAO (111471609)

I. Abstract:

Evolution of data mining technologies has led to a wonderful stage: classification and prediction from raw, noisy and fuzzy data. Decision Tree Classifier is among the forerunners in classification and prediction algorithms. Its relatively simple nature and accuracy of classification makes it useful in real life scenario. In this paper we first implement Decision Tree classifier and then try to analyse the performance of our implementation with an open source implementation of Support Vector Machine (SVM) Algorithm. Basic criterions which are considered while performing evaluation are - the accuracy of the classifier and time analysis of decision tree and SVM. Along with this, attempt is made to analyse the performance variations for decision tree algorithm with respect to various attribute selection methods such as Information Gain, Gain ratio and Gini Index.

II. Introduction:

Due to development in field of computer technology and computer network technology, the degree of information quality is getting higher and higher; people's ability of using information technology to collect and produce data is substantially enhanced. In this era of information, it impossible to stay away from information, and discovering useful knowledge and improving the effectiveness of information utilization are problems which need to be addressed urgently. It was under this background that Data Mining (DM) technology came into being and was developed. Data mining is a process to extract information and knowledge from a large number of incomplete, noisy, fuzzy and random data. In these data, the information and knowledge are implicit, which people do not know in advance, but are potentially useful.

Classification methods aim to identify the classes that objects belong to, from some descriptive traits. They find utility in a wide range of human activities and particularly in automated decision making. Decision trees are a very effective method of supervised learning. It makes the partitioning of a dataset into groups as homogeneous as possible in terms of the variable to be predicted. It takes as input a set of classified data, and outputs a tree that resembles an orientation diagram where each end node (leaf) is a decision (a class) and each non-final node (internal) represents a test. Each leaf represents the decision of belonging to a class of data, verifying all test paths from the root to the leaf. The basic learning approach of decision tree is greedy algorithm, which uses a recursive top-down approach of creating a decision tree structure. The tree is simpler, and technically it seems easy to use. In fact, it is more interesting to get a tree that is adapted to the probabilities of variables to be tested. Mostly, balanced tree is a good result. If a sub-tree can only lead to a unique solution, then all sub-trees can be reduced to the simple conclusion, which simplifies the process and does not change the final result. While this might suit any type of data, decision tree has inclination towards attributes having large values i.e. cardinality of the attribute is large. This will be discussed towards the end of report. Next section presents a brief discussion about

some existing work done in this domain. Generic algorithm for Decision tree is discussed followed by our implementation methodology. We discuss our experimental results and conclusion towards the end of the report.

III. Related Work:

There is lot of work being done in field of classification and prediction; discussing all of those we consider it as out of scope for this paper. In this section, we will discuss six most famous classification techniques besides Decision Tree.

- **Linear Regression:**

Linear regression is perhaps one of the most well-known and well-understood algorithms in statistics and machine learning. It relies on predictive modelling. Predictive modelling is primarily concerned with minimizing the error of a model or making the most accurate predictions possible, at the expense of descriptiveness. The representation of linear regression is an equation that describes a line that best fits the relationship between the input variables (x) and the output variables (y), by finding specific weightings for the input variables called coefficients (B). For in-depth understanding consider following equation:

$$y = B_0 + B_1 * x$$

Linear regression models predicts class variable 'y' for given input 'x'. The goal of the linear regression learning algorithm is to find the values for the coefficients B₀ and B₁. Different techniques can be used to learn the linear regression model from data, such as a linear algebra solution for ordinary least squares and gradient descent optimization.

- **Naïve Bayes:**

Naive Bayes is a simple but surprisingly powerful algorithm for predictive modelling. The model is comprised of two types of probabilities that can be calculated directly from your training data:

- 1) The probability of each class
- 2) The conditional probability for each class given each x value.

Once calculated, the probability model can be used to make predictions for new data using Bayes Theorem. When your data is real-valued it is common to assume a Gaussian distribution (bell curve) so it can easily estimate these probabilities. Naive Bayes is called naive because it assumes that each input variable is independent. This is a strong assumption and unrealistic for real data, nevertheless, the technique is very effective on a large range of complex problems.

- **K – nearest Neighbours:**

The KNN algorithm is very simple and also effective. The model representation for KNN is the entire training dataset. Predictions are made for a new data point by searching through the entire training set for the K most similar instances (the neighbours) and summarizing the output variable for those K instances. For regression problems, this might be the mean output variable, for classification problems this might be the mode (or most common) class value.

The trick is in how to determine the similarity between the data instances. The simplest technique if attributes are all of the same scale (all in inches for example) is to use the Euclidean distance, a number you can calculate directly based on the differences between each input variable

KNN can require a lot of memory or space to store all of the data, but only performs a calculation (or learn) when a prediction is needed, just in time. You can also update and curate your training instances over time to keep predictions accurate. It is suggested to use those input variables that are most relevant to predicting the output variable.

The idea of distance or closeness can break down in very high dimensions (lots of input variables) which can negatively affect the performance of the algorithm.

- **Support Vector Machine:**

Support Vector Machines are perhaps one of the most popular and talked about machine learning algorithms. A hyperplane is a multi-dimensional plane that splits the input variable space. In SVM, a hyperplane is selected to best separate the points in the input variable space by their class, either class 0 or class 1. In two-dimensions, one can visualize this as a line and let's assume that all of our input points can be completely separated by this line. The SVM learning algorithm finds the coefficients which results in the best separation of the classes by the hyperplane.

The distance between the hyperplane and the closest data points is referred to as the margin. The best or optimal hyperplane that can separate the two classes is the line that has the largest margin. Only these points are relevant in defining the hyperplane and in the construction of the classifier. These points are called the support vectors. They support or define the hyperplane. In practice, an optimization algorithm is used to find the values for the coefficients that maximize the margin.

- **Random Forest:**

Random Forest is a type of ensemble machine learning algorithm called Bootstrap Aggregation or bagging. Bootstrap is a powerful statistical method for estimating a quantity from a data sample, such as a mean. In bagging, the same approach is used, but instead for estimating entire statistical models, most commonly decision trees. Multiple samples of the training data are taken then models are constructed for each data sample. When one needs to make a prediction for new data, each model makes a prediction and the predictions are averaged to give a better estimate of the true output value.

Random forest is a tweak on this approach where decision trees are created so that rather than selecting optimal split points, suboptimal splits are made by introducing randomness. The models created for each sample of the data are therefore more different than they otherwise would be, but still accurate in their unique and different ways. We can combine their prediction results for a better estimate of the true underlying output value.

- **Deep Neural Network:**

Deep-learning networks are distinguished from the more commonplace single-hidden-layer neural networks by their depth; that is, the number of node layers through which data passes in a multistep process of pattern recognition.

Earlier versions of neural networks were shallow, composed of one input and one output layer, and at most one hidden layer in between. More than three layers (including input and output) qualify as “deep” learning. So deep is a strictly defined, technical term that means more than one hidden layer.

In deep-learning networks, each layer of nodes trains on a distinct set of features based on the previous layer output. The further you advance into the neural net, the more complex the features your nodes can recognize, since they aggregate and recombine features from the previous layer. Deep-learning networks perform automatic feature extraction without human intervention, unlike most traditional machine-learning algorithms.

Classification Algorithm	Use-case (Training Dataset Size)	Accuracy	Complexity
Linear Regression	Moderate	Moderate	Moderate
Naïve Bayes	Moderate or large	High(for categorical data)	Low
K-Nearest	Small to moderate	Moderate	Low
SVM	Large	High	Moderate
Random Forest	Large	High	Moderate
Deep Neural Network	Large to very Large	High	High

Table 1: General Comparison of Classification Algorithms.

IV. Generic Algorithm:

The Decision Tree classification algorithm involves two stages. First stage is training of the model and in the second stage, decision tree predicts the outcome for the data provided to the model. We will identify each stream of data as a tuple and outcome as a class variable. Training data consists of 'r' numbers of instances, and 'n' number of attributes: $A = \{a_0, a_1, a_2, \dots, a_n\}$ where A represents list of attributes and the last attribute is the class variable. During the initial scan of the database, all distinct class variables are calculated and count of each variable is stored in local memory. Decision tree consists of node and branches. Node dictates the name of test on tuple, whereas each branch represents possible outcome of that test.

1. Model training:

If dataset contains all tuples of the same class then a node is created and marked as leaf node. Class variable is assigned to it and returned. In case dataset does not contain classes of one single type and attribute list is empty, the class variable returned is determined by majority voting of classes present.

If both conditions are not met, this means that dataset consists of non-zero number of attributes and there are at least two distinct tuples. If this is the case, algorithm starts building the Decision Tree. Attribute selection method is applied to select best attribute(s) for partitioning dataset. Attribute list 'A' and dataset 'D' are the inputs provided to attribute selection method. One of the following attribute selection methods can be then applied according to requirements:

- a) Information Gain: Given the dataset 'D', attribute list 'A' and list of distinct class variable 'C' calculate expected information gain needed to classify tuple in D by,

$$Info(D) = \sum_{i=0}^{C.length() - 1} p_i \log_2(p_i)$$

Here p_i represents non-zero probability of tuple X belonging to class C_i . Now to select attribute A_i as an partitioning attribute,

$$Info_{A_i}(D) = \sum_{i=0}^{D.length() - 1} \frac{|D_i|}{|D|} * Info(D_i)$$

where D_i are the datasets created after applying partitioning.

Calculate information gain by,

$$Info_Gain_{(A_i)} = Info(D) - Info_{A_i}(D)$$

After calculating Information Gain for each of the attribute, select attribute having maximum Information gain as a splitting point at current node.

- b) Gain Ratio: Normalization to information can be applied by introducing split information. Split Information for attribute can be calculated as:

$$Split_Info_{A_i}(D) = \sum_{i=0}^{D.length() - 1} \frac{|D_i|}{|D|} * \log_2\left(\frac{|D_i|}{|D|}\right)$$

Calculate Gain Ratio for each attribute by,

$$Gain_Ratio_{(A_i)} = \frac{Gain(A_i)}{Split_Info_{A_i}(D)}$$

Attribute with maximum Gain Ratio is selected as a partitioning attribute at current state of algorithm.

- c) Gini Index: Impurity in data can be calculated by means of Gini Index, impurity is measured in dataset by:

$$Gini(D) = 1 - \sum_{i=0}^{C.length() - 1} p_i^2$$

For each of the attribute, calculate Gini Index as,

$$Gini_{Ai}(D) = \frac{|D1|}{|D|} Gini(D1) + \frac{|D2|}{|D|} Gini(D2)$$

where D1 and D2 are the two datasets created after applying splitting function.

Based on reduction in impurity, the attribute that maximizes the reduction becomes a splitting attribute.

$$\Delta Gini(A) = Gini(D) - GiniA(D)$$

One of the attribute selection methods will provide a best attribute to split on at current moment. Attribute A_i will behave as splitting criterion. Node is assigned with label of A_i name.

Now at this moment we have used attribute A_i and now it must be marked. This can happen only if splitting attribute is discrete valued and it allows multi-way splits then to remove current attribute from A: list of attribute.

At this point, D_j which is an subset of original dataset is considered. D_j consists of tuples which satisfy outcomes of given splitting attribute. Each time condition is checked if subset of dataset is empty. If it is empty then the node is marked as leaf node and respective class variable is assigned. Also if D_j is a pure subset, then define it as a leaf node and assign respective class variable to it. Else, recursive calls are made to the algorithm with modified attribute list and D_j . If there is no data tuple left or all the subsets generated are pure then stopping condition for training phase is occurred. These steps basically will construct the decision tree which will be readily available for future prediction and classification.

2. Classification and Prediction:

This step assumes that the decision tree has been built in the previous part. Dataset in this step has one less attribute: Class variable. Based on previous data computations, decision tree has to classify the incoming tuple in its most possible class variable. Here tuple X does not have class variable. This tuple is then fed to pre-built decision tree. Nodes in this tree represents test on attributes. Each corresponding attribute is then tested. Based on the results, traverse the tree in depth first manner. Unless leaf node is reached, this procedure is repeated. Once we reach the leaf node in decision tree, which is nothing but the class variable C_i for list of C, is returned to user.

V. **Implementation:**

The main implementation of the Decision tree algorithm can be divided into three main subparts:

- Model Training
- Classifier Validation
- Classification and Prediction

For these functionalities we have split up the given training dataset in 20-80 proportion. 80% of it will belong to model training while remaining 20% will be used to validate classifier implementation.

- Model Training

The implementation of the program begins at Main.java which contains the main method. The first action performed by the main method is to read the input dataset from the file and store the values in *DataSet.java* class object variable. The *DataSet.java* class consists of a `List<String[]>` variable which will hold the input training data read from the .csv file. Each attribute value of one row is stored in the String array, and multiple rows are stored in the List. *DataSet.java* also consists of `Map<String, Double>` which will hold the values of the class attribute as well as a count of the values present, and String which will hold the majority class attribute value. The following is a code snippet of *DataSet.java*:

```
public class DataSet {  
  
    List<String[]> data = new ArrayList();  
    Map<String, Double> classes = new HashMap();  
    String majorityClass = null;  
  
}
```

The DataSet object stores information regarding the dataset whenever a new dataset is created by partition. As a result, by storing this value for every successive partition of the dataset, we are able to generalize the algorithm regardless of the attribute selection metric applied, thus increasing the efficiency and generalization of our implementation.

Once the data has been read into the DataSet object, a new `List<Integer>` is created to store the numbers mapped to attributes of the DataSet, i.e., for example, if the DataSet contains 10 attributes then, a list of 10 integers are created which will be used to correspond to the 10 attributes of the dataset. The DataSet object holding the input values and the newly created list are then passed to the *buildTree()* method, which will be used to construct the tree.

Within this *buildTree()* method, first a check is performed to check if the DataSet object all have the same class values or if the attribute list is empty. If either of these conditions are true, then a new Node object initialized with the majorityClass string label is returned to the calling function. The following code snippet shows how these conditions are implemented:

```
if (dataset.classes.keySet().size() == 1) {  
    return new Node(dataset.majorityClass);  
}  
if (attrList.isEmpty()) {  
    return new Node(dataset.majorityClass);  
}
```

Here, Node is an object defined in *Node.java* class. This class contains an integer to store the attribute number (with a default value of -1), a String variable to hold the label of the node, a `Set<String>` which will hold subsets of attributes (required for CART), and an `ArrayList<BranchDecisions>` in order to hold the different branches which can be created for the node. It also contains constructors which will

either fill the label value of the node or will fill the node with attribute number which is selected in the attribute selection methods. The following code snippet shows how *Node.java* is implemented.

```
public class Node {

    int attr = -1;
    Set<String> attrSet; // not used
    String label = "ERROR";

    ArrayList<BranchDecision> branches;

    Node(String label){
        this.label = label;
        this.branches = new ArrayList<>();
        attrSet = new HashSet<String> ();
    }

    Node(int attr){
        this.attr = attr;
        this.branches = new ArrayList<>();
        attrSet = new HashSet<String> ();
    }

}
```

BranchDecision is an object defined in *BranchDecision.java* class which contains a String variable to hold the condition for the decisions, a Set<String> to hold subset of attribute values (required for CART), a DataSet object to hold the partitioned dataset (which is obtained after applying attribute selection methods) and a Node object which will be the child node for the decision tree. If either of the above 'if' conditions are satisfied, a new node is created with the majority value of the target class and is returned to the calling function. This newly created node is the leaf node of the Decision Tree. The following code snippet shows how *BranchDecisions.java* is implemented:

```
public class BranchDecision {

    String equalTo;
    Set<String> attrSet;
    DataSet partitionedDataSet;
    Node resultant;

}
```

If neither of the 'if' conditions are satisfied, the attribute selection methods are called. There is a static variable defined in *Main.java*, which is used to set which attribute selection metric is to be used (ID3, C45 or CART). If 'ID3' metric is set for the attribute selection method, then *AttributeSelector()* method of *ID3.java* class is called. This method takes DataSet and attribute list as parameters and selects the attribute to be used for partitioning the dataset by calculating the Information Gain. If 'C4.5' metric is set for the attribute selection method, then *AttributeSelector()* method of *C45.java* is called. This method also takes DataSet and attribute list as parameters and selects the attribute to be used for partitioning the dataset by calculating the Gain Ratio. If the 'CART' metric is set for the attribute selection method, then *AttributeSelector()* method of *CART.java* class is called. This method also takes DataSet and attribute list as parameters and selects the attribute to be used for partitioning the dataset by calculating the Gini Index.

All of the attribute selection methods return a Node object containing the selected attribute which will be used to further partition the DataSet. The selected attribute is removed from the attribute list and the remaining attributes are used for partitioning the tuples growing the subtrees for the partitions. If the partitioned dataset is empty, a leaf node is created with the majority value of the class attribute and returned to the calling function. If the partitioned dataset is not empty, *buildTree()* method is recursively called in order to grow the partitioned subtree created. Once all the recursive calls are ended, the initial *buildTree()* method will return the root Node object of the created tree which will be the Decision Tree. This ends the training phase for the model.

- Classifier Validation

In order to validate the created Decision tree, the test data file is read. This test data will not have the target class attribute values present (but these values will be known), and each of the read tuple is passed to the *validation()* method, which iteratively calls the *predict()* method, which will use the root Node of the created decision tree to give the output of the tuple. This output is matched with the expected output available with the user in order to obtain the accuracy of the Decision Tree classifier.

- Classification and Prediction

In order to classify more data, files similar to the one created in the previous step (test data file) can be passed to the *predict()* method to obtain the output.

VI. Example Illustrating our Implementation:

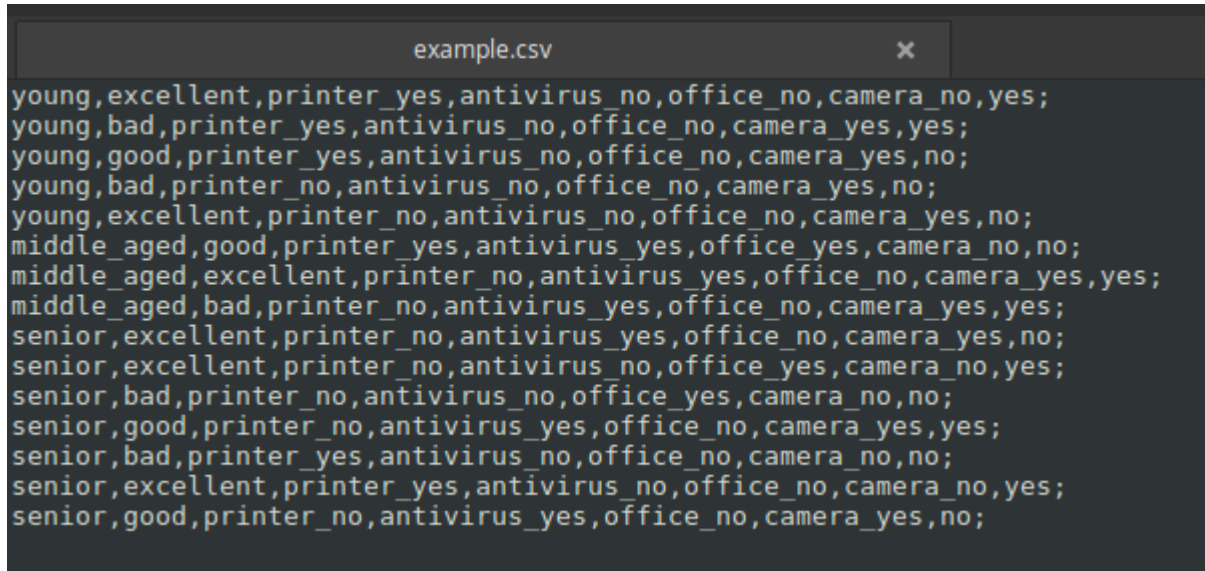
We have created a small dataset example to illustrate the way in which our implementation works. Consider the following dataset available with us for a computer store consisting of 15 different customer transactions:

Age Range	Customer Behavior	Brought Printer	Brought Antivirus	Brought Office Software	Brought Camera	Buys Computer (target attribute for classification)
young	excellent	yes	no	no	no	yes
young	bad	yes	no	no	yes	yes
young	good	yes	no	no	yes	no
young	bad	no	no	no	yes	no
young	excellent	no	no	no	yes	no
middle_aged	good	yes	yes	yes	no	no
middle_aged	excellent	no	yes	no	yes	yes

middle_aged	bad	no	yes	no	yes	yes
senior	excellent	no	yes	no	yes	no
senior	excellent	no	no	yes	no	yes
senior	bad	no	no	yes	no	no
senior	good	no	yes	no	yes	yes
senior	bad	yes	no	no	no	no
senior	excellent	yes	no	no	no	yes
senior	good	no	yes	no	yes	no

Table 2: Test Dataset

In order to better understand as well as explain this example, we have appended the item name to yes/no such as in the example .csv file shown below:



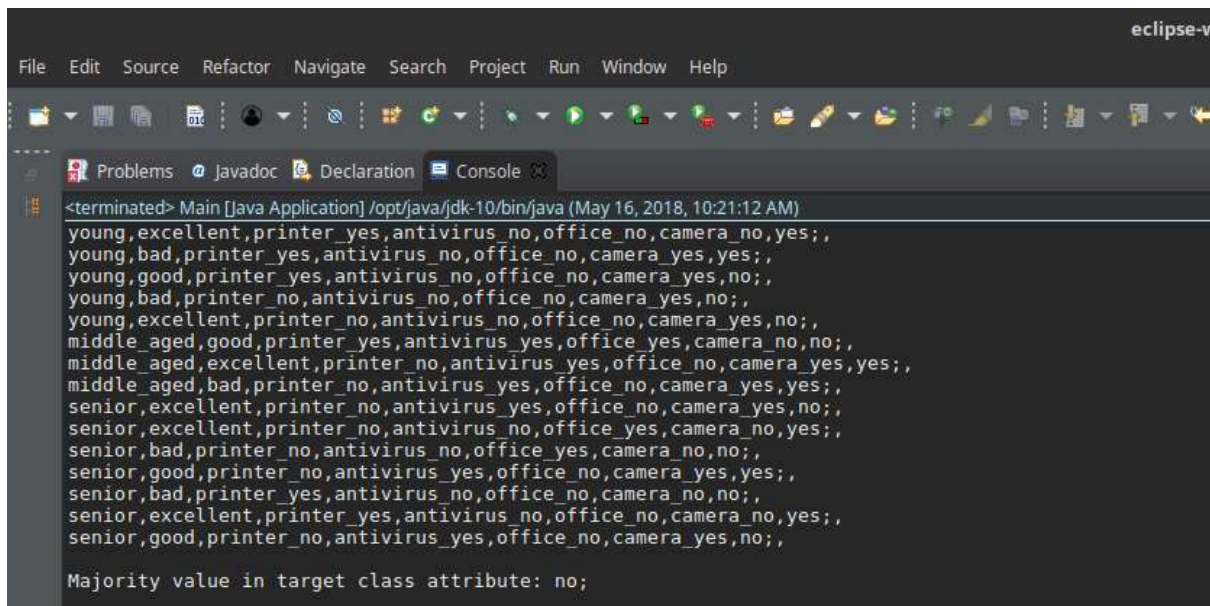
```

example.csv
young,excellent,printer_yes,antivirus_no,office_no,camera_no,yes;
young,bad,printer_yes,antivirus_no,office_no,camera_yes,yes;
young,good,printer_yes,antivirus_no,office_no,camera_yes,no;
young,bad,printer_no,antivirus_no,office_no,camera_yes,no;
young,excellent,printer_no,antivirus_no,office_no,camera_yes,no;
middle_aged,good,printer_yes,antivirus_yes,office_yes,camera_no,no;
middle_aged,excellent,printer_no,antivirus_yes,office_no,camera_yes,yes;
middle_aged,bad,printer_no,antivirus_yes,office_no,camera_yes,yes;
senior,excellent,printer_no,antivirus_yes,office_no,camera_yes,no;
senior,excellent,printer_no,antivirus_no,office_yes,camera_no,yes;
senior,bad,printer_no,antivirus_no,office_yes,camera_no,no;
senior,good,printer_no,antivirus_yes,office_no,camera_yes,yes;
senior,bad,printer_yes,antivirus_no,office_no,camera_no,no;
senior,excellent,printer_yes,antivirus_no,office_no,camera_no,yes;
senior,good,printer_no,antivirus_yes,office_no,camera_yes,no;

```

Figure 1: test.csv

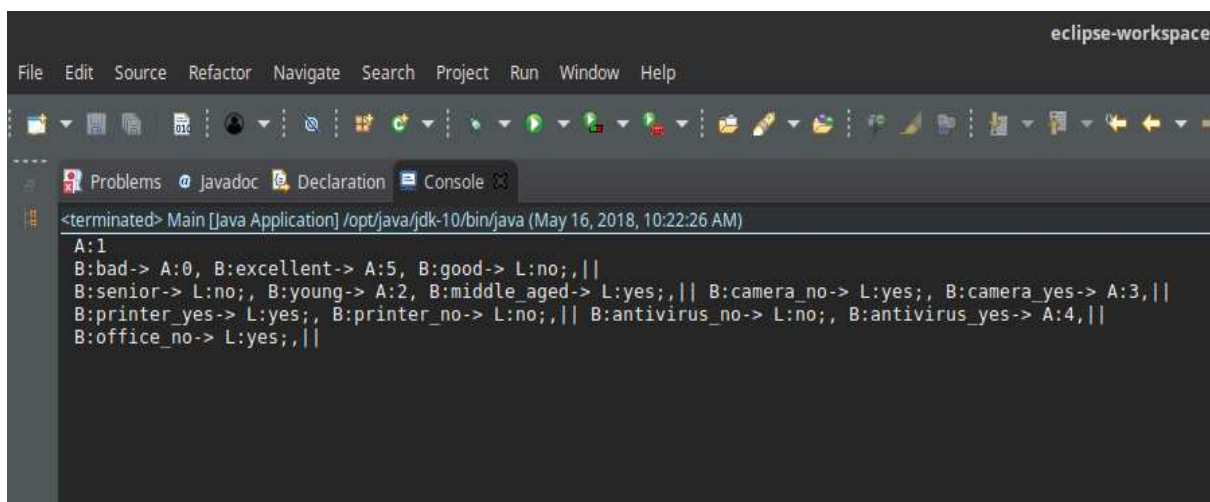
The following figure shows the values stored in the DataSet object after the input file is read:



```
<terminated> Main [Java Application] /opt/java/jdk-10/bin/java (May 16, 2018, 10:21:12 AM)
young,excellent,printer_yes,antivirus_no,office_no,camera_no,yes;,
young,bad,printer_yes,antivirus_no,office_no,camera_yes,yes;,
young,good,printer_yes,antivirus_no,office_no,camera_yes,no;,
young,bad,printer_no,antivirus_no,office_no,camera_yes,no;,
young,excellent,printer_no,antivirus_no,office_no,camera_yes,no;,
middle_aged,good,printer_yes,antivirus_yes,office_yes,camera_no,no;,
middle_aged,excellent,printer_no,antivirus_yes,office_no,camera_yes,yes;,
middle_aged,bad,printer_no,antivirus_yes,office_no,camera_yes,yes;,
senior,excellent,printer_no,antivirus_yes,office_no,camera_yes,no;,
senior,excellent,printer_no,antivirus_no,office_yes,camera_no,yes;,
senior,bad,printer_no,antivirus_no,office_yes,camera_no,no;,
senior,good,printer_no,antivirus_yes,office_no,camera_yes,yes;,
senior,bad,printer_yes,antivirus_no,office_no,camera_no,no;,
senior,excellent,printer_yes,antivirus_no,office_no,camera_no,yes;,
senior,good,printer_no,antivirus_yes,office_no,camera_yes,no;,
Majority value in target class attribute: no;
```

Figure 2: Dataset and Majority class

Once the DataSet object and attribute list is created, *buildTree()* method is called using *id3AttributeSelector()* attribute selection method which builds the Decision Tree. The created tree is displayed in the figure below:



```
<terminated> Main [Java Application] /opt/java/jdk-10/bin/java (May 16, 2018, 10:22:26 AM)
A:1
B:bad-> A:0, B:excellent-> A:5, B:good-> L:no;||
B:senior-> L:no;, B:young-> A:2, B:middle_aged-> L:yes;|| B:camera_no-> L:yes;, B:camera_yes-> A:3,||
B:printer_yes-> L:yes;, B:printer_no-> L:no;|| B:antivirus_no-> L:no;, B:antivirus_yes-> A:4,||
B:office_no-> L:yes;||
```

Figure 3: Decision Tree

This Decision tree created can be explained as follows. Initially, the arraylist which was created to hold all the attributes of the dataset. This can be visualized as follows:

List of Attributes						
[0]	[1]	[2]	[3]	[4]	[5]	[6]
Age Range	Customer Behavior	Bought Printer	Bought Antivirus	Bought Office Software	Bought Camera	Buys Computer (target class attribute)

Figure 4: Attribute List Representation

These list numbers (or attribute numbers) are used in the output decision tree. In the output figure, A stands for a node, B stands for a branch (or edge) and L stands for the leaf node. Thus, (A:1) means that the attribute Customer Behavior (list number 1) is the first attribute selected using the attribute selection algorithm. The row following it contains (B:bad-> A:0) which means that if the attribute value of the parent (A:1 in this case) has an attribute value of bad, then the following child node is (A:0). '|' in between the output is used to separate the children nodes of different parent nodes. Similarly all the nodes and edges of the decision tree can be obtained. The following figure shows a more relatable figure of the obtained output tree:

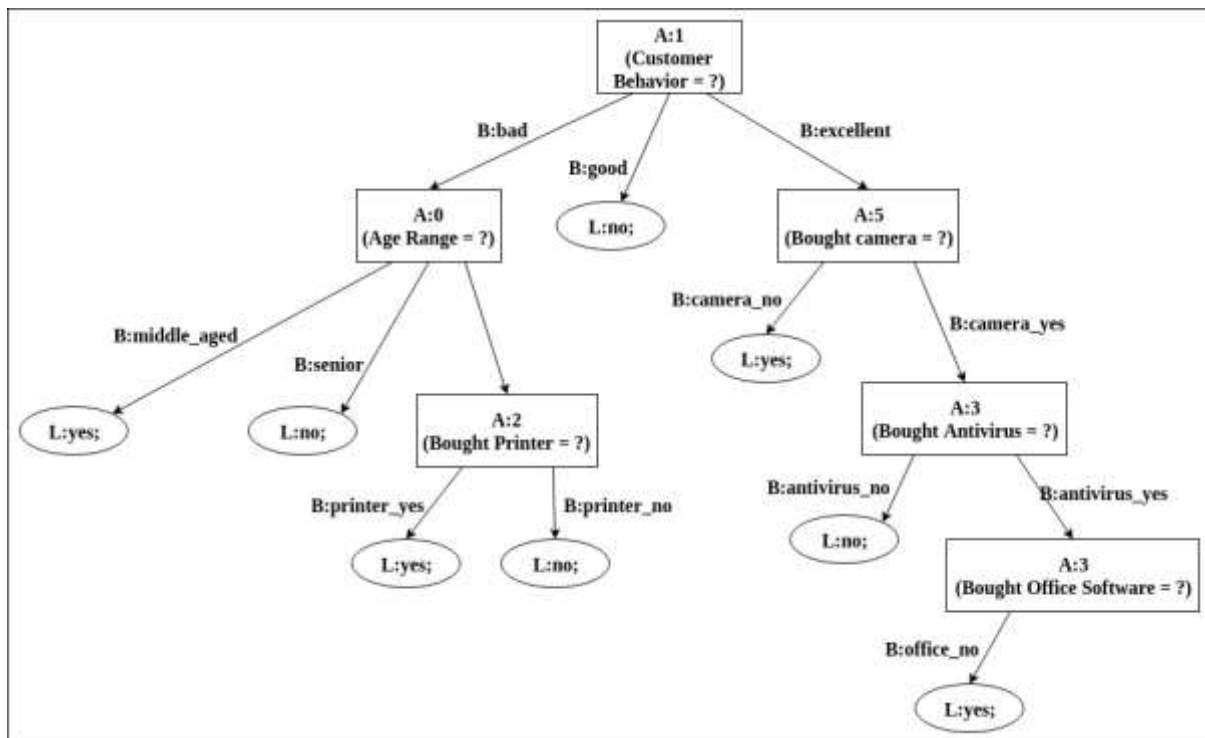
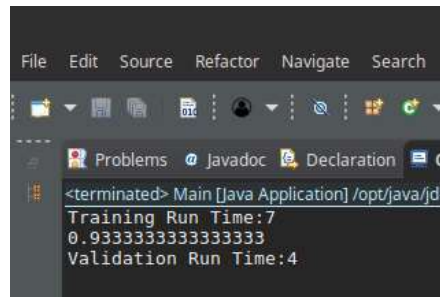


Figure 5: Decision Tree Representation

This figure is a more clearer representation of the output which is obtained by our implementation. This figure also allows us to understand the output decision tree obtained by us.

Once we have obtained this decision tree, we can validate our implementation by running the different test cases with known recommendations - including the ones used for creating the decision tree (though these values will be overfitted). The inputs in this case do not contain the class attribute

values; these are obtained by running the inputs through the decision tree. The following figure shows the result obtained when validation is run from another file:



```
<terminated> Main [Java Application] /opt/java/jdk
Training Run Time:7
0.9333333333333333
Validation Run Time:4
```

Figure 6: Output

As can be seen from the results, most of the validation tuples provide a true positive result. Since our validation is at a percentage of ~93.33%, we can say that the Decision tree classifier created has a high percentage of accuracy. Since validation for this small self-implemented dataset is correct, we can extend our hypothesis that our implementation for Decision Tree algorithm is valid and can be applied to larger datasets too.

VII. Running the SVM Classifier:

In order to compare how our implemented algorithm performed in comparison to the current existing classification algorithms, we decided to run the “Dresses_Attribute_Sales Data Set” as well as “Mushroom Data Set” available in the UCI Machine Learning repository in an SVM classifier. This section provides details on how the SVM was implemented.

We have written a simple python script which uses ‘svm’ library in order to create the SVM classifier. The following figure shows the script used:

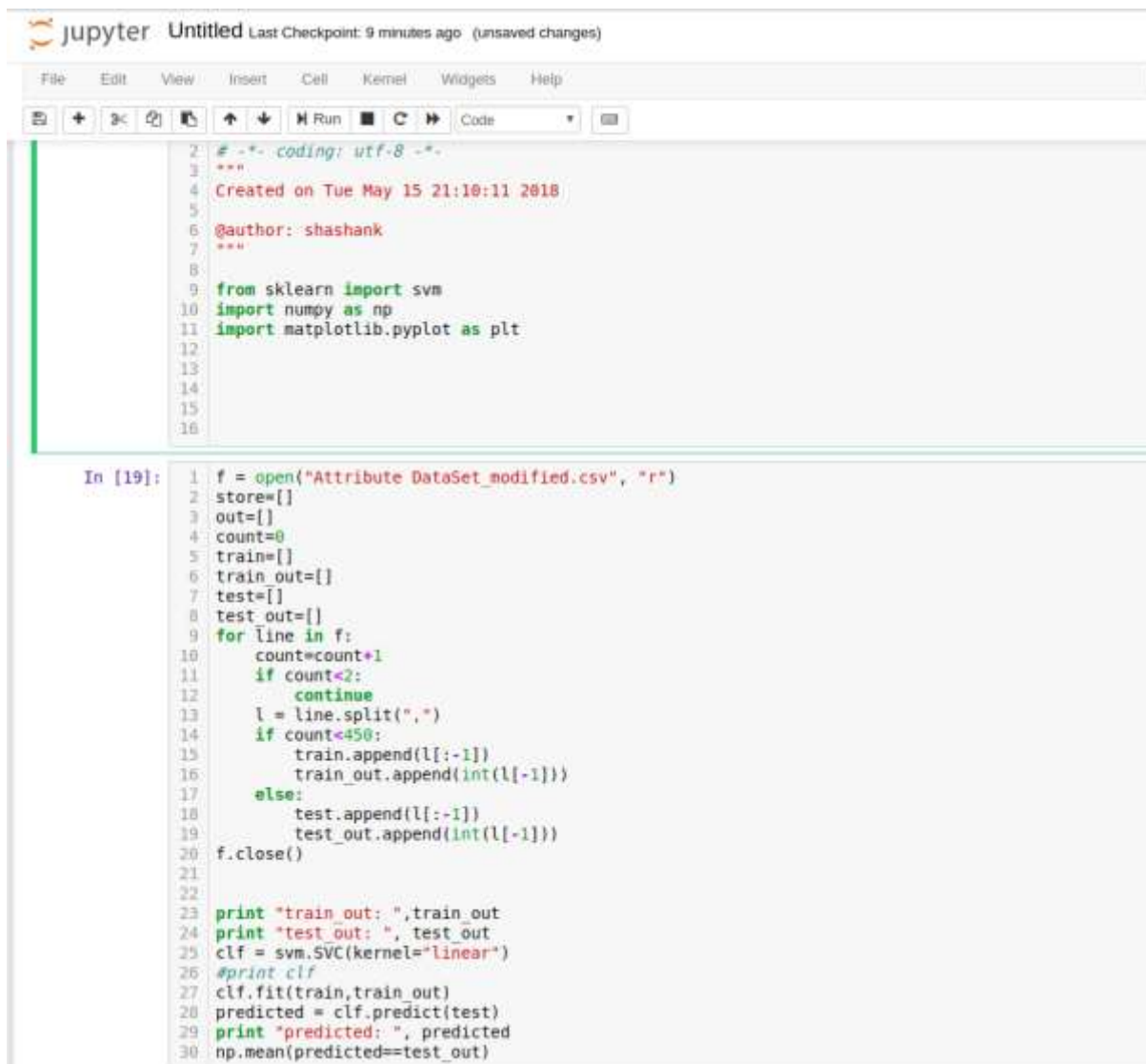


Figure 7: SVM Script in Python

However, the svm library required numerical valued dataset as parameters, while the Dresses_Attribute_Sales Data Set was nominal. We therefore had to digitize the nominal values of the dataset, i.e., distinct values in every column were given a value starting from one. Once the dataset was digitized, we used the file to create arrays - training data arrays (train array for holding all attribute values other than the target class attribute and train_out which held the values of the target class attribute, both for the training data set) and the test data arrays (test array for holding all attribute values other than the target class attribute and test_out which held the values of the target class attribute, both for the testing data set). The train_out and test_out arrays are shown in the following figure:

The reason for this low accuracy is the small size of the dataset which results in smaller training data available to train and build the SVM classifier. The execution time taken for the SVM model to be trained as well as validated was 43.4239 ms and is also captured as below:

```
In [2]: 1 time1 = time.time()
2
3 f = open("Attribute DataSet_modified.csv", "r")
4 store=[]
5 out=[]
6 count=0
7 train=[]
8 train_out=[]
9 test=[]
10 test_out=[]
11 for line in f:
12     count=count+1
13     if count<2:
14         continue
15     l = line.split(",")
16     if count<450:
17         train.append(l[:-1])
18         train_out.append(int(l[-1]))
19     else:
20         test.append(l[:-1])
21         test_out.append(int(l[-1]))
22 f.close()
23
24
25 #print "train out: ", train_out
26 #print "test out: ", test_out
27 clf = svm.SVC(kernel="linear")
28 #print clf
29 clf.fit(train, train_out)
30 predicted = clf.predict(test)
31 #print "predicted: ", predicted
32 np.mean(predicted==test_out)
33 time2 = time.time()
34 print (time2 - time1) * 1000, "ms"
43.4238910675 ms
```

Figure 12: Running Time

Similar analysis for the “Mushroom Data Set” for a training data set of ~6500 tuples (out of a total of ~8500 tuples) leads to responses as shown in the figure:

```
Time taken to train model: 8.51029992104 seconds
Time taken to train model: 0.0216999053955 seconds
('Accuracy: ', '0.9427692307692308')
```

Figure 13: Mahroom dataset with 6500 records

On increasing the number of tuples in the training data set to ~7500 tuples, it was observed that the accuracy of the Decision tree created was increased to the figures as shown below:

```
Time taken to train model: 12.2103061676 seconds
Time taken to train model: 0.00952100753784 seconds
('Accuracy: ', '0.9616')
```

Figure 14: Mahroom dataset with 7500 records

We have run SVM classifiers for these two datasets. Comparisons with the Decision Tree implemented by us have been recorded in the experimental results section.

VIII. Experimental Results

Fully functional code for Decision Tree classifier was written in Java environment for Eclipse, whereas python script was written for SVM. Testing and experiments were conducted on Lenovo Yoga 710 with 8 GB RAM and 2.70GHz processor. Primarily we tested accuracy and runtime for the 'test' dataset which we have mentioned previously. For real time understanding of the performance we have used following datasets:

1. Dress_Set: No. of Attribute: 12, No. of Records: ~700
2. Mushroom: No. of Attributes: 23, No. of Records: ~8500

Along with this to analyse change in accuracy we have modified training dataset by reducing its number of instances. Discussion about this is made in later half of the experimental results discussion.

Experiment 1:

Analysis of total run time for SVM and Decision Tree classifier with different attribute selection methods.

	ID-3	C 4.5	CART	SVM
Test	0	5	0	-
Dress_Set	31	6	1262985	43.42
Mushroom	160	172	907	512

Table 1: Running time comparison

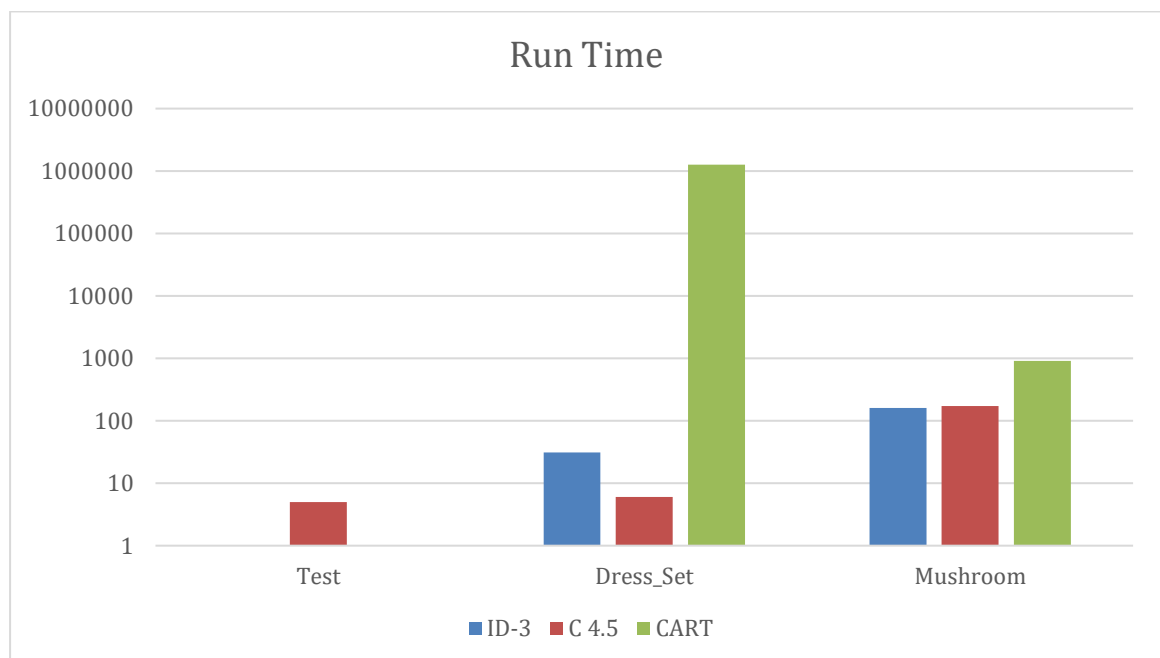


Figure 15: Running Time

ANOMALY-1: We can see that there is a drastic difference in timings for building the Tree for Dress_Set dataset using CART. This is discussed in Anomaly analysis section.

Experiment 2:

Accuracy comparison for SVM and Decision tree classifier implemented with different attribute selection methods:

	ID-3	C 4.5	CART	SVM
Test	86.66	86.66	93.33	-
Dress_Set	50.98	56.86	37.28	59.6154
Mushroom	96.05	96.05	95.56	94.27

Table 2: Model Accuracy

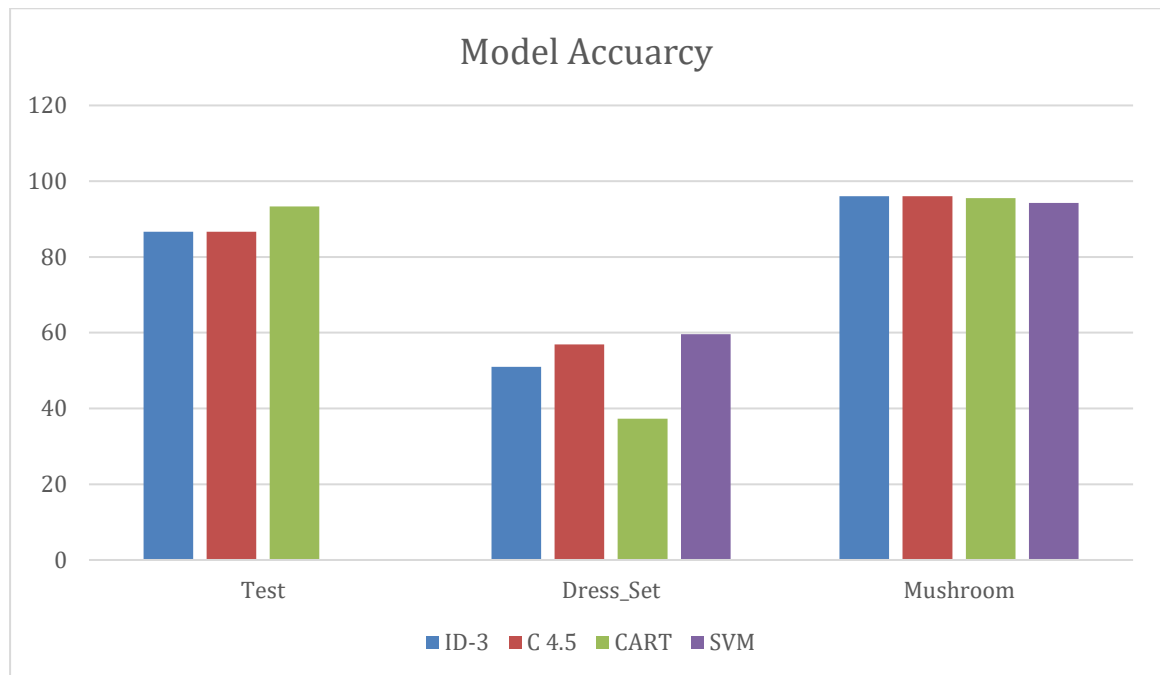


Figure 16: Accuracy Comparison

Our observation for drastic change in accuracy for mushroom dataset is due number of records available for training the model. Here, if we have more number of instances are available for training the model, then it has more number of observation available to compare with. As scope of the model increases, it can more accurately predict the future outcomes based on previous data model has learned. To verify our verdict, we tried to reduce the number of records for Dress_Set and Mushroom data set. Following graph indicate change in accuracy with change in number of records for training dataset.

Dress_Set

Method	350	450
ID 3	58	50.98
C 4.5	56	56.86
CART	35	37

Table 3: Dress_Set Analysis based on Records

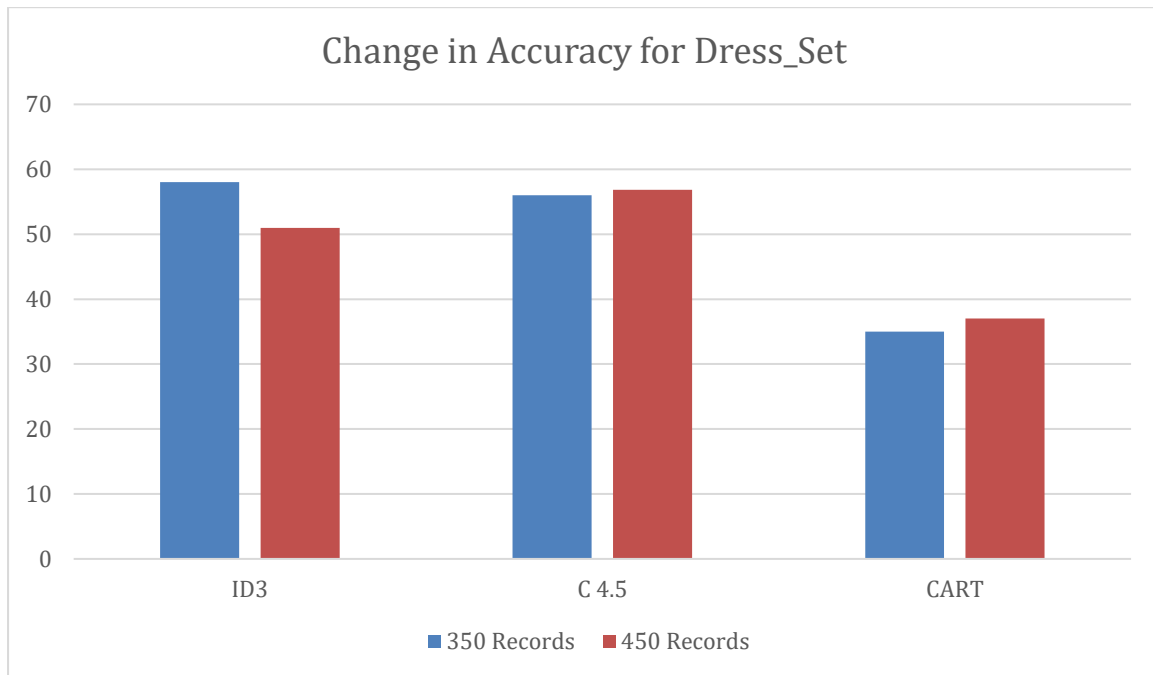


Figure 17: Dress_Set Analysis for change in Records count

Mushroom Dataset:

Method	5500	6500
ID 3	80.03	96.05
C 4.5	80.33	96.05
CART	86.73	95.56

Table 4: Mushroom Analysis for change in records

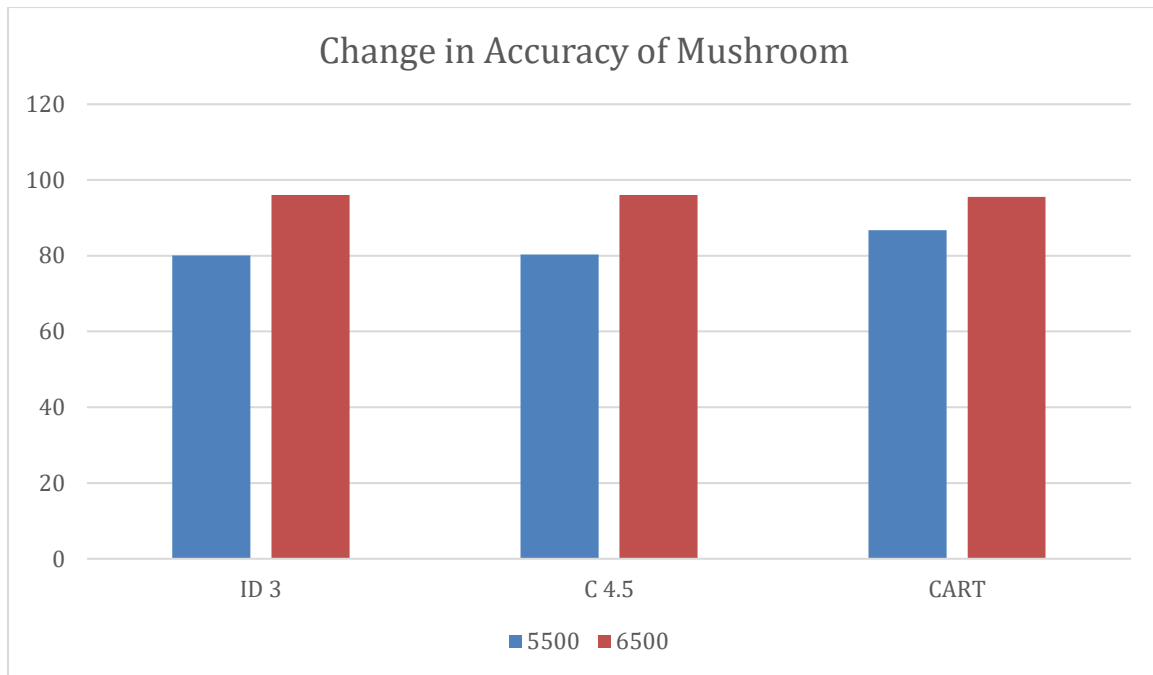


Figure 18: Mushroom Analysis for change in records

Mushroom Dataset on SVM:

Classifier	6500	7500
SVM	94.27	96.16

Table 5: SVM analysis for change in records

Here both the graphs confirm our analysis which suggests, reduction in number of records for training the model will affect the accuracy while predicting. Also experiment performed on SVM classifier were no different. Change in accuracy can be observed for SVM as well.

ANOMALY-2: For the Dress_Set dataset, in case of ID 3 even though number of records are increased there is drop in accuracy of the model. This anomaly is discussed in Anomaly analysis section.

Anomaly Analysis:

```

29 A:3
30 B:[1]-> A:10, B:[2, 3, 4, 5]-> A:0,||
31 B:[11, 12, 4, 9]-> A:0, B:[1, 2, 3, 6, 8, 10]-> A:0,|| B:[1, 6, 7, 9]-> A:1, B:[11, 12, 2, 3, 4, 5, 10]-> A:7,||
32 B:[4, 6]-> L:1, B:[1, 2, 3, 9]-> A:8,|| B:[1, 0, 10]-> A:8, B:[2, 3, 4, 5, 6, 9]-> A:2,|| B:[1]-> A:9, B:[2, 3, 4, 5]-> A:0,|| B:[1, 14, 3, 5, 6, 10]-> A:4, B:[11, 12, 13, 18, 2, 4, 7, 8, 9]-> A:8,||
33 B:[5]-> L:1, B:[1, 4]-> A:7,|| B:[4, 8, 9]-> L:0, B:[1, 5, 6]-> A:7,|| B:[4, 6]-> A:5, B:[1, 2, 3, 5]-> A:9,|| B:[15, 7]-> L:1, B:[11, 1, 2, 3, 16, 5]-> A:7,|| B:[15, 5, 17]-> L:0, B:[1, 3, 4, 16, 18, 6, 9]-> A:5,|| B:[15, 4, 7]-> A:9, B:[1, 2, 13, 3, 5, 6, 8, 9]-> A:0,|| B:[11, 8]-> A:5, B:[1, 4, 5, 18, 7]-> A:5,||
34 B:[4]-> A:2, B:[1, 12, 2, 3, 5]-> L:0,|| B:[3, 6]-> L:0, B:[1, 2, 5]-> A:2,|| B:[3, 5]-> L:0, B:[1, 7]-> A:1,|| B:[5]-> A:0, B:[1, 2, 13, 3, 4, 16, 17, 7, 8, 9]-> A:7,|| B:[8, 10]-> L:1, B:[1, 12, 2, 5, 6]-> A:8,|| B:[5, 6]-> A:9, B:[1, 3, 7]-> A:9,|| B:[4]-> L:0, B:[1, 3, 19]-> A:10,|| B:[2, 3, 5]-> A:2, B:[11, 12, 4, 10]-> L:0,|| B:[3]-> L:0, B:[5, 7]-> L:1,|| B:[3]-> A:1, B:[1, 4, 5, 6, 7]-> A:3,||
35 B:[1]-> L:1, B:[5]-> L:0,|| B:[4]-> L:0, B:[1, 2]-> A:4,|| B:[3]-> L:0, B:[1, 2, 5]-> L:1,|| B:[5]-> L:1, B:[2, 9]-> L:0,|| B:[6]-> L:0, B:[11, 1, 12, 2, 3, 4, 5, 20, 10]-> A:8,|| B:[3, 10]-> L:1, B:[1, 4, 5, 7, 18]-> A:0,|| B:[1, 8]-> L:1, B:[13, 4, 5, 16]-> A:8,|| B:[2, 14]-> A:2, B:[1, 12, 3, 4, 16, 5, 8, 9]-> A:10,|| B:[4]-> L:0, B:[11, 1, 2, 3]-> L:1,|| B:[1, 2]-> A:9, B:[3, 4, 5]-> A:9,|| B:[1, 3]-> L:0, B:[2, 4]-> A:4,|| B:[2, 3]-> L:0, B:[4, 5]-> A:9,||
36 B:[2]-> L:0, B:[1, 3]-> L:1,|| B:[8]-> A:0, B:[1, 4, 5, 17, 18]-> A:2,|| B:[7, 9]-> A:2, B:[1, 6]-> A:5,|| B:[4]-> A:1, B:[1, 9]-> L:0,|| B:[1]-> L:1, B:[2]-> L:0,|| B:[10]-> L:0, B:[1, 2, 3, 4]-> A:3,|| B:[11, 3, 8, 9]-> L:0, B:[1, 12, 13, 2, 4, 16]-> A:10,|| B:[3, 4, 6]-> A:7, B:[1, 2, 16, 5, 9]-> A:4,|| B:[1]-> L:1, B:[3, 7]-> L:0,|| B:[4, 16]-> A:0, B:[1, 2, 8]-> A:10,||
37 B:[2]-> L:0, B:[4, 5]-> L:1,|| B:[1]-> A:7, B:[2, 5]-> L:1,|| B:[2, 4]-> L:1, B:[1, 5]-> A:3,|| B:[3]-> A:2, B:[1, 5, 6]-> L:0,|| B:[2, 4]-> L:1, B:[3, 5]-> L:0,|| B:[4]-> A:4, B:[2, 3, 5]-> A:7,|| B:[2, 6]-> A:0, B:[1, 3, 4, 8]-> A:7,|| B:[3]-> L:0, B:[1, 5]-> L:1,|| B:[1, 2]-> L:0, B:[3, 5]-> A:10,|| B:[2]-> L:1, B:[5]-> L:0,|| B:[1]-> A:2, B:[2, 3, 4, 8]-> L:0,||
38 B:[1]-> L:0, B:[12, 2, 3, 5, 20, 10]-> L:1,|| B:[4]-> L:1, B:[2, 3, 5]-> L:0,|| B:[2]-> L:1, B:[4]-> L:0,|| B:[1]-> L:1, B:[15]-> L:0,|| B:[3]-> A:4, B:[1, 2, 5, 6, 21]-> A:0,|| B:[3]-> L:1, B:[2, 5]-> L:0,|| B:[1]-> A:9, B:[14, 3, 5, 6, 10]-> A:5,|| B:[3, 4]-> A:1, B:[12, 1, 8]-> L:0,|| B:[2]-> L:1, B:[3, 4]-> L:0,||
39 B:[2]-> L:0, B:[3]-> L:1,|| B:[6]-> A:2, B:[1, 7, 9]-> L:1,|| B:[1]-> A:4, B:[13, 2, 4]-> A:3,|| B:[4, 8]-> A:5, B:[11, 1, 5, 9]-> A:4,|| B:[1]-> L:1, B:[2]-> L:0,||
40 B:[1]-> L:1, B:[2]-> A:6,|| B:[1]-> A:5, B:[3, 9]-> L:0,|| B:[3]-> L:0, B:[2, 4, 5]-> L:1,|| B:[1, 5]-> L:0, B:[3, 7]-> L:1,|| B:[5]-> L:0, B:[1, 2, 3]-> A:3,||
41 B:[1]-> L:0, B:[2]-> L:1,|| B:[6]-> L:1, B:[1, 3, 5, 17]-> A:2,|| B:[5]-> A:1, B:[2, 3, 4]-> L:1,||
42 B:[1]-> A:3, B:[2]-> L:0,|| B:[1]-> L:1, B:[2]-> A:2,||
43 B:[4]-> L:1, B:[2, 3]-> A:5,|| B:[1]-> L:1, B:[2]-> L:0,||
44 B:[3]-> L:1, B:[1, 17]-> A:1,||
45 B:[1]-> L:0, B:[2]-> A:3,||
46 B:[2]-> L:0, B:[3]-> A:0,||
47 B:[2]-> A:10, B:[5]-> L:1,||
48 B:[1]-> L:1, B:[4]-> L:0,||

```

Figure 19: Decision Tree for Dress_Set dataset built using CART.

```

26 A:4
27 B:[a, l, n]-> A:19, B:[p, c, s, f, y]-> L:p,||
28 B:[r]-> L:p, B:[u, w, h, k, n]-> A:11,||
29 B:[k]-> L:p, B:[s, f]-> A:1,||
30 B:[g]-> L:p, B:[s, f, y]-> A:21,||
31 B:[l]-> A:2, B:[p, d, u, w, g, m]-> L:e,||
32 B:[w]-> L:p, B:[c, n]-> L:e,||

```

Figure 20: Decision Tree for Mushroom dataset built using CART

From the above two figures, it can be observed that there is a vast difference in the size of the trees created for the two datasets using CART. Also, the anomaly of time difference (Anomaly-1) discussed earlier in Experiment 1 is because building this huge binary tree using CART for Dress_Set dataset requires more time than the other.

From our observations, we see that Dress_set dataset is more impure than Mushroom dataset. As CART builds a binary tree, for an impure dataset, most of the attributes will be partitioned/split multiple times. These splits occur due to the impure combinations/subsets of the attribute's distinct values. These splits occur on various levels of the Decision tree leading to increase in the size of the tree.

From this observation we can conclude that number of levels/size of the tree is directly proportional to the impurity of the dataset in case of CART.

For an impure dataset, we see that ID3 performs better as the attribute is split only once over its distinct values (cardinality).

However, for a pure dataset, CART is a better option as it can partition the attributes into mostly pure partitions, leading to less branches or partitions over the distinct levels. For the pure dataset, the attribute is split into fewer partitions (or split fewer times) because the purity of the subsets/combinations is high, leading to denser Decision Tree. ID3 splits the purer dataset into more partitions which are not necessarily required.

Analysing the anomaly observed in accuracy change due to dataset sizes (Anomaly-2), we believe that even though generally larger training datasets builds the classifier with better accuracy, there are a few situations where reducing the dataset improves dataset's purity. This might be because the records that were removed decreased the purity of the dataset. And from above analysis of purity, we see that a dataset with higher purity builds to a classifier with higher accuracy.

IX. Conclusion:

Over the course of this assignment, we have implemented and validated the Decision Tree Classifier. Decision Tree Classifiers show a great deal of potential in many pattern recognition problems. One of the primary features of Decision Tree Classifier is its flexibility; for example, the capability of using different feature subsets and decision rules at different stages of classification and the capability of trade-offs between classification accuracy and time space efficiency.

X. Bibliography

[1] Jiawei Han, Micheline Kamber, "Data Mining Concepts and Techniques", 2nd edition.

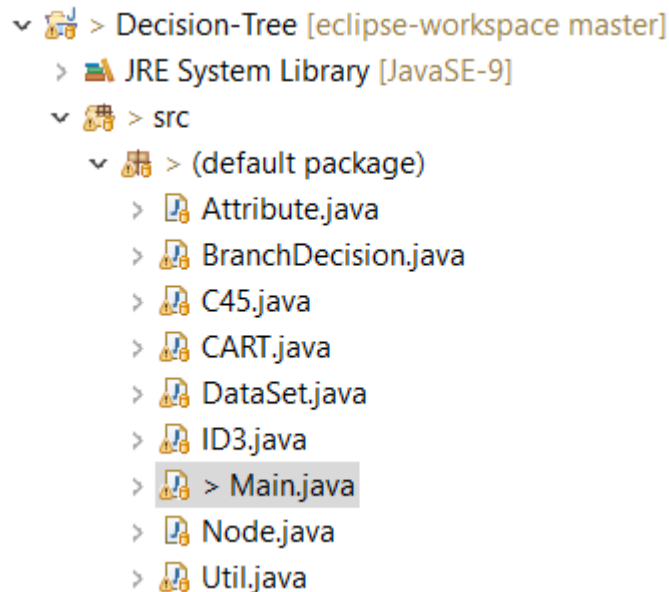
[2] Dataset sources

- <https://archive.ics.uci.edu/ml/datasets/mushroom>
- https://archive.ics.uci.edu/ml/datasets/dress_set

XI. Appendix:

Please find all source code attached with the mail.

File Structure:



Main.java class

```
public class Main {
    //Dress_Set, mushroom, test
    //Dress_Set_P, mushroom_p, test_p
    static final String trainingFilePath = "Dress_Set_300.csv";
    static final String validationFilePath = "Dress_Set_P_300.csv";
    static final String method = "CART"; // ID3, C45, CART

    public static void main(String[] args) {
        DataSet dataset = Util.readCSV(trainingFilePath);

        /*Print Training Dataset*/
        // Util.printDataSet(dataset);

        /*Print Majority Class*/
        // System.out.println(dataset.majorityClass);

        long startTime = System.currentTimeMillis();
        List<Integer> attrList = new ArrayList();
        for (int i = 0; i < dataset.data.get(0).length - 1; i++) {
            attrList.add(i);
        }
        Node root = buildTree(dataset, attrList);

        long endTime = System.currentTimeMillis();
        System.out.println("Training Run Time:" + (endTime - startTime));

        //System.out.println("Tree Built");

        /*Print Tree*/
        Util.printTree(root);
    }
}
```

```

        startTime = System.currentTimeMillis();
        validation(root);
        endTime = System.currentTimeMillis();
        System.out.println("Validation Run Time:" + (endTime - startTime));
    }

    private static void validation(Node root) {
        DataSet validationData = Util.readCSV(validationFilePath);
        double count = 0;
        for (String[] strings : validationData.data) {
            String result = predict(root, strings);
            /*if (null == result) {
                System.out.println();
            }*/
            if (null != result &&
result.equalsIgnoreCase(strings[strings.length - 1])) {
                count++;
            }
        }
        System.out.println(count / validationData.data.size());
    }

    public static String predict(Node root, String[] inpRecord) {
        if (root.attr == -1) {
            return root.label;
        }
        for (BranchDecision branch : root.branches) {
            if (null != branch.equalTo) {
                if (branch.equalTo.equals(inpRecord[root.attr])) {
                    return predict(branch.resultant, inpRecord);
                }
            } else {
                if (branch.attrSet.contains(inpRecord[root.attr])) {
                    return predict(branch.resultant, inpRecord);
                }
            }
        }
        return null;
    }

    public static Node buildTree(DataSet dataset, List<Integer> attrList) {
        Node curr;
        if (dataset.classes.keySet().size() == 1) {
            return new Node(dataset.majorityClass);
        }
        if (attrList.isEmpty()) {
            return new Node(dataset.majorityClass);
        }
        if (method.equalsIgnoreCase("ID3")) {
            curr = ID3.AttributeSelector(dataset, attrList);
        } else if (method.equalsIgnoreCase("C45")) {
            curr = C45.AttributeSelector(dataset, attrList);
        } else {
            curr = CART.AttributeSelector(dataset, attrList);
        }

        /* make condition here-> moved to attrSelection */
    }

```

```

        // attrList.remove(Integer.valueOf(curr.attr));

        if (curr.attr == -1)
            return new Node(dataset.majorityClass);

        for (BranchDecision branch : curr.branches) {
            if (branch.partitionedDataSet.data.isEmpty()) {
                return new Node(dataset.majorityClass);
            } else {
                branch.resultant = buildTree(branch.partitionedDataSet,
attrList);
            }
        }

        return curr;
    }
}

```

Node.java class

```

public class Node {

    int attr = -1;
    Set<String> attrSet; // not used
    String label = "ERROR";

    ArrayList<BranchDecision> branches;

    Node(String label){
        this.label = label;
        this.branches = new ArrayList<>();
        attrSet = new HashSet<String> ();
    }

    Node(int attr){
        this.attr = attr;
        this.branches = new ArrayList<>();
        attrSet = new HashSet<String> ();
    }
}

```

BranchDecision.java class

```

public class BranchDecision {

    String equalTo;
    Set<String> attrSet;
    DataSet partitionedDataSet;
    Node resultant;
}

```

ID3.java class

```

public class ID3 {

    public static Node AttributeSelector(DataSet dataset, List<Integer>
attrList) {

        Double infoD = 0.0;
        for (String c : dataset.classes.keySet()) {
            double temp = dataset.classes.get(c) / dataset.data.size();
            infoD -= temp * (Math.Log(temp) / Math.Log(2));
        }

        Double maxGain = -1.0;
        int selectedAttr = -1;
        Map<String, DataSet> selectedDatasets = null;
        for (Integer i : attrList) {
            Map<String, DataSet> djs = new HashMap();
            for (String[] str : dataset.data) {

                if (!djs.containsKey(str[i])) {
                    djs.put(str[i], new DataSet());
                }
                djs.get(str[i]).data.add(str);

                if (!djs.get(str[i]).classes.containsKey(str[str.length
- 1])) {
                    djs.get(str[i]).classes.put(str[str.length - 1],
1.0);

                    if (null == djs.get(str[i]).majorityClass) {
                        djs.get(str[i]).majorityClass =
str[str.length - 1];
                    }
                } else {
                    djs.get(str[i]).classes.put(str[str.length - 1],
djs.get(str[i]).classes.get(str[str.length - 1]) + 1);
                }

                if
(djs.get(str[i]).classes.get(djs.get(str[i]).majorityClass) <
djs.get(str[i]).classes
.get(str[str.length - 1])) {
                    djs.get(str[i]).majorityClass = str[str.length -
1];
                }
            }
            Double infoDA = 0.0;
            for (DataSet dj : djs.values()) {
                double weight = ((double) dj.data.size()) /
dataset.data.size();

                Double infoDj = 0.0;
                for (Map.Entry<String, Double> c :
dj.classes.entrySet()) {
                    double temp = c.getValue() / dj.data.size();
                    infoDj -= temp * (Math.Log(temp) / Math.Log(2));
                }
                infoDA += weight * infoDj;
            }
            Double gain = (infoD - infoDA);
            if (gain > maxGain) {

```

```

        maxGain = infoD - infoDA;
        selectedAttr = i;
        selectedDatasets = djs;
    }
    if(selectedAttr == -1)
        System.out.println("ERROR");
}

Node result = new Node(selectedAttr);
for (Map.Entry<String, DataSet> dj : selectedDatasets.entrySet()) {
    BranchDecision branch = new BranchDecision();
    branch.equalTo = dj.getKey();
    branch.partitionedDataSet = dj.getValue();
    result.branches.add(branch);
}
attrList.remove(Integer.valueOf(result.attr));

// System.out.println(selectedAttr + ":" + maxGain);

return result;
}
}

```

C45.java class

```

public class C45 {

    public static Node AttributeSelector(DataSet dataset, List<Integer>
attrList) {

        Double infoD = 0.0;
        for (String c : dataset.classes.keySet()) {
            double temp = dataset.classes.get(c) / dataset.data.size();
            infoD -= temp * (Math.Log(temp) / Math.Log(2));
        }

        Double maxGain = -1.0;
        int selectedAttr = -1;
        Map<String, DataSet> selectedDatasets = null;
        for (Integer i : attrList) {
            Map<String, DataSet> djs = new HashMap();
            for (String[] str : dataset.data) {

                if (!djs.containsKey(str[i])) {
                    djs.put(str[i], new DataSet());
                }
                djs.get(str[i]).data.add(str);

                if (!djs.get(str[i]).classes.containsKey(str[str.length
- 1])) {
                    djs.get(str[i]).classes.put(str[str.length - 1],
1.0);

                    if (null == djs.get(str[i]).majorityClass) {
                        djs.get(str[i]).majorityClass =
str[str.length - 1];
                    }
                } else {
                    djs.get(str[i]).classes.put(str[str.length - 1],

```

```

        djs.get(str[i]).classes.get(str[str.length - 1]) + 1);
    }

    if
(djs.get(str[i]).classes.get(djs.get(str[i]).majorityClass) <
djs.get(str[i]).classes
        .get(str[str.length - 1])) {
        djs.get(str[i]).majorityClass = str[str.length -
1];
    }
}
Double infoDA = 0.0;
Double splitInfoDA = 0.0;

for (DataSet dj : djs.values()) {
    double weight = ((double) dj.data.size()) /
dataset.data.size();
    Double infoDj = 0.0;
    for (Map.Entry<String, Double> c :
dj.classes.entrySet()) {
        double temp = c.getValue() / dj.data.size();
        infoDj -= temp * (Math.Log(temp) / Math.Log(2));
    }
    infoDA += weight * infoDj;
    splitInfoDA -= weight * (Math.Log(weight) /
Math.Log(2));
}
Double gain = 0.0;
if(splitInfoDA != 0)
    gain = (infoD - infoDA)/splitInfoDA;
if (gain > maxGain) {
    maxGain = infoD - infoDA;
    selectedAttr = i;
    selectedDatasets = djs;
}
if(selectedAttr == -1)
    System.out.println("ERROR");
}

Node result = new Node(selectedAttr);
for (Map.Entry<String, DataSet> dj : selectedDatasets.entrySet()) {
    BranchDecision branch = new BranchDecision();
    branch.equalTo = dj.getKey();
    branch.partitionedDataSet = dj.getValue();
    result.branches.add(branch);
}
attrList.remove(Integer.valueOf(result.attr));

// System.out.println(selectedAttr + ":" + maxGain);

return result;
}
}

```

CART.java class

```

public class CART {

    public static Node AttributeSelector(DataSet dataset, List<Integer>
attrList) {

        double giniD = getGini(dataset);

        Double maxGini = -1.0;
        int selectedAttr = -1;
        BranchDecision leftBranch = new BranchDecision();
        BranchDecision rightBranch = new BranchDecision();
        for (Integer i : attrList) {
            Map<String, DataSet> djs = new HashMap();
            for (String[] str : dataset.data) {

                if (!djs.containsKey(str[i])) {
                    djs.put(str[i], new DataSet());
                }
                djs.get(str[i]).data.add(str);

                if (!djs.get(str[i]).classes.containsKey(str[str.length
- 1])) {
                    djs.get(str[i]).classes.put(str[str.length - 1],
1.0);

                    if (null == djs.get(str[i]).majorityClass) {
                        djs.get(str[i]).majorityClass =

str[str.length - 1];
                    }
                } else {
                    djs.get(str[i]).classes.put(str[str.length - 1],

djs.get(str[i]).classes.get(str[str.length - 1]) + 1);
                }

                if
(djs.get(str[i]).classes.get(djs.get(str[i]).majorityClass) <
djs.get(str[i]).classes

.get(str[str.length - 1])) {
                    djs.get(str[i]).majorityClass = str[str.length -
1];
                }
            }

            /*if(djs.keySet().size() == 1)
                System.out.println("ERROR");*/
            List<String> dVAttr = new ArrayList(djs.keySet());
            for (int j = 1; j < djs.keySet().size(); j++) {
                Set<Set<String>> combinations =
Util.combinations(dVAttr, j);
                for (Set<String> branch1 : combinations) {
                    DataSet D1 = Util.combineDataSet(branch1, djs);
                    Set<String> branch2 = new HashSet(djs.keySet());
                    branch2.removeAll(branch1);
                    DataSet D2 = Util.combineDataSet(branch2, djs);
                    double giniA = getGiniA(dataset, D1, D2, giniD);
                    if (giniA > maxGini) {
                        maxGini = giniA;
                        selectedAttr = i;
                        leftBranch.attrSet = branch1;

```



```

        rightBranch.attrSet = branch2;
        leftBranch.partitionedDataSet = D1;
        rightBranch.partitionedDataSet = D2;
    }
    if(selectedAttr == -1)
        System.out.println("ERROR");
    }
}
/*if(selectedAttr == -1)
    System.out.println("ERROR");*/

}

Node result = new Node(selectedAttr);
if(null != leftBranch.partitionedDataSet)
    result.branches.add(leftBranch);
if(null != rightBranch.partitionedDataSet)
    result.branches.add(rightBranch);

//attrList.remove(Integer.valueOf(result.attr));

// System.out.println(selectedAttr + ":" + maxGain);

return result;
}

private static double getGini(DataSet dataset) {
    double giniD = 0.0;
    for (String c : dataset.classes.keySet()) {
        double pi = dataset.classes.get(c) / dataset.data.size();
        giniD -= pi * pi;
    }
    giniD++;
    return giniD;
}

private static double getGiniA(DataSet D, DataSet D1, DataSet D2, double
giniD) {
    double giniDA = 0.0;
    giniDA = (((double)(D1.data.size()) / D.data.size()) * getGini(D1))
+ (((double)(D2.data.size()) / D.data.size()) * getGini(D2));
    return giniD - giniDA;
}
}

```