Decision Tree Algorithm in Python

By Alan George & Matthew Huber

Instructor: Alex Doboli

ESE 327

Section 1: Implementing Decision Tree Algorithm

When implementing the decision tree algorithm in python, we based our algorithm on the one covered in lecture. In class, we learned that we have to first build the tree using 70% of the data from a dataset, and then use the remaining 30% of the data to test the accuracy of the tree. The first function we worked on was the GenerateDecisionTree function. We have our function take in five parameters; Dataset, which is simply the entire dataset read from the file formatted to be a list of lists, attribute_list and attribute_list_ref, which both reference the list of the attributes of the dataset. We added attribute_list_ref to keep the list constant for use of indexing since the values are removed from the attribute_list variable. This will be explained more later. The selection_string parameter is used to select the attribute selection method and the last_col parameter is used as the index for the class column of the dataset. The attribute_list and attribute_list_ref variables are created by us manually, and we also input the last_col and selection_string parameters ourselves.

Within the GenerateDecisionTree function, we followed the pseudocode gone over in lecture pretty closely. We first start out by creating a new node we call root. Node is a class that we created. When a node object is initialized, it takes in two parameters. These are attribute, which represents the splitting criterion of the node, and parent_label which represents the choice made by the splitting criterion of a node's parent. An example of this would be if a node had a parent who's attribute was age, its parent_label would be either <=30, 31…40, or >40. In the case of the root node, its parent_label is set to None. Each node also has a list of its children. The Node class has three other functions. The first is addChild which accepts the child node as a parameter and appends it to the children_list of the node. The second is addFeature which changes the parent_label value to the parameter of the function. The last is display which recursively displays the decision tree.

Our first ending condition in the GenerateDecisionTree function checks the dataset to see if all remaining rows belong to the same class. If this is true, then the function returns a leaf node with the attribute value labeled as the class. Here we use the last_col index to check the class value of each row of the dataset. Our second ending condition of the function checks to see if the length of the attribute_list variable is zero. If this is true, then the function returns a leaf node with its attribute value labeled as the majority class remaining in the dataset. To do this we created a function called majorityClasses which accepts the Dataset and last_col values as parameters. This function uses a helper function we created called numOfClasses which accepts the same parameters. The numOfClasses function iterates through the database rows and creates a list of all class labels and a corresponding list with the frequency that each of them appear in the dataset. The majorityClasses function then uses these lists and returns the label with the highest frequency. The node's attribute value is assigned to this value and the function returns that node

Next, we move onto the AttributeSelectionMethod function. This function accepts the same parameters as the GenerateDecisionTree function and returns the attribute and the splitting_feature for the gini index case. The function first checks the selection_string parameter to see which method it wants to use. The first method is the Gini index. To do this we first found the gini index for the entire dataset based on the formula discussed in lecture, and then we found the gini index of each attribute still in the attribute list, which required the previous gini index of the entire dataset. Finally, we found the change in gini of each attribute and returned the value and feature that was the maximum change in gini.

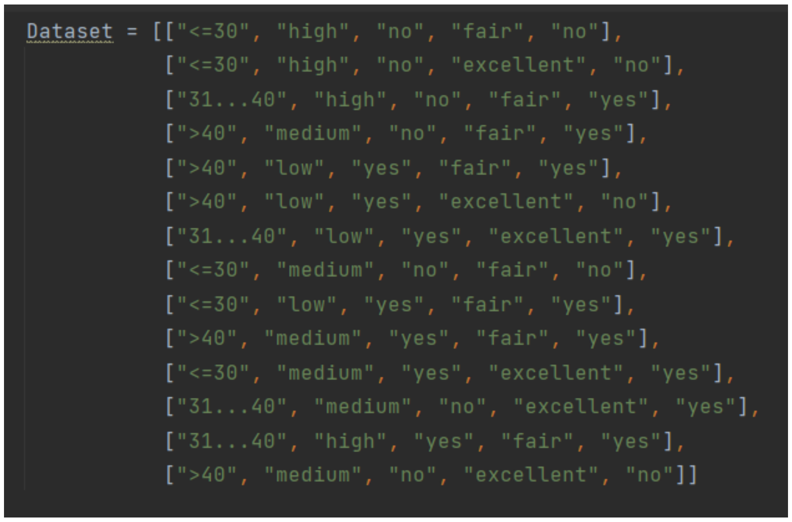
The other two methods are information gain and gain ratio. The information gain is found by subtracting the information from a specific attribute from the information from the whole dataset. The gain ratio is just the information gain divided by split info. Both of these methods return the attribute that provided the maximum value for the method. We also used the logic and formulas for each method that we learned in lecture. In our results, we found that the information gain and gain ratio produced very similar results.

After the attribute is returned from the function, we assign the node's attribute value with that value. We then store the index of the attribute in the list before we remove it from the list. From there, we have an if statement checking if the selection_statement value was "GINI". If it was, we must take into account that gini is binary, and treat it a bit differently. We first use the previously assigned index variable to iterate through the Dataset and to sort the rows into two lists: one in which the specified feature matches the one returned from the attribute selection method function, and one for the rows that do not match. From here we have the same ending condition in which we check each of the newly created lists to see if all rows have the same class. If this is true, you create a child node labeled with the splitting feature as the attribute, and then append the child node to the original node's children list. If this is not true, we recursively call the GenerateDecisionTree function on each of the newly created datasets before appending the child nodes to the root's children list. Finally, you return the root node.

If the selection_string was not "GINI", then you would do the code in the else part of the if statement. Here, you would iterate through the Dataset and create a list of all the features of the attribute designated by the index variable. Then for each feature in the list, you create a smaller sub dataset of all rows that match that feature. If the list is empty and none of the rows matched the feature, then a child node would be created with the attribute being labeled with the most frequent class in the dataset. This child node would be appended to the root nodes children list. If the list was not empty, then the GenerateDecisionTree function would be recursively called on the sub dataset, and the output of that function would be the child node appended to the root node's children list. Finally, the GenerateDecisionTree function returns the root node variable.

Section 2: Testing Decision Tree Algorithm on a Small Dataset

With a small dataset, we were able to accurately create decision trees made from the three metrics discussed in class (Gini Index, Information Gain, and Gain Ratio). The small dataset we chose to test our algorithm on first is the same 14-instance dataset we covered in class, that had the attributes "age", "income". "student", and "credit_rating", with the class "buys_computer". The "age" attribute has the features "<= 30", "31…40", and ">40". "income" has "low", "medium", and "high"; "student" has "yes" and "no", and "credit_rating" has "fair" and "excellent". The class "buys_computer" has the classes "yes" and "no". In terms of utilizing the dataset in our code, we wrote out the entire dataset into our main function and ran our code on the dataset, while the datasets that we used from the UC Irvine Machine Learning Repository utilize the pandas library to read the datasets as .csv files and to transform them into a "list of lists". This "list of lists" represents how the overall dataset is one whole list, with each of its elements being a row, which itself is represented as a list of its elements. We also represented our small dataset as a "list of lists" for consistency. Below is the small dataset represented as a "list of lists" in our main function for our Decision Tree Algorithm.



```
Dataset = [["<=30", "high", "no", "fair", "no"],
           ["<=30", "high", "no", "excellent", "no"],
           ["31...40", "high", "no", "fair", "yes"],
           [">40", "medium", "no", "fair", "yes"],
           [">40", "low", "yes", "fair", "yes"],
           [">40", "low", "yes", "excellent", "no"],
           ["31...40", "low", "yes", "excellent", "yes"],
           ["<=30", "medium", "no", "fair", "no"],
           ["<=30", "low", "yes", "fair", "yes"],
           [">40", "medium", "yes", "fair", "yes"],
           ["<=30", "medium", "yes", "excellent", "yes"],
           ["31...40", "medium", "no", "excellent", "yes"],
           ["31...40", "high", "yes", "fair", "yes"],
           [">40", "medium", "no", "excellent", "no"]]
```

Fig. 1

When we produce the Gini Index-decided decision tree, we believe that the decision tree should look like below:

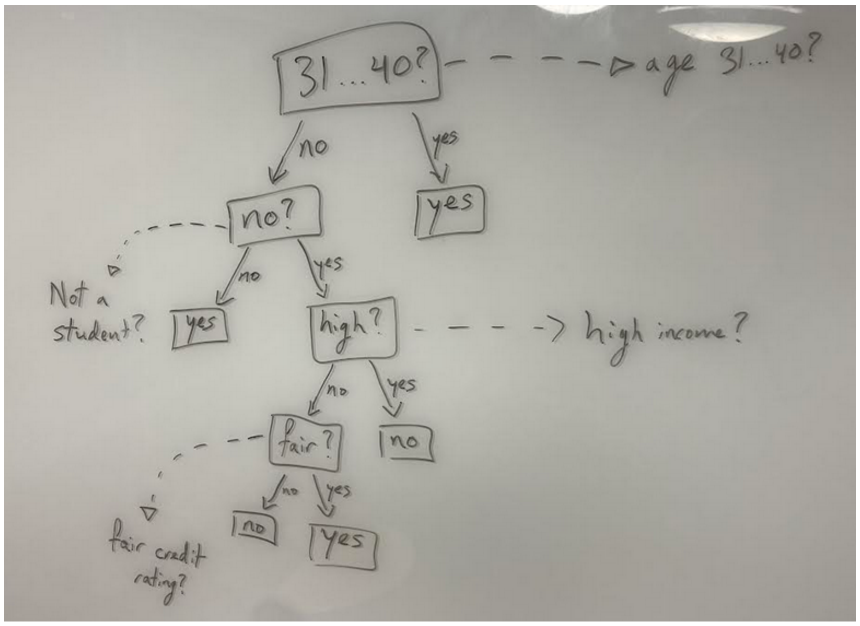


Fig. 2

With the Gini Index, the goal for our decision tree is act as a binary decision tree where each split node represents the best splitting criterion for that attribute for the data points in question. Finding the best splitting criterion does not solely depend on the individual features for a certain attribute (like for “age” as an example, the individual features would be “<=30”, 31…40” and “>40”) but their unique combinations between them (like “<=30”, 31…40”, “>40”, “<=30, 31…40”, “<=30, 40” and “31…40, >40”). These unique combinations do not include the empty combination “{}” or the combination that includes all features “<=30, 31…40, >40”. From these combinations, the best splitting criterion is found by which combination has the largest difference between the Gini Index for the dataset in question and the Gini Index for the combination.

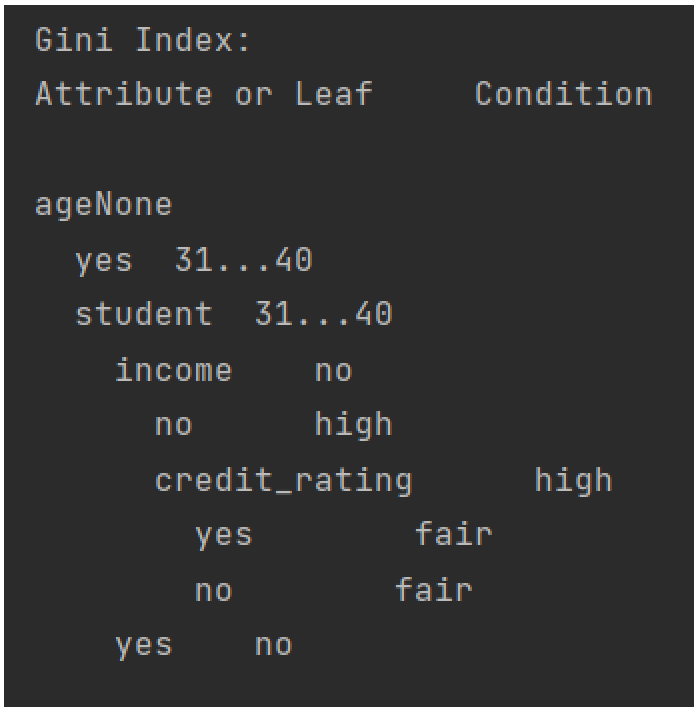
For each split node to be decided, the best splitting criterion is chosen from all of the combinations for each attribute. In terms of computing, this can be very intensive depending on the number of attributes that you have in your dataset, and how many specific features each attribute has in the dataset. In the case for the small dataset, there are only 4 attributes in question, with the maximum number of features for an attribute to have only being 3, so finding the best splitting criterion should not be too intensive. What is also important to note is that finding the splitting criterion should partition the dataset into only two subsets, with one half of the dataset satisfying the criterion (if a row in the dataset has one of its features included in the splitting criterion; the "yes” condition on the arrows following to a child node) and the oth
er half not satisfying the criterion (the “no” condition on the arrows going to the child node).

In terms of reading our decision tree, from the whole dataset, the best splitting criterion was if a row in the dataset has “31…40” as its feature for the “age” attribute. For each of a split node’s children, it can either be a split node or a leaf node. In this case, we have both, where the condition that a row in the dataset does not have “31…40” as its feature for “age” leads to another split node deciding if a row in the dataset has “no” in the “student” attribute (or if a row in the dataset is a student or not; “no?”), and the condition satisfying the “31…40” splitting criterion leads to a leaf node with the value of “yes”. Due to the nature of our data, this is something that is entirely possible, since from the dataset, all 4 data points where “31…40” is its feature for “age” all have the class “yes” for “buys_computer”. As such, that is what the two immediate children from the first split node mean in the decision tree above.

When a splitting criterion is chosen for a split node, the attribute that is associated with that splitting criterion is removed from the current list of attributes available, as the attribute would already be referenced by that node from all data points. The goal with this decision tree and for all decision trees is to reduce the amount of attributes available to look at for a data point as you traverse down a decision tree. The more you traverse down, the more attributes you would have already considered, and the smaller the dataset you are looking at becomes.

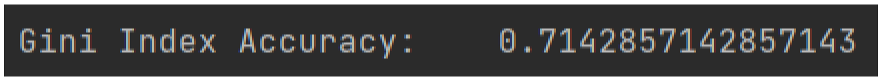
With that in mind, most of the nodes (split and leaf) were decided with this approach, where a splitting criterion is calculated and selected for each split node (“high?” meaning if a data point has high income or not, “fair?” meaning if a data point has a fair credit_rating or not), and a leaf node is decided if all of the nodes at that point have the same class. However, there are instances where we reach an outcome in the decision tree where all of the attributes in the dataset have been considered, but there are some data points that still remain which all do not share the same class (their classes differ between them). In those cases, then the majority class in that situation must be calculated, and that majority class becomes the value for that respective leaf node. In some situations, this leads to a decrease in the accuracy of the decision tree, as an apparent amount of data points from a dataset would be miscategorized due to the majority class calculated for less “clear-cut” decisions in the decision tree.

In our small dataset, two of our leaf nodes were this case; one of them being the “yes” leaf node that is the child of the “fair?” split node, and the other is the “yes” leaf node from the “no?” split node. Due to these situations, we expect that 4 out of the 14 data points should be categorized incorrectly, as these 4 data points fall under the minority data points considered as the children of their respective split nodes. Miscategorizing 4 out of the 14 data points gives us an expected accuracy of 10/14, or 0.71428…, or around 71.4% accuracy. Below is the output that was generated by our code in displaying what the Gini Index-decision tree looks like, and what the accuracy calculated was:



```
Gini Index:
Attribute or Leaf     Condition

ageNone
  yes  31...40
  student  31...40
    income    no
      no      high
      credit_rating      high
        yes        fair
        no        fair
    yes    no
```

Fig. 3



```
Gini Index Accuracy:    0.7142857142857143
```

Fig, 4

To read this output correctly, we established some rules in terms of what nodes are the children of other nodes. The root node is signified by the node that is not indented at all, in this case being “ageNone”, which represents the attribute of the root node is “age” and the “parent_label” of the node is “None” (or NULL like in C). The root node’s children are the “yes 31…40” leaf node and the “student   31…40” node, since they both share the same level of indentation as each other. Since there are only two children to each split node in the Gini Index-decision tree, the first child represents the node that satisfies the splitting criterion, and the second child represents the node that does not satisfy the splitting criterion. For each of the children nodes, the “parent_label” represents the splitting criterion of its parent node. With this notation, we can see that our expected decision tree created with the Gini Index matches our output for the generated Gini Index-decision tree, along with our expected accuracy matching the actual accuracy produced by our algorithm. What we believe is that given that the exact same dataset can be used to create the Gini Index decision tree, the exact same decision tree can be produced (the algorithm is inherently deterministic). However, as something we noticed when testing with datasets from UC Irvine, due to how we split the original dataset into two datasets for testing and training, the splitting inherently produces different datasets for the training and testing every time, which can produce different results and a different tree altogether when using the Gini Index metric.

Below are two figures representing our expected decision tree for both the Information Gain and Gain Ratio metrics, along with our generated decision trees for both metrics:

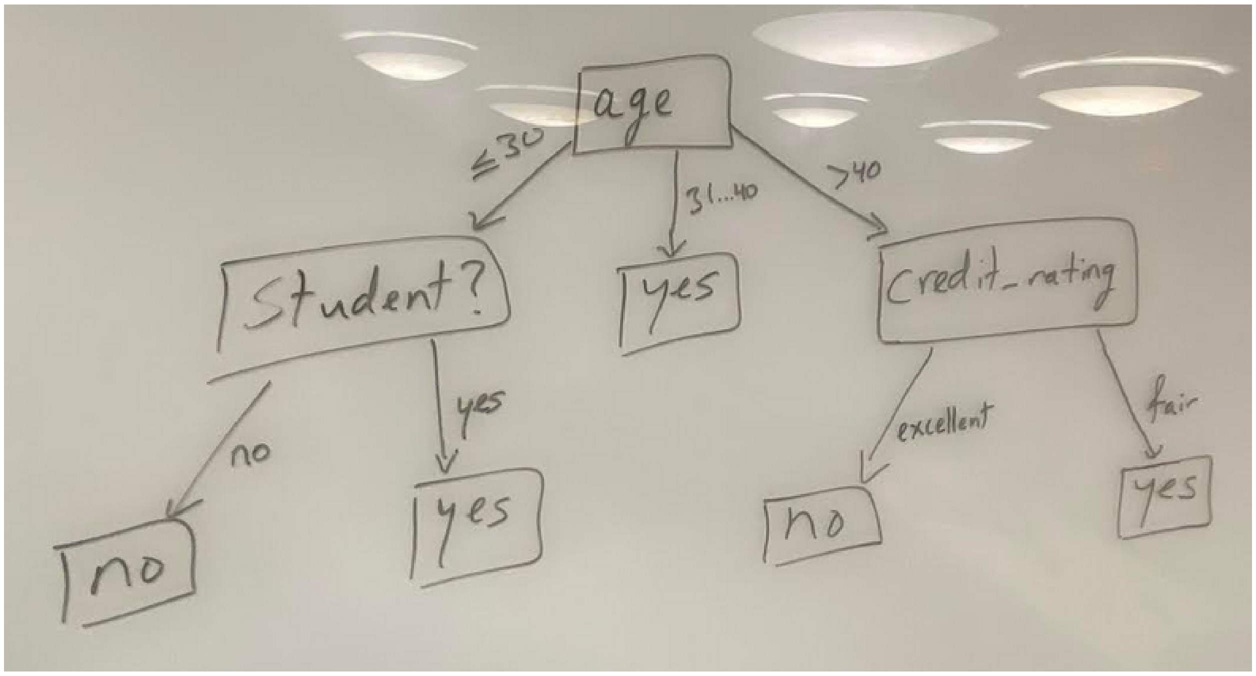

age
≤30
31...40
>40
Student?
yes
Credit_rating
no
yes
excellent
fair
no
yes
no
yes

Fig. 5

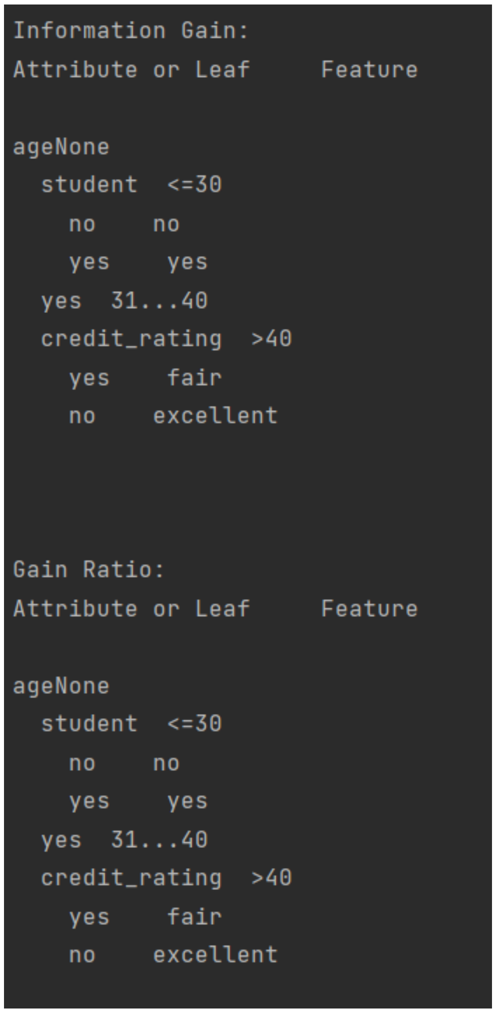
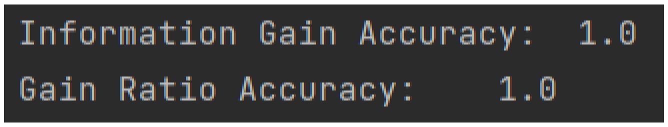

```
Information Gain:
Attribute or Leaf     Feature

ageNone
  student  <=30
    no    no
    yes    yes
  yes  31...40
  credit_rating  >40
    yes    fair
    no    excellent



Gain Ratio:
Attribute or Leaf     Feature

ageNone
  student  <=30
    no    no
    yes    yes
  yes  31...40
  credit_rating  >40
    yes    fair
    no    excellent
```

Fig. 6

Between Information Gain and Gain Ratio as metrics, both metrics share a lot of similarities. This is due to how both metrics heavily utilize the Gain of their datasets when deciding the best attribute to pick as the split node. With both metrics, unlike Gini Index, the attribute picked allows for more than two children to exist for a split node, where the children themselves represent data points that correlate to each respective feature of the parent split node's attribute. For example in Fig. 5, the root node's attribute is "age", and the children from the root node all have separate attributes ("student", "yes", "credit_rating"), but their parent_labels represent the 3 different features representing “age” (“<=30”, “31…40”, and “>40” respectively).

With both Information Gain and Gain Ratio, the children nodes are categorized as either split nodes (where each split node now represents a new attribute that has the best Gain compared to the other attributes), or as a leaf node (created similarly with all or the majority of data points sharing the same classes, just like with Gini Index). Also like with Gini Index, if an attribute is picked, then that attribute cannot be picked again later on in the decision tree. As such we believe that our expected result for the decision tree for Information Gain and Gain Ratio should be the same as in Fig. 5. Both metrics will produce the same tree since both metrics utilize Gain extensively when picking an attribute for a split node, along with Gain Ratio as a metric overall being a more reliable and stable version of the Information Gain metric; producing almost similar to exactly similar decision trees between them .From Fig. 5, we also believe that the accuracy should also be 100% since all 14 of the data points from the small dataset can be correctly categorized by the decision tree. Below is a snippet from our console showing the accuracy of our generated Information Gain and Gain Ratio trees for the small dataset:



```
Information Gain Accuracy:  1.0
Gain Ratio Accuracy:    1.0
```

Fig. 7

Looking at Fig. 6, we can see that both the Information Gain and Gain Ratio metrics produce the same decision tree as each other, and as Fig. 5. Looking at Fig. 7 as well, our expected accuracies also match together with the generated accuracies for both metrics;

validating that our algorithm can correctly produce decision trees with Information Gain and Gain Ratio as metrics. From our small dataset, we are also confident that we can produce likewise results for larger datasets that can be found in the UC Irvine Machine Learning Repository.

Section 3: Decision Tree Algorithm on UC Irvine Datasets

Name	Instances	Attributes	Classes	Execution Time	Gini Accuracy	Information Gain Accuracy	Gain Ratio Accuracy
Cars	1728	6	4	12s	0.625	0.687	0.687
Nursery	12960	8	4	16min 14s	0.419	0.785	0.785
Lymphography	148	18	4	0s	0.5	1.0	1.0
Hayes-roth	160	4	3	0s	0.410	0.513	0.513
Tic-tac-toe	958	9	2	4s	0.638	0.767	0.767
Lenses	24	4	3	0s	0.714	1.0	1.0

Balance-Scale	625	4	3	3s	0.465	0.540	0.540
MONK's Problems	432	6	2	0s	0.558	0.829	0.907
Sample Dataset (NOT FROM UC IRVINE)	14	4	2	0s	0.714	1.0	1.0

During our very thorough testing period we found some weaknesses in our code. The execution time of our program exponentially increased as the number of features or instances increased. In the table above, an example of this is the Nursery Dataset. The dataset has over 12,000 instances and only 8 attributes. The program took over 16 minutes to execute and while our information gain and gain ratio attribute selection methods were accurate, the gini index method was not with a value of less than 0.5. This can be for a multitude of reasons, however we believe that the nature of our code does not bode well for the gini index method when there are a lot of instances in a dataset because we have multiple for loops that iterate over the entire dataset and most of the time these for loops are nested within or above other for loops that also dramatically increase the execution time of our program. In most cases where our execution wasn't instantaneous we found that the gini tree would get printed out after a long period of time and the rest of the program would finish in a small fraction of the time it took to generate the gini tree. One dataset we were unable to test was the Mushroom dataset. When we attempted to test this dataset, our program ran for over three hours before we terminated the execution. This dataset is very large, with over 8000 instances and 21 attributes, our program was not designed in a way to efficiently process that large amount of data. We feel that an algorithm with less of a dependency on for loops iterating through every row in the dataset would have performed better.

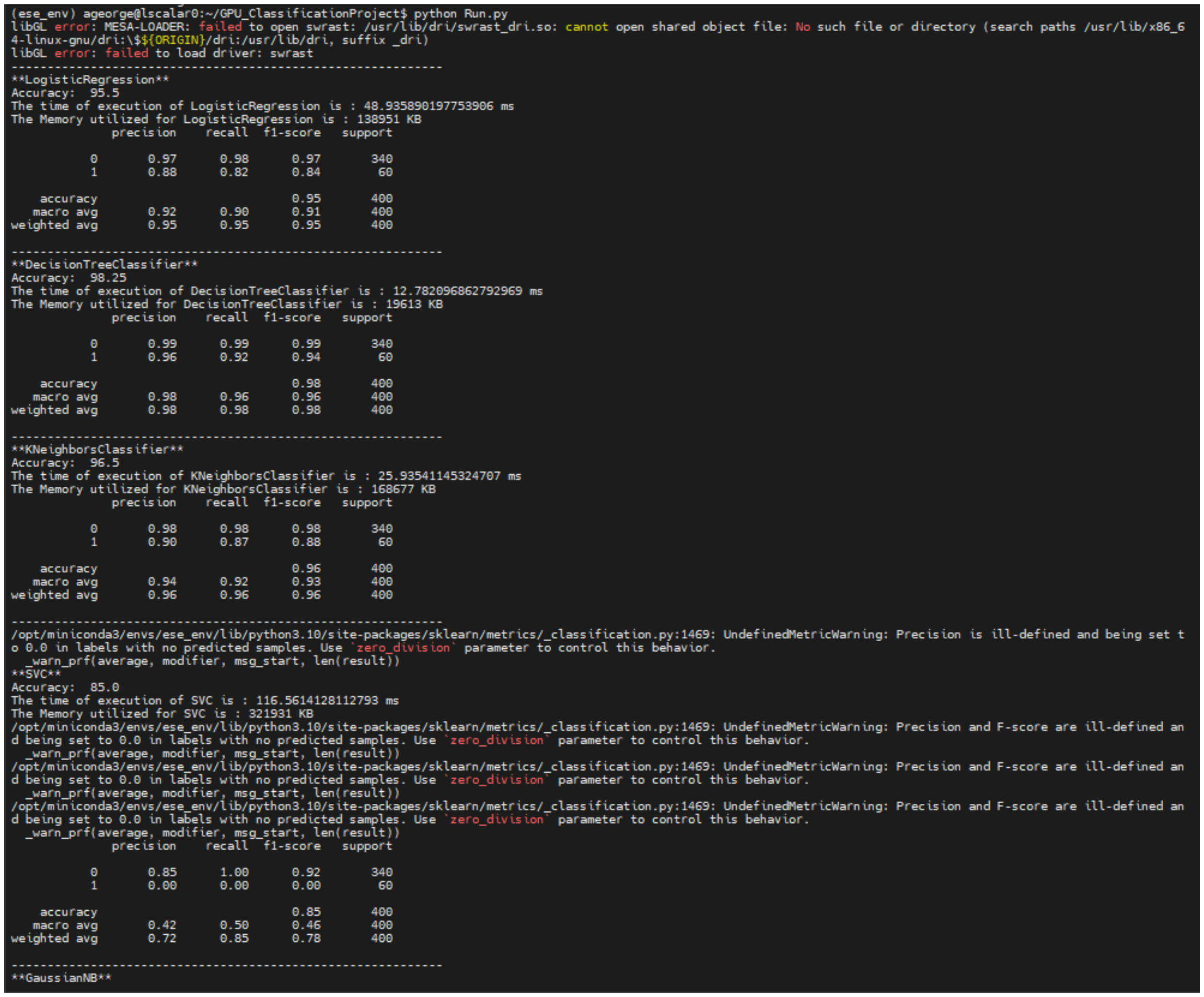
For each dataset we split it into a training set and a testing set with a 70-30 ratio. The pandas library offers a feature to randomly split data into smaller dataframes. This split is random and that reflects when we run the code because our accuracy outputs and tree outputs vary with each execution. We found that the difference between the accuracy measurements could be very drastic which does make it difficult to actually measure how accurate our decision tree algorithm is.

In our testing period, we tested datasets of all shapes and sizes. There was one dataset in particular that had an extremely high accuracy, and it was the Lenses dataset. This dataset is very small with only 24 instances, 4 features and 3 classes. After running our program, the dataset had a gini accuracy of 0.714, and information gain accuracy of 1.0, and a gain ratio accuracy of 1.0 as well. This dataset actually produced the same accuracy results as the sample dataset we studied in lecture which we found very interesting. We believe that this dataset was able to reach an accuracy of 1.0 while the other datasets were not able to because this dataset is so small that it is able to create leaf nodes more often than the large datasets that must resort to using the majority classes function to determine the leaf nodes because there is such a large disparity in the number of instances versus the number of features. The reason that the gini accuracy is so much lower than the other two methods is because of this same reason. The gini index method requires the use of the majority classes function while also limiting the number of child nodes to 2 for each node is not the most effective way of classifying data and therefore cannot compare to the other attribute selection methods.

Before we were able to input the datasets into our program, we had to perform a lot of preprocessing onto the data. A large number of the datasets had the features listed as numbers instead of the name of the feature. Our code was not able to execute with these datasets when they were filled with integers instead of unique labels for all of the features. Once we found and replaced all of the numbers with their corresponding labels, the program ran completely fine. We believe that the code did not run because since there were so many features with the same identifiers, our algorithm would not be able to distinguish between features for different attributes and that the accuracy of our algorithm would not be properly represented because of this.

Section 4: Decision Tree Algorithm Compared to Other Algorithms on GPU

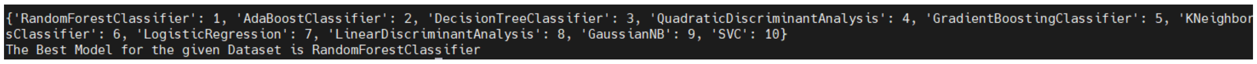
Below is the result of running the code given to test various classification algorithms together on the same dataset all on the GPU Server on campus. To test, we utilize MobaXTerm to sign into the GPU Server, and the original dataset used to test is called “original.csv”. The classifier given include Logistic Regression, Decision Tree, K-Neighbors, SVC, Gaussian Naive-Bayesian, Linear Discriminant Analysis, Quadratic Discriminant Analysis, Random Forest Classifier, Ada Boost, and Gradient Boosting Classifier. Fig. 8 represents the results of the classification Python script for each of the classifiers described above, which include the average accuracy, and the precision, recall, f1-score, and support for each of the classes in the dataset. What is important to note is due to the nature of the datasets you test, some of the metrics for the precision, recall, f1-score, and support cannot be calculated due to a “zero-division error”, which in this case, zero’s out all of the metrics for a certain class. (precision = recall = f1_score = 0.0). This in this case, inaccurately affects the overall accuracy of the classifier for the dataset, which misrepresents the reliability of a specific classifier.



```
(ese_env) ageorge@lscalar0:~/GPU_ClassificationProject$ python Run.py
libGL error: MESA-LOADER: failed to open swrast: /usr/lib/dri/swrast_dri.so: cannot open shared object file: No such file or directory (search paths /usr/lib/x86_6
4-linux-gnu/dri:\$${ORIGIN}/dri:/usr/lib/dri, suffix _dri)
libGL error: failed to load driver: swrast
------------------------------------------------------------
**LogisticRegression**
Accuracy:  95.5
The time of execution of LogisticRegression is : 48.935890197753906 ms
The Memory utilized for LogisticRegression is : 138951 KB
              precision    recall  f1-score   support

           0       0.97      0.98      0.97       340
           1       0.88      0.82      0.84        60

    accuracy                           0.95       400
   macro avg       0.92      0.90      0.91       400
weighted avg       0.95      0.95      0.95       400

------------------------------------------------------------
**DecisionTreeClassifier**
Accuracy:  98.25
The time of execution of DecisionTreeClassifier is : 12.782096862792969 ms
The Memory utilized for DecisionTreeClassifier is : 19613 KB
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       340
           1       0.96      0.92      0.94        60

    accuracy                           0.98       400
   macro avg       0.98      0.96      0.96       400
weighted avg       0.98      0.98      0.98       400

------------------------------------------------------------
**KNeighborsClassifier**
Accuracy:  96.5
The time of execution of KNeighborsClassifier is : 25.93541145324707 ms
The Memory utilized for KNeighborsClassifier is : 168677 KB
              precision    recall  f1-score   support

           0       0.98      0.98      0.98       340
           1       0.90      0.87      0.88        60

    accuracy                           0.96       400
   macro avg       0.94      0.92      0.93       400
weighted avg       0.96      0.96      0.96       400

------------------------------------------------------------
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision is ill-defined and being set t
o 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
**SVC**
Accuracy:  85.0
The time of execution of SVC is : 116.5614128112793 ms
The Memory utilized for SVC is : 321931 KB
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined an
d being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined an
d being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined an
d being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
              precision    recall  f1-score   support

           0       0.85      1.00      0.92       340
           1       0.00      0.00      0.00        60

    accuracy                           0.85       400
   macro avg       0.42      0.50      0.46       400
weighted avg       0.72      0.85      0.78       400

------------------------------------------------------------
**GaussianNB**
```

Fig. 8

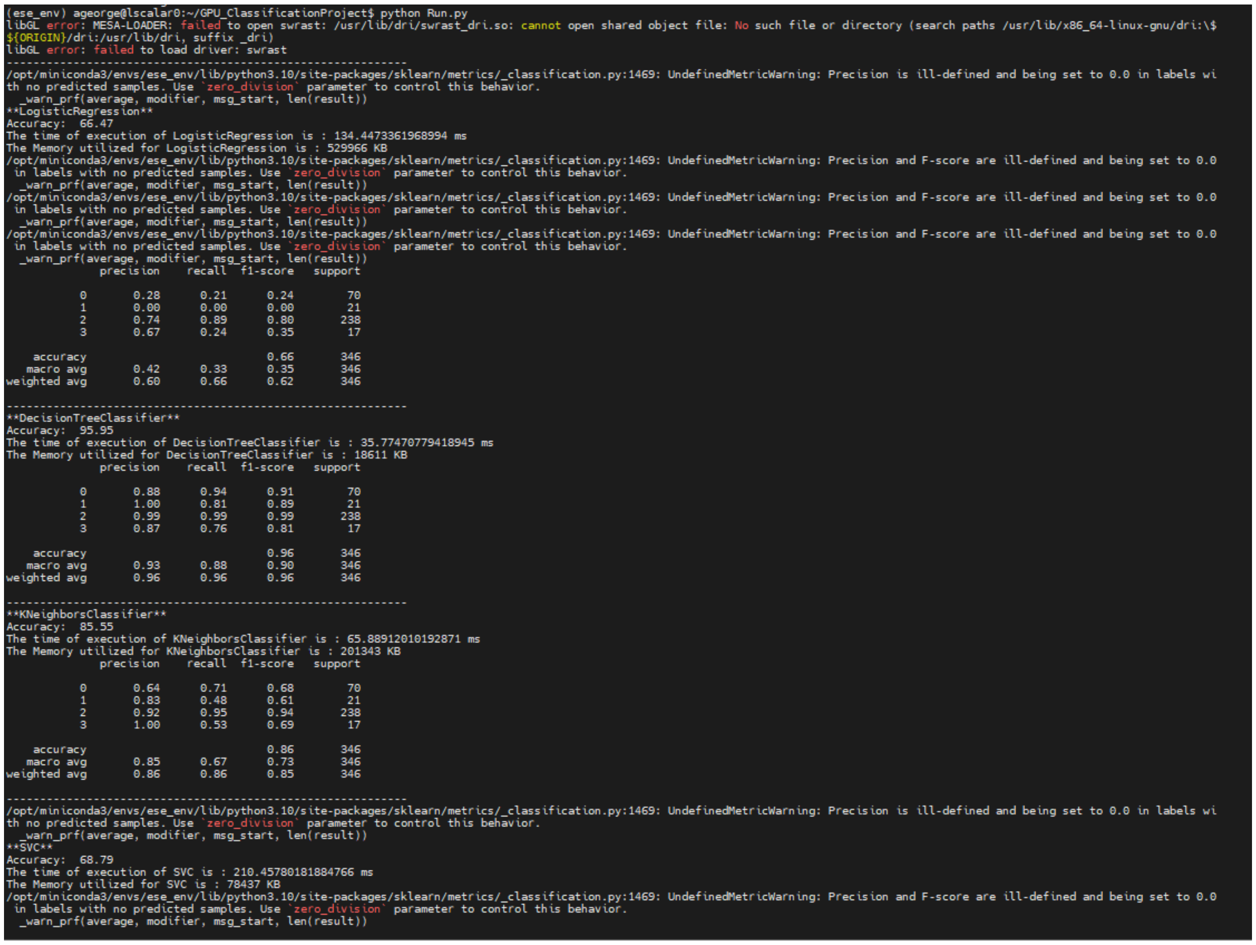
In Fig. 9 below, the results from the classification Python script represent a ranking of all 10 classifiers by their accuracy for the given dataset. Rank 1 represents the best classifier in terms of accuracy, and Rank 10 represents the worst accuracy. For the original test dataset used for the Python script, the rankings are seen below.



```
{'RandomForestClassifier': 1, 'AdaBoostClassifier': 2, 'DecisionTreeClassifier': 3, 'QuadraticDiscriminantAnalysis': 4, 'GradientBoostingClassifier': 5, 'KNeighbor
sClassifier': 6, 'LogisticRegression': 7, 'LinearDiscriminantAnalysis': 8, 'GaussianNB': 9, 'SVC': 10}
The Best Model for the given Dataset is RandomForestClassifier
```

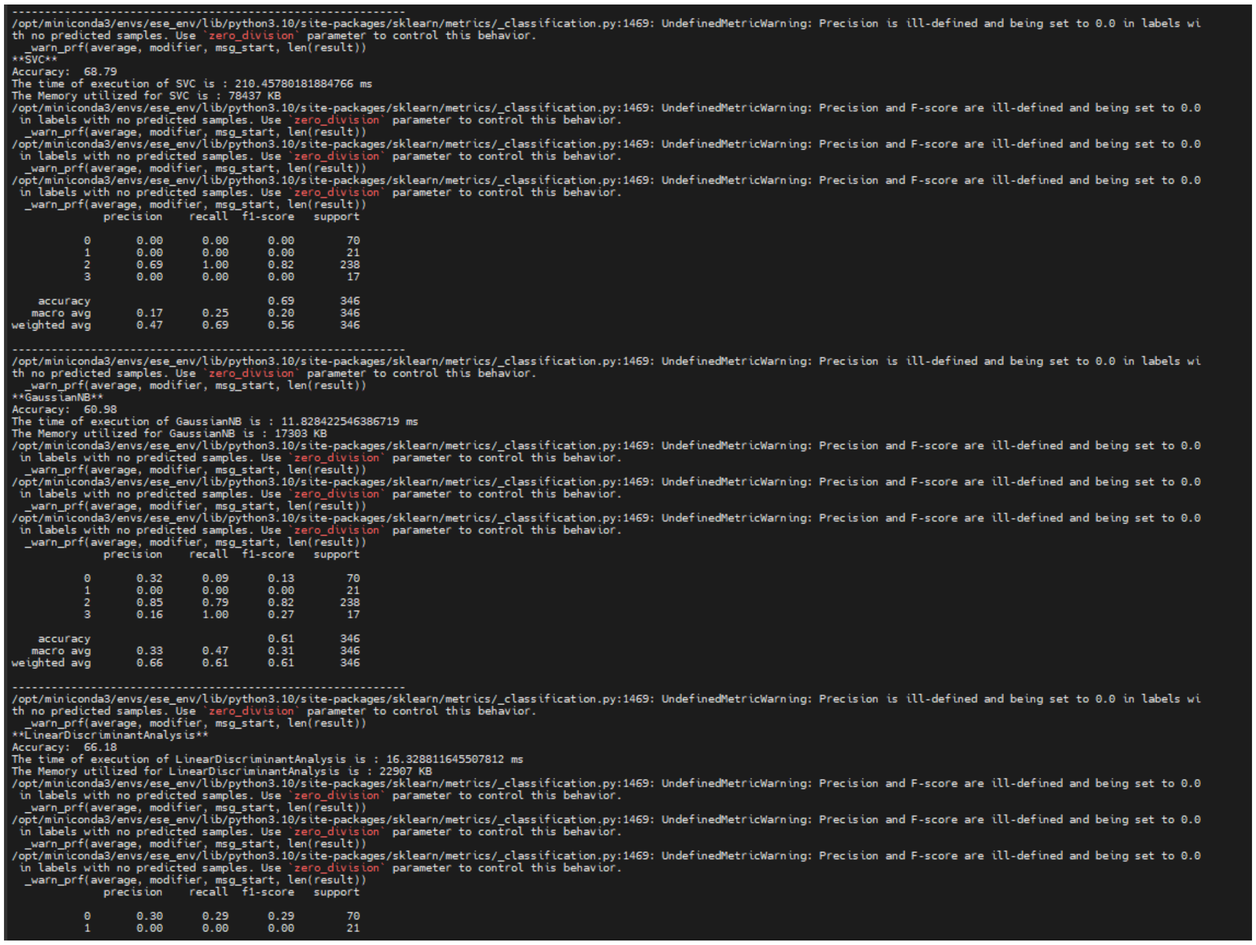
Fig. 9

Figures 10, 11, and 12 below show our version of the cars dataset being run by multiple algorithms sourced from libraries. It is evident in the figures that there are a large number of errors from the algorithms when our dataset is used. These errors are divide-by-zero errors which is the same error we ran into when running our own algorithm on Google Colab. There were a few classifier algorithms that did work. These were gradient boosting, random forest, k-neighbors classifier and of course, decision trees. Even though these algorithms did successfully execute the code, some of the values ended up being zeroed out anyway, which makes the resulting accuracy values meaningless, aside from the fact that they were already meaningless because 6 of the 10 algorithms failed to execute. One thing to note is that the decision tree algorithm from the library on the GPU was a lot faster than the decision tree algorithm that we wrote. When our algorithm ran in Google Colab for the Cars dataset, it took 12 seconds, whereas in the GPU, with the algorithm from the library, it took only 35.77ms. Even after all the errors from our dataset, it was still evident that our algorithm is less efficient than the library function by a few orders of magnitude.



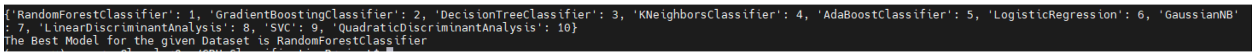
```
(ese_env) ageorge@lscalar0:~/GPU_ClassificationProject$ python Run.py
libGL error: MESA-LOADER: failed to open swrast: /usr/lib/dri/swrast_dri.so: cannot open shared object file: No such file or directory (search paths /usr/lib/x86_64-linux-gnu/dri:\$
${ORIGIN}/dri:/usr/lib/dri, suffix _dri)
libGL error: failed to load driver: swrast
------------------------------------------------------------
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels wi
th no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
**LogisticRegression**
Accuracy:  66.47
The time of execution of LogisticRegression is : 134.4473361968994 ms
The Memory utilized for LogisticRegression is : 529966 KB
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
              precision    recall  f1-score   support

           0       0.28      0.21      0.24        70
           1       0.00      0.00      0.00        21
           2       0.74      0.89      0.80       238
           3       0.67      0.24      0.35        17

    accuracy                           0.66       346
   macro avg       0.42      0.33      0.35       346
weighted avg       0.60      0.66      0.62       346

------------------------------------------------------------
**DecisionTreeClassifier**
Accuracy:  95.95
The time of execution of DecisionTreeClassifier is : 35.77470779418945 ms
The Memory utilized for DecisionTreeClassifier is : 18611 KB
              precision    recall  f1-score   support

           0       0.88      0.94      0.91        70
           1       1.00      0.81      0.89        21
           2       0.99      0.99      0.99       238
           3       0.87      0.76      0.81        17

    accuracy                           0.96       346
   macro avg       0.93      0.88      0.90       346
weighted avg       0.96      0.96      0.96       346

------------------------------------------------------------
**KNeighborsClassifier**
Accuracy:  85.55
The time of execution of KNeighborsClassifier is : 65.88912010192871 ms
The Memory utilized for KNeighborsClassifier is : 201343 KB
              precision    recall  f1-score   support

           0       0.64      0.71      0.68        70
           1       0.83      0.48      0.61        21
           2       0.92      0.95      0.94       238
           3       1.00      0.53      0.69        17

    accuracy                           0.86       346
   macro avg       0.85      0.67      0.73       346
weighted avg       0.86      0.86      0.85       346

------------------------------------------------------------
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels wi
th no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
**SVC**
Accuracy:  68.79
The time of execution of SVC is : 210.45780181884766 ms
The Memory utilized for SVC is : 78437 KB
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Fig. 10



```
-------------------------------------------------------------
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels wi
th no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
**SVC**
Accuracy:  68.79
The time of execution of SVC is : 210.45780181884766 ms
The Memory utilized for SVC is : 78437 KB
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        70
           1       0.00      0.00      0.00        21
           2       0.69      1.00      0.82       238
           3       0.00      0.00      0.00        17

    accuracy                           0.69       346
   macro avg       0.17      0.25      0.20       346
weighted avg       0.47      0.69      0.56       346

-------------------------------------------------------------
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels wi
th no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
**GaussianNB**
Accuracy:  60.98
The time of execution of GaussianNB is : 11.828422546386719 ms
The Memory utilized for GaussianNB is : 17303 KB
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
              precision    recall  f1-score   support

           0       0.32      0.09      0.13        70
           1       0.00      0.00      0.00        21
           2       0.85      0.79      0.82       238
           3       0.16      1.00      0.27        17

    accuracy                           0.61       346
   macro avg       0.33      0.47      0.31       346
weighted avg       0.66      0.61      0.61       346

-------------------------------------------------------------
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels wi
th no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
**LinearDiscriminantAnalysis**
Accuracy:  66.18
The time of execution of LinearDiscriminantAnalysis is : 16.328811645507812 ms
The Memory utilized for LinearDiscriminantAnalysis is : 22907 KB
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1469: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0
 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
              precision    recall  f1-score   support

           0       0.30      0.29      0.29        70
           1       0.00      0.00      0.00        21
```

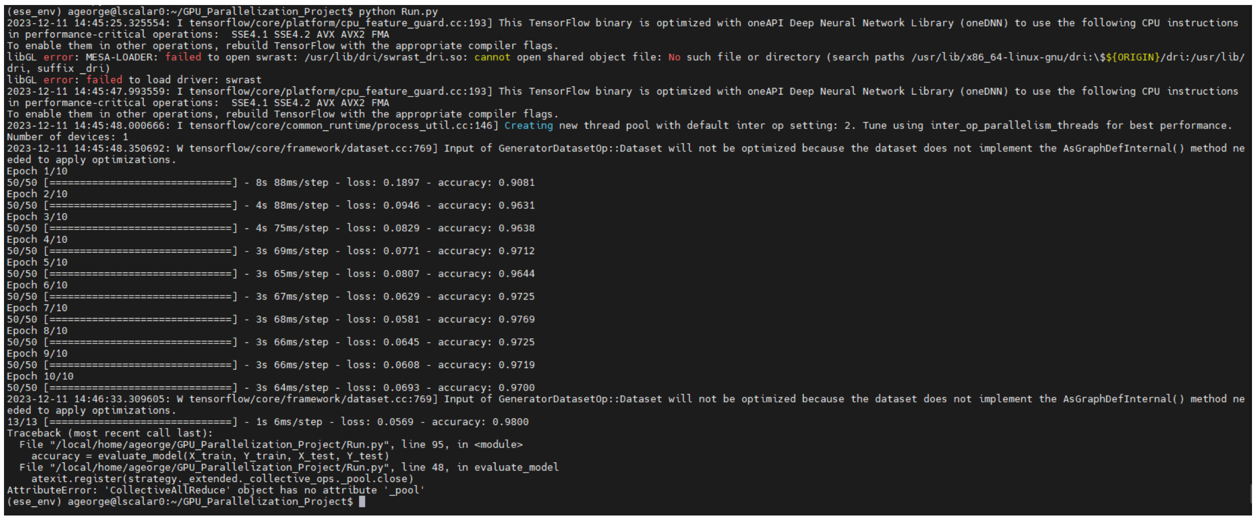
Fig. 11



```
{'RandomForestClassifier': 1, 'GradientBoostingClassifier': 2, 'DecisionTreeClassifier': 3, 'KNeighborsClassifier': 4, 'AdaBoostClassifier': 5, 'LogisticRegression': 6, 'GaussianNB'
: 7, 'LinearDiscriminantAnalysis': 8, 'SVC': 9, 'QuadraticDiscriminantAnalysis': 10}
The Best Model for the given Dataset is RandomForestClassifier
```

Fig. 12

Below represents the result from the parallelization python script using the same original dataset as specified during the classification project ("original.csv"). With this dataset, the results show that going from 0 epochs to 10 epochs (or iterations) the accuracy of classification of the data rises from 0.9081 to 0.9700, which makes sense since given more iterations through the neural network, the accuracy of classification should increase as well. Also, due to how the neural network runs in parallel with multiple GPUs in the GPU server, the overall execution time for each epoch is also surprisingly efficient, given the dataset as a whole includes 2000 data points.



```
(ese_env) ageorge@lscalar0:~/GPU_Parallelization_Project$ python Run.py
2023-12-11 14:45:25.325554: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions
in performance-critical operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
libGL error: MESA-LOADER: failed to open swrast: /usr/lib/dri/swrast_dri.so: cannot open shared object file: No such file or directory (search paths /usr/lib/x86_64-linux-gnu/dri:\$${ORIGIN}/dri:/usr/lib/
dri, suffix _dri)
libGL error: failed to load driver: swrast
2023-12-11 14:45:47.993559: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions
in performance-critical operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-12-11 14:45:48.000666: I tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool with default inter op setting: 2. Tune using inter_op_parallelism_threads for best performance.
Number of devices: 1
2023-12-11 14:45:48.350692: W tensorflow/core/framework/dataset.cc:769] Input of GeneratorDatasetOp::Dataset will not be optimized because the dataset does not implement the AsGraphDefInternal() method ne
eded to apply optimizations.
Epoch 1/10
50/50 [==============================] - 8s 88ms/step - loss: 0.1897 - accuracy: 0.9081
Epoch 2/10
50/50 [==============================] - 4s 88ms/step - loss: 0.0946 - accuracy: 0.9631
Epoch 3/10
50/50 [==============================] - 4s 75ms/step - loss: 0.0829 - accuracy: 0.9638
Epoch 4/10
50/50 [==============================] - 3s 69ms/step - loss: 0.0771 - accuracy: 0.9712
Epoch 5/10
50/50 [==============================] - 3s 65ms/step - loss: 0.0807 - accuracy: 0.9644
Epoch 6/10
50/50 [==============================] - 3s 67ms/step - loss: 0.0629 - accuracy: 0.9725
Epoch 7/10
50/50 [==============================] - 3s 68ms/step - loss: 0.0581 - accuracy: 0.9769
Epoch 8/10
50/50 [==============================] - 3s 66ms/step - loss: 0.0645 - accuracy: 0.9725
Epoch 9/10
50/50 [==============================] - 3s 66ms/step - loss: 0.0608 - accuracy: 0.9719
Epoch 10/10
50/50 [==============================] - 3s 64ms/step - loss: 0.0693 - accuracy: 0.9700
2023-12-11 14:46:33.309605: W tensorflow/core/framework/dataset.cc:769] Input of GeneratorDatasetOp::Dataset will not be optimized because the dataset does not implement the AsGraphDefInternal() method ne
eded to apply optimizations.
13/13 [==============================] - 1s 6ms/step - loss: 0.0569 - accuracy: 0.9800
Traceback (most recent call last):
  File "/local/home/ageorge/GPU_Parallelization_Project/Run.py", line 95, in <module>
    accuracy = evaluate_model(X_train, Y_train, X_test, Y_test)
  File "/local/home/ageorge/GPU_Parallelization_Project/Run.py", line 48, in evaluate_model
    atexit.register(strategy._extended._collective_ops._pool.close)
AttributeError: 'CollectiveAllReduce' object has no attribute '_pool'
(ese_env) ageorge@lscalar0:~/GPU_Parallelization_Project$ 
```

Fig. 13

After a long process, we unfortunately were not able to create a very efficient algorithm. There are a multitude of bugs, and it is not efficient enough to handle very large datasets. In the extremely small range of datasets where the algorithm can classify, it does do so and in some cases was quite accurate. In our eyes, being able to write a decision tree algorithm from scratch that can classify anything is a success. It would be interesting to learn more about why the algorithm did not work for those cases, and why it worked for the other cases. We feel that we learned a lot about decision trees regardless if our result was not completely correct.

Appendix: Source Code for Decision Tree Algorithm

```
import pandas as pd  # pandas library reads csv file
https://pandas.pydata.org/docs/user_guide/io.html
import itertools  # for finding all combinations in a list (GINI INDEX)
import math


class Node:
    def __init__(self, attribute, parent_label):
        self.attribute = attribute  # splitting criterion
        self.children = []  # dictionary of children nodes
        self.parent_label = parent_label

    def addChild(self, child):  # dictionary is setup in {key: value}
format
        self.children.append(child)  # add child to node.children

    def addFeature(self, feature):
        self.parent_label = feature

    def display(self, depth=0):  # prints fpTree in vertical format(if
nodes line up vertically they have the same parent node)
        prefix = '  ' * depth
        print(f"{prefix}{self.attribute}{prefix}{self.parent_label}")
        for child in self.children:
            child.display(depth + 1)


def numOfClasses(Dataset, last_col):
    countList = []
    nameList = []
    for row in Dataset:
        row_index = Dataset.index(row)
        if Dataset[row_index][last_col] not in nameList:
            nameList.append(Dataset[row_index][last_col])
            countList.append(1)
        else:
            countList[nameList.index(Dataset[row_index][last_col])] += 1
    return countList, nameList

def majorityClasses(Dataset, last_col):
    temp_count, temp_name = numOfClasses(Dataset, last_col)
    maxCount = max(temp_count)
    return temp_name[temp_count.index(maxCount)]


def GenerateDecisionTree(Dataset, attribute_list, attribute_list_ref,
selection_string, last_col):
    # create new node
    root = Node(None, None)
    # if (tuples in Dataset are of the same class)
    # return node as a leaf labeled as class c
    c = Dataset[0][last_col]  # literal class
    classList = []
    for row in Dataset:
        classList.append(row[last_col])  # class list

    if (all(value == c for value in classList)):
        root = Node(c, None)
        return root
    # if (attribute_list == {})
    if (len(attribute_list) == 0):
        # return node as a leaf labeled with the majority class in dataset
        root = Node(majorityClasses(Dataset, last_col), None)
        return root

    # attribute = Attribute_Selection_Method(Attribute_List, String)
    attribute, splitting_feature = Attribute_Selection_Method(Dataset,
attribute_list, attribute_list_ref, selection_string, last_col)
    # label node with splitting criterion
    root = Node(attribute, None)
    # if (splitting criterion is categorical/discrete and math related,
splitsa are allowed)
    # return attribute_list = attribute_list - {attribute}
    index = attribute_list_ref.index(attribute)
    attribute_list.remove(attribute)
    # for (each outcome j of the splitting criterion) {
```

```
    # Dj = set of tuples in D satisfying outcome j
    # if (Dj == NULL)
    # attach a leaf labeled with the majority class of  D
    # else attach the node returned by GenerateDecisionTree()
    # }

    # creating feature list
    if selection_string == "GINI":
        D_1 = []                            # meets condition
(yes)
        D_2 = []                            # doesn't meet condition
(no)

        for row in Dataset:
            row_index = Dataset.index(row)
            if Dataset[row_index][index] in splitting_feature:
                D_1.append(row)
            else:
                D_2.append(row)

       # check if all classes in D_1 are the same
        # make that a leaf node of that class
        c_D_1 = D_1[0][last_col]
        classList_D_1 = []
        for row in D_1:
            classList_D_1.append(row[last_col])  # class list

        if (all(value == c_D_1 for value in classList_D_1)):
            child_1 = Node(c_D_1, splitting_feature[0])
            root.addChild(child_1)
        else:
            child_1 = GenerateDecisionTree(D_1, attribute_list,
attribute_list_ref, selection_string, last_col)
            child_1.addFeature(splitting_feature[0])  # probably wrong!
            root.addChild(child_1)

       # check if all classes in D_2 are the same
        # make that a leaf node of that class
        c_D_2 = D_2[0][last_col]
        classList_D_2 = []
        for row in D_2:
            classList_D_2.append(row[last_col])  # class list

        if (all(value == c_D_2 for value in classList_D_2)):
            child_2 = Node(c_D_2, splitting_feature[0])
            root.addChild(child_2)
        else:
            child_2 = GenerateDecisionTree(D_2, attribute_list,
attribute_list_ref, selection_string, last_col)
            child_2.addFeature(splitting_feature[0])
            root.addChild(child_2)

        return root

    else:
        feature_list = []
        for row in Dataset:
            row_index = Dataset.index(row)
            if Dataset[row_index][index] not in feature_list:
                feature_list.append(Dataset[row_index][index])

        for feature in feature_list:
            D_j = []
            for row in Dataset:
                row_index = Dataset.index(row)
                if Dataset[row_index][index] == feature:
                    D_j.append(Dataset[row_index])  # creating D_j

            if len(D_j) == 0:
                child = Node(majorityClasses(Dataset, last_col), feature)
                root.addChild(child)
            else:
                child = GenerateDecisionTree(D_j, attribute_list,
attribute_list_ref, selection_string, last_col)
                child.addFeature(feature)
                root.addChild(child)

        return root
```

```
def Attribute_Selection_Method(Dataset, attribute_list,
attribute_list_ref, selection_string, last_col):
    selection = selection_string
    if selection == "GINI":
        temp_count, temp_name = numOfClasses(Dataset, last_col)
        sum = 0
        for class_name in temp_name:
            d_i = []
            for row in Dataset:
                if row[last_col] == class_name:
                    d_i.append([row])
            p_i = (len(d_i) / len(Dataset)) ** 2
            sum += p_i

        gini_d = 1 - sum

        gini_diff_val= []
        gini_diff_list = []

        gini_max_attr_diff = []

        unique_comb_list = []

        for a in attribute_list:
            singular_fea = []  # singular features for an attribute
            unique_comb = []  # unique combinations of features from an
attribute

            for row in Dataset:  # singular_fee list creation
                row_index = Dataset.index(row)
                index = attribute_list_ref.index(a)
                if Dataset[row_index][index] not in singular_fea:  #
column-row indexing
                    singular_fea.append(Dataset[row_index][index])

            num_of_combinations = 2 ** len(singular_fea) - 2
            if len(singular_fea) == 2:

unique_comb.append(list(itertools.combinations(singular_fea, 1)))
            else:
                for i in range(1, (len(singular_fea))):  # unique_comb
list creation (1 to n-1)

unique_comb.append(list(itertools.combinations(singular_fea, i)))

            unique_comb_list = []
            for j in unique_comb:
                for k in j:
                    unique_comb_list.append(list(k))  # unique_comb is
list of lists

            for i in range(len(unique_comb_list)):
                yes_count = 0
                no_count = 0
                yes_list = []
                no_list = []

                for row in Dataset:
                    row_index = Dataset.index(row)
                    if Dataset[row_index][index] in unique_comb_list[i]:
                        yes_list.append(Dataset[row_index])  # D1
                        yes_count += 1
                    else:
                        no_list.append(Dataset[row_index])  # D2
                        no_count += 1

                # Gini(D1)
                temp_count_yes = []
                temp_name_yes = []
                temp_count_yes, temp_name_yes = numOfClasses(yes_list,
last_col)
                sum_1 = 0
                for class_name in temp_name_yes:
                    D1 = []
                    for row in yes_list:
                        row_index = yes_list.index(row)
                        if row[last_col] == class_name:
                            D1.append([row])
                    p_i_1 = (len(D1) / len(yes_list)) ** 2
                    sum_1 += p_i_1
                gini_D1 = 1 - sum_1

                # Gini(D2)
                temp_count_no = []
                temp_name_no = []
                temp_count_no, temp_name_no = numOfClasses(no_list,
last_col)
                sum_2 = 0
                for class_name in temp_name_no:
                    D2 = []
                    for row in no_list:
                        row_index = no_list.index(row)
                        if row[last_col] == class_name:
                            D2.append([row])
                    p_i_2 = (len(D2) / len(no_list)) ** 2
                    sum_2 += p_i_2

                gini_D2 = 1 - sum_2

                # Gini(a)
                gini_a = ((yes_count / len(Dataset)) * gini_D1) +
((no_count / len(Dataset)) * gini_D2)

                gini_diff_list.append([gini_d - gini_a,
unique_comb_list[i]])
                gini_diff_val.append(gini_d - gini_a)

            max_gini_diff = max(gini_diff_val)

gini_max_attr_diff.append(gini_diff_list[gini_diff_val.index(max_gini_diff
)])

            gini_diff_list = []
            gini_diff_val = []

            index += 1

        max_attr_diff = max(gini_max_attr_diff)
        return attribute_list[gini_max_attr_diff.index(max_attr_diff)],
max_attr_diff[1]

    elif selection == "INFORMATION GAIN":
        temp_count, temp_name = numOfClasses(Dataset, last_col)
        sum = 0
        for class_name in temp_name:
            d_i = []
            for row in Dataset:
                row_index = Dataset.index(row)
                if row[last_col] == class_name:
                    d_i.append([row])
            p_i = (len(d_i) / len(Dataset))
            addend = p_i * math.log(p_i, 2)
            sum += addend

        info_d = sum * -1

        gain_diff_list = []

        gain_max_attr_diff = []

        for a in attribute_list:
            singular_fea = []  # singular features for an attribute
            info_a_i = 0

            for row in Dataset:
                row_index = Dataset.index(row)  # singular_fee list
creation
                index = attribute_list_ref.index(a)
                if Dataset[row_index][index] not in singular_fea:  #
column-row indexing
                    singular_fea.append(Dataset[row_index][index])

            for feature in singular_fea:
                d_i_j_count = 0
                d_i_j = []

                for row in Dataset:
                    row_index = Dataset.index(row)
```

```python
                    if Dataset[row_index][index] == feature:  # d_i_j
                        d_i_j_count += 1
                        d_i_j.append(row)

                # Info(D_ij)
                temp_count_j, temp_name_j = numOfClasses(d_i_j, last_col)
                sum_j = 0
                for class_name in temp_name_j:
                    d_i_j_yes = []
                    for row in d_i_j:
                        row_index = d_i_j.index(row)
                        if row[last_col] == class_name:
                            d_i_j_yes.append(row)
                    p_i_j = (len(d_i_j_yes) / len(d_i_j))
                    addend_j = p_i_j * math.log(p_i_j, 2)
                    sum_j += addend_j

                info_d_i_j = sum_j * -1

                info_a_i += ((d_i_j_count / len(Dataset)) * info_d_i_j)

            gain_a_i = info_d - info_a_i
            gain_diff_list.append(gain_a_i)

        max_gain_attr_diff = max(gain_diff_list)

        return attribute_list[gain_diff_list.index(max_gain_attr_diff)],
"No Splitting Feature (IG)"

    else:  # GAIN RATIO
        temp_count, temp_name = numOfClasses(Dataset, last_col)
        sum = 0
        for class_name in temp_name:
            d_i = []
            for row in Dataset:
                row_index = Dataset.index(row)
                if row[last_col] == class_name:
                    d_i.append([row])
            p_i = (len(d_i) / len(Dataset))
            addend = p_i * math.log(p_i, 2)
            sum += addend

        info_d = sum * -1

        gain_ratio_list = []

        for a in attribute_list:
            singular_fea = []  # singular features for an attribute
            info_a_i = 0

            for row in Dataset:
                row_index = Dataset.index(row)  # singular_fee list
creation
                index = attribute_list_ref.index(a)
                if Dataset[row_index][index] not in singular_fea:  #
column-row indexing
                    singular_fea.append(Dataset[row_index][index])

            split_info = 0

            for feature in singular_fea:
                d_i_j_count = 0
                d_i_j = []

                feature_count = 0
                for row in Dataset:
                    row_index = Dataset.index(row)
                    if Dataset[row_index][index] == feature:  # d_i_j
                        d_i_j_count += 1
                        d_i_j.append(row)
                        feature_count += 1

                # Info(D_ij)
                temp_count_j, temp_name_j = numOfClasses(d_i_j, last_col)
                sum_j = 0
                for class_name in temp_name_j:
                    d_i_j_yes = []
                    for row in d_i_j:
                        row_index = d_i_j.index(row)
                        if row[last_col] == class_name:
                            d_i_j_yes.append(row)
                    p_i_j = (len(d_i_j_yes) / len(d_i_j))
                    addend_j = p_i_j * math.log(p_i_j, 2)
                    sum_j += addend_j

                info_d_i_j = sum_j * -1

                info_a_i += ((d_i_j_count / len(Dataset)) * info_d_i_j)

                feature_addend = ((feature_count / len(Dataset)) *
math.log((feature_count / len(Dataset)), 2)) * -1

                split_info += feature_addend

            gain_a_i = info_d - info_a_i

            gain_ratio_a = (gain_a_i / split_info)
            gain_ratio_list.append(gain_ratio_a)

        max_gain_ratio = max(gain_ratio_list)

        return attribute_list[gain_ratio_list.index(max_gain_ratio)], "No
Splitting Feature (GR)"

def test_tree_g(dataset, attribute_list_ref, root, last_col):
  col_index = attribute_list_ref.index(root.attribute)
  accuracy = 0
  print(attribute_list_ref)
  for row in dataset:
    ptr = root
    row_index = dataset.index(row)
    while(ptr.children != []):
        if ptr.children[0].parent_label == dataset[row_index][col_index]:
            if ptr.children[0].attribute not in attribute_list_ref:
                ptr = ptr.children[0]
                break
            else:
                col_index =
attribute_list_ref.index(ptr.children[0].attribute)
            ptr = ptr.children[0]
        else:
            if ptr.children[1].attribute not in attribute_list_ref:
                ptr = ptr.children[1]
                break
            else:
                col_index =
attribute_list_ref.index(ptr.children[1].attribute)
            ptr = ptr.children[1]

    if dataset[row_index][last_col] == ptr.attribute:
        accuracy += 1

  accuracy = accuracy / len(dataset)
  return accuracy

def test_tree(dataset, attribute_list_ref, root, last_col):
  accuracy = 0
  print(attribute_list_ref)
  for row in dataset:
    col_index = attribute_list_ref.index(root.attribute)
    ptr = root
    row_index = dataset.index(row)
    while_flag = True
    i = 0
    while(1):
        for i in range(len(ptr.children)):
            if ptr.children[i].parent_label ==
dataset[row_index][col_index]:           # you have to go to child node
                if ptr.children[i].attribute not in attribute_list_ref:
# leaf node
                    while_flag = False
                    # flag to break out of while loop
                    break
                else:
# another child nodeS
                    col_index =
attribute_list_ref.index(ptr.children[i].attribute)
                break
                # flag to break out of for loop
        if ptr.children == []:
            if while_flag == False:
                break

        ptr = ptr.children[i]

    if dataset[row_index][last_col] == ptr.attribute:
        accuracy += 1

  accuracy = accuracy / len(dataset)
  return accuracy

def main():
    Dataframe = pd.read_csv("/content/lymphography_fixed.csv") # read data
into pandas dataframe from csv file
    Dataframe.astype(str)
    # Dataframe2 = Dataframe[Dataframe.columns[1:]] #remove first column
of labels
    Dataset = Dataframe.values.tolist() #convert to list of lists
    d1 = Dataframe.sample(frac = 0.7) #tran
    d2 = Dataframe.drop(d1.index) #test
    dataset1 = d1.values.tolist()
    dataset2 = d2.values.tolist()

    last_col = 18  # last col for classes (dependent on us)
    index = 0  # index for each attribute

    attribute_list =    ["lymphatics", "block_of_affere",
"bl._of_lymph._c", "bl._of_lymph._s",
                         "by_pass", "extravasates", "regeneration_of",
"early_uptake_in",
                         "lymphatics", "lym.nodes_dimin",
"lym.nodes_enlar", "changes_in_lym.",
                         "defect_in_node", "changes_in_node",
"changes_in_stru", "special_forms",
                         "dislocation_of", "exclusion_of_no",
"no._of_nodes_in"]

    attribute_list_ref = ["lymphatics", "block_of_affere",
"bl._of_lymph._c", "bl._of_lymph._s",
                         "by_pass", "extravasates", "regeneration_of",
"early_uptake_in",
                         "lymphatics", "lym.nodes_dimin",
"lym.nodes_enlar", "changes_in_lym.",
                         "defect_in_node", "changes_in_node",
"changes_in_stru", "special_forms",
                         "dislocation_of", "exclusion_of_no",
"no._of_nodes_in"]
    #attribute_list = ["age", "income", "student", "credit_rating"]
    #attribute_list_ref = ["age", "income", "student", "credit_rating"]
    #Dataset = [["<=30", "high", "no", "fair", "no"],
   #            ["<=30", "high", "no", "excellent", "no"],
    #           ["31...40", "high", "no", "fair", "yes"],
     #          [">40", "medium", "no", "fair", "yes"],
      #         [">40", "low", "yes", "fair", "yes"],
       #        [">40", "low", "yes", "excellent", "no"],
       #        ["31...40", "low", "yes", "excellent", "yes"],
        #       ["<=30", "medium", "no", "fair", "no"],
         #      ["<=30", "low", "yes", "fair", "yes"],
          #     [">40", "medium", "yes", "fair", "yes"],
           #    ["<=30", "medium", "yes", "excellent", "yes"],
            #   ["31...40", "medium", "no", "excellent", "yes"],
         #      ["31...40", "high", "yes", "fair", "yes"],
          #     [">40", "medium", "no", "excellent", "no"]]

    root1 = GenerateDecisionTree(dataset1, attribute_list,
attribute_list_ref, "GINI", last_col)
    print("\nAttribute or Leaf     Meets Condition? -> Condition")
    root1.display()
    print("\n")

    attribute_list = ["lymphatics", "block_of_affere", "bl._of_lymph._c",
"bl._of_lymph._s",
                         "by_pass", "extravasates", "regeneration_of",
"early_uptake_in",
                         "lymphatics", "lym.nodes_dimin",
"lym.nodes_enlar", "changes_in_lym.",
                         "defect_in_node", "changes_in_node",
"changes_in_stru", "special_forms",
                            "dislocation_of", "exclusion_of_no",
"no._of_nodes_in"]
    root2 = GenerateDecisionTree(dataset1, attribute_list,
attribute_list_ref, "INFORMATION GAIN", last_col)
    print("\nAttribute or Leaf     Feature")
    root2.display()
    print("\n")

    attribute_list = ["lymphatics", "block_of_affere", "bl._of_lymph._c",
"bl._of_lymph._s",
                            "by_pass", "extravasates", "regeneration_of",
"early_uptake_in",
                            "lymphatics", "lym.nodes_dimin",
"lym.nodes_enlar", "changes_in_lym.",
                            "defect_in_node", "changes_in_node",
"changes_in_stru", "special_forms",
                            "dislocation_of", "exclusion_of_no",
"no._of_nodes_in"]
    root3 = GenerateDecisionTree(dataset1, attribute_list,
attribute_list_ref, "GAIN RATIO", last_col)
    print("\nAttribute or Leaf     Feature")
    root3.display()
    print("\n")

    accuracy1 = test_tree_g(dataset2, attribute_list_ref, root1, last_col)
    print(accuracy1)

    accuracy2 = test_tree(dataset2, attribute_list_ref, root2, last_col)
    print(accuracy2)

    accuracy3 = test_tree(dataset2, attribute_list_ref, root3, last_col)
    print(accuracy3)

if __name__ == "__main__":
    main()
```