Jake Lin and Tyler Shaw
ESE 327: Fundamental Algorithms for Machine Learning Systems
Professor Doboli
Project 2 Report: Implementing the Decision Tree Induction Algorithm for Classification

Implementation:

Reading in the values

First, we read the input and make a 2d vector of strings, where each element of the vector corresponds to a different tuple and each tuple is a data entry made up of strings.

Accounting for continuous data

Then, we parse the 2d vector and for the first row, we check each column to see if the entry is a string or number. We convert all of the numerical values to doubles and find the minimum and maximum values by sorting the values. We find the midpoint of the minimum and maximum. The midpoint serves as the splitting point for that column. We go back to the original dataset and go back to the column. We then classify if it is less than or equal to the splitting point or greater than the splitting point for each item in the column.

Creating attribute_list

Next, we generate the possible outcomes for each attribute. We used the size of the first row of the dataset to correspond to how many columns are in the original dataset and also how many attributes there are in the dataset.

Ensure attribute_list =attribute_list- splitting_attribute is accounted for in each recursive call

Next, we generate the decision tree by passing in the original dataset, the attribute list passed by value, and the attribute list passed by reference. We passed the attribute list by reference and by value so that one could always stay the same, but one could be modified to reflect the attributes being removed. First, we create a pointer of type tree node. We ensure that attributes that are eliminated are removed from the passed by reference attribute list. We compare the two attribute lists and if the sizes are the same, we remove the attribute just used. We then eliminate the columns of the dataset that correspond to the removed attributes.

Sorting the data for easier use in attribute_selection_method

We create a 2d vector of lists called sorted tuples. We pass the dataset into a method that calculates the frequency of each combination of attribute outcomes and its respective class for a particular attribute.

Finding majority class
Next, we find the majority class by checking the last column of each data entry, which contains the class, and finding the maximum frequency of each class. We create another tree node labeled with the majority class and check if every item is of the same class. If it is, we return the root tree node labeled with the majority class originally constructed at the beginning of the method. We then check if the attribute list is empty and if it is, we also return root labeled with the majority class.

Call attribute_selection_method
We only pass sorted tuples into the attribute selection method. First, we find the total number of rows in the dataset. We implement the information gain method. The splitting attribute method returns an integer that represents which column of the dataset should be used as the splitting attribute. We also developed methods for gain ratio and gini index. Those can be used by changing the integer value attribute_selection_method to 2 or 3 respectively.

attribute_list= attribute_list - splitting_attribute
If we are not using gini index, we go to that attribute list and delete that element from attribute list.

Create partitions and pass them
We create a three dimensional vector of strings. We find the different combinations of the selected attribute and split the dataset up into smaller datasets each with a common outcome of the splitting attribute.
We check each of the smaller datasets to see if there is only one column, in which case, there is only the class left and we terminate the execution of the method and return the class of that column. One by one, we pass each smaller dataset recursively into the method again.

Pseudocode:
Generate decision tree(dataset, attributelist, & attributelist(pass by reference), attribute_selection_method){
//1 for attribute_selection_method is information gain, 2 is gain ratio, 3 is gini index

Create a new node called root of type tree node
Struct Tree node- contains vector or linkedlist of sub_attributes and attribute_types/outcomes
Compare attributelist with attributelist(pass by reference)
if(attributelist does contain a attribute not in attributelist(pass by ref) then update dataset until every column is the same as the attribute (pass by ref)

```
Call sort_tuples(data)
    Create a 3d structure called sorted_data
    for(each column in data){
    If the item and its class exists in sorted_data, then add to count in sorted_data.at(i).at(j).at(k)
    Else add to sorted_data.at(i).at(j)

Call find_majority(data)
    find_majority-> for(data at (last column)) -> each class type
    if(data.at(last column) is already considered, then increment count
    Find the class with the highest count

If (majority class is== to data size)
Return root= majority class

If (attribute list size is 0)
Return root= majority class

If attribute_selection_method is 1
    Call attribute_selection_method(sorted data)
       Attribute_selection_method:
       for(each sorted_data.at(i) ){- each attribute
       Calculate the information gain
       Find max of information gain and return as int

If attribute_selection_method is 2
    Call attribute_selection_method_gainratio(sorted data)
       Attribute_selection_method_gainratio:
       for(each sorted_data.at(i) ){- each attribute
       Calculate the gain ratio
       Find max of gain ratio and return as int

If attribute_selection_method is 3
    Call attribute_selection_method_gini(sorted data)
       Attribute_selection_method:
       for(each sorted_data.at(i) ){- each attribute
       Generate all possible combinations of outcomes of each attribute
       for(each combination of outcomes)
            if(combination was not already used)
              Calculate the gini index
              Find min of gini index and return as int
```

```
If (attribute selection method is not 3 (gini index))
   Using int position of splitting attribute, erase it from attribute list

Create a 3d data structure
For(each row in data ){
    Find the column of splitting attribute
    Push entire row excluding column of splitting attribute to new 3d data structure
    if(new 3d structure at(i).at(0) size ==1)//meaning only one column, the class col left
       root->sub_attributes.pushback a node with that class

Else
   root->sub_attributes.push_back(generate_decision_tree(data, attributelist, attributelist,
attribute_selection_method))
}
```

Results:

We tested our algorithm by splitting the data into two groups. One group was used to create the decision tree, while the other group was classified using the decision tree. The group sizes are easily modifiable. Then, we compared the decision tree results with the actual class of the data entry. We calculated the accuracy of our classifier as the number of correct entries classified over the number of total entries classified. Unfortunately, we were not able to fully implement the gini index algorithm. We did write code for the gini index, but the code got stuck in an infinite loop we were unable to resolve. Therefore, our results provided below will only be for information gain and gain ratio. We were unable to test our own algorithm on the GPU server, but we were able to test the benchmarks on the server.

We tested our algorithm on the following datasets:
Dataset 1: congressional voting records, 435 entries, 16 attributes, class at beginning
Dataset 2: breast cancer, 286 entries, 9 attributes, class at beginning
Dataset 3: mushrooms, 8124 entries, 22 attributes, class at beginning
Dataset 4: soybean small, 47 entries, 35 attributes, class at end
Dataset 5: adult stretch, 20 entries, 4 attributes, class at end
Dataset 6: anneal, 798 entries, 38 attributes, class at end
Dataset 7: balance scale, 625 entries, 4 attributes, class at end
Dataset 8: cmc, 1473 entries, 9 attributes, class at end
Dataset 9: chess, 3200 entires, 36 attributes, class at end
Dataset 10: adult, 32561 entries, 14 attributes, class at end

For each dataset, we created a decision tree with 25%, 50%, 75%, and 100% of the dataset and classified the rest of the data. For 100%, we classified all of the data. We used information gain attribute selection for each of these measurements. For 100%, we also measured the accuracy with gain ratio. We found that information gain and gain ratio produced similar results, so we only took one measurement of gain ratio, which is for a decision tree with 25% of the data. We include our accuracy results in the table below. We have also attached images of the binary trees produced by using all of the data for both information gain and gain ratio.

Table of Accuracy Results:

Dataset Number	25% of data accuracy	50% of data accuracy	75% of data accuracy	100% of data accuracy (info gain)	100% of data accuracy (gain ratio)
1	96.06%	93.18%	95.45%	97.24%	97.24%
2	6.51%	62.24%	73.61%	77.62%	77.97%
3	69.28%	93.70%	100%	100%	100%
4	16.67%	16.67%	16.67%	100%	100%
5	100%	100%	100%	100%	100%
6	91.18%	93.25%	76.5%	78.07%	96.74%
7	31.56%	31%	27.33%	32.16%	32.16%
8	40.09%	25.51%	18.20%	43.31%	43.42%
9	30.46%	95.25%	95.88%	95.40%	95.18%
10	76.08%	81.57%	80.96%	75.96%	76.42%

Dataset 1: congressional voting records
Decision Tree (information gain):

```
attribute3:
attribute3 type 0: y ->  attribute10
attribute3 type 1: ? ->  democrat
attribute3 type 2: n ->  democrat

attribute10:
attribute10 type 0: ? ->  republican
attribute10 type 1: n ->  attribute14
attribute10 type 2: y ->  attribute8

attribute14:
attribute14 type 0: n ->  attribute2
attribute14 type 1: y ->  attribute9
attribute14 type 2: ? ->  republican

attribute2:
attribute2 type 0: n ->  republican
attribute2 type 1: y ->  attribute15

attribute15:
attribute15 type 0: y ->  republican
attribute15 type 1: ? ->  attribute0

attribute0:
attribute0 type 0: y ->  republican
attribute0 type 1: n ->  democrat
```

*****The tree is structured so that the first attribute is the root, if for example the type is y, then y->attribute10, that is a branch with another splitting node (attribute10), and if for example its ?, ?->democrat, that is a leaf with class democrat**

Decision Tree (gain ratio):

```
attribute3:
attribute3 type 0: y ->  attribute10
attribute3 type 1: ? ->  democrat
attribute3 type 2: n ->  democrat

attribute10:
attribute10 type 0: ? ->  republican
attribute10 type 1: n ->  attribute14
attribute10 type 2: y ->  attribute8

attribute14:
attribute14 type 0: n ->  attribute2
attribute14 type 1: y ->  attribute9
attribute14 type 2: ? ->  republican

attribute2:
attribute2 type 0: n ->  republican
attribute2 type 1: y ->  attribute15

attribute15:
attribute15 type 0: y ->  republican
attribute15 type 1: ? ->  attribute0

attribute0:
attribute0 type 0: y ->  republican
attribute0 type 1: n ->  democrat
```

Accuracy Results:

```
republican republican
republican republican
republican republican
accuracy: 0.972414
```

**This is a snippet of the results from the testing phase, where the left side is the actual class of the item, and the right side is the predicted class. We calculated the accuracy by finding the number of correctly classified items over the total number of items to be classified.

Dataset 2: breast cancer

Decision Tree (information gain):

```
attribute5:
attribute5 type 0: > 2.000000 ->  attribute4
attribute5 type 1: <= 2.000000 ->  no-recurrence-events

attribute4:
attribute4 type 0: no ->  attribute8
attribute4 type 1: yes ->  recurrence-events
attribute4 type 2: ? ->  no-recurrence-events

attribute8:
attribute8 type 0: no ->  attribute7
attribute8 type 1: yes ->  recurrence-events

attribute7:
attribute7 type 0: left_low ->  attribute0
attribute7 type 1: central ->  no-recurrence-events
attribute7 type 2: right_up ->  no-recurrence-events
attribute7 type 3: left_up ->  no-recurrence-events
attribute7 type 4: ? ->  recurrence-events

attribute0:
attribute0 type 0: <= 45.000000 ->  attribute2
attribute0 type 1: > 45.000000 ->  no-recurrence-events

attribute2:
attribute2 type 0: > 25.000000 ->  attribute1
attribute2 type 1: <= 25.000000 ->  no-recurrence-events

attribute1:
attribute1 type 0: premeno ->  attribute3
```

Decision Tree (gain ratio):

```
attribute5:
attribute5 type 0: > 2.000000 ->  attribute4
attribute5 type 1: <= 2.000000 ->  no-recurrence-events

attribute4:
attribute4 type 0: no ->  attribute3
attribute4 type 1: yes ->  recurrence-events
attribute4 type 2: ? ->  no-recurrence-events

attribute3:
attribute3 type 0: <= 12.000000 ->  attribute8
attribute3 type 1: > 12.000000 ->  no-recurrence-events

attribute8:
attribute8 type 0: no ->  attribute2
attribute8 type 1: yes ->  recurrence-events

attribute2:
attribute2 type 0: > 25.000000 ->  attribute7
attribute2 type 1: <= 25.000000 ->  no-recurrence-events

attribute7:
attribute7 type 0: left_low ->  attribute0
attribute7 type 1: left_up ->  no-recurrence-events
attribute7 type 2: right_up ->  no-recurrence-events
attribute7 type 3: ? ->  recurrence-events
attribute7 type 4: central ->  recurrence-events

attribute0:
attribute0 type 0: <= 45.000000 ->  attribute1
```

Dataset 3: mushrooms

Decision Tree (information gain):

```
attribute4:
attribute4 type 0: p ->  p
attribute4 type 1: a ->  e
attribute4 type 2: l ->  e
attribute4 type 3: n ->  attribute19
attribute4 type 4: f ->  p
attribute4 type 5: c ->  p
attribute4 type 6: y ->  p
attribute4 type 7: s ->  p
attribute4 type 8: m ->  p

attribute19:
attribute19 type 0: n ->  e
attribute19 type 1: k ->  e
attribute19 type 2: w ->  attribute21
attribute19 type 3: h ->  e
attribute19 type 4: r ->  p
attribute19 type 5: o ->  e
attribute19 type 6: y ->  e
attribute19 type 7: b ->  e

attribute21:
attribute21 type 0: w ->  e
attribute21 type 1: l ->  attribute2
attribute21 type 2: d ->  attribute7
attribute21 type 3: g ->  e
attribute21 type 4: p ->  e
```

Decision Tree (gain ratio):

```
attribute4:
attribute4 type 0: p ->  p
attribute4 type 1: a ->  e
attribute4 type 2: l ->  e
attribute4 type 3: n ->  attribute19
attribute4 type 4: f ->  p
attribute4 type 5: c ->  p
attribute4 type 6: y ->  p
attribute4 type 7: s ->  p
attribute4 type 8: m ->  p

attribute19:
attribute19 type 0: n ->  e
attribute19 type 1: k ->  e
attribute19 type 2: w ->  attribute16
attribute19 type 3: h ->  e
attribute19 type 4: r ->  p
attribute19 type 5: o ->  e
attribute19 type 6: y ->  e
attribute19 type 7: b ->  e

attribute16:
attribute16 type 0: w ->  attribute7
attribute16 type 1: y ->  p

attribute7:
attribute7 type 0: b ->  e
attribute7 type 1: n ->  attribute6
```

Dataset 4: soybean small

Decision Tree (information gain):

```
attribute21:
attribute21 type 0: <= 1.000000 ->  attribute20
attribute21 type 1: > 1.000000 ->  attribute1

attribute20:
attribute20 type 0: > 1.000000 ->  D1
attribute20 type 1: <= 1.000000 ->  D3

attribute1:
attribute1 type 0: <= 0.000000 ->  D2
attribute1 type 1: > 0.000000 ->  D4
```

Decision Tree (gain ratio):

```
attribute21:
attribute21 type 0: <= 1.000000 ->  attribute20
attribute21 type 1: > 1.000000 ->  attribute1

attribute20:
attribute20 type 0: > 1.000000 ->  D1
attribute20 type 1: <= 1.000000 ->  D3

attribute1:
attribute1 type 0: <= 0.000000 ->  D2
attribute1 type 1: > 0.000000 ->  D4
```

Dataset 5: adult stretch

Decision Tree (information gain):

```
attribute3:
attribute3 type 0: ADULT ->  T
attribute3 type 1: CHILD ->  F
```

Decision Tree (gain ratio):

```
attribute3:
attribute3 type 0: ADULT ->  T
attribute3 type 1: CHILD ->  F
```

Dataset 6: anneal

Decision Tree (information gain):

```
attribute11:
attribute11 type 0: G ->  attribute2
attribute11 type 1: E ->  3
attribute11 type 2: D ->  3
attribute11 type 3: F ->  3
attribute11 type 4: ? ->  2

attribute2:
attribute2 type 0: A ->  attribute4
attribute2 type 1: ? ->  3
attribute2 type 2: W ->  3
attribute2 type 3: M ->  2
attribute2 type 4: R ->  3

attribute4:
attribute4 type 0: <= 42.000000 ->  3
attribute4 type 1: > 42.000000 ->  attribute16

attribute16:
attribute16 type 0: M ->  attribute36
attribute16 type 1: B ->  3
attribute16 type 2: ? ->  3

attribute36:
attribute36 type 0: <= 300.000000 ->  3
attribute36 type 1: > 300.000000 ->  attribute27
```

Decision Tree (gain ratio):

```
attribute23:
attribute23 type 0: ? ->  attribute24
attribute23 type 1: Y ->  1

attribute24:
attribute24 type 0: ? ->  attribute19
attribute24 type 1: Y ->  5

attribute19:
attribute19 type 0: ? ->  attribute1
attribute19 type 1: C ->  5

attribute1:
attribute1 type 0: C ->  attribute18

attribute18:
attribute18 type 0: ? ->  attribute22

attribute22:
attribute22 type 0: ? ->  attribute25

attribute25:
attribute25 type 0: ? ->  attribute28

attribute28:
attribute28 type 0: ? ->  attribute29

attribute29:
attribute29 type 0: ? ->  attribute30
```

Dataset 7: balance scale

Decision Tree (information gain):

```
attribute0:
attribute0 type 0: B ->  attribute1
attribute0 type 1: R ->  5
attribute0 type 2: L ->  1

attribute1:
attribute1 type 0: <= 3.000000 ->  attribute2
attribute1 type 1: > 3.000000 ->  4

attribute2:
attribute2 type 0: <= 3.000000 ->  attribute3
attribute2 type 1: > 3.000000 ->  4

attribute3:
attribute3 type 0: <= 3.000000 ->  1
attribute3 type 1: > 3.000000 ->  1
```

Decision Tree (gain ratio):

```
attribute0:
attribute0 type 0: B ->  attribute1
attribute0 type 1: R ->  5
attribute0 type 2: L ->  1

attribute1:
attribute1 type 0: <= 3.000000 ->  attribute2
attribute1 type 1: > 3.000000 ->  4

attribute2:
attribute2 type 0: <= 3.000000 ->  attribute3
attribute2 type 1: > 3.000000 ->  4

attribute3:
attribute3 type 0: <= 3.000000 ->  1
attribute3 type 1: > 3.000000 ->  1
```

Dataset 8: cmc

Decision Tree (information gain):

```
attribute1:
attribute1 type 0: <= 2.000000 ->  attribute0
attribute1 type 1: > 2.000000 ->  1

attribute0:
attribute0 type 0: <= 32.000000 ->  attribute7
attribute0 type 1: > 32.000000 ->  1

attribute7:
attribute7 type 0: > 2.000000 ->  attribute5
attribute7 type 1: <= 2.000000 ->  1

attribute5:
attribute5 type 0: > 0.000000 ->  attribute8
attribute5 type 1: <= 0.000000 ->  1

attribute8:
attribute8 type 0: <= 0.000000 ->  attribute6
attribute8 type 1: > 0.000000 ->  2

attribute6:
attribute6 type 0: <= 2.000000 ->  attribute2
attribute6 type 1: > 2.000000 ->  3

attribute2:
attribute2 type 0: > 2.000000 ->  attribute4
attribute2 type 1: <= 2.000000 ->  3
```

Decision Tree (gain ratio):

```
attribute1:
attribute1 type 0: <= 2.000000 ->  attribute0
attribute1 type 1: > 2.000000 ->  1

attribute0:
attribute0 type 0: <= 32.000000 ->  attribute7
attribute0 type 1: > 32.000000 ->  1

attribute7:
attribute7 type 0: > 2.000000 ->  attribute8
attribute7 type 1: <= 2.000000 ->  1

attribute8:
attribute8 type 0: <= 0.000000 ->  attribute4
attribute8 type 1: > 0.000000 ->  1

attribute4:
attribute4 type 0: > 0.000000 ->  attribute5
attribute4 type 1: <= 0.000000 ->  3

attribute5:
attribute5 type 0: > 0.000000 ->  attribute6
attribute5 type 1: <= 0.000000 ->  1

attribute6:
attribute6 type 0: <= 2.000000 ->  attribute2
attribute6 type 1: > 2.000000 ->  3
```

Dataset 9: chess

Decision Tree (information gain):

```
attribute20:
attribute20 type 0: f ->  attribute9
attribute20 type 1: t ->  won

attribute9:
attribute9 type 0: f ->  attribute32
attribute9 type 1: t ->  nowin

attribute32:
attribute32 type 0: f ->  attribute31
attribute32 type 1: t ->  nowin

attribute31:
attribute31 type 0: f ->  attribute5
attribute31 type 1: t ->  nowin

attribute5:
attribute5 type 0: f ->  attribute34
attribute5 type 1: t ->  won

attribute34:
attribute34 type 0: t ->  attribute14
attribute34 type 1: f ->  nowin

attribute14:
attribute14 type 0: n ->  attribute22
attribute14 type 1: w ->  won
attribute14 type 2: b ->  attribute0
```

Decision Tree (gain ratio):

```
attribute20:
attribute20 type 0: f ->  attribute9
attribute20 type 1: t ->  won

attribute9:
attribute9 type 0: f ->  attribute32
attribute9 type 1: t ->  nowin

attribute32:
attribute32 type 0: f ->  attribute31
attribute32 type 1: t ->  nowin

attribute31:
attribute31 type 0: f ->  attribute13
attribute31 type 1: t ->  nowin

attribute13:
attribute13 type 0: f ->  attribute27
attribute13 type 1: t ->  nowin

attribute27:
attribute27 type 0: f ->  attribute5
attribute27 type 1: t ->  nowin

attribute5:
attribute5 type 0: f ->  attribute34
attribute5 type 1: t ->  won
```

Dataset 10: adult

Decision Tree (information gain):

```
attribute7:
attribute7 type 0: Not-in-family ->  attribute3
attribute7 type 1: Husband ->  <=50K
attribute7 type 2: Wife ->  <=50K
attribute7 type 3: Own-child ->  <=50K
attribute7 type 4: Unmarried ->  <=50K
attribute7 type 5: Other-relative ->  <=50K

attribute3:
attribute3 type 0: Bachelors ->  attribute6
attribute3 type 1: HS-grad ->  <=50K
attribute3 type 2: 9th ->  <=50K
attribute3 type 3: Masters ->  <=50K
attribute3 type 4: Assoc-acdm ->  <=50K
attribute3 type 5: Assoc-voc ->  <=50K
attribute3 type 6: 7th-8th ->  <=50K
attribute3 type 7: Some-college ->  <=50K
attribute3 type 8: Doctorate ->  >50K
attribute3 type 9: 1st-4th ->  <=50K
attribute3 type 10: 10th ->  <=50K
attribute3 type 11: Preschool ->  <=50K
attribute3 type 12: 11th ->  <=50K
attribute3 type 13: Prof-school ->  >50K
attribute3 type 14: 5th-6th ->  <=50K
attribute3 type 15: 12th ->  <=50K

attribute6:
attribute6 type 0: Adm-clerical ->  attribute10
attribute6 type 1: Exec-managerial ->  <=50K
attribute6 type 2: Handlers-cleaners ->  <=50K
```

Decision Tree (gain ratio):

```
attribute10:
attribute10 type 0: <= 49999.000000 ->  attribute5
attribute10 type 1: > 49999.000000 ->  >50K

attribute5:
attribute5 type 0: Never-married ->  attribute11
attribute5 type 1: Married-civ-spouse ->  <=50K
attribute5 type 2: Divorced ->  <=50K
attribute5 type 3: Married-spouse-absent ->  <=50K
attribute5 type 4: Separated ->  <=50K
attribute5 type 5: Married-AF-spouse ->  <=50K
attribute5 type 6: Widowed ->  <=50K

attribute11:
attribute11 type 0: <= 2178.000000 ->  attribute12
attribute11 type 1: > 2178.000000 ->  >50K

attribute12:
attribute12 type 0: <= 50.000000 ->  attribute7
attribute12 type 1: > 50.000000 ->  <=50K

attribute7:
attribute7 type 0: Not-in-family ->  attribute3
attribute7 type 1: Own-child ->  <=50K
attribute7 type 2: Unmarried ->  <=50K
attribute7 type 3: Other-relative ->  <=50K
```

Server Results:
GPU Classification Algorithms:

```
------------------------------------------------------------
**LogisticRegression**
Accuracy:  95.5
The time of execution of LogisticRegression is : 49.726247787475586 ms
The Memory utilized for LogisticRegression is : 222839 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:  98.5
The time of execution of DecisionTreeClassifier is : 12.81881332397461 ms
The Memory utilized for DecisionTreeClassifier is : 19842 KB
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       340
           1       0.97      0.93      0.95        60

    accuracy                           0.98       400
   macro avg       0.98      0.96      0.97       400
weighted avg       0.98      0.98      0.98       400

------------------------------------------------------------
**KNeighborsClassifier**
Accuracy:  96.5
The time of execution of KNeighborsClassifier is : 25.962114334106445 ms
The Memory utilized for KNeighborsClassifier is : 170854 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

------------------------------------------------------------
```

```
**SVC**
Accuracy:  85.0
The time of execution of SVC is : 120.86272239685059 ms
The Memory utilized for SVC is : 321810 KB
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/m
ed and being set to 0.0 in labels with no predicted samples. Use `z
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/m
ed and being set to 0.0 in labels with no predicted samples. Use `z
  _warn_prf(average, modifier, msg_start, len(result))
/opt/miniconda3/envs/ese_env/lib/python3.10/site-packages/sklearn/m
ed and being set to 0.0 in labels with no predicted samples. Use `z
  _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**
Accuracy:  93.25
The time of execution of GaussianNB is : 11.609554290771484 ms
The Memory utilized for GaussianNB is : 17595 KB
              precision    recall  f1-score   support

           0       0.95      0.98      0.96       340
           1       0.84      0.68      0.75        60

    accuracy                           0.93       400
   macro avg       0.89      0.83      0.86       400
weighted avg       0.93      0.93      0.93       400

------------------------------------------------------------
```

```
------------------------------------------------------------
**LinearDiscriminantAnalysis**
Accuracy:  95.0
The time of execution of LinearDiscriminantAnalysis is : 29.453516006469727 ms
The Memory utilized for LinearDiscriminantAnalysis is : 22924 KB
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       340
           1       0.95      0.70      0.81        60

    accuracy                           0.95       400
   macro avg       0.95      0.85      0.89       400
weighted avg       0.95      0.95      0.95       400

------------------------------------------------------------
**QuadraticDiscriminantAnalysis**
Accuracy:  98.25
The time of execution of QuadraticDiscriminantAnalysis is : 16.483306884765625 ms
The Memory utilized for QuadraticDiscriminantAnalysis is : 13371 KB
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       340
           1       0.98      0.90      0.94        60

    accuracy                           0.98       400
   macro avg       0.98      0.95      0.96       400
weighted avg       0.98      0.98      0.98       400

------------------------------------------------------------
**RandomForestClassifier**
Accuracy:  98.25
The time of execution of RandomForestClassifier is : 452.75259017944336 ms
The Memory utilized for RandomForestClassifier is : 90934 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

------------------------------------------------------------
**AdaBoostClassifier**
```

```
------------------------------------------------------------
**AdaBoostClassifier**
Accuracy:  98.75
The time of execution of AdaBoostClassifier is : 189.30435180664062 ms
The Memory utilized for AdaBoostClassifier is : 52342 KB
              precision    recall  f1-score   support

           0       0.99      1.00      0.99       340
           1       0.98      0.93      0.96        60

    accuracy                           0.99       400
   macro avg       0.99      0.97      0.97       400
weighted avg       0.99      0.99      0.99       400

------------------------------------------------------------
**GradientBoostingClassifier**
Accuracy:  98.25
The time of execution of GradientBoostingClassifier is : 256.25109672546387 ms
The Memory utilized for GradientBoostingClassifier is : 71881 KB
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       340
           1       0.98      0.90      0.94        60

    accuracy                           0.98       400
   macro avg       0.98      0.95      0.96       400
weighted avg       0.98      0.98      0.98       400

------------------------------------------------------------
```

GPU Parallelization Algorithms:

```
t implement the AsGraphDefInternal() method needed to apply optimizations.
Epoch 1/10
50/50 [==============================] - 8s 85ms/step - loss: 0.1922 - accuracy: 0.9087
Epoch 2/10
50/50 [==============================] - 4s 77ms/step - loss: 0.0929 - accuracy: 0.9613
Epoch 3/10
50/50 [==============================] - 4s 74ms/step - loss: 0.0877 - accuracy: 0.9656
Epoch 4/10
50/50 [==============================] - 4s 72ms/step - loss: 0.0716 - accuracy: 0.9694
Epoch 5/10
50/50 [==============================] - 4s 72ms/step - loss: 0.0596 - accuracy: 0.9744
Epoch 6/10
50/50 [==============================] - 3s 69ms/step - loss: 0.0682 - accuracy: 0.9700
Epoch 7/10
50/50 [==============================] - 4s 70ms/step - loss: 0.0664 - accuracy: 0.9706
Epoch 8/10
50/50 [==============================] - 4s 74ms/step - loss: 0.0639 - accuracy: 0.9744
Epoch 9/10
50/50 [==============================] - 3s 69ms/step - loss: 0.0548 - accuracy: 0.9787
Epoch 10/10
50/50 [==============================] - 3s 65ms/step - loss: 0.0608 - accuracy: 0.9750
2023-12-10 15:33:52.510877: W tensorflow/core/framework/dataset.cc:769] Input of Generator
```

Conclusions:

On average, our algorithm had an accuracy rate of about 80%. We had 100% accuracy rates for some datasets and low accuracy rates below 60% for other sets. For our largest dataset with 32000 entries and 14 attributes, running the program took about 10 seconds and 442 MB of memory. The benchmark algorithms all took significantly less time (less than 1 second) and less memory (10-200 MB), while having higher rates of accuracy (80-99%).

We noticed fluctuations in the accuracy of our algorithm depending on how much of the data was used. Some datasets had increases as we added more data to the tree, while others had decreases or stayed the same throughout. In particular, one interesting case was the cmc dataset (dataset 8).

The accuracy peaked at 75% when 20% of the dataset was used then dropped until around 70% of the dataset was used. Then, the accuracy increased from there peaking again at 75% when all of the data was used. This dataset is a good example of making the tree too big. When the tree becomes too large, some of the branches may be cases that happen once or twice but are not actually a pattern in the larger dataset. We did not prune the tree, so it is likely that many of the branches in the 50% and 75% of the dataset trees were for edge cases.

Although the gain ratio attribute selection method is supposed to be an optimization of the information gain method, we found it had very little impact on the accuracy rate. The difference in accuracy was no greater than 1% for almost all of the datasets. One exception was the anneal dataset (dataset 6), where gain ratio improved the performance by 18%. Interestingly, for this dataset, the information gain performance decreased when we went from 75% of data to 100% of data used to build the tree.

We found one crucial flaw that we believe negatively impacts the accuracy of our algorithm. For both information gain and gain ratio, every iteration of the loop removes one attribute from selection. Since there are many more branches than attributes, we will run out of attributes quickly. This means that many attributes will just be assigned the majority class of its subset instead of another splitting attribute, resulting in numerous mispredictions. Since the algorithm is recursive, branches that are visited earlier will be given precedence over branches that are visited later, even if an attribute is a better split for a later branch. We were not able to test gini index, but we believe it would have a massive increase in accuracy for all datasets because attributes are not removed from the dataset after each iteration since we can generate multiple combinations for each attribute. Additionally, randomizing the data entries used to create the tree might also improve the accuracy, as we noticed that certain data sets had entries of the same class clustered together, which could result in data biasing if not dealt with properly.