A customizable decision tree algorithm

Forrest Bankston

Graduate Computer Science Student

Troy University

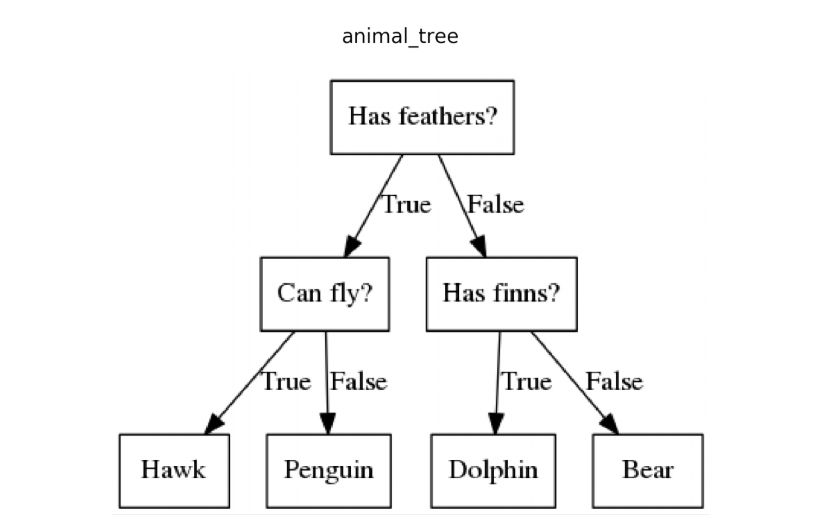


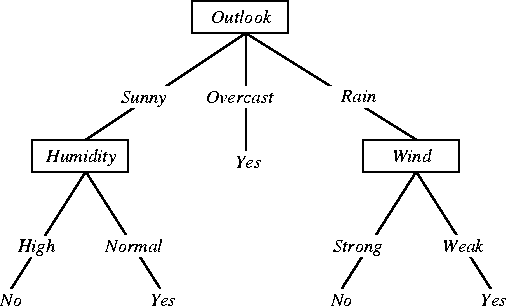
Figure 2: Decision Tree with Multi-Class Classification

*Abstract* — Decision Trees are commonly used in machine learning tasks, such as developing prediction models for some kind of target variable. Decision trees are used for many reasons such as their low time complexity and accuracy when compared to other models. However, the main thing that makes decision trees stand out is their simplicity and familiarity. Most people have worked with a decision tree in some shape or form and will have no problem interpreting them from a basic standpoint. There are several popular decision tree learning algorithms that have been proposed such as the ID3, and C4.5. This paper will attempt to showcase these methods as well as their implementations to discuss their advantages and disadvantages. This paper will also introduce my own customized decision tree algorithm written in python that is intended to give the user a customizable decision tree without sacrificing accuracy or speed.

Keywords— Decision Trees, Machine Learning, Prediction, Classification Tree, Regression Tree

# Introduction

Decision Trees are easily one of the most popular decision-making models used not only in machine learning but also in fields like engineering, business, and law. Decision trees will typically follow a tree-like structure that is sorted from the top-down from the root to the leaf nodes. Each of the nodes in the tree will represent a test of an attribute of the instance, and the branches will represent a possible value of the attribute. The root node is obviously the topmost node, internal nodes are nodes that have a child node, and terminal nodes are nodes that do not have any children. Below are two simple examples of decision trees. In **Figure 1** we are predicting whether or not we will play tennis based on the outlook feature from the popular play tennis or play golf dataset. This is an example of a decision tree for binary classification. **Figure 2** is classifying multiple possibilities of animals and is therefore multi-class classification. For the purpose of machine learning, it is very important to note from **Figure 1** that a certain value, *overcast*, seems like it could possibly be more influential for predicting our target variable than the other feature values. Both decision trees use only categorical data, but keep in mind that decision trees are also capable of handling and predicting quantitative data too depending on the choice of algorithm used.



There have been several noteworthy decision tree learning algorithms introduced over the years, such as ID3 (Iterative Dichotomiser 3) **[1],** and its successor in the C4.5**[2].** A more modern and very common method would be the CART algorithm, which stands for Classification and Regression Trees **[3].** CART algorithms have many applications and are capable of producing a classification or regression tree depending on if the dependent variable is categorical or numerical, respectively. CART algorithms are very popular and an optimized version of it is used when employing the scikit-learn decision tree package. CHAID stands for Chi-Square automatic interaction detection and is not quite as popular as the other methods but still useful **[4].** This paper will give an introduction to these decision tree algorithms and showcase some test results. The last section of this paper will introduce my own customizable decision tree algorithm in python with its own test results.

# ID3

The purpose of machine learning in this context is to move beyond the training set and create a decision tree that is capable of correctly classifying unseen data objects as well as the training objects. To achieve this goal, the decision tree will need to capture and represent some kind of meaningful relationship between the class of an object and its attribute values. Therefore, it is theorized that given the choice of two accurate decision trees, it is sensible to prefer the simpler one on the basis that it is more likely to visualize a structure that is inherent to the problem. This means that ID3 follows the Occam’s Razor principle by preferring the simplest possible tree. **Figure 3** showcases another, much more complex decision tree for the play tennis dataset. While this decision tree is technically correct for the dataset, it is much more complex, and takes away one of the key advantages of decision trees: simplicity and interpretability. Therefore, we would much prefer to use the simpler decision tree as it would be expected to correctly classify more results while also being a better visual representation. Based off of this, you might think that a good approach for a decision tree algorithm would be to calculate all possible decision trees, and then select the simplest one to use. This is a simple approach in theory but will run into extremely high time complexities in cases when the task is anything but small. ID3 is somewhat designed to be the opposite of that approach. The purpose of ID3 is to construct an accurate decision tree with a low time complexity for a dataset that is large and has many features. However, Quinlan claims that this approach does not absolutely guarantee that it will find the best decision tree possible for every instance, however. **[1].** Therefore, the ID3 algorithm is only capable of achieving local optimum and not global optimum.

Figure 1: Decision Tree classifier formed from the play tennis/golf dataset

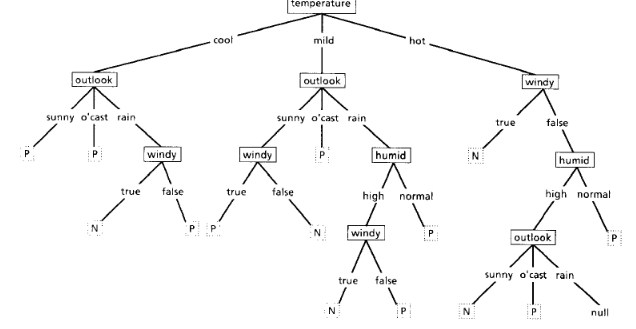


Figure 3: A complex decision tree

## How ID3 Works

In order to keep decision trees as simple as possible, ID3 aims to determine which feature to test first using a greedy strategy (select the best local option on each iteration and never backtrack).ID3 has an iterative structure, meaning that it will execute a sequence of statements:

* A subset of the training set is chosen at random and a decision tree is formed from this subset that will correctly classify all of the objects in the subset

A picture containing text

Description automatically generated

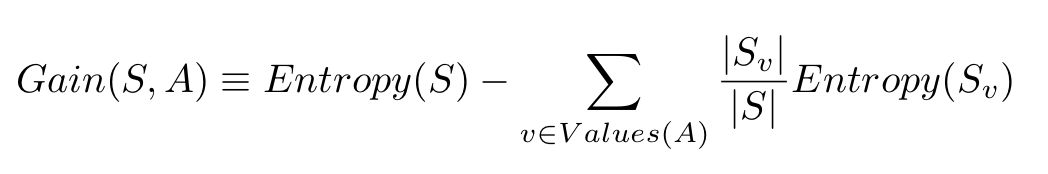


Figure 4: Entropy and Information Gain Formulas

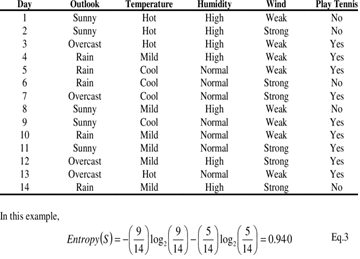
* Then, all other objects in the training set will be classified using this tree
* If the tree is correct for all of the objects, then it will be correct for the entire training set and the process can terminate
* If the tree is not correct for all the objects, then a selection will be made of the incorrectly classified objects to add to the subset, and the process will continue

In other words: The ID3 algorithm will examine all of the possible features and choose the one that will then maximize gain (A), form a tree, and then recursively do this same process until subtrees are formed for all of the subsets. How well a given attribute maximizes gain by splitting the training set is known as **information gain.** This means that if we ask this feature as a question, will it give us some meaningful relationship between the dataset and target variable? The feature that has the most information gain will then be used as the root node so that it will be tested first and keep the decision tree as simple as possible. **Entropy** is a measurement of disorder. In this context it is used as a measurement of uncertainty to measure the information gain of a node. We could say that a dataset of only blue dots would have low(actually zero) entropy, while mixed blues, greens, and reds would have a much higher entropy:  

The formulas for entropy as well as information gain are given below in **Figure 4.** Entropy is the first formula, and P­­­i is simply the probability of a class occurring. Information gain is the second formula. S refers to the entire set of examples from the dataset. A is the feature being tested. This feature is being tested to see if it would make a good split in the dataset. |S| refers to the number of examples, and |Sv| is the number of examples for the current value of attribute A.

## Example Calculation

The example below in **Figure 5** shows the golf/tennis data set with 14 objects, 9 of class P(yes) and 5 of class N(no). Information gain is found by using a calculated entropy measurement for each specific attribute so that we can determine which one is the most influential and put it at the root of the tree. Following the example below in **Figure 5,** you can see that outlook is selected as the highest information gain feature and should be put at the root of the tree. The feature with the lowest entropy will be the one that has the highest information gain. Humidity has the next highest information gain and should therefore be close to the root, followed by Wind and Temperature. After building the tree, it can be used to classify a case by starting at the root and moving through until encountering a leaf. At every one of the non-leaf decision nodes, the test determines the case’s outcome, and the attention will shift to the root of the subtree that corresponds to this particular outcome. Once this process eventually comes to a leaf, the class of the case is then predicted at this leaf. This is the basis behind building and using a decision classifier tree in this context. The ID3 algorithm is certainly amazing, especially considering it was founded around 50 years ago. Quinlan claims that in only a few iterations, this approach will find several correct decision trees for training sets of up to thirty-thousand objects with up to fifty features. However, this is the first of many notable decision tree algorithms and there have been many improvements built upon it. The ID3 algorithm is certainly amazing, especially considering it was founded around 50 years ago. Quinlan claims that in only a few iterations, this approach will find several correct decision trees for training sets of up to thirty-thousand objects with up to fifty features. However, this is the first of many notable decision tree algorithms and there have been many improvements built upon it.



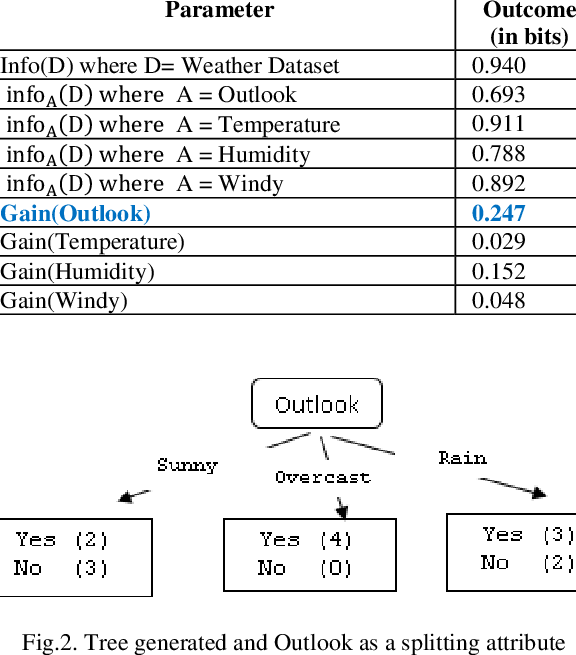


Figure 5: Example Entropy and Information Gain Calculation

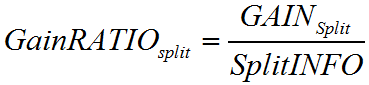
# C4.5

The ID3 decision tree algorithm that has been previously described gives a basic approach to inducing decision trees: a top-down, recursive, depth-first partitioning of a dataset that begins at the root node and finishes at the leaf nodes. ID3 is fast and creates a simple decision tree with understandable decision rules. This algorithm will generally work quite well in theory but has some issues that simply cannot be ignored. The main things that were overlooked when designing ID3 were that it can only handle categorical features and a clean dataset There are many reasons why you would want to be able to use quantitative features for a decision tree that will be discussed soon. A clean dataset is also not possible in many cases and most real-life datasets will have some kind of noise, such as missing feature values. This section will aim to explain the direct successor to the ID3 algorithm that was also introduced by Quinlan: C4.5. The C4.5 algorithm was made with the goal of keeping the advantageous properties of the ID3 while providing a solution for its main faults. Therefore, this section will mainly cover how C4.5 has improved over its predecessor with a variety of new modifications.

## Information Gain Ratio

Before going further into the C4.5 algorithm, it is important to understand a key issue with a measurement of the ID3 algorithm. As stated above, the ID3 algorithm will use entropy to calculate information gain, which then helps to decide how to build the most efficient tree. This approach is very simple and works well, but also encounters some issues. The main notable drawback of information gain is that it strongly prefers features that have many levels, or classes. Many features have multiple possible values and this is most likely the case in real-life situations as well. The entropy-based information gain used in ID3 highly prefers features with many possible values because it can then split the data into many small subsets. However, this is not good because these small subsets will tend to have no correlation between the target feature and the descriptive feature and make the decision tree more complex. A useful way of addressing this problem that is employed in C4.5 is the **information gain ratio**, or simply just **gain ratio**. The information gain ratio can be computed by dividing the information gain of a feature by some **split information value.** The split information value will represent the potential information generated by splitting the training set **D** into **v** partitions, corresponding to **v** outcomes on attribute **A**. In other words, the calculated gain ratio will take into account the number and size of branches when choosing a feature to split the data. The formula for gain ratio as well as split info is given below in **Figure 6:**

Quinlan declares in **[2]** that the information gain ratio criterion is very robust and typically gives a better choice of test than the previous information gain criterion. Quinlan also declares that the information gain ratio can be advantageous in many cases even when all features are binary but have different proportions of cases that are associated with these two binary values.



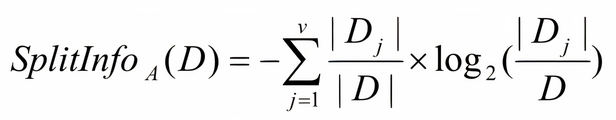


Figure 6: Gain Ratio and Split Info Formulas

## Unknown Feature Values

It is simply a sad fact of life that data will oftentimes have missing values in real-life situations. This could occur for any number of reasons, such as the value was not recorded when the rest of the data was recorded, or the person deciphering the information could not complete it correctly. Regardless of the reason, real-life datasets oftentimes have missing values and ID3 is not capable of handling a data entry with a missing value. Assuming that we want to optimize the ID3 algorithm to able to change these things, there are two available choices: Discard a significant portion of available data or amend the algorithm.

Thankfully, Quinlan made the choice of amending the algorithm. There are many possible choices when deciding how to handle missing data **[5].** Most are based on either filling in these missing feature values with the most probable value or looking at the probability distribution of these feature values. Some of these approaches are definitely inferior, but there is no single method that stands out as being the best in all cases. C4.5 will generally assume that unknown test outcomes are distributed probabilistically according to some relative frequency of already known outcomes. Then a case with an unknown test outcome will be divided into fragments with proportional weights to these relative frequencies, resulting in a single case that can follow multiple paths down the tree. To give a good example of this, we can refer back to the play tennis data set from **Figure 5**. Let us suppose that we have an instance in the dataset where the outlook is sunny, but we are missing a value for humidity. Also suppose our training data has 2 instances in which the outlook is sunny, humidity was below 75, and the result was play. Finally, suppose that the dataset also has 3 instances in which the outlook is sunny, humidity is above 75, and the result was don’t play. Therefore, for the testing instance with a missing humidity attribute, the C4.5 algorithm will return a probability distribution of (0.4, 0.6) that corresponds to (Play, Don’t Play). Being able to handle missing values in datasets is a huge advantage that C4.5 has over ID3. This is the basics of just one method for handling missing values, but once again, there are many acceptable methods. In a future section, more specifically when talking about CART, this paper will introduce another method of handling missing feature values that is based upon using a surrogate split.

## Quantitative Data

In the introduction of this paper, it states that the two example decision trees **(Figure 1, 2)** use only categorical data but that quantitative data can also be used depending on the decision tree algorithm. An important upgrade from ID3 is that C4.5 can handle quantitative values in tests (the original implementation of C4.5 by Quinlan in **[2]** notes that feature values can only be discrete and not continuous. Most future versions of C4.5 have been modified to deal with this and support both discrete and continuous values). Being able to handle quantitative data is important for accuracy, but mainly because real-world datasets often have these values. This brings up the obvious question of how does the C4.5 algorithm handle quantitative values and build a decision tree accordingly? There are many possible methods that could be used to answer this question, and different implementations of C4.5 use different methods. The basis, however, is choosing some threshold(midpoint) and splitting the list into those whose attribute value is above the threshold and then those that are less than or equal to it. But how is the threshold at which to make the split chosen? Once again, there are many possible answers to this question. Quinlan’s approach in **[2]** chooses the largest value in the whole training set that does not exceed the midpoint of this value.

## Pruning

So far in this section, we have described three amazing upgrades that were made to upgrade ID3 to C4.5: Information Gain Ratio, handling missing features values, and handling quantitative data. This subsection will introduce another fascinating optimization that was included in the C4.5 algorithm: **Pruning.** The method shown so far for constructing decision trees will continue to subdivide the set of training cases until each of the subsets in the partition contains only a single class, or until no test offers any improvement. This can work well depending on the dataset, but often results in a very complex tree that overfits the data by inferring much more structure than is justified by the training cases. Overfitting of a decision tree is more and more likely to occur as the tree gets deeper or more complicated. This begs the question of how can we modify the decision tree algorithm to solve this issue? As stated, the solution to this is **pruning,** and there are two different variations of it: **Pre-pruning,** and **post-pruning.** Pre-pruning is defined as preventing a complete induction of the training set, or not allowing the tree to grow to its full possible length. Post-pruning is much more common and is defined as letting the tree grow to its full length, and then removing some of its structure that proves to be less useful. Pre-pruning is initially very attractive because it does not waste time assembling structure that will not be used in the final version of the tree. However, pre-pruning is simply just not as good in practice as it is in theory. Too high of a stopping threshold will utterly terminate division before the benefits of the subsequent splits become clear. On the contrary, too low of a stopping value results in little to no simplification of the tree.

Quinlan initially employed pre-pruning for C4.5 decision trees, but eventually moved to post-pruning. C4.5 now uses post-pruning by allowing decision trees to grow without limits to their full length, and then pruning the overfitted tree that is produced. Overall, growing and post-pruning trees has shown to be slower but more reliable. This paper previously discussed how decision trees that are too complex will be subject to inaccuracy and poor interoperability. At the same time, pruning a decision tree will almost invariably cause it to misclassify some of the training cases. No matter what the case, there will always be trade-offs. However, post-pruning has proven over time to be an effective method for simplifying decision trees while maintaining accuracy and not sacrificing too much speed. An example of post-pruning a decision tree is shown in **Figure 7.** As you can see, the right half of the initial tree is completely pruned to result in a “no loan” value. The theory behind pruning in this case is that if the income has a value of less than 30k, the resulting decision will be “no loan” a majority of the time and there is no reason to keep that certain part of the tree. If the algorithm decides that the right side of the tree leads to “no loan” a significant number of times, such as in this case, then it will be pruned completely. As stated, however, this can lead to inaccuracies because the pruned tree might incorrectly classify instances in which the customer has less than 30k income but still gets a loan. Some kind of error rate is generally used as a measurement to compare the error in predictions made by a decision tree before/after pruning. To measure this error rate, some portion of the training set can be set aside to instead serve as a validation dataset. This is known as **reduced error pruning** and is basically a form of cross-validation.

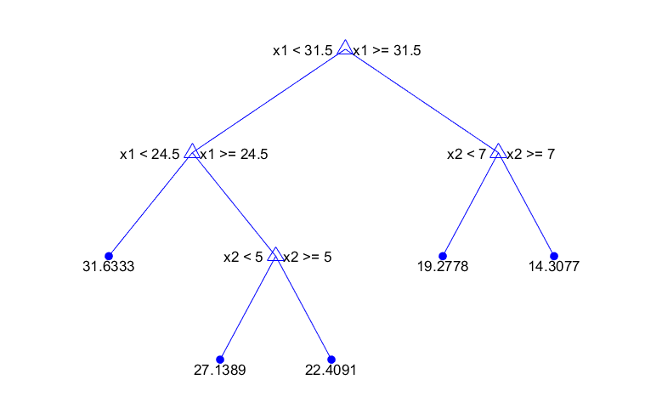


Figure 8: Regression Tree predicting a quantitative value

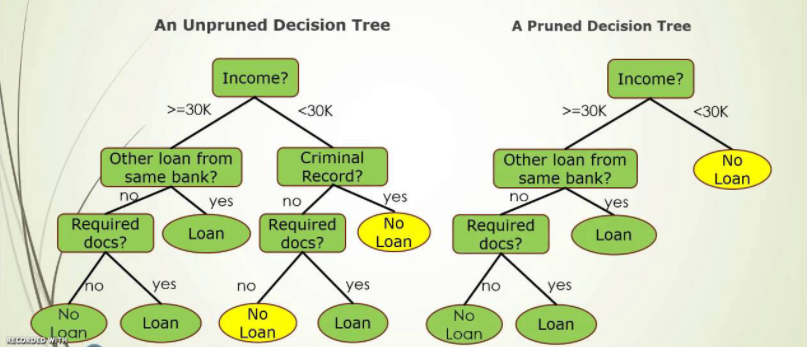


Figure 7: An example of pruning

.

## C5.0

This paper has introduced the ID3 algorithm and its successor, the C4.5 algorithm. The C4.5 algorithm has been extremely popular over the course of history since its inception for its substantial improvements over ID3. C4.5 ranked very highly in data mining algorithm competitions and was known as a landmark accomplishment in machine learning. In more recent years, Quinlan has also introduced the C5.0 algorithm for building decision trees **[6].** C5.0 has been marketed commercially and offers a good number of improvements over its predecessor. This paper will not cover exactly how C5.0 makes these improvements as it would be somewhat redundant. However, it is important to know that the algorithm has increased efficiency in basically every area possible. As well as increased efficiency, C5.0 also has support for several machine learning extensions to be applied:

* **Speed/Memory Usage -** C5.0 is faster than C4.5 by several orders of magnitude and more memory efficient
* **Smaller Decision Trees –** C5.0 typically produces smaller decision trees which is very important for accuracy and readability
* **Boosting –** C5.0 has support for boosting which is a family of algorithms to improve machine learning (Random Forest, AdaBoost)

# CART

This paper has so far introduced the ID3 decision tree algorithm, as well as its optimized successor in the C4.5. The reader should have a fairly good idea by now of the basis behind decision tree algorithms and how they can be used to classify and predict data. Therefore, the following sections will aim to not provide as much detail on decision trees themselves, but instead focus on the differences between decision tree algorithms. The C4.5 algorithm brought with it many new modifications and improvements. One of the huge modifications made to C4.5 was that it was made to be capable of handling quantitative feature values in the dataset. Regardless, C4.5 is only capable of classification. This is not an issue in the decision trees we have seen so far since the target variable is always categorical. However, consider a case in which you are given several features of houses (location, size, number of rooms, etc.) and want to predict the price of a similar house. C4.5 can use this quantitative data for prediction, but it cannot actually make a prediction of quantitative data. Therefore, Classification and Regression (CART) Trees were introduced. Classification trees are used in the previously described algorithms, but the actual CART algorithm behaves differently. This paper has already introduced several examples of classification trees, but the following example in **Figure 8** is an example of a regression tree:

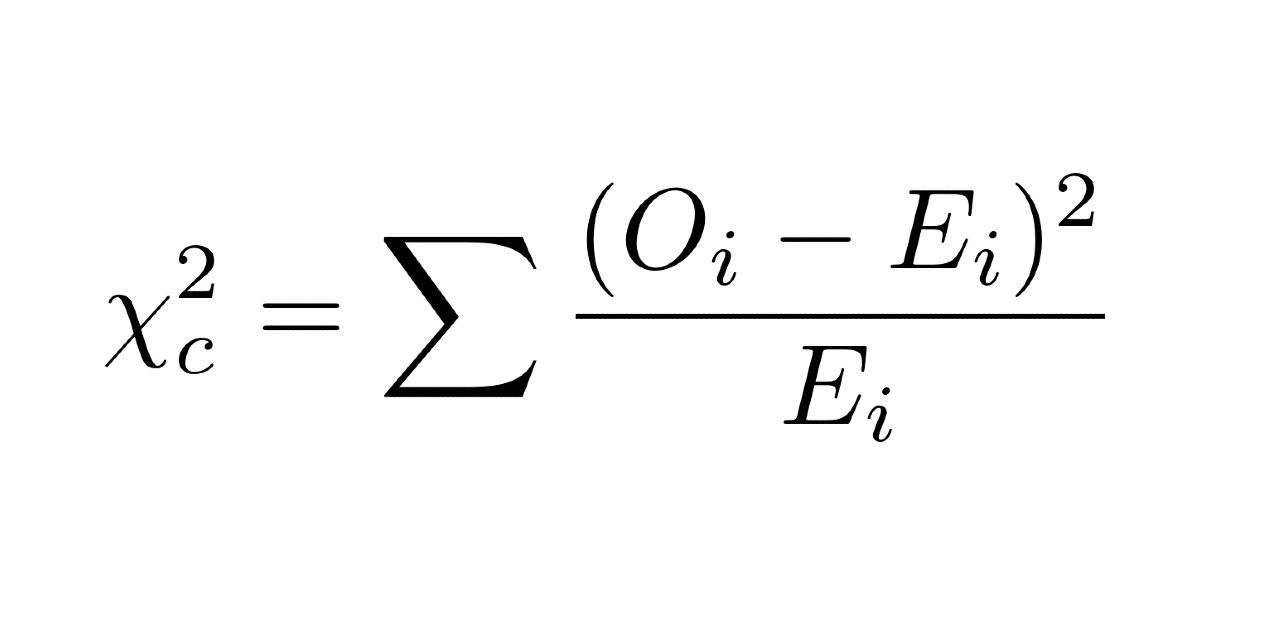


Figure 0: Chi-Squared Formula

## Gini impurity/Surrogate Splits

Instead of using entropy as an impurity measurement, CART uses Gini impurity. Overall, this metric is quite similar but has some differences. The main difference is that instead of multiplying a class’s probability by log2 of itself, Gini will square this class’s probability. Overall, this will usually lead to Gini impurity having a better time complexity without the need for a log calculation. However, Data analysts/scientists should always experiment with both metrics in order to find what works best for their current project. The formula for Gini impurity is given below in **Figure 9:**

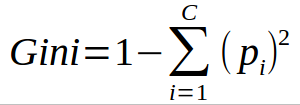


Figure 9: Gini impurity formula

CART also handles missing values slightly differently. Instead of using fractional cases, CART uses surrogate splits. A surrogate split is basically meant to find a replacement split in the case of a missing value. This is done by examining all of the other splits and searching for the one that will give a division of training points most similar to the optimal split. Following this same line of thought, the second-best surrogate split could also be found in a case where the best feature as well as its top surrogate are both missing.

## Binary/Multi Splits

Apart from these differences, C4.5 and CART function similarly overall. However, there is another difference that is not quite as noticeable. C4.5 will only make multi-way splits, while CART only makes binary splits. For an example of a decision tree with multi-way splits, refer to **Figure 1.** For an example of a decision tree with only binary splits, refer to **Figure 2.** The tree in the first figure makes a split and then makes several inquiries. The tree in the second figure only splits between two branches at a single time **[7].**

# CHAID

CHAID is yet another implementation of a decision tree algorithm. CHAID stands for chi-square automatic interaction detection and actually does not see that much use in the field of Machine Learning. Strangely enough, this algorithm was introduced before even ID3, but is still very useful. The main difference between CHAID and other algorithms is its impurity measurement, which is chi squared. This formula is rather simple and is theoretically less computationally taxing than entropy, but more taxing than Gini. The formula is the sum of the observed values subtracted by the expected value, squared, and then divided by the expected value. Because chi is squared, the square root is then taken to result in the final answer. The formula for chi-squared is given in **Figure 10:**

Chi-squared is overall easy to grasp, but still fairly different from the other impurity measurements. Therefore, an example calculation will be shown in **Figure 11**. Consider the same play golf dataset and testing the outlook feature from **Figure 5:**

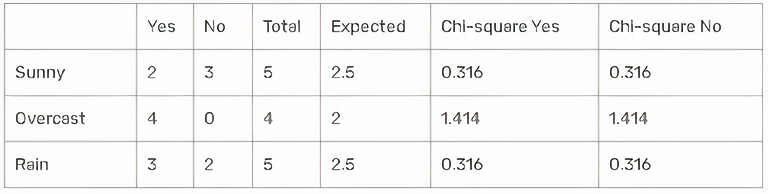


Figure 11: Chi-Squared Table

The total column is the total sum of decisions for each row and the expected column = total/2 because there are two decision classes. Using the formula: √(( 2 - 2.5 )2 / 2.5 ) = 0.316 where the actual is 2 and expected is 2.5. Add up all the values to find chi-square for outlook: 0.316 + 0.316 + 1.414 + 1.414 + 0.316 + 0.316 = 4.092. Similar analysis gives: Temperature = 2.569 Humidity = 3.207 Wind = 1.604. Therefore, we will again select outlook as the root node because it has the highest chi-square value.

# Testing Decision Tree Algorithms

The primary Python library that I have used for testing purposes over the course of this project is chefboost for decision tree algorithms. This is a very convenient library that is capable of implementing not only the ID3 decision tree algorithm, but also C4.5, CART, CHAID, and more. A dataset can be loaded into a pandas data frame, and chefboost will create a decision tree classifier in the form of a python file. To test the results of these algorithms, we will be using the Breast Cancer Wisconsin dataset. This is a very popular machine learning dataset and is very useful for many reasons. This dataset only has two possible decision values of Benign or Malignant, meaning that this will be binary classification. This dataset has 569 instances to test and thirty-two features values. The feature values consist of quantitative measurements of a tumor such area, smoothness, perimeter, etc. Because this dataset contains quantitative values, ID3 technically could not be applied. However, it is fair to say that C4.5 will receive similar if not better results. **Results:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | C4.5 | CART | CHAID |
| Accuracy | 98.95 % | 100.0 % | 99.65 |
| Execution Time | 61.16 Seconds | 30.27 Seconds | 40.13 Seconds |

As you can see, the CART algorithm theoretically performed the best with no misclassifications and a good time complexity. CHAID performed similarly, and C4.5 performed the worst, mainly because of its speed. However, the fact that CART received 100% accuracy is probably not a good thing. Recall that machine learning models are said to be overfitting if they fit the data too well. Because CART classified every case correctly, it is quite possible that it is overfitting the data and might have trouble classifying future datasets. Therefore, because CHAID had a good time complexity and a few misclassifications, it is safe to say that CHAID actually performed the best in this case.

# My Implementation

My main goals for the implementation part of this project are as follows:

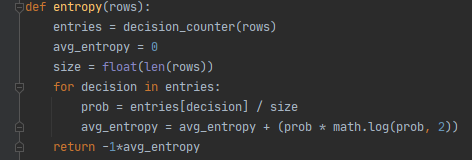
1. Use what I have learned from this course/research to implement my own code from scratch without using (almost) any libraries to actually recreate the process of a decision tree algorithm in python
2. The decision tree algorithm should be customizable
3. The decision tree algorithm should be comparable in accuracy/speed to be competitive with other popular machine learning libraries

## Pseudocode/Python Code

I wanted to implement my own decision tree algorithm to be able to actually fully understand and demonstrate the process of how a decision tree is built and pruned in python. I wanted the decision tree algorithm to be more customizable than the popular libraries because I felt as if they didn’t have enough customization options for the user. Finally, I wanted my own algorithm to be comparable to these other libraries in terms of accuracy and speed so that using my own algorithm would not be a disadvantage to the user. The following is a pseudocode of the algorithm in order to get a general feel for the process:

1. Retrieve dataset (list of rows)
2. Calculate entropy/Gini (based on user selection)
3. Create a list of available inquiries or questions to ask at a specific node
4. Partition the rows into True/False rows based on the inquiry
5. Calculate information gain from entropy/Gini selection from user
6. Update the highest information gain based on each feature or inquiry
7. Update the best inquiry or feature based on this calculated information gain
8. Split the node based on the best inquiry and repeat from step 1 until there are only leaf nodes remaining in the tree
9. Prune the tree while checking accuracy after each pruning (if post-pruning)

Note that I put many comments in my actual code but not in these screenshots in order to save space. Below in **Figure 12** is an example of the entropy and Gini calculations. The user can select their preferred criterion before running the program.



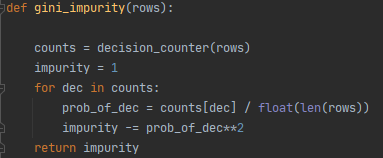


Figure 12: Entropy/Gini Calculations in Python

For some of the class definitions such as for a leaf node and a regular decision node, I followed along with some of the documentation from the scikit-learn developer API. I originally intended to actually use the scikit-learn developer API to create my own customizable decision tree algorithm. However, this ended up being slightly outside the scope of the project. Regardless, this documentation was massively helpful to help me get a good idea of how to build the classes I needed before diving into all the logic that would be required. The class definition for my decision nodes is seen below in **Figure 13.** You will see that a decision node consists of an inquiry, as well as the depth of the node and a unique ID. The decision node class also contains a true branch/false branch in response to the inquiry being made, A.K.A. the feature being used to split the dataset.

The following is one of the most important functions in the whole program. **Figure 14** below displays the function to calculate the best split to make at each node. The beginning of this block defines the best gain and best inquiry to be used throughout the function. The program will then exclude the final column from the data set so that only features are used in the calculation and not labels. The program will then attempt to make a split but only if it actually divides the dataset. If a split is made, the information gain will then be calculated based off of the user’s current impurity metric. Finally, the best feature to split upon along with its information gain will be returned. The function is given below:

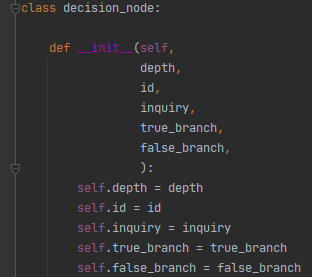


Figure 13: Class Definition for a Decision Node

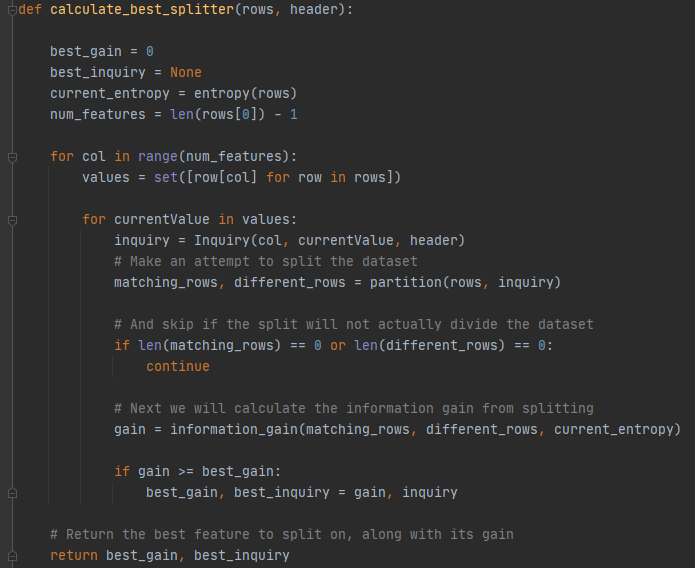


Figure 14: calculate\_best\_splitter Function

One final example is also one of the most important components of the algorithm: Post-pruning. We start off by assuming that pruning the tree will not increase its accuracy. We then prune the tree and calculate accuracy after pruning. If the accuracy of the tree after pruning is improved, then the change will be kept. Additionally, if at any point pruning results in an accuracy of 100%, then pruning will stop immediately. This is a modification I made to improve time complexity in some cases. Afterwards, if the result of pruning increased accuracy, then the system will print this to the user along with the accuracy and final node that was pruned. If pruning does not increase the accuracy, this will instead be outputted to the user.

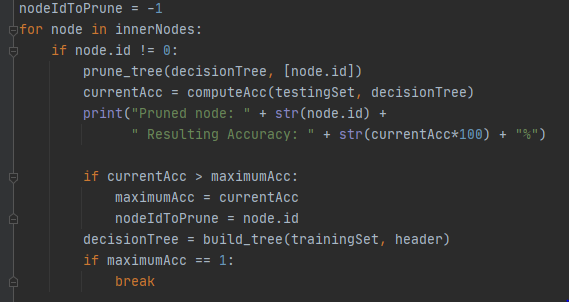
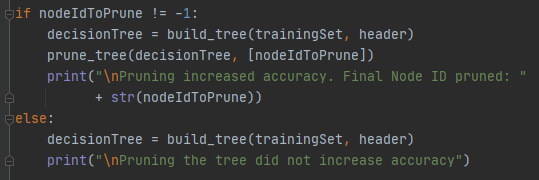
 

Figure 15: Pruning the Tree

## Testing Results

Running my decision tree algorithm on the same breast cancer dataset yielded solid results. For research purposes, I also tested my algorithm on another dataset that has many more test instances but far less attributes. This is the Car Acceptability data set that has 1728 instances. This dataset uses features such as a car’s safety and price to predict whether or not the car will be acceptable to a customer wanting to buy a car. There are five possible decisions, so this is a multi-class classification problem. For these tests, I am using Gini impurity and post-pruning. The results of testing my algorithm on both datasets are below:

|  |  |  |
| --- | --- | --- |
|  | Breast Cancer | Car Acceptability |
| Accuracy | 96.0% | 98.0% |
| Execution Time | 1:34.77 (one minute and thirty-four seconds) | 0:17.15 (seventeen seconds) |

As you can see, my own decision tree algorithm is providing solid results. Based off of the varying results between datasets, I am able to conclude that my algorithm performs well on datasets with many instances but struggles more on those with many features.

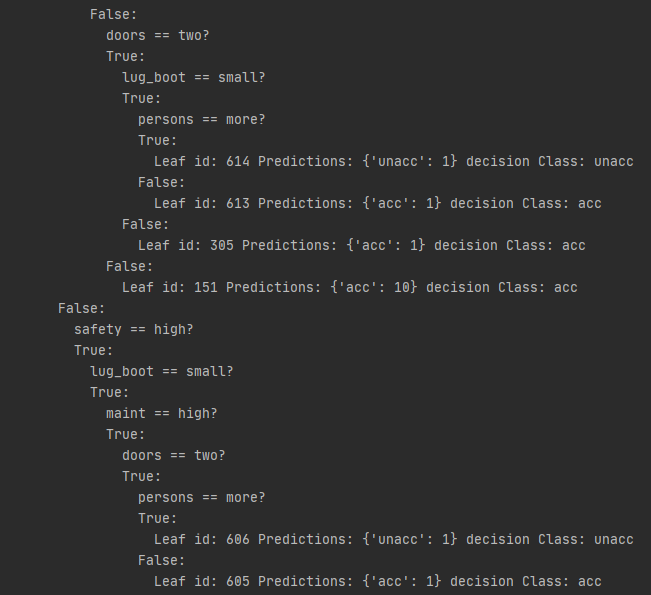
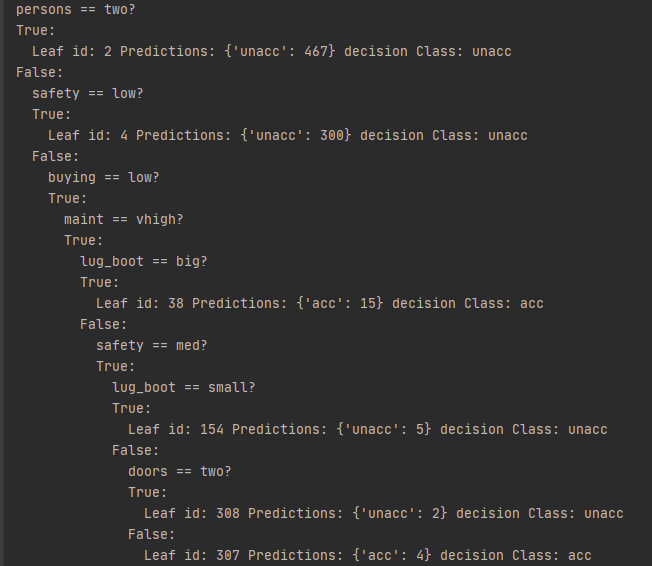


Figure 16: Decision Tree Output

## Final Conclusion/Future Work

As stated, my goals for this project were to create my own decision tree algorithm in python almost completely from scratch only using minimal libraries. I also wanted this algorithm to be customizable and comparable in speed to other popular decision tree libraries. As stated above, I originally wanted to use the sci-kit learn developer API to accomplish this, but that is beyond the scope of this project. Nevertheless, I was definitely successful in my first goal. The only libraries used are sci-kit learn for cross-validation and pandas for loading the dataset into a data frame. I also used two built-in modules: datetime and math. However, datetime is simply for calculating the execution time of the program, and the math library is for the log calculation when measuring entropy.

Customizability is definitely one of the most challenging parts. As stated above, the user can edit their preferred impurity criterion as well as their preferred pruning method by directly editing the python code. I was able to achieve this level of customization; however, I did not fully accomplish my original goals for this part of the project. My original plan was to have as much customization as sci-kit learn and incorporate a simple GUI for customization options. I intend to keep working on this project to be able to implement both a GUI as well as more customization options. As far as the accuracy and execution time, it definitely does not completely match that of other popular libraries. However, my goal was to make it comparable to them and I definitely achieved this goal. When using the breast cancer dataset, my implementation was only down about 3% in accuracy and roughly 30-50 seconds slower. I am very satisfied with the accuracy of my algorithm, but it needs more optimization in order to actually match the speed of other popular decision tree libraries.

When testing on the car acceptability dataset, my algorithm performed much faster which leads me to believe that my main priority for speed improvements should be handling datasets with many features. In addition, I have also concluded that specifically the post-pruning part of the program is the most computationally taxing by far. Therefore, my future goals for this project are as follows:

* Implement more customization options for the user
* Implement a GUI to allow the user to select their dataset and customization options
* Optimize the algorithm as much as possible to reduce execution time, most specifically post-pruning

To conclude, a sample final output of the program is given in **Figure 16**. You will see that this output follows the same basic structure of a decision tree: Make an inquiry. Make a prediction on the true branch and make another inquiry on the false branch. This will repeat until a prediction can be made from the false node, leaving only leaf nodes in the subtree. The process will then repeat for any remaining subtrees until the tree is complete.

##### References

1. Quinlan, J.R. 1986, *Induction of Decision trees*, Machine Learning.
2. Quinlan, J.R. *C4.5: Programs for Machine Learning*. San Mateo, California: Morgan Kaufmann Publishers, Inc.; 1993
3. Breiman, L, Friedman, J H, Olshen, R A, and Stone, C J, 1984, *Classification and regression trees*: Wadsworth, Inc.
4. Kass, G. V. *An Exploratory Technique for Investigating Large Quantities of Categorical Data*. Journal of the Royal Statistical Society. Series C (Applied Statistics), vol. 29, no. 2, 1980, pp. 119–127.
5. Quinlan, J.R. (1989). Unknown attribute values in induction. *Proceedings of the Sixth International Machine Learning Workshop (*pp. 164-168). San Mateo, CA: Morgan Kaufmann
6. Rutvija Pandya, Jayati Pandya, *C5.0Algorithm to Improved Decision Tree with Feature Selection and Reduced Error Pruning, International Journal of Computer Applications (0975 –8887)Volume 117 –No. 16, May 2015*
7. Ad Feelders, *Handling Missing Data in Trees: Surrogate Splits or Statistical Imputation?, Principles of Data Mining and Knowledge Discovery, 1999, Volume 1704*