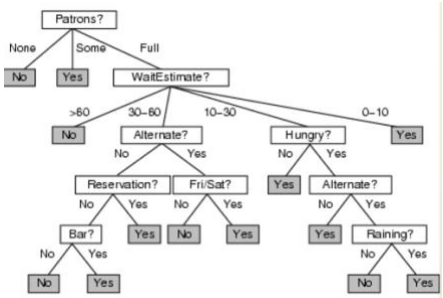
Supervised Learning

Author: Timothy Morren

Project Description: Develop a software agent in Python to learn an ID3 decision tree from labeled classification data.

Decision trees: A decision tree is a model that quantifies/models a decision-making process. It can then be applied in the future to predict which decision will be made for a given stimulus. For example, let's say we want a decision tree for if we will wait for a table when we go to a restaurant. First, we see how many people are there; if its empty, we will not wait; if there are some seats; we will wait; and if its full, we will ask for the wait time. If the wait time is over 60 mins, we will not wait; if the wait time is less than 10, we will wait; and if the wait is between 30-60, we will ask if there is an alternative; if the wait is between 10-30 we ask if we are hungry. And so on. As shown below (figure 1.).



(figure 1.)

Using the Decision tree: To use the decision tree above we are given a situation with all the necessary attributes (Patrons= Full, Wait Estimate= 20, Alternate= Yes, Reservations= No, Bar= No, Fri/Sat= No, Hungry= Yes, Alternate= No, Raining= Yes) and we follow the tree using the attributes at each node to make a decision (I.e. Decision tree). The classification label or response of the tree is when we reach a leaf node (or a node with no sub tree) since it will tell us whether or not we wait. When given attributes and a decision tree we can classify the situation class label (yes wait or no wait). With the given attributes above and figure 1. we can follow the tree and see that we will indeed wait.

Making a Decision tree: There are a myriad of equivalent decision trees that model the same situation. For example, we can see that we could have asked for the wait estimate first but then we would required us to ask how full it is for both a 30-60 min and 10-30 min wait, which would make a decision tree that is larger than the original but still equivalent. Intuitively a larger decision tree would be slower at classification since there are now, on average, more paths to follow to reach a leaf. Thus, a smaller decision tree is better. So how do we create the smallest branching tree? In this paper we will look at a method called ID3.

ID3: ID3 (Iterative Dichotomiser 3) is a decision tree algorithm used for classification tasks. It constructs a tree by recursively partitioning the dataset based on the features' information gain. A correct decision tree will classify objects in the same proportion as their representation in the class.

Information Gain: A metric to measure the effectiveness of attributes in classifying the dataset. It quantifies how much uncertainty in the classification is reduced after splitting the data based on a particular attribute. The higher information gain implies the more useful the attribute is for splitting the data.

Partitioning the Data: After selecting the attribute, ID3 divides the dataset into subsets based on the attribute values.

Recursive Process: It then recursively applies the same process to each subset, creating branches for each value of the chosen attribute, until it either creates a pure leaf node (with only one class) or reaches a stopping criterion.

Stopping Criteria: The recursion stops under conditions like reaching a leaf node, reaching a maximum depth, or if further splitting doesn't significantly increase information gain.

Coding ID3: To code a ID3 decision tree we just follow the steps outlined above; however, we need a mathematical way to calculate information gain, thus finally allowing us to know how to partition the data correctly.

Entropy: To find the max information gain, or how much uncertainty in the classification is reduced after splitting the data based on a particular attribute, we need a way to measure the impurity or randomness in the new datasets so we can compare if it was a good split or not. Entropy, in this context, is the measure of impurity or randomness in a data set. It's calculated by the formula:

$$Entropy(S) = -\sum_{i=1}^{c} p_i * log_2(p_i)$$

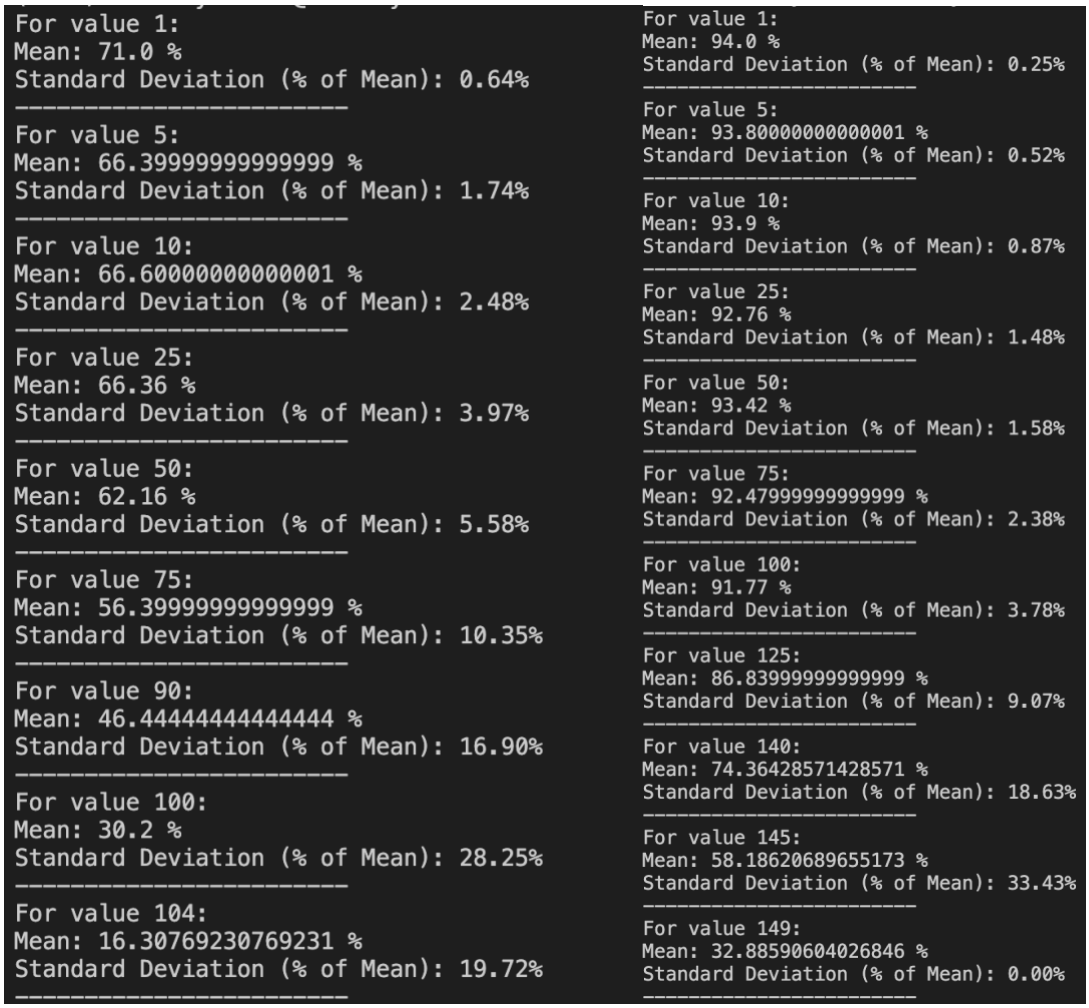
Where, Entropy(S) is the entropy of the dataset S, c is the number of classes in the dataset, and p_i is the probability of occurrence of class "i" in the dataset S. The entropy is maximum when the classes are equiprobable (i.e., when the dataset is perfectly impure or random), and it is minimum (zero) when the dataset contains only one class (i.e., it is perfectly pure).

Applying Entropy: To find the best attribute to split the data on we find the entropy of all the possible split points and compare the total entropy of the system before the split to the weighted entropy of both sub trees after the split.

Testing: To test the ID3 code we first need to understand the input to the ID3 code, or any *Supervised Learning Model*. Supervised learning means we give the model known, classified data (i.e. training data) to make the model with. Since the entropy equation relies on knowing c, the number of classes in the data set, and p_i, the probability of occurrence of class "i" in the dataset, we must have at least one known class in the given training data. In theory, the more training data a model has, the more accurate the probability of occurrence will be and thus a better model for classification. The other input is a validation data set, or a data set with all the attributes need to classify the data but no class attached (or it is hidden from the model). Once the ID3 classifies it we can uncover the attached class and compare if the model predicted the correct class for that data (given attributes).

Code performance: In the outputs below (figure 2.) and (figure 3.) we tested two separated data sets (cancer-data.txt, iris-data.txt) using v = [1, 5, 10, 25, 50, 75, 90 ,100 ,104] and v = [1, 5, 10, 25, 50, 75, 100, 125, 145, 149] , respectively, where "v" is the number of validation data given to the model. "M" is the total number of examples in the data set. (M – v) is the number of training examples. So, the larger v (number of validation examples) is, the smaller M (number of training examples) must get since the data set size is constant at 105 and 150. Below we see the average, or mean, percent of training data correctly classified by the model and the standard deviation of the model's mean percent of training data correctly classified by each v after 100 iterations of randomly splitting the data into training and validation datasets.

(Figure 2.)



(Figure 3.)

Interpreting the results: A higher mean is good since it directly compares if the model predicted the correct class. The mean says, "the model will predict the correct classification about "mean" percent of the time". While the standard deviation tells us how confident the model is, since a lower standard deviation means even if the mean is off, it's not off by much. Meaning this amount of training data is more consistent in creating a model that will predict the classification mean percent of the time. Standard deviation might be a way to tell if the training data is easily classifiable.

Reasoning the results: A simple comparison between the means of the two data sets above will shows that (figure 3.) has a better model for classification than (figure 2.) and is more confident. One possible reason for this discrepancy is that in figure 3's data set there are only 3 possible classes compared to figures 2's 6 possible classifications. There is also more data to train from in figure 3. Both of these factors play into how entropy is calculated. from the data above

Limitation of the code: One issue, shown just above, is that the more possible classifications the model must predict, the less accurate the model will be. Also, the more training data the better the model.

Code optimization: To keep the runtime manageable, the sorted split points were all stored in an array that was never changed, just looped through. Also, for testing, the model could be tested in parallel since the training and output of one model doesn't affect the other models (i.e. independent programs).

References:

https://numpy.org/doc/stable/reference/generated/numpy.bincount.html

explains numpy’s bincount and eargmax