Project 4

Jordan White

May 12th, 2021

1 Part A: Decision Tree Supervised Learner

1.1 Conceptual Overview

A Decision Tree Learner is an algorithm that performs supervised learning using labeled data (input-output vector pairs) in order to construct a decision tree that can effectively classify unseen examples of the data. A decision tree maps many possible sets of assignments for the attributes to a prescribed class assignment (or to a continuous value range). Each attribute is stored in an internal node, each edge contains a possible value for the attribute node it originates from, and each leaf node contains the prescribed classification for that set of assignments.
One of the goals of decision tree learners is to learn the smallest decision tree possible that demonstrates accurate classification of input data. It does this by ordering the attributes from most to least influential on the class type, and expanding the most influential attributes first.

1.2 Project Overview

In this project, I programmed a Decision Tree Learner algorithm in Python 3. The dataset I used was a classification dataset with four discrete classes and six categorical attributes. The possible class values were 'unacceptable', 'acceptable', 'good', and 'very good'. I reduced the latter three classes into one class called 'acceptable or better' so there were only two classes to work with, allowing me to employ simple Boolean classification algorithms.
I also analyzed my algorithm using two different splitting functions: entropy measure and the Gini Index. Entropy measure orders the significance of attributes by the information gain of assigning values to them, while the Gini Index orders the significance of attributes by their probability of being incorrectly classified.
To analyze the performance of my Decision Tree Learner, I downloaded a tree library called anytree from this website. This library includes a function that prints the entire tree in human readable format, which I used to print out the decision tree.
I tested the classification accuracy of the resulting decision tree over five randomized train/test splits for the data-set. I repeated this procedure for both entropy measure and the Gini Index. The results are explained below.

1.3 Results

The decision tree that was formed using the entropy measure splitting function is shown in figure 1. The decision tree formed using the Gini Index splitting function is shown in figure 2. These tree were output to the terminal using the anytree function.

As you can see from Figure 3, the accuracy of the decision tree from Figure 1, which used entropy measure, when taken over five randomized test/train splits, was about 73.1%. This is only slightly better than chance (since about 70.1% of the dataset of car.txt belonged to the 'unacceptable' class). This sub-optimal performance, if not due to inaccurate attribute ordering or a bug in my code, may be due to a level of noise in the data that a simple decision tree structure is not equipped to handle.

As you can see from Figure 4, the accuracy of the decision tree from Figure 2, which used Gini index, when taken over 5 test/train splits, was about 68.5%. This is slightly worse than the entropy measure splitting function, which intuitively makes sense; This is because the entropy measure splitting function calculates the amount of information gained by assigning values to each attribute, which seems more powerful than simply calculating the probability of incorrect assignment for each attribute.

1.4 Reference

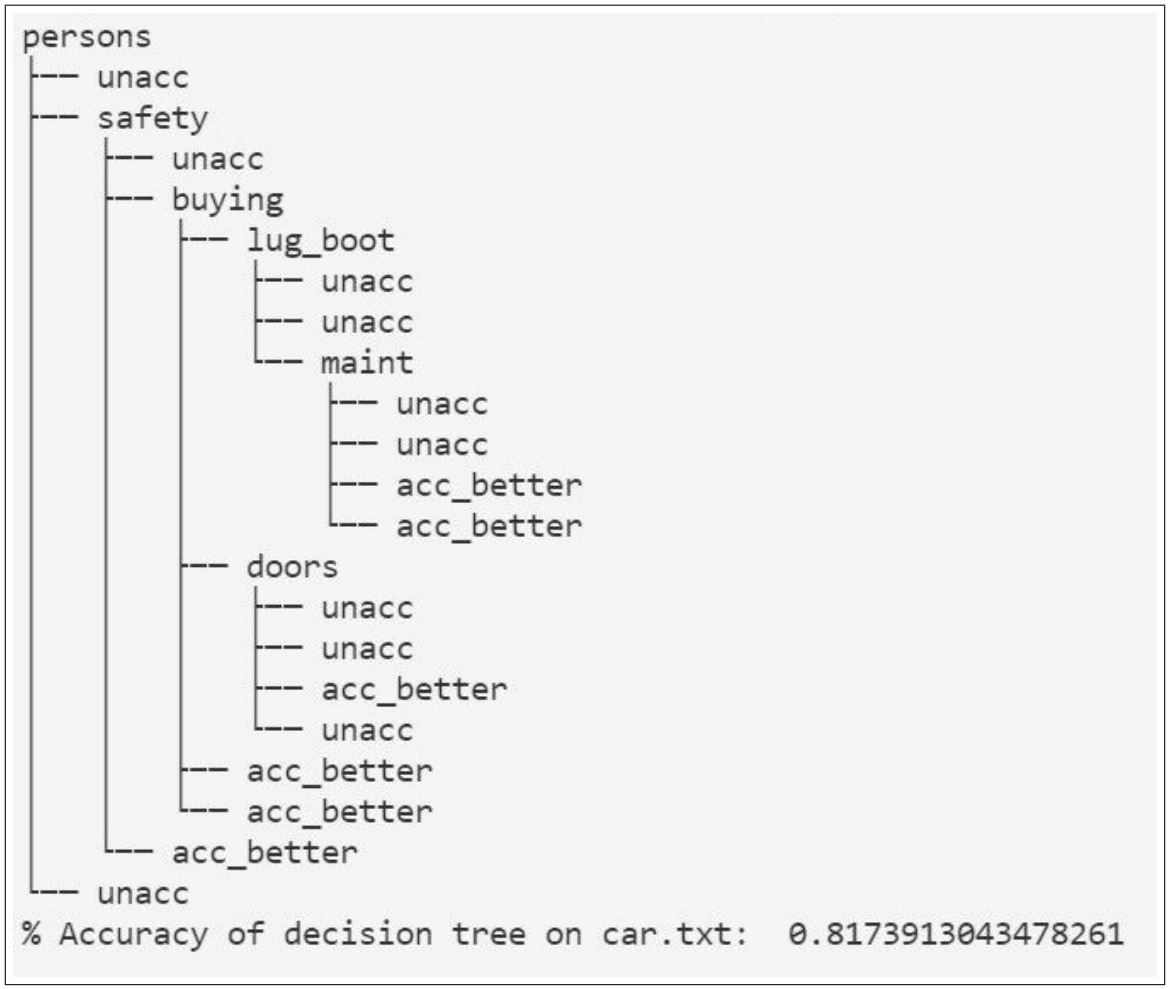

Figure 1: Terminal output displaying the decision tree structure of my program when using the entropy measure splitting function. Note the 'persons' attribute was chosen as the root note, because it had the highest information gain after assignment.

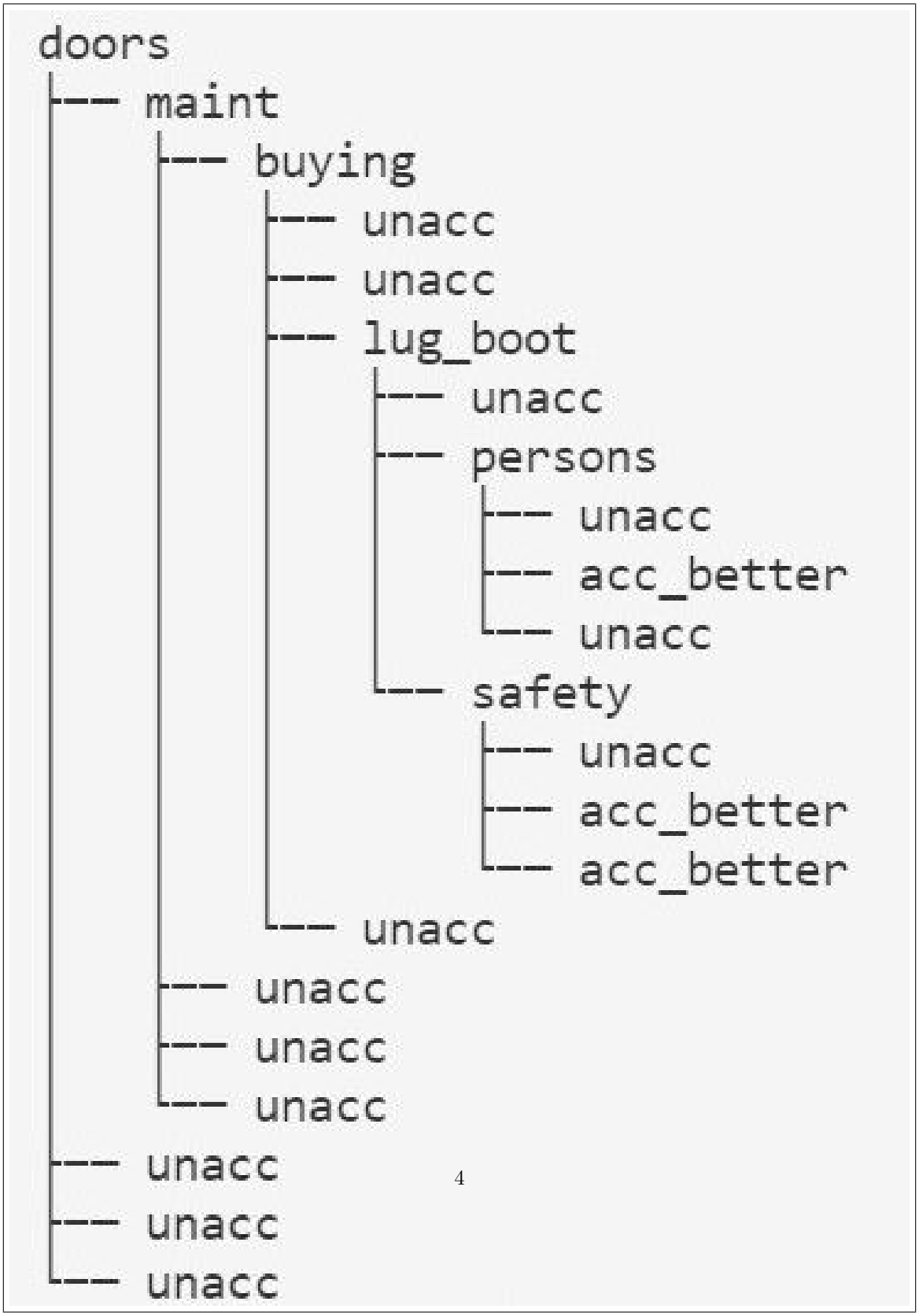

Figure 2: Terminal output displaying the decision tree structure of my program when using the Gini index splitting function. Note the 'doors' attribute was chosen as the root note, because was the 'purest' attribute (it had the lowest Gini index).

```
% Accuracy of decision tree on 5 instances of car.txt:  0.7310144927536232
```

Figure 3: The accuracy of the decision tree from Figure 1 (entropy measure) when classifying unseen data from car.txt.

```
% Accuracy of decision tree on 5 instances of car.txt using Gini Index:  0.6857971014492754
```

Figure 4: The accuracy of the decision tree from Figure 2 (Gini index) when classifying unseen data from car.txt.

2 Part B: Artificial Neural Network Supervised Learner

2.1 Conceptual Overview

An Artificial Neural Network is a data structure which typically consists of layers of forward pointing nodes. Each node accumulates the inputs from the nodes in the previous layer, weights each input, calculates its own activation value, and transfers its activation value to the nodes in the next layer. Thus, by collecting input values in the first layer, a neural network can forward propagate values to come up with a set of output values in the last layer. The purpose of a neural network is to perform supervised classification. A neural network can improve its ability to calculate regression/classification outputs given a set of labeled inputs. The way a neural network improves accuracy is by forward propagating output on a training set of data, calculating the error of its calculation, and backward propagating this error through its layers of nodes, to adjust the weights of the nodes accordingly.

2.2 Project Overview

In this project, I programmed a neural network from scratch in Python 3. I tested on three different data-sets and several different parameters; The parameters I varied were the number of hidden nodes in each hidden layer, the number of hidden layers, and the data set being evaluated. I also had to vary the number of input nodes and output nodes in the neural network depending on the data-set I was evaluating at the time.
I used three different datasets for training and testing my neural network. The first is a dataset called 'seeds_dataset.txt' with 210 input-vector/output-class pairs. Each pair had one of three possible discrete class values, corresponding to seven continuous attributes values. Another data set I used was called 'iris_dataset.txt', which had four continuous attributes and one of three possible classes for each data pair. The last data-set was called 'labeled_examples.txt', which you will recall from Project 1 as having two continuous input attributes and one of two discrete class values for each data pair.
In order to make my neural network classify easier, I normalized the values of the inputs of each of these data-sets to the range used by my activation function (which is the sigmoid function). So I normalized the input values to the range of 0-1. This drastically improved classification accuracy.
I once again employed the randomized train/test split on all datasets, so that the data I trained my neural network on was distinct from the data I tested it on. I gave my network 300 epochs to train on each dataset, which included a full cycle of forward propagation, backward propagation, and weight updates in each epoch.

2.3 Results

Figure 5 compares the accuracy of the neural network when evaluating each of the three datasets (all other parameters being equal). My neural network classified the data in the iris-dataset and the seeds-dataset with over 90% accuracy each time; This is significant because the class types are evenly distributed in each of these data-sets, which means my neural network performs much better than chance on these datasets.
My network is slightly less successful classifying the labeled-examples dataset, coming to about

82.6% classification accuracy. The reason for this may be because of a higher degree of noise in the data. This is still a good degree of accuracy however, as this dataset has two evenly distributed classes, and so the guessing strategy would only result in about 50% accuracy.

Figure 6 compares the accuracy of my neural network when classifying the seeds-dataset for three different cases: Using four hidden nodes, six hidden nodes, and eight hidden nodes (each in only one hidden layer). As you can see, the accuracy increases the more nodes are added to the hidden layer; Starting at 86.6% accuracy for four hidden nodes, the accuracy jumps to 93.3% for six hidden nodes, and finally to 94.28% for eight hidden nodes. This is an predictable result in light of the universal approximation theorem, which states that any continuous function can be approximated by a neural network given there is no limit to the number of nodes in the hidden layer.

Figure 7 shows how my neural network's classification accuracy changes as the number of hidden layers increases. I tested the accuracy of my network on the seeds-dataset using one, three, and five hidden layers. As you can see, the accuracy of my network is good when using only one hidden layer (87.6%), and it improves in the case of three hidden layers to 93.3%. However, I was surprised to see that the classification accuracy dropped significantly in the case of five hidden layers, down to 30.9%, which is comparable to the accuracy of the guessing strategy (because this dataset has three classes). I have two hypothesis for why the accuracy dropped for five hidden layers: Either my neural network needs more time to train because of the additional layers (I only allowed 300 epochs of training), or there is a problem in my algorithm which does not allow for generalization to an arbitrary number of hidden layers. I tested my network on up to 1000 epochs, so I assume the issue is with the generalization of my algorithm. I plan to survey my code in the near future to ascertain why this is the case.

2.4 Reference

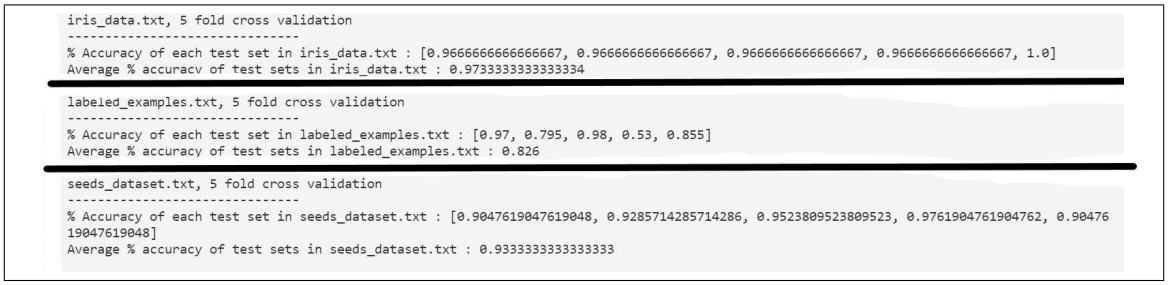

Figure 5: Accuracy of my neural network when classifying data from each of the three datasets.

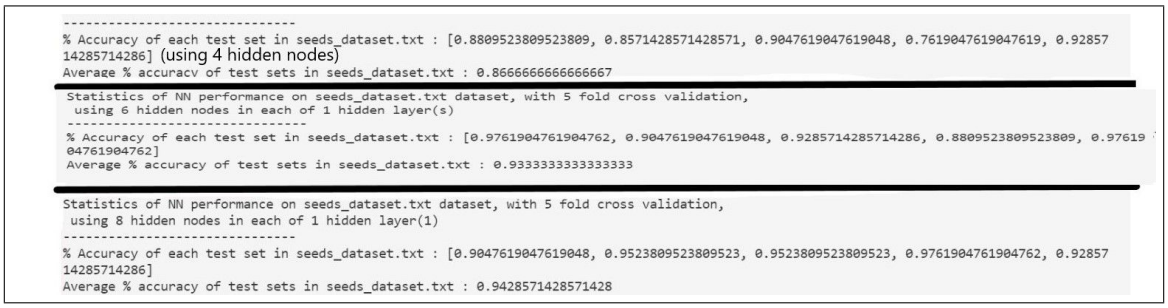

Figure 6: Accuracy of the neural network when using four, six, and eight hidden nodes (on the seeds-dataset).

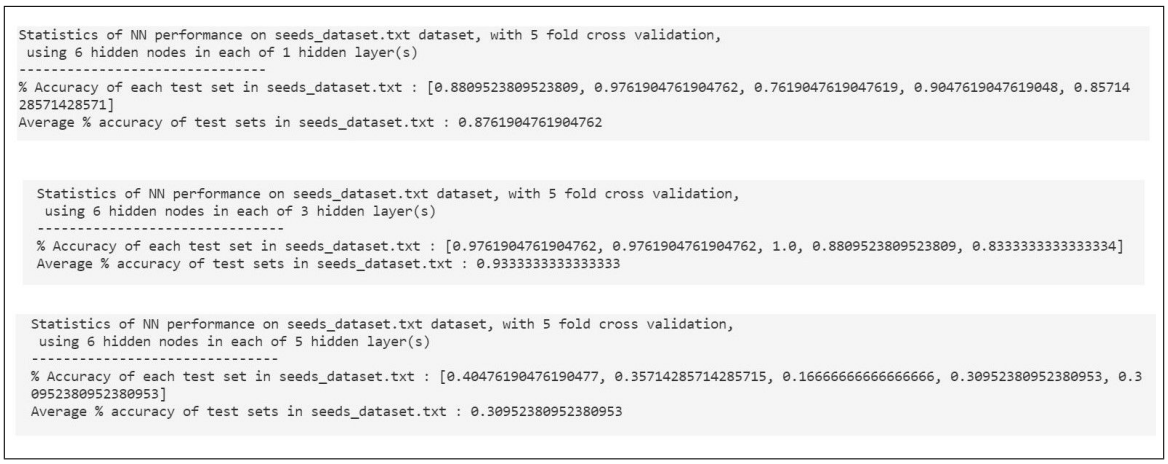

Figure 7: Accuracy of the neural network when using one, three, and five hidden layers (on the seeds-dataset).