Project 1: Decision Tree Implementation

1.1 Greedy learning decision tree algorithm

The decision tree learns by splitting data from one node into two based on the values of a single attribute of the given data at a time, until either a consesus about the class label is reached or the data points are all equal, at which point the algorithm cannot learn more from the data.

Prerequisite:

The algorithm assumes that all data point values are floating point numbers or integers, and the class labels catagorical variables, strings.

Steps:

1. Receive node, with all data values and class label list.
2. If all class labels are equal, that label is chosen, and the node is a leaf, which means no more steps are taken for this node.
   Else if all data points have the same values in each column, the most common label is chosen, and the node is a leaf.
   Else continue the algorithm with this node.
3. Choose the attribute, or column, with highest information gain available.
4. Split the rows of data and the class labels on the average value in the chosen column.
5. Run the algorithm recursively with both new nodes.

Our implementation.

We use the dataclasses module for simpler class definitions. The algorithm is split into three dataclasses, `Tree`, `Tode` and `Data`. The class `Data` is a simple class containing lists of the data split into both test and train data and further into feature values and class labels. `Node` contains all the information needed to find out if the node is a leaf or if it should be split further, as well as information needed later when building the tree and predicting the class label of new data points. For example it's child nodes, if any. `Tree` contains all the logic for building the tree, aka training the algorithm. Also included is the prediction and pruning functions, more on those later.

Impurity measure.

Information gain is calculated using the change in entropy from before splitting to after the split, for each attribute, before deciding which column to split on. Entropy is a measure of uncertainty, is this case with two possible labels, 1 means that the labels are split 50-50 and 0 means that all class labels are the same. The impurity of each node is lowered as much as possible, which, most often, leads to overfitting. Meaning the algorithm is trained too well on the training data and will therefore perform worse on the test data. The entropy also relies on the probability of any label being chosen among the class labels, so a function to calculate this probability is included. The label you find the probability for does not matter, since, again, there are only two possible labels and the entropy uses the probability of both of them.

1.2 Gini index

Added alternative impurity measure, the gini index, as a setting for choosing the best attribute to split the data on. Implemented a function to calculate the gini index which takes the sum of the square of both the probabilities (for each class label) and subtracts them from 1. Gini index is chosen as impurity measure by setting the `impurity_measure` parameter in `learn()` to `"gini"`. See tree.py in the zipped code, and the README.md for more information on how to use the algorithm.

1.3 Reduced-error pruning implementation

Pruning added as an extra setting in the `learn()` function described earlier. Pruning uses a seperate chunk of the data for comparison. After the tree is built, or trained, the pruning feature compares the accuracy of a subtree (node) to the pruning data. Then it checks if replacing the node with a leaf, using the most common label in the node as the chosen class label, does not reduce the accuracy on the pruning data. If the accuracy is equal, or better, the node is permanently changed to a leaf, retaining the label chosen earlier. The reason for using pruning is to reduce how well the model fits the training data, so it can make better predictions on other data. Pruning can be enabled by setting the `prune` parameter to True.

1.4 Evaluation

Refer to the main.py file in the code.
We use the pandas library to read the data, then we made a split_data() method which, given a split ratio, splits the data into training and testing data and returns it as an instance of the `Data` class (Reference main and 1.1). The data consists of 10 columns of feature values and a class label, either "g" or "h", meaning gamma and hadron, respectively. The feature values are represented as numbers, distributed over an interval, with very few reappearing values.
For training the algorithm, it performs better the greater the amount of training data used, up to a certain point. The split ratio given represents the size of the portion of training data. The time it takes to train the algorithm also goes up logarithmically for a split ratio going up to about 0.25, from which the time changes very little, given that pruning is being applied.
The performance of the algorithm varies a little between runs, but it usually lies around 80 percent, with pruning taking it closer to 85. It was hard to find any significant difference between choosing entropy and gini index as the impurity measure, we were at least not able to document anything conclusive. Although it seems like the gini index is the better option. The performance is evaluated by comparing a prediction (`predict()` , tree.py) on unseen data to the actual class label of that data.

1.5 Comparison

We chose to compare the algorithm to Scikit Learn's decision tree classifier. Our implementation consistently performs better in accuracy than sklearn's algorithm. What it lacks in accuracy it makes up for in speed, it is significantly faster than our implementation.