Project #1

Implementing decision trees

1.1 Implement a decision tree learning algorithm from scratch

The code contains three functions that are used when building the tree

- learn()
- best_split()
- impurity()

learn() is the main function, where the greedy ID3 algorithm runs recursively. When choosing where to do a split, best_split() is called to iterate through the available features and calculate information gain. To calculate information gain, best_split() calls impurity() which calculates the impurity of the given datasets. The impurity is then used in the calculation for information gain.

For simplicity reasons, the information gain is always calculated based on a split on the average value of the feature. The feature with the highest information gain is then selected for the final split. This approach does not guarantee the best split, but it makes for efficient coding. To find the absolute best split, one would have to calculate information gain for all possible splits across all features, for every split.

When the best split is selected, best_split() returns a dictionary of information to learn(), where it is embedded into a node class. As learn() runs recursively, the tree is built from the root node.

1.2 Add Gini index

The Gini index is simply another impurity measure and can be used instead of entropy. To calculate information gain with Gini, call learn() with the parameter *impurity_measure="gini"*. This is passed onto impurity() where we find this snippet of code:

```
if impurity_measure == "entropy":
    #Calculate and return entropy
    return -np.multiply(ProbX, np.log2(ProbX)).sum()

elif impurity_measure == "gini":
    #Calculate and return Gini index
    return np.multiply(ProbX, 1-ProbX).sum()
```

The *ProbX* variable is an array of the probabilities for each unique value in the array X.

1.3 Add reduced-error pruning

To include pruning in the algorithm, simply set the variable *pruning = True*. This will split the training data into training and pruning data before building the tree. When building the tree, the pruning tags along with the training data and is split with the same criteria. When creating a node, the code stores information in the node about which labels from the pruning data ended up in the node. It will also store the majority label for the node. With this approach it is easy to calculate the accuracy for a majority label in a node and calculate the accuracy for the two child nodes. If the accuracy for the majority label is equal or greater to the combined accuracy for the two child nodes, we prune.

To make sure that we start pruning at the bottom and then work up, we traverse the tree and only prune if we can find a node which has two leaf nodes as children. We then prune iteratively until we cannot make more changes.

1.4 Evaluate your algorithm

We will now use validation data to select the setting for the final model; entropy or gini, pruning or no pruning. The table below shows the results from testing. The seed used is 123.

Impurity measure	Entropy		Gini	
Pruning	No	Yes	No	Yes
Training accuracy	100%	94.6%	100%	94.8%
Validation accuracy	80.0%	81.7%	78.7%	81.7%

From the results, we see that entropy does better than Gini when not pruning but pruning results in the exact same validation accuracy for both impurity measures. We then select entropy with pruning as our model and test this on the unseen test data.

The test data yields an accuracy of 82.0%.

The accuracy of the test data should reflect the expected accuracy on unseen data. However, in this case we had an increase in accuracy, which is surprising as validation data often underestimates errors.

1.5 Compare to an existing implementation

We compare the code to the tools from the sklearn library. The code beneath is all you need to build a tree and test its accuracy.

```
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X_train, Y_train)
score = clf.score(X_test, Y_test)
```

While it is much faster at 0.5 seconds compared to the original code which takes 6.3 seconds to execute, it has the approximately same test accuracy at 82.5%. This, however, is without pruning or any sort of hyperparameter like max depth. With some tuning I expect the sklearn decision tree to outperform my own tree.

That being said, with some extra lines of code I would be able to implement a max depth for the tree and a best_split() function that can find a better split than the average.