Project 1 – Implementing Decision Trees

Ilan Valencius

To make my explanation more concise, the inputs for functions will be excluded unless they are needed. Attached are the printed out docstrings describing all functions in *decision_tree.py* if there is any confusion on inputs/what anything does.

Training

The core structure in my model is a *Node(),* implemented in *structures.py*. A node can either be a 'node' or a 'leaf', a condition defined when a node is given a class label rather than a condition. To learn a full decision tree, a root node is passed to the *learn()* function. This function converts the given impurity measure to a corresponding *int* as well as creates a prune dataset if 'pruning' is set to true. The root node is then passed to *learn_node(),* then pruned if necessary. To keep most of the training functions in one file, functions such as *learn_node()* were not implemented as part of the *Node()* class but could easily be to eliminate the need to pass the root node as a parameter.

In *learn_node(),* the first two cases are tested to determine if it should become a leaf. A node will become a leaf if all the class labels are the same or if all feature values are the same. If that is the case the node is made into a leaf via *node.make_leaf()* and returned. If not, the node is trained. For every feature, the data is split via the mean of the feature – *split_by_mean(X, Y, mean, col=c).* Based on this split, the resultant entropy or gini index is returned depending on the metric parameter specified in *determine_metric()*. If the metric is entropy, information gain is determined by the *entropy()* function which has an input 'conditional=True/False' in order to do entropy – conditional entropy. If the conditional flag is set to False, the proportion of every class is determined and then summed appropriately xn/N * log2(xn/N). If conditional is True, the entropy of each branch of data is determined unconditionally then summed (lines 51-60). The *gini()* function doesn't need to take in multiple branches of data, it just takes in one array of labels, 'Y'. For each class in gini, the proportion is squared then summed and subtracted from one. The resultant gini indexes of each branch are then added, using weights, and then, subtracted from the gini index of the node pre-split (line 103).

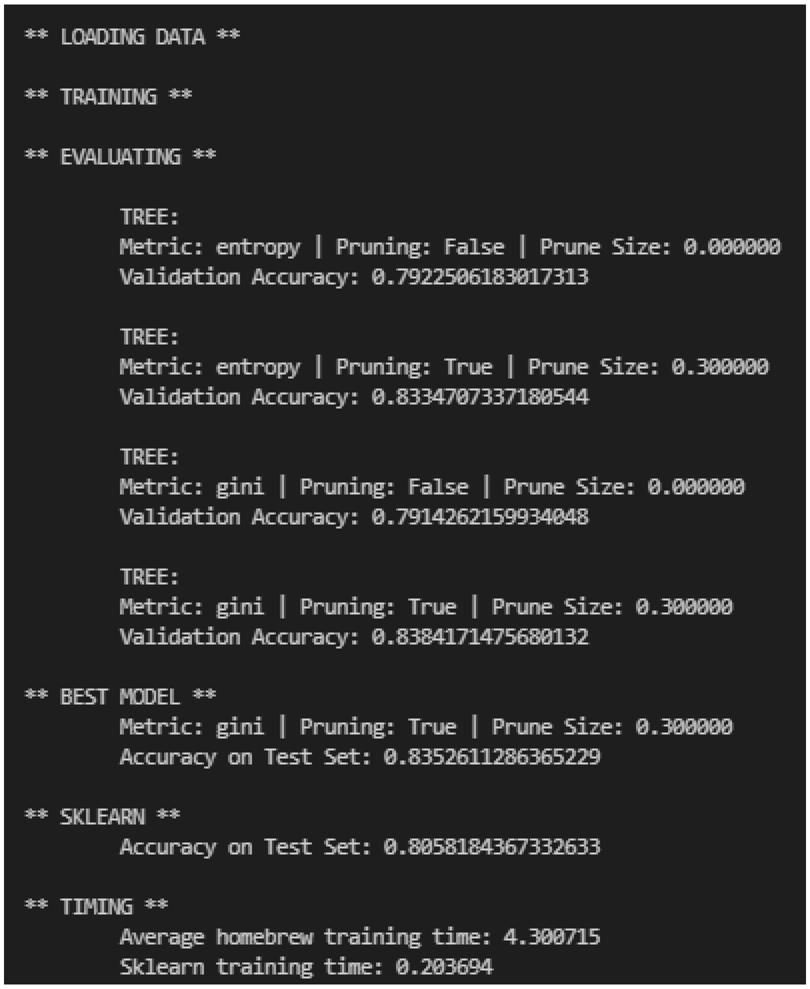
Learn_node() then proceeds to determine the feature index with the maximum information gain and then adds this, and the associated mean that was used to split the feature, to the given node via *node.add_condition(feature_idx, Means[feature_idx]).* Like splitting the data, the node classifies a feature to the right if *x[self.feature_idx] >= self.mean*. The training data for the node is then split based on the selected feature and the root nodes children are trained recursively. The node is then returned.

Pruning

If a node is a leaf, or no pruning data ends up at a node, it is not pruned and returned as is. If a node is pruned, the training data and prune data are split based upon the nodes condition and each child of the node is trained with their respective data before training the root node. This allows for a leaf-up reduced-error-pruning approach. As outlined in lines 236-245, the error classification rate of the node is determined with 'no split' (node becomes a leaf with majority label), or 'with split' (node remains as is) using the pruning data. If the error 'no_split' is less than the error 'with_split', all data in the node is cleared and it is given the majority label, becoming a leaf. The node is then returned.

Performance Assessment (sample output attached)

The performance of every parameter combination is then tested with a validation size 0.15, test size 0.15, and prune size of 0.3 if the 'pruning' parameter is set to true (all inside *main()*, lines 371 on). Once every model was trained, the accuracy was determined on the validation set using the *get_acc()* function. The model with the best accuracy on this validation set was then determined to be the optimal model and the accuracy of the selected model was determined on the test set. This model was generally entropy with pruning (although gini with pruning is extremely close). Given the large dataset, both metrics perform well, but pruning definitely improves accuracy on the validation and test set given the ability of a pruned model to generalize better. Compared to the sklearn model, my implementation produces a model that is roughly 3% better. The sklearn model is still very competitive (beating my non-pruned models), despite the lack of pruning. This is because of the ability of sklearn to split based on values *other* than the mean of the feature. The slightly worse accuracy, most likely caused by a lack of ability to prune a DecisionTreeClassifier() tree, is made up for in speed. The sklearn model is trained in about 0.2 seconds while mine takes a little over 4. This is the result of speed on the part of the sklearn package, utilizing multithreading for instance. My slower model train time could also be the result of efficiency errors on my part. Processes such as *np.unique(_, return_counts=True),* are time intensive and used quite often. Pruning is also time intensive, as every node must be trained and then pruned.



```
** LOADING DATA **

** TRAINING **

** EVALUATING **

        TREE:
        Metric: entropy | Pruning: False | Prune Size: 0.000000
        Validation Accuracy: 0.7922506183017313

        TREE:
        Metric: entropy | Pruning: True | Prune Size: 0.300000
        Validation Accuracy: 0.8334707337180544

        TREE:
        Metric: gini | Pruning: False | Prune Size: 0.000000
        Validation Accuracy: 0.7914262159934048

        TREE:
        Metric: gini | Pruning: True | Prune Size: 0.300000
        Validation Accuracy: 0.8384171475680132

** BEST MODEL **
        Metric: gini | Pruning: True | Prune Size: 0.300000
        Accuracy on Test Set: 0.8352611286365229

** SKLEARN **
        Accuracy on Test Set: 0.8058184367332633

** TIMING **
        Average homebrew training time: 4.300715
        Sklearn training time: 0.203694
```

decision_tree

Modules

numpy   sklearn.tree

Functions

determine_metric(Y, Y_split1, Y_split2, metric)
```
Determines the value of the determined metric on a node

Args:
    Y (ndarray): full data labels
    Y_split1 (ndarray): one subset of Y
    Y_split2 (ndarray): other subset of Y
    metric (int): determines which metric to use

Returns:
    float: information gain or gini index of a node
```

entropy(Y, Y_split1, Y_split2, conditional)
```
Determines the entropy

Args:
    Y (ndarray): Array before split
    Y_split1 (ndarray): subset of Y
    Y_split2 (ndarray):  other subset of Y
    conditional (bool): if entropy is determined conditionally

Returns:
    float: entropy
```

get_acc(node, X, Y)

gini(Y)
```
Determines the gini index of an array Y

Args:
    Y (ndarray): array of labels

Returns:
    float: gini_index
```

learn(root, X, Y, impurity_measure, pruning=False, prune_sz=0, seed=None)
```
Trains a root node

Args:
    root (Node()): root node to be trained
    X (ndarray): training features
    Y (ndarray): training labels
    impurity_measure (int): determines impurity measure
    pruning (bool, optional): determines whether to prune. Defaults to False.
    prune_sz (int, optional): size of prune dataset. Defaults to 0.
    seed (int, optional): sets seed for splitting prune data. Defaults to None.

Returns:
    [type]: [description]
```

learn_Node(node, X, Y, metric)
```
Learns a node based on X, Y, metric

Args:
    node (Node()): node to be
    X (ndarray): feature of
    Y (ndarray): labels
```

```
Learns a node based on X, Y, metric

Args:
    node (Node()): node to be
    X (ndarray): feature of
    Y (ndarray): labels
    metric (int): determines gini or entropy

Returns:
    Node(): node with children learned as well
```

load_magic(filename)

```
Loads data from MAGIC Gamma Telescope

Args:
    filename (file object): file to parse

Returns:
    (ndarray, ndarray): X, Y s.t. X is features and Y is labels
```

log2(x, /)

```
Return the base 2 logarithm of x.
```

main()

```
Organizes function calls for training and evaluating decision tree models
```

prune(node, X_prune, Y_prune, X_train, Y_train)

```
Prunes a Node() node

Args:
    node (Node()): node to be pruned
    X_prune (ndarray): pruning features
    Y_prune (ndarray): pruning labels
    X_train (ndarray): training features
    Y_train (ndarray): training labels

Returns:
    Node(): pruned node with children pruned
```

split(node, X, Y)

```
Splits X, Y, based on condition of a node

Args:
    node (Node()): determines how data is split
    X (ndarray): feature values
    Y (ndarray): labels

Returns:
    (ndarray, ndarray, ndarray, ndarray): X1, Y1, X2, Y2 s.t. X1, Y1 classified as true under condition of node
```

split_by_mean(X, Y, mean, col)

```
Splits X and Y

Args:
    X (ndarray): features
    Y (ndarray): labels
    mean (float): value to split on
    col (int): column of feature which is split

Returns:
    (ndarray, ndarray, ndarray, ndarray): X1, X2, Y1, Y2 s.t for all x in X1, Y1, x[col] >= mean
```

timer = perf_counter(...)

```
perf_counter() -> float

Performance counter for benchmarking.
```