Inf283_oblig1

Nikolai Grieg

September 2018

Building the Decision Tree

A tree structure was used to make the decision tree. The anytree[1] library was used for this. The library contains basic tree functionality, such as is_leaf and get_children methods. Anytree also contained a convenient way to print the decision tree in console. The console output didn't play well with latex, but it will be printed when running the code.

Calculating Information Gain

The explanation here will use H for entropy. The calculation is done in three steps recursively. Information Gain using entropy is defined as:

$$\Delta H = H - \sum_k \frac{m_k}{m} H_k$$

Similarly for gini index by replacing H with G:

$$\Delta G = G - \sum_k \frac{m_k}{m} G_k$$

Where m_k is the instances of class k, and m is the amount of considered instances. H and G are defined as following:

$$H = -\sum_{i=1}^{K} p_k log_2 p_k$$

$$G = 1 - \sum_{i=1}^{K} p_k^2$$

Where p_k is the probability of element k.

First, the method calculate_impurity_base() calculates H. Second, calculate_impurity_dict() calculates each H_k, which is stored as value in a dict with key=class name. ΔH is then calculated in information_gain(), using the impurity dict and H the following way:

[1] https://anytree.readthedocs.io/en/latest/

```python
def information_gain(impurity_base, df, impurity_measure):
    impurity_dict = calculate_impurity_dict(df, impurity_measure)
    IG = impurity_base
    for key, value in impurity_dict.items():
        count = df.iloc[:, 1].tolist().count(key)
        total = len(df.iloc[:, 1].tolist())
        intermediate = count / total * value
        IG -= intermediate
    return IG
```

In the method recurse(), ΔH is calculated for each column in data set. The best column is then chosen, and appended to the tree as a decision_node. All the classes in the column are then appended do the decision_node as leaves with type decision_edge at the current juncture in the recursion. Then recurse() is called for each of the decision_edge nodes, repeating until all leaves are categories.

The data structure

The tree structure is composed of 4 types of nodes: root, decision_state, decision_edge, and class. The decision_edge node type is way to implement edges that hold variables, and can be though of as edges. The downwards traversal pattern will always go root - decision_state - decision_edge - class/decision_state repeating. The decision edges are such a connection between decision_state nodes and class nodes, or the next level of decision_state node. If print_tree() is run with show_variables=True, the node types will be printed with the tree.

Pruning the tree

The pruning process starts by splitting the training set into train and prune set. The prune set size is chosen as 0.2 here. Then the errors are calculated on the nodes in the tree. This is done by traversing upwards from the predicted leaf node, and assigning +1 error to each node where majority label did not match the label of the data point. The pruning in the prune() method then follows the algorithm as given by the pseudocode in Assignment 3 (slightly modified):

```
Q = queue
T = tree
put each leaf in Q
while Q is not empty:
    node = Q.next
    if node is not root: # root will always be last, and has no parents
        parent = node.parent
        if parent is not in Q
            add parent to Q
        if node is not leaf
            M = majority for node
```

```
            E = errors from prune set on node
            R = sum of errors on descendant nodes
            if R >= E
                replace node in T by leaf with category M

return T
```

Classification of Mushrooms

The data is read into a dataframe with pandas as shown in Environment.py. All rows containing one '?' element is dropped, which reduces the size of the dataset by about 30% (number of rows from 8416 to 5936). Alternative methods could have been used to solve the undefined entries, but this was the prescribed method by the assignment. Assuming the unidentifiable elements were randomly distributed, this would be fine, as the dataset is still fairly large after dropping the rows. If there was a pattern to the unidentifiable elements, some information could potentially be lost this way.

The data is then split into train and test sets, by random sampling, with size of the test set being 0.2. This value is chosen in order to not get too low training set size when pruning, as 20% of the remaining training set is used for this. This could be remedied by using cross validation when doing pruning. Random states (all of them set to 1) are used throughout the implementation in order to have predictable results for debugging and reproducibility.

Results

The implementation reaches 100% accuracy on the test set (with random_state=1), both with and without pruning, and with both impurity measurements. With this train_test_split, the tree is the same regardless of pruning. The same accuracy is achieved on the same data split with sklearns DecisionTreeClassifier().

Potential improvements

Cross validation for the pruning could potentially improve the pruning process. Encoding of the categorical variables could also improve the algorithms efficiency (reduced time) for larger datasets.