sklearn_feature_importance_solution

April 24, 2023

1 Feature importance method in sci-kit learn (Solution)

We'll get a sense of how feature importance is calculated in sci-kit learn, and also see where it gives results that we wouldn't expect.

Sci-kit learn uses gini impurity to calculate a measure of impurity for each node. Gini impurity, like entropy is a way to measure how "disorganized" the observations are before and after splitting them using a feature. So there is an impurity measure for each node.

In the formula, freq_{i} is the frequency of label "i". C is the number of unique labels at that node.

$Impurity = \sum_{i=1}^{C} -freq_i * (1 - freq_i)$

The node importance in sci-kit learn is calculated as the difference between the gini impurity of the node and the gini impurity of its left and right children. These gini impurities are weighted by the number of data points that reach each node.

$NodeImportance = w_i Impurity_i - (w_{left} Impurity_{left} + w_{right} Impurity_{right})$

The importance of a feature is the importance of the node that it was split on, divided by the sum of all node importances in the tree. You'll get to practice this in the coding exercise coming up next!

For additional reading, please check out this blog post The Mathematics of Decision Trees, Random Forest and Feature Importance in Scikit-learn and Spark

```
In [ ]: import sys
        !{sys.executable} -m pip install numpy==1.14.5
        !{sys.executable} -m pip install scikit-learn==0.19.1
        !{sys.executable} -m pip install graphviz==0.9
```

```
In [ ]: import sklearn
        from sklearn import tree
        import numpy as np
        import graphviz
```

1.1 Generate data

We'll generate features and labels that form the "AND" operator. So when feature 0 and feature 1 are both 1, then the label is 1, else the label is 0. The third feature, feature 2, won't have an effect on the label output (it's always zero).

```
In [ ]: """
        Features 0 and 1 form the AND operator
        Feature 2 is always zero.
        """
        N = 100
        M = 3
        X = np.zeros((N,M))
        X.shape
        y = np.zeros(N)
        X[:1 * N//4, 1] = 1
        X[:N//2, 0] = 1
        X[N//2:3 * N//4, 1] = 1
        y[:1 * N//4] = 1

In [ ]: # observe the features
        X

In [ ]: # observe the labels
        y
```

1.2 Train a decision tree

```
In [ ]: model = tree.DecisionTreeClassifier(random_state=0)
        model.fit(X, y)
```

1.3 Visualize the trained decision tree

```
In [ ]: dot_data = sklearn.tree.export_graphviz(model, out_file=None, filled=True, rounded=True,
        graph = graphviz.Source(dot_data)
        graph
```

1.4 Explore the tree

The source code for Tree has useful comments about attributes in the Tree class. Search for the code that says `cdef class Tree:` for useful comments.

```
In [ ]: # get the Tree object
        tree0 = model.tree_
```

1.5 Tree attributes are stored in lists

The tree data are stored in lists. Each node is also assigned an integer 0,1,2...
Each node's value for some attribute is stored at the index location that equals the node's assigned integer.

For example, node 0 is the root node at the top of the tree. There is a list called children_left. Index location 0 contains the left child of node 0.

left and right child nodes

```
children_left : array of int, shape [node_count]
        children_left[i] holds the node id of the left child of node i.
        For leaves, children_left[i] == TREE_LEAF. Otherwise,
        children_left[i] > i. This child handles the case where
        X[:, feature[i]] <= threshold[i].
children_right : array of int, shape [node_count]
        children_right[i] holds the node id of the right child of node i.
        For leaves, children_right[i] == TREE_LEAF. Otherwise,
        children_right[i] > i. This child handles the case where
        X[:, feature[i]] > threshold[i].
```

```
In [ ]: print(f"tree0.children_left: {tree0.children_left}")
        print(f"tree0.children_right: {tree0.children_right}")
```

So in this tree, the index positions 0,1,2,3,4 are the numbers for identifying each node in the tree. Node 0 is the root node. Node 1 and 2 are the left and right child of the root node. So in the list children_left, at index 0, we see 1, and for children_right list, at index 0, we see 2.

-1 is used to denote that there is no child for that node. Node 1 has no left or right child, so in the children_left list, at index 1, we see -1. Similarly, in children_right, at index 1, the value is also -1.

features used for splitting at each node

```
    feature : array of int, shape [node_count]
        feature[i] holds the feature to split on, for the internal node i.
```

```
In [ ]: print(f"tree0.feature: {tree0.feature}")
```

The feature 1 is used to split on node 0. Feature 0 is used to split on node 2. The -2 values indicate that these are leaf nodes (no features are used for splitting at those nodes).

number of samples in each node

```
n_node_samples : array of int, shape [node_count]
        n_node_samples[i] holds the number of training samples reaching node i.

weighted_n_node_samples : array of int, shape [node_count]
        weighted_n_node_samples[i] holds the weighted number of training samples
        reaching node i.
```

```
In [ ]: print(f"tree0.n_node_samples : {tree0.n_node_samples}")
        print(f"tree0.weighted_n_node_samples : {tree0.weighted_n_node_samples}")
```

The weighted_n_node_samples is the same as n_node_samples for decision trees. It's different for random forests where a sub-sample of data points is used in each tree. We'll use weighted_n_node_samples in the code below, but either one works when we're calculating the proportion of samples in a left or right child node relative to their parent node.