Assignment 1 - Decision Tree Classifiers

Overview

This decision tree classifier is implemented as an extension of the anytree (https://anytree.readthedocs.io/en/latest/index.html) module. By inheriting from the any tree node class, the tree structure is easily maintained and all custom code is simply added to the class.

Data manipulation is performed using pandas (https://pandas.pydata.org/), which has many low-level functions built in to perform functions not directly related to the Decision Tree Classifier algorithms that are the focus of the assignment.

Part 1: Splitting Algorithm

The splitting algorithm is implemented as follows

```
if stop criterion have not been met:
    compute the optimal split among all features and all split values of each featu
re
    split the samples into two subsamples based on this optimal split
    for each sample subset, apply the splitting algorithm
```

The stopping criterion, as specified, are that either the sample set is pure or all features of the set are identical. Pandas was used to quickly perform both of these checks. Purity was checked by looking at the number of unique labels within the set. Feature similarity was checked by ensuring that more than one sample still existed when duplicate features were removed

Part 2: Pruning Algorithm

The pruning algorithm is implemented as follows:

```
Set the best observed accuracy to 0
At the root node:
    Make a list of all descendants of the node
    For each descendant:
        temporarily remove the node and its descendants
        calculate the accuracy of the new tree
        Add the node back into the tree
        if the new accuracy is greater than the current best accuracy:
            update best accuracy
            track the node that whose removal gave this accuracy
    Remove the best node (and its descendants) that gives the best resultant accura
cy
```

Anytree makes this process very simple, since many of these steps simply involve changing a node's parent temporarily and then iterating over the new tree

Part 3: Data Analysis

In [1]:
```
#import pandas for dataset manipulation, anytree for tree visualization, and s
upporting modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os as os
import anytree as at
import anytree.exporter as texporter
```

In [11]:

```
class DTCNode(at.Node):
    def __init__(self, name, samples, features,impurityType, resultVal, parent
=None):
        self.name = name
        self.totalNodes = 1
        self.totalLeafNodes = 0
        self.parent = parent
        self.splitFeature = None
        self.splitValue = np.nan
        self.samples = samples
        self.trueSamples = None
        self.falseSamples = None
        self.features = features
        self.resultVal = resultVal
        self.label = self.computeLabel()
        self.impurityType = impurityType
        self.impurity = self.computeImpurity(self.samples)
        self.root.totalNodes += 1


        if not self.stopCriterionIsMet():
            [self.splitFeature, self.splitValue, self.trueSamples, self.falseS
amples, isLeaf] = self.computeOptimalSplit()

            if isLeaf:
                #print("I'm a leaf node!")
                #print("Leaf Node Samples", self.samples)
                self.root.totalLeafNodes += 1
                self.name = self.label
            else:
                self.name = self.name + "\n" + self.splitFeature + " <= " + st
r(self.splitValue)
                #print("Splitting samples into child nodes!")
                #if not self.parent == None:
                #    print("Parent Samples\n",self.parent.samples)
                #    print("Parent Samples Value Count\n", self.parent.samples
[self.parent.samples.columns[0]].value_counts())
                #print("Split Feature\n",self.splitFeature)
                #print("Split Value\n",self.splitValue)
                #print("True Samples\n", self.trueSamples)
                #print("False Samples\n",self.falseSamples)
                DTCNode("True", self.trueSamples, self.features, self.impurity
Type, True, parent=self)
                DTCNode("False", self.falseSamples, self.features, self.impuri
tyType, False, parent=self)
        else:
            #print("I'm a leaf node!")
            #print("Leaf Node Samples", self.samples)
            self.root.totalLeafNodes += 1
            self.name = self.label
        if parent == None:
            texporter.DotExporter(self).to_picture('TestTree.png')

    def stopCriterionIsMet(self):
        return self.samplesArePure(self.samples) or self.sampleFeaturesAreIden
tical(self.samples)
```

```
    # samples are pure if all the labels are the same
    # use pandas built-ins to do a bit of the heavy lifting
    # by counting the number of unique values within the label column,
    # if it is pure, there will only be a single unique value
    def samplesArePure(self, samples):
        pure = samples[samples.columns[0]].nunique() == 1
        return pure

    # again use pandas built-ins for heavy lifting
    # by removing all rows with duplicate feature values.
    # If all features are identical for all samples,
    # only one row will be remaining
    def sampleFeaturesAreIdentical(self, samples):
        identical = len(samples.drop_duplicates(self.features).index) == 1
        return identical

    # iterate through all features(and possible splits within those features)
    # in the node's samples and find the feature/split combo that minimizes th
e resultant
    # child node's weighted impurity.
    def computeOptimalSplit(self):
        bestImpurity = 1
        bestFeature = None
        bestValue = None
        bestTrueSubSample = None
        bestFalseSubSample = None
        #iterate through each feature
        for feature in self.features:
            #create an ordered list of values
            orderedValueList = self.samples[feature].sort_values(ascending=Tru
e).drop_duplicates()
            #create array of potential split values by averaging adjacent valu
es in the list
            splitVals = []
            orderedValueList=orderedValueList.values
            for i in range(1,orderedValueList.size-1):

                #print(i)
                #print(orderedValueList[i-1])
                #print(orderedValueList[i])
                splitVals.append((orderedValueList[i]+orderedValueList[i-1])/2
)
            for value in splitVals:
                [tempTrueSamples, tempFalseSamples] = self.split(feature, valu
e)
                tempImpurity = self.computeWeightedImpurity(self.samples,
                                                            tempTrueSamples,
                                                            tempFalseSamples)
                if tempImpurity < bestImpurity:
                    #print("Found a better split criterion!")
                    #print(bestFeature, bestValue)
                    bestImpurity = tempImpurity
                    bestFeature = feature;
                    bestValue = value
                    bestTrueSubSample = tempTrueSamples
                    bestFalseSubSample = tempFalseSamples
```

```
                    #print(bestTrueSubSample)
                    #print(bestFalseSubSample)
        return [bestFeature, bestValue, bestTrueSubSample, bestFalseSubSample,
 bestFeature==None]

    # returns two subsamples by splitting a node's samples based on a specifie
d
    # feature and value
    # Uses pandas for heavy lifting
    def split(self, feature, value):
        # pandas supports boolean indexing, which makes this pretty trivial
        trueSubSamples = self.samples[self.samples[feature] <= value]
        falseSubSamples = self.samples[self.samples[feature] > value]
        return [trueSubSamples, falseSubSamples]

    def computeImpurity(self, samples):
        if self.impurityType == "Gini":
            return self.giniImpurity(samples)
        elif self.impurityType == "Entropy":
            return self.entropyImpurity(samples)
        else:
            raise ValueError("A valid impurity measure type was not specifie
d!")

    # calculates the weighted impurity of a node's children
    # this function takes many shortcuts due to the assumptions
    # that it will only be computing the weighted impurity for
    # a group of two nodes (the two children of a parent node)
    def computeWeightedImpurity(self, parentSamples, subSamples1, subSamples2
):
        totalSamples = len(parentSamples.index)
        totalSubSamples1 = len(subSamples1.index)
        totalSubSamples2 = len(subSamples2.index)
        return ((totalSubSamples1/totalSamples)*self.computeImpurity(subSample
s1) +
                (totalSubSamples2/totalSamples)*self.computeImpurity(subSample
s2))

    # calculate the gini index of a set of samples
    # this function takes many shortcuts due to the assumption
    # that it will only be computing the gini index of binary data
    def giniImpurity(self, samples):
        #check if the sample set is pure
        if self.samplesArePure(samples):
            return 0
        #print(samples)
        value_counts = samples[samples.columns[0]].value_counts()
        labelOneCounts = value_counts[0]
        labelTwoCounts = value_counts[1]
        totalCounts = labelOneCounts + labelTwoCounts
        return 1 - np.square(labelOneCounts/totalCounts) - np.square(labelTwoC
ounts/totalCounts)



    # calculate the entropy of a set of samples
    # this function takes many shortcuts due to the assumption
    # that it will only be computing the gini index of binary data
```

```
    def entropyImpurity(self, samples):
        #check if the sample set is pure
        if self.samplesArePure(samples):
            return 0
        value_counts = samples[samples.columns[0]].value_counts()
        labelOneCounts = value_counts[0]
        labelTwoCounts = value_counts[1]
        totalCounts = labelOneCounts + labelTwoCounts
        return (-(labelOneCounts/totalCounts)*np.log2(labelOneCounts/totalCoun
ts) -
                (labelTwoCounts/totalCounts)*np.log2(labelTwoCounts/totalCount
s))

    def computeLabel(self):
        # assign a label to the node that is the label
        # with the largest frequency within the nodes samples
        # value_counts returns frequency counts in descending order
        # by default, so grab the index of the first one
        label = self.samples[self.samples.columns[0]].value_counts().index[0]

        return label

    def classify(self, sample):
        if self.is_leaf:
            return self.label
        elif len(self.children) == 1:
            return self.children[0].classify(sample)
        else:

            if (sample[self.splitFeature].values[0] <= self.splitValue) == sel
f.children[0].resultVal:
                return self.children[0].classify(sample)
            else:
                return self.children[1].classify(sample)

    def performClassification(self, samples, labels):
        results = []
        for i in range(0, len(samples.index)):
            results.append(self.classify(samples.iloc[[i]]))
        accurateResults = np.equal(np.asarray(results), np.asarray(labels))
        return np.sum(accurateResults)/len(samples.index)

    def pruneSingleGreedyNode(self, validation, testing):
        if len(self.children) == 0:
            return [self.performClassification(samples), True]
        bestValidationAccuracy = 0
        bestNode = None
        for node in self.descendants:
            parent = node.parent
            node.parent = None
            validationAccuracy = self.performClassification(validation)
            node.parent = parent;
            if validationAccuracy > bestValidationAccuracy:
                bestValidationAccuracy = validationAccuracy
                bestNode = node
        for node in bestNode.descendants:
            self.totalNodes -= 1
```

```
            if node.is_leaf:
                self.totalLeafNodes -= 1
        bestNode.parent = None
        testAccuracy = self.performClassification(testing)
        return [testAccuracy, bestValidationAccuracy, False]
```

Data Analysis Cont'd

The next step is to read in the datasets. The first run will use dataset_1

In [29]:
```
# Read in training dataset to be used
training = pd.read_csv('cancer_datasets_v2/training_1.csv')
#training.head()
trainingLabels = training[training.columns[0]]
```

In [30]:
```
# create a list of attributes for use later
# only need to do this once since all data has the same features
features = training.columns[1:]
```

In [31]:
```
# Read in validation dataset to be used
validation = pd.read_csv('cancer_datasets_v2/validation_1.csv')
#validation.head()
validationLabels = validation[validation.columns[0]]
```

In [32]:
```
# Read in test datasets to be used
test = pd.read_csv('cancer_datasets_v2/testing_1.csv')
#test.head()
testLabels = test[test.columns[0]]
```

Data Analysis: Training

The following block trains the tree on this data, using the **Entropy** impurity measure.

In [33]:
```
# Train the tree
t1 = DTCNode("Root", training, features, "Entropy",True, parent=None)
```

Data Analysis: Training Results

The following code blocks provide the total number of nodes and leaf nodes as well as the classification accuracy on the training and testing data.

In [19]:
```
print("Total Nodes", t1.totalNodes)
print("Total Leaf Nodes", t1.totalLeafNodes)
```

```
Total Nodes 34
Total Leaf Nodes 17
```

In [20]:
```
print("Training Accuracy: ", t1.performClassification(training, trainingLabels
))
print("Test Accuracy: ", t1.performClassification(test, testLabels))
```

```
Training Accuracy:  1.0
Test Accuracy:  0.9298245614035088
```

Data Analysis: Training with other dataset

The same analsysis is performed with the dataset_2

In [21]:
```
# Read in training dataset to be used
training = pd.read_csv('cancer_datasets_v2/training_2.csv')
#training.head()
trainingLabels = training[training.columns[0]]
```

In [23]:
```
# create a list of attributes for use later
# only need to do this once since all data has the same features
features = training.columns[1:]
```

In [24]:
```
# Read in validation dataset to be used
validation = pd.read_csv('cancer_datasets_v2/validation_2.csv')
#validation.head()
validationLabels = validation[validation.columns[0]]
```

In [25]:
```
# Read in test datasets to be used
test = pd.read_csv('cancer_datasets_v2/testing_2.csv')
#test.head()
testLabels = test[test.columns[0]]
```

In [26]:
```
# Train the tree
t2 = DTCNode("Root", training, features, "Entropy",True, parent=None)
```

In [27]:
```
print("Total Nodes", t2.totalNodes)
print("Total Leaf Nodes", t2.totalLeafNodes)
```

```
Total Nodes 30
Total Leaf Nodes 15
```

In [28]:
```
print("Training Accuracy: ", t2.performClassification(training, trainingLabels
))
print("Test Accuracy: ", t2.performClassification(test, testLabels))
```

```
Training Accuracy:  0.9978021978021978
Test Accuracy:  0.8771929824561403
```

Data Analysis: Dataset Comparison

As can be see, dataset 1 has a much higher initial accuracy, for both testing and training data.

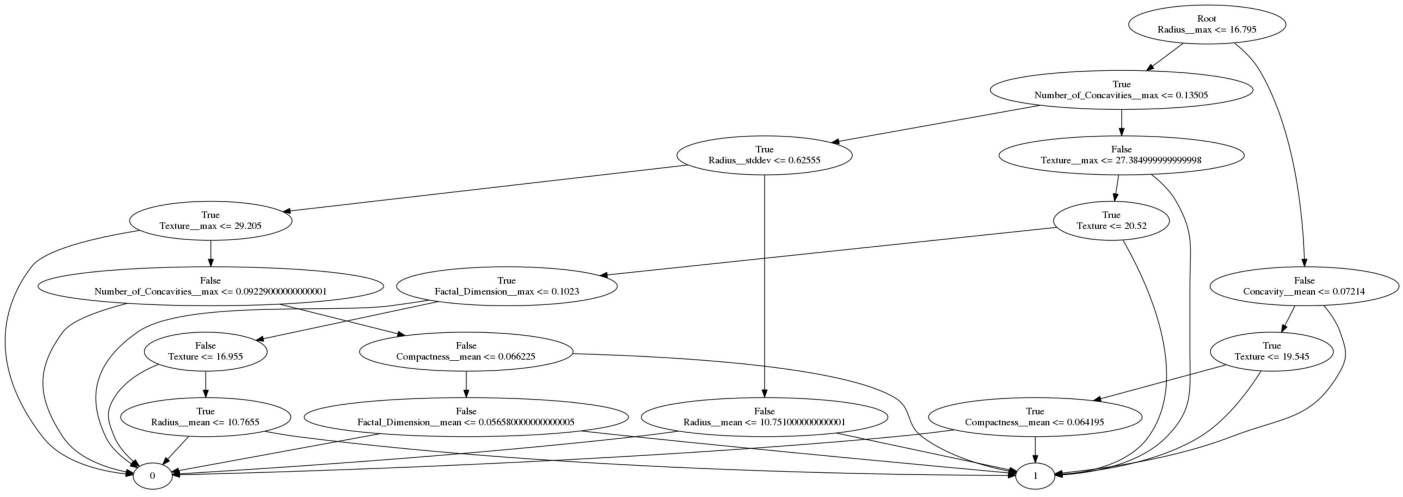
Data Analysis: Pruning

Unfortunately I ran out of time before getting to this section. The pruning algorithm is implemented (see code), but I was not able to perform the analysis asked.

Tree Visualization

The following figure is a graphical representation of a tree trained on **dataset 1** and using **Entropy** as the impurity measure. When viewing the tree, the **True/False** values at the top of each node represent the result of evaluating its parents feature split. The feature split of each node is then listed.

Note that due to the limitations of anytree's utilization of graphviz, all nodes with the same name are treated as a single node. Hence, the visualization only shows two leaf nodes named with the two possible labels, when in fact each arrow pointing to the leaf node is actually a distinct node. This is verified by the count metrics implemented in the code to determine the total number of nodes and leaf nodes (see above)



This picture will also be submitted separately.

In [ ]:
```
```