An Introduction to the decision tree

Introduction

The decision tree is one of the most commonly used classification techniques. One of the best things about decision trees is that humans can easily understand the data.

Background

I'll follow the ID3 algorithm, which tells us how to split the data and when to stop splitting it. The basic idea is to construct the decision tree by employing a top-down, greedy search through the given sets to test each attribute at every tree node.

Two the most important concepts in the ID3 algorithm are ***entropy*** and ***inforamtion gain***. These concepts are used in information theory. You can read more about ID3 and information concepts at https://en.wikipedia.org/wiki/ID3_algorithm or can read a great brief intro (https://www.cise.ufl.edu/~ddd/cap6635/Fall-97/Short-papers/2.htm) https://dzone.com/articles/machine-learning-with-decision-trees

In this tip, I will use formulas from https://www.cise.ufl.edu/~ddd/cap6635/Fall-97/Short-papers/2.htm

Assume that, we want to predict which player will receive the Golden Ball award (https://en.wikipedia.org/wiki/FIFA_Club_World_Cup_awards#Golden_Ball) in 2018. A player want to receive the Golden Ball award, he must receive the Golden Shoe award or/and he is the Champion Leage Winner. Data is collected in the following table to build a decision tree:

No.	Has Goden Shoe?	Is Champion Leage Winner?	Goden Ball?
1	Yes	Yes	Yes
2	Yes	Yes	Yes
3	No	Yes	No
4	Yes	No	No
5	Yes	No	No

The target classification can be **Yes** or **No**. First feature is **'Has Golden Shoe'** and the second feature is **'Is Champion Leage Winner'**. The values of these features are Yes or No. For simplicity, I will assign labels to our features as folows:

A -> Has Golden Shoe

B -> Is Champion Leage Winner

And for convenient, I also convert values of features from **Yes/No** to **1/0** correspondingly.

Using the Code

In Python, We can write a **dataset** based on data table above by using a function that called createDataSet:

```
def createDataSet():
	dataSet = [[1, 1, 'yes'],
		   [1, 1, 'yes'],
		   [0, 1, 'no'],
		   [1, 0, 'no'],
		   [1, 0, 'no']]
	labels = ['A','B']
	return dataSet, labels
```

Entropy of our **dataset** can be calculated:

Entropy(dataset) = - (2/5)Log2(2/5) – (3/5)Log2(3/5) = 0.9709505944546686 ~ 0.971

Our dataset is a collection of 5 examples with 2 YES and 3 NO examples.

In Python, the first, we need to count number of ‘yes’ and number of ‘no’ in the final column of **dataset**:

```
# number of data instances or number of rows of our dataSet
numIntances = len(dataSet)
# create a dictionary whose keys are the values in the final column
keyCounts = {}
# for each instance or row
for instance in dataSet:
		# return value in the final column
		currentKey = instance[-1]
		# If a key was not encountered previously, one is created
		if currentKey not in keyCounts.keys():
			keyCounts[currentKey] = 0
		# For each key, keep track of how many times this key occurs
```

```
            keyCounts[currentKey] += 1
```

If we run this code, ***keyCounts*** will look like this:

```
{'yes': 2, 'no': 3}
```

With ***keyCounts***, we can calculate entropy of our **dataset**:

```
# our entropy
entropy = 0.0
# for each key
for key in keyCounts:
      # calculate the probability of this key
      prob = float(keyCounts[key])/numIntances
      # calculate entropy
      entropy -= prob * log(prob,2)
```

And now, we can create a function that called **calcEntropy** to calculate entropy of dataset:

```
def calcEntropy(dataSet):
      # number of data instances or number of rows of our dataSet
      numIntances = len(dataSet)
      # create a dictionary whose keys are the values in the final column
      keyCounts = {}
      # for each instance or row
      for instance in dataSet:
            # return value in the final column
            currentKey = instance[-1]
            # If a key was not encountered previously, one is created
            if currentKey not in keyCounts.keys():
                  keyCounts[currentKey] = 0
            # For each key, keep track of how many times this key occurs
            keyCounts[currentKey] += 1
      # our entropy
      entropy = 0.0
      # for each key
      for key in keyCounts:
```

```
            # calculate the probability of this key
            prob = float(keyCounts[key])/numIntances
            # calculate entropy
            entropy -= prob * log(prob,2)
    return entropy
```

Now, we want to decide whether we should split the data based on the first feature or the second feature, in other words, we need to find which feature (A or B) will be the root node in our decision tree. To do this, we need to calculate information gain of each feature.

For feature A, suppose there are 4 occurrences of A = 1 and 1 occurrence of A = 0. For A = 1, 2 of the examples are Yes and 2 are No. For A = 0, 1 is No. So, information gain of A:

Gain(S, A) = Entropy(S) – (4/5)*Entropy(S1) – (1/5)*Entropy(S0) ~ 0.971 – 4/5 ~ 0.171

Entropy(S0) = 0 because all members of S0 belong to the same class (No)

Entropy(S1) = 1 because the collection contains an equal number of positive (Yes) and negative (No) examples

For feature B, suppose there are 3 occurrences of B = 1 and 2 occurrence of B = 0. For B = 1, 2 of the examples are Yes and 1 is No. For B = 0, 2 are No. So, information gain of B:

Gain(S, B) = Entropy(S) – (3/5)*Entropy(S1) – (2/5)*Entropy(S0)

Entropy(S1) = -(2/3)Log2(2/3)-(1/3)Log2(1/3) = 0.9182958340544896~0.918

Entropy(S0) = 0 because all members of S0 belong to the same class (No)

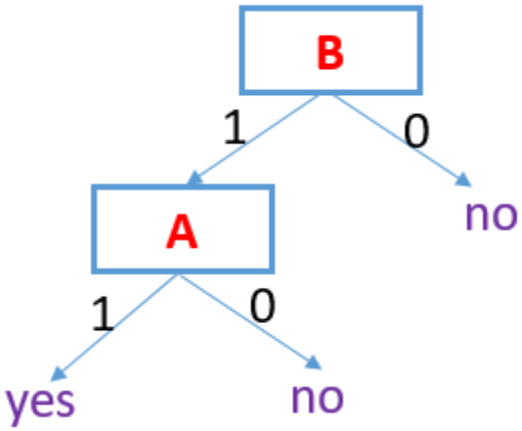
Gain(S,B) = 0.971 – (3/5)*0.918 ~ 0.420

Clearly, information gain of the B feature is higher than information gain of the A feature. Therefore, it is used as the root node in our decision tree. The next question is, should the A feature be tested at the ‘1’ branch node or the ‘2’ branch node?

Gain(S0,A) = 0 so the result of ‘0’ branch node of the B is ‘no’ and the ‘1’ branch node is the A feature.

This process goes on until all data is classified perfectly or we run out of features.

My tree looks like this:



Now, let's look our dataSet again:

$$\begin{bmatrix} 1 & 1 & yes \\ 1 & 1 & yes \\ 0 & 1 & no \\ 1 & 0 & no \\ 1 & 0 & no \end{bmatrix}$$

In Python, the A feature is the first column of **dataset** and its index is 0, the B feature is the second column of **dataset** and its index is 1, and the result of prediction is the final column. In the first column of **dataset**, there are 4 occurrences of '1' and 1 occurrence of '0', and there are 2 Yes of 1 and there are 2 No of 0. How to split the **dataset** based on a specific feature and its value? The following ***splitDataSet*** function is a solution:

```
def splitDataSet(dataSet, feature, value):
	subDataSet = []
	for row in dataSet:
		if row[feature] == value:
			reducedRow = row[:feature]
			reducedRow.extend(row[feature +1:])
			subDataSet.append(reducedRow)
	return subDataSet
```

If column = 0 and value = 1 then

```
subDataSet = [[1, 'yes'], [1, 'yes'], [0, 'no'], [0, 'no']]
```

If you are Python beginner, you can be confused about Python code above. If so, you can refer here (https://docs.python.org/2/tutorial/datastructures.html) to get a little of Python knowledge.

With the ***splitDataSet*** function above and the entropy of our **dataset**, we can calculate the information gain of a feature (such as A) as the following ***infoGain*** functtion:

```
def infoGain(dataSet, feature):
    # calculate entropy of dataset
    baseEntropy = calcEntropy(dataSet)
    # entropy of subdataset
    newEntropy = 0.0
    featList = [example[feature] for example in dataSet]
    # Get list of unique values, ex: [0,0,1,1,1] -> {0,1}
    uniqueVals = set(featList)
    # iterate over all the unique values of this feature and split the data
    # for each feature
    for value in uniqueVals:
        # split dataset based on feature and value
        subDataSet = splitDataSet(dataSet, feature, value)
        # The new entropy is calculated and summed up for all the unique
        # values of that feature
        prob = len(subDataSet)/float(len(dataSet))
        newEntropy += prob * calcEntropy(subDataSet)
    # the information gain of feature is the reduction in entropy
    gain = baseEntropy - newEntropy
    return gain
```

Gain(S, A) = infoGain(dataSet, 0) = 0.17095059445466854 ~ 0.171

Gain(S,B) = infoGain(dataSet, 1) = 0.4199730940219749 ~ 0.420

When we have the information gain of features, it's easy to find the best feature based on the best gain:

```
def chooseBestFeature(dataSet):
    numFeatures = len(dataSet[0]) - 1
    bestInfoGain = 0.0; bestFeature = -1
    for i in range(numFeatures):
        gain = infoGain(dataSet,i)
        if (gain > bestInfoGain):
```

```
                        bestInfoGain = gain
                        bestFeature = i
            return bestFeature
```

Now, we can create our decision tree. I created the decision tree as the following Python code:

```
# find the majority result class ('yes' or 'no') that occurs with the greatest
frequency.
def majorityClass(classList):
      classCount={}
      for vote in classList:
            if vote not in classCount.keys(): classCount[vote] = 0
            classCount[vote] += 1
      sortedClassCount = sorted(classCount.items(),key=operator.itemgetter(1),
reverse=True)
      return sortedClassCount[0][0]
# create my decision tree
def createTree(dataSet,labels,level):
      classList = [example[-1] for example in dataSet]
      # Stop when all result classes are equal
      if classList.count(classList[0]) == len(classList):
            return classList[0]
      # When no more features, return majority class
      if len(dataSet[0]) == 1:
            return majorityClass (classList)
      bestFeat = chooseBestFeature(dataSet)
      bestFeatLabel = labels[bestFeat]
      if level == 0:
            myTree = "Root =>   " + bestFeatLabel + '\n\n'
      else:
            myTree = bestFeatLabel + '\n\n'
      del(labels[bestFeat])
      featValues = [example[bestFeat] for example in dataSet]
      # Get list of unique values, ex: [0,0,1,1,1] -> {0,1}
```

```
	uniqueVals = set(featValues)
	# levels of tree
	level = level + 1
	myTree += "Level " + str(level) + " =>   "
	# iterate over all the unique values from our chosen feature and recursively
	# call createTree() for each split of the dataset
	for value in uniqueVals:
		subLabels = labels[:]
		myTree += 'Branch ' + str(value)+ ' of ' + bestFeatLabel + ' : '
		myTree += createTree(splitDataSet(dataSet, bestFeat,
value),subLabels,level) + ' '
	return myTree
```

if running code above, my decision tree will look like this:

Root => B

Level 1 => Branch 0 of B : no Branch 1 of B : A

Level 2 => Branch 0 of A : no Branch 1 of A : yes

You can try plotting a decision tree for yourself by using the matplotlib.

So far, we can collect and order all of our functions above and put them together into the file that called ***decisionTree.py***. The content of this file maybe look like this:

```
	def createDataSet():
	…
	def calcEntropy(dataSet):
	…
	def infoGain(dataSet, feature):
	…
	def splitDataSet(dataSet, feature, value):
	…
	def chooseBestFeature(dataSet):
	…
```

```
def majorityCnt(classList):
…
def createTree(dataSet,labels,level):
…
```

You can get a completed one of this file here.

We can create a file, called ***Test.py***, to test our functions as follows:

```
import decisionTree


# create a dataset and labels
myDat,labels= decisionTree.createDataSet()
print(myDat)
# calculate the entropy of the dataset
entropy = decisionTree.calcEntropy(myDat)
print(entropy)
# calculate information gain of A feature
gainA = decisionTree.infoGain(myDat,0)
print(gainA)
# calculate information gain of B feature
gainB = decisionTree.infoGain(myDat,1)
print(gainB)
# split the dataset based on A feature and its 1 value
splitA = decisionTree.splitDataSet(myDat,0,1)
print(splitA)
# choose the best feature
best = decisionTree.chooseBestFeature(myDat)
print(best)
# create my decision tree
tree = decisionTree.createTree(myDat,labels,0)
print(tree)
```

Points of Interest

We have created conclusions about data such as “This data instance is in this class!” The next questtion will be “What if we assign a probability to a data instance belonging to a given class?” The answer will be discussed in the next tip.