100 Nice code organization and use of classes to organize tree building.

Information-based Learning using Decision Trees

Julian Carrasquillo - CIS678, WIN20

Specification

The basic idea is to write a program that, given a collection of training data for a classification problem, generates a Decision Tree via the ID3 algorithm.

Background

Decision trees are hierarchical data structures functioning as classifier systems. They are constructed based on a set of training data for which the value of the target function is known (i.e. they are a form of Supervised Learning). ID3 is a greedy algorithm that generates shortest-path decision trees.

Implementation

To implement the algorithm, you will need:

- A measure of purity (e.g. Entropy):

$$Entropy(S) = -\sum_{i=1}^{k} p_i log_2(p_i)$$

where S is the collection of examples, k is the number of categories, and pi is the ratio of the cardinality of category i to the cardinality of S, as in $p_i = \frac{N_i}{N}$

- The formula for Information Gain:

$$Gain(S,a) = Entropy(S) - \sum_{v=values(a)} \frac{|S_v|}{|S|} Entropy(S_v)$$

  where values(a) is the set of all possible values for attribute a, and Sv is the subset of set S for which attribute a has value v.

A custom Python class `node` will be used to house the various attributes that can be associated with a node in our decision tree. For example, it will include variables `parent_node` , `child_node` , `num_members` , `class_makeup` , and `entropy` , as well as a `.split_node()` method.

Data Reading

The given Datafile format is:

NumTargets
targetNames
NumAttributes
attributeName numValues attributeValues (discrete) real (continuous)
NumExamples
attributeValues targetValue

In [558]:

```
import pandas as pd
import numpy as np

temp = pd.read_table('data/contact-lenses.data', header = None).head(12)

for i in range(2,7):
    print(temp[0][i])
```

```
4
age,3,young,pre-presbyopic,presbyopic
prescription,2,myope,hypermetrope
astigmatism,2,no,yes
tear-rate,2,reduced,normal
```

Note that the comma-separated data starts at row 8 above. To assist in making the program more robust, we will decouple the algorithm from handling this specific, somewhat unconventional file type. The header contains important information, so a function could be used to extract what we need and export the dataframe.

Specifically, we want to extract the column names using the `NumAttributes` line in the file. This tells us how many columns we will ultimately have and where the actual data table starts.

In [559]:

```
def read_data_file(file):
    import re
    col_names =[]
    temp = pd.read_table(file, header = None)
    num_cols = temp[0][2]

    # get column names
    for i in range(3, 3 + int(num_cols)):
        col_names.append(re.match("^([A-Za-z-_]+),", temp[0][i]).group(1))

    # attached final column name to fit table
    col_names.append('target')

    df = pd.DataFrame(temp[0][3 + int(num_cols) + 1:].str.split(',', expand =
True)).reset_index(drop = True)
    df.columns = col_names
    return df
```

Now we can use these data sets to build our trees!

In [560]:
```
read_data_file('data/fishing.data')
```

Out[560]:

	Wind	Water	Air	Forecast	target
0	Strong	Warm	Warm	Sunny	Yes
1	Weak	Warm	Warm	Sunny	No
2	Strong	Warm	Warm	Cloudy	Yes
3	Strong	Moderate	Warm	Rainy	Yes
4	Strong	Cold	Cool	Rainy	No
5	Weak	Cold	Cool	Rainy	No
6	Weak	Cold	Cool	Sunny	No
7	Strong	Moderate	Warm	Sunny	Yes
8	Strong	Cold	Cool	Sunny	Yes
9	Strong	Moderate	Cool	Rainy	No
10	Weak	Moderate	Cool	Sunny	Yes
11	Weak	Moderate	Warm	Sunny	Yes
12	Strong	Warm	Cool	Sunny	Yes
13	Weak	Moderate	Warm	Rainy	No

In [561]:
```
fishing = read_data_file('data/fishing.data')
contacts = read_data_file('data/contact-lenses.data')
iris = read_data_file('data/iris.data')
```

Building out Utility

Despite the overall design of the problem including data structures and logic flow, we should build a modular toolkit to get key values. Let's start with entropy. It should take a dataframe and return a number based on the target column.

In [562]:
```
def calc_entropy(df, target):
    from math import log
    total_records = df.shape[0]
    total_entropy = 0
    num_classes = np.unique(df[target]).shape[0]
    if num_classes == 1: num_classes = num_classes + 1

    for item in np.unique(df[target]):
        item_records = df[np.array(df[target] == item)].shape[0]
        total_entropy = total_entropy - ((item_records / total_records) * log(
item_records / total_records, num_classes))
    return total_entropy
```

In [563]:
```
calc_entropy(fishing, 'target')
```

Out[563]: 0.9852281360342516

Note the dynamic base of our `log` calculation. In order to keep the entropy calculation between `0` and `1`, we scale with a log base equal to the number of classes. In situations where a single class is available in our dataframe (`num_classes == 1`), adding `1` helps us avoid an error of finding a log base 1 number.

Now that we have the method to calculate entropy, we can use it to compare different splits in our data. It's important to note that if a split has already occurred on a category variable, than the algorithm will never split on that column because the information gain will always be `0`. Said differently, if we "split" on a column with one value, there will never be information gained.

In [564]:

```
def find_best_split(df, target):
    best_sub_class = {}
    # get starting entropy
    parent_entropy = calc_entropy(df, target)
    parent_rows = df.shape[0]
    best_gain = 0

    for col in df.columns:
        if col != target:
            sub_class_sum = 0
            sub_dfs = []
            for sub_class in np.unique(df[col]):
                sub_temp = df[df[col]  == sub_class]
                sub_rows = sub_temp.shape[0]
                sub_class_sum = sub_class_sum + (sub_rows / parent_rows)*calc_
entropy(sub_temp, target)
                sub_dfs.append(sub_temp)
            if parent_entropy - sub_class_sum > best_gain:
                best_sub_class = {col : {"information_gain" : parent_entropy -
sub_class_sum},
                                  "dfs" : sub_dfs }
                best_gain = parent_entropy - sub_class_sum

    return best_sub_class
```

In [565]:
```
find_best_split(fishing, 'target')
```

Out[565]:
```
{'Forecast': {'information_gain': 0.2638091738835463},
 'dfs': [      Wind Water   Air Forecast target
  2  Strong  Warm  Warm   Cloudy    Yes,
        Wind     Water   Air Forecast target
  3   Strong  Moderate  Warm    Rainy    Yes
  4   Strong      Cold  Cool    Rainy     No
  5     Weak      Cold  Cool    Rainy     No
  9   Strong  Moderate  Cool    Rainy     No
  13    Weak  Moderate  Warm    Rainy     No,
        Wind     Water   Air Forecast target
  0   Strong      Warm  Warm    Sunny    Yes
  1     Weak      Warm  Warm    Sunny     No
  6     Weak      Cold  Cool    Sunny     No
  7   Strong  Moderate  Warm    Sunny    Yes
  8   Strong      Cold  Cool    Sunny    Yes
  10    Weak  Moderate  Cool    Sunny    Yes
  11    Weak  Moderate  Warm    Sunny    Yes
  12  Strong      Warm  Cool    Sunny    Yes]}
```

The `Node` Class

To package our information, we can fold these utility functions into a custom `node` class. Given a dataframe and target, we can

- store them
- calculate the entropy
- apply class and overall member counts, as well as suggest a highest vote classification
- find the best split for the specific node
- set up attributes for parent / child lineage

Using this method allows us to keep associated values packged together. It is also easier to think of an abstract object with various attributes that make it distinct from others. It also saves us the mental calories of building out various variables holding different values:

- `root_node_entropy`
- `root_node_df`
- `layer1_node1_children`
- `layer3_node4_best_split`
- ...

There are various components of the `Node` class worth exploring.

- the `def __init__` sets up the `Node` object with starting information. Given a dataframe `df` and a target column, we can calculate everything else.
- `calc_entropy` and `find_best_split` are similar to before, with a main change in `find_best_split` being the output format. Instead of the embedded {*Splitting_Attr* : {"Information Gain" : ..., "dfs" : ...}}, we have an easier to parse {"feature" : *Splitting_Attr*, "Information Gain":...,"dfs":...}

In [566]:

```
class Node:
    def __init__(self, df, target):
        self.id = 0
        self.df = df
        self.target = target
        self.entropy = self.calc_entropy()
        self.members = df.shape[0]
        self.classes = self.count_classes()
        self.classify = max(self.classes, key=lambda key: self.classes[key])
        self.best_sub_class = self.find_best_split()
        self.parent = []
        self.children = []

    def calc_entropy(self):
        from math import log
        total_records = self.df.shape[0]
        total_entropy = 0
        num_classes = np.unique(self.df[self.target]).shape[0]
        if num_classes == 1: num_classes = num_classes + 1

        for item in np.unique(self.df[self.target]):
            item_records = self.df[np.array(self.df[self.target] == item)].sha
pe[0]
            total_entropy = total_entropy - ((item_records / total_records) *
log(item_records / total_records, num_classes))

        return total_entropy

    def count_classes(self):
        classes = {}
        for item in np.unique(self.df[self.target]):
            classes.update({item : self.df[self.df[self.target] == item].shape
[0]})
        return classes

    def find_best_split(self):
        best_sub_class = {}
        # get starting entropy
        parent_entropy = self.entropy
        parent_rows = self.members
        best_gain = 0

        for col in self.df.columns:
            if col != self.target:
                sub_class_sum = 0
                sub_dfs = []
                for sub_class in np.unique(self.df[col]):
                    sub_temp = self.df[self.df[col]  == sub_class]
                    sub_rows = sub_temp.shape[0]
                    sub_class_sum = sub_class_sum + (sub_rows / parent_rows)*c
alc_entropy(sub_temp, self.target)
                    sub_dfs.append(sub_temp)
                if parent_entropy - sub_class_sum > best_gain:
                    best_sub_class = {"feature" : col,
                                      "information_gain" : parent_entropy - su
b_class_sum,
```

```python
                                  "dfs" : sub_dfs }
                        best_gain = parent_entropy - sub_class_sum
            self.best_sub_class = best_sub_class
            return best_sub_class

    def print_attr(self):
        print("Node ID: ", self.id)
        print("Node Parent: ", self.parent)
        print("Node Children: ", self.children)
        print("Node Members: ", self.members)
        print("Node Classes: ", self.classes)
        print("Node Classification: ", self.classify)
```

In [567]:

```python
test_node = Node(fishing, 'target')

print("Node Dataframe: \n", test_node.df)
test_node.print_attr()
```

```
Node Dataframe: 
       Wind     Water   Air Forecast target
0    Strong      Warm  Warm    Sunny    Yes
1      Weak      Warm  Warm    Sunny     No
2    Strong      Warm  Warm   Cloudy    Yes
3    Strong  Moderate  Warm    Rainy    Yes
4    Strong      Cold  Cool    Rainy     No
5      Weak      Cold  Cool    Rainy     No
6      Weak      Cold  Cool    Sunny     No
7    Strong  Moderate  Warm    Sunny    Yes
8    Strong      Cold  Cool    Sunny    Yes
9    Strong  Moderate  Cool    Rainy     No
10     Weak  Moderate  Cool    Sunny    Yes
11     Weak  Moderate  Warm    Sunny    Yes
12   Strong      Warm  Cool    Sunny    Yes
13     Weak  Moderate  Warm    Rainy     No
Node ID:  0
Node Parent:  []
Node Children:  []
Node Members:  14
Node Classes:  {'No': 6, 'Yes': 8}
Node Classification:  Yes
```

Once we learn the best split within a node, we can use the output `df` s from `find_best_split` to make new node instances.

In [568]:
```
def make_new_layer(node):
    nodes = []
    for i in range(len(node.best_sub_class['dfs'])):
        temp_node = Node(node.best_sub_class['dfs'][i], node.target)
        temp_node.parent = [node.id]
        node.children.append(temp_node.id)
        nodes.append(temp_node)

    return nodes
```

In [569]:
```
test_sub_nodes = make_new_nodes(test_node)
```

In [570]:
```
test_sub_nodes[2].df
```

Out[570]:

	Wind	Water	Air	Forecast	target
0	Strong	Warm	Warm	Sunny	Yes
1	Weak	Warm	Warm	Sunny	No
6	Weak	Cold	Cool	Sunny	No
7	Strong	Moderate	Warm	Sunny	Yes
8	Strong	Cold	Cool	Sunny	Yes
10	Weak	Moderate	Cool	Sunny	Yes
11	Weak	Moderate	Warm	Sunny	Yes
12	Strong	Warm	Cool	Sunny	Yes

Now that we have nodes and a way to split them, we need a way to associate nodes together and form a lineage. We can use another class structure to accomplish that. This encapsulation will also help us handle the `node.id` attribute within a single tree structure. Adding some logic to handle nodes at the end of a lineage, as well as a `max_depth` argument to give a little more control to the user, we orchestrate a layer building process using the abstraction function `build_tree`.

With the `tree` class in place, we need a way to allow a record to traverse down the nodes into its own bin. Because each `node` knows its `best_sub_class`, we can compare the value of our test record against the unique value within each subclass data frame. Using the `children` and `test_layer` attributes associated with each node, we can track the `current_node`.

In the `.predict()` method, we capture the above logic and recursively call the function until the `current_node` has no more descendents. Said differently, we move until we reach the last node and then call then read the `classification` attribute. Once we extract out our value, we reset the test attributes to set the object up for another prediction.

In [571]:
```
class Tree:
    def __init__(self, root_node):
        self.current_tree = {0 : {root_node.id : root_node}}
        self.root = root_node
        self.current_layer = 0
        self.test_layer = 0
        self.current_node_id = 1
        self.current_node = root_node
        self.classification = ""

    def build_tree(self, max_depth):
        # make a tree up to max_depth where max_depth = 1 is a tree with just
 the root node.
        while self.current_layer < (max_depth - 1):
            self.make_new_layer(max_depth)

    def predict(self, record):
        if (self.current_node.best_sub_class != {}) & (self.test_layer <= self
.current_layer):
            splitting_feature = self.current_node.best_sub_class['feature']
            if self.current_node.children != []:
                for child in self.current_node.children:
                    if (record[splitting_feature] in self.current_tree[self.te
st_layer + 1][child].df[splitting_feature].unique()):
                        self.current_node = self.current_tree[self.test_layer
+ 1][child]
                        self.test_layer = self.test_layer + 1
                        break
            self.predict(record)
        else:
            self.classification = self.current_node.classify
            self.current_node = self.root
            self.test_layer = 0
        return self.classification

    def make_new_layer(self, max_depth):
        nodes = {}
        # iterate over each node in the current layer
        for layer_node in self.current_tree[self.current_layer].values():
            # if the node can be split further...
            if layer_node.best_sub_class != {}:
                # iterate over each of these subclass dfs and make a node
                for i in range(len(layer_node.best_sub_class['dfs'])):
                    temp_node = Node(layer_node.best_sub_class['dfs'][i], laye
r_node.target)
                    temp_node.parent = [layer_node.id]
                    temp_node.id = self.current_node_id
                    self.current_node_id = self.current_node_id + 1
                    layer_node.children.append(temp_node.id)
                    # add all of these nodes to a dictionary
                    nodes.update({temp_node.id : temp_node})
        # if the dictionary is unempty, update the current tree with this new
 layer of nodes.
        if nodes != {}:
            self.current_layer = self.current_layer + 1
            self.current_tree.update({self.current_layer : nodes})
```

```
        else:
            # if we have an empty node set, then no more layers will be made
            # need to set current layer to max_depth to override build_tree lo
op
            self.current_layer = max_depth
```

We can now apply the `Node` and `Tree` classes to build out decisions trees for both of the categorical data sets. Using `apply`, we can predict each record.

In [572]:
```
fish_node = Node(fishing, 'target')
fish_tree = Tree(fish_node)
fish_tree.build_tree(10)

contact_node = Node(contacts, 'target')
contact_tree = Tree(contact_node)
contact_tree.build_tree(10)
```

In [573]:
```
fishing['pred'] = fishing.apply(fish_tree.predict, axis = 1)

contacts['pred'] = contacts.apply(contact_tree.predict, axis = 1)
```

In [574]:
```
fishing
```

Out[574]:

	Wind	Water	Air	Forecast	target	pred
0	Strong	Warm	Warm	Sunny	Yes	Yes
1	Weak	Warm	Warm	Sunny	No	No
2	Strong	Warm	Warm	Cloudy	Yes	Yes
3	Strong	Moderate	Warm	Rainy	Yes	Yes
4	Strong	Cold	Cool	Rainy	No	No
5	Weak	Cold	Cool	Rainy	No	No
6	Weak	Cold	Cool	Sunny	No	No
7	Strong	Moderate	Warm	Sunny	Yes	Yes
8	Strong	Cold	Cool	Sunny	Yes	Yes
9	Strong	Moderate	Cool	Rainy	No	No
10	Weak	Moderate	Cool	Sunny	Yes	Yes
11	Weak	Moderate	Warm	Sunny	Yes	Yes
12	Strong	Warm	Cool	Sunny	Yes	Yes
13	Weak	Moderate	Warm	Rainy	No	No

In [575]:
```
contacts
```

Out[575]:

	age	prescription	astigmatism	tear-rate	target	pred
0	young	myope	no	reduced	none	none
1	young	myope	no	normal	soft	soft
2	young	myope	yes	reduced	none	none
3	young	myope	yes	normal	hard	hard
4	young	hypermetrope	no	reduced	none	none
5	young	hypermetrope	no	normal	soft	soft
6	young	hypermetrope	yes	reduced	none	none
7	young	hypermetrope	yes	normal	hard	hard
8	pre-presbyopic	myope	no	reduced	none	none
9	pre-presbyopic	myope	no	normal	soft	soft
10	pre-presbyopic	myope	yes	reduced	none	none
11	pre-presbyopic	myope	yes	normal	hard	hard
12	pre-presbyopic	hypermetrope	no	reduced	none	none
13	pre-presbyopic	hypermetrope	no	normal	soft	soft
14	pre-presbyopic	hypermetrope	yes	reduced	none	none
15	pre-presbyopic	hypermetrope	yes	normal	none	none
16	presbyopic	myope	no	reduced	none	none
17	presbyopic	myope	no	normal	none	none
18	presbyopic	myope	yes	reduced	none	none
19	presbyopic	myope	yes	normal	hard	hard
20	presbyopic	hypermetrope	no	reduced	none	none
21	presbyopic	hypermetrope	no	normal	soft	soft
22	presbyopic	hypermetrope	yes	reduced	none	none
23	presbyopic	hypermetrope	yes	normal	none	none

Discussion

Because we did not prune our tree, we will get an output that perfectly captured each record. This is of course not particularly useful in the real world, but it is a good place to start. Though we introduced a `max_depth` argument in our tree builder method, the recursive nature of the function needs to be revisited. In the situation where the tree was shorter than it could be, the method caused an infinite loop! This is likely due to how the method checks for the next level. As opposed to checking for children nodes, it checks for whether or not the node itself has a better sub class. If we prune our tree, a node may still have a split it could make.

The next level would be to incorporate numeric columns. Not only would this open up the range of datasets we could build trees for, but it also sets us up to tackle gradient boosted trees. These are iterative trees that build on the *errors* of the tree before it. A flavor of such an algorithm, `xgboost` , is a consistent winner in many Kaggle competitions. It was developed by Tianqi Chen at the University of Washington.