In the name of God

Decision Tree Python code report

by

Parto Eshraghi

Machine learning

Dr Seyedin

Fall of 1400

Introduction

We are going to apply decision tree on Adult dataset from UCI, and we used just some of the features as below:

1) Work class (8 values)

2) Education (16 values)

3) Marital-status (7 values)

4) Occupation (14 values)

5) Relationship (6 values)

6) Race (5 values)

7) Sex (2 values)

8) native-country (41 values)

For example in dataset file we have:

<=50K, State-gov, Bachelors, Never-married, Adm-clerical, Not-in-family, White, Male, United-States

(The first element is the label)

First with 10000 data we're going to train our Tree and then with another 10000 data we'll test it.

Code explanation

We have 3 python files: Result, DecisionTree, TreeNode.

- In Result

We read data from file and as we want to use some random of train data for training and some other for validation, we define 3 functions in this file as below:

```
def load_data(add):
    data = pd.read_csv(add)
    data.columns = ['Lable','workclass','education','marital-
status','occupation','relationship','race','sex','native-country']
    data = pd.DataFrame(data)
    data = data.replace({'<=50K':0})
    data = data.replace({'>50K':1})
    return data

def pre_train(train_data,percentage):
    idx = range(len(train_data))
    idx = list(idx)
    random.shuffle(idx)
    aa = int(percentage*len(train_data)/100)
    idx = idx[:aa]
    x1 = train_data.iloc[idx,:]
    x1.index = range(aa)
    x = x1.iloc[:, 1:]
    y = x1.iloc[:, 0]
    return x,y

def train_val_split(data,percentage):
    idx = range(len(data))
    idx = list(idx)
    random.shuffle(idx)
    aa = int(percentage*len(data)/100)
    idx1 = idx[:aa]
    idx2 = idx[aa:]
    x_tr = data.iloc[idx1,:]
    x_val = data.iloc[idx2,:]
    x_tr.index = range(aa)
    x_val.index = range(len(data)-aa)
    x_tr_ = x_tr.iloc[:, 1:]
    y_tr_ = x_tr.iloc[:, 0]
    x_val_ = x_val.iloc[:, 1:]
    y_val_= x_val.iloc[:, 0]
```

```
        return x_tr_,y_tr_,x_val_,y_val_
```

Application of load_data is obvious, but in pre_train base on a given percentage we randomly choose some data and then we put the features in X and labels in Y, and we do the same in train_val_split and we just use the rest of data for validation.

- In TreeNode

We define a class that stores our tree nodes, each node has ids, children, entropy, depth, split_attribute, order and label.

Ids is list include pattern indexes with this feature.

Children include nodes that come after this node and attached to this node with branches.

Entropy is obvious. Depth is the maximum number of nodes in one path that come after this node.

Split_attribute that actually calculate in DecisionTree file, helps us with finding attribute of this node. Order include values of this feature in data. And label is the label (in this case 0, 1 under some conditions).

Class has 4 methods, a constructor, a represent, set_properties, set_label.

```
class TreeNode(object):
    def __init__(self, ids = [], children = [], entropy = 0, depth = 0):
        self.ids = ids
        self.entropy = entropy
        self.depth = depth
        self.split_attribute = None
        self.children = children
        self.order = None
        self.label = None

    def __repr__(self):
        if self.split_attribute != None :return self.split_attribute
        else: return 'leaf'

    def set_properties(self, split_attribute, order):
        self.split_attribute = split_attribute
        self.order = order
```

```
    def set_label(self, label):
        self.label = label
```

- In DecisionTree

First we define entropy function that calculate entropy that we're going to use it later, as below:

```
def entropy(freq):
    freq_0 = freq[np.array(freq).nonzero()[0]]
    prob_0 = freq_0/float(freq_0.sum())
    return -np.sum(prob_0*np.log2(prob_0))
```

then we define our DecisionTree class, in general this class has 6 variables, root(first node with maximum entropy), max_depth, Ntrain which is number of data, data that include data without labels, target which include all labels of all features, label (here is 0, 1 only) and more.

Now we're going through methods:

At first we have constructor:

```
def __init__(self, max_depth = 10):
        self.root = None
        self.max_depth = max_depth
        self.Ntrain = 0
        self.nodes = []
```

With these two methods,

```
def _entropy(self, ids):
        if len(ids) == 0: return 0
        ids = list(ids)
        freq = np.array(self.target[ids].value_counts())
        return entropy(freq)

    def _set_label(self, node):
        target_ids = list(node.ids)
        node.set_label(self.target[target_ids].mode()[0])
```

we can calculate entropy and if after using all features we weren't able to classify we use _set_label and put in class with the most label.

For creating Tree we use 4 methods as below and we actually implement ID3 algorithm with these four:

```
def fit(self, data, target, x_val, y_val):
        self.Ntrain = data.count()[0]
        self.data = data
        self.attributes = list(data)
        self.target = target
        self.labels = target.unique()

        ids = range(self.Ntrain)
        self.root = tn.TreeNode(ids = ids, entropy = self._entropy(ids), depth =
0)
        queue = [self.root]

        while queue:
            node = queue.pop()

            if node.depth < self.max_depth:
                node.children = self._split(node)
                if not node.children:
                    self._set_label(node)
                queue += node.children
                self.nodes += node.children
            else:
                self._set_label(node)
        self.acc = self.evaluation(x_val,y_val,True)
```

In fit method first we choose the root and then with help of _split method we can find the children of each node base on values of each feature and entropy. At the end we use validation data in the last line and evaluate accuracy.

```
    def _split(self, node):
        ids = node.ids
        best_gain = 0
        best_splits = []
        best_attribute = None
        order = None
        sub_data = self.data.iloc[ids, :]
        for i, att in enumerate(self.attributes):
            values = self.data.iloc[ids, i].unique().tolist()
            if len(values) == 1: continue
```

```
            splits = []
            for val in values:
                sub_ids = sub_data.index[sub_data[att] == val].tolist()
                splits.append([sub_id for sub_id in sub_ids])
                H = 0
            for split in splits:
                H += len(split)*self._entropy(split)/len(ids)
            gain = node.entropy - H
            if gain > best_gain:
                best_gain = gain
                best_splits = splits
                best_attribute = att
                order = values
        node.set_properties(best_attribute, order)
        child_nodes = [tn.TreeNode(ids = split,
                    entropy = self._entropy(split), depth = node.depth + 1) for
split in best_splits]
        return child_nodes
```

In _split we define attributes (work class, education, marital-status, occupation, relationship, race, sex, and native-country) then we enumerate in them and for each attribute we find every value that it can take (unique()) and find indexes of every value and put in list and make 2D array (splits) now that we have the number of each value of each attribute we can easily find IG and arrange our nodes.

```
    def predict(self, new_data):

        npoints = new_data.count()[0]
        labels = [None]*npoints
        for n in range(npoints):
            x = new_data.iloc[n, :]
            node = self.root
            while node.children:
                if x[node.split_attribute] in node.order:
                    node = node.children[node.order.index(x[node.split_attribute]
)]
                else:
                    node.label = self._set_label(node)
                    break
            labels[n] = node.label
        return labels

    def evaluation(self,x_test,y_test,show_acc=False):
        z = self.predict(x_test)
```

```
        count = 0
        for i in range(len(y_test)):
            if y_test[i] == z[i]:
                count += 1
        if show_acc:
            print((count/len(x_test))*100)
        self.acc = count/len(x_test)*100
        return count/len(x_test)*100
```

at last in predict, we predict the label for our test data, and in evaluation we find if the predicted labels are as same as the real labels and find accuracy.

In the last method we are using pruning for a better answer:

```
    def pruning(self,x2,y2):
        max_depth = 0
        for n in self.nodes:
            if n.depth > max_depth:
                max_depth = n.depth
        prun_list = []
        for n in self.nodes:
            if n.children:
                prun_list.append(n)
        prun_list1 = []
        for dep in range(1,max_depth):
            for n in prun_list:
                if n.depth == dep: prun_list1.append(n)
        before_pr = len(prun_list1)
        acc = 100
        last_acc = self.acc
        nodes = []
        val_acc = []
        while acc +.2 >= last_acc:
            pr_node = prun_list1.pop()
            if pr_node.children:
                last_node = pr_node.children
                pr_node.children = []
                self._set_label(pr_node)
                last_acc = self.acc
                acc = self.evaluation(x2,y2,False)
                before_pr -= 1
                nodes.append(before_pr)
                print(before_pr)
```

```
                val_acc.append(self._eval(x2,y2,True))
        pr_node.children = last_node
        self.acc = self.evaluation(x2,y2)

        return nodes,val_acc,max_depth
```

with pruning we achieved a smaller and better Tree.

When you run the code you can see how pruning works and reduce the nodes and increase accuracy.

The result for 75 percent of train data for train and 25 percent for validation and 100 percent of test data is:

```
87.40874087408741
None
87.40874087408741
```