ECOLE CENTRALE SUPELEC
MASTER DSBI

DECISION MODELING
TUTORIAL

AN UNDERSTANDABLE DECISION MODEL : THE DECISION TREES

Decision trees are one of the simpler machine-learning methods. They are a completely transparent method of classifying observations, which, after training, look like a series of if-then statements arranged into a tree. Unlike most other classifiers, the models produced by decision trees are easy to interpret, the list of numbers in a Bayesian classifier will tell you how important each word is, but you really have to do the calculation to know what the outcome will be. A neural network is even more difficult to interpret, since the weight of the connection between two neurons has very little meaning on its own. You can understand the reasoning process of a decision tree just by looking at it, and you can even convert it to a simple series of if-then statements. Indeed once you have a decision tree, it's quite easy to see how it makes all of its decisions. Just follow the path down the tree that answers each question correctly and you'll eventually arrive at an answer. Tracing back from the node where you ended up gives a rationale for the final classification.

We look at a way to represent a decision tree, at code for constructing the tree from real data, and at code for classifying new observations.

Python language is only used here as an example in the implementation of this model, but you can choose another language (Java, C++, . . .). To learn about Python, please visit this website `https://ocw.mit.edu/courses/electrical-engineering-and-cc` `6-189-a-gentle-introduction-to-programming-using-python-january-iap-2011/lectures/` or some tutorials in the web.

The first step is to create a representation of a tree. Create a new class called decisionnode, which represents each node in the tree :

```
class decisionnode:
 def __init__(self,col=-1,value=None,results=None,tb=None,fb=None):
  self.col=col
  self.value=value
  self.results=results
  self.tb=tb
  self.fb=fb
```

Each node has five instance variables, all of which may be set in the initializer :
- `col` : is the column index of the criteria to be tested ;
- `value` : is the value that the column must match to get a true result ;
- `results` : stores a dictionary of results for this branch. This is `None` for everything, except endpoints.
- `tb` and `fb` are `decisionnode` which are the next nodes in the tree if the result is true or false, respectively.

The functions that create a tree return the root node, which can be traversed by following its True or False branches until a branch with results is reached.

Predicting the behaviors of users for a website

Sometimes when a high-traffic site links to a new application that offers free accounts and subscription accounts, the application will get thousands of new users. Many of these users are driven by curiosity and are not really looking for that particular type of application, so there is a very low likelihood that they will become paying customers. This makes it difficult to distinguish and follow up with likely customers, so many sites resort to mass-emailing everyone who has signed up, rather than using a more targeted approach.

To help with this problem, it would be useful to be able to predict the likelihood that a user will become a paying customer. You know by now that you can use a Bayesian classifier or neural network to do this. However, clarity is very important in this case, if you know the factors that indicate a user will become a customer, you can use that information to guide an advertising strategy, to make certain aspects of the site more accessible, or to use other strategies that will help increase the number of paying customers.

For this example, imagine an online application that offers a free trial. Users sign up for the trial and use the site for a certain number of days, after which they can choose to upgrade to a basic or premium service. As users sign up for the free trial, information about them is collected, and at the end of the trial, the site owners note which users chose to become paying customers.

To minimize annoyance for users and sign them up as quickly as possible, the site does not ask them a lot of questions about themselves instead, it collects information from the server logs, such as the site that referred them, their geographical location, how many pages they viewed before signing up, and so on. If you collect the data and put it in a table, it might look like the following Table of User behavior and final purchase decision for a web site :

Referrer	Location	Read FAQ ?	Pages viewed	Service chosen
Slashdot	USA	Yes	18	None
Google	France	Yes	23	Premium
Digg	USA	Yes	24	Basic
Kiwitobes	France	Yes	23	Basic
Google	Angleterre	No	21	Premium
(direct)	Allemagne	No	12	None
(direct)	Angleterre	No	21	Basic
Google	USA	No	24	Premium
Slashdot	France	Yes	19	None
Digg	USA	No	18	None
Google	Angleterre	No	18	None
Kiwitobes	Angleterre	No	19	None
Digg	Allemagne	Yes	12	Basic
Slashdot	Angleterre	No	21	None
Google	Angleterre	Yes	18	Basic
Kiwitobes	France	Yes	19	Basic

The final column in each row indicates whether or not the user signed up ; this Service column is the value you want to be able to predict.

We will try to implement an algorithm called CART (Classification and Regression Trees). To build the decision tree, the algorithm first creates a root node. By considering all the observations in the table, it chooses the best variable to divide up the data. To do this, it looks at all the different variables and decides which condition (for example, "Did the user read the FAQ ?") would separate the outcomes (which service the user signed up for) in a way that makes it easier to guess what the user will do.

1. Create a list `my_data` containing all the rows of the previous table as follows :
   `my_data=[['slashdot','USA','Yes',18,'None'],...]`

   Add in your code the function `divideset` below which divides the rows into two sets based on the data in a specific column. This function takes a list of rows, a column number, and a value to divide into the column. In the case of Read FAQ, the possible values are Yes or No, and for Referrer, there are several possibilities. It then returns two lists of rows : the first containing the rows where the data in the specified column matches the value, and the second containing the rows where it does not.

```
# Divides a set on a specific column. Can handle numeric
# or nominal values

def divideset(rows,column,value):
  # Make a function that tells us if a row is in
  # the first group (true) or the second group (false)
  split_function=None
  if isinstance(value,int) or isinstance(value,float):
     split_function=lambda row:row[column]>=value
  else:
     split_function=lambda row:row[column]==value

  # Divide the rows into two sets and return them
  set1=[row for row in rows if split_function(row)]
  set2=[row for row in rows if not split_function(row)]
  return (set1,set2)
```

The code creates a function to divide the data called `split_function`, which depends on knowing if the data is numerical or not. If it is, the true criterion is that the value in this column is greater than value. If the data is not numeric, `split_function` simply determines whether the column's value is the same as value. It uses this function to divide the data into two sets, one where `split_function` returns true and one where it returns false.

Hence `divideset(my_data,2,'yes')` returns the following two lists `(set1,set2)` :

```
set1=[['slashdot', 'USA', 'Yes', 18, 'None'],
      ['google', 'France', 'Yes', 23, 'Premium'],
      ['digg', 'USA', 'Yes', 24, 'Basic'] ,
      ['kiwitobes','France', 'Yes', 23, 'Basic'],
      ['slashdot', 'France', 'Yes', 19, 'None'],
      ['digg',  'Allemagne', 'Yes', 12, 'Basic'],
      ['google', 'Angleterre', 'Yes', 18, 'Basic'],
      ['kiwitobes', 'France', 'Yes', 19, 'Basic']]

set2=[['google', 'Angleterre', 'No', 21, 'Premium'],
      ['direct', 'New zeland', 'No', 12, 'None'],
      ['direct', 'Angleterre', 'No', 21, 'Basic'],
      ['google', 'USA', 'No', 24, 'Premium'],
      ['digg','USA', 'No', 18, 'None'],
      ['google', 'Angleterre', 'No', 18, 'None'],
      ['kiwitobes', 'Angleterre', 'No', 19, 'None'],
      ['slashdot','Angleterre', 'No', 21, 'None']]
```

2. Our casual observation that the chosen variable is not very good may be accurate, but to choose which variable to use in a software solution, you need a way to measure how mixed a set is. What you want to do is find the variable that creates the two sets with the least possible mixing. The first function you'll need is one to get the counts of each result in a set.

   Add the function `uniquecounts` below. It finds all the different possible outcomes and returns them as a dictionary of how many times they each appear. This is used by the other functions to calculate how mixed a set is. There are a few different metrics for measuring this, and we considered here the *Entropy* measure.

```
  # Create counts of possible results (the last column of
  # each row is the result)
  def uniquecounts(rows):
    results={}
    for row in rows:
       # The result is the last column
```

```
      r=row[len(row)-1]
      if r not in results: results[r]=0
      results[r]+=1
  return results
```

For instance, `uniquecounts(my_data)` returns the dictionary
`{'None': 7, 'Premium': 3, 'Basic': 6}`

3. Entropy, in information theory, is the amount of disorder in a set, how mixed a set is. The entropy function calculates the frequency of each item (the number of times it appears divided by the total number of rows), and applies these formulas :

$$p(i) = \mathit{frequency}(\text{outcome}) = \mathit{count}(\text{outcome})/\mathit{count}(\text{total rows})$$
$$\mathit{Entropie} = \sum_{i\in A} -p(i) \times \log_2(p(i))$$

   where A is the set of all the outcomes and $p(i)$ the frequency of the outcome i.

   By using the function `uniquecounts`, builds the function `entropie` computing the entropie of a given set of data. For instance, we have :

```
>>> entropie(set1)
1.2987949407
```

4. Now, we will create the function `buildtree`, a recursive function that builds the tree by choosing the best dividing criteria for the current set. Analyze the following program and test it with your data.

```
def buildtree(rows,scoref=entropie):
   if len(rows)==0: return decisionnode()
   current_score=scoref(rows)

   # Set up some variables to track the best criteria
   best_gain=0.0
   best_criteria=None
   best_sets=None

   column_count=len(rows[0])-1
   for col in range(0,column_count):
     # Generate the list of different values in
     # this column
     column_values={}
     for row in rows:
       column_values[row[col]]=1
     # Now try dividing the rows up for each value
     # in this column
     for value in column_values.keys():
       (set1,set2)=diviserjeu(rows,col,value)


       # Information gain
       p=float(len(set1))/len(rows)
       gain=current_score-p*scoref(set1)-(1-p)*scoref(set2)
       if gain>best_gain and len(set1)>0 and len(set2)>0:
         best_gain=gain
         best_criteria=(col,value)
         best_sets=(set1,set2)

   # Create the subbranches
   if best_gain>0:
      trueBranch=buildtree(best_sets[0])
      falseBranch=buildtree(best_sets[1])
      return decisionnode(col=best_criteria[0],value=best_criteria[1],
                          tb=trueBranch,fb=falseBranch)

    else:
      return decisionnode(results=uniquecounts(rows))
```

5. The following functions will help you to print your decision tree.

```
def printtree(tree,indent=''):
  # Is this a leaf node?
  if tree.results!=None:
    print str(tree.results)

  else:
  # Print the criteria
    print str(tree.col)+':'+str(tree.value)+'? '

    # Print the branches
    print indent+'V->',
    printtree(tree.tb,indent+'  ')
    print indent+'F->',
    printtree(tree.fb,indent+'  ')
```

We should obtain :

```
>>> printtree(my\_data,indent='')
0:google?
V-> 3:21?
  V-> {'Premium': 3}
  F-> 2:Yes?
    V-> {'Basic': 1}
    F-> {'None': 1}
F-> 0:slashdot?
  V-> {'None': 3}
  F-> 2:Yes?
    V-> {'Basic': 4}
    F-> 3:21?
      V-> {'Basic': 1}
      F-> {'None': 3}
```

6. Analyze the following python program and test it with your data.

```
def classify(observation,tree):
  if tree.results!=None:
    return tree.results

  else:
    v=observation[tree.col]
    branch=None
    if isinstance(v,int) or isinstance(v,float):
      if v>=tree.value: branch=tree.tb
      else: branch=tree.fb
    else:
      if v==tree.value: branch=tree.tb
      else: branch=tree.fb
    return classify(observation,branch)
```

When to Use Decision Trees

Probably the biggest advantage of decision trees is how easy it is to interpret a trained model. After running the algorithm on our example problem, we not only end up with a tree that can make predictions about new users, we also get the list of questions used to make those determinations. From this you can see that, for instance, users who find the site through Slashdot never become paid subscribers, but users who find the site through Google and view at least 20 pages are likely to become premium subscribers. This, in turn, might allow you to alter your advertising strategy to target sites that give you the highest quality traffic. We also learn that certain variables, such as the user's country of origin, are not important in determining the outcome. If data is difficult or expensive to collect and we learn that it is not important, we know that we can stop collecting it.

Unlike some other machine-learning algorithms, decision trees can work with both categorical and numerical data as inputs.Decision trees also allow for probabilistic assignment of data. With some problems, there is not enough information to always make a correct distinction, a decision tree may have a node that has several possibilities and can't be divided any more.

However, there are definitely drawbacks to the decision tree algorithm used here. While it can be very effective for problems with only a few possible results, it can't be used effectively on datasets with many possibilities. In our example, the only outcomes are none, basic, and premium. If there were hundreds of outcomes instead, the decision tree would grow very complicated and would probably make poor predictions.

The other big disadvantage of the decision trees described here is that while they can handle simple numerical data, they can only create greater-than/less-than decision points. This makes it difficult for decision trees to classify data where the class is determined by a more complex combination of the variables. For instance, if the results were determined by the differences of two variables, the tree would get very large and would quickly become inaccurate.

In sum, decision trees are probably not a good choice for problems with many numerical inputs and outputs, or with many complex relationships between numerical inputs, such as in interpreting financial data or image analysis. Decision trees are great for datasets with a lot of categorical data and numerical data that has breakpoints. These trees are the best choice if understanding the decision-making process is important; as you've observed, seeing the reasoning can be as important as knowing the final prediction.