Algorithm for Customer profiling

Suppose we have a list of visitors. Our **target attribute** is if they bought a subscription to our service (possible values are None, Basic or Premium). To predict their behavior in a transparent way, we will use **decision trees**.

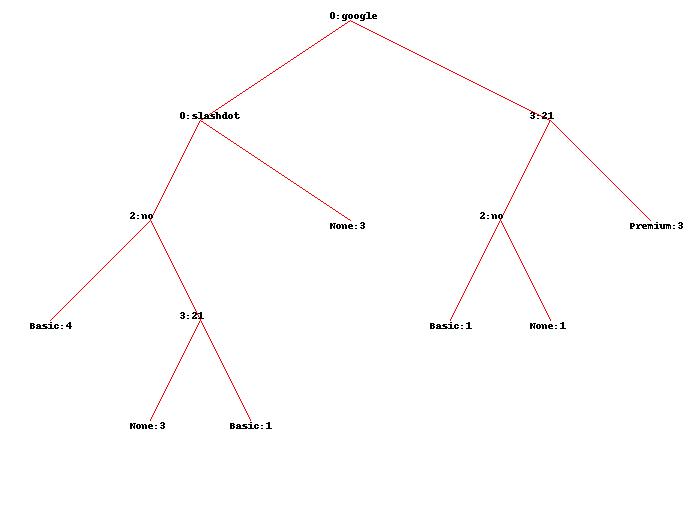
We collected data on 16 visitors. The data is represented like this :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***REFERER*** | ***COUNTRY*** | ***READ FAQ*** | ***NO. OF WEBSITES VISITED*** | ***SUBSCRIPTION(TARGET ATTRIBUTE)*** |
| Slashdot | USA | Yes | 18 | None |
| Google | France | Yes | 23 | Premium |
| Digg | USA | Yes | 24 | Basic |
| Kiwitobes | France | Yes | 23 | Basic |
| Google | UK | No | 21 | Premium |
| (direct) | New Zealand | No | 12 | None |
| (direct) | UK | No | 21 | Basic |
| Google | USA | No | 24 | Premium |
| Slashdot | France | Yes | 19 | None |
| Digg | USA | No | 18 | None |
| Google | UK | No | 18 | None |
| Kiwitobes | UK | No | 19 | None |
| Digg | New Zealand | Yes | 12 | Basic |
| Slashdot | UK | No | 21 | None |
| Google | UK | Yes | 18 | Basic |
| Kiwitobes | France | Yes | 19 | Basic |

**Using the current data, we want to build a predictive model that will take the form of a tree, as shown below. This decision tree will help us classify future observations.** For instance, according to this tree, if a new observation/visitor has Google as Referrer (1st decision node called 0:google), has read more than 21 pages (2nd decision node, on the right called 3:21), it will probably buy a Premium subscription - as already 3 previous observations have done (leaf Premium:3).

The answer to each question is "False" on the left branch, and "True" on the right branch. The first number refers to the number of the column (starting with column 0 = Referer as Python start to count with zero) that is concerned with the question.

Note that **not all** features were used to classify observations (e.g. country is not used) and some features are used multiples times (e.g. referer). Indeed, we will see that our algorithm will pick up the **best decision rules** to split groups.



Code for Profiling in python

my\_data=[['slashdot','USA','yes',18,'None'],

['google','France','yes',23,'Premium'],

['digg','USA','yes',24,'Basic'],

['kiwitobes','France','yes',23,'Basic'],

['google','UK','no',21,'Premium'],

['(direct)','New Zealand','no',12,'None'],

['(direct)','UK','no',21,'Basic'],

['google','USA','no',24,'Premium'],

['slashdot','France','yes',19,'None'],

['digg','USA','no',18,'None'],

['google','UK','no',18,'None'],

['kiwitobes','UK','no',19,'None'],

['digg','New Zealand','yes',12,'Basic'],

['slashdot','UK','no',21,'None'],

['google','UK','yes',18,'Basic'],

['kiwitobes','France','yes',19,'Basic']]

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

***# 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)

***# 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

***# Probability that a randomly placed item will***

***# be in the wrong category***

def giniimpurity(rows):

total=len(rows)

counts=uniquecounts(rows)

imp=0

for k1 in counts:

p1=float(counts[k1])/total

for k2 in counts:

if k1==k2: continue

p2=float(counts[k2])/total

imp+=p1\*p2

return imp

***# Entropy is the sum of p(x)log(p(x)) across all***

***# the different possible results***

def entropy(rows):

from math import log

log2=lambda x:log(x)/log(2)

results=uniquecounts(rows)

*# Now calculate the entropy*

ent=0.0

for r in results.keys():

p=float(results[r])/len(rows)

ent=ent-p\*log2(p)

return ent

def printtree(tree,indent=''):

if tree.results!=None:

print str(tree.results)

else:

***# Print the criteria***

print str(tree.col)+':'+str(tree.value)+'? '

***# Print the branches***

print indent+'T->',

printtree(tree.tb,indent+' ')

print indent+'F->',

printtree(tree.fb,indent+' ')

def getwidth(tree):

if tree.tb==None and tree.fb==None: return 1

return getwidth(tree.tb)+getwidth(tree.fb)

def getdepth(tree):

if tree.tb==None and tree.fb==None: return 0

return max(getdepth(tree.tb),getdepth(tree.fb))+1

from PIL import Image,ImageDraw

def drawtree(tree,jpeg='tree.jpg'):

w=getwidth(tree)\*100

h=getdepth(tree)\*100+120

img=Image.new('RGB',(w,h),(255,255,255))

draw=ImageDraw.Draw(img)

drawnode(draw,tree,w/2,20)

img.save(jpeg,'JPEG')

def drawnode(draw,tree,x,y):

if tree.results==None:

# Get the width of each branch

w1=getwidth(tree.fb)\*100

w2=getwidth(tree.tb)\*100

***# Determine the total space required by this node***

left=x-(w1+w2)/2

right=x+(w1+w2)/2

***# Draw the condition string***

draw.text((x-20,y-10),str(tree.col)+':'+str(tree.value),(0,0,0))

***# Draw links to the branches***

draw.line((x,y,left+w1/2,y+100),fill=(255,0,0))

draw.line((x,y,right-w2/2,y+100),fill=(255,0,0))

***# Draw the branch nodes***

drawnode(draw,tree.fb,left+w1/2,y+100)

drawnode(draw,tree.tb,right-w2/2,y+100)

else:

txt=' \n'.join(['%s:%d'%v for v in tree.results.items()])

draw.text((x-20,y),txt,(0,0,0))

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)

def prune(tree,mingain):

***# If the branches aren't leaves, then prune them***

if tree.tb.results==None:

prune(tree.tb,mingain)

if tree.fb.results==None:

prune(tree.fb,mingain)

***# If both the subbranches are now leaves, see if they***

***# should merged***

if tree.tb.results!=None and tree.fb.results!=None:

***# Build a combined dataset***

tb,fb=[],[]

for v,c in tree.tb.results.items():

tb+=[[v]]\*c

for v,c in tree.fb.results.items():

fb+=[[v]]\*c

***# Test the reduction in entropy***

delta=entropy(tb+fb)-(entropy(tb)+entropy(fb)/2)

if delta<mingain:

***# Merge the branches***

tree.tb,tree.fb=None,None

tree.results=uniquecounts(tb+fb)

def mdclassify(observation,tree):

if tree.results!=None:

return tree.results

else:

v=observation[tree.col]

if v==None:

tr,fr=mdclassify(observation,tree.tb),mdclassify(observation,tree.fb)

tcount=sum(tr.values())

fcount=sum(fr.values())

tw=float(tcount)/(tcount+fcount)

fw=float(fcount)/(tcount+fcount)

result={}

for k,v in tr.items(): result[k]=v\*tw

for k,v in fr.items(): result[k]=v\*fw

return result

else:

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 mdclassify(observation,branch)

def variance(rows):

if len(rows)==0: return 0

data=[float(row[len(row)-1]) for row in rows]

mean=sum(data)/len(data)

variance=sum([(d-mean)\*\*2 for d in data])/len(data)

return variance

def buildtree(rows,scoref=entropy):

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)=divideset(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 sub branches***

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))