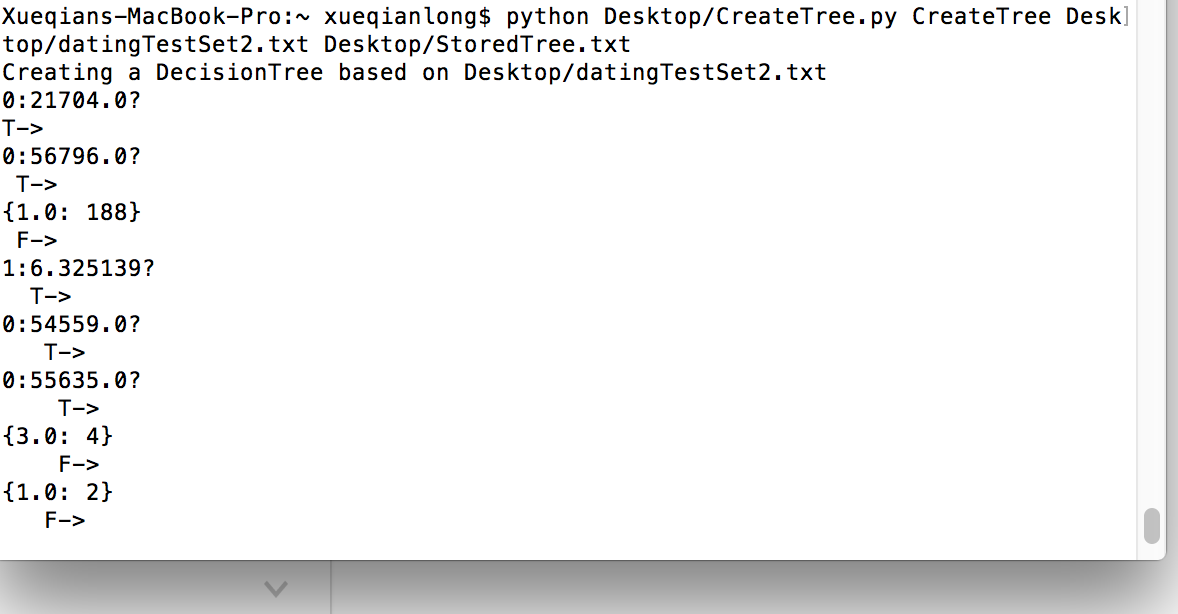
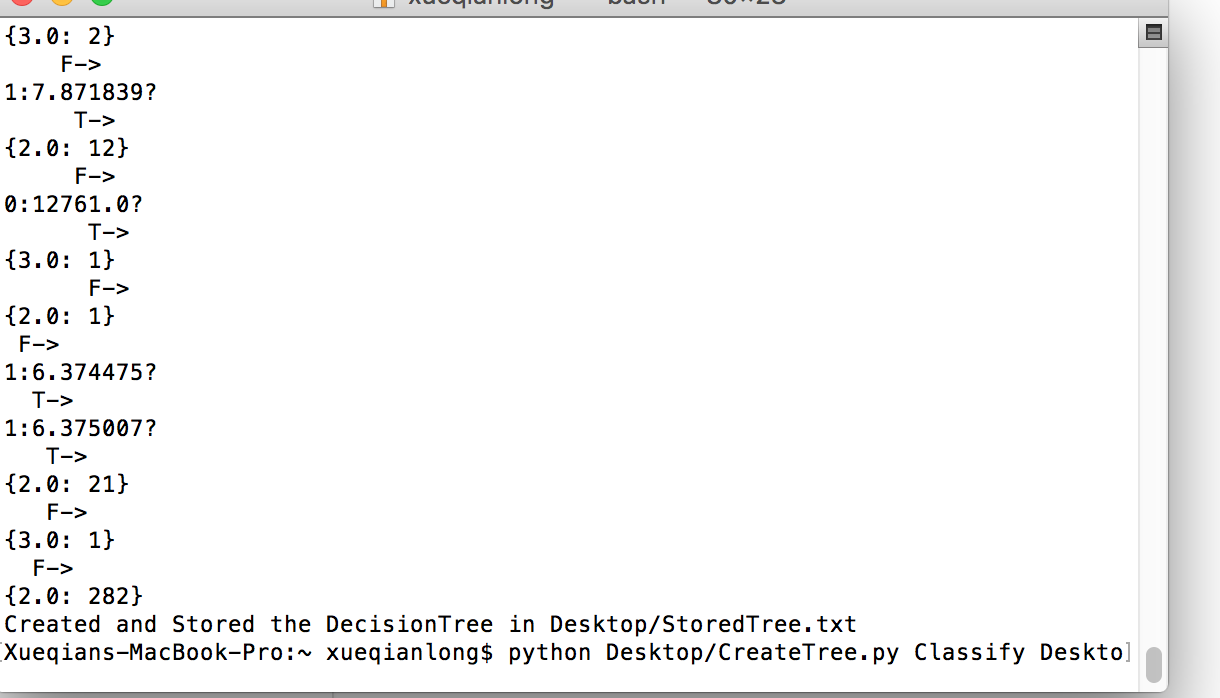
Decision Tree in Python

1. 程序运行：
2. 生成决策树：python Desktop/CreateTree.py CreateTree Desktop/datingTestSet2.txt Desktop/StoredTree.txt

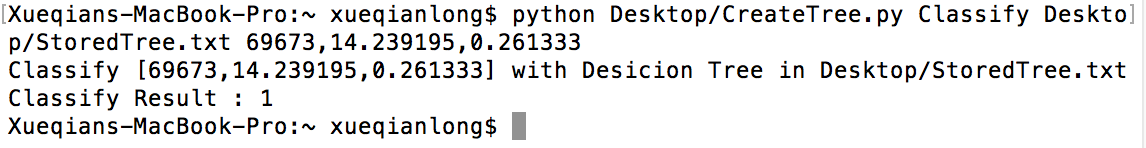
参数1为“CreateTree”时生成决策树，并将其存储于名为参数3的文件中，其中训练集为参数2文件中的数据。





1. 利用决策树分类： python Desktop/CreateTree.py Classify Desktop/StoredTree.txt 69673,14.239195,0.261333

参数1为“Classify”时利用决策树分类，决策树来自名为参数2的文件中，待分类数据为参数3



1. 决策树基本思路：

1.由决策树的根节点到叶节点的每一条路径构建一条规则；

2.路径内部结点的特征对应规则的条件；

3.叶节点的类对应规则的结论.

决策树学习算法主要由三部分构成：

1.特征选择

2.决策树生成

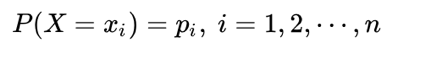
3.决策树的剪枝

1. 决策树特征――信息增益：

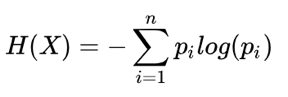
熵(entropy)

在信息论与概率论中，熵(entropy)用于表示\*\*随机变量不确定性的度量\*\*。

设X是一个有限状态的离散型随机变量，其概率分布为

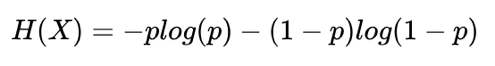


则随机变量X的熵定义为

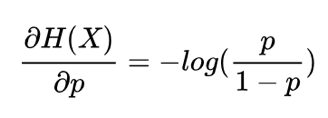


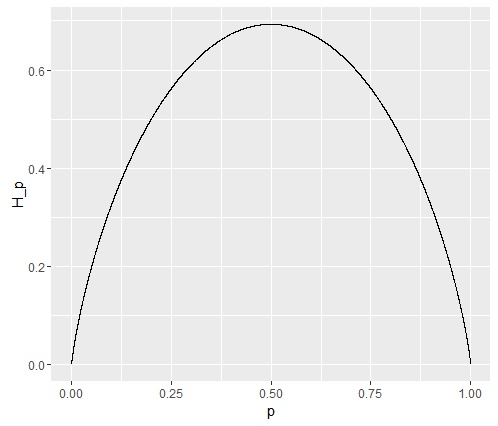
熵越大，则随机变量的不确定性越大。

当随机变量只有0,1两种取值时，假设P (X = 1) = p,则有



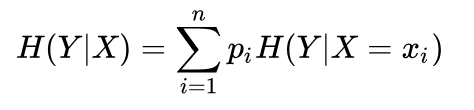
从而有，





条件熵(conditional entropy)

随机变量X给定的条件下，随机变量Y的条件熵P(Y|X)定义为：



信息增益(information gain)

信息增益表示的是：得知特征X的信息而使得类Y的信息的不确定性减少的程度。

具体定义如下。

特征A对训练数据集D的信息增益 g (D, A) 定义为集合D的经验熵 H (D) 与特征A给定条件下D的经验条件熵 H (D|A) 之差，即



一般地，熵 H (Y) 与条件熵 H (Y|X) 之差称为互信息(mutual information).

根据信息增益准则进行特征选择的方法是：对训练数据集D，计算其每个特征的信息增益，并比它们的大小，从而选择信息增益最大的特征。

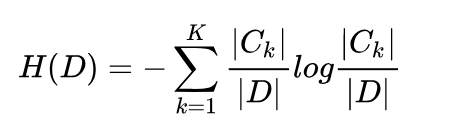
假设训练数据集为D，样本容量为|D|,有k个类别Ck , |Ck|为类别 Ck的样本个数。某一特征A有n个不同的取值a1, a2, a3…an。根据特征A的取值可将数据集D划分为n个子集D1, D2…Dn, |Di|为 Di的样本个数。并记子集Di中属于类

Ck 的样本的集合为 Dik, |Dik| 为 Dik 的样本个数。

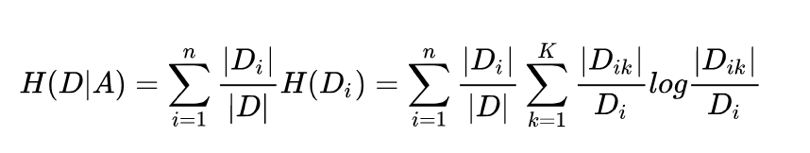
则信息增益的算法如下：

- 输入：训练数据集D和特征A；

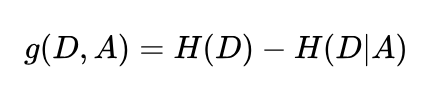
- 输出：特征A对训练数据集D的信息增益 g (D, A).

- (1) 计算数据集D的经验熵 H (D).  


- (2) 计算特征A对数据集D的经验条件熵 H (D|A).



- (3) 计算信息增益



# 对y的各种可能的取值出现的个数进行计数.。其他函数利用该函数来计算数据集和的混杂程度

def uniquecounts(rows):

results = {}

for row in rows:

#计数结果在最后一列

r = row[len(row)-1]

if r not in results:results[r] = 0

results[r]+=1

return results # 返回一个字典

# 熵

def entropy(rows):

from math import log

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

results = uniquecounts(rows)

#开始计算熵的值

ent = 0.0

for r in results.keys():

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

ent = ent - p\*log2(p)

return ent

1. 生成决策树

ID3算法

ID3算法的核心是在决策树的各个结点上应用信息增益准则进行特征选择。具体做法是：

1.从根节点开始，对结点计算所有可能特征的信息增益，选择信息增益最大的特征作为结点的特征，并由该特征的不同取值构建子节点；

2.对子节点递归地调用以上方法，构建决策树；

3.直到所有特征的信息增益均很小或者没有特征可选时为止。

#定义节点的属性

class decisionnode:

def \_\_init\_\_(self,col = -1,value = None, results = None, tb = None,fb = None):

self.col = col # col是待检验的判断条件所对应的列索引值

self.value = value # value对应于为了使结果为True，当前列必须匹配的值

self.results = results #保存的是针对当前分支的结果，它是一个字典

self.tb = tb ## desision node,对应于结果为true时，树上相对于当前节点的子树上的节点

self.fb = fb ## desision node,对应于结果为true时，树上相对于当前节点的子树上的节点

#在某一列上对数据集进行拆分。可应用于数值型或因子型变量

def divideset(rows,column,value):

#定义一个函数，判断当前数据行属于第一组还是第二组

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

# 将数据集拆分成两个集合，并返回

set1 = [row for row in rows if split\_function(row)]

set2 = [row for row in rows if not split\_function(row)]

return(set1,set2)

# 以递归方式构造树

def buildtree(rows,scoref = entropy):

if len(rows)==0 : return decisionnode()

current\_score = scoref(rows)

# 定义一些变量以记录最佳拆分条件

best\_gain = 0.0

best\_criteria = None

best\_sets = None

column\_count = len(rows[0]) - 1

for col in range(0,column\_count):

#在当前列中生成一个由不同值构成的序列

column\_values = {}

for row in rows:

column\_values[row[col]] = 1 # 初始化

#根据这一列中的每个值，尝试对数据集进行拆分

for value in column\_values.keys():

(set1,set2) = divideset(rows,col,value)

# 信息增益

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)

#创建子分支

if best\_gain>0:

trueBranch = buildtree(best\_sets[0]) #递归调用

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))

#存储

def storeTree(inputTree, filename):

import pickle

fw = open(filename, 'wb') #以二进制读写方式打开文件

pickle.dump(inputTree, fw) #pickle.dump(对象, 文件，[使用协议])。序列化对象

# 将要持久化的数据“对象”，保存到“文件”中，使用有3种，索引0为ASCII，1是旧式2进制，2是新式2进制协议，不同之处在于后者更高效一些。

#默认的话dump方法使用0做协议

fw.close() #关闭文件

# 决策树的显示

def printtree(tree,indent = ''):

# 是否是叶节点

if tree.results!=None:

print (str(tree.results))

else:

# 打印判断条件

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

#打印分支

print (indent+"T->")

printtree(tree.tb,indent+" ")

print (indent+"F->")

printtree(tree.fb,indent+" ")

# 对新的观测数据进行分类

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)

1. 决策树的减枝：

如果对训练集建立完整的决策树，会使得模型过于针对训练数据，拟合了大部分的噪声，即出现过度拟合的现象。为了避免这个问题，有两种解决的办法：

1.当熵减少的数量小于某一个阈值时，就停止分支的创建。这是一种贪心算法。

2.先创建完整的决策树，然后再尝试消除多余的节点，也就是采用减枝的方法。

方法1存在一个潜在的问题：有可能某一次分支的创建不会令熵有太大的下降，但是随后的子分支却有可能会使得熵大幅降低。因此，我们更倾向于采用剪枝的方法。

具体的算法如下：

1. 计算每个节点的经验熵；

2. 递归地从树的叶节点向上回缩，如果将某一个父节点的所有叶节点合并，能够使得其损失函数减小，则进行剪枝，将父节点变成新的叶节点；

3. 返回2，直到不能继续合并。

def prune(tree,mingain):

# 如果分支不是叶节点，则对其进行剪枝

if tree.tb.results == None:

prune(tree.tb,mingain)

if tree.fb.results == None:

prune(tree.fb,mingain)

# 如果两个子分支都是叶节点，判断是否能够合并

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

#构造合并后的数据集

tb,fb = [],[]

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

tb+=[[v]]\*c

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

fb+=[[v]]\*c

#检查熵的减少量

p = float(len(tb))/len(tb+fb)

delta = entropy(tb+fb)- p\*entropy(tb) - (1 - p)\*entropy(fb)

if delta < mingain:

# 合并分支

tree.tb,tree.fb = None,None

tree.results = uniquecounts(tb+fb)

1. 文件读写

def main():

＃创建决策树部分

if sys.argv[1] == "CreateTree":

print ("Creating a DecisionTree based on " + sys.argv[2])

f = open(sys.argv[2],"r")

line = f.readline()

my\_data = [];

＃按行读取文件，存储于list中

while line:

my\_data.append(line)

line = f.readline()

f.close()

＃删除每行数据的tab键和换行键，将每一行的string分割成一个list

for i in range (len(my\_data)):

my\_data[i] = my\_data[i][:-1]

my\_data[i] = re.split(r'\t+', my\_data[i])

for i in range(len(my\_data)):

for j in range(len(my\_data[0])):

my\_data[i][j] = float(my\_data[i][j])

myTree = buildtree(my\_data)

prune(myTree, 0.1)

printtree(myTree)

storeTree(myTree, sys.argv[3])

print ("Created and Stored the DecisionTree in " + sys.argv[3])

＃利用决策树进行分类

if sys.argv[1] == "Classify":

print (sys.argv[1] + " [" + sys.argv[3] + "] with Desicion Tree in " + sys.argv[2])

fr = open(sys.argv[2], 'rb')

myTree = pickle.load(fr)

fr.close()

＃将待分类数据存于list中

classifyArray = sys.argv[3].split(',')

for i in range(len(classifyArray)):

classifyArray[i] = float(classifyArray[i])

result = classify(classifyArray, myTree)

for k,v in result.items():

print ("Classify Result : " + str(int(k)))