"""

Cody Valle, Andrew Burke, Katreina Carpenter

Data Mining Homework 5 Program

"""

import copy

import math

import random

""" Reads a csv file and returns a table (list of lists) """

def read\_csv(filename):

import csv

the\_file = open(filename, 'r')

the\_reader = csv.reader(the\_file, dialect='excel')

table = []

for row in the\_reader:

if len(row) > 0:

table.append(row)

the\_file.close()

return table

"""

Counts occurences of an element in a column

"""

def count\_occurences(table, index, value):

count = 0

for row in table:

if row[index] == value:

count += 1

return count

"""

Gets a list of all different categorical values

"""

def get\_categories(table, index):

values = []

for row in table:

if row[index] not in values:

values.append(row[index])

return values

"""

Partitions the passed in table into k folds.

"""

def partition\_into\_folds(table, k, class\_index):

folds = [[] for \_ in range(k)] # Create disjoint empty lists

table.sort(key=lambda x: x[class\_index]) # Sort by class\_index

cur\_fold = 0

for row in table:

folds[cur\_fold].append(row)

cur\_fold = (cur\_fold + 1) % k

return folds

"""Returns the class frequencies for each attribute value:

{att\_val:[{class1: freq, class2: freq, ...}, total], ...}

"""

def attribute\_frequencies(instances, att\_index, class\_index):

# Get unique list of attribute and class values

att\_vals = get\_categories(instances, att\_index)

class\_vals = get\_categories(instances, class\_index)

# Initialize the result

result = {v: [{c: 0 for c in class\_vals}, 0] for v in att\_vals}

# Build up the frequencies

for row in instances:

label = row[class\_index]

att\_val = row[att\_index]

result[att\_val][0][label] += 1

result[att\_val][1] += 1

return result

def calc\_enew(instances, att\_index, class\_index):

# Calculate the partition stats for att\_index (see below)

freqs = attribute\_frequencies(instances, att\_index, class\_index)

# find E\_new from freqs (calc weighted avg)

E\_new = 0

for att\_val in freqs:

D\_j = float(freqs[att\_val][1])

#print D\_j

probs = [(c/D\_j) for (\_, c) in freqs[att\_val][0].items()]

#print probs

E\_D\_j = -sum([0. if p == 0.0 else p \* math.log(p,2) for p in probs])

#print E\_D\_j

E\_new += D\_j \* E\_D\_j

#print ''

return E\_new / len(instances)

"""

Calculates the least entropy of the passed in attribute values

"""

def calculate\_least\_entropy(table, indices, class\_index):

min\_ent = 20000

for index in indices:

ent = calc\_enew(table, index, class\_index)

#print 'Entropy:',ent

if ent < min\_ent:

min\_ent = ent

ret = index

return ret

"""

Creates a Decision Tree based on the passed in attributes

"""

def decision\_tree(table, attrs, class\_index):

""" Check three conditions """

# No rows

if len(table) == 0:

return (None,None)

# No attributes

if len(attrs) == 0:

l = float(len(table))

values = get\_categories(table, class\_index)

options = []

for value in values:

options.append((value, count\_occurences(table, class\_index, value) / l))

return (None, options)

# All labels are the same

labels = get\_categories(table, class\_index)

num\_max\_label = max([count\_occurences(table, class\_index, label) for label in labels])

if num\_max\_label == len(table):

return (None, [(table[0][class\_index], 1.0)])

# Calculate the smallest entropy

index = calculate\_least\_entropy(table, attrs, class\_index)

# Partition on that index

values = get\_categories(table, index)

partitions = {value : [] for value in values}

for row in table:

partitions[row[index]].append(row)

# Create a decision tree on each partition

new\_attrs = copy.deepcopy(attrs)

new\_attrs.pop(attrs.index(index))

sub\_trees = {}

for value in values:

sub\_trees[value] = decision\_tree(partitions[value], new\_attrs, class\_index)

return (index, sub\_trees)

"""

Uses the decision tree to determine the class label

"""

def decide(tree, x, class\_labels):

if tree[0] == None:

prob = random.uniform(0, 1)

options = tree[1]

for option in options:

prob -= option[1]

if prob <= 0:

return option[0]

if x[tree[0]] not in tree[1]:

return random.choice(class\_labels)

return decide(tree[1][x[tree[0]]], x, class\_labels)

"""

Creates a testing set and training set from a list of folds, on the index of the test fold.

"""

def create\_test\_and\_train\_from\_folds(folds, index):

training\_set = []

# Build a new training set excluding the current fold (index)

for j in range(len(folds)):

if index != j:

training\_set += folds[j]

return (training\_set, folds[index])

"""

Performs step 1 of the homework

"""

def step1(table):

from tabulate import tabulate

class\_labels = get\_categories(table, SURVIVED)

folds = partition\_into\_folds(copy.deepcopy(table), 10, SURVIVED)

# Confusion matrix

matrix = [[0] \* 2 for \_ in range(2)]

for i in range(10):

training\_set, test\_set = create\_test\_and\_train\_from\_folds(folds, i)

tree = decision\_tree(training\_set, [CLASS, AGE, SEX], SURVIVED)

for row in test\_set:

predicted = decide(tree, row, class\_labels)

matrix[class\_labels.index(row[SURVIVED])][class\_labels.index(predicted)] += 1

tabbed\_table = []

for i in range(len(class\_labels)):

total = sum(matrix[i]) \* 1.0

tabbed\_table.append([class\_labels[i]] +

[matrix[i][j] for j in range(len(matrix[i]))] +

[total] +

[100. \* matrix[i][i] / (total if total != 0 else 1)])

print tabulate(tabbed\_table, headers = ['SURVIVED', 'yes', 'no', 'Total', 'Recognition (%)'])

"""

Performs step 2 of the homework

"""

def step2(table):

from tabulate import tabulate

discretize\_mpg(table)

discretize\_weight(table)

class\_labels = get\_categories(table, MPG)

folds = partition\_into\_folds(copy.deepcopy(table), 10, MPG)

# Confusion matrix

matrix = [[0] \* 10 for \_ in range(10)]

for i in range(10):

training\_set, test\_set = create\_test\_and\_train\_from\_folds(folds, i)

tree = decision\_tree(copy.deepcopy(table), [CYLINDERS, WEIGHT, MODEL\_YEAR], MPG)

for row in test\_set:

predicted = decide(tree, row, class\_labels)

matrix[int(i) - 1][int(predicted) - 1] += 1

tabbed\_table = []

for i in range(10):

total = sum(matrix[i]) \* 1.0

tabbed\_table.append([str(i + 1)] +

[matrix[i][j] for j in range(len(matrix[i]))] +

[total] +

[100. \* matrix[i][i] / (total if total != 0 else 1)])

print tabulate(tabbed\_table, headers = ['MPG', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Total', 'Recognition (%)'])

"""

Gets all rules for the tree

"""

def get\_rules(tree):

if tree[0] == None:

ret = "THEN"

for label, odds in tree[1]:

ret += ' ' + label + ': ' + str(round(odds \* 100, 1)) + '%'

return [ret]

else:

loop\_rules = []

for key,sub\_tree in tree[1].iteritems():

rules = get\_rules(sub\_tree)

for rule in rules:

to\_append = rule if rule[:4] == 'THEN' else 'AND ' + rule

loop\_rules.append('attr' + str(tree[0]) + ' == ' + key + ' ' + to\_append)

return loop\_rules

"""

Prints out the rules for the decision tree

"""

def step3(table):

tree = decision\_tree(table, [CLASS, AGE, SEX], SURVIVED)

rules = get\_rules(tree)

for rule in rules:

print 'IF ' + rule

"""

Takes a table and removes every row that has an 'NA' present

"""

def clean\_auto\_data(table):

ret = []

for row in table:

found = False

for value in row:

if value == 'NA':

found = True # There is an element in this row with 'NA'

break

if not found: # If there are no elements with 'NA'

ret.append(row) # Add the row to the return table

return ret

"""

Takes an instance and returns a label based on the MPG value.

"""

def discretize\_mpg(table):

for row in table:

if float(row[MPG]) <= 13:

row[MPG] = '1'

elif float(row[MPG]) <= 15:

row[MPG] = '2'

elif float(row[MPG]) <= 17:

row[MPG] = '3'

elif float(row[MPG]) <= 20:

row[MPG] = '4'

elif float(row[MPG]) <= 24:

row[MPG] = '5'

elif float(row[MPG]) <= 27:

row[MPG] = '6'

elif float(row[MPG]) <= 31:

row[MPG] = '7'

elif float(row[MPG]) <= 37:

row[MPG] = '8'

elif float(row[MPG]) <= 45:

row[MPG] = '9'

else:

row[MPG] = '10'

"""

Converts the weight column to categories

"""

def discretize\_weight(table):

for row in table:

if float(row[WEIGHT]) <= 1999:

row[WEIGHT] = '1'

elif float(row[WEIGHT]) <= 2499:

row[WEIGHT] = '2'

elif float(row[WEIGHT]) <= 2999:

row[WEIGHT] = '3'

elif float(row[WEIGHT]) <= 3499:

row[WEIGHT] = '4'

else:

row[WEIGHT] = '5'

"""

The main function for this program

"""

def main():

# Some index references to use globally

global MPG

MPG = 0

global CYLINDERS

CYLINDERS = 1

global DISPLACEMENT

DISPLACEMENT = 2

global HORSEPOWER

HORSEPOWER = 3

global WEIGHT

WEIGHT = 4

global ACCELERATION

ACCELERATION = 5

global MODEL\_YEAR

MODEL\_YEAR = 6

global ORIGIN

ORIGIN = 7

global CAR\_NAME

CAR\_NAME = 8

global MSRP

MSRP = 9

global LABEL

LABEL = 10

global CLASS

CLASS = 0

global AGE

AGE = 1

global SEX

SEX = 2

global SURVIVED

SURVIVED = 3

auto\_table = read\_csv('auto-data.txt') # Read in the automotive data

auto\_table = clean\_auto\_data(auto\_table) # Cleans the table

titanic\_table = read\_csv('titanic.txt')[1:]

step1(copy.deepcopy(titanic\_table))

step2(copy.deepcopy(auto\_table))

step3(copy.deepcopy(titanic\_table))

if \_\_name\_\_ == '\_\_main\_\_':

main()