"""

Andrew Burke, Katreina Carpenter, Cody Valle

HW3

Performs k-Nearest Neighbor and Linear Regression algorithms on the 'auto-data.txt' dataset.

"""

import math

import copy

""" 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

""" Makes floats """

def get\_column\_as\_floats(table, index):

column = []

for row in table:

if row[index] != 'NA':

column.append(float(row[index]))

return column

"""

Takes a table and a number, then splits the table into a training set

and a test set, where the size of the test set is the number passed in.

"""

def create\_test\_and\_train(table, k):

import random

test = []

for \_ in range(k):

i = random.randint(0, len(table) - 1)

test.append(table[i])

table.pop(i)

return test, table

"""

Takes a table and normalizes the column of the passed in index

"""

def normalize\_table(table, index):

column = get\_column\_as\_floats(table, index)

mini = min(column)

maxi = max(column)

minmax = (maxi - mini) \* 1.0

for row in table: # Calculate the normalized value for each row

row[index] = (float(row[index]) - mini) / minmax

return table

"""

Takes a table and an index, calculates the minimum and

divisor used to normalize a value, then returns those.

"""

def get\_normalize\_factors(table, index):

column = get\_column\_as\_floats(table, index)

mini = min(column)

return (mini, (max(column) - mini) \* 1.0)

"""

Takes a row, an index, and the two normalizing factors for that index.

Normalizes the index value of the row, then returns it.

"""

def normalize\_row(row, index, (mini, minmax)):

row[index] = (float(row[index]) - mini) / minmax

return row

""" Gets the slope and intercept of the group of points """

def slope\_intercept(xs, ys):

ybar = sum(ys) / float(len(ys)) # average y value

xbar = sum(xs) / float(len(xs)) # average x value

nume = sum([(xs[i] - xbar)\*(ys[i] - ybar) for i in range(len(xs))])

denom = sum([(xs[i] - xbar)\*\*2 for i in range(len(xs))])

m = nume / denom

b = ybar - m\*xbar

return (m, b)

""" Gets points from the table """

def points(table, xindex, yindex):

xs = []

ys = []

for row in table:

if row[xindex] == 'NA' or row[yindex] == 'NA':

continue

xs.append(row[xindex])

ys.append(row[yindex])

xs = [float(x) for x in xs]

ys = [float(y) for y in ys]

return (xs,ys)

"""

Creates a classifier that predicts mpg values

using least squares linear regression based on

vehicle weight.

"""

def step1(table, num):

print '==========================================='

print 'STEP 1: Linear Regression MPG Classifier'

print '==========================================='

# run linear regression for mpg based on weight

(xs, ys) = points(table, WEIGHT, MPG)

(m, b) = slope\_intercept(xs, ys)

# take num random instances and put them in test

test, the\_rest = create\_test\_and\_train(table, num)

# make a copy of test to display real labels

actual = copy.deepcopy(test)

# label mpg values

for row in test:

row[MPG] = float(row[WEIGHT]) \* m + b

row += [label\_row\_on\_MPG(row)]

for row in actual:

row += [label\_row\_on\_MPG(row)]

for i in range(len(test)):

print " instance: " + str(test[i])

print " class: " + str(test[i][LABEL]) + ", actual: " + str(actual[i][LABEL])

"""

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

"""

def label\_row\_on\_MPG(instance):

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

return 1

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

return 2

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

return 3

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

return 4

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

return 5

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

return 6

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

return 7

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

return 8

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

return 9

else:

return 10

"""

Takes a training set and an instance, and returns a label

"""

def classify\_step2(training, instance, k):

import operator

neighbors = []

mpgmini, mpgminmax = get\_normalize\_factors(training, MPG)

cylmini, cylminmax = get\_normalize\_factors(training, CYLINDERS)

wgtmini, wgtminmax = get\_normalize\_factors(training, WEIGHT)

accmini, accminmax = get\_normalize\_factors(training, ACCELERATION)

def difference(row1, row2, index, mini, minmax):

return ((float(row1[index]) - mini) / minmax) - ((float(row2[index]) - mini) / minmax)

for row in training:

# Calculate Euclidean distance on normalized values

distance = 0.0

distance += difference(row, instance, MPG, mpgmini, mpgminmax) \*\* 2

distance += difference(row, instance, CYLINDERS, cylmini, cylminmax) \*\* 2

distance += difference(row, instance, WEIGHT, wgtmini, wgtminmax) \*\* 2

distance += difference(row, instance, ACCELERATION, accmini, accminmax) \*\* 2

root = math.sqrt(distance)

neighbors.append([root, row[LABEL]]) # Add the distance and label to neighbors

neighbors.sort(key=lambda x: x[0]) # Sort by root

# Determine the most occuring label

labels = {}

for i in range(k):

labels[neighbors[i][1]] = labels.get(neighbors[i][1], 0) + 1

""" max item from dictionary from: http://stackoverflow.com/questions/268272/getting-key-with-maximum-value-in-dictionary """

return max(labels.iteritems(), key=operator.itemgetter(1))[0]

"""

Creates a nearest neighbor classifier for mpg that

uses the number of cylinders, weight, acceleration

attributes to predict mpg for k=5.

"""

def step2(table):

print '==========================================='

print 'STEP 2: k=5 Nearest Neighbor MPG Classifier'

print '==========================================='

for row in table: # Label all the rows

row += [label\_row\_on\_MPG(row)]

test\_set, training\_set = create\_test\_and\_train(table, 5) # Create a test and training set from the data

for instance in test\_set:

print ' instance: ', instance

print ' class: ', classify\_step2(training\_set, instance, 5), 'actual: ', instance[LABEL]

"""

Partitions the passed in table into k folds.

"""

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

i = 0

ret = [[] for \_ in range(k)]

for row in table:

ret[i].append(row)

i = (i + 1) % k

return ret

"""

Takes a training set and an instance, then uses linear regression to

guess what the label should be for the instance.

"""

def linear\_regression(training, instance):

row = copy.deepcopy(instance) # Create a modifiable copy

# run linear regression for mpg based on weight

xs, ys = points(training, WEIGHT, MPG)

m, b = slope\_intercept(xs, ys)

row[MPG] = float(instance[WEIGHT]) \* m + b # Calculate the mpg based on weight

return label\_row\_on\_MPG(row) # Return the class label

"""

Performs the first approach of step3.

Uses random subsampling with k = 10.

"""

def step3\_approach1(table):

print "Random Subsample (k=10, 2:1 Train/Test)"

for row in table: # Label all the rows

row += [label\_row\_on\_MPG(row)]

## random subsample ##

test, training = create\_test\_and\_train(table, 10)

# find number of correct predictions / total

lr\_correct = 0.0

nn\_correct = 0.0

for row in test:

if linear\_regression(training, row) == row[LABEL]:

lr\_correct += 1

if classify\_step2(training, row, 5) == row[LABEL]:

nn\_correct += 1

lr\_accuracy = lr\_correct / len(test)

nn\_accuracy = nn\_correct / len(test)

print ' Linear Regression: accuracy = ', lr\_accuracy, ', error rate = ', 1 - lr\_accuracy

print ' k Nearest Neighbors: accuracy = ', nn\_accuracy, ' error rate = ', (1 - nn\_accuracy)

"""

Performs the second approach of step3.

Uses stratified k-fold cross validation with k = 10.

"""

def step3\_approach2(table):

print "Stratified 10-Fold Cross Validation"

for row in table: # Label all the rows

row += [label\_row\_on\_MPG(row)]

folds = partition\_into\_folds(table, 10) # Partitions the data into k folds

lr\_correct = 0.0 # Number of correct linear regression predictions

nn\_correct = 0.0 # Number of correct nearest neaighbor predictions

total = 0 # Uh, the total number of predictions

# Do the linear regression on the folds

for i in range(len(folds)):

training\_set = [] # Clean the training set

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

for j in range(len(folds)):

if i != j:

training\_set += folds[j]

# Test the current fold against the training set

for instance in folds[i]:

if linear\_regression(training\_set, copy.deepcopy(instance)) == instance[LABEL]:

lr\_correct += 1

if classify\_step2(training\_set, copy.deepcopy(instance), 5) == instance[LABEL]:

nn\_correct += 1

total += 1

# Calculate the accuracy and error rate

lr\_accuracy = lr\_correct / total

nn\_accuracy = nn\_correct / total

print ' Linear Regression: accuracy = ', lr\_accuracy, ', error rate = ', (1 - lr\_accuracy)

print ' k Nearest Neighbors: accuracy = ', nn\_accuracy, ' error rate = ', (1 - nn\_accuracy)

"""

Computes the multi-class predictive accuracy and

error rate of the two classifiers using separate

training and test sets.

"""

def step3(table):

print '==========================================='

print 'STEP 3: Predictive Accuracy'

print '==========================================='

step3\_approach1(copy.deepcopy(table))

step3\_approach2(copy.deepcopy(table))

"""

Creates confusion matrices for each classifier based

on the stratified 10-fold cross validation results.

"""

def step4(table):

from tabulate import tabulate

print '==========================================='

print 'STEP 4: Confusion Matrices'

print '==========================================='

for row in table: # Label all the rows

row += [label\_row\_on\_MPG(row)]

# Set up a 10x10 double array, with all elements initialized to 0

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

for \_ in range(len(table)):

instance = table[0] # The current test row

del table[0] # Remove the instance from the table, so that the table is the training set

predicted = linear\_regression(table, instance) - 1 # Get the predicted rank

actual = instance[LABEL] - 1 # Get the actual rank

data\_table[actual][predicted] += 1 # Increment the confusion matrix table

table.append(instance) # Add the removed row back into the table

tabbed\_table = [] # The data for passing into tabulate

for i in range(len(data\_table)):

total = sum(data\_table[i]) \* 1.0

if total == 0:

total = 1

tabbed\_table.append([i + 1] + # The first column is the actual rank

data\_table[i] +

[total] + # The second to last row is the total

[data\_table[i][i] / total \* 100]) # The last row is the recognition percent

# Print it all out

print 'Linear Regression (Stratified 10-Fold Cross Validation Results):'

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

# Set up a 10x10 double array, with all elements initialized to 0

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

for \_ in range(len(table)):

instance = table[0] # The current test row

del table[0] # Remove the instance from the table, so that the table is the training set

predicted = classify\_step2(table, instance, 5) - 1 # Get the predicted rank

actual = instance[LABEL] - 1 # Get the actual rank

data\_table[actual][predicted] += 1 # Increment the confusion matrix table

table.append(instance) # Add the removed row back into the table

tabbed\_table = [] # The data for passing into tabulate

for i in range(len(data\_table)):

total = sum(data\_table[i]) \* 1.0

if total == 0:

total = 1

tabbed\_table.append([i + 1] + # The first column is the actual rank

data\_table[i] +

[total] + # The second to last row is the total

[data\_table[i][i] / total \* 100]) # The last row is the recognition percent

# Print it all out

print ' '

print 'k = 5 Nearest Neighbor (Stratified 10-Fold Cross Validation Results):'

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

"""

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

"""

def clean\_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

""" Main function for this file """

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

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

table = clean\_data(table) # Cleans the table

# Do the work

step1(copy.deepcopy(table), 5)

step2(copy.deepcopy(table))

step3(copy.deepcopy(table))

step4(copy.deepcopy(table))

""" Entry point """

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

main()