# Import useful functions

import matplotlib.pylab as plt

from tabulate import tabulate

import numpy as np

import math

import random

# Indexes of this dataset

global OBSERVATION; OBSERVATION = 0

global YEAR; YEAR = 1

global MONTH; MONTH = 2

global DAY; DAY = 3

global LATITUDE; LATITUDE = 4

global LONGITUDE; LONGITUDE = 5

global ZONAL; ZONAL = 6

global MERIDIONAL; MERIDIONAL = 7

global HUMIDITY; HUMIDITY = 8

global AIR\_TEMP; AIR\_TEMP = 9

global SEA\_TEMP; SEA\_TEMP = 10

#########################

# Utility functions

#########################

""" 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' and row[yindex] != 'NA':

xs.append(float(row[xindex]))

ys.append(float(row[yindex]))

return (xs,ys)

""" Partitions the passed in table into k folds """

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

import random

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

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

for row in table:

folds[random.choice(range(k))].append(row)

return folds

""" 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):

return folds[index], [folds[j][i] for j in range(len(folds)) for i in range(len(folds[j])) if j != index]

""" Creates """

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

folds = partition\_into\_folds(table, k, index)

return folds[0], [folds[j][i] for j in range(len(folds)) for i in range(len(folds[j])) if j != 0]

""" Gets a list of all different categorical values """

def get\_categories(table, index):

return list(set([row[index] for row in table if row[index] != None]))

""" Counts occurences of an element in a column """

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

return len([True for row in table if row[index] == value])

""" Strips a table of all rows on an index that are None """

def strip(table, index):

return [row for row in table if row[index]]

""" Cleans the table of all rows with at least one None """

def clean(table, indices = None):

if indices == None:

indices = list(range(len(table[0])))

keep = []

rem = []

for row in table:

if None in [row[i] for i in indices]:

rem.append(row)

else:

keep.append(row)

return keep, rem

""" Performs summary statistics. Min, max, average, mode, and median. """

def summary\_statistics(table, atts, indices):

# Calculates stats for the table on the index

def stats(index):

# Counts occurences of an attribute in the table

def count(val):

return len([True for x in table if x == val])

rows = [x[index] for x in table]

att\_freqs = [(x, count(x)) for x in set(rows)]

att\_freqs.sort(key=lambda x:x[1])

mode = att\_freqs[-1][0]

table.sort(key=lambda x:x[index])

med\_index = -len(table) / 2 + len(table) - 1

med = table[med\_index][ZONAL]

if len(table) % 2 == 1:

med += table[med\_index + 1][ZONAL]

med /= 2.

return [min(rows), max(rows), float(sum(rows)) / len(table), med, mode]

# Prepare the table for printing

tab\_table = [[atts[i]] + stats(i) for i in indices]

# Print summary statistics

print 'Summary Statistics'

print tabulate(tab\_table, headers = ['Metric', 'Min', 'Max', 'Avg', 'Med', 'Mode'])

""" Creates a bootstrap set """

def bootstrap(table, number = None):

return [random.choice(table) for \_ in range(len(table) if not number else number)]

""" Partitions the passed in table into stratified k folds """

def stratify(table, k, class\_index):

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

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

for i,row in enumerate(table):

folds[i % k].append(row)

return folds

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

def test\_and\_train(folds, index):

return ([f for j in range(len(folds)) for f in folds[j] if index != j], folds[index])

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

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

"""

def attribute\_frequencies(instances, att\_index, class\_index, class\_labels = None):

# Get unique list of attribute and class values

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

if class\_labels == None:

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:

result[row[att\_index]][0][row[class\_index]] += 1

result[row[att\_index]][1] += 1

return result

""" Calculates the E\_new of the passed in instances """

def calc\_enew(instances, att\_index, class\_index, class\_labels = None):

# 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])

E\_new += D\_j \* -sum([0. if p == 0.0 else p \* math.log(p,2)

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

return E\_new / len(instances)

""" Calculates the least entropy of the passed in attribute values """

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

ret = [(calc\_enew(table, i, class\_index), i) for i in indices]

return min(ret, key=lambda x:x[0])[1]

""" Determines the best split points for the dataset using dummy threads """

def determine\_split\_points(table, index, class\_index, num\_threads = 16):

""" Does work in a thread for the determine\_split\_points\_threadpool function """

def split\_points\_worker((start, stop)):

return [(calc\_enew(table[:i], index, class\_index, labels) \* len(table[:i])

+calc\_enew(table[i:], index, class\_index, labels) \* len(table[i:]),

i) for i in range(max(1, start), stop)]

from multiprocessing.dummy import Pool as ThreadPool

import pickle

# Sort on index and get all possible labels

table.sort(key=lambda x:x[index])

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

# Set up arguments

l = float(len(table))

args = [(int(l/num\_threads \* i + .5), int(l/num\_threads \* (i+1) + .5)) for i in range(num\_threads)]

# Get the results

pool = ThreadPool(num\_threads)

results = [r for rs in pool.map(split\_points\_worker, args) for r in rs]

# Join all the threads

pool.close()

pool.join()

# Sort the results

results.sort(key=lambda x:x[1])

# Save to a file

with open(str(index) + '->' + str(class\_index) + '.pickle', 'wb') as handle:

pickle.dump(results, handle)

#########################

# Functions for plotting

#########################

""" Creates a plot of the locations of the buoys """

def buoy\_location\_plot(table, show = False):

# We modulate longitude because coordinates wrap at -180 and 180

coords = [(i[LONGITUDE] % 360, i[LATITUDE]) for i in table]

xs = []

ys = []

for coord in coords:

xs.append(coord[0])

ys.append(coord[1])

plt.figure() # Reset the figure

plt.suptitle('Buoy Coordinates', fontsize=12, fontweight='bold')

plt.grid(True) # Turn on the grid

plt.scatter(xs, ys, c='g') # Plot the data

# We have to override the actual numbers with normal data

xlabels = [120, 140, 160, 0, -160, -140, -120, -100]

plt.xticks(range(120, 270, 20), xlabels)

plt.yticks(range(-10, 10, 2))

plt.xlim(min(xs) - 5, max(xs) + 5) # Set the x limits

plt.ylim(min(ys) - 0.2, max(ys) + 0.2) # Set the y limits

plt.xlabel('Longitude')

plt.ylabel('Latitude')

if show:

plt.show()

plt.savefig('buoy\_location.pdf') # Save the figure

plt.close() # Clean up

""" Scatterplot of sea\_temp versus month """

def sea\_temp\_vs\_month\_plot(table, show = False):

coords = [(i[MONTH], i[SEA\_TEMP]) for i in table]

xs = []

ys = []

for coord in coords:

xs.append(coord[0])

ys.append(coord[1])

plt.figure() # Reset the figure

plt.suptitle('Sea Temperature vs. Month', fontsize=12, fontweight='bold')

plt.grid(True) # Turn on the grid

plt.scatter(xs, ys, c='b') # Plot the data

# We have to override the actual numbers with normal data

xlabels = ['January','February','March','April','May','June','July']

xlabels += ['August','September','October','November','December']

plt.xticks(range(1,13), xlabels,rotation=65)

plt.yticks(range(int(min(ys)), int(max(ys))))

plt.xlim(0, 13) # Set the x limits

plt.ylim(min(ys) - 3, max(ys) + 3) # Set the y limits

plt.xlabel('Month')

plt.ylabel('Sea Temperature (C)')

plt.gcf().subplots\_adjust(bottom=0.20, top = 0.90)

if show:

plt.show()

plt.savefig('sea\_temp\_vs\_month.pdf') # Save the figure

plt.close() # Clean up

""" Creates a scatter plot of sea temp vs year """

def sea\_temp\_vs\_year\_box(table, show = False):

coords = [(i[YEAR], i[SEA\_TEMP]) for i in table]

xs = []

ys = []

for coord in coords:

xs.append(coord[0])

ys.append(coord[1])

plt.figure() # Reset the figure

plt.suptitle('Sea Temperature vs. Year', fontsize=12, fontweight='bold')

plt.grid(True) # Turn on the grid

plt.scatter(xs, ys, c='b') # Plot the data

# We have to override the actual numbers with normal data

xlabels = list(set(xs))

xlabels.sort()

xlabels = ['19' + str(x) for x in xlabels]

plt.xticks(range(min(xs),max(xs) + 1),xlabels,rotation=40)

plt.yticks(range(int(min(ys)), int(max(ys))))

plt.xlim(min(xs) - 1, max(xs) + 1) # Set the x limits

plt.ylim(min(ys) - 0.4, max(ys) + 0.4) # Set the y limits

plt.xlabel('Year')

plt.ylabel('Sea Temperature (C)')

plt.gcf().subplots\_adjust(bottom=0.16, top = 0.90)

if show:

plt.show()

plt.savefig('sea\_temp\_vs\_year.pdf') # Save the figure

plt.close() # Clean up

""" Creates a scatter plot of sea temperature vs air temperature """

def sea\_temp\_vs\_air\_temp\_box(table, show = False):

coords = [(i[AIR\_TEMP], i[SEA\_TEMP]) for i in table]

xs = []

ys = []

for coord in coords:

xs.append(coord[0])

ys.append(coord[1])

plt.figure() # Reset the figure

plt.suptitle('Sea Temperature vs. Air Temperature', fontsize=12, fontweight='bold')

plt.grid(True) # Turn on the grid

plt.scatter(xs, ys, c='b') # Plot the data

# We have to override the actual numbers with normal data

xlabels = list(set([int(x) for x in xs]))

xlabels.sort()

plt.xticks(range(int(min(xs)),int(max(xs)) + 1),xlabels)

plt.yticks(range(int(min(ys)), int(max(ys))))

plt.xlim(min(xs) - 0.4, max(xs) + 0.4) # Set the x limits

plt.ylim(min(ys) - 0.4, max(ys) + 0.4) # Set the y limits

plt.xlabel('Air Temperature (C)')

plt.ylabel('Sea Temperature (C)')

plt.gcf().subplots\_adjust(bottom=0.10, top = 0.90)

if show:

plt.show()

plt.savefig('sea\_temp\_vs\_air\_temp.pdf') # Save the figure

plt.close() # Clean up

""" Creates various plots of the data """

def do\_plots(table, show = False):

buoy\_location\_plot(table, show)

sea\_temp\_vs\_month\_plot(table, show)

sea\_temp\_vs\_year\_box(table, show)

sea\_temp\_vs\_air\_temp\_box(table, show)

#########################

# Functions for classifiers

#########################

""" Creates a Decision Tree based on the passed in attributes """

def decision\_tree(table, attrs, class\_index, F = 0):

import copy

""" 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)])

""" Create node and/or subtrees """

# Calculate the smallest entropy on random F attrs

sub\_attrs = copy.deepcopy(attrs)

if F > 0:

while len(sub\_attrs) > F:

del sub\_attrs[random.randint(0, len(sub\_attrs) - 1)]

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

# Partition on that index

values = get\_categories(table, attrs.index(index))

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

for row in table:

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

# Create a decision tree on each partition

new\_attrs = copy.deepcopy(attrs)

del new\_attrs[attrs.index(index)]

sub\_trees = {}

for value in values:

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

return (attrs.index(index), sub\_trees)

""" Uses a decision tree to determine the class label """

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

import random

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]

#print x

#print ''

#print tree[0]

#print ''

#print tree[1]

#print ''

#print ''

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 bootstrap set """

def bootstrap(table, number = None):

return [random.choice(table) for \_ in range(len(table) if not number else number)]

""" Tests a tree and returns its accuracy """

def test\_tree(tree, test, class\_index, class\_labels):

total, correct = len(test), 0

for row in test:

if row[class\_index] == decide(tree, row, class\_labels):

correct += 1

return float(correct) / total

""" Creates an ensemble of decision trees """

# table is a subset of the whole table (2/3)

# attrs are the attributes to use for the decision tree

# class\_index is the index of the class to guess

# class\_labels are all the possible labels

# M is the number of trees to keep, M < N

# N is the number of trees to make

# F is the number of attributes to randomly select

def random\_forest(table, attrs, class\_index, class\_labels, M, N, F):

trees = []

for \_ in range(N):

# Bootstrap for this tree

boot = bootstrap(table)

# Create the tree

tree = decision\_tree(boot, attrs, class\_index, F)

# Check its accuracy using 20 percent of the passed in table

test = bootstrap(table, int(len(table) \* 0.2))

accuracy = test\_tree(tree, test, class\_index, class\_labels)

# Add it to the list

trees.append((tree, accuracy))

# Sort the trees by accuracy

trees.sort(key = lambda x:x[1], reverse=True)

# Get the M best trees

return [trees[i][0] for i in range(M)]

""" Makes a decision using a random forest """

def decide\_random\_forest(forest, instance, class\_labels):

from collections import Counter

guesses = [decide(tree, instance, class\_labels) for tree in forest]

return Counter(guesses).most\_common()[0][0]

""" Linear regression approach """

def linear\_regression\_approach(table):

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

print 'Classifier 1: Linear Regression Sea Temperature'

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

# Get slope and intercept for line of best fit

(xs, ys) = points(table, AIR\_TEMP, SEA\_TEMP)

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

# Test the accuracy

correct = 0

for row in table:

if row[SEA\_TEMP] == int(round(m\*row[AIR\_TEMP] + b, 0)):

correct += 1

# Print results

accuracy = float(correct) / len(table)

print "Accuracy: " + str(accuracy)

print "Error Rate: " + str(1 - accuracy)

print ''

return accuracy

""" k-NN approach """

def k\_nn\_approach(table):

from collections import Counter

import math

# Will return the most likely label

def classify(training, instance, k, indices):

neighbors = []

for row in training:

# Calculate Euclidean distance on normalized values

distance = 0.0

for i in indices:

distance += (row[i]-instance[i]) \*\* 2

neighbors.append([math.sqrt(distance), row[SEA\_TEMP]]) # Add the distance and label to neighbors

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

top\_k = [x[1] for x in neighbors[:k]]

return Counter(top\_k).most\_common()[0][0]

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

print 'Classifier 2: k-Nearest Neighbor Sea Temperature'

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

# Create folds

num\_folds = 10

folds = partition\_into\_folds(table, num\_folds, SEA\_TEMP)

indices = [HUMIDITY, ZONAL, MERIDIONAL, AIR\_TEMP]

# Get results

correct = 0

total = 0

for i in range(num\_folds):

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

tree = decision\_tree(training\_set, indices, SEA\_TEMP)

for row in test\_set:

if classify(training\_set, row, 5, indices) == row[SEA\_TEMP]:

correct += 1

total += 1

# Print results

accuracy = float(correct) / total

print "Accuracy: " + str(accuracy)

print "Error Rate: " + str(1 - accuracy)

print ''

return accuracy

""" Decision tree approach """

def decision\_tree\_approach(table):

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

print 'Classifier 3: Decision Tree Sea Temperature'

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

# Set up

k = 10

class\_labels = get\_categories(table, SEA\_TEMP)

folds = partition\_into\_folds(table[:100], k, SEA\_TEMP)

indices = [MONTH, HUMIDITY, ZONAL, MERIDIONAL, LATITUDE, LONGITUDE, AIR\_TEMP]

# Get results

correct = 0

total = 0

for i in range(k):

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

tree = decision\_tree(training\_set, indices, SEA\_TEMP)

for row in test\_set:

total += 1

if decide(tree, row, class\_labels) == row[SEA\_TEMP]:

correct += 1

# Print results

accuracy = float(correct) / total

print "Accuracy: " + str(accuracy)

print "Error Rate: " + str(1 - accuracy)

print ''

return accuracy

""" Random Forest approach """

def random\_forest\_approach(table):

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

print 'Classifier 4: Random Forest Sea Temperature'

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

# Set up

labels = get\_categories(table, SEA\_TEMP)

k = 10

indices = [MONTH, HUMIDITY, ZONAL, MERIDIONAL, LATITUDE, LONGITUDE, AIR\_TEMP]

folds = partition\_into\_folds(table, k, SEA\_TEMP)

# Make the forest

M = 21

N = 120

F = 3

# Get the results

correct = 0

total = 0

for i in range(k):

test, train = create\_test\_and\_train\_from\_folds(folds, 0)

forest = random\_forest(train, indices, SEA\_TEMP, labels, M, N, F)

for row in test:

total += 1

if decide\_random\_forest(forest, row, labels) == row[SEA\_TEMP]:

correct += 1

# Print results

accuracy = float(correct) / total

print "Accuracy: " + str(accuracy)

print "Error Rate: " + str(1 - accuracy)

print ''

return accuracy

""" Compares various classifiers """

def compare\_classifiers(table\_raw, table\_categorized, table\_normalized, att\_names):

linear\_accuracy = linear\_regression\_approach(table\_categorized)

nearest\_accuracy = k\_nn\_approach(table\_normalized)

tree\_accuracy = decision\_tree\_approach(table\_categorized)

forest\_accuracy = random\_forest\_approach(table\_categorized)

#########################

# Functions for main

#########################

""" Loads the El Nino Dataset, and returns the attributes and converted rows. """

def load\_elnino():

import csv

with open('elnino.csv', 'r') as the\_file:

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

atts = the\_reader.next()

del atts[4]

table = [tuple([int(i) for i in row[:4]] +

[float(i) for i in row[5:7]] +

[float(i) if i != '.' else None for i in row[7:12]])

for row in the\_reader if len(row) > 0]

return table, atts

""" Categorizes the values in the El Nino dataset """

def categorize(table):

table.sort(key = lambda x:x[HUMIDITY])

categorized = []

for index, row in enumerate(table):

l = [row[OBSERVATION], row[YEAR], row[MONTH], row[DAY]]

l += [round(row[LATITUDE], 0),

round(row[LONGITUDE], 0)]

l += [round(row[ZONAL], 0),

round(row[MERIDIONAL], 0)]

""" These indices were determined as the best split points using entropy """

if index <= 12700:

l += [0]

elif index <= 22404:

l += [1]

elif index <= 68315:

l += [2]

elif index <= 33514:

l += [3]

elif index <= 46139:

l += [4]

elif index <= 58407:

l += [5]

elif index <= 70118:

l += [6]

elif index <= 79184:

l += [7]

elif index <= 92999:

l += [8]

elif index <= 92999:

l += [9]

elif index <= 100131:

l += [10]

else:

l += [11]

l += [int(round(row[AIR\_TEMP], 0)),

int(round(row[SEA\_TEMP], 0))]

categorized.append(tuple(l))

return categorized

""" Normalizes the continuous data at the indices specified """

def normalize(table):

indices = [LATITUDE, LONGITUDE, ZONAL, MERIDIONAL, HUMIDITY, AIR\_TEMP]

# Get the minimums and maximums

minmax = {}

for i in indices:

mn = min(table, key=lambda x:x[i])[i]

mx = max(table, key=lambda x:x[i])[i]

minmax[i] = (mn, mx - mn)

# Normalize the rows and add to return

ret = []

for row in table:

add = []

for i in range(len(row)):

if i in indices:

add += [(row[i]-minmax[i][0]) / minmax[i][1]]

else:

add += [row[i]]

ret.append(add)

# Categorize the sea temp

for row in ret:

row[SEA\_TEMP] = int(round(row[SEA\_TEMP], 0))

return ret

"""

Main function. Loads the dataset, performs summary statistics, creates plots,

creates and compares classifiers to guess temperatures based on weather and date.

"""

def main():

# Load the data

elnino, atts = load\_elnino()

elnino\_raw, fill = clean(elnino, [YEAR, MONTH, LATITUDE, LONGITUDE, HUMIDITY, ZONAL, MERIDIONAL, AIR\_TEMP, SEA\_TEMP])

elnino\_categorized = categorize(elnino\_raw)

elnino\_normalized = normalize(elnino\_raw)

# For speed/debugging

elnino\_categorized = bootstrap(elnino\_categorized, 3000)

elnino\_normalized = bootstrap(elnino\_normalized, 3000)

# Print summary statistics

summary\_statistics(elnino\_raw, atts, [ZONAL, MERIDIONAL, HUMIDITY, AIR\_TEMP, SEA\_TEMP])

# Create plots (True to show the plots, False to just save them)

do\_plots(elnino\_raw, True)

# Run classifiers

compare\_classifiers(elnino\_raw, elnino\_categorized, elnino\_normalized, atts)

""" To make this an executable """

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

main()