John Soderstrom
CS 4342
Project 2

I used a random seed to ensure results were predictable. I tested several seeds, but had trouble finding one with a validation error under .6 on the ensemble model. Most seeds resulted in an ensemble training error of 0, but occasional increased to .1.

Output

```
Model 1
[[0, 0, 1, '+'],
 [0, 0, 1, '+'],
 [1, 1, 0, '-'],
 [1, 0, 1, '+'],
 [1, 1, 0, '-'],
 [1, 0, 0, '+'],
 [1, 1, 0, '-'],
 [1, 1, 0, '-'],
 [1, 0, 1, '+'],
 [1, 1, 0, '-']]
Tree 1
root: 1
 left child: +
 right child: -

Model 2
[[1, 0, 0, '+'],
 [0, 1, 1, '-'],
 [0, 0, 0, '+'],
 [0, 1, 0, '+'],
 [1, 0, 0, '+'],
 [0, 0, 0, '+'],
 [0, 1, 0, '+'],
 [1, 0, 1, '+'],
 [0, 1, 0, '+'],
 [0, 0, 1, '+']]
Tree 2
root: 2
 left child: +
 right child: 1
  left child: +
  right child: -

Model 3
[[0, 0, 1, '+'],
```

```
 [1, 0, 1, '+'],
 [1, 0, 0, '+'],
 [0, 1, 1, '-'],
 [1, 1, 0, '-'],
 [1, 1, 0, '-'],
 [0, 0, 0, '+'],
 [1, 1, 0, '-'],
 [1, 1, 0, '-'],
 [0, 1, 0, '+']]
Tree 3
root: 1
 left child: +
 right child: 0
  left child: 2
   left child: +
   right child: -
  right child: -

Model 4
[[0, 1, 0, '+'],
 [1, 0, 0, '+'],
 [1, 1, 0, '-'],
 [0, 0, 0, '+'],
 [1, 0, 1, '+'],
 [0, 1, 1, '-'],
 [1, 1, 0, '-'],
 [0, 0, 0, '+'],
 [1, 0, 0, '+'],
 [0, 1, 0, '+']]
Tree 4
root: 1
 left child: +
 right child: 0
  left child: 2
   left child: +
   right child: -
  right child: -

Model 5
[[1, 1, 0, '-'],
 [1, 1, 0, '-'],
 [0, 0, 0, '+'],
 [1, 1, 0, '-'],
 [1, 0, 1, '+'],
 [1, 0, 0, '+'],
 [0, 1, 1, '-'],
```

[0, 0, 0, '+'],
[0, 1, 0, '+'],
[1, 1, 0, '-']]
Tree 5
root: 1
 left child: +
 right child: 0
  left child: 2
   left child: +
   right child: -
  right child: -

Model 6
[[0, 0, 0, '+'],
[0, 1, 0, '+'],
[1, 1, 0, '-'],
[0, 1, 1, '-'],
[1, 0, 1, '+'],
[0, 0, 1, '+'],
[1, 1, 0, '-'],
[0, 0, 0, '+'],
[1, 0, 0, '+'],
[1, 1, 0, '-']]
Tree 6
root: 1
 left child: +
 right child: 0
  left child: 2
   left child: +
   right child: -
  right child: -

Model 7
[[1, 0, 1, '+'],
[0, 1, 0, '+'],
[0, 1, 1, '-'],
[1, 1, 0, '-'],
[1, 1, 0, '-'],
[0, 0, 0, '+'],
[1, 1, 0, '-'],
[1, 0, 0, '+'],
[1, 1, 0, '-'],
[0, 0, 1, '+']]
Tree 7
root: 1
 left child: +

right child: 0
  left child: 2
   left child: +
   right child: -
  right child: -

Model 8
[[1, 1, 0, '-'],
 [0, 0, 1, '+'],
 [1, 0, 0, '+'],
 [1, 0, 1, '+'],
 [1, 1, 0, '-'],
 [0, 1, 1, '-'],
 [0, 0, 0, '+'],
 [0, 1, 0, '+'],
 [1, 1, 0, '-'],
 [1, 0, 1, '+']]
Tree 8
root: 1
 left child: +
 right child: 0
  left child: 2
   left child: +
   right child: -
  right child: -

Model 9
[[1, 0, 1, '+'],
 [0, 1, 1, '-'],
 [1, 0, 0, '+'],
 [0, 0, 1, '+'],
 [1, 0, 1, '+'],
 [1, 0, 0, '+'],
 [1, 0, 0, '+'],
 [1, 1, 0, '-'],
 [1, 1, 0, '-'],
 [1, 1, 0, '-']]
Tree 9
root: 1
 left child: +
 right child: -

Model 10
[[1, 1, 0, '-'],
 [1, 0, 0, '+'],
 [0, 0, 0, '+'],

```
 [0, 0, 1, '+'],
 [1, 0, 0, '+'],
 [0, 0, 1, '+'],
 [1, 1, 0, '-'],
 [0, 1, 1, '-'],
 [1, 0, 1, '+'],
 [0, 0, 1, '+']]
Tree 10
root: 1
 left child: +
 right child: -

Ensemble training error: 0.0
Ensemble validation error: 0.6
```

Source Code

```
####################################
# Author: John Soderstrom
# Due: 5/1/2020
#
# Creates an ensemble model (composed of a given number of models)
# to classify data.
#
# Data is expected to be in the format [0, 1, ... , "+"] with n columns
#   where columns from 1 to n-1 are 0 or 1, and column n is "+" or "-".
#
# Once all models are generated, using testEnsemble(data) on the Bagging object
# will query each data point on all models and get the majority classification.
# When complete, it will return the error rate (0 - 1) of the given data
# on the ensemble model.
#
# Use print() on the bagging object to print out all trees used to
# form the ensemble and the data used to form it.
#
# Printed trees used indentation to show the depth, and label
# each node as the root, or a left/right child of its parent.
# The labels are printed with them, where a number indicated the
# column/attribute used to split the node, and a classification (+/-)
# means the line has ended and that is the result of the decision tree.
#####################################

import math, random, pprint

###
# Contains a list of decision trees used to form an ensemble model.
# Requires the original training data, number of rounds or models to form,
# and a random seed to make sure each run is predictable.
#
# Stores the data used to create each tree for easy access with them.
###
class Bagging:
   ###
   # Set random seed, store original training data, and prepare
   # to create a given number of trees from rounds.
   ###
   def __init__(self, trainingData, rounds, randomSeed):
      random.seed(randomSeed)
      self.trainingData = trainingData
      self.rounds = rounds
      self.trees = []
```

```
    self.randomData = []
    self._generateTrees()

###
# Called on initialization, creates a given number of trees.
# After each is finished, it will get the training error
# and if error is over 50% (.5), it will create a replacement.
# Successful trees are stored.
###
def _generateTrees(self):
    for i in range(self.rounds):
        error = 1
        # Repeat until given a tree with success rate over 50%.
        while error >= .5:
            randomData = self.baggingData()
            temp = Tree(self.randomData[i])
            error = temp.getError(self.trainingData)
        self.trees.append(temp)

###
# After trees are completed, test given data on all trees/ensemble model.
# Tallies a majority vote from each tree for each data point, then checks
# the vote against the actual classification.
###
def testEnsemble(self, data):
    error = 0
    # Test each point 1 by 1
    for i in data:
        numPlus = 0
        numMinus = 0
        # Tally classification results from each tree
        for j in self.trees:
            label = j.query(i)
            if label is "+":
                numPlus += 1
            else:
                numMinus += 1
        # Get the majority vote and test classification
        label = ""
        if numMinus > numPlus:
            label = "-"
        else:
            label = "+"
        if i[-1] is not label:
            error += 1
        print("Actual Classification: '{}' Ensemble Guess: '{}'".format(i[-1], label))
```

```
        # Return error rate of the model on given data
        return error / len(data)

    ###
    # Randomly select points from original training data to use in
    # each tree's creation. Uniform random distribution from bagging notes.
    ###
    def baggingData(self):
        newData = []
        num = len(self.trainingData)
        for count, i in enumerate(self.trainingData):
            index = random.randint(0, num - 1)
            newData.append(self.trainingData[index])
        self.randomData.append(newData)

    ###
    # Print model information including the randomized data used to create
    # it and the model itself.
    ###
    def print(self):
        for i in range(self.rounds):
            print("Model {}".format(i + 1))
            pprint.pprint(self.randomData[i])
            print("Tree {}".format(i + 1))
            self.trees[i].print()
            print()

###
# Given a set of training data, Tree will create a decision tree.
# Expects all lists in data to have the same length. The final column
# should be a +/- classifier. All other columns should contain 0/1.
###
class Tree:
    ###
    # Sets the root node for the tree and passes data to build.
    ###
    def __init__(self, data):
        attCols = [i for i in range(len(data[0]) - 1)]
        self.root = Node(-1, 0, None, None)
        rootLabel = self.majority(data)
        self.buildTree(self.root, data, attCols, rootLabel)

    ###
    # Build a full decision tree starting from the root node.
    # Continues each line until a stopping condition is met.
    # Tests for stopping conditions are checked through endLine.
```

```
# Narrows down list of columns to check and data as the tree splits.
###
def buildTree(self, node, data, cols, parLabel):
    # Check if this node is an end
    if self.endLine(node, data, cols, parLabel):
        return;

    # Check gain for each input column not already used
    bestCol = -1
    bestSplit = -1
    for i in cols:
        newSplit = self.branchGain(data, i)
        if newSplit > bestSplit:
            bestCol = i
            bestSplit = newSplit
    # Remove the column number for later splits
    cols.remove(bestCol)

    # Separate data in this node by value on split
    dataLeft = []
    dataRight = []
    # References to elements in data are preferred here, no changes made
    for i in data:
        # Always move values of 0 in best column to left child
        if i[bestCol] is 0:
            dataLeft.append(i)
        # Always move values of 1 in best column to right child
        else:
            dataRight.append(i)

    # Assign proper label to the current node, then create children
    node.label = bestCol
    node.gain = bestSplit
    node.left = Node(-1, 0, None, None)
    node.right = Node(-1, 0, None, None)

    # Get majority label in case a child has no data
    parLabel = self.majority(data)

    # Data references are split, column removed from further checking
    self.buildTree(node.left, dataLeft, cols, parLabel)
    self.buildTree(node.right, dataRight, cols, parLabel)

###
# Test for ways to split the current node
# If no data, use the parent's majority label
```

```
# If there is no way to split, use the current node's majority label
###
def endLine(self, node, data, cols, parLabel):
    # If there is no data, use the parent's label and return
    if len(data) is 0:
        node.label = parLabel
        return True

    # If there are attributes and columns to split on,
    # do not end the line.
    if self.testAttribute(data, cols):
        return False

    # If no columns remain to split, get the majority vote.
    # In the event of a tie, it will be "+"
    newLabel = self.majority(data)

    # Assign label to final node in the line
    node.label = newLabel
    return True

###
# Generates the gain of a node split by a given attribute
# When comparing gain values, the highest result is the best choice.
# data expects a list with at least 2 columns
# col expects an integer between 0 and length of list in data minus 1
###
def branchGain(self, data, col):
    # List 0 is left node, list 1 is right node
    # First entry in either node is "-", second is "+"
    branch = [[0, 0],
              [0, 0]]
    for i in data:
        # Set class to add for to "-" by default
        incClass = 0
        if i[-1] is "+":
            incClass = 1
        # Add to the chosen node, then by class
        branch[i[col]][incClass] += 1

    # Generate entropy for both children nodes
    childError = []
    for i in branch:
        totNode = i[0] + i[1]
        # Prevent an error when a child has no data
        if totNode is 0:
```

```
            childError.append(0)
        else:
            first = i[0] / totNode
            second = i[1] / totNode
            err = 0
            # Prevent errors when a child is pure
            if first > 0:
                err -= first * math.log(first)
            if second > 0:
                err -= second * math.log(second)
            childError.append(err)

    # Get values to weight entropy
    leftSum = sum(branch[0])
    rightSum = sum(branch[1])
    totalSum = len(data)

    # Subtracted weighted entropy from 1 to get gain for the current node on that split
    return 1 - (leftSum * childError[0] + rightSum * childError[1]) / totalSum

###
# Returns the classification with the majority of remaining data
# In event of a tie, "+" is used.
# Data is expected to have already been checked for length > 0
###
def majority(self, data):
    if len(data) is 0:
        print("Error: no data to get majority of")
        return -1
    numPlus = 0
    numMinus = 0
    for i in data:
        if i[-1] is "+":
            numPlus += 1
        else:
            numMinus += 1
    if numMinus > numPlus:
        return "-"
    else:
        return "+"

###
# Test if there are attributes and classifications to
# split data on for another step in the decision tree.
###
def testAttribute(self, data, cols):
```

```
    # Test presence of both + and -
    # Fails test if all classifications are one option
    # There is nothing to split in that case
    numPlus = 0
    numMinus = 0
    for i in data:
        if i[-1] is "+":
            numPlus += 1
        else:
            numMinus += 1
    if numPlus is 0 or numMinus is 0:
        return False

    # Test presence of an attribute to split on in all remaining columns
    # If no columns have values to split on, tell endLine to end it
    for i in cols:
        pres0 = False
        pres1 = False
        for j in data:
            if j[i] is 0:
                pres0 = True
            else:
                pres1 = True
        if pres0 and pres1:
            return True

    # Even if there are classifications to split, if no column has
    # multiple attributes, there is no way to split it.
    return False

###
# On a complete decision tree, test data on it. Returns the error
# rate from 0 to 1.
###
def getError(self, data):
    error = 0
    # Query each data point one by one and test for misclassification
    # Increment error count if it was misclassified.
    for i in data:
        label = self.query(i)
        if label is not i[-1]:
            error +=1
    # Convert error to a decimal
    return error / len(data)

###
```

```
    # Given a single data point, query the decision tree to get
    # the result, either "+" or "-"
    ###
    def query(self, data):
        currentNode = self.root
        # Continue through decision tree until at an end node
        while currentNode.label is not "+" and currentNode.label is not "-":
            # Go to left child on 0, right child on 1
            if data[currentNode.label] is 0:
                currentNode = currentNode.left
            else:
                currentNode = currentNode.right
        # Return the final result of the query
        return currentNode.label

    ###
    # Preorder print starting from the root node. Node's print
    # will travel through the remaining nodes in the tree.
    ###
    def print(self):
        self.root.print()

###
# Node class stores access to left and right child
# as well as the label the node splits on (column number),
# or +/- if the line ends.
#
# Print functionality expects starting at the root (as the
# tree class automatically does), and prints each node in
# preorder, showing the label and indenting at each depth.
###
class Node:
    ###
    # Label should be a column number or classification
    # shown in last column of the data.
    # left and right give access to children.
    ###
    def __init__(self, label, gain, left, right):
        self.label = label
        self.gain = gain
        self.left = left
        self.right = right

    ###
    # Preorder print from given starting node (called root,
    # even if it's just for a subtree) and its children.
```

```
    ###
    def print(self, loc = "root", depth = 0):
        line = "{}{}: {}".format(" " * depth, loc, self.label)
        print(line)
        if(self.left):
            self.left.print("left child", depth + 1)
        if(self.right):
            self.right.print("right child", depth + 1)

# Columns: (index = Instance) A, B, C, Class
#   A,B,C = 0 or 1, Class = +, -
training = [[0, 0, 0, "+"],
            [0, 0, 1, "+"],
            [0, 1, 0, "+"],
            [0, 1, 1, "-"],
            [1, 0, 0, "+"],
            [1, 0, 0, "+"],
            [1, 1, 0, "-"],
            [1, 0, 1, "+"],
            [1, 1, 0, "-"],
            [1, 1, 0, "-"]]
validation = [[0, 0, 0, "+"],
              [0, 1, 1, "+"],
              [1, 1, 0, "+"],
              [1, 0, 1, "-"],
              [1, 0, 0, "+"]]



results = Bagging(training, 10, 67)
results.print()
print("Ensemble training error: {}".format(results.testEnsemble(training)))
print("Ensemble validation error: {}".format(results.testEnsemble(validation)))
```