```
"""
This class provides an implementation for a decision tree.

@author Lucas Kushner
"""

class CoverageNode:
    """
    This class represents a node of a tree (data structure).

    Each node keeps track of its own name, and the links that it has.
    Each of the links contains the connecting node and its probability (weight).
    """

    def __init__(self, feature):
        self.feature = feature
        self.children = {}

    def addChild(self, binLimits, childNode):
        self.children[binLimits] = childNode

    def train(self, data, target):

        # Currently, this method will override any previous value of the leaf node
        # Could be updated to keep track of values in this node and at the end of training, use the most
common value
        if len(self.children) == 0:
            self.feature = target

        else:
            index = self.feature
            value = data[index]

            for bin in self.children:
                if bin[0] <= value and bin[1] >= value:
                    self.children[bin].train(data, target)


    def determine(self, data):

        # If the current node has no children then it is a leaf node, and therefore
        # is the predicted output for the data
        if len(self.children) == 0:
            return self.feature

        #Otherwise, recursive traverse the tree until you reach a leaf
        else:
            index = self.feature
            value = data[index]

            for bin in self.children:
                if bin[0] <= value and bin[1] >= value:
                    return self.children[bin].determine(data)


from numpy import *

def readData(startLine, endLine, numFeatures):
    """ Read the given lines of input and output data from the data file covtype.data """

    # Read in the lines of data from the file
    f = open('covtype.data', 'r')
    lines = f.readlines()[startLine : endLine]

    inputs = []
    outputs = []
```

```
    # Parse the data into integers, then choose only the desired data
    for line in lines:
        parsedValues = [int(value) for value in line.split(',')]
        inputs.append(parsedValues[0:numFeatures])
        outputs.append(parsedValues[54])

    return hstack((array(inputs), array(outputs).reshape(endLine - startLine,1)))

def calculateFeatureStatistics(data, numBins):
    """
    Calculates the probaility of a data to fall into a particular bin.

    data should be a vector containing all of the data for a particular feature.

    Returns the range of the bins, as well as
    an array of probabilities which match up with the bins.
    """

    # We use the min and max values of the data to uniformly create bins
    binSize = math.ceil((max(data) - min(data))/numBins)

    bins = []
    binCounts = []

    lastBinMin = min(data)
    for i in range(0, numBins):
        bins.append((lastBinMin, lastBinMin + binSize))
        binCounts.append(0)
        lastBinMin += binSize + 1

    for i in range(0, len(data)):
        for j in range (0, len(bins)):
            if data[i] >= bins[j][0] and data[i] <= bins[j][1]:
                binCounts[j] += 1

    binProbabilities = [double(count)/len(data) for count in binCounts]

    return bins, binProbabilities

def calculateFeatureEntropy(binProbabilities):
    """
    Calculates the total entropy of a given feature.

    To find the entropy for the entire feature, we sum up the entropy of each if its
    individual bins.
    """

    totalEntropy = 0
    for p in binProbabilities:
        totalEntropy += calculateEntropy(p)

    return totalEntropy

def calculateEntropy(p):
    """ Calculates the entropy of the given number. """

    if p != 0:
        return -p * log2(p)
    else:
        return 0

def createTree(dataTuples):
    """
    Recursively creates the decision tree.
    This method assumes that the dataTuples are sorted in descending order of feature entropy.
    """
```

```
    dataTuples = dataTuples[:]

    # If we've reached the end, then return a leaf node
    if len(dataTuples) == 0:
        return CoverageNode(0)

    tree = CoverageNode(dataTuples[0][1])
    bins = dataTuples[0][2]

    shavedTuples = dataTuples[1:len(dataTuples)]

    for bin in bins:
        tree.addChild(bin, createTree(shavedTuples))

    return tree

def calculateError(predictionResultMatrix):
    """ Calculates the percent error of the given prediction result matrix. """

    wrongAnswers = 0
    for entry in predictionResultMatrix:
        if entry != 0:
            wrongAnswers += 1

    return (float(wrongAnswers) / predictionResultMatrix.size) * 100

def run(numTrainingData, numTestData, numFeatures, numBins):
    """ Run the algorithm with the given number of data, features and bins. """

    trainingData = readData(0, numTrainingData, numFeatures)
    testData = readData(numTrainingData + 1, numTrainingData + numTestData + 1, numFeatures)

    dataTuples = []

    # Find the bins, probabilities and entropies for the different variables
    for i in range(trainingData.shape[1]-1):
        values = trainingData[:,i]
        bins, probs = calculateFeatureStatistics(values, numBins)
        entropy = calculateFeatureEntropy(probs)

        dataTuples.append((entropy, i, bins, probs))

    # Sort the list of tuples so to find the order in which they should be added in the tree
    dataTuples = sorted(dataTuples, reverse = True)

    tree = createTree(dataTuples)

    # Train the tree
    for entry in trainingData:
        tree.train(entry, entry[len(entry)-1])

    # Verify the tree will accurately predict the training data
    predictions = []
    for entry in trainingData:
        predictions.append(tree.determine(entry))

    # Every non-zero entry in the following array is a mis-prediction
    predictionResultTrainingMatrix = abs(array(predictions) - trainingData[:, trainingData.shape[1]-1])
    print  calculateError(predictionResultTrainingMatrix)


    # See how well the tree predicts the test data
    predictions = []
    for entry in testData:

        # Check if the predicted value is "None". This indicates that one of the
        # values in the input vector fell out of the range for that feature determined in training.
```

```
        pred = tree.determine(entry)
        if (pred == None):
            print entry
            pred = 100

        predictions.append(pred)

    predictionResultTestMatrix = abs(array(predictions) - testData[:, testData.shape[1]-1])
    print calculateError(predictionResultTestMatrix)


def main():

    numTrainingData = 400
    numTestData = 500
    numFeatures = 3
    numBins = 3

    for i in range(0, 15):
        for j in range(0, 10):
            print numFeatures, numBins,
            run(numTrainingData, numTestData, numFeatures, numBins)
            numFeatures += 1
        numBins += 1

    print "\nDone."

if __name__ == "__main__":
    main()
```