CS536 HW2: Decision Trees

Student Name: Xuenan Wang

NetID: xw336

---

1. Generating Decision Trees

1) K = 3, M = 5

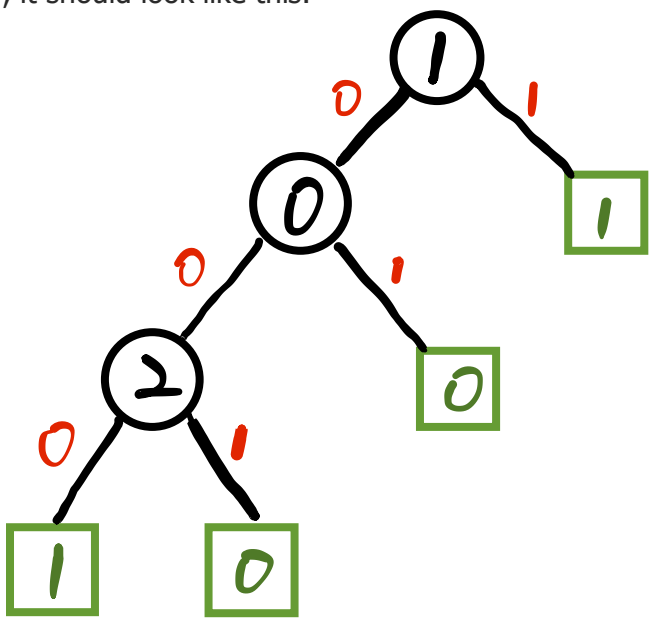
   I generated a training data set as following:

```
➜  ~ python "/Users/xuenan/GitHub/CourseWork/CS536 Machine Learning/DecisionTree.py"
data: [[0, 0, 0], [1, 1, 1], [1, 0, 0], [1, 1, 1], [0, 0, 1]]
Y: [1, 1, 0, 1, 0]
```

2) A decision tree generated based on above data using ID3:

```
--- New Level ---
feature #: 1
result: {1: 1}
--- New Level ---
feature #: 0
result: {1: 0}
--- New Level ---
feature #: 2
result: {0: 1, 1: 0}
```

   Training error should always be 0. Because the model is completely based on our training data set and for all data points in that set, the information are all included in the model correctly. Graphically, it should look like this:



3) For K = 4, M = 30, my generated data and Y are:

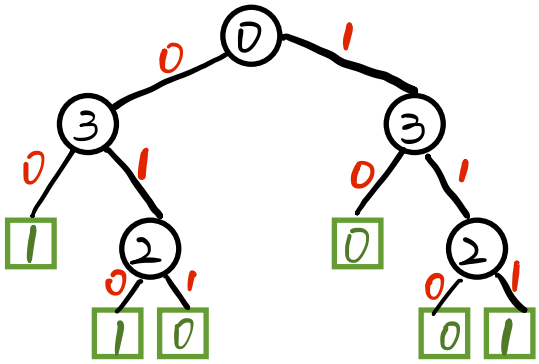
Data: [[1, 1, 0, 0], [0, 0, 0, 0], [1, 1, 1, 1], [0, 0, 0, 0], [0, 1, 1, 1],
[0, 0, 1, 0], [1, 1, 1, 1], [0, 0, 0, 0], [0, 0, 1, 0], [1, 1, 1, 0],
[0, 1, 1, 0], [1, 0, 0, 0], [0, 0, 1, 1], [0, 0, 0, 1], [0, 0, 0, 0],
[0, 0, 0, 0], [1, 1, 1, 0], [0, 0, 0, 0], [1, 0, 0, 0], [1, 1, 1, 1],
[1, 1, 1, 1], [1, 0, 0, 1], [1, 0, 1, 1], [1, 1, 1, 0], [0, 0, 0, 0],
[0, 0, 0, 1], [0, 0, 0, 0], [1, 1, 1, 0], [1, 0, 0, 0], [0, 0, 0, 0]]

Y: [0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1]

Decision Tree Structure:

```
--- New Level ---
feature #: 0
--- New Level ---
feature #: 3
result: {0: 1}
feature #: 3
result: {0: 0}
--- New Level ---
feature #: 2
result: {0: 1, 1: 0}
feature #: 2
result: {0: 0, 1: 1}
```

Graphically:



This makes sense. According to the function we define Y, the weighted average is more likely tilts high when X_0 is 1, which will make Y equals to 1. For this particular K value, the weight for $[x_1, x_2, x_3]$ is [0.36900369003690037, 0.3321033210332104, 0.2988929889298893]. Since X_{i+1} is ¾ depends on X_i, $Y = (¾ * 0.369 + ¾^2 * 0.332 + ¾^3 * 0.299)X_0$ which is $Y = 0.589X_0$. This indicate that for half of the cases, the Y should be depending on X_0.

I tried to repeat this experiment 1000 times and counted how many times for each level depends on which feature. This is what I got:

```
[[492, 199, 155, 154], [757, 61, 696, 470], [327, 109, 626, 738], [51, 0, 98, 192]]
```

Explanation of this result: For level 1, there are 492 times depends on X_0, 199 times depends on X_1, 155 times depends on X_2 and 154 times depends on X_3. Etc.

This result also proved my conclusion above. Higher levels tend to depend on X_0 and lower levels tend to depend on X_3.

4) I calculated error rate based on k = 4, trainM = 30, testM = 10. The result is as following:

```
➜  ~ python "/Users/xuenan/GitHub/CourseWork/CS536 Machine Learning/DecisionTree.py"
data: [[1, 0, 0, 0], [0, 1, 1, 1], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 1], [0, 0, 0, 1], [1, 1, 1, 0], [0, 0, 1,
 1], [1, 1, 1, 1], [1, 1, 1, 1], [0, 0, 1, 1], [0, 0, 1, 1], [1, 0, 0, 0], [1, 0, 0, 0], [0, 1, 0, 1], [0, 0, 0, 0
], [1, 1, 1, 0], [0, 1, 1, 1], [0, 0, 0, 0], [1, 0, 0, 0], [1, 1, 1, 1], [0, 1, 1, 1], [0, 0, 0, 0], [0, 1, 1, 1],
 [0, 0, 1, 1], [0, 0, 0, 0], [1, 1, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [1, 1, 1, 1]]
Y: [0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1]
Decision tree model trained with k = 4 , m = 30
--- New Level ---
feature #: 2
--- New Level ---
feature #: 0
result: {0: 1, 1: 0}
feature #: 0
--- New Level ---
feature #: 3
result: {0: 1, 1: 0}
feature #: 3
result: {0: 0, 1: 1}
Error rate: 0.0
```

The error rate I got is 0, which makes sense because my data dimension is 4 and there are maximum 16 combinations. I repeated this process for 20 times and this is what I got:

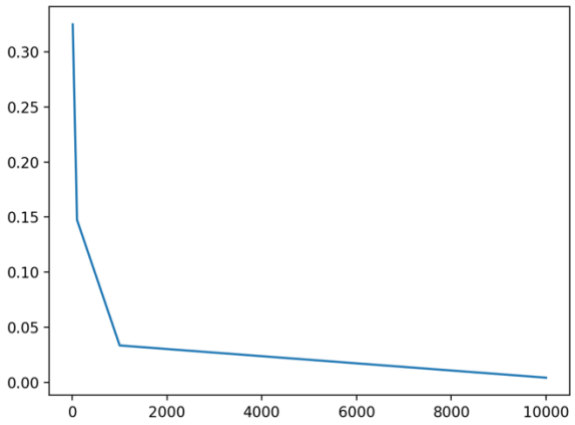
```
Error rate: 0.3
Error rate: 0.1
Error rate: 0.0
Error rate: 0.0
Error rate: 0.2
Error rate: 0.2
Error rate: 0.1
Error rate: 0.1
Error rate: 0.0
Error rate: 0.3
Error rate: 0.0
Error rate: 0.0
Error rate: 0.1
Error rate: 0.1
Error rate: 0.1
Error rate: 0.0
Error rate: 0.1
Error rate: 0.2
Error rate: 0.1
Error rate: 0.1
```

The average error rate is 0.105, which indicate that the previous result of 0 is reasonable.

5) I consider there are totalM data in a data set, and I separate these data into training data set(80%) and test data set(20%). The data size I choose is [10, 100, 1000, 10000]. For each data size, I run the model for 20 times and take average error rate, and the results are as following:

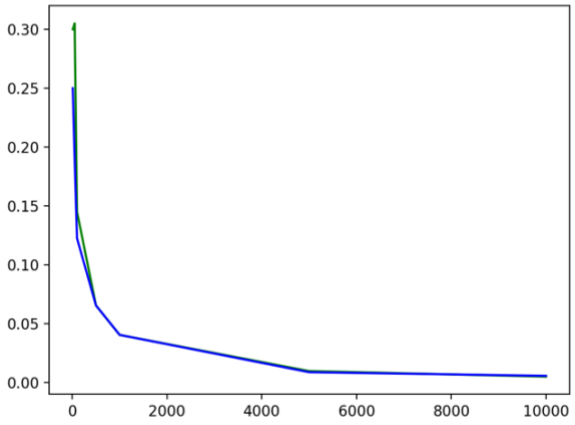
```
→  ~ python "/Users/xuenan/GitHub/CourseWork/CS536 Machine Learning/DecisionTree.py"
Decision tree model trained with k = 10 , m = 8
Decision tree model trained with k = 10 , m = 80
Decision tree model trained with k = 10 , m = 800
Decision tree model trained with k = 10 , m = 8000
totalM: [10, 100, 1000, 10000]
error: [0.325, 0.14750000000000002, 0.03350000000000001, 0.004275000000000001]
```

To plot this out:

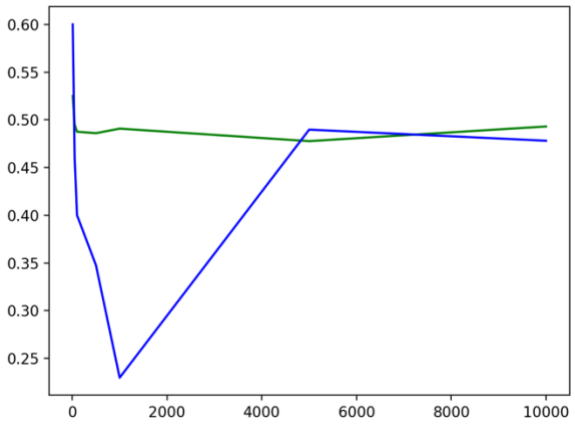


As the result indicates, the more data we have, the lower error rate we get. And after some point, for this experiment looks like about 10000, the error rate are decreasing but decreasing slowly and we can be satisfied with this result.

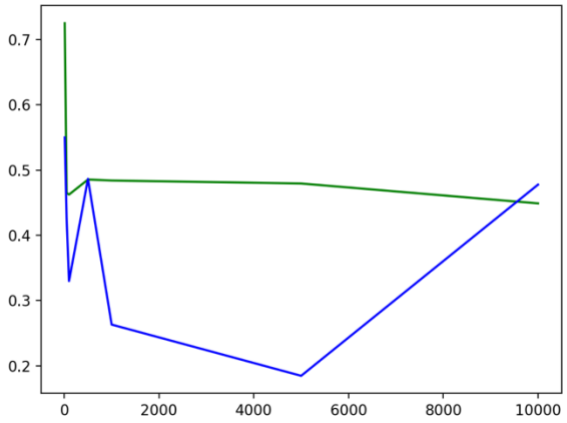
6) Instead of calculating information gains, I focused on how vary a feature can be. The variety of a feature is calculated by abs(#zeros-#ones). And when separating matrix, we choose the minimum variety of features. By doing this, we are actually choosing the features with basically similar number of zeros and ones. In this way, we can see that the matrixes after separated will be evenly distributed. The result are as following.



The **green line** is the model using ID3 algorithm and the **blue line** is the one trained with variety I mentioned above. We can see that with k = 10 and m in range of [10-10000], the performance are very closed.
I tried to expand this experiment with bigger m. I then set m as [10, 50, 100, 500, 1000, 5000, 10000]. Following is my result.



We can see that for m = 1000, the error rate is significantly low. I repeated this again.



We can see that for this time, m = 1000 and m = 5000 are significantly low. I don't understand this result, however, this may indicates some mathematical reasons I didn't find.

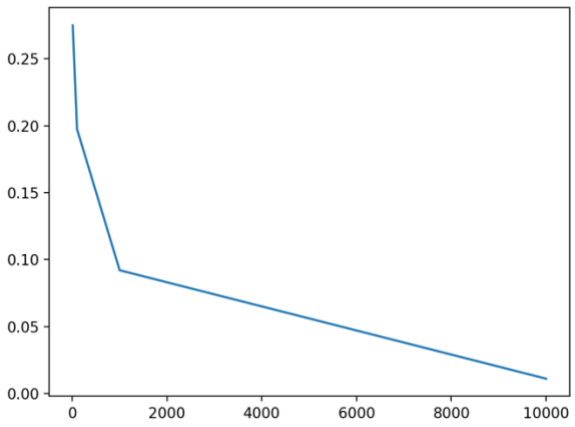
2. Pruning Decision Trees

1) For the new data scheme, here is a example when m = 1.

```
➜  ~ python "/Users/xuenan/GitHub/CourseWork/CS536 Machine Learning/DecisionTree.py"
Data for Q2: [[0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1]]
Y for Q2: [1]
```

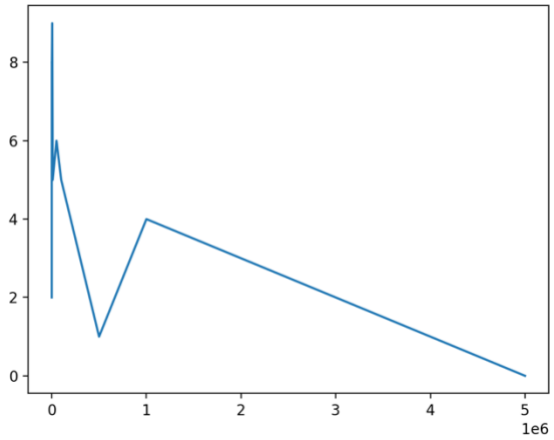
Similar with 5) in question #1, following are my results:

```
➜  ~ python "/Users/xuenan/GitHub/CourseWork/CS536 Machine Learning/DecisionTree.py"
Decision tree model trained with k = 21 , m = 8
Decision tree model trained with k = 21 , m = 80
Decision tree model trained with k = 21 , m = 800
Decision tree model trained with k = 21 , m = 8000
totalM: [10, 100, 1000, 10000]
error: [0.275, 0.1975, 0.09200000000000001, 0.011000000000000005]
```

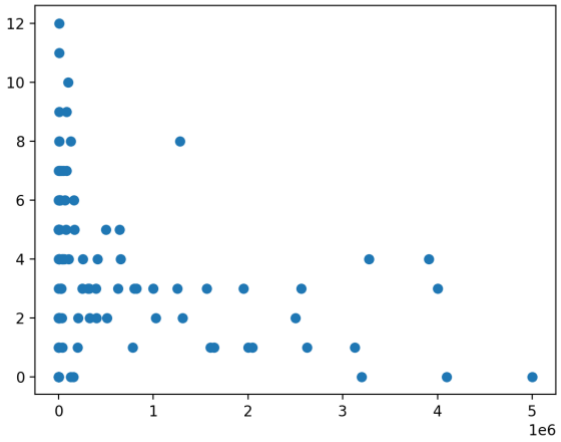


We can see that with M increase, error decrease. This meet our expectation.

2) I set m as [50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000]. Randomly generate a decision tree using that m and count how many noise(X_{15}-X_{20}) does it have in the tree. Repeat 5 time of this process for each m. Following is what I got.



We can see that when m is bigger than 500000, my decision tree doesn't have noise. However, since the decision tree is random generated, we need more sample in order to come to a conclusion. I redesign the experiment as following: keep m in a minimal heap(AKA priority queue), starting from m = 5, for each m, add [m*2, m*5, m*10] to the heap. For each step, we generate a decision tree based on m that is on the top of the heap(min m). If the m is calculated, I skip this. After I found 5 decision trees that have 0 noise when m >= 10000(if m is too small then we can easily have decision tree without noise). And the figure below is the result I got.

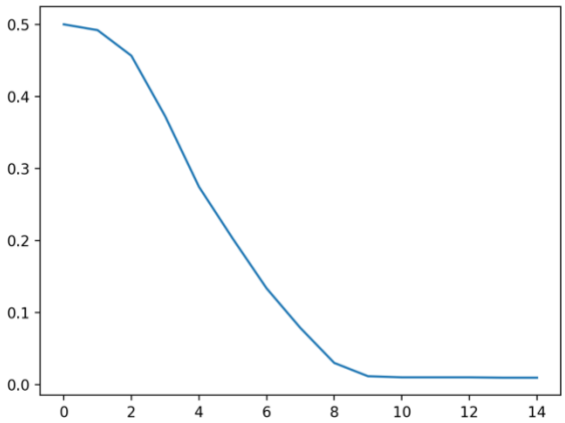


We can see that approximately with m bigger than 3000000, a lots of decision tree will have tendency to have zero noise. From this, I would say that to avoid fitting on this noise, I will try to pick m bigger than 5000000.

3) I generated a data set using m = 10000, and make trainM = 8000, testM = 2000.

   a) Pruning by Depth:

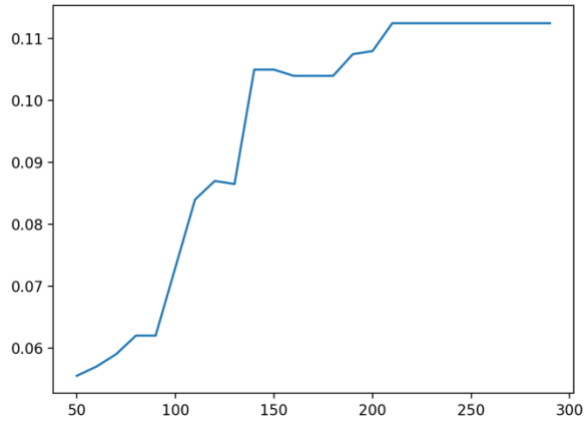
      I trained a decision tree using ID3 algorithm with data size of 8000. Then use test data to travel this tree level by level. For each level, I consider how many results are below this level and predict a result based on probability of ones and zeros. I think for level = 0, there are no useful information for prediction. Therefore the probability should be ½. This is the result I got.



      We can see from the figure that for depth bigger than 8, we have very low error rate. That should be a threshold for me to pruning decision tree by depth.

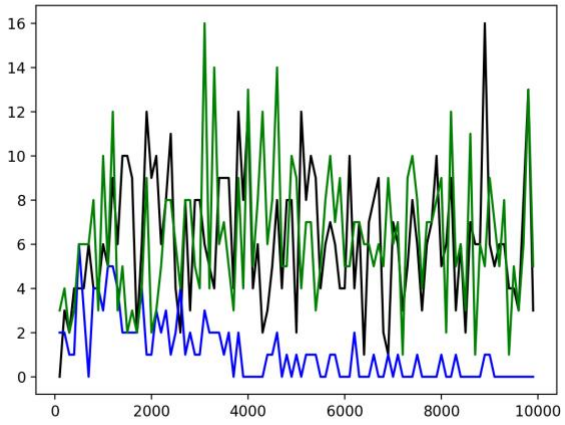
   b) Pruning by Sample Size:

      I set a threshold for data size during training process that if the remaining data is less than this threshold, I would stop the training and predict result based on majority vote. I tested this over multiple thresholds and this is the result.



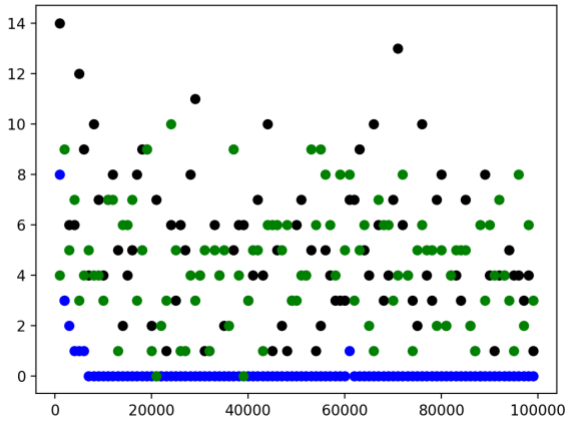
As we can see, for threshold less than 100, the error rate is considered much lower than others. Therefore, the threshold for pruning decision tree by size should be 100(or even smaller if we can put up with the additional computational resources comes along).

4) I compute pruning by depth and pruning by size together. And I plot the result with original model. All three models are using same training data and same testing data. The result is as following.



The **black line** is the original model, **blue line** is the model pruned by depth and **green line** is pruned by size. We can see that with decision tree pruned with depth of 8, the prediction error is significantly reduced. This make sense, because the noise is more likely appear in the lower level of my decision tree considering those noise is less likely linked with output. Therefore, by cutting out the lower levels, the result is more likely to be accurate.

I repeated this experiment again and this time I plotted it as scatter figure.



For this time, it's clearer that pruning by depth will give out better results.

And for pruning by size, we don't really see a significant difference between original model and pruned model.

Source Code (Python 3)

```
from random import randint
from math import log2
import matplotlib.pyplot as plt
import heapq

class Data:
    def __init__(self, k, m):
        # number of features
        self.k = k
        if self.k < 2:
            print('ERROR: k must be larger or equal to 2!')
            return
        # number of data points
        self.m = m
        self.data = []
        self.Y = []
        for _ in range(m):
            self.data.append(self.generateData())
            self.Y.append(self.calculateY(self.data[-1]))
        self.data2 = []
        self.Y2 = []
        for _ in range(m):
            self.data2.append(self.generateData2())
            self.Y2.append(self.calculateY2(self.data2[-1]))

    def generateData(self):

        def getNxtBinaryValue(pre):
            dic = {1:pre, 2:pre, 3:pre, 4:1-pre}
            return dic[randint(1, 4)]

        data = [randint(0, 1)]
```

```
    for _ in range(self.k-1):
        data.append(getNxtBinaryValue(data[-1]))
    return data

def generateData2(self):

    def getNxtBinaryValue(pre):
        dic = {1:pre, 2:pre, 3:pre, 4:1-pre}
        return dic[randint(1, 4)]

    data = [randint(0, 1)]
    for _ in range(14):
        data.append(getNxtBinaryValue(data[-1]))
    for _ in range(6):
        data.append(randint(0, 1))
    return data

def calculateY(self, X):
    sumW = sum([0.9**i for i in range(2, self.k+1)])
    # print('sumW:', sumW)
    w = [0.9**i/sumW for i in range(2, self.k+1)]
    # print('w:', w)
    weightedAvg = sum([w[i]*X[i+2] for i in range(self.k-2)])
    # print('weightedAvg:', weightedAvg)
    return X[0] if weightedAvg >= 0.5 else 1-X[0]

def calculateY2(self, X):

    def majority(xx):
        ctr = [0, 0]
        for elem in xx:
            ctr[elem] += 1
        return 0 if ctr[0] >= ctr[1] else 1

    return majority(X[1:8]) if X[0] == 0 else majority(X[8:15])
```

```
class TreeNode:
    def __init__(self, measurement=None):
        self.measurement = measurement
        self.result = {}
        self.branch = {}
        self.levelPrediction = [None, None]
        self.isEnd = False


class DecisionTree:
    def __init__(self, data, Y, size = 0, trainByLevel = False, trainBySize = False, trainByDiff = False):
        self.data = data
        self.Y = Y
        # print('data:', self.data)
        # print('Y:', self.Y)
        self.sizeThreshold = size
        if trainByLevel:
            self.root = self.ID3ByLevel(TreeNode(), self.data, self.Y, list(range(len(data[0]))), 1)
        elif trainBySize:
            self.root = self.ID3BySize(TreeNode(), self.data, self.Y, list(range(len(data[0]))))
        elif trainByDiff:
            self.root = self.trainByDiff(TreeNode(), self.data, self.Y, list(range(len(data[0]))))
        else:
            self.root = self.ID3(TreeNode(), self.data, self.Y, list(range(len(data[0]))))
        # print('Decision tree model trained with k =', len(data[0]), ', m =', len(data))
        self.level = []
        self.height = self.setTreeHeight()

    def setSizeThreshold(self, threshold):
        self.sizeThreshold = threshold

    def calcIG(self, data, Y, features):
        ig = []
        ys = [sum(Y), len(Y)-sum(Y)]
        # Try to avoid 'math domain error'
        yEntropy = 0
```

```
    for y in ys:
        if y > 0:
            yEntropy += (y/len(Y))*log2(y/len(Y))
    yEntropy *= -1
    # yEntropy = ((yOnes/len(Y))*log2(yOnes/len(Y)) + (yZeros/len(Y))*log2(yZeros/len(Y))) * -1
    for j in range(len(data[0])):
        if j not in features:
            ig.append(0)
            continue
        featureVal = [[],[]]
        entropy = []
        for i in range(len(data)):
            # print('i:', i, ', j:', j)
            featureVal[data[i][j]].append(i)
        for vals in featureVal:
            valY = [0, 0]
            for v in vals:
                valY[Y[v]] += 1
            # Try to avoid 'math domain error'
            curEntropy = 0
            for valYElem in valY:
                if valYElem > 0:
                    curEntropy += (valYElem/len(vals)) * log2(valYElem/len(vals))
            curEntropy *= -1
            # curEntropy = ((valY[0]/len(vals))*log2(valY[0]/len(vals)) + (valY[1]/len(vals))*log2(valY[1]/len(vals))) * -1
            entropy.append(curEntropy)
        curIG = yEntropy - ((len(featureVal[0])/len(data))*entropy[0] + (len(featureVal[1])/len(data))*entropy[1])
        ig.append(curIG)
    return ig

# need to rewrite
def separateData(self, data, Y, pivot):
    dataZeros = []
    dataOnes = []
    yZeros = []
    yOnes = []
```

```
    for i in range(len(data)):
        if data[i][pivot] == 0:
            dataZeros.append(data[i])
            yZeros.append(Y[i])
        else:
            dataOnes.append(data[i])
            yOnes.append(Y[i])
    return dataZeros, yZeros, dataOnes, yOnes

def allSame(self, Y):
    if not Y: return True
    for i in range(1, len(Y)):
        if Y[0] != Y[i]:
            return False
    return True

def getDiff(self, data, index):
    zeros = 0
    ones = 0
    for i in range(len(data)):
        for j in range(len(data[0])):
            if j == index:
                if data[i][j] == 0:
                    zeros += 1
                else:
                    ones += 1
    return zeros, ones

def calcDifference(self, data, features):
    import sys
    array = [sys.maxsize for _ in range(len(data[0]))]
    # print('array:', array)
    # print('features:', features)
    for f in features:
        zeros, ones = self.getDiff(data, f)
        array[f] = abs(zeros-ones)
```

```
    return array.index(min(array))

def trainByDiff(self, node, data, Y, features):
    pivot = self.calcDifference(data, features)
    node.measurement = pivot
    dataZeros, yZeros, dataOnes, yOnes = self.separateData(data, Y, pivot)
    nxtFeatures = features[:]
    nxtFeatures.remove(pivot)
    node.levelPrediction[0] = len(Y) - sum(Y)
    node.levelPrediction[1] = sum(Y)
    if len(data) <= self.sizeThreshold:
        node.isEnd = True
    # print('node.measurement:', node.measurement, 'Y:', Y)
    if not yZeros or self.allSame(yZeros):
        node.result[0] = yZeros[0]
    else:
        node.branch[0] = self.ID3(TreeNode(), dataZeros, yZeros, nxtFeatures)
    if not yOnes or self.allSame(yOnes):
        node.result[1] = yOnes[0]
    else:
        node.branch[1] = self.ID3(TreeNode(), dataOnes, yOnes, nxtFeatures)
    return node

def ID3(self, node, data, Y, features):
    ig = self.calcIG(data, Y, features)
    pivot = ig.index(max(ig))
    node.measurement = pivot
    dataZeros, yZeros, dataOnes, yOnes = self.separateData(data, Y, pivot)
    nxtFeatures = features[:]
    nxtFeatures.remove(pivot)
    node.levelPrediction[0] = len(Y) - sum(Y)
    node.levelPrediction[1] = sum(Y)
    if len(data) <= self.sizeThreshold:
        node.isEnd = True
    # print('node.measurement:', node.measurement, 'Y:', Y)
    if not yZeros or self.allSame(yZeros):
```

```
            node.result[0] = yZeros[0]
        else:
            node.branch[0] = self.ID3(TreeNode(), dataZeros, yZeros, nxtFeatures)
        if not yOnes or self.allSame(yOnes):
            node.result[1] = yOnes[0]
        else:
            node.branch[1] = self.ID3(TreeNode(), dataOnes, yOnes, nxtFeatures)
        return node

    def ID3BySize(self, node, data, Y, features):
        if len(data) <= self.sizeThreshold:
            return node
        ig = self.calcIG(data, Y, features)
        pivot = ig.index(max(ig))
        node.measurement = pivot
        dataZeros, yZeros, dataOnes, yOnes = self.separateData(data, Y, pivot)
        nxtFeatures = features[:]
        nxtFeatures.remove(pivot)
        node.levelPrediction[0] = len(Y) - sum(Y)
        node.levelPrediction[1] = sum(Y)
        if len(data) <= self.sizeThreshold:
            node.isEnd = True
        # print('node.measurement:', node.measurement, 'Y:', Y)
        if not yZeros or self.allSame(yZeros):
            node.result[0] = yZeros[0]
        else:
            node.branch[0] = self.ID3(TreeNode(), dataZeros, yZeros, nxtFeatures)
        if not yOnes or self.allSame(yOnes):
            node.result[1] = yOnes[0]
        else:
            node.branch[1] = self.ID3(TreeNode(), dataOnes, yOnes, nxtFeatures)
        return node

    def ID3ByLevel(self, node, data, Y, features, level):
        if level >= 8:
            return node
```

```
    ig = self.calcIG(data, Y, features)
    pivot = ig.index(max(ig))
    node.measurement = pivot
    dataZeros, yZeros, dataOnes, yOnes = self.separateData(data, Y, pivot)
    nxtFeatures = features[:]
    nxtFeatures.remove(pivot)
    node.levelPrediction[0] = len(Y) - sum(Y)
    node.levelPrediction[1] = sum(Y)
    if len(data) <= self.sizeThreshold:
        node.isEnd = True
    # print('node.measurement:', node.measurement, 'Y:', Y)
    if not yZeros or self.allSame(yZeros):
        node.result[0] = yZeros[0]
    else:
        node.branch[0] = self.ID3ByLevel(TreeNode(), dataZeros, yZeros, nxtFeatures, level+1)
    if not yOnes or self.allSame(yOnes):
        node.result[1] = yOnes[0]
    else:
        node.branch[1] = self.ID3ByLevel(TreeNode(), dataOnes, yOnes, nxtFeatures, level+1)
    return node

def predictByProbability(self, probability):
    sample = randint(1, sum(probability))
    return 0 if sample <= probability[0] else 1

def printTree(self):
    queue = [[self.root.measurement, self.root]]
    while queue:
        tmp = []
        print('--- New Level ---')
        curLevel = []
        for name, node in queue:
            print('feature #:', name)
            curLevel.append(name)
            print('node level prediction:', node.levelPrediction)
            print('node isEnd:', node.isEnd)
```

```
            if node.result:
                print('result:', node.result)
            if node.branch:
                for i in node.branch.keys():
                    tmp.append([node.branch[i].measurement, node.branch[i]])
        self.level.append(curLevel)
        queue = tmp

def setTreeHeight(self):
    queue = [self.root]
    h = 0
    while queue:
        tmp = []
        h += 1
        for node in queue:
            if node.branch:
                for i in node.branch.keys():
                    tmp.append(node.branch[i])
        queue = tmp
    return h

def predict(self, testData):
    if not self.root:
        print('ERROR: Train the model first!')
        return
    else:
        node = self.root
        while node:
            cur = testData[node.measurement]
            if cur in node.result:
                return node.result[cur]
            else:
                node = node.branch[cur]

def predictByLevel(self, testData):
    print('testdata:', testData)
```

```
    if not self.root:
        print('ERROR: Train the model first!')
        return
    else:
        prediction = []
        node = self.root
        while node:
            print('node.measurement:', node.measurement)
            cur = testData[node.measurement]
            if cur in node.result:
                prediction.append(node.result[cur])
                break
            else:
                prediction.append(self.predictByProbability(node.levelPrediction))
                node = node.branch[cur]
        if len(prediction) < self.height:
            lp = len(prediction)
            for _ in range(self.height-lp):
                prediction.append(prediction[-1])
        return prediction

def predictBySize(self, testData):
    if not self.root or self.sizeThreshold == 0:
        print('ERROR: Train the model first!')
        return
    else:
        node = self.root
        while node:
            cur = testData[node.measurement]
            if cur in node.result:
                return node.result[cur]
            elif node.isEnd:
                if node.levelPrediction[0] < node.levelPrediction[1]:
                    return 1
                elif node.levelPrediction[0] > node.levelPrediction[1]:
                    return 0
```

```
                else:
                    return randint(0, 1)
            else:
                node = node.branch[cur]

    def calcError(self, data, Y):
        errorCtr = 0
        for i in range(len(data)):
            if self.predict(data[i]) != Y[i]:
                errorCtr += 1
        return errorCtr / len(data)

    def countNoise(self):
        queue = [self.root]
        numNoise = 0
        noise = {15, 16, 17, 18, 19, 20}
        while queue:
            tmp = []
            for node in queue:
                if node.measurement in noise:
                    numNoise += 1
                if node.branch:
                    for i in node.branch.keys():
                        tmp.append(node.branch[i])
            queue = tmp
        return numNoise


if __name__ == '__main__':
    # # Question 1.1
    # k = 10
    # trainM = 80
    # trainData = Data(k, trainM)

    # # Question 1.2
    # # expected IG: [0.01997309402197489, 0.4199730940219749, 0.01997309402197489]
```

```
# # data = [[0, 0, 0], [1, 1, 1], [1, 0, 0], [1, 1, 1], [0, 0, 1]]
# # Y = [1, 1, 0, 1, 0]
# model = DecisionTree(trainData.data, trainData.Y)
# # print(model.calcIG(data, Y))
# # print(model.separateData(data, Y, 1))
# # print(model.ID3(TreeNode(), data, Y, [0,1,2]))

# # Question 1.3
# model.printTree()
# testM = 20
# testData = Data(k, testM)
# # print(model.predict([1,1,1,1]))
# # for _ in range(20):
# #     testData = Data(k, testM)
# #     print('Error rate:', model.calcError(testData.data, testData.Y))

# # Question 1.4
# levelCtr = [[0 for i in range(4)] for j in range(4)]
# for _ in range(1000):
#     trainData = Data(k, m)
#     model = DecisionTree(trainData.data, trainData.Y)
#     model.printTree()
#     for i in range(len(model.level)):
#         for j in model.level[i]:
#             levelCtr[i][j] += 1
# print(levelCtr)

# # Question 1.5
# k = 10
# totalM = [10, 100, 1000, 10000]
# error = []
# for m in totalM:
#     trainM = int(0.8 * m)
#     testM = int(0.2 * m)
#     trainData = Data(k, trainM)
#     model = DecisionTree(trainData.data, trainData.Y)
```

```
#     errorRate = 0
#     for _ in range(20):
#         testData = Data(k, testM)
#         errorRate += model.calcError(testData.data, testData.Y)
#     error.append(errorRate/20)
# print('totalM:', totalM)
# print('error:', error)
# plt.plot(totalM, error)
# plt.show()

# # Question 1.6
# k = 1000
# totalM = [10, 50, 100, 500, 1000, 5000, 10000]
# error = []
# errorDiff = []
# for m in totalM:
#     trainM = int(0.8 * m)
#     testM = int(0.2 * m)
#     trainData = Data(k, trainM)
#     model = DecisionTree(trainData.data, trainData.Y)
#     modelDiff = DecisionTree(trainData.data, trainData.Y, trainByDiff=True)
#     errorRate = 0
#     errorRateDiff = 0
#     for _ in range(20):
#         testData = Data(k, testM)
#         errorRate += model.calcError(testData.data, testData.Y)
#         errorRateDiff += modelDiff.calcError(testData.data, testData.Y)
#     error.append(errorRate/20)
#     errorDiff.append(errorRateDiff/20)
# print('totalM:', totalM)
# print('error:', error)
# print('errorDiff:', errorDiff)
# plt.plot(totalM, error, 'g')
# plt.plot(totalM, errorDiff, 'b')
# plt.show()
```

```
# # Question 2.1
# k = 10
# totalM = [10, 100, 1000, 10000]
# error = []
# for m in totalM:
#     trainM = int(0.8 * m)
#     testM = int(0.2 * m)
#     trainData = Data(k, trainM)
#     model = DecisionTree(trainData.data2, trainData.Y2)
#     errorRate = 0
#     for _ in range(20):
#         testData = Data(k, testM)
#         errorRate += model.calcError(testData.data2, testData.Y2)
#     error.append(errorRate/20)
# print('totalM:', totalM)
# print('error:', error)
# plt.plot(totalM, error)
# plt.show()

# # Question 2.2
# k = 10
# totalM = [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000]
# noise = []
# numIter = 5
# for m in totalM:
#     print('Training Decision Tree with M =', m)
#     curNoise = 0
#     for _ in range(numIter):
#         data = Data(k, m)
#         model = DecisionTree(data.data2, data.Y2)
#         curNoise += model.countNoise()
#         # print(curNoise)
#     noise.append(curNoise/numIter)
# # totalM: [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000]
# # noise: [0.0, 1.2, 2.4, 3.4, 4.4, 8.8, 5.0, 5.6, 3.2, 2.6, 2.4]
# totalM = [5]
```

```
# avgNoise = 1
# numIter = 1
# plotM = []
# visitedM = set()
# m = 0
# zeros = 0
# while m < 10000 or zeros < 5:
#     m = heapq.heappop(totalM)
#     # m = totalM.pop(0)
#     if m in visitedM:
#         continue
#     visitedM.add(m)
#     plotM.append(m)
#     print('Training Decision Tree with M =', m)
#     curNoise = 0
#     for _ in range(numIter):
#         data = Data(k, m)
#         model = DecisionTree(data.data2, data.Y2)
#         curNoise += model.countNoise()
#         # print(curNoise)
#     avgNoise = curNoise/numIter
#     noise.append(avgNoise)
#     if m > 10000 and avgNoise == 0:
#         zeros += 1
#     print('current noise', avgNoise)
#     print('current progress:', zeros, '/ 5')
#     # if not totalM:
#     #     totalM.append(m*2)
#     #     totalM.append(m*5)
#     #     totalM.append(m*10)
#     heapq.heappush(totalM, m*2)
#     heapq.heappush(totalM, m*5)
#     heapq.heappush(totalM, m*10)
# print('plotM:', plotM)
# print('noise:', noise)
# fig, ax = plt.subplots()
```

```
# plt.scatter(plotM, noise)
# plt.show()


# # Question 2.3 a
# k = 5 # k here doesn't affect anything
# m = 10000
# trainM = int(0.8 * m)
# testM = int(0.2 * m)
# trainData = Data(k, trainM)
# model = DecisionTree(trainData.data2, trainData.Y2)
# model.printTree()
# testData = Data(k, testM)
# error = [[0, 0]]
# for i in range(testM):
#     level = model.predictByLevel(testData.data2[i])
#     print('testData.Y2:', testData.Y2[i])
#     for j in range(len(level)):
#         if j >= len(error):
#             error.append([0, 0])
#         error[j][1] += 1
#         if level[j] != testData.Y2[i]:
#             error[j][0] += 1
# print(error)
# lenError = len(error)
# levels = list(range(0, lenError+1))
# errorRate = [0.5]
# for i in range(lenError):
#     errorRate.append(error[i][0]/error[i][1])
# plt.plot(levels, errorRate)
# plt.show()


# # Question 2.3 b
# k = 5
# m = 10000
# trainM = int(0.8 * m)
# testM = int(0.2 * m)
```

```
# trainData = Data(k, trainM)
# testData = Data(k, testM)
# threshold = list(range(50, 300, 10))[::-1]
# errorList = []
# for t in threshold:
#     model = DecisionTree(trainData.data2, trainData.Y2, t)
#     error = 0
#     for i in range(testM):
#         rst = model.predictBySize(testData.data2[i])
#         if rst != testData.Y2[i]:
#             error += 1
#     errorList.append(error)
# errorBySize = []
# for e in errorList:
#     errorBySize.append(e/testM)
# print('errorList:', errorList)
# print('errorBySize:', errorBySize)
# plt.plot(threshold, errorBySize)
# plt.show()

# # Question 2.4&5
# k = 10
# # totalM = [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000]
# totalM = list(range(1000, 100000, 1000))
# noise = []
# noiseLevel = []
# noiseSize = []
# numIter = 1
# for m in totalM:
#     print('Training Decision Tree with M =', m)
#     curNoise = 0
#     for _ in range(numIter):
#         data = Data(k, m)
#         model = DecisionTree(data.data2, data.Y2)
#         curNoise += model.countNoise()
#     noise.append(curNoise/numIter)
```

```
#     curNoise = 0
#     for _ in range(numIter):
#         data = Data(k, m)
#         model = DecisionTree(data.data2, data.Y2, trainByLevel=True)
#         curNoise += model.countNoise()
#     noiseLevel.append(curNoise/numIter)
#     curNoise = 0
#     for _ in range(numIter):
#         data = Data(k, m)
#         model = DecisionTree(data.data2, data.Y2, trainBySize=True)
#         curNoise += model.countNoise()
#     noiseSize.append(curNoise/numIter)
# print('totalM:', totalM)
# print('noise:', noise)
# print('noiseLevel:', noiseLevel)
# print('noiseSize:', noiseSize)
# # plt.plot(totalM, noise, 'k')
# # plt.plot(totalM, noiseLevel, 'b')
# # plt.plot(totalM, noiseSize, 'g')
# plt.scatter(totalM, noise, c='k')
# plt.scatter(totalM, noiseLevel, c='b')
# plt.scatter(totalM, noiseSize, c='g')
# plt.show()
```