Programmer Assignment #4 Report

1.result between different datasets:
(1)iris(4 attributes, 3 classes, 150 instances)

```
iris.data                 iris.data
0.8910000000000002        0.9121999999999996
```

(2)glass(10 attributes, 7 classes, 214 instances)

```
glass.data                glass.data
0.8386111111111113        0.7919444444444447
```

(3)ionosphere(34 attributes, 2 classes, 351 instances)

```
ionosphere.data           ionosphere.data
0.8606837606837603        0.8617948717948711
```

結論：iris表現最好，它的dataset有最少的attributes和instances

2.result between different sizes of the training and validation subsets:
(1)training—1/2, validation—1/2(glass / ionosphere)

```
glass.data                ionosphere.data
0.7970093457943925        0.8410795454545452
```

(2)training—2/3, validation—1/3

```
glass.data                ionosphere.data
0.8490277777777782        0.878205128205273
```

(3)training—3/4, validation—1/4

```
glass.data                ionosphere.data
0.8429629629629625        0.8673863636363638
```

結論：training data從1/2到2/3效果提升最明顯

3.result between different numbers of trees:
(1)100 trees

```
glass.data              ionosphere.data
0.7942592592592591      0.8488636363636364
```

(2)500 trees

```
glass.data              ionosphere.data
0.8401851851851864      0.8782272727272724
```

(3)1000 trees

```
glass.data              ionosphere.data
0.8696851851851886      0.8833409090909081
```

結論：越大的tree number可以看出會有越高的精準度
4.result between different attributes to consider:
(1)sqrt(10) / sqrt(34)

```
glass.data              ionosphere.data
0.8443055555555559      0.8719318181818179
```

(2)sqrt(10) X 2 / sqrt(34) X 2

```
glass.data              ionosphere.data
0.9412500000000006      0.8704545454545449
```

(3)sqrt(10) X 3 / sqrt(34) X 3

```
glass.data              ionosphere.data
0.9325000000000004      0.8680341880341875
```

結論：原本以為attribute的大
小會影響精準度非常多，但似乎除了sqrt(10)->sqrt(10)X2影響比較大之外，並沒有太大變化，可能和attribute數量有關

5.result between different size of tree(giving limits that having minimum number of samples per node):
(1)no limits:

```
glass.data                  ionosphere.data
0.8648611111111112          0.8662393162393154
```

(2)every node should have at least 20 samples:

```
glass.data                  ionosphere.data
0.779027777777778           0.8510256410256405
```

(3)every node should have at least 40 samples:

```
glass.data                  ionosphere.data
0.665138888888889           0.83076923076923
```

結論：增加了限制後很明顯準度下降，但是優點是速度變快了

6.result between Extremely random forest and normal forest:
(1)Extremely random forest:

```
glass.data                  ionosphere.data
0.842222222222222           0.7030769230769228
```

(2)normal forest:

```
glass.data                  ionosphere.data
0.864444444444444           0.8233333333333331
```

結論：extremely random forest的選擇非常隨機，因此會降低準度，除此之外，時間不一定會減少，因為如果每次選attribute都剛好選到無法將data分開的attribute會浪費很多時間，但也可能是我的資料大小較小，沒辦法看出extremely random forest的優勢

Note：整份code都是我自己寫的，並沒有參考網路，因此沒有附上相關資料

7.Observations:
如果某個方法可以增加速度，有很大概率會失去準度，因此如果有方法可以在不犧牲準度下增加速度都是可以考慮的好方法，反之亦然，而如果要對速度獲準度犧牲的話就要視情況而定

8.Interpretation: (python)
主要就是依照pdf的流程，先以tree bagging, attribute bagging建立subset，然後依照每個節點的gini index分割資料，直到全部資料被分類，在重複這個動作來建森林。

9.Things I have learnt:
這次有觀察到比較明顯的是iris表現相較之下比較不穩定，差別在於它只有四個attribute，且資料量也較少，因此後面的實驗我盡量不以iris做測試

10.remaining questions:
目前沒有增加速度又不會犧牲準度的做法，還在嘗試中

11.idea of future investigation:
attribute bagging 和 tree bagging 的大小比例都是未來可以測試的內容。

```
import numpy as np
# 4, 10, 34
attributes = 10
# 3, 7, 2
classes = 7
# 150, 214, 351
Instances = 214
trainingDataNum = int(Instances*(2/3))
validationDataNum = Instances - trainingDataNum
treeNum = 100

# read file
# iris.data, glass.data, ionosphere.data
fp = open('glass.data', "r")
lines = fp.readlines()
data = []
# may have to change between len(lines) or
len(lines)-1 through different data
for i in range(len(lines)):
    lines[i] = lines[i].split(',')
    lines[i][attributes] = lines[i][attributes]
[:-1]
    data.append(lines[i])
fp.close()
for i in range(len(data)):
    for j in range(attributes):
        data[i][j] = float(data[i][j])
```

```
# training subset and validation subset
import random
random.shuffle(data)
trainingSubset = []
validationSubset = []
for i in range(trainingDataNum):
    trainingSubset.append(data[i])
for i in range(trainingDataNum,
trainingDataNum+validationDataNum):
    validationSubset.append(data[i])
```

```
import random
import math
# attribute bagging
def selectAttributes(n):
    temp = []
    for i in range(n-1):
        temp.append(i)
    random.shuffle(temp)
    num = int(math.sqrt(n))
    ans = []
    for i in range(num):
        ans.append(temp[i])
    return ans
# tree bagging
def selectData(data, n):
    ans = []
    for i in range(n):
```

```
        tt = random.randint(0, len(data)-1)
        ans.append(data[tt])
    return ans
def splittingNode(trainingData,
trainingAttribute):
    # select a attribute
    bestAttribute = 0
    bestThreshold = 0
    lowestImpurity = 1
    for i in range(len(trainingAttribute)):
        temparr = []
        V = []
        # select a threshold for a given
attribute
        for j in range(len(trainingData)):
            T = []
            T.append(trainingData[j]
[trainingAttribute[i]])
if(trainingData[j[trainingAttribute[i]] not in
V):
V.append(trainingData[j[trainingAttribute[i]])
            T.append(trainingData[j]
[len(trainingData[0])-1])
            temparr.append(T)
        # temparr saves one of the attribute
data and a tag
        temparr.sort(key = lambda s: s[0])
```

```
        # V saves all values of given
attributes
        V.sort()
        threshold = []
        # threshold are decided by V
        for j in range(len(V)-1):
            threshold.append((V[j]+V[j+1])/2)
        # decide which threshold is better
        betterThreshold = 0
        lowestGini = 1
        for j in range(len(threshold)):
            alpha = []
            beta = []
            # use threshold to devide data into
two sets, which is alpha and beta
            for k in range(len(temparr)):
                if(temparr[k][0]>threshold[j]):
                    alpha.append(temparr[k][1])
                else:
                    beta.append(temparr[k][1])
            alpha.sort()
            beta.sort()
            # alphaCount and betaCount holds
the sum of every class
            alphaCount = []
            betaCount = []
            string = alpha[0]
            count = 0
```

```
            for k in range(len(alpha)):
                if(string == alpha[k]):
                    count+=1
                else:
                    alphaCount.append(count)
                    string = alpha[k]
                    count = 1
            alphaCount.append(count)
            string = beta[0]
            count = 0
            for k in range(len(beta)):
                if(string == beta[k]):
                    count+=1
                else:
                    betaCount.append(count)
                    string = beta[k]
                    count = 1
            betaCount.append(count)
            # calculate Gini index
            alphaGini = 1
            betaGini = 1
            alphaSum = len(alpha)
            betaSum = len(beta)
            for k in range(len(alphaCount)):
                alphaGini-=(alphaCount[k]/
alphaSum)*(alphaCount[k]/alphaSum)
            for k in range(len(betaCount)):
```

```
                betaGini-=(betaCount[k]/
betaSum)*(betaCount[k]/betaSum)
            Gini = (alphaSum/
(alphaSum+betaSum))*alphaGini+(betaSum/
(alphaSum+betaSum))*betaGini
            if(Gini < lowestGini):
                lowestGini = Gini
                betterThreshold = threshold[j]
        if(lowestGini < lowestImpurity):
            lowestImpurity = lowestGini
            bestAttribute =
trainingAttribute[i]
            bestThreshold = betterThreshold
    newAlpha = []
    newBeta = []
    for i in range(len(trainingData)):
        if(trainingData[i][bestAttribute] <
bestThreshold):
            newAlpha.append(trainingData[i])
        else:
            newBeta.append(trainingData[i])
    ans = []
    ans.append(lowestImpurity)
    ans.append(bestAttribute)
    ans.append(bestThreshold)
    ans.append(newAlpha)
    ans.append(newBeta)
    return ans
```

```
# following is used to implement extremely
random forest
# import random
# def splittingNode(trainingData,
trainingAttribute):
#     i =
random.randint(0,len(trainingAttribute)-1)
#     temparr = []
#     V = []
#     # select a threshold for a given
attribute
#     for j in range(len(trainingData)):
#         T = []
#         T.append(trainingData[j]
[trainingAttribute[i]])
#         if(trainingData[j]
[trainingAttribute[i]] not in V):
#             V.append(trainingData[j]
[trainingAttribute[i]])
#         T.append(trainingData[j]
[len(trainingData[0])-1])
#         temparr.append(T)
#     # temparr saves one of the attribute data
and a tag
#     temparr.sort(key = lambda s: s[0])
#     # V saves all values of given attributes
#     V.sort()
```

```
#     threshold = []
#     # threshold are decided by V
#     for j in range(len(V)-1):
#         threshold.append((V[j]+V[j+1])/2)
#     # decide which threshold is better
#     betterThreshold = 0
#     lowestGini = 1
#     for j in range(len(threshold)):
#         alpha = []
#         beta = []
#         # use threshold to devide data into
two sets, which is alpha and beta
#         for k in range(len(temparr)):
#             if(temparr[k][0] > threshold[j]):
#                 alpha.append(temparr[k][1])
#             else:
#                 beta.append(temparr[k][1])
#         alpha.sort()
#         beta.sort()
#         # alphaCount and betaCount holds the
sum of every class
#         alphaCount = []
#         betaCount = []
#         string = alpha[0]
#         count = 0
#         for k in range(len(alpha)):
#             if(string == alpha[k]):
#                 count+=1
```

```
#             else:
#                 alphaCount.append(count)
#                 string = alpha[k]
#                 count = 1
#         alphaCount.append(count)
#         string = beta[0]
#         count = 0
#         for k in range(len(beta)):
#             if(string == beta[k]):
#                 count+=1
#             else:
#                 betaCount.append(count)
#                 string = beta[k]
#                 count = 1
#         betaCount.append(count)
#         # calculate Gini index
#         alphaGini = 1
#         betaGini = 1
#         alphaSum = len(alpha)
#         betaSum = len(beta)
#         for k in range(len(alphaCount)):
#             alphaGini-=(alphaCount[k]/
alphaSum)*(alphaCount[k]/alphaSum)
#         for k in range(len(betaCount)):
#             betaGini-=(betaCount[k]/
betaSum)*(betaCount[k]/betaSum)
```

```
#         Gini = (alphaSum/
(alphaSum+betaSum))*alphaGini+(betaSum/
(alphaSum+betaSum))*betaGini
#         if(Gini < lowestGini):
#             lowestGini = Gini
#             betterThreshold = threshold[j]
#     newAlpha = []
#     newBeta = []
#     bestAttribute = i
#     bestThreshold = betterThreshold
#     for i in range(len(trainingData)):
#         if(trainingData[i][bestAttribute] <
bestThreshold):
#             newAlpha.append(trainingData[i])
#         else:
#             newBeta.append(trainingData[i])
#     ans = []
#     ans.append(lowestGini)
#     ans.append(bestAttribute)
#     ans.append(bestThreshold)
#     ans.append(newAlpha)
#     ans.append(newBeta)
#     return ans
```

```
class Tree():
```

```
    def __init__(self):
        self.root = Node(0, 0)
class Node():
    def __init__(self, attribute, threshold):
        self.attribute = attribute
        self.threshold = threshold
        self.leafnode = False
        self.classification = ""

# build a tree
def buildTree(root, trainingData,
trainingAttributes):
    finish = True
    if(trainingData != []):
        string = trainingData[0]
[len(trainingData[0])-1]
        for i in range(len(trainingData)):
            if(trainingData[i]
[len(trainingData[0])-1] != string):
                finish = False
                break
        if(finish == False):
            returnData =
splittingNode(trainingData, trainingAttributes)
            Node1 = Node(0,0)
            Node2 = Node(0,0)
            root.attribute = returnData[1]
            root.threshold = returnData[2]
```

```
            root.left = Node1
            root.right = Node2
            trainingAttributes =
selectAttributes(attributes)
            buildTree(Node1, returnData[3],
trainingAttributes)
            buildTree(Node2, returnData[4],
trainingAttributes)
        else:
            root.leafnode = True
            root.classification =
trainingData[0][len(trainingData[0])-1]
    else:
        root.leafnode = True
        root.classification = ""

# main
sum = 0
# build a forest
for n in range(treeNum):
    decisionTree = Tree()
    trainingData = selectData(trainingSubset,
trainingDataNum)
    buildTree(decisionTree.root, trainingData,
selectAttributes(attributes))
    validationData =
selectData(validationSubset, validationDataNum)
    correct = 0
```

```
    wrong = 0
    for i in range(len(validationData)):
        position = decisionTree.root
        while(position.leafnode == False):
            if(validationData[i]
[position.attribute] > position.threshold):
                position = position.right
            else:
                position = position.left
        if(validationData[i]
[len(validationData[i])-1] ==
position.classification):
            correct+=1
        else:
            wrong+=1
    sum+=(correct/(correct+wrong))
print("glass.data")
print(sum/treeNum)
```