536

Xinyang Wang

189001002

1 HW2

1.1

Before answering the questions, let's first take a look at the structure of the code.

```
class DescisonTree():
    def __init__(self,m,k):
        self.m = m
        self.k = k
        self.tree = [None]*(2**self.k-1)
        self.label = [None]*(2**(self.k+1)-1)

    def GenerateDataPoint(self):
        '''generate data according to the given pattern '''

    def Partition(self,matrix,x):
        '''partition the data matrix based on x, \
        return two matrixs according to value of x '''

    def InformationGain(self,matrix):
        '''compute the IGs and return max IG and its corresponding x '''

    def GenerateTree(self,matrix,id = 0):
        '''recursively building tree and label'''

    def GetError(self, matrix):
        '''get error of a built tree  '''

    if __name__ == '__main__':
```

Then we go back to question 1. For a given value of k and m, the following part of code generates a training data set based on the given pattern

```
def GenerateDataPoint(self):
        '''generate data according to the given pattern '''
        #compute weight list
        w = []
        denominator = 0
        for i in range(2,self.k+1):
            denominator += 0.9**i
        for i in range(1,self.k+1):
```

```
            w_i = 0.9**i/denominator
            w.append(w_i)

        #compute data point
        matrix = np.zeros((self.m, self.k+1),'int')
        for i in range(self.m):
            value = 0
            matrix[i][0] = 1 if random.random()<0.5 else 0
            for j in range(1,self.k):
                matrix[i][j] = matrix[i][j-1] if \
                random.random()<0.75 else 1-matrix[i][j-1]
                value += matrix[i][j]*w[j]
            matrix[i][self.k] = matrix[i][0] if value>=1/2 else 1-matrix[i][0]

        return matrix
        print(f'generate {self.m} data points successful ')
```

1.2

The following four parts relatively achieve three functions: partitioning the data; compute information gain; building decision tree; return the error of a built tree.

```
def Partition(self,matrix,x):
        '''partition the data matrix based on x, \
        return two matrixs according to value of x '''
        list_x_0 = []
        list_x_1 = []
        for i in range(matrix.shape[0]):
            if matrix[i][x] == 0:
                list_x_0.append(matrix[i])
            else:
                list_x_1.append(matrix[i])
        mat0 = np.array(list_x_0).reshape(-1,self.k+1)
        mat1 = np.array(list_x_1).reshape(-1,self.k+1)
        return mat0,mat1
```

```
def InformationGain(self,matrix):
        '''compute the IGs and return max IG and its corresponding x '''
        #compute H(Y)
        count_y = 0
        for i in range(matrix.shape[0]):
            count_y = count_y+1 if matrix[i][self.k] == 1 else count_y
        p_y = count_y/matrix.shape[0]
        H_y = -p_y*math.log(p_y,2)-(1-p_y)*math.log((1-p_y),2)

        #compute each H(Y/Xi)
        list_IG = []
        for j in range(self.k):
            count_x_1 = 0    #compute sum of Xi=1
            count_x_1_y_1 = 0    #compute sum of yi=1 when Xi=1
            count_x_0_y_1 = 0    #compute sum of yi=1 when Xi=0
```

```
            for i in range(matrix.shape[0]):
                if matrix[i][j] == 1:
                    count_x_1 += 1
                    if matrix[i][self.k] == 1:
                        count_x_1_y_1 += 1
                else:
                    if matrix[i][self.k] == 1:
                        count_x_0_y_1 += 1

            p1 = count_x_1/matrix.shape[0]
            p0 = 1-p1
            p11 = count_x_1_y_1/count_x_1 if count_x_1 != 0 else 0
            p10 = 1-p11
            p01 = count_x_0_y_1/(matrix.shape[0]-count_x_1) if\
            (matrix.shape[0]-count_x_1) != 0 else 0
            p00 = 1-p01
            #print(p1,p0,p11,p10,p01,p00)

            H_yx = p1*(-p11*math.log((p11+0.0000001),2)-\
            p10*math.log((p10+0.0000001),2)) + \
            p0*(-p01*math.log((p01+0.0000001),2)- \
            p00*math.log((p00+0.0000001),2))
            IG = H_y-H_yx
            list_IG.append([j,IG])
        return max(list_IG, key=lambda x: x[1])
```

```
def GenerateTree(self,matrix,id = 0):
        '''recursively building tree and label'''
        #print('id',id)
        count = 0
        for i in range(matrix.shape[0]):
            if matrix[i][-1]==0:
                count += 1

        #print('bizhi',count/matrix.shape[0])
        #input()
        if count/matrix.shape[0]>0.95 :
            self.label[id] = 0
            return
        if count/matrix.shape[0]<0.05 :
            self.label[id] = 1
            return

        pos, IG = self.InformationGain(matrix)
        #print('pos&IG',pos,IG)
        #input()
        self.tree[id] = pos+1
        mat0, mat1 = self.Partition(matrix, pos)
        self.GenerateTree(mat0,2*id+1)
        #print('id', id)
        self.GenerateTree(mat1,2*id+2)
```

```
def GetError(self, matrix):
        '''get error of a built tree  '''
        if matrix.shape[1]-1 != self.k:
            print('depth of matrix does not comply with depth of the built tree')
            return 0

        right = 0
        for i in range(matrix.shape[0]):
            pos = 0
            id = self.tree[pos]
            while id:
                if matrix[i][id-1] == 0:
                    pos = 2*pos+1
                else:
                    pos = 2*pos+2
                if pos > len(self.tree):
                    id = None
                    break
                id = self.tree[pos]
            predict = self.label[pos]
            if predict == matrix[i][-1]:
                right += 1

        return 1-right/matrix.shape[0]
```

1.3

For k = 4 and m = 30, here is a sample and the corresponding decision tree.

```
original matrix generated by k=4 m=30:
[[0 0 1 0 1]
 [0 0 0 0 1]
 [1 1 1 1 1]
 [1 1 0 0 0]
 [0 0 0 0 1]
 [0 0 0 0 1]
 [0 1 1 1 0]
 [1 1 0 0 0]
 [0 0 0 0 1]
 [0 0 1 1 0]
 [0 1 1 1 0]
 [0 1 0 0 1]
 [0 0 1 1 0]
 [1 1 0 0 0]
 [1 1 1 1 1]
 [0 0 0 0 1]
 [1 1 0 0 0]
 [0 0 0 0 1]
 [0 0 1 0 1]
 [1 0 0 0 0]
 [1 1 0 1 1]
```

```
 [1 1 1 1 1]
 [1 1 0 0 0]
 [0 0 0 0 1]
 [0 0 0 0 1]
 [0 0 1 1 0]
 [0 0 1 1 0]
 [0 0 1 1 0]
 [0 0 1 1 0]
 [0 0 0 1 1]]
```

Based on the data points above, we get the decision tree as follows:

```
decision tree of original matrix:
[3, 1, 1, None, 4, 4, None, None, None, None, None, None, None, None]

corresponding label:
[None, None, None, 1, None, None, 1, None, None, 0, 1, 1, 0, None, None,\
None, None, None, None, None, None, None, None, None, None, None, \
None, None, None, None]
```

Below we draw the tree:

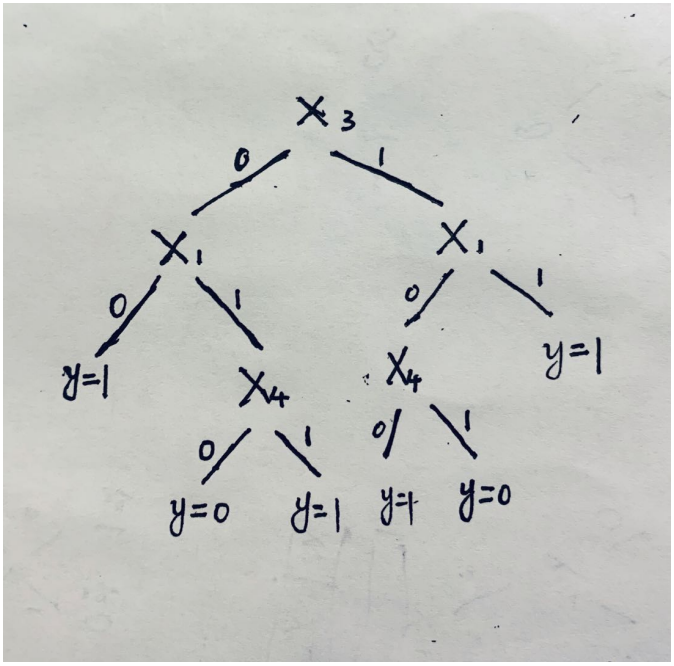

Figure 1: The Decision Tree

As we can see from above, the depth of the tree is 3, which is less than 4. So our approach helps reduce the depth of tree.

1.4

Based on k=4 and m=30, we first generate a decision tree from original 30 data points. Then we generate 2000 new data points according to the same scheme. Finally we try to fit the decision tree to new data, and get the average

error, which is around 0.06.

```
if __name__ == '__main__':
    decision_tree = DescisonTree(30,4)
    original = decision_tree.GenerateDataPoint()
    print(f'original matrix generated by k=4 m=30:\n{original}')

    decision_tree.GenerateTree(original)
    print(f'decision tree of original matrix:\n{decision_tree.tree}\n')
    print(f'corresponding label:\n{decision_tree.label}')

    error_train = decision_tree.GetError(original)
    print(f'training error = {error_train}')

    #generate new data points to get test error
    decision_tree.m = 2000
    new = decision_tree.GenerateDataPoint()
    print(f'new matrix generated by same k and m:\n{new}')

    error_test = decision_tree.GetError(new)
    print(f'test error = {error_test}')
```

1.5

When k=10, there are 2^{10} possible data. That is to say, if we have 2^{10} data points, we are likely to see every possible situation. So we take m from 100 to 5001(which is much bigger than 2^{10}) with step 100. For each m, we repeat for 50 times and compute the average of $|error_{train} - error_{test}|$
Below is the difference drawn according to above scheme. And from this we can see that, when number of data points increases, the difference between $error_{train}$ and $error_{test}$ decreases. That is because we are likely to see every possible data, thus the decision tree we get can generalize better.

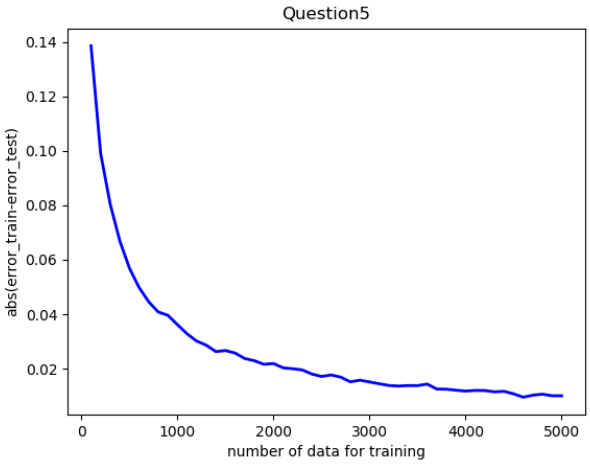

Figure 2: Difference

1.6

I implement Gini-Impurity in generating decision tree.(I heard about Gini-Impurity from others, and learned how it works from the Internet) Gini impurity is a measure of how likely a element is incorrectly labeled. Though they have different definition, they have a similar curve as mentioned on some websites.
Below is the main code for Gini-impurity.

```
def Gini(self,matrix):
        '''compute the Gini impurity and return min Gini \
        and its corresponding x '''

        list_gini = []
        for j in range(self.k):
            count_x_1 = 0    #compute sum of Xi=1
            count_x_1_y_1 = 0    #compute sum of yi=1 when Xi=1
            count_x_0_y_1 = 0    #compute sum of yi=1 when Xi=0
            for i in range(matrix.shape[0]):
                if matrix[i][j] == 1:
                    count_x_1 += 1
                    if matrix[i][self.k] == 1:
                        count_x_1_y_1 += 1
                else:
                    if matrix[i][self.k] == 1:
                        count_x_0_y_1 += 1

            p1 = count_x_1/matrix.shape[0]
            p0 = 1-p1
            p11 = count_x_1_y_1/count_x_1 if count_x_1 != 0 else 0
            p10 = 1-p11
            p01 = count_x_0_y_1/(matrix.shape[0]-count_x_1) if\
            (matrix.shape[0]-count_x_1) != 0 else 0
            p00 = 1-p01
            #print(p1,p0,p11,p10,p01,p00)

            if p0==0 or p1==0:    #if last time use x1 to part, \
            this time p0=0 or p1 =0 when computing based on x1
                continue
            gini = p0*(1-p00**2-p01**2)+p1*(1-p10**2-p11**2)

            list_gini.append([j,gini])
        #print(list_gini)
        #input()
        return min(list_gini, key=lambda x: x[1])
```

Now we do what we did in question 5 (we take m from 100 to 5001 with step 100. For each m, we repeat for 50 times),we can get the difference curve in Figure 3:
When we put the difference curve of question 5 and question 6 into one graph as in Figure 4(blue line is for question 5; red line is for question 6). We can see that both approach have similar performances.

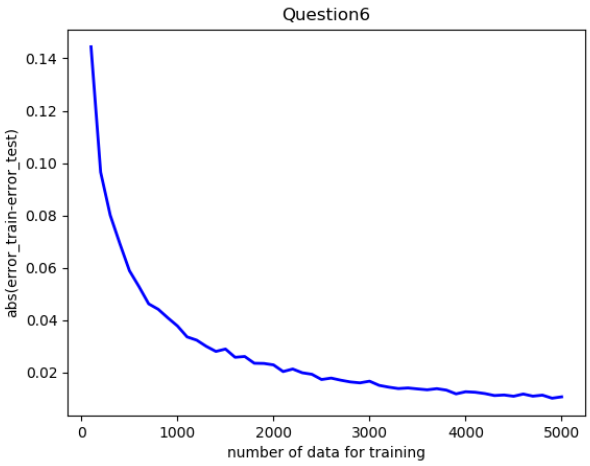

Figure 3: Difference

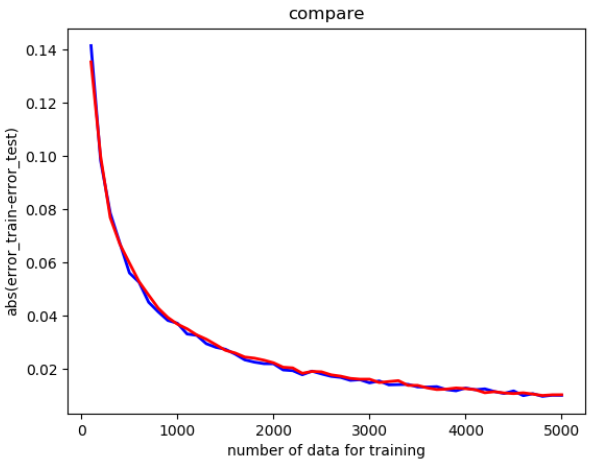

Figure 4: compare