Datamining Homework : Classification

นาย ดรันภพ เป็งคำตา

580610642

รูปแบบของการทดลอง

เป็นการทดลองเพื่อทดสอบการทำงาน และประสิทธิภาพของการทำ Classification ของ algorithm 2 แบบ คือ Neural network และ Decision tree ซึ่งทั้งสองจะทดสอบความแม่นยำด้วย การใช้ K-fold validation

รูปแบบของข้อมูลที่ใช้ทดสอบ

เป็นชุดข้อมูลของนักศึกษาจำนวน 395 คน ประกอบด้วย attribute ทั้งหมด 33 อย่าง ซึ่งประกอบด้วยข้อมูลลักษณะต่าง ๆดังนี้ เป็นattributeที่เป็นไปได้ 2 ค่า (binary), เป็นจำนวนเต็มในช่วง ๆ หนึ่ง (numeric) และเป็น ข้อความต่าง ๆ กันจำนวนหนึ่ง (nominal) โดยattribute ที่ต้องการจำแนกคือ G3 เป็นข้อมูลของเกรดที่นักศึกษาคนนั้นได้เมื่อจบคอสนั้น ๆ ซึ่งเป็นค่าที่เป็นไปได้ในช่วง [0, 20]

ดังนั้นจึงจะให้ผลของการทดสอบเป็นการจำแนกคลาสผลลัพธ์ 21 แบบ แต่ละคลาสคลาสจะเป็นตัวแทนของค่าตั้งแต่ 0 – 20

การ Preprocess ข้อมูล

เนื่องจากข้อมูลที่ได้มา พบว่าเป็นข้อมูลที่ไม่มี missing value จึงไม่จำเป็นที่จะต้องจัดการกับปัญหานี้ ในขั้นตอนนี้ได้ทำการ normalize ข้อมูลทั้งหมดให้อยู่อยู่ในช่วง [0, 1] อย่างเท่าเทียมกัน ยกเว้นเว้น attribute **‘G3’** เนื่องจากเป็น output class ที่ต้องการทำการจำแนก และ เนื่องจากพบว่า attribute ทั้งหมด มีข้อมูลอยู่สามแบบ จังได้ออกแบบ function สำหรับการ normalize ไว้เพื่อความสะดวกในการใช้งาน

- Function สำหรับ normalize ข้อมูล

```
def normalize(input):
    input['school'] = input['school'].apply(lambda x:binary('GP', x))
    input['sex'] = input['sex'].apply(lambda x:binary('F', x))
    input['age'] = input['age'].apply(lambda x:scaling(15, 22, x))
    input['address'] = input['address'].apply(lambda x:binary('U', x))
    input['famsize'] = input['famsize'].apply(lambda x:binary('LE3', x))
    input['Pstatus'] = input['Pstatus'].apply(lambda x:binary('T', x))
    input['Medu'] = input['Medu'].apply(lambda x:scaling(0, 4, x))
    input['Fedu'] = input['Fedu'].apply(lambda x:scaling(0, 4, x))
    input['Mjob'] = input['Mjob'].apply(lambda x:nominal(['teacher', 'health',
'services', 'at_home', 'other'], x))
    input['Fjob'] = input['Fjob'].apply(lambda x:nominal(['teacher', 'health',
'services', 'at_home', 'other'], x))
    input['reason'] = input['reason'].apply(lambda x:nominal(['home',
'reputation', 'course', 'other'], x))
    input['guardian'] = input['guardian'].apply(lambda x:nominal(['mother',
'father', 'other'], x))
    input['traveltime'] = input['traveltime'].apply(lambda x:scaling(1, 4, x))
    input['studytime'] = input['studytime'].apply(lambda x:scaling(1, 4, x))
    input['failures'] = abs(input['failures'].apply(lambda x:scaling(1, 4, x)))
    input['schoolsup'] = input['schoolsup'].apply(lambda x:binary('yes', x))
    input['famsup'] = input['famsup'].apply(lambda x:binary('yes', x))
    input['paid'] = input['paid'].apply(lambda x:binary('yes', x))
    input['activities'] = input['activities'].apply(lambda x:binary('yes', x))
    input['nursery'] = input['nursery'].apply(lambda x:binary('yes', x))
    input['higher'] = input['higher'].apply(lambda x:binary('yes', x))
    input['internet'] = input['internet'].apply(lambda x:binary('yes', x))
    input['romantic'] = input['romantic'].apply(lambda x:binary('yes', x))
    input['famrel'] = input['famrel'].apply(lambda x:scaling(1, 5, x))
    input['freetime'] = input['freetime'].apply(lambda x:scaling(1, 5, x))
    input['goout'] = input['goout'].apply(lambda x:scaling(1, 5, x))
    input['Dalc'] = input['Dalc'].apply(lambda x:scaling(1, 5, x))
    input['Walc'] = input['Walc'].apply(lambda x:scaling(1, 5, x))
    input['health'] = input['health'].apply(lambda x:scaling(1, 5, x))
    input['absences'] = input['absences'].apply(lambda x:scaling(0, 93, x))
    input['G1'] = input['G1'].apply(lambda x:scaling(0, 20, x))
    input['G2'] = input['G2'].apply(lambda x:scaling(0, 20, x))
```

- **Function สำหรับ normalize ข้อมูลที่เป็นช่วงค่า (numeric)**

```
def scaling(min, max, x):
    return (x - min)/(max - min)
```

- **Function สำหรับ normalize ข้อมูลที่เป็นสองค่า (binary)**

```
def binary(a, x):
    return 1 if x == a else 0
```

- **Function สำหรับ normalize ข้อมูลที่เป็นชุดของข้อความ (nominal)**

```
def nominal(p_list, x):
    for n, i in enumerate(p_list):
        if i == x:
            return scaling(0, len(p_list)-1, n)
```

การทำ K – fold validation

เป็นการทำการเทรนโมเดล และทดสอบข้อมูล ทั้งหมด k รอบ แล้วนำค่าความแม่นยำของทุกรอบมาเฉลี่ยกัน โดยในทีนี้ได้ใช้ค่า k=10 โดยในแต่ละ fold ได้ทำการแบ่งข้อมูลออกเป็น ข้อมูลสำหรับการ เทรนโมเดล 356 ตัว และ สำหรับการทดสอบ 39 ตัว โดยทำการเลือกข้อมูลโดยวิธีการสุ่ม โดยรับประกันว่าจะไม่มีข้อมูลที่ซ้ำกันถูกหยิบออกมา

Neural network

เป็นการออกแบบโค้ดการทำงานตามหลักการของ OOP ซึ่งจะเลียนแบบโครงสร้างการทำงานตามหลักการทำงานของ neuron network กล่าวคือ มีการสร้างคลาส Neuron Network ซึ่งประกอบด้วยคลาสย่อย คือ คลาสของ Layer และในคลาสของ Layer ยังประกอบด้วยคลาสของ Node ดังนั้นจะทำให้โค้ดที่ได้มีการทำงานที่มีระเบียบและโครงสร้างการทำงานที่ชัดเจน สามารถเพิ่มลดจำนวน Layer และ Node ได้อย่างง่ายดายตามหลักการทำงานของ OOP

การประกาศ และการเพิ่ม layer ของ neural network

ตอนทดสอบได้ทำการทดลองเพิ่มและลด เลเยอร์ลงหลายครั้งแต่กลับไม่ได้ผลการทดลองที่น่าพอใจนัก จึงจะขอกล่าวในบทสรุปถัดไป

```
nn = neuralNetwork.NeuralNetwork(len(i[0])-1, 0.01)  # 0.01 Learning rate

nn.addHidden(21)    #   Hidden layers
nn.addHidden(18)
nn.addHidden(15)
nn.addHidden(13)
nn.addHidden(10)

nn.addHidden(21)    #   output layer
```

การ Normalize และการ เทรน

จะเห็นได้ว่ามีการตั้งจำนวนการเทรนไว้ที่ 20 รอบ เนื่องจากการเทรนแต่ละรอบกินระยะว่าประมาณ 20 วินาที จึงไม่ได้สามารถที่จะเพิ่มจำนวนรอบได้มากนักเนื่องจากข้อจำกัดของเวลา

```
                input.normalize(i[0]) # training set
                input.normalize(i[1]) # test set

                for r in range(20):   # epoch
                    for j in range(len(i[0])):  #feed each row

                            inp = list(i[0].iloc[j][:-1])
                            expect = i[0].iloc[j][-1:]
                            nn.train(inp, expect)
```

การ ทดสอบของ neural network

ทดสอบด้วยการตรวจสอบว่าคลาสที่ออกได้ เท่ากับผลลัพธ์หรือไม่ หากถูกต้อง จะเป็น 1 หากไม่จะเป็น 0 เลย ไม่ได้มีการประเมินความใกล้เคียง เช่น หากผลได้ 9 แต่ที่ถูกต้องคือ 10 ก็จะยังถูกนับว่าเป็น 0

```
                for j in range(len(i[1])):  #feed test
                        inp = list(i[1].iloc[j][:-1])
                        expect = i[1].iloc[j][-1:]
                        nn.setInput(inp)
                        nn.allProcess()
                        o = nn.classify()
                        if o == int(expect):
                            success += 1
```

ผลการการทดสอบ

FOLD	ACCURACY	FOLD	ACCURACY
1	0.23076923076923078	6	0.10256410256410256
2	0.10256410256410256	7	0.1282051282051282
3	0.15384615384615385	8	0.10256410256410256
4	0.1794871794871795	9	0.23076923076923078
5	0.10256410256410256	10	0.2564102564102564
AVERAGE ACCURACY		**0.158974358974358935**	

สรุปผลการทดสอบของ neural network

จากการทดสอบเห็นได้ว่า ค่าความถูกต้องที่ได้นั้นน้อยมาก แสดงให้เห็นถึงความผิดพลาดอย่างชัดเจน คาดว่าความผิดพลาดเกิดมาจากการที่กำหนดจำนวนรอบในการเทรนน้อยเกินไป และไม่ได้มีการ optimization การคำนวณที่ดี จึงส่งผลให้เกิดความล่าช้าในการคำนวณและทำให้รอบการเทรนน้อยกว่าที่ควรจะเป็น

ทั้งยังไม่ได้ทดสอบกับโครงสร้าง neural network ที่หลากหลายมากพอ กล่าวคือ ไม่ได้ทดสอบกับ layer ที่หลากหลายขึ้น และ จำนวนnode ในแต่ละเลเยอต่าง ๆ กันไป

ซึ่งเป็นความผิดพลาดทองผู้จัดทำที่ไม่ได้ทำให้ละเอียดและรอบคอบมากกว่านี้

Decision tree

เป็นการสร้าง tree สำหรับการ classify ข้อมูลตาม algorithm ID4.5 โดยในการสร้าง tree นี้ได้ตั้งสมมุติฐานว่า ในแต่ละชั้นของ tree จะแบ่ง class ของ เป้าหมายให้ออกเป็น 2 คลาส โดยจะใช้การแบ่งจากจุดกึ่งกลางของคลาสที่เป็นไปได้ ยกตัวอย่างเช่น root node จะมีคลาสเป็นเป็นได้ 21 คลาส โดยจะทำการแบ่งเป็นสองคลาส คือ 0-9 และ 10-20 และใช้ ID4.5 เลือก attribute ในการแบ่ง และจะทำไปเรื่อย ๆ จนสุด

โครงสร้างของ class Decision tree

```
class DecisionTree:

    class Node:
        def __init__(self, target):
            self.target = target
            self.child = []
            self.attr = ''
            self.avaiable_attr = []
            self.theshold = 0

        def classify(self, input):
            if len(self.target) == 1:

                return self.target[0]
            else:
                o = input[self.attr]
                if o <= self.theshold:
                    return self.child[0].classify(input)
                else:
                    return self.child[1].classify(input)
    #   END Node
    def __init__(self, target):
        self.tree = []
#   END DecisionTree
```

Function สำหรับสร้าง decision tree

```
def buildTree(data, root, attr, count=1 ):
    root_node = root
    if len(root_node.target) <= 1:
        pass
    else:
        m_p = len(root_node.target) / 2
        l_t = root_node.target[:math.floor(m_p)]
        r_t = root_node.target[math.floor(m_p):]
        d_l = data.loc[data['G3'] <= l_t[-1:][0]]
        d_r = data.loc[data['G3'] >  l_t[-1:][0]]
        d_ls = len(d_l)
        d_rs = len(d_r)
        info = getInfo(d_ls, d_rs)
        attr_info = []
        g_info = []
        for i in root_node.avaiable_attr:
            x, y = attrInfoGain(i, 'G3', data, d_l, d_r)
            attr_info.append(x)
            g_info.append(y)
        gain = np.full(len(attr_info), info) - attr_info
        max = [0,0]
        for n, i in enumerate(gain):
            tmp = i/g_info[n]
            if tmp > max[0]:
                max = [tmp, n]
        root_node.attr = root_node.avaiable_attr.pop(max[1])
        root_node.theshold = findMid(data, root_node.attr)
        if count == 0:
            l_node = DecisionTree.Node(l_t)
            r_node = DecisionTree.Node(r_t)
        else:
            l_node = DecisionTree.Node(root_node.target)
            r_node = DecisionTree.Node(root_node.target)
        l_node.avaiable_attr = cp.copy(root_node.avaiable_attr)
        r_node.avaiable_attr = cp.copy(root_node.avaiable_attr)

        root_node.child.append(l_node)
        root_node.child.append(r_node)

        buildTree(data, l_node, 'G3')
        buildTree(data, r_node, 'G3')
```

ผลการทำงานของ Decision tree

FOLD	ACCURACY	FOLD	ACCURACY
1	0.10256410256410256	**6**	0.10256410256410256
2	0.1794871794871795	**7**	0.1282051282051282
3	0.15384615384615385	**8**	0.2564102564102564
4	0.07692307692307693	**9**	0.20512820512820512
5	0.07692307692307693	**10**	0.05128205128205128
AVERAGE ACCURACY		**0.1102564102564102 29**	

สรุปผลการทดลอง

เห็นได้ว่าผลที่ได้จาก Decision tree มีความแม่นยำค่อนข้างต่ำ ถึงต่ำมาก เพราะจากสมมุติฐานที่ตั้งขึ้นมา จะทำให้ tree มีความสูงได้เพียง $log_2(n)$ เท่า ซึ่งจะได้ใช้ attribute ในการclassify เพียงเท่าความสูงเท่านั้น จึงส่งผลให้ ความแม่นยำต่ำถึงขนาดนี้

เมื่อนำมาเปรียบเทียบกับ neural network จะเห็นว่า neural network มีความแม่นยำที่มากกว่าเพียงเล็กน้อยและยังถูกจัดว่า ต่ำ อยู่ดี จึงได้ข้อสรุปที่ว่า

- ควรให้เวลาในการเทรน neural network มากกว่านี้
- ควรศึกษา algorithm ของ decision มาใหม่ เพราะว่าสมมุติฐานที่ตนตั้งขึ้นมาหละหลวมเกินไป
- ควรเริ่มทำงานให้ไวกว่านี้ เพราะงานที่ได้นอกจากไม่มีเวลาให้พอแล้ว ยังไม่มีเวลาในการศึกษา decision tree ให้เข้าใจยิ่งขึ้น
- งานนี้ทำตามความเข้าใจของตนและความรู้ที่รวบรวมมาได้ ผลลัพธ์ความแม่นยำจึงเป็นดังที่เห็น

ภาคผนวก

code ของ neural network

```
import numpy as np
import copy as cp

class NeuralNetwork:
    class Layer:
        def __init__(self, s, input, type='fully'):
                self.size = s
                self.input = input
                self.nodes = []
                self.output = []
                self.type = type   # fully / direct
                self.setupNodes()

        def setupNodes(self):
            if self.type == 'fully':
                for i in range(self.size):
                    i_size = len(self.getInput())
                    rand_w = self.getRandW(i_size)
                    rand_b = self.getRandBias()
                    # print('rand_w', rand_w)
                    # print('rand_b', rand_b)
                    tmp_node = self.LinearNode(self.getInput(), rand_w, rand_b)
                    self.nodes.append(tmp_node)
            else:
                for i in range(self.size):
                    tmp_node = self.ActivationNode(self.getInput()[i], 'sig')
                    self.nodes.append(tmp_node)

        def process(self):
            tmp = []
            for n, i in enumerate(self.nodes):
                if self.type == 'fully':
                    i.setInput(self.getInput())
                else:
                    i.setInput(self.getInput()[n])
                i.process()
                tmp_out = i.getOutput()
                tmp.append(tmp_out)
            self.output = tmp
            return tmp
```

```
        def getInput(self):
            if self.input[0].__class__ == list:
                ip = self.input[0]
            else:
                ip = self.input
            return ip
        def getOutput(self):
            return self.output
        def getRandW(self, size):
            return 1 * np.random.random(size) - 0   #(-1,1)
        def getRandBias(self, size=1):
            return 2 * np.random.random(size) - 1   #(-1,1)

        def setInput(self, input):
            self.input = input

        class Node:
            def __init__(self, n_input=[]):
                self.input = n_input
                self.output = 0.0

            def process(self, out):
                self.setOutput(out)

            def getInput(self):
                return self.input
            def getOutput(self):
                return self.output

            def setInput(self, n_input):
                self.input = n_input
            def setOutput(self, value):
                self.output = value

            # END Node class #
```

```python
class LinearNode(Node):
    def __init__(self, n_input=[], n_w=[], n_b=0):
        super().__init__(n_input)
        self.setWeight(n_w)
        self.bias = n_b

    def applyW(self, delta):
        # print('DELTA ', len(delta))
        # print('W ', len(self.weight))

        w_tmp = []
        for n, i in enumerate(delta):
            # tmp = []
            dump = self.weight[n]
            # print('Before', self.weight[n])
            dump += i
            # print('applying ', i)
            # print('After', self.weight[n])
            w_tmp.append(dump)
        # print('New tmp',w_tmp)
        self.weight = w_tmp

    def getSumWeight(self):
        sum = 0
        for i in range(len(self.input)):
            sum += self.input[i] * self.weight[i]
        return sum
    def setWeight(self, n_weight):
        if len(n_weight) == len(self.input):
            self.weight = n_weight
        else:
            raise ValueError('Weigths size not match the Input size')

    def getWeight(self):
        return self.weight

    def process(self):
        sumwb = self.getSumWeight() + self.bias
        super().process(sumwb)
    # END LinearNode class #
```

```
        class ActivationNode(Node):
            def __init__(self, n_input=[], f='sig'):
                self.func = f
                super().__init__(n_input)

            def activationFunction(self, input, f='sig', order=0):
                if order == 0:
                    if f == 'sig':  #sigmoid
                        return 1.0/(1+np.exp(0-input))
                    else:   #tanh
                        return np.tanh(input)
                else:
                    if f == 'sig':
                        fx = self.activationFunction(input,f,order-1)
                        return fx * (1 - fx)
                    else:
                        fx = self.activationFunction(input,f,order-1)
                        return 1 - np.power(fx, 2)

            def process(self):
                fx = self.activationFunction(self.getInput(), self.func)
                super().process(fx)
            # END ActivationNode class #

    def __init__(self, n, lr=0.5):
        self.input = np.full(n, 0)
        self.layers = []
        self.learning_rate = lr

    def addHidden(self, n):
        layer_size = len(self.layers)
        ip = []
        if layer_size == 0:
            ip = self.input
        else:
            ip = self.layers[layer_size-1].getOutput()
        self.addLinearLayer(n, ip)
        ip = self.layers[layer_size].getOutput()
        self.addActivationLayer(n, ip)
        self.allProcess()

    def addLinearLayer(self, n, input):
        tmp_layer = self.Layer(n, input)
        tmp_layer.process()
        self.layers.append(tmp_layer)
```

```
    def addActivationLayer(self, n, input):
        tmp_layer = self.Layer(n, input, 'direct')
        tmp_layer.process
        self.layers.append(tmp_layer)

    def setInput(self, input):
        size = len(self.layers)
        self.input = input
        if size != 0:
            self.layers[0].input = input

    def getLayerList(self):
        for i in self.layers:
            print('%6s | %d Nodes'%(i.type, i.size))
            for j in i.nodes:
                # print('%sInput :%s'%(''.ljust(7), j.input))
                try:
                    print('%sWeight :%s'%(''.ljust(9), j.weight))
                    print('\n')
                    print('%sBias :%s'%(''.ljust(9), j.bias))
                except:
                    pass
                # print('%sOutput :%s'%(''.ljust(11), j.output))
            print('\n')

    def getInput(self):
        return self.input
    def getOutput(self):
        return self.layers[-1:][0].getOutput()

    def allProcess(self, v=False):
        if v:
            print('\nBegin All process----------')
        interm = []
        for n, i in enumerate(self.layers):
            if v:
                print('Begin New Layer')
            if n != 0:
                i.setInput(interm)
            interm = i.process()
        if v:
            print('End All process----------\n')
```

```
    def classify(self):
        return np.argmax(self.getOutput())

    def train(self, input, expect):
        self.setInput(input)
        self.allProcess()
        exp = self.classGen(int(expect),len(self.layers[-1:][0].getOutput()))
        grad = []
        o = []
        for n, i in enumerate(reversed(self.layers)):   #find gradiant
            if n % 2 == 0:
                ot = i.getOutput()
                for j in ot:
                    o.append([j, i.nodes[0].activationFunction(j,order=1)])
                continue
            g_tmp = []
            for nj, j in enumerate(i.nodes):
                dedn = 0
                if n == 1:
                    dedn = o[nj][0] - exp[nj]
                else:
                    sum = 0
                    for nk, k in enumerate(grad[-1:][0]):
                        sum += k* self.layers[(len(self.layers)-1)-(n-
2)].nodes[nk].getWeight()[nj]
                    dedn = sum
                g_tmp.append(dedn*o[nj][1])
            grad.append(g_tmp)
            o = []
        o = []
        for n, i in enumerate(reversed(self.layers)):   #apply weight
            count = 0

            if n % 2 == 0:
                o = i.getOutput()
                continue
            else:
                for nj, j in enumerate(i.nodes):
                    delta = []
                    for nx, x in enumerate(j.weight):
                        a = grad[count][nj]
                        b = o[nj]
                        dt = 0 - self.learning_rate * a * b
                        delta.append(dt)
                    j.applyW(delta)
```

```
                    j.bias = j.bias - self.learning_rate * grad[count][nj] *
j.bias
                count += 1
        self.allProcess()
        return 0

    def classGen(self, out, size):
        t = np.full(size, 0)
        t[out] = 1
        return t

    # END NeuralNetwork class #
```

code การทำงาน ของ decision tree

```
import numpy as np
import math
import copy as cp

class DecisionTree:

    class Node:
        def __init__(self, target):
            self.target = target
            self.child = []
            self.attr = ''
            self.avaiable_attr = []
            self.theshold = 0

        def classify(self, input):
            if len(self.target) == 1:
                # print(self.target[0])
                return self.target[0]
            else:
                o = input[self.attr]
                if o <= self.theshold:
                    # print('Go l')
                    return self.child[0].classify(input)
                else:
                    # print('Go r')
                    return self.child[1].classify(input)
    #   END Node
```

```
    def __init__(self, target):
        self.tree = []
#   END DecisionTree

def findMid(data, attr):
    agg = data.sort_values(by=[attr])
    mid = (agg[attr].max() - agg[attr].min())/2
    return mid

def buildTree(data, root, attr, count=1 ):
    root_node = root
    if len(root_node.target) <= 1:
        pass
    else:
        m_p = len(root_node.target) / 2
        l_t = root_node.target[:math.floor(m_p)]
        r_t = root_node.target[math.floor(m_p):]
        d_l = data.loc[data['G3'] <= l_t[-1:][0]]
        d_r = data.loc[data['G3'] >  l_t[-1:][0]]
        d_ls = len(d_l)
        d_rs = len(d_r)
        info = getInfo(d_ls, d_rs)
        attr_info = []
        g_info = []
        for i in root_node.avaiable_attr:
            x, y = attrInfoGain(i, 'G3', data, d_l, d_r)
            attr_info.append(x)
            g_info.append(y)
        gain = np.full(len(attr_info), info) - attr_info
        max = [0,0]
        for n, i in enumerate(gain):
            tmp = i/g_info[n]
            if tmp > max[0]:
                max = [tmp, n]
        root_node.attr = root_node.avaiable_attr.pop(max[1])
        root_node.theshold = findMid(data, root_node.attr)
        if count == 0:
            l_node = DecisionTree.Node(l_t)
            r_node = DecisionTree.Node(r_t)
        else:
            l_node = DecisionTree.Node(root_node.target)
            r_node = DecisionTree.Node(root_node.target)
        l_node.avaiable_attr = cp.copy(root_node.avaiable_attr)
        r_node.avaiable_attr = cp.copy(root_node.avaiable_attr)
```

```
        root_node.child.append(l_node)
        root_node.child.append(r_node)
        buildTree(data, l_node, 'G3')
        buildTree(data, r_node, 'G3')


def getInfo(ai, bi):
    a = ai + 1
    b = bi + 1
    s = a+b
    return (-(a/s)*math.log2(a/s)) - (b/s)*math.log2(b/s)

def attrInfoGain(s_attr, o_attr, data, d_l, d_r):
    t_l = d_l[[s_attr, o_attr]]
    t_r = d_r[[s_attr, o_attr]]
    mp = findMid(data, s_attr)
    # print(mp, s_attr)
    l_p = len(t_l.loc[t_l[s_attr] <= mp])
    l_n = len(t_l.loc[t_l[s_attr] >  mp])
    r_p = len(t_r.loc[t_r[s_attr] <= mp])
    r_n = len(t_r.loc[t_r[s_attr] >  mp])
    # print(l_p, l_n, r_p, r_n)
    sum = l_p + l_n + r_p + r_n
    return ((l_p+l_n)/sum) * (getInfo(l_p, l_n)) + ((r_p+r_n)/sum) *
(getInfo(r_p, r_n)), getInfo(l_p, r_p)
```

Code ส่วนการ รับไฟล์และอื่น ๆ

```
import numpy as np
import pandas as pd
import os
import math
input_file = {1:'student-mat.csv', 2:'student-por.csv'}

def loadFile(file_number=1):
    if not file_number in input_file:
        file_number = 1

    fpath = os.path.join('dataset', input_file[file_number])
    return pd.read_csv(fpath, sep=';', header=0, na_values='?')def kFold(k,
data):
    data_len = len(data)
    bin_size = math.floor(data_len / k)
    remainder = data_len % bin_size
    res = []
    print('dLen', data_len)
    print('bSize', bin_size)
    print('rem', remainder)

    shared_train = data.tail(remainder)
    data.drop(data.tail(remainder).index,inplace=True)
    new_len = len(data)

    for i in range(k):
        if i == 0:
            train = data.tail(new_len - (bin_size * (i+1)))
            test = data.head(bin_size)
        elif i == k-1:
            train = data.head(i * bin_size)
            test = data.tail(bin_size)
        else:
            train = data.head(i * bin_size).append(data.tail(new_len - (bin_size
* (i+1))))
            test = data.head((i+1) * bin_size).tail(bin_size)
        train = train.append(shared_train)
        res.append([train, test])

    return res
```

```
def normalize(input):
    input['school'] = input['school'].apply(lambda x:binary('GP', x))
    input['sex'] = input['sex'].apply(lambda x:binary('F', x))
    input['age'] = input['age'].apply(lambda x:scaling(15, 22, x))
    input['address'] = input['address'].apply(lambda x:binary('U', x))
    input['famsize'] = input['famsize'].apply(lambda x:binary('LE3', x))
    input['Pstatus'] = input['Pstatus'].apply(lambda x:binary('T', x))
    input['Medu'] = input['Medu'].apply(lambda x:scaling(0, 4, x))
    input['Fedu'] = input['Fedu'].apply(lambda x:scaling(0, 4, x))
    input['Mjob'] = input['Mjob'].apply(lambda x:nominal(['teacher', 'health',
'services', 'at_home', 'other'], x))
    input['Fjob'] = input['Fjob'].apply(lambda x:nominal(['teacher', 'health',
'services', 'at_home', 'other'], x))
    input['reason'] = input['reason'].apply(lambda x:nominal(['home',
'reputation', 'course', 'other'], x))
    input['guardian'] = input['guardian'].apply(lambda x:nominal(['mother',
'father', 'other'], x))
    input['traveltime'] = input['traveltime'].apply(lambda x:scaling(1, 4, x))
    input['studytime'] = input['studytime'].apply(lambda x:scaling(1, 4, x))
    input['failures'] = abs(input['failures'].apply(lambda x:scaling(1, 4, x)))
    input['schoolsup'] = input['schoolsup'].apply(lambda x:binary('yes', x))
    input['famsup'] = input['famsup'].apply(lambda x:binary('yes', x))
    input['paid'] = input['paid'].apply(lambda x:binary('yes', x))
    input['activities'] = input['activities'].apply(lambda x:binary('yes', x))
    input['nursery'] = input['nursery'].apply(lambda x:binary('yes', x))
    input['higher'] = input['higher'].apply(lambda x:binary('yes', x))
    input['internet'] = input['internet'].apply(lambda x:binary('yes', x))
    input['romantic'] = input['romantic'].apply(lambda x:binary('yes', x))
    input['famrel'] = input['famrel'].apply(lambda x:scaling(1, 5, x))
    input['freetime'] = input['freetime'].apply(lambda x:scaling(1, 5, x))
    input['goout'] = input['goout'].apply(lambda x:scaling(1, 5, x))
    input['Dalc'] = input['Dalc'].apply(lambda x:scaling(1, 5, x))
    input['Walc'] = input['Walc'].apply(lambda x:scaling(1, 5, x))
    input['health'] = input['health'].apply(lambda x:scaling(1, 5, x))
    input['absences'] = input['absences'].apply(lambda x:scaling(0, 93, x))
    input['G1'] = input['G1'].apply(lambda x:scaling(0, 20, x))
    input['G2'] = input['G2'].apply(lambda x:scaling(0, 20, x))
    # input['G3'] = input['G3'].apply(lambda x:scaling(0, 20, x))
```

```
def scaling(min, max, x):
    return (x - min)/(max - min)

def binary(a, x):
    return 1 if x == a else 0

def nominal(p_list, x):
    for n, i in enumerate(p_list):
        if i == x:
            return scaling(0, len(p_list)-1, n)
```

Code ส่วนการทำงานหลัก

```
import input
import neuralNetwork
import decisionTree as dt
import sys

if __name__ == '__main__':
    raw_data = input.loadFile().sample(frac=1)   #Load and shuffle data
    kf_data = input.kFold(10, raw_data)

    error = []
    for n, i in enumerate(kf_data): #each fold
        if 0 <= n < 10:
            print('%2d   ::   Train size : %d | Test size : %d'%(n+1, len(i[0]),
len(i[1])))
            print('%s'%(''.ljust(50,'-')))

            if sys.argv[1] == 'nn': #   Neuron network
                nn = neuralNetwork.NeuralNetwork(len(i[0])-1, 0.01)

                nn.addHidden(21)    #   Hidden layers
                nn.addHidden(18)
                nn.addHidden(15)
                nn.addHidden(13)
                nn.addHidden(10)

                nn.addHidden(21)    #   output layer

                input.normalize(i[0])
                input.normalize(i[1])
```

```
            for r in range(20):   # epoch
                print(r)
                for j in range(len(i[0])):  #feed each row

                        inp = list(i[0].iloc[j][:-1])
                        expect = i[0].iloc[j][-1:]
                        nn.train(inp, expect)

            success = 0
            for j in range(len(i[1])):  #feed test
                # if 0 <= j < 5:
                    inp = list(i[1].iloc[j][:-1])
                    expect = i[1].iloc[j][-1:]
                    nn.setInput(inp)
                    nn.allProcess()
                    o = nn.classify()
                    if o == int(expect):
                        success += 1
            # nn.getLayerList()
            t_e = success/float(len(i[1]))
            print("This error :", t_e)
            error.append(t_e)

        else:   #Decision tree
            input.normalize(i[0])
            input.normalize(i[1])
            target = [xp for xp in range(21)]
            root = dt.DecisionTree.Node(target)
            root.avaiable_attr = list(i[0])[:-1]
            dt.buildTree(i[0], root, 'G3')
            success = 0
            for j in range(len(i[1])):
                result = root.classify(i[1].iloc[j])
                if result == int(i[1].iloc[j]['G3']):
                    success += 1
            t_e = success/float(len(i[1]))
            print("This error :", t_e)
            error.append(t_e)


a_error = sum(error) / 10.0
print('ERROR is', a_error)
```