实验报告

课程名称：数据挖掘

学　　院：计算机科学与工程学院

专　　业：软件工程　班　　级：2016-1 班

姓　　名：孔令鑫　学　　号:201601060810

2019 年 5 月 20 日

山 东 科 技 大 学 教 务 处 制

实验报告 _____页

组　　别		姓　名	孔令鑫	同组实验者	
实验项目名称	Apriori 算法挖掘数据中的频繁项集			实验日期	4 月 3 日
教师评语					
实验成绩：			指导教师（签名）： 年　月　日		

一、实验目的

1.加强对 Apriori 算法的理解；

2.锻炼分析问题、解决问题并动手实践的能力；

二、实验内容

编程实现 Apriori 算法；

三、实验要求

编程实现 Apriori 算法，加深对其理解。

四、实验准备

1.看懂 Apriori 算法的基本思想；

2.上网查阅相关资料。

五、问题

一个超级市场的销售系统记录了顾客购物的情况。表中记录了 5 个顾客的购物单。超市经理想知道商品之间的关联，要求列出那些同时购买的、且支持度≥40%的商品名称。

六、算法思想

Apriori 算法是一种最有影响力的挖掘布尔关联规则的频繁项集的算法，它是由 Rakesh Agrawal 和 Ramakrishnan Skrikant 提出的。它使用一种称作逐层搜索的迭代方法，k-项集用于探索（k+1-项集。首先，找

出频繁 1-项集的集合。该集合记作 L1。L1 用于找频繁 2-项集的集合 L2，而 L2 用于找 L2，如此下去，直到不能找到 k-项集。

七、实验步骤

（1）、预处理：数据来自课件，把数据统计到 excel 表格中，为了显示方便，使用字母代替具体的商品，每组有关联的商品组合为一行，导出为.csv 文件（每个单项之间用“,”分隔）。

序号	商品代码序列
1	A,C,D
2	B,C,E
3	A,B,C,E
4	B,E

（2）运行环境及语言：

Windows 10 专业版、JetBrains PyCharm 2017.3.2 x64、python 3.6

（3）伪代码：

```
def apriori():
 Ck: Candidate itemset of size k
 Lk : frequent itemset of size k
 L1 = {frequent items};
 for (k = 1; Lk !=∅; k++) do begin
     Ck+1 = candidates generated from Lk;
     self-joining Lk
     pruning
     for each transaction t in database do
         Calculate support of Ck+1
         Lk+1  = candidates in Ck+1 more than min_support
 end
 return   k Lk;

 def self-joining (Lk-1) :
 insert into Ck
 select p.item1, p.item2, …, p.itemk-1, q.itemk-1
 from Lk-1 p, Lk-1 q
```

```
    where p.item1=q.item1, …, p.itemk-2=q.itemk-2, p.itemk-1 < q.itemk-1

    def pruning():
    forall itemsets c in Ck do
    forall (k-1)-subsets s of c do
    if (s is not in Lk-1) then delete c from Ck
```

（4）代码见附录

（5）实验结果：

```
    初始数据: [['A', 'C', 'D'], ['B', 'C', 'E'], ['A', 'B', 'C', 'E'], ['B', 'E']]
    统计每个单项的支持度：
    Lk: k=1  {'A'} support:  0.5
    Lk: k=1  {'C'} support:  0.75
    Lk: k=1  {'B'} support:  0.75
    Lk: k=1  {'E'} support:  0.75
    开始查找频繁 2 项集：
    Ck: [frozenset({'C', 'A'}), frozenset({'B', 'A'}), frozenset({'A', 'E'}), frozenset({'B',
'C'}), frozenset({'C', 'E'}), frozenset({'B', 'E'})]
    Lk: k=2  {'C', 'A'} support:  0.5
    Lk: k=2  {'B', 'C'} support:  0.5
    Lk: k=2  {'C', 'E'} support:  0.5
    Lk: k=2  {'B', 'E'} support:  0.75
    开始查找频繁 3 项集：
    Ck: [frozenset({'C', 'A', 'E'}), frozenset({'B', 'C', 'E'})]
    Lk: k=3  {'B', 'C', 'E'} support:  0.5
    开始查找频繁 4 项集：
    Ck: []
```

（6）实验结果分析

Apriori 算法简单易懂，从 1 项开始，不断的生成候选项集，再根据最小支持度筛选出频繁项集，直到频繁项集为空集。从结果可以看出，单项集中达到 0.4 支持度的有 4 项，2 项集中达到 0.4 的有 4 项，3 项集中有 1 项，到 4 项集中，已经没有达到最小支持度 0.4 的。算法结束。

八、遇见的问题及解决方法

遇见的第一个问题就是，在 python 中 set 放入 dict 之后报错 unhashable 的问题，就不能实现对应每个项进行扫描数据库统计支持度，经过上网查阅相关的资料，最终使用 frosenset，即冻结的集合，它是不可变的，存在哈希值。只有 hashable 的元素才能放入 list 和 dict 等集合。还有一些自己在实现的时候没有完全清楚思路就动手的问题，以后要避免。

九、实验心得

这是数据挖掘的第一个实验，自己独立解决的第一个数据挖掘问题，真正体会到了数据挖掘课程的魅力，正好在最近的数学建模问题中可以运用到了 apriori 算法，解决了找出出险情况与驾驶习惯之间的关联关系，apriori 简单易学习，容易理解，但运用 python 实现算法对动手能力要求还是很高的，好久不用 python，写了几个 bug，浪费了一些时间，还好都解决了。

代码（详见我的 github: https://github.com/skpupil/dataMining）：

```
import copy
def apriori(dataSet, minSupport):#apriori 算法主框架函数
    print("统计每个单项的支持度：")
    C1 = createcandidates(dataSet)# 构建初始候选项集 C1
    D = [set(var) for var in dataSet]#形成集合
    L1, LS = scanDataSet(D, C1, minSupport)# 构建初始的频繁项集
    L = [L1]# L 初始为最初的 L1
    k = 2# 项集应该含有 2 个元素，所以 k=2
    while (len(L[k - 2]) > 0):
        print("开始查找频繁"+str(k)+"项集：")
        Ck = aprioriGen(L[k - 2], k)
        print("Ck: "+str(Ck))
        Ck2 = has_infrequent_subset(L, Ck, k)# 剪枝，减少计算量
        Lk, supK = scanDataSet(D, Ck2, minSupport)# 候选项集支持度和最小支持度进
          行比较,得到 Lk
        LS.update(supK)# 将新的项集的支持度数据加入原来的结果字典中
        L.append(Lk)#添加生成的频繁项集
        k += 1#k 结束，k 自增
    return L[:-1], LS# 返回所有满足条件的频繁项集的列表，和所有候选项集的支持度信
          息

def scanDataSet(D, Ck, minSupport):#扫描数据库，统计支持度
    subSetCount = {}
    for tid in D:
        for can in Ck:
            if can.issubset(tid):# 检查候选 k 项集中的每一项的所有元素是否都出现在
        每一个事务中，若 true，则加 1
    subSetCount[can] = subSetCount.get(can, 0) + 1# subSetCount 为候选支持度计数，
        get()返回值，如果值不在字典中则返回默认值 0。
    numItems = float(len(D))
```

```
    returnList = []
    supportData = {}# 选择出来的频繁项集，未使用先验性质
    for key in subSetCount:
        support = subSetCount[key] / numItems  # 每个项集的支持度
        if support >= minSupport:  # 将满足最小支持度的项集，加入 returnList
            print("Lk: k=" + str(len(key)) + "  " + str(set(key)) + " support:  " + str(support))
            returnList.insert(0, (key))
            supportData[key] = support  # 汇总支持度数据
    return returnList, supportData

def aprioriGen(Lk, k):  # Aprior 算法生成候选 Ck
    Ck = []
    for i in range(len(Lk)):
        L1 = list(Lk[i])[: k - 2]# 只需取前 k-2 个元素相等的候选频繁项集即可组成元
             素个数为 k+1 的候选频繁项集
        for j in range(i + 1, len(Lk)):
            L2 = list(Lk[j])[: k - 2]
            L1.sort()
            L2.sort()
            if L1 == L2:#集合的 union 方法
                Ck.append((Lk[i]) | (Lk[j]))
    return Ck

def has_infrequent_subset(L, Ck, k):#剪枝，任一频繁项集的所有非空子集也必须是频繁
                的，反之，如果某个候选的非空子集不是频繁的
    Ckc = copy.deepcopy(Ck)#复制
    for i in Ck:
        p = [t for t in i]
        i_subset = binaryfindsub(p)
        subsets = [i for i in i_subset]
        for each in subsets:
            if each!=[] and each!=p and len(each)<k:
                if frozenset(each) not in [t for z in L for t in z]:
                    Ckc.remove(i)
                    break
    return Ckc

def binaryfindsub(items): #二进制枚举找找子集
    N = len(items)
    for i in range(2**N):
        combo = []
        for j in range(N):
            if(i >> j ) % 2 == 1:
                combo.append(items[j])
```

```
        yield combo

def readData(fileName):
    f=open(fileName,'r',encoding="utf-8")
    lines=f.readlines()
    dataSet=[]
    for line in lines[0:]:
        print("line: "+line)
        line=line.strip().split(',')
        dataSet.append(line)
    print(str(dataSet))
    return dataSet

def createcandidates(dataSet):#构建初始单项集
    C1 = []
    for transaction in dataSet:
        for item in transaction:
            if [item] not in C1:
                C1.append([item])
    C1.sort()
    return [frozenset(var) for var in C1]

if __name__ == '__main__':
    myDat = readData("./dataOfApriori.csv")
    print("初始数据:  "+str(myDat))
    L, LS = apriori(myDat, 0.4)
```

实验报告

组　　别		姓　名	孔令鑫	同组实验者	
实验项目名称	FP-Tree 算法挖掘数据中的频繁项集			实验日期	4 月 17 日
教师评语					
实验成绩：			指导教师（签名）： 年　月　日		

一、实验目的

1.加强对 FP-Tree 算法的理解；

2.锻炼分析问题、解决问题并动手实践的能力；

二、实验内容

编程实现 FP-Tree 算法；

三、实验要求

编程实现 FP-Tree 算法，加深对其理解。

四、实验准备

1.看懂 FP-Tree 算法的基本思想；

2.上网查阅相关资料。

五、问题：同实验一。

六、算法思想

FP-growth 算法是基于 Apriori 原理的，通过将数据集存储在 FP（Frequent Pattern)树上发现频繁项集，但不能发现数据之间的关联规则。FP-growth 算法只需要对数据库进行两次扫描，而 Apriori 算法在求每个潜在的频繁项集时都需要扫描一次数据集，所以说 Apriori 算法是不高效的。其中算法发现频繁项集的过程是：1、扫描数据库 TDB，得到 F-list;

2、扫描数据库 TDB，构建最初的 FP-tree;

3、自底向上构建各单频繁项的条件 FP-tree 进行挖掘；（找出 P 的条件模式基础，再构建 P 的条件模式树）FP 表示的是频繁模式，其通过链接来连接相似元素，被连起来的元素可以看成是一个链表。将事务数据表中的各个事务对应的数据项按照支持度排序后，把每个事务中的数据项按降序依次插入到一棵以 NULL 为根节点的树中，同时在每个结点处记录该结点出现的支持度。FP-growth 算法的流程为：首先构造 FP 树，然后利用它来挖掘频繁项集。在构造 FP 树时，需要对数据集扫描两边，第一遍扫描用来统计频率，第二遍扫描至考虑频繁项集。

七、实验步骤

（1）预处理

把数据统计到 excel 表格中，每行代表一个元组，导出为.csv 文件（每个单项之间用“,”分隔）

序号	商品代码序列
1	a，b，c，d，e，f，g，h
2	a，f，g
3	b，d，e，f，j
4	a，b，d，i，k
5	a，b，e，g

（2）运行环境：

Windows 10 专业版、JetBrains PyCharm 2017.3.2 x64、python 3.6

（3）伪代码：

输入：已经构造好的 FP-Tree，项集 α（初值为空），最小支持度 min_sup;

输出：事务数据集 D 中的频繁项集 L;

```
L 初值为空
if Tree 只包含单个路径 P then
    for 路径 P 中节点的每个组合（记为 β） do
        产生项目集 α ∪ β，其支持度 support 等于 β 中节点的最小支持度数;
```

```
            return L = L ∪ 支持度数大于 min_sup 的项目集 β∪α
    else        //包含多个路径
        for Tree 的头表中的每个频繁项 αf do
            产生一个项目集 β = αf∪α，其支持度等于 αf 的支持度;
            构造 β 的条件模式基 B，并根据该条件模式基 B 构造 β 的条件 FP- 树 Treeβ;
            if Treeβ≠Φ then
                递归调用 FP-Growth(Treeβ, β);
            end if
        end for
    end if
```

（4）实验代码见附录

（5）实验结果

```
最小支持度： 3
初始数据： [['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'], ['a', 'f', 'g'], ['b', 'd', 'e', 'f',
'j'], ['a', 'b', 'd', 'i', 'k'], ['a', 'b', 'e', 'g']]
树建立完成，开始查找频繁模式

构造['g']-条件的 FP-Tree 前： ['g'] 支持度： 3
构造['a', 'g']-条件的 FP-Tree 后： ['a', 'g'] 支持度： 3

构造['f']-条件的 FP-Tree 前： ['f'] 支持度： 3

构造['e']-条件的 FP-Tree 前： ['e'] 支持度： 3
构造['b', 'e']-条件的 FP-Tree 后： ['b', 'e'] 支持度： 3

构造['d']-条件的 FP-Tree 前： ['d'] 支持度： 3
构造['b', 'd']-条件的 FP-Tree 后： ['b', 'd'] 支持度： 3

构造['b']-条件的 FP-Tree 前： ['b'] 支持度： 4
```

构造['a', 'b']-条件的 FP-Tree 后： ['a', 'b'] 支持度： 3

构造['a']-条件的 FP-Tree 前： ['a'] 支持度： 4

（6）实验结果分析：

实验结果与课件中的正确结果吻合，以['g']为叶子节点，可以得到频繁模式['a', 'g']，以['f']为叶子节点，找不到满足最小支持度的频繁模式，后面['e']、['d']、['b']，分别找到频繁模式['b','e']、['b','d']、['a', 'b']，['a']为叶子节点找不到满足最小支持度的频繁模式。程序结束。

八、遇见的问题及解决办法

FP-Tree 中树的构建和操作是难点，思路虽然清晰，但代码实现起来比较复杂，遇见了通过叶子节点去找父亲的问题，首先想到了通过一个单独的函数，遍历整棵树，写完之后看网上的代码，觉得完全没有必要增加算法复杂度，在定义节点加一个元素指向它的父节点即可。之后参考网上的写法，进行了重构。

九、实验心得

FP-Tree 相比于 apriori 算法有很明显的优势，它用到了数据结构中树的知识，只需两次扫描数据库，然后构建一颗高度压缩的 FP-tree，再在这棵树中进行递归挖掘，减少大量扫描数据库的 I/O 时间；把问题进行了分解，先建立树结构，再找叶子节点的条件模式基础，构造频繁模式；不用产生大量候选项，再进行计算，筛选，减少了计算量；在事务平均长度较长且密集的数据集下效果好。

代码（见我的 github: https://github.com/skpupil/dataMining）：

```
import collections
import itertools

class node:# 定义节点
    def __init__(self, val, char):
        self.val = val  # 用于定义当前的计数
        self.char = char  # 用于定义当前的字符是多少
        self.children = {}  # 用于存储孩子
        self.next = None  # 用于链表，链接到另一个孩子处
        self.father = None  # 构建条件树时向上搜索
        self.visit = 0  # 用于链表的时候观察是否已经被访问过了
        self.nodelink = collections.defaultdict()
        self.nodelink1 = collections.defaultdict()

class FPTree():
    def __init__(self):
        self.root = node(-1, 'root')
        self.FrequentItem = collections.defaultdict(int)  # 用来存储频繁项集
        self.res = []

    def BuildTree(self, data):  # 建立 fp 树的函数,data 应该以 list[list[]]的形式，其中内部的 list 包含了商品的名称，以字符串表示
        for line in data:  # 取出第一个 list，用 line 来表示
            root = self.root
            for item in line:  # 对于列表中的每一项
                if item not in root.children.keys():  # 如果 item 不在 dict 中
                    root.children[item] = node(1, item)  # 创建一个新的节点
                    root.children[item].father = root  # 用于从下往上寻找
                else:
                    root.children[item].val += 1  # 否则，计数加 1
                root = root.children[item]  # 往下走一步
                if item in self.root.nodelink.keys():  # 创建链表，如果这个 item 在 nodelink 中已经存在了
                    if root.visit == 0:  # 如果这个点没有被访问过
                        self.root.nodelink1[item].next = root
                        self.root.nodelink1[item] = self.root.nodelink1[item].next
                        root.visit = 1  # 被访问了

                else:  # 如果这个 item 在 nodelink 中不存在
                    self.root.nodelink[item] = root
                    self.root.nodelink1[item] = root
```

```
                    root.visit = 1
        print('树建立完成，开始查找频繁模式')
        return self.root

    def IsSinglePath(self, root):
        # print('是否为单路径')
        if not root:
            return True
        if not root.children: return True
        a = list(root.children.values())
        if len(a) > 1:
            return False
        else:
            for value in root.children.values():
                if self.IsSinglePath(value) == False: return False
            return True
    def FP_growth(self, Tree, a, HeadTable):  # Tree 表示树的根节点，a 用列表表示的频繁项集,HeadTable 用来表示头表
        # 我们首先需要判断这个树是不是单路径的，创建一个单路径函数 IsSinglePath(root)
        if self.IsSinglePath(Tree):  # 如果是单路径的
            # 对于路径中的每个组合，记作 b，产生模式，b 并 a，support = b 中节点的最小支持度
            root, temp = Tree, []  # 创建一个空列表来存储
            while root.children:
                for child in root.children.values():
                    temp.append((child.char, child.val))
                    root = child
            ans = []# 产生每个组合
            for i in range(1, len(temp) + 1):
                ans += list(itertools.combinations(temp, i))
            # print('ans = ',ans)
            for item in ans:
                mychar = [char[0] for char in item] + a
                mycount = min([count[1] for count in item])
                if mycount >= support:
                    print("构造"+str(mychar)+"-条件的 FP-Tree 后：  "+str(mychar)+" 支持度："+str(mycount))
                    #print()
                    self.res.append([mychar, mycount])
        else:  # 不是单路径，存在多个路径
            root = Tree
            HeadTable.reverse()  # 首先将头表逆序

            for (child, count) in HeadTable:  # child 表示字符，count 表示支持度
                b = [child] + a  # 新的频繁模式
```

```
                # 开始构造 b 的条件模式基
                print()
                print("构造"+str(b)+"-条件的 FP-Tree 前： "+str(b) + " 支持度： "+str(count))
                self.res.append([b, count])
                tmp = Tree.nodelink[child]  # 此时为第一个节点从这个节点开始找,tmp 一直保持在链表当中
                data = []  # 用来保存条件模式基
                while tmp:  # 当 tmp 一直存在的时候
                    tmpup = tmp  # 准备向上走
                    res = [[], tmpup.val]  # 用来保存条件模式

                    while tmpup.father:
                        res[0].append(tmpup.char)
                        tmpup = tmpup.father

                    res[0] = res[0][::-1]  # 逆序
                    data.append(res)  # 条件模式基保存完毕
                    tmp = tmp.next
                # 条件模式基构造完毕，储存在 data 中，下一步是建立 b 的 fp-Tree

                # 统计词频
                CountItem = collections.defaultdict(int)
                for [tmp, count] in data:
                    for i in tmp[:-1]:
                        CountItem[i] += count

                for i in range(len(data)):
                    data[i][0] = [char for char in data[i][0] if CountItem[char] >= support]  # 删除掉不符合的项
                    data[i][0] = sorted(data[i][0], key=lambda x: CountItem[x], reverse=True)  # 排序

                # 此时数据已经准备好了，我们需要做的就是构造条件树
                root = node(-1, 'root')  # 创建根节点，值为-1，字符为 root
                for [tmp, count] in data:  # item 是 [list[],count] 的形式

                    tmproot = root  # 定位到根节点
                    for item in tmp:  # 对于 tmp 中的每一个商品
                        # print('123',item)
                        # CountItem1[item] += 1
                        if item in tmproot.children.keys():  # 如果这个商品已经在 tmproot 的孩子中了
                            tmproot.children[item].val += count  # 更新值
                        else:  # 如果这个商品没有在 tmproot 的孩子中
```

```
                            tmproot.children[item] = node(count, item)  # 创建一个新的节点
                            tmproot.children[item].father = tmproot  # 方便从下往上找
                        tmproot = tmproot.children[item]  # 往下走一步

                        # 根据这个 root 创建链表
                        if item in root.nodelink.keys():  # 这个 item 在 nodelink 中存在
                            if tmproot.visit == 0:
                                root.nodelink1[item].next = tmproot
                                root.nodelink1[item] = root.nodelink1[item].next
                                tmproot.visit = 1
                        else:  # 这个 item 在 nodelink 中不存在
                            root.nodelink[item] = tmproot
                            root.nodelink1[item] = tmproot
                            tmproot.visit = 1

                if root:  # 如果新的条件树不为空
                    NewHeadTable = sorted(CountItem.items(), key=lambda x: x[1], reverse=True)

                    for i in range(len(NewHeadTable)):
                        if NewHeadTable[i][1] < support:
                            NewHeadTable = NewHeadTable[:i]
                            break

                    self.FP_growth(root, b, NewHeadTable)  # 我们需要创建新的 headtable

                # return root#成功返回条件树

    def PrintTree(self, root):  # 层次遍历打印树
        if not root: return
        res = []
        if root.children:
            for (name, child) in root.children.items():
                print(str(name)+" "+str(child.val))
                res += [name + ' ' + str(child.val), self.PrintTree(child)]
            return res
        else:
            return

if __name__ == '__main__':
    data = readData("./dataOfFPTree.csv")#readData 与 apriori 中相同，本程序省去
    print("初始数据：  "+str(data))
    data = data
    support = 3
    print("最小支持度：  "+support)
```

```
    CountItem = collections.defaultdict(int)# 统计单项的频率
    for line in data:
        for item in line:
            CountItem[item] += 1
    a = sorted(CountItem.items(), key=lambda x: x[1], reverse=True)# 将 dict 按照频率从大到小排序,并且删除掉频率过小的项
    for i in range(len(a)):
        if a[i][1] < support:
            a = a[:i]
            break
    for i in range(len(data)):# 更新 data 中，每一笔交易的商品顺序
        data[i] = [char for char in data[i] if CountItem[char] >= support]
        data[i] = sorted(data[i], key=lambda x: CountItem[x], reverse=True)
    obj = FPTree()
    root = obj.BuildTree(data)
    obj.FP_growth(root, [], a)
```

实验报告

组　　别		姓　名	孔令鑫	同组实验者	
实验项目名称	ID3、C4.5、CART 决策树			实验日期	5 月 11 日
教师评语					
实验成绩：		指导教师（签名）： 年　月　日			

一、实验目的

1.加强对 ID3、C4.5、CART 算法的理解；

2.锻炼分析问题、解决问题并动手实践的能力；

二、实验内容

编程实现 ID3、C4.5、CART 算法；

三、实验要求

编程实现 ID3、C4.5、CART 算法，加深对其理解。

四、实验准备

1.看懂 ID3、C4.5、CART 算法的基本思想；

2.上网查阅相关资料。

五、问题

根据给出的数据，挖掘其中有价值的信息，找到什么样的人购买电脑，以便进行精准推销。

六、算法思想

ID3：事件 a_i 的信息量 $I(a_i)$ ：

$$I(a_i) = p(a_i)\log_2 \frac{1}{p(a_i)}$$

假设有 n 个互不相容的事件 $a_1, a_2 ... a_i$,它们中有且仅有一个发生，则其平均的信息量可如下度量：

$$I(a_1, a_2, ..., a_n) = \sum_{i=1}^{n} I(a_i) = \sum_{i=1}^{n} p(a_i)\log_2 \frac{1}{p(a_i)}$$

熵越大，随机变量的不确定性就越大。那么构造一颗决策树，就是为了让决策树的叶子节点的熵降低直到为 0,也就是说所有叶子节点的分类都是明确的，它的信息没有任何不确定性，这时我们就完成了决策树的构建。

上式，对数底数可以为任何数，不同的取值对应了熵的不同单位。通常取 2，并规定当 p(a_i)=0 时

$$I(a_i) = p(a_i)\log_2 \frac{1}{p(a_i)}$$

C4.5 建树过程与 ID3 相似，只是采用信息增益率来划分子树：

$$GainRatio(D, a) = \frac{Gain(D, a)}{IV(a)}$$

$$IV(a) = -\sum_{v=1}^{V} \frac{|D^v|}{|D|}\log_2 \frac{|D^v|}{|D|}$$

CART 选择使子节点的 GINI 值或者回归方差最小的属性作为分裂的方案。

分类树：

$$Gain = \sum_{i \in I} p_i \cdot Gini$$

$$Gini = 1 - \sum_{i \in I} p_i^{\,2}$$

回归树：

$$Gain = \sum_{i \in I} \sigma_i$$

$$\sigma = \sqrt{\sum_{i \in I} x_i^2 - n\mu^2}$$

剪枝：表面误差率增益值的计算公式：

$$\alpha = \frac{R(t) - R(T)}{N(T) - 1}$$

七、实验步骤

数据预处理：

计数	年龄	收入	学生	信誉	买不买计算机
64	青	高	否	良	不买
64	青	高	否	优	不买
128	中	高	否	良	买
60	老	中	否	良	买
64	老	低	是	良	买
64	老	低	是	优	不买
64	中	低	是	优	买
128	青	中	否	良	不买
64	青	低	是	良	买
132	老	中	是	良	买
64	青	中	是	优	买
32	中	中	否	优	买
32	中	高	是	良	买
63	老	中	否	优	不买
1	老	中	否	优	买

课件中的数据导入 excel 表格中，导出为.csv 文件（以“,”分隔）。

（2）运行环境：

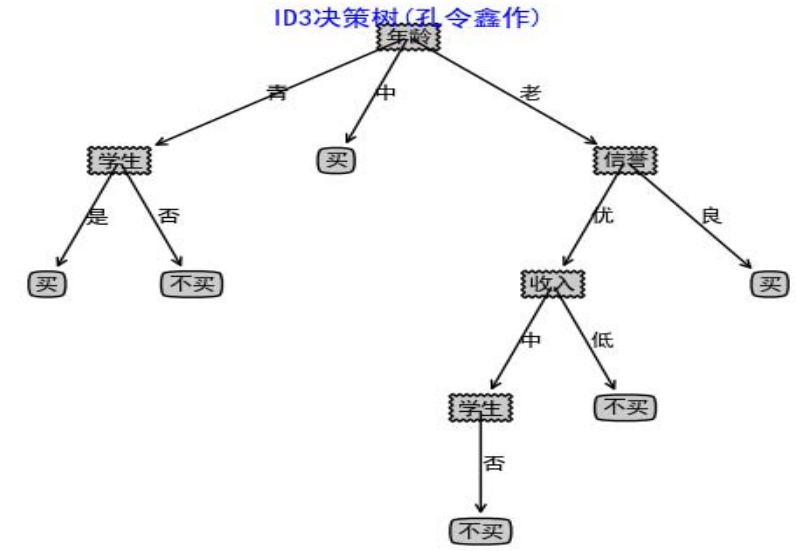
Windows 10 专业版、JetBrains PyCharm 2017.3.2 x64、python 3.6

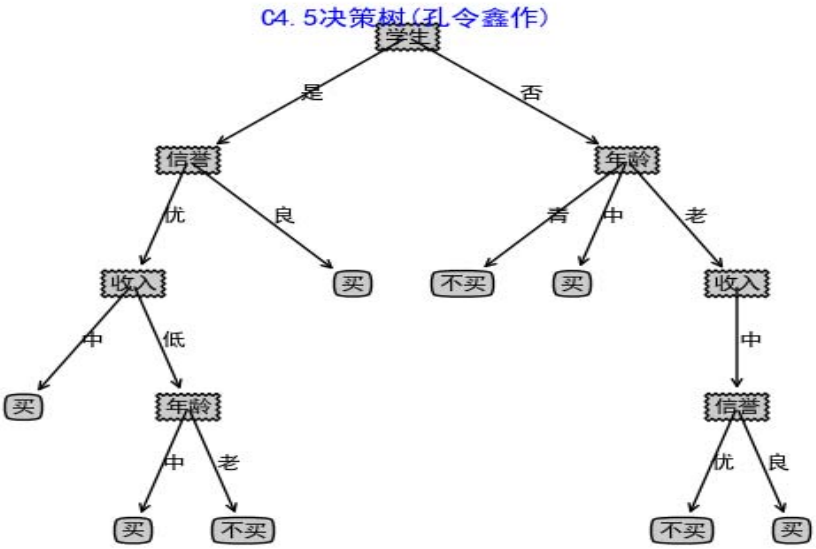
（3）伪代码

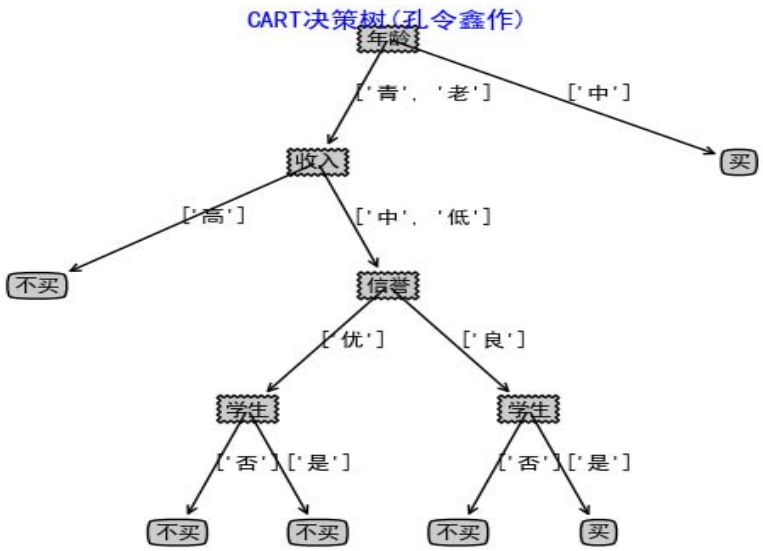
```
int ID3(Node *p){
    double HD=Empirical_entropy(p);//信息熵
    double HDA[10];//条件熵
    for i=0:p->attrsize:
        HDA[i]=Conditional_entropy(i,p);
        G[i]=HD-HDA[i]//信息增益
    return 信息增益最大的特征;
}
int C4_5(Node* p){
    double HD=Empirical_entropy(p);
    for i=0:p->attrsize:
```

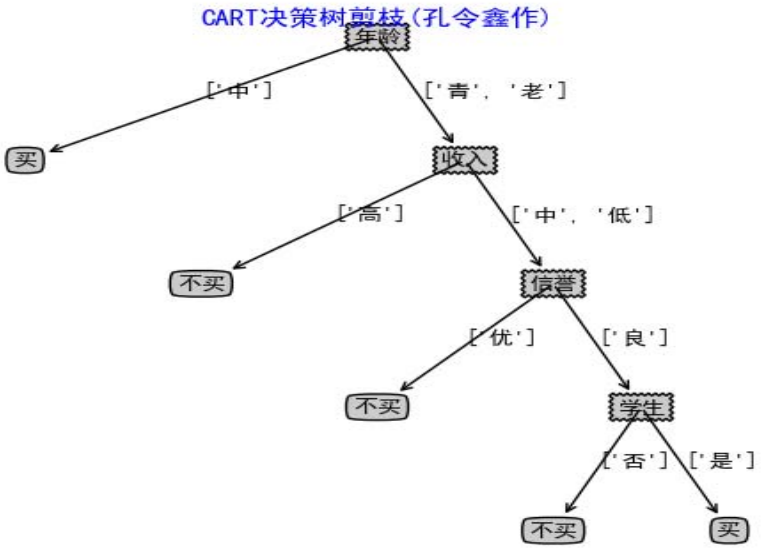
```
        HDA=Conditional_entropy(i,p);//条件熵
        double sp_info=SplitInfo(i,p);//特征的熵
        gainRatio[i]=(HD-HDA)/sp_info;//信息增益率
    return 信息增益率最大的特征;
}
double gini(int attr_index,Node* p){
    return 索引位置为 attr_index 的特征的 gini 系数 ;
}
int CART(Node* p){
    double Gini[10];
    for i=0:p->attrsize:
        Gini[i]=gini(i,p);
    return 基尼系数最小的特征;
}
void decide_final_label(Node* p){
    多数投票，在数据集中    正标签多 return 1;     负标签多 return -1;   }
bool meet_with_bound(Node* p){
    if 符合边界条件一：所有数据集样本的标签都相同; return 1;
    if 边界条件二：所有数据集的每个特征的取值都相同; return 1;
    else return 0;//即继续向下分支
}
void recursive(Node *p){
    if(meet_with_bound(p)){
        decide_final_label(p);//决定这个叶子结点的标签
        return;
    }
    未达到边界，则继续向下分支
    int attr_chosen=choose_attr(p);//选择一个当前特征集中最优代表性的特征
    int index_chosen;
    找出这个被选中的特征在特征集里的索引位置即 index_chosen;
    int maxnum,minnum;
    分别求出特征取值的最大最小值;
    for i=minnum:maxnum :
        Node *tem=new Node;
        新节点的特征集是父节点特征集除去被选中的哪一个特征后所剩的特征集;
        新节点的数据集是父节点被选中特征的取值为 i 的那些数据样本的集合（删掉被选中特征那一列）
        if 被选中特征确实有 i 这个取值
            将 tem 与父节点连接起来
        else 释放 tem 节点
    for i=0:子节点数目:
        recursive(p->children[i]);
}
```

（4）实验结果：


ID3决策树(孔令鑫作)
年龄
青
中
老
学生
买
信誉
是
否
优
良
买
不买
收入
买
中
低
学生
不买
否
不买


C4.5决策树(孔令鑫作)
学生
是
否
信誉
年龄
优
良
青
中
老
收入
买
不买
买
收入
中
低
中
买
年龄
信誉
中
老
优
良
买
不买
不买
买





（5）实验结果分析

从结果可以看出，ID3、C4.5、CART 得到的树结构均不相同，所使用的数据集充分体现了三种决策树算法的区别。

ID3 是根据信息增益进行树的分裂，在结算信息增益的时候，对于属性值数目多的属性，如年龄分为老中青，得出的信息增益明显大于 2 个属性值的属性，所以，更多属性值的属性进行了优先分裂。

C4.5 算法是对 ID3 算法的改进，ID3 算法是用信息增益来选择特征的，而信息增益的缺点是比较偏向选择取值较多的特征，如果有的特征取值比其他特征的取值多很多的话，这个特征基本就直接被认为是最重要的

特征，但实际却未必。因此在 C4.5 中，为了改善这种情况，引入了对取值数量的惩罚，得到信息增益率作为特征重要性的衡量标准。如果某一个特征取值越多，那么对他的惩罚越严重，其信息增益率也能得到控制。CART 与以上两种算法区别最为明显，首先，ID3 和 C4.5 可以是多叉树，而 CART 是二叉树，因此，在构建 CART 树时，必须先对多属性进行枚举，计算 Gain，根据 Gain 小的进行划分。在属性选择时，CART 采用 GINI 值进行。gini 系数值越小表示不确定性越小，当一个节点中所有样本都是一个类时，GINI 系数为零。我们选取 gini 系数最小的特征作为决策点。

在根据表面误差增益值进行剪枝，当然，也可以根据测试集误差进行剪枝，对于一棵子树，若在剪掉之后，测试集上的误差反而变大，则不剪掉，否则，剪枝。从结果看来，由于叶子节点的分类采用的多数表决法，因此出现了最终学生为“是”和“否”，结果均为不买的情况。根据表面误差增益率只能剪掉最小的一个子树，即上述子树。如果数据量不大的情况下，我认为可以用表面误差增益值，但是数据量充足的情况下，采用测试集验证误差的方式剪枝更为合理。因为表面误差增益值，把叶子节点的减少量考虑进去了，也就是说，误差增加了，但是叶子节点减少更多，分叉次数明显少了，表面误差增益值仍然很小，就进行剪枝。并不是很合理，只适用小数据集。最好还是采用测试集验证误差的方式剪枝。

八、遇见的问题及解决办法

ID3、C4.5、CART 三个算法，实现起来并不那么容易，尤其是 CART，在把属性化成两组的时候，在分别计算 gini 值，比较麻烦。剪枝的时候遇见 bug，困了我两天，最后发现问题出在进行其他树的建立的时候，破坏了原始数据集，因此用一个变量提前存一下，结果还是不行，经过查阅资料，发现拷贝分深拷贝和浅拷贝，用一个变量用=赋值，两个变量数据共享，=赋值在内存中指向同一个对象，如果是可变类型，修改一个，另一个必定改变，如果是不可变类型如 str，修改其中一个，另一个不会变。因此，我需要进行深拷贝，即重新开辟一块空间，把内容复制进新的空间中。

九、实验心得

决策树是一个完全的可解释性的算法，通俗易懂，通过这次学习决策树，我还了解到了在各个大数据比赛上大放异彩的 xgboost，作为多个 CART 的组合，是解决问题的大杀器，并且，在数学建模网络赛上我学以致用，完成了 xgboost 建模解决实际问题的任务，很开心学到这么有用的东西。

主要代码（读数据程序重复、画图代码程序过繁，省去。全部代码见我的 github）：

```
def calEnt(dataSet):
    print("进入 calENT 的 dataset: "+str(dataSet))
    labelCount={}
    lenn=0
    for line in dataSet:
        currentLabel=line[-1]
        if currentLabel not in labelCount.keys():
            labelCount[currentLabel]=0
        #for i in range(0,line[0]):
        labelCount[currentLabel]+=int(line[0])
        lenn+=int(line[0])
        #print("line[0]: "+line[0])
    print("labelCount:  "+str(labelCount))
    ent=0.0

    for i in labelCount:
        print("---------"+str(labelCount[i])+" "+str(lenn))
        r=float(labelCount[i])/lenn
        ent=ent-r*log(r,2)
    return labelCount,lenn,ent,dataSet

def calGini(dataSet):
    labelCount = {}
    lenn = 0
    for line in dataSet:
        currentLabel = line[-1]
        if currentLabel not in labelCount.keys():
            labelCount[currentLabel] = 0
        # for i in range(0,line[0]):
        labelCount[currentLabel] += int(line[0])
        lenn += int(line[0])
        # print("line[0]: "+line[0])
    print("labelCount:  " + str(labelCount))
    ent = 0.0
    preGini = 1.0
    for i in labelCount:
        print("---------" + str(labelCount[i]) + " " + str(lenn))
        r = float(labelCount[i]) / lenn
        preGini=preGini-r*r
    return preGini
def ID3_chooseBestFeatureToSplit(dataset):#ID3 算法
    print("调用选择属性算法")
```

```
    numFeatures=len(dataset[0])-1
    labelCount,lenn,ent,dataSet=calEnt(dataset)
    baseEnt=ent
    print("baseEnt"+str(baseEnt))
    bestInfoGain=0.0
    bestFeature=-1
    for i in range(1,numFeatures): #遍历所有特征
        #for example in dataset:
            #featList=example[i]
        featList=[example[i]for example in dataset]
        uniqueVals=set(featList) #将特征列表创建成为 set 集合，元素不可重复。创建唯一的分类标签
列表
        newEnt=0.0
        for value in uniqueVals:     #计算每种划分方式的信息熵
            print("ID3 划分："+str(value))
            subdataset,lenn=splitdataset(dataset,i,value)
            p=lenn/float(lenns)#float(len(dataset))(len(subdataset))
            labelCount, lenn, ent, dataSet=calEnt(subdataset)
            newEnt+=p*ent
            print("ID3:ent:"+str(ent))
            print("ID3:p"+str(p))
        print("ID3:allnewent"+str(newEnt))
        infoGain=baseEnt-newEnt
        print(u"ID3 中第%d 个特征的信息增益为：%.4f"%(i,infoGain))
        if (infoGain>bestInfoGain):
            bestInfoGain=infoGain    #计算最好的信息增益
            bestFeature=i
    return bestFeature


def C45_chooseBestFeatureToSplit(dataset):#C4.5 算法
    numFeatures=len(dataset[0])-1
    labelCount, lenn, baseEnt, dataSet=calEnt(dataset)
    bestInfoGain_ratio=0.0
    bestFeature=-1
    for i in range(1,numFeatures): #遍历所有特征
        featList=[example[i]for example in dataset]
        uniqueVals=set(featList) #将特征列表创建成为 set 集合，元素不可重复。创建唯一的分类标签
列表
        newEnt=0.0
        IV=0.0
        for value in uniqueVals:     #计算每种划分方式的信息熵
            print("C4.5 划分：" + str(value))
            subdataset, lenn=splitdataset(dataset,i,value)
```

```
            p=lenn/float(lenns)#len(subdataset)/float(len(dataset))
            labelCount, lenn, ent, dataSet=calEnt(subdataset)
            newEnt+=p*ent
            IV=IV-p*log(p,2)
        infoGain=baseEnt-newEnt
        if (IV == 0): # fix the overflow bug
            continue
        infoGain_ratio = infoGain / IV #这个 feature 的 infoGain_ratio
        print(u"C4.5 中第%d 个特征的信息增益率为：%.3f"%(i,infoGain_ratio))
        if (infoGain_ratio >bestInfoGain_ratio):  #选择最大的 gain ratio
            bestInfoGain_ratio = infoGain_ratio
            bestFeature = i #选择最大的 gain ratio 对应的 feature
    return bestFeature


def CART_chooseBestFeatureToSplit(dataset):#CART 算法

    numFeatures = len(dataset[0]) - 1
    bestGini = float('inf') #初始化为最大值
    bestFeature = -1
    bestHuaFen = []
    for i in range(1,numFeatures):
        featList = [example[i] for example in dataset]
        uniqueVals = set(featList)
        lei = huafenMethod(uniqueVals)
        gini = 1.0
        print("lei:  "+str(lei))
        #eachhuafen
        for eachhuafen in lei:
            print("CART 划分：" + str(eachhuafen))
            gini=0.0
            for eachLei in eachhuafen:
                preGini=1.0
                retdataset,lenn,labelCount=splitdataset2(dataset,i,eachLei)
                p=lenn/float(lenns)
                for labels in labelCount:
                    preGini=preGini-pow(labelCount[labels]/lenns,2)
                gini=gini+p*preGini
                print("Gini: "+str(gini))
                if (gini < bestGini):
                    bestGini = gini
                    bestFeature = i
                    bestHuaFen = eachhuafen
        print(u"CART 中第%d 个特征的基尼值为：%.3f"%(i,gini))
```

```
    return bestFeature,bestHuaFen


def ID3_createTree(dataset,labels):#利用 ID3 算法创建决策树
    print("进入 ID3 create_tree"+str(dataset))
    classList=[]
    classList=[str(example[-1]) for example in dataset]
    print("classList:"+str(classList))
    if classList.count(classList[0]) == len(classList):# 类别完全相同，停止划分
        return classList[0]
    if len(dataset[0]) == 2:# 遍历完所有特征时返回出现次数最多的
        return majorityCnt(dataset)
    bestFeat = ID3_chooseBestFeatureToSplit(dataset)

    print("best:"+str(bestFeat))
    print("labels:"+str(labels))
    bestFeatLabel = labels[bestFeat]
    print(u"此时最优索引为："+(bestFeatLabel))
    ID3Tree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    featValues = [example[bestFeat] for example in dataset]# 得到列表包括节点所有的属性值
    uniqueVals = set(featValues)
    for value in uniqueVals:
        subLabels = labels[:]
        dataSet, lenn =splitdataset(dataset, bestFeat, value)
        #if bestFeat != -1:
        ID3Tree[bestFeatLabel][value] = ID3_createTree(dataSet, subLabels)
    return ID3Tree

def C45_createTree(dataset,labels):#C4.5 创建决策树
    print("进入 C45"+str(dataset))
    classList=[]
    for example in dataset:
        print(example)
        classList.append(example[-1])
    if classList.count(classList[0]) == len(classList):
        # 类别完全相同，停止划分
        return classList[0]
    if len(dataset[0]) == 2:
        # 遍历完所有特征时返回出现次数最多的
        return majorityCnt(dataset)
    bestFeat = C45_chooseBestFeatureToSplit(dataset)
    bestFeatLabel = labels[bestFeat]
    print(u"此时最优索引为："+(bestFeatLabel))
```

```
    C45Tree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    # 得到列表包括节点所有的属性值
    featValues = [example[bestFeat] for example in dataset]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        subLabels = labels[:]
        dataSet, lenn=splitdataset(dataset, bestFeat, value)
        C45Tree[bestFeatLabel][value] = C45_createTree(dataSet, subLabels)
    return C45Tree

def CART_createTree(dataset,labels):#CART 创建决策树
    print("进入 CART 建树：  "+str(dataset))
    classList = []
    for example in dataset:
        print(example)
        classList.append(example[-1])
    if classList.count(classList[0]) == len(classList):# 类别完全相同，停止划分
        return classList[0]
    if len(dataset[0]) == 2:# 遍历完所有特征时返回出现次数最多的
        return majorityCnt(dataset)
    bestFeat,bestHuaFen = CART_chooseBestFeatureToSplit(dataset)
    bestFeatLabel = labels[bestFeat]
    print(u"此时最优索引为："+(bestFeatLabel))
    CARTTree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    featValues = []# 得到列表包括节点所有的属性值
    for classs in bestHuaFen:
        print(str(classs))
        featValues.append(str(classs))
    print((featValues))
    uniqueVals = set(featValues)
    for value in uniqueVals:
        subLabels = labels[:]
        retdataset, lenn, labelCount=splitdataset2(dataset, bestFeat, value)
        CARTTree[bestFeatLabel][value] = CART_createTree(retdataset, subLabels)
    return CARTTree

def majorityCnt(classList):#多数表决法
    #print("进入 major"+str(classList))
    classCont={}
    for vote in classList:
        if vote[-1] not in classCont.keys():
            classCont[vote[-1]]=0
```

```
        classCont[vote[-1]]+=int(vote[0])
    sortedClassCont=sorted(classCont.items(),key=operator.itemgetter(1),reverse=True)
    #print("major 返回了什么"+str(sortedClassCont[0][0]))
    return sortedClassCont[0][0]
def pruningTree(inputTree, dataset,  labels):
    print("inputTree"+str(inputTree))
    firstStr = list(inputTree.keys())[0]
    secondDict = inputTree[firstStr]  # 获取子树
    #classList = [example[-1] for example in dataSet]
    classList = []
    print("dataset"+str(dataset))
    #dataset = dataset[0]
    for example in dataset:
        print("example: "+str(example))
        classList.append(example[-1])

    if classList.count(classList[0]) == len(classList):
        # 类别完全相同，停止划分
        return classList[0]
    if len(dataset[0]) == 2:
        # 遍历完所有特征时返回出现次数最多的
        return majorityCnt(dataset)

    featKey = copy.deepcopy(firstStr)
    labelIndex = labels.index(featKey)
    subLabels = copy.deepcopy(labels)
    del (labels[labelIndex])
    print("jinfor"+str(classList))
    retdataset=[]
    rettestset=[]
    print("list(secondDict.keys())"+str(list(secondDict.keys())))
    print("labelIndex"+str(labelIndex))
    for key in list(secondDict.keys()):
        if isTree(secondDict[key]):
            # 深度优先搜索,递归剪枝
            retdataset,lenn,labelCount = splitdataset2(dataset, labelIndex, key)
            #rettestset, lenn, labelCount = splitdataset2(testData, labelIndex, key)
            print("ret: "+str(retdataset))
            if len(retdataset[0]) > 2 :
                inputTree[firstStr][key] = pruningTree(secondDict[key], retdataset,
copy.deepcopy(labels))
    err = calcTestErr(inputTree, dataset, subLabels)
    print("err"+str(err))
    if  err>0.2 : #testMajor(majorityCnt(dataset), testData):
```

```
        # 剪枝后的表面误差增益超过阈值，不剪枝
        return inputTree
    else:
        # 剪枝，原父结点变成子结点，其类别由多数表决法决定
        print("classList"+str(classList))
        return majorityCnt(dataset)
```

实验报告 _____页

组　　别		姓　名	孔令鑫	同组实验者	
实验项目名称	HMM 算法			实验日期	5 月 18 日
教师评语					
实验成绩：			指导教师（签名）： 年　月　日		

一、实验目的

1.加强对 HMM 算法的理解；

2.锻炼分析问题、解决问题并动手实践的能力；

二、实验内容

根据问题，编程实现 HMM 算法；

三、实验要求

编程实现 HMM 算法，加深对其理解。

四、实验准备

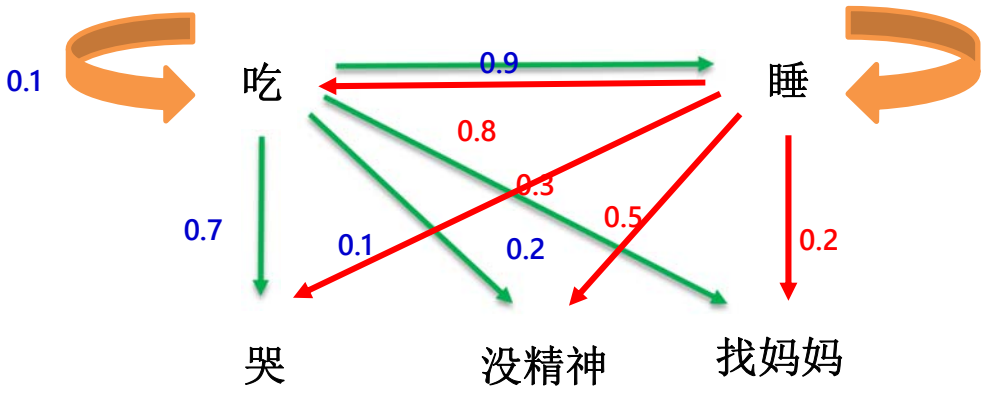
1.看懂 HMM 算法的基本思想；

2.上网查阅相关资料。

五、问题

HMM 模型中，初始化概率 π={吃=0.3,睡=0.7}；隐藏状态 N=2；可观测状态 M=3；转移概率矩阵和生成概率矩阵分别是：

$$A=\begin{pmatrix}0.1 & 0.9\\ 0.8 & 0.2\end{pmatrix} \qquad B=\begin{pmatrix}0.7 & 0.1 & 0.2\\ 0.3 & 0.5 & 0.2\end{pmatrix}$$



编程解决以下两个问题：

1、已知整个模型，宝宝的行为依次是哭 -> 没精神 ->找妈妈，计算产生这些行为的概率。

2、已知整个模型，宝宝的行为依次是哭 -> 没精神 ->找妈妈，计算这三个行为下，宝宝的状态最可能是什么。

六、算法思想

对于 HMM 模型，首先我们假设 Q 是所有可能的隐藏状态的集合，V 是所有可能的观测状态的集合，即：

$$Q=\{q1,q2,...,qN\},V=\{v1,v2,...vM\}$$

其中，N 是可能的隐藏状态数，M 是所有的可能的观察状态数。

对于一个长度为 T 的序列，I 对应的状态序列，O 是对应的观察序列，即：

$$I=\{i1,i2,...,iT\},O=\{o1,o2,...oT\}$$

其中，任意一个隐藏状态 $it\in Q$,任意一个观察状态 $ot\in V$

HMM 模型做了两个很重要的假设如下：

1） 齐次马尔科夫链假设。即任意时刻的隐藏状态只依赖于它前一个隐藏状态，这个我们在 MCMC(二)马尔科夫链中有详细讲述。当然这样假设有点极端，因为很多时候我们的某一个隐藏状态不仅仅只依赖于前一个隐藏状态，可能是前两个或者是前三个。但是这样假设的好处就是模型简单，便于求解。如果在时刻 t 的隐藏状态是 $it=qi$,在时刻 t+1 的隐藏状态是 $it+1=qj$， 则从时刻 t 到时刻 t+1 的 HMM 状态转移概率 aij 可以

表示为：

$$a_{ij} = p(i_t + 1 = q_j \mid i_t = q_i)$$

这样 aij 可以组成马尔科夫链的状态转移矩阵 A:

$$A = [a_{ij}]_{N*N}$$

2） 观测独立性假设。即任意时刻的观察状态只仅仅依赖于当前时刻的隐藏状态，这也是一个为了简化模型的假设。如果在时刻 t 的隐藏状态是 it=qj, 而对应的观察状态为 ot=vk, 则该时刻观察状态 vk 在隐藏状态 qj 下生成的概率为 bj(k),满足：

$$b_j(k) = P(o_t = v_k \mid i_t = q_j)$$

这样 bj(k)可以组成观测状态生成的概率矩阵 B:

$$B = [b_j(k)]_{N*M}$$

除此之外，我们需要一组在时刻 t=1 的隐藏状态概率分布Π:

$$\Pi = [\pi(i)]_N \text{ 其中 } \pi(i) = P(i_1 = q_i)$$

一个 HMM 模型，可以由隐藏状态初始概率分布Π，状态转移概率矩阵 A 和观测状态概率矩阵 B 决定。Π,A 决定状态序列，B 决定观测序列。因此，HMM 模型可以由一个三元组 λ 表示如下：

$$\lambda = (A, B, \Pi)$$

七、实验步骤

（1）数据预处理

把各概率用 dict 存储，详见代码。

（2）运行环境：

Windows 10 专业版、JetBrains PyCharm 2017.3.2 x64、python 3.6

（3）python 代码足够简洁，伪代码省去。

（4）实验结果

第一问答案：序列： ['哭','没精神','找妈妈'] 概率: 0.02688

第二问答案： 矩阵

[[0.21 0.0168 0.01512]

[0.21 0.0945 0.00378]]

第 1 步 吃或睡

第 2 步 睡

第 3 步 吃

（4）实验结果分析

HMM 的实现主要涉及 python 中矩阵的操作，我使用了 numpy，功能强大，大大简化了程序，numpy 也是机器学习中常用的第三方包，值得深入学习，熟练掌握。HMM 算法步骤不难理解，动手实践的时候只需要理清思路即可。

计算过程：

$$\sigma_{11} = P(O_{cry}, S_{eat}) = 0.3*0.7 = 0.21$$
$$\sigma_{12} = P(O_{cry}, S_{slp}) = 0.7*0.3 = 0.21$$

$$\sigma_{21} = P(O_{cry}, O_{tired}, S_{eat}) = MAX(0.21*0.1, 0.21*0.8)*0.1 = 0.0168$$
$$\sigma_{22} = P(O_{cry}, O_{tired}, S_{slp}) = MAX(0.21*0.2, 0.21*0.9)*0.5 = 0.0945$$

$$\sigma_{31} = P(O_{cry}, O_{tired}, O_{find}, S_{eat}) = MAX(0.0168*0.1, 0.0945*0.8)*0.2 = 0.01512$$
$$\sigma_{32} = P(O_{cry}, O_{tired}, O_{find}, S_{slp}) = MAX(0.0945*0.2, 0.0168*0.9)*0.2 = 0.00378$$

八、遇见的问题及解决办法

HMM 算法，实现起来并不那么容易，这是学到第一个和矩阵有密切关系的算法，刚开始没有使用 numpy，用循环进行矩阵操作，十分麻烦，numpy 是机器学习算法实现的一大利器，很有学习价值。遇到的问题主要是 dict 转矩阵，单纯的自己编码，得不到很理想的代码风格，用 numpy 就很好了。还有自己在编码过程中，自己以为自己已经熟悉计算过程了，就没有写下来，导致 bug 频繁，不得不输出一下找 bug。

十、实验心得

经过这几次的数据挖掘实验，大大增强了自己的动手能力，对算法有了更加深刻的理解，每一步的计算过程必须了然于心，才能保证高效的编码。想不好就动手，往往自己挖坑，坑自己，会浪费很多时间。数据挖掘还有很多很有用的算法值得继续深入学习，比如 xgboost 算法，是 CART 的更进一步使用，在大数据竞赛上大放异彩，深度学习方兴未艾，要想在数据挖掘机器学习的学习中更进一步，读研究生很有必要。

主要代码（全部代码见我的 github）：

```
import numpy as np

states = ('吃','睡')
observations = ('哭','没精神','找妈妈')
start_probability = {'吃':0.3,'睡':0.7}
transition_probability = {
    '吃':{'吃':0.1,'睡':0.9},
    '睡':{'吃':0.8,'睡':0.2}
}
emission_probability = {
    '吃':{'哭':0.7,'没精神':0.1,'找妈妈':0.2},
    '睡':{'哭':0.3,'没精神':0.5,'找妈妈':0.2}
}

def generate_index_map(lables):
    id2label = {}
    label2id = {}
    i = 0
    for l in lables:
        id2label[i] = l
        label2id[l] = i
        i += 1
    return id2label, label2id



def convert_map_to_vector(map_, label2id):#将概率向量从 dict 转换成一维 array
    v = np.zeros(len(map_), dtype=float)
    for e in map_:
        v[label2id[e]] = map_[e]
    return v


def convert_map_to_matrix(map_, label2id1, label2id2):#将概率转移矩阵从 dict 转换成矩阵
    m = np.zeros((len(label2id1), len(label2id2)), dtype=float)
    for line in map_:
        for col in map_[line]:
            m[label2id1[line]][label2id2[col]] = map_[line][col]
    return m


def forward(obs_seq):#前向算法
```

```
    N = A.shape[0]
    print("A"+str(A))
    T = len(obs_seq)

    F = np.zeros((N, T))# F 保存前向概率矩阵
    F[:, 0] = pi * B[:, obs_seq[0]]

    for t in range(1, T):
        for n in range(N):
            F[n, t] = np.dot(F[:, t - 1], (A[:, n])) * B[n, obs_seq[t]]
    return F

def viterbi(trainsition_probability,emission_probability,start_probability,observations):
    # 最后返回一个 Row*Col 的矩阵结果
    Row = np.array(trainsition_probability).shape[0]
    Col = len(observations)
    #定义要返回的矩阵
    F=np.zeros((Row,Col))
    #初始状态
    F[:,0]=start_probability*np.transpose(emission_probability[:,observations[0]])
    for t in range(1,Col):
        list_max=[]
        for n in range(Row):
            list_x=list(np.array(F[:,t-1])*np.transpose(trainsition_probability[:,n]))
            #print("list_x: "+str(list_x)+"\nF: "+str(F))#获取最大概率
            list_p=[]
            for i in list_x:
                list_p.append(i*10000)
            list_max.append(max(list_p)/10000)
        F[:,t]=\
            np.array(list_max)*\
            np.transpose(emission_probability[:,observations[t]])
        print("cal 之后："+str(F))
    return F
if __name__ == '__main__':
    states_id2label, states_label2id = generate_index_map(states)
    observations_id2label, observations_label2id = generate_index_map(observations)
    print(states_id2label, states_label2id)
    print(observations_id2label, observations_label2id)
    A = convert_map_to_matrix(transition_probability, states_label2id, states_label2id)
    print("A:"+str(A))
    B = convert_map_to_matrix(emission_probability, states_label2id, observations_label2id)
    print("B"+str(B))
    observations_index = [observations_label2id[o] for o in observations]
```

```
    pi = convert_map_to_vector(start_probability, states_label2id)
    print("pi:: "+str(pi))

    observations = [0,1,2]#,1,2
    F=forward(observations)
    print(str(F))
    ans = 0.0
    for indexx in F:
        ans+=indexx[-1]
    print("第一问答案：序列：  "+str([observations_id2label[index] for index in observations])+"
概率：  %.5f" %(ans))

    F =  viterbi(A,B,pi,observations)
    print("第二问矩阵"+str(F))
    flag = 0
    for yy in range(0,len(observations)):
        maxx = 0
        print("第"+str(yy+1)+"步")
        for xx in range(0,A.shape[0]):
            if maxx==0:
                maxx=F[xx][yy]
            elif F[xx][yy]>maxx :
                maxx=F[xx][yy]
                flag=xx
                print(states_id2label[1])

            elif xx==A.shape[0]-1 and maxx==F[xx][yy]:
                for statuss in states_id2label.keys():
                    print(states_id2label[statuss])
            else:
                print(states_id2label[0])
```