贝叶斯网络

- 手动实现贝叶斯网络构建推理，尝试使用pomegranate库进行贝叶斯网络构建推理。
  - **手动实现构建贝叶斯网络并进行推理求出联合概率和条件概率。**
  - **调用pomegranate库构建贝叶斯网络并进行推理。**
    - **推理时，使用几种方式求出条件概率。**
    - **根据D分离判断条件独立性，从而省略求部分变量联合概率的步骤。**
    - **利用链式法则将联合概率转化为求条件概率。**

（1）算法原理

- 公式说明
  - 变量相互独立

    $P(X|Y)=P(X)$ 或 $P(Y|X)=P(Y)$ 或 $P(X,Y)=P(X)P(Y)$
  - 条件概率

    $P(Y|X)=\frac{P(X,Y)}{P(X)}$
  - 贝叶斯规则

    $P(Y|X)=\frac{P(X|Y)P(Y)}{P(X)}$

    $P(X,Y)=P(X|Y)P(Y)=P(Y|X)P(X)$
  - 链式法则

    $P(X_1,X_2,\ldots,X_n)=P(X_1|X_2,\ldots,X_n)*P(X_2|X_3,\ldots,X_n)*\ldots*P(X_{n-1}|X_n)*P(X_n)$

- 贝叶斯网络-变量定义

  根据所求的问题，归纳需要求解的所有变量，将每个变量用一个结点表示，对变量进行排序。

- 贝叶斯网络-结构学习
  - 从给定的数据集进行学习，得出贝叶斯网络结构，确定变量与变量之间的因果关系。
  - 基于评分搜索方法
    - 评分函数
      - 基于贝叶斯统计评分函数

        根据贝叶斯公式$P(Y|X)=\frac{P(X|Y)P(Y)}{P(X)}$，对于样本集X，假设网络拓扑结构Y的先验概率为$P(Y)$，从而得出网络结构Y的后验概率，通过比较后验概率的大小选出最优。K2评分是其中一种。
      - 基于信息论的评分函数

在贝叶斯基础上进行优化，使其不依赖于先验概率，对于比较大的独立样本，可以接近于抽样分布。

        - 搜索策略
            - 启发式搜索算法

              在状态空间中对每一个搜索位置进行评估，得到最好的位置，再从该位置进行搜索直到目标，可以节省大量无效的搜索代价，提高搜索效率。
    - 基于约束的方法
        - 对训练数据集统计测试，包括条件独立性测试，确定变量之间的依赖关系-条件独立性，通过构造有向无环图从而表示变量之间的关系，其中信息论中信息流的度量可以衡量条件独立性，通常采用互信息和条件互信息进行变量之间条件独立性测试。
    - 基于评分搜索和基于约束相结合方法
        - 将两种方法的优点进行结合，首先使用基于约束方法初步建立贝叶斯网络，再采用贪婪搜索方法基于该网络在等价网络结构空间中搜索最优网络结构。
    - 随机抽样
        - 马尔可夫蒙特卡罗算法
        - 将随机抽样思想引入到贝叶斯网络结构学习中，虽然可以提高学习精度，但是选择的算法有限，就目前的MHS算法来说，在抽样过程中融合性低，收敛速度慢。

- 贝叶斯网络-参数学习
    - 最大似然估计

      在假定标签的前提下发生对应特征的概率，这是可以完全基于已有的数据集，通过计算在每个标签下对应各个特征出现的频数，计算其频率，然后求出其联合概率。
    - 贝叶斯估计

      在最大似然估计基础上，加入先验概率，根据贝叶斯公式，由于分母都一样，因此可以根据前面两项最大化后验概率。
    - EM算法
        - E步骤：对于有缺失的数据，对数据进行添加增补，使其成为完整的数据集。
        - M步骤：根据修补的完整的数据集，计算其最大似然估计，从而得出新的一轮参数概率，对其进行更新学习。

- **贝叶斯网络-推理**

  推理是本实验的重要部分，推理分别使用pomegranate库实现和手动构建贝叶斯网络推理实现。
    - pomegranate
        - pomegranate是一种概率模型，可以解决贝叶斯网络和马尔可夫模型问题，对于贝叶斯，它将概率模型视为概率分布，根据样本和相关的权重进行更新。
        - 在本实验中，首先将每个变量的相关取值的概率存储，如果是单一变量，则直接使用离散分布表存储，如果是涉及因果关系的变量，需要使用条件概率分布表存储；
        - 声明变量结点，存储变量的名字、取值概率分布。
        - 使用库函数构建贝叶斯网络。
        - 往网络中添加变量结点，同时将具有因果关系的变量建立连接。

        - 对于联合概率，有两种方法，一是将其扩展为求全概率，循环遍历累加得到结果，二是使用链式法则将其转化为条件概率进行求解；对于条件概率，使用内置函数直接预测给定变量条件下，查询变量的概率。
    - 手动构建贝叶斯网络和手动推理

      (变量符号以task2为例)

        - 声明类结点存储每一个变量，包括变量的名字，该变量涉及的所有变量，变量各种取值的概率。
        - 构建贝叶斯网络，给每一个变量创建一个类结点，将变量的因果关系视作父子结点，如果变量B是依赖于变量A，则变量A为父结点，变量B为子结点。
        - 将当前变量的所有父结点（包括自身）的变量名存储进自身的变量列表，同时使用字典存储变量各种取值对应的概率，键值表示当前变量以及父结点变量的各种取值情况。
        - 求全概率时，每个变量都有固定的值，利用贝叶斯公式和链式法则扩展为每个变量结点中指定值概率的相乘值，比如对于$P(B,E,A,J,M)$，我们可以将其化为：

          $P(B,E,A,J,M)=P(B)*P(E)*P(A|B,E)*P(J|A)*P(M|A)$

          对每个结点取特定值，求出每个概率值，将其累乘得到全概率。
        - 求联合概率$P(B,E)$时，将其扩展为求全概率$P(B,E,A,J,M)$，将已知变量B,E设为指定值，循环遍历累加所有未知变量A,J,M取值的全概率得到最终的结果。
        - 求边缘概率$P(J)$方法跟求联合概率一样。
        - 求条件概率$P(J|B,E)$时，使用条件概率公式将其扩展为$P(J|B,E)=\frac{P(J,B,E)}{P(B,E)}$，按照求联合概率的方式分别求出$P(J,B,E)$，$P(B,E)$，再相除得出结果。

(2) 伪代码

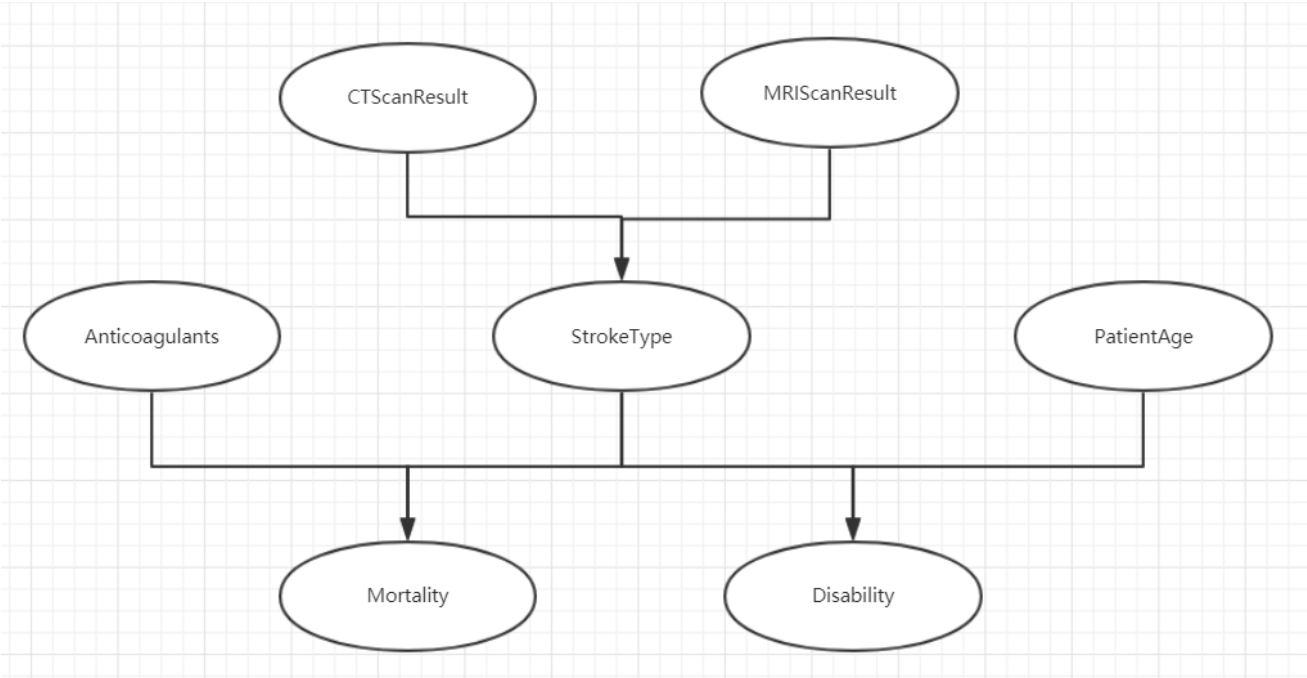
- pomegrante库直接调用即可，不用写伪代码，只写手动构建贝叶斯网络和推理的伪代码。

```
#计算条件概率
Algorithm condition_probability
  input: query,condition #查询变量和已知变量及其对应的值
  output: rate #概率
    #利用链式公式条件概率转化为求分子和分母的联合概率
    take conditional probability into two joint probability a and b
    #分别求出两者的联合概率
    a <- joint_probability(query+condition)
    b <- joint_probability(condition)
    #相除为条件概率
    return a/b

#计算联合概率
Algorithm joint_probability
  input: query #已知变量及其取值
  output: rate #概率
    rate <= 0
    #对于其余未知变量，遍历所有可能取值，分别求出其全概率并进行相加
    for every value in unknown variables:
        rate += full_probability(all variables of specific value)

    #最后结果即为已知变量的联合概率
```

```
    return rate

#求全概率
Algorithm Full_probability
  input: node_list,variable_list #变量结点集合，变量已知值集合
  output: rate #概率
    rate <- 1
    #按顺序遍历每个结点
    for node in node_list:
        #根据变量取值集合更新变量结点的取值，得出其概率
        update node according to specific value of variables from variable_list
        #将得出的概率进行累乘
        rate <- rate * node.rate
    #最后结果为全概率
    return rate
```

(3) 贝叶斯网络结构图

- task3结构图



(4) 关键代码

A. pomegranate库进行贝叶斯网络构建和推理。

- 以Diagnosing问题为例。
- 对各个变量的取值建立离散分布。

```
#构建离散分布，对于只涉及一个变量的单节点，只需要输入该变量的各个取值范围即可
P = DiscreteDistribution({0:0.10,1:0.30,2:0.60})
C = DiscreteDistribution({0:0.7,1:0.3})
Mr = DiscreteDistribution({0:0.7,1:0.3})
A = DiscreteDistribution({0:0.5,1:0.5})
```

- 对于有因果关系的变量，建立条件概率分布表。（以变量S为例，其他不列举）

```
#对于变量S，是基于变量C和变量Mr得出的条件概率，因此需要建立条件概率表，将数据输入
S = ConditionalProbabilityTable(
    [[0,0,0,0.8],
     [0,1,0,0.5],
     [1,0,0,0.5],
     [1,1,0,0],

     [0,0,1,0],
     [0,1,1,0.4],
     [1,0,1,0.4],
     [1,1,1,0.9],

     [0,0,2,0.2],
     [0,1,2,0.1],
     [1,0,2,0.1],
     [1,1,2,0.1]
    ],[C,Mr])
```

- 对每一个变量声明一个结点存储，然后初始化取值范围概率和名字。

```
#声明各个变量的结点，将其取值分布和变量名字作为参数传入
a1 = State(P,name='P')
a2 = State(C,name='C')
a3 = State(Mr,name='Mr')
a4 = State(S,name='S')
a5 = State(A,name='A')
a6 = State(Mo,name='Mo')
a7 = State(D,name='D')
```

- 构建贝叶斯网络。

```
#调用BayesianNetwork函数构建贝叶斯网络
bayes_model = BayesianNetwork("Diagnosing Problem")
#添加结点到贝叶斯网络中
bayes_model.add_states(a1,a2,a3,a4,a5,a6,a7)
```

- 由于变量之间有因果关系，因此有条件依赖性，需要对有因果关系的变量建立连接。

```
#根据条件概率表，将各个变量的连接关系依次建立
bayes_model.add_transition(a2,a4)
bayes_model.add_transition(a3,a4)
bayes_model.add_transition(a5,a6)
bayes_model.add_transition(a4,a6)
bayes_model.add_transition(a4,a7)
bayes_model.add_transition(a1,a7)
#运行贝叶斯网络
bayes_model.bake()
```

- **两种方法进行推理求条件概率**
- 一是扩展循环对联合概率求和的方式求出：（以第一小问的做法为例，其他不再重复）
  - 求变量C和P时，根据D分离可知两变量是条件独立的，因此可以直接相乘，简化运算。

```
#将条件概率展开为分子/分母形式，分别求分子和分母的联合概率
    t1 = 0
    t2 = 0
    #求P、C、M的联合概率，将其进行扩展，进行循环遍历，将所有符合条件的全概率相加
    for i1 in range(2):
        for i2 in range(3):
            for i3 in range(2):
                for i4 in range(3):
                    t1 += bayes_model.probability([0,0,i1,i2,i3,1,i4])
    #求变量P和C的联合概率，由于两者是条件独立的，因此可以直接相乘，避免展开求联合概率
    t2 = 0.10 * 0.70
    p1 = t1 / t2
```

- 二是通过predict_proba求出基于已知变量值的剩余变量的取值概率即条件概率求出。

```
    #利用predict_proba可以得出已知变量条件下其他变量的各个取值概率，
    #其返回是一个列表，列表中的元素是离散分布，定位到指定的变量，
    #同时对其进行probability求解特定的值，即可得到对应的概率。
    pr1 = bayes_model.predict_proba({'P':0,'C':0})[5].probability(1)
    pr2 = bayes_model.predict_proba({'P':2,'Mr':0})[6].probability(2)
    pr3 = bayes_model.predict_proba({'P':2,'C':1,'Mr':0})[3].probability(2)
    pr4 = bayes_model.predict_proba({'P':0,'A':0,'S':2})[5].probability(0)
    pr5 = bayes_model.probability([0,0,1,2,0,0,2])
```

B. 手动实现构建贝叶斯网络和推理。

- 以Burglary问题为例。

- 使用类Node存储每一个变量结点，存储变量名字，与该变量相关的所有变量名字集合，变量各种取值的概率（字典）。

```
#存储变量的结点
class Node():
    #存储变量名字、与该变量相关的变量名字集合
    def __init__(self,name,variable_list):
        self.name = name
        self.variable_list = variable_list
    #变量各种取值的概率
    def set_proba(self,proba):
        self.proba = proba
```

- 构建贝叶斯网络，先创建结点，然后再对每个变量赋予各种取值的概率，初始化已知变量的取值和变量的总数目。

```
def build_node(self):
        #创建变量结点
        self.B = Node('B', ['B'])
        self.E = Node('E', ['E'])
        self.A = Node('A', ['A', 'B', 'E'])
        self.J = Node('J', ['J', 'A'])
        self.M = Node('M', ['M', 'A'])

        #生成各个变量结点的条件概率
        self.B.set_proba({'0': 0.999, '1': 0.001})
        self.E.set_proba({'0': 0.998, '1': 0.002})
        self.A.set_proba({'111': 0.95, '011': 0.05, '110': 0.94, '010': 0.06,
                '101':0.29, '001': 0.71, '100': 0.001, '000': 0.999})
        self.J.set_proba({'11': 0.9, '01': 0.1, '10': 0.05, '00': 0.95})
        self.M.set_proba({'11': 0.7, '01': 0.3, '10': 0.01, '00': 0.99})
        #已知变量的取值
        self.var_list = [['B',0],
                         ['E',0],
                         ['A',0],
                         ['J',0],
                         ['M',0]]
        #变量的数目
        self.variable_num = 5
```

- 将条件概率使用链式法则扩展，分别求分子和分母的联合概率。

```
    #计算条件概率
    def condition_probability(self,query,condition):
        #将条件概率利用链式法则扩展
        #首先计算分母的全概率

        t2 = self.joint_probability(condition)
```

```
        #下面更新分子的变量集合以及各个变量的特定取值
        new_query = {}
        for q in query:
            new_query[q] = query[q]
        for c in condition:
            new_query[c] = condition[c]
        t1 = self.joint_probability(new_query)
        #将分子与分母相除得出条件概率
        return t1 / t2
```

- 首先给每个变量赋予可能的取值集合，其中已知变量取给定的值，求出在已知条件下所有情况的全概率相加得出联合概率。

```
    #计算联合概率
    def joint_probability(self,query):
        domain = []
        #给每个变量赋予取值的范围，用二维列表表示
        for i in range(self.variable_num):
            #对于已知变量，取特定值
            if self.var_list[i][0] in query:
                tmp = [query[self.var_list[i][0]]]
                domain.append(tmp)
                continue
            #未知变量则有所有取值可能
            else:
                domain.append([0,1])
        p = 0
        #循环遍历，求出在指定变量值的所有情况全概率相加
        for i in domain[0]:
            for j in domain[1]:
                for k in domain[2]:
                    for x in domain[3]:
                        for y in domain[4]:
                            p += self.probability([i,j,k,x,y])
        #p为所求的联合概率
        return p
```

- 初始化变量结点的顺序，给变量赋予特定值。

```
    #计算前进行变量集合初始化
    def probability(self,value_list):
        #声明变量结点顺序集合
        node_list = [self.B,self.E,self.A,self.J,self.M]
        #给已知变量赋予特定值
        for i in range(len(self.var_list)):
            self.var_list[i][1] = value_list[i]
        #计算全概率
        res = self.calculate_pro(node_list,self.var_list)
        return res
```

- 根据贝叶斯公式，全概率等价于链式法则中所有变量在特定取值下的概率相乘，其中链式法则可化简为对应的条件概率，因此按照变量顺序，取出每个变量在特定值下的概率并且进行累乘。

```
    #计算全概率的具体实现
    def calculate_pro(self,node_list,variable_list):
        res = 1
        #利用链式法则，按照变量顺序依次相乘
        for i in range(len(node_list)):
            #对于每一个变量结点，检查包含哪些变量
            for j in range(0,i+1):
                #检查当前变量是否在变量结点的变量列表中
                if variable_list[j][0] in node_list[i].variable_list:
                    #取特定值的概率
                    node_list[i] = self.take_specific_value(node_list[i],variable_list[j]
[0],variable_list[j][1])
            #将特定值的概率进行累乘
            res = res * list(node_list[i].proba.values())[0]
        return res
```

- 对于特定的变量的特定值，在变量中找到对应取值的概率，并且返回新的结点，其取值集合已经更新。

```
    #对变量进行取特定值
    def take_specific_value(self,node,variable,value):
        #定位变量列表中取特定值变量的位置
        index = node.variable_list.index(variable)
        #新的取值集合
        new_proba = {}
        #遍历原变量取值的key值
        for key in node.proba:
            #如果符合取值，则放入新集合
            if key[index] == str(value):
                new_proba[key] = node.proba[key]
        #生成新的变量结点
        new_node = Node(node.name,node.variable_list)
        new_node.set_proba(new_proba)

        return new_node
```

- 将数据结构和操作封装后，求条件概率时，依次传入查询的变量和取值，已知的变量和取值。

```
p4 = bayes.condition_probability({'J':1,'M':0},{'B':0})
print('P(J,~M | ~B): ',p4)
```

求联合概率时，直接传入各个变量和其对应的取值即可。

```
p2 = bayes.joint_probability({'B':1,'E':1,'A':1,'J':1,'M':1})
print('P(B,E,A,J,M): ',p2)
```

（5）创新点&优化

- 手动构建贝叶斯网络并实现推理求条件概率、联合概率、边缘概率。
- 根据D分离判断变量的条件独立性，从而省略求部分变量的联合概率的步骤。
  - 在求解task3中的第一问的条件概率$P(M|P,C)=\frac{P(M,P,C)}{P(P,C)}$时，我们分别需要求出分子和分母的联合概率，此时对于分母$P(P,C)$，根据贝叶斯结构图和D分离的知识，我们可以知道变量P和C是独立的。因此使用链式法则得到$P(P,C)=P(P)*P(C|P)$，而由独立性可知$P(C|P)=P(C)$，因此可得P(P,C) = P(P)*P(C)。
- 利用链式法则可以将联合概率转化为条件概率，然后再使用predict_proba函数求解其条件概率，比如求task2中的第一问$P(M,J)$，根据链式法则$P(M,J)=P(M)*P(M|P)$，再使用predict_proba函数求解即可，而不需要进行扩展，依次求出5个变量的全概率再相加，减少运算必要性。

（5）实验结果

实验结果展示

- 经过比对验证，结果正确。
- Three_gate:

```
P(A,C,B):  0.11111111111111109
P(A,C,A):  0.0
```

- Burglary

```
P(J,M):  0.0020841002390000014
P(B,E,A,J,M):  1.1969999999999995e-06
P(A | J,~M):  0.7606920388631567
P(J,~M | ~B):  0.049847948999999996
```

- Diagnosing

```
Pr1:  0.5948499999999999
Pr2:  0.42100000000000004
Pr3:  0.10000000000000042
Pr4:  0.1000000000000002
Pr5:  5.250000000000009e-06
```

拓展思考

- K2算法：
  - 对于评分函数，在结构学习中，我们是通过求给定数据D下，具有最大后验概率的网络结构S得出。

    $P(S|D) = \frac{P(S,D)}{P(D)}$

    由于分母都一样，因此不需要求分母，主要是求分子的概率公式，而分子可化为$P(S) * P(D|S)$，我们对于每种结构设定一个概率P(S)，由于假设每种概率服从均匀分布，因此P(S)为常数，所以只需要计算P(D|S)，通过对变量的取值展开的推导过程得出最终我们所见的评分函数。
  - 在搜索策略中，通过遍历每一个变量，比较评分函数高低，求出对应的父变量集，使用贪心搜索算法得到最大值。给定变量的顺序，按照这个顺序，变量之间的因果关系是可以确定的，即比如a在b之前，a只能为b的父结点，只存在从a到b的边，不能反过来，u为每个变量最多父变量个数，遍历每个变量，选择评分函数最高的父变量加入集合，直到评分函数达到最高停止算法。
- K2算法改进的思路

1. 对于评分函数，由于其计算是使用累乘方法计算，因此当数目比较大或比较小时，容易出现上溢或下溢，因此参考朴素贝叶斯的优化方法，可以对评分函数取对数为$log[g(i,\pi_i)]$,将乘法转化为加法，计算出每一项结果，直接将其相加，而不是相乘，提高训练效果。
2. 执行K2算法时，可以比较不同节点的初始化顺序，挑选最终得到网络结构最好的那个。
3. 在K2算法上改进，从相反思维入手，一开始将所有变量建立全连接的网络，一直使用贪心算法从结构中去除边，再结合K2的结果进行比较，从而得出更优的网络结构。
4. K2算法虽然性能比较好，但是有可能解空间非常复杂，为了缓解这个问题，可以结合爬山算法，采用启发式的思想，基于深度优先，在当前节点上，将其与周围邻居的节点进行比较，如果当前节点的评分最高，则直接返回，反之就用最高的邻居节点代替当前节点，不断循环从而达到最优状态，可以避免遍历所有，提高选择的效率；因为爬山算法不是全面搜索，因此结果具有局部性，因此可以融入回溯原理，不断搜索最优解，解决学习中存在的会收敛于局部最优的问题。