ASSIGNMENT - 2

Report

on

Bayesian inference on the given data

by

Shruti Sureshan (M21CS015)
Ajit Kumar (M21CS017)
Tejaswee A (M21CS064)

Contents

1 Scenario: Bayesian Inference: 3

2 Approach Devised: 3

2.1 Inputs …………………………………………………………………………… 3
2.2 Finding Node Wise Information ………………………………………………… 4
2.3 Find All possible combinations ………………………………………………… 4
2.4 Compute the parent list ………………………………………………………… 4
2.5 Computing the count for each combination in the sample ………………….. 5
2.6 Compute Conditional Probability ……………………………………………… 5
2.7 Compute Joint Probability ……………………………………………………… 6
2.8 Calculate Expectation-Maximization …………………………………………7

3 Output: 8

4 Result: 9

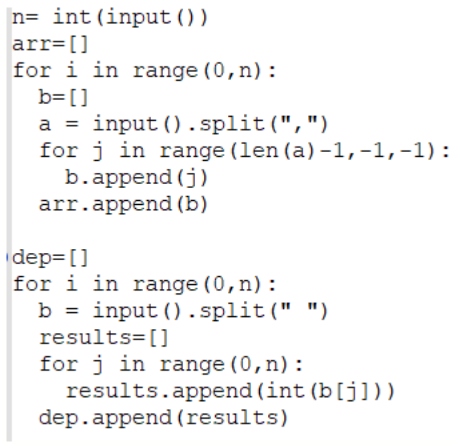
1 Problem: Bayesian Inference on the given data

Bayesian network is a directed acyclic graphical representation of a set of variables and their conditional dependencies. Each variable is represented as a node in the graph and a directed edge between the nodes represents the parent-child relationship between the considered nodes. In this assignment we will estimate probability distributions or parameters of a given network. For this, we will make use of a dataset containing samples of values observed for different variables.

2 Approach Devised:

2.1 Inputs:

Code snippet for taking inputs:



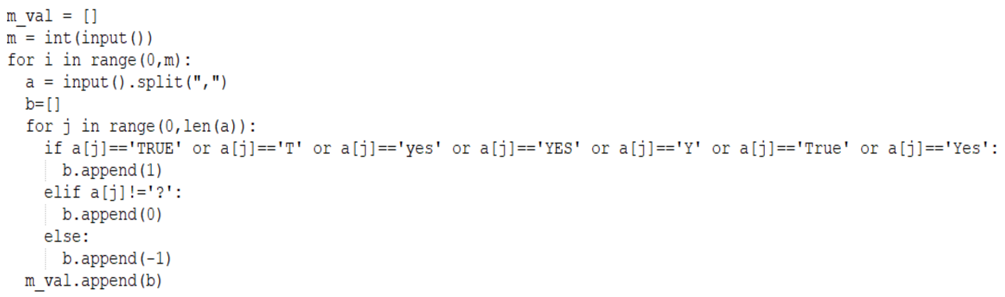
```
n= int(input())
arr=[]
for i in range(0,n):
  b=[]
  a = input().split(",")
  for j in range(len(a)-1,-1,-1):
    b.append(j)
  arr.append(b)

dep=[]
for i in range(0,n):
  b = input().split(" ")
  results=[]
  for j in range(0,n):
    results.append(int(b[j]))
  dep.append(results)
```

Explanation:

The code snippet above takes the inputs such as the no of nodes, a comma separated list of all the possible values of the nodes and a matrix of 0’s and 1’s representing conditional dependencies.

1. n: no. of variable or nodes (N_1, N_2, …., N_n)
2. a: Possible values for the nodes
3. arr: List of the possible values of the nodes in binary
4. dep: Dependency matrix

Code snippet for taking inputs for the samples:



```
m_val = []
m = int(input())
for i in range(0,m):
  a = input().split(",")
  b=[]
  for j in range(0,len(a)):
    if a[j]=='TRUE' or a[j]=='T' or a[j]=='yes' or a[j]=='YES' or a[j]=='Y' or a[j]=='True' or a[j]=='Yes':
      b.append(1)
    elif a[j]!='?':
      b.append(0)
    else:
      b.append(-1)
  m_val.append(b)
```

Explanation:

The code snippet above takes the inputs such as no of samples and a comma separated list of the observed values for each sample.

1. m: no of samples
2. a: Comma separated list of the sample values
3. b: Sample values represented as 0 if FALSE, 1 if TRUE and -1 otherwise
4. m_val: The list of sample values in 0's, -1's and 1's

2.2 Find Node Wise information:

Code snippet for finding node wise information:

```
list_nodes = []
for i in range(0,n):
  arr =[]
  for j in range(0,m):
    arr.append(m_val[j][i])
  list_nodes.append(arr)
```

Explanation:

The code snippet mentioned above finds the node wise information by representing the matrix of sample values as a column wise list. The m_val is appended in the list list_nodes with the iteration of i and j loop.

2.3 Find All possible combinations:

```
from itertools import product
comb_values=[]
comb_values = list(set(product([0,1],repeat=n)))
```

Explanation:

The code snippet mentioned above finds all the possible combinations of 0's and 1's for that particular no of input nodes. The list comb_values will give all possible combinations using itertools.product().

2.4 Compute the parent list:

Code snippet for computing the parent list:

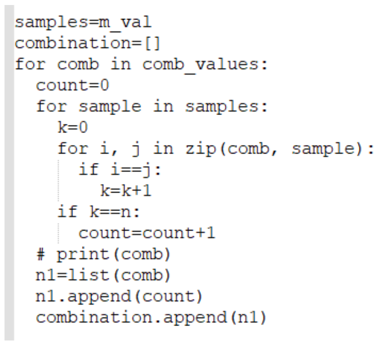
```
parent_list=[]
for i in range(0,n):
  dep_node=[]
  for j in range(0,n):
    if dep[j][i]==1:
      dep_node.append(j)
  parent_list.append(dep_node)
```

Explanation:

The code snippet mentioned above computes the parent list by considering the dependencies. dep[j][i] is a 2D dependency matrix. When the value in the dependency matrix dep[j][i] is 1, we will append that node in the list dep_node. Finally, the list parent_list gives the list of parents for each node in the network.

2.5 Computing the count for each combination in the sample:

Code snippet for computing the count for each combination:

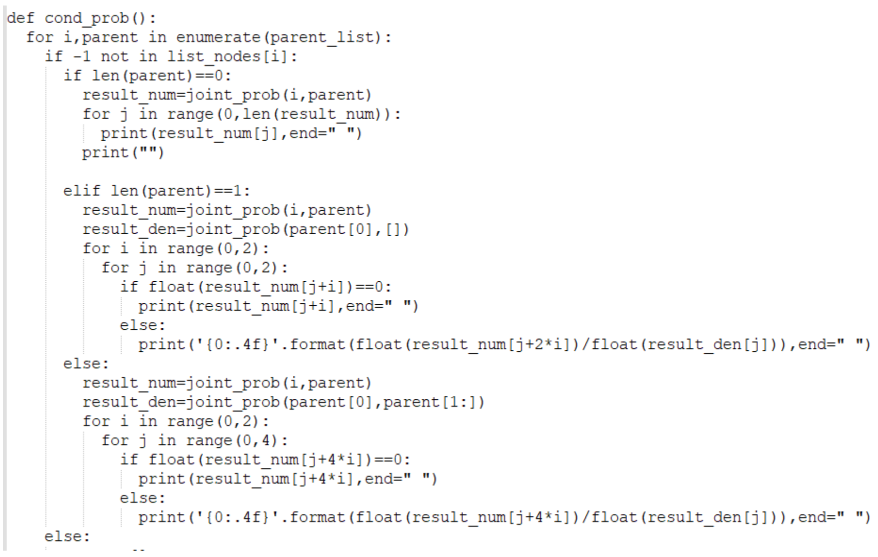


```
samples=m_val
combination=[]
for comb in comb_values:
  count=0
  for sample in samples:
    k=0
    for i, j in zip(comb, sample):
      if i==j:
        k=k+1
    if k==n:
      count=count+1
  # print(comb)
  n1=list(comb)
  n1.append(count)
  combination.append(n1)
```

Explanation:

The code snippet mentioned above computes the count for each combination. The variable count is initially set to 0 and is incremented with each iteration. It stores the total count value for each of the combinations using zip(). The final values are appended in the matrix named combination to keep a count of samples satisfying the corresponding combination of values.

2.6 Compute Conditional Probability:



```
def cond_prob():
  for i,parent in enumerate(parent_list):
    if -1 not in list_nodes[i]:
      if len(parent)==0:
        result_num=joint_prob(i,parent)
        for j in range(0,len(result_num)):
          print(result_num[j],end=" ")
        print("")

      elif len(parent)==1:
        result_num=joint_prob(i,parent)
        result_den=joint_prob(parent[0],[])
        for i in range(0,2):
          for j in range(0,2):
            if float(result_num[j+i])==0:
              print(result_num[j+i],end=" ")
            else:
              print('{0:.4f}'.format(float(result_num[j+2*i])/float(result_den[j])),end=" ")
      else:
        result_num=joint_prob(i,parent)
        result_den=joint_prob(parent[0],parent[1:])
        for i in range(0,2):
          for j in range(0,4):
            if float(result_num[j+4*i])==0:
              print(result_num[j+4*i],end=" ")
            else:
              print('{0:.4f}'.format(float(result_num[j+4*i])/float(result_den[j])),end=" ")
    else:
```

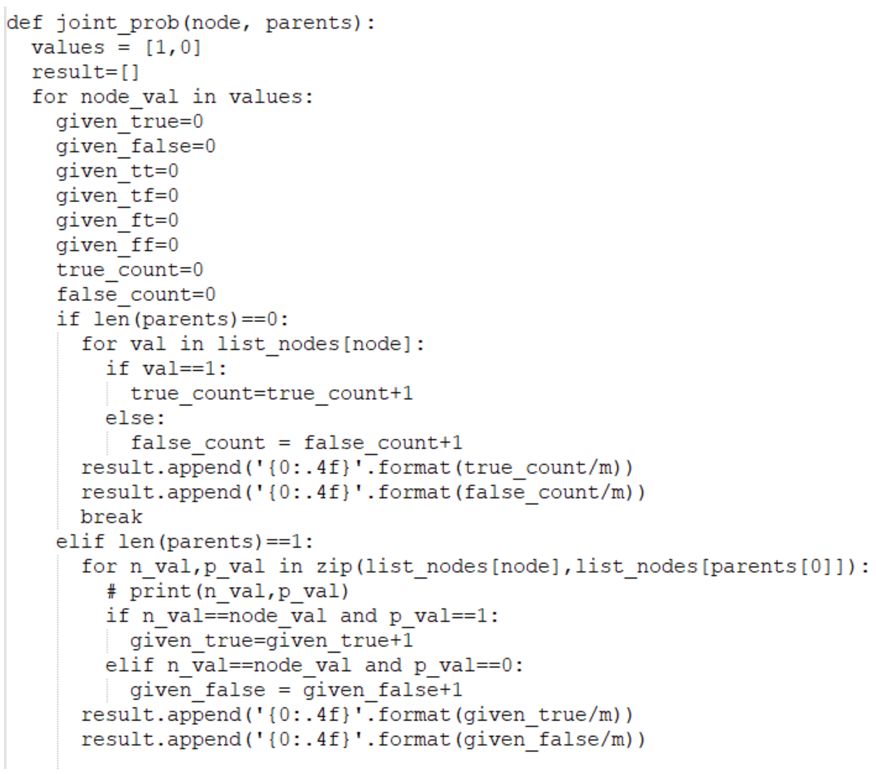
Explanation:

The code snippet shown above computes the conditional probability considering the possible combinations from the sample given as input.
It takes into consideration of two cases:

i.when there exists ‘?’ in the sample space
ii. when there is no existence of ‘?’ in a particular column of the sample data.

Returns corresponding conditional probability with a precision up to 4 decimal points.

2.7 Compute Joint probability:



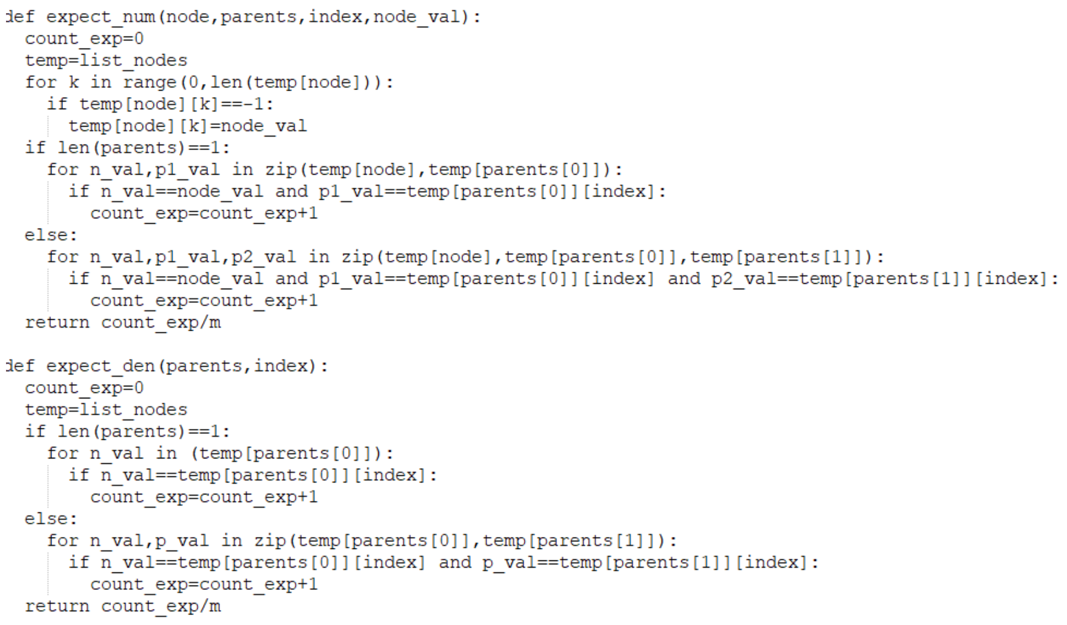
```
def joint_prob(node, parents):
  values = [1,0]
  result=[]
  for node_val in values:
    given_true=0
    given_false=0
    given_tt=0
    given_tf=0
    given_ft=0
    given_ff=0
    true_count=0
    false_count=0
    if len(parents)==0:
      for val in list_nodes[node]:
        if val==1:
          true_count=true_count+1
        else:
          false_count = false_count+1
      result.append('{0:.4f}'.format(true_count/m))
      result.append('{0:.4f}'.format(false_count/m))
      break
    elif len(parents)==1:
      for n_val,p_val in zip(list_nodes[node],list_nodes[parents[0]]):
        # print(n_val,p_val)
        if n_val==node_val and p_val==1:
          given_true=given_true+1
        elif n_val==node_val and p_val==0:
          given_false = given_false+1
      result.append('{0:.4f}'.format(given_true/m))
      result.append('{0:.4f}'.format(given_false/m))
```

Explanation:

Joint_prob() function calculates the joint probabilities of nodes considering the likelihood of more than one event occurring together at a given instance of time. It also takes into the account of calculating probabilities of independent and dependent nodes.

Returns the probabilities with a precision up to 4 decimal places.

2.8: Calculating Expectation-Maximization:



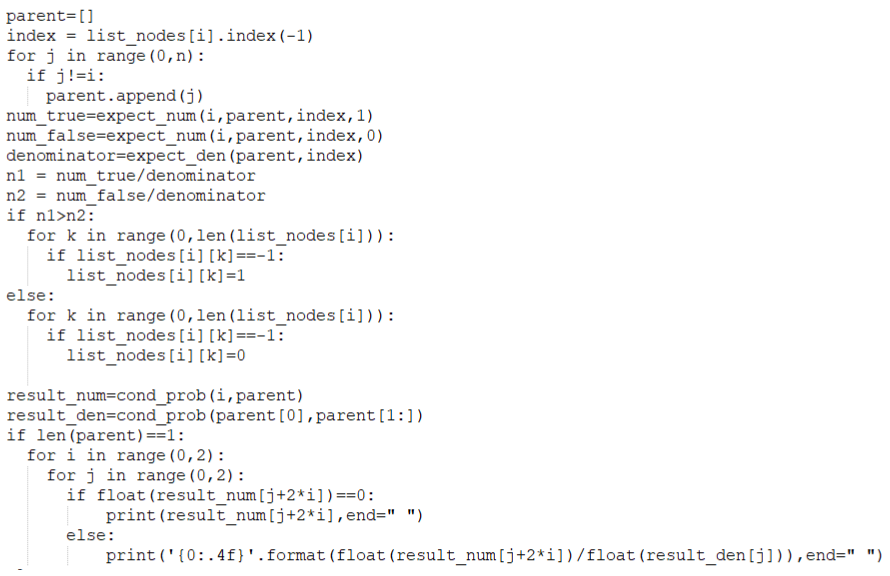
```
def expect_num(node,parents,index,node_val):
  count_exp=0
  temp=list_nodes
  for k in range(0,len(temp[node])):
    if temp[node][k]==-1:
      temp[node][k]=node_val
  if len(parents)==1:
    for n_val,p1_val in zip(temp[node],temp[parents[0]]):
      if n_val==node_val and p1_val==temp[parents[0]][index]:
        count_exp=count_exp+1
  else:
    for n_val,p1_val,p2_val in zip(temp[node],temp[parents[0]],temp[parents[1]]):
      if n_val==node_val and p1_val==temp[parents[0]][index] and p2_val==temp[parents[1]][index]:
        count_exp=count_exp+1
  return count_exp/m

def expect_den(parents,index):
  count_exp=0
  temp=list_nodes
  if len(parents)==1:
    for n_val in (temp[parents[0]]):
      if n_val==temp[parents[0]][index]:
        count_exp=count_exp+1
  else:
    for n_val,p_val in zip(temp[parents[0]],temp[parents[1]]):
      if n_val==temp[parents[0]][index] and p_val==temp[parents[1]][index]:
        count_exp=count_exp+1
  return count_exp/m
```

Expectation Step (E-Step):

Above mentioned code snippet estimates the missing values in the given sample by calculating the joint probabilities. Here, the missing value is represented by '?' which was converted to '-1' for simplicity.

Hence, we initially assume the missing value to be 1 and calculate the probabilities accordingly. We then obtain a matrix containing no missing values.

Maximization Step (M-Step):



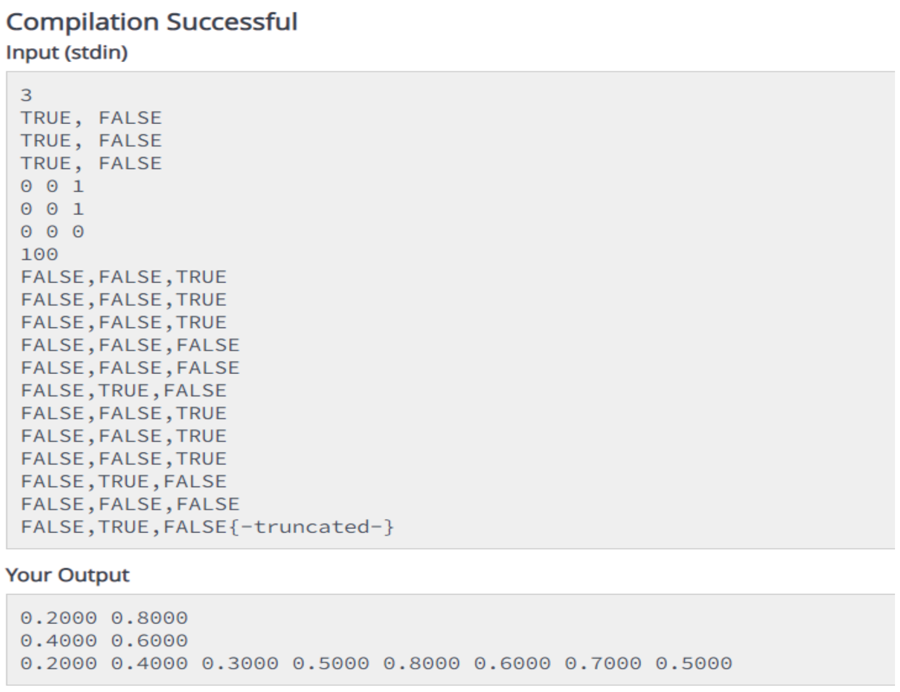
```
parent=[]
index = list_nodes[i].index(-1)
for j in range(0,n):
  if j!=i:
    parent.append(j)
num_true=expect_num(i,parent,index,1)
num_false=expect_num(i,parent,index,0)
denominator=expect_den(parent,index)
n1 = num_true/denominator
n2 = num_false/denominator
if n1>n2:
  for k in range(0,len(list_nodes[i])):
    if list_nodes[i][k]==-1:
      list_nodes[i][k]=1
else:
  for k in range(0,len(list_nodes[i])):
    if list_nodes[i][k]==-1:
      list_nodes[i][k]=0

result_num=cond_prob(i,parent)
result_den=cond_prob(parent[0],parent[1:])
if len(parent)==1:
  for i in range(0,2):
    for j in range(0,2):
      if float(result_num[j+2*i])==0:
          print(result_num[j+2*i],end=" ")
      else:
          print('{0:.4f}'.format(float(result_num[j+2*i])/float(result_den[j])),end=" ")
```

The afore-mentioned step is used to update the parameters according to the maximum likelihood obtained in the expectation step (E-step).

Both the steps are repeated until we reach a convergence point post which there will not a significant difference in the probabilities obtained. Hence, will conclude by considering the value associated with the corresponding probability.

3 Output:

1. Output to the sample test case provided with 100 samples and 3 nodes with Boolean values (TRUE/FALSE).



Compilation Successful

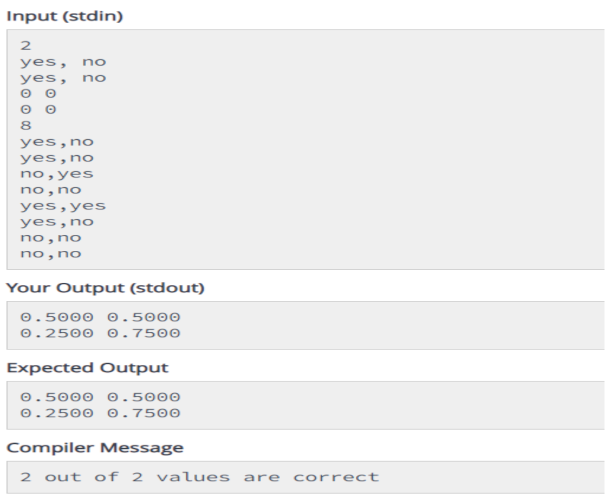
Input (stdin)

```
3
TRUE, FALSE
TRUE, FALSE
TRUE, FALSE
0 0 1
0 0 1
0 0 0
100
FALSE,FALSE,TRUE
FALSE,FALSE,TRUE
FALSE,FALSE,TRUE
FALSE,FALSE,FALSE
FALSE,FALSE,FALSE
FALSE,TRUE,FALSE
FALSE,FALSE,TRUE
FALSE,FALSE,TRUE
FALSE,FALSE,TRUE
FALSE,TRUE,FALSE
FALSE,FALSE,FALSE
FALSE,TRUE,FALSE{-truncated-}
```

Your Output

```
0.2000 0.8000
0.4000 0.6000
0.2000 0.4000 0.3000 0.5000 0.8000 0.6000 0.7000 0.5000
```

2. Output to the sample test case provided with 8 samples and 2 nodes with Boolean values (yes/no).



Input (stdin)

```
2
yes, no
yes, no
0 0
0 0
8
yes,no
yes,no
no,yes
no,no
yes,yes
yes,no
no,no
no,no
```

Your Output (stdout)

```
0.5000 0.5000
0.2500 0.7500
```

Expected Output

```
0.5000 0.5000
0.2500 0.7500
```

Compiler Message

```
2 out of 2 values are correct
```

4 Results:

The probability distribution of all variables taking into consideration of dependency on other Nodes, is printed as the output.