Shi Xingyue

Dr. Rao Yanghui

Artificial Intelligence

14 December 2018

Lab 11: Variable Elimination – Burglary Problem

[Abstract]

In this lab, I implemented 4 functioned required for variable elimination to calculate conditional probability in burglary problem, namely "inference", "multiply", "sumout" and "restrict". The last 3 functions are called in the first one in sequence.

1 Methods

1.1 Algorithm

1.1.1 Function "inference"

This function calls the following three functions, namely "restrict", "multiply" and "sumout" one by one. It finishes the algorithm of variable elimination by doing these steps:

- restrict all the evidence variables to their given values;
- for each hidden variable in ordered list of hidden variables, eliminate it by replacing the factors including it with their summed-out product;
- multiply remaining factors referring only to the query variables and then normalize their product.

1.1.2 Function "restrict"

Once we know the value of a variable, this variable is "restricted" to this value. Thus this variable should be removed from variable list and conditional probability table's keys.

I take advantage of the slicing operation of string in python when writing this function.

1.1.3 Function “multiply”

This function requires the variable list and cpt to be updated. The new variable list is the union of the two factors’ variable lists. The keys of the new cpt are all possible values of the new variable lists, while the values are point-wise product of the two factor’s cpt values.

1.1.4 Function “sumout”

A variable from a factor can be summed out to produce a new factor. For example, variable X from factor f is summed out to produce new factor h = $\sum_X f$,which is defined as h(Y) = $\sum_X f(X, Y)$.

1.2 Pseudo-code

Below is the pseudo-code of function inference. The other 3 functions are called within it. Since the other 3 functions are quite simple and readable, you maybe would like to see full codes in the next section.

```
inference(factorList, queryVariables, orderedListOfHiddenVariables, evidenceList):

        for ev in evidenceList:
                restrict this variable in each factor in factorList

        for var in orderedListOfHiddenVariables:
                multiply all factors including var
                sum out var
                remove all the factors including var in factorList
                append summed-out product to factorList

        multiply all remaining factors
        normalization
```

Code 1 Psuedo-code of inference

1.3 Key Codes

1.3.1 Function “inference”

```
def inference(factorList, queryVariables,
      orderedListOfHiddenVariables, evidenceList):
            for ev in evidenceList:
                  to_append=[]
                  to_remove=[]
                  for factor in factorList:
                        #factor.printInf()
                        if ev in factor.varList:
                              #if this factor has only 1 variable
                              #it is removed after restriction
                              if len(factor.varList)==1:
                                    factorList.remove(factor)
                                    continue
                              factor_new=factor.restrict(ev,evidenceList[ev])
                              to_append.append(factor_new)
                              to_remove.append(factor)
                              """
                              appending and removing within loop causes chaos!
                              factorList.append(factor_new)
                              factorList.remove(factor)
                              """
                  for append in to_append:
                        factorList.append(append)
                  for remove in to_remove:
                        factorList.remove(remove)

            for var in orderedListOfHiddenVariables:
                  flag=0
                  g=Node(var,[var])
                  remove_list=[]
                  for factor in factorList:
                        if var in factor.varList:
                              if flag==0:
                                    g=factor
                                    flag=1
                                    remove_list.append(factor)
                              else:
                                    g=g.multiply(factor)
                                    remove_list.append(factor)

                  for remove_factor in remove_list:
                        factorList.remove(remove_factor)
                  factorList.append(g.sumout(var))
            res = factorList[0]
            for factor in factorList[1:]:
                  res = res.multiply(factor)
            total = sum(res.cpt.values())
            res.cpt = {k: v/total for k, v in res.cpt.items()}
            return res
```

Code 2 inference

1.3.2 Function “restrict”

```
def restrict(self, variable, value):
            """function that restricts a variable to some value
            in a given factor"""
            new_var_list=[]
            new_cpt={}


            if variable in self.varList and len(self.varList)==1:
                  empty_node=Node(variable,[variable])
                  empty_node.setCpt({Util.to_binary(int(value),1):1,Util.to_binary(int(1-value),1):0})
                  return empty_node


            index =self.varList.index(variable)


            new_var_list=self.varList[:index]+self.varList[index+1:]


            for key in self.cpt:
                  new_cpt[key[:index]+key[index+1:]]=self.cpt[key[:index]+str(value)+key[index+1:]]


            new_node=Node("f" + str(new_var_list), new_var_list)
            new_node.setCpt(new_cpt)
            return new_node
```

Code 3 restrict

1.3.3 Function “multiply”

```
def multiply (self,factor):
            """
            multiply with another factor
            CODE HERE
            """
            #the new varList the union of 2 old varLists
            new_var_list=[]
            new_cpt={}
            cpt_ref={}
            common=[]
            common_self_index=[]
            common_factor_index=[]
            distinct_self=[]
            distinct_factor=[]


            common=[variable for variable in factor.varList if variable in self.varList]
            common_self_index=[self.varList.index(variable) for variable in self.varList if variable in factor.varList]
            common_factor_index=[factor.varList.index(variable) for variable in factor.varList if variable in self.varList]
            distinct_self=[variable for variable in self.varList if variable not in factor.varList]
            distinct_factor=[variable for variable in factor.varList if variable not in self.varList]


            for key in self.cpt:
                  key_common=''.join([key[i] for i in range(len(self.varList)) if i in common_self_index])
                  key_self=''.join([key[i] for i in range(len(self.varList)) if i not in common_self_index])
                  if key_common not in cpt_ref:
                        cpt_ref[key_common]={}
                  cpt_ref[key_common][key_self]=self.cpt[key]


            for key in factor.cpt:
                  key_common=''.join([key[i] for i in range(len(factor.varList)) if i in common_factor_index])
                  key_factor=''.join([key[i] for i in range(len(factor.varList)) if i not in common_factor_index])
                  key_temp=cpt_ref[key_common]
                  for key2 in key_temp:
                        key_combined=key_common+key2+key_factor
                        #print("combined")
                        #print (key_combined)
                        new_cpt[key_combined]=key_temp[key2]*(factor.cpt[key])


            #print (new_cpt)


            new_var_list=common+distinct_self+distinct_factor


            new_node=Node("f"+str(new_var_list),new_var_list)
            new_node.setCpt(new_cpt)
            return new_node
```

Code 4 multiply

1.3.4 Function “sumout”

```
def sumout(self, variable):
            """function that sums out a variable given a factor"""
            new_var_list=[]
            new_cpt={}
            for var in self.varList:
                  if var!=variable:
                        new_var_list.append(var)
            pos=self.varList.index(variable)
            n=len(self.varList)
            for i in range(2**(n-1)):
                  p1=i+int(i/(2**(n-pos-1)))
                  p2=i+int(i/(2**(n-pos-1)))+2**(n-pos-1)
                  t1=Util.to_binary(int(p1),n)
                  t2=Util.to_binary(int(p2),n)
                  new_cpt[Util.to_binary(int(i),n-1)]=self.cpt[t1]+self.cpt[t2]
            new_node = Node("f" + str(new_var_list), new_var_list)
            new_node.setCpt(new_cpt)
            return new_node
```

Code 5 sumout

2 Results and Analysis

```
C:\Users\mibyb\Documents\课程\大三上\人工智能\lab9_变量消除>python lab11.py
P(Alarm)=
0.0025164420000000002

P(J&&~M)=
0.05005487546100001

P(A|J&&~M)=
0.013573889331307633

P(B|A)=
0.373551228281836

P(B|J~M)=
0.0051298581334013015

P(J&&~M|~B)=
0.049847949
```

Figure 1 My running result. The probabilities are identical to the ones on slides.