School of Data and Computer Science, Sun Yat-sen University

E09 Variable Elimination

16337188 Ouyang Runyu

November 2, 2018

Contents

1 VE **3**

2 Task **6**

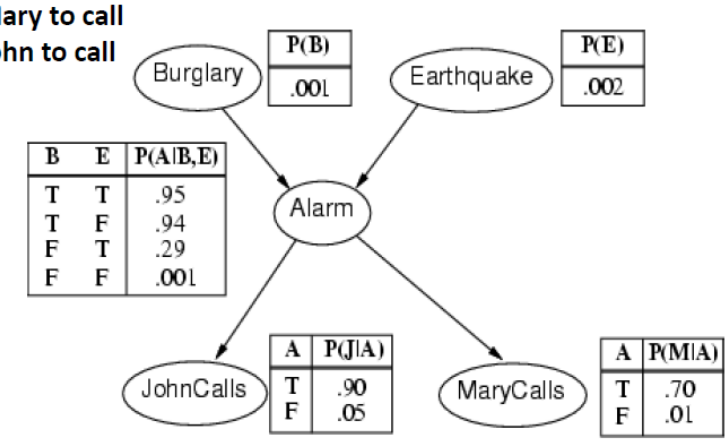
3 Codes and Results **7**

1 VE

The burglary example is described as following:

- **A burglary can set the alarm off**
- **An earthquake can set the alarm off**
- **The alarm can cause Mary to call**
- **The alarm can cause John to call**

Note that these tables only provide the probability that Xi is true.
(E.g., Pr(A is true|B,E))
The probability that Xi is false is 1- these values




```
P(Alarm) =
0.002516442

P(J&&~M) =
0.050054875461

P(A |J&&~M) =
0.0135738893313

P(B |A) =
0.373551228282

P(B |J&&~M) =
0.0051298581334

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

Here is a VE template for you to solve the burglary example:

```
class VariableElimination:
    @staticmethod
    def inference(factorList, queryVariables,
    orderedListOfHiddenVariables, evidenceList):
        for ev in evidenceList:
            #Your code here
        for var in orderedListOfHiddenVariables:
            #Your code here
        print "RESULT:"
```

```
        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()}
        res.printInf()
    @staticmethod
    def printFactors(factorList):
        for factor in factorList:
            factor.printInf()
class Util:
    @staticmethod
    def to_binary(num, len):
        return format(num, '0' + str(len) + 'b')
class Node:
    def __init__(self, name, var_list):
        self.name = name
        self.varList = var_list
        self.cpt = {}
    def setCpt(self, cpt):
        self.cpt = cpt
    def printInf(self):
        print "Name = " + self.name
        print " vars " + str(self.varList)
        for key in self.cpt:
            print "    key: " + key + " val : " + str(self.cpt[key])
        print ""
    def multiply(self, factor):
        """function that multiplies with another factor"""
        #Your code here
        new_node = Node("f" + str(newList), newList)
        new_node.setCpt(new_cpt)
        return new_node
```

```
    def sumout(self, variable):
        """function that sums out a variable given a factor"""
        #Your code here
        new_node = Node("f" + str(new_var_list), new_var_list)
        new_node.setCpt(new_cpt)
        return new_node
    def restrict(self, variable, value):
        """function that restricts a variable to some value
        in a given factor"""
        #Your code here
        new_node = Node("f" + str(new_var_list), new_var_list)
        new_node.setCpt(new_cpt)
        return new_node
# create nodes for Bayes Net
B = Node("B", ["B"])
E = Node("E", ["E"])
A = Node("A", ["A", "B","E"])
J = Node("J", ["J", "A"])
M = Node("M", ["M", "A"])

# Generate cpt for each node
B.setCpt({'0': 0.999, '1': 0.001})
E.setCpt({'0': 0.998, '1': 0.002})
A.setCpt({'111': 0.95, '011': 0.05, '110':0.94,'010':0.06,
'101':0.29,'001':0.71,'100':0.001,'000':0.999})
J.setCpt({'11': 0.9, '01': 0.1, '10': 0.05, '00': 0.95})
M.setCpt({'11': 0.7, '01': 0.3, '10': 0.01, '00': 0.99})

print "P(A) **********"
VariableElimination.inference([B,E,A,J,M], ['A'], ['B', 'E', 'J','M'], {})

print "P(B | J~M) **********"
VariableElimination.inference([B,E,A,J,M], ['B'], ['E','A'], {'J':1,'M':0})
```

2 Task

- You should implement 4 functions: `inference`, `multiply`, `sumout` and `restrict`. You can turn to Figure 1 and Figure 2 for help.
- Please hand in a file named E09_YourNumber.pdf, and send it to ai_2018@foxmail.com

The VE Algorithm

Given a Bayes Net with CPTs F, query variable Q, evidence variables **E** (observed to have values e), and remaining variables **Z**. Compute Pr(Q|**E**)

1. Replace each factor $f \in F$ that mentions a variable(s) in **E** with its restriction $f_{\mathbf{E}=e}$ (this might yield a "constant" factor)
2. For each Z_j– in the order given –eliminate $Z_j \in \mathbf{Z}$ as follows:
   1. Let $f_1, f_2, \ldots, f_k$ be the factors in F that include Z_j
   2. Compute new factor $g_j = \sum_{Z_j} f_1 \times f_2 \times \ldots \times f_k$
   3. Remove the factors f_i from F and add new factor g_j to F
3. The remaining factors refer only to the query variable Q. Take their product and normalize to produce Pr(Q|**E**).

The Product of Two Factors

- Let f(**X**,**Y**) & g(**Y**,**Z**) be two factors with variables **Y** in common
- The ***product*** of f and g, denoted h = f × g (or sometimes just h = fg), is defined:

h(**X**,**Y**,**Z**) = f(**X**,**Y**) × g(**Y**,**Z**)

f(A,B)		g(B,C)		h(A,B,C)			
ab	0.9	bc	0.7	abc	0.63	ab~c	0.27
a~b	0.1	b~c	0.3	a~bc	0.08	a~b~c	0.02
~ab	0.4	~bc	0.8	~abc	0.28	~ab~c	0.12
~a~b	0.6	~b~c	0.2	~a~bc	0.48	~a~b~c	0.12

Figure 1: VE and Product

Summing a Variable Out of a Factor

- Let f(X,**Y**) be a factor with variable X (**Y** is a set)
- We ***sum out*** variable X from f to produce a new factor h = Σ_X f, which is defined:

h(**Y**) = $\Sigma_{x \in Dom(X)}$ f(x,**Y**)

f(A,B)		h(B)	
ab	0.9	b	1.3
a~b	0.1	~b	0.7
~ab	0.4		
~a~b	0.6		

No error in the table. Here $f(A, B)$ is not $P(AB)$, but $P(B|A)$.

Restricting a Factor

- Let f(X,**Y**) be a factor with variable X (**Y** is a set)
- We ***restrict*** factor f *to* X=*a* by setting X to the value *a* and "deleting" incompatible elements of f's domain . Define h = $f_{X=a}$ as: h(**Y**) = f(a,**Y**)

f(A,B)		h(B) = $f_{A=a}$	
ab	0.9	b	0.9
a~b	0.1	~b	0.1
~ab	0.4		
~a~b	0.6		

Figure 2: Sumout and Restrict

3 Codes and Results

The codes are in VE.py

VE.py

```
class VariableElimination:
    @staticmethod
    def inference(factorList, queryVariables,
    orderedListOfHiddenVariables, evidenceList):
        for ev, value in evidenceList.items():
            #Your code here
            for i, factor in enumerate(factorList):
                if ev in factor.varList:
                    new_node = factor.restrict(ev, value)
                    factorList.pop(i)
                    factorList.insert(i, new_node)

        for var in orderedListOfHiddenVariables:
            var_factors = [factor for factor in factorList if var in
               factor.varList]

            if len(var_factors) > 0:
                var_res = var_factors[0]
                temp = var_res
                for factor in var_factors[1:]:
                    var_res = var_res.multiply(factor)
                    factorList.remove(factor)

            var_res = var_res.sumout(var)
            factorList.remove(temp)
            if len(var_res.varList) > 0:
                factorList.append(var_res)

        print("RESULT:")
```

```
        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()}
        res.printInf()
    @staticmethod
    def printFactors(factorList):
        for factor in factorList:
            factor.printInf()

class Util:
    @staticmethod
    def to_binary(num, len):
        return format(num, '0' + str(len) + 'b')

class Node:
    def __init__(self, name, var_list):
        self.name = name
        self.varList = var_list
        self.cpt = {}
    def setCpt(self, cpt):
        self.cpt = cpt
    def printInf(self):
        print("Name = " + self.name)
        print("  vars " + str(self.varList))
        for key in self.cpt:
            print("    key: " + key + " val : " + str(self.cpt[key]))
        print()

    def multiply(self, factor):
        """function that multiplies with another factor"""
        #Your code here
```

```
        new_cpt = {}
        same_variables = [var for var in self.varList if var in factor.
            varList]
        if len(same_variables) > 0:
            variable = same_variables[0]
            i = self.varList.index(variable)
            j = factor.varList.index(variable)

            for x1, y1 in self.cpt.items():
                for x2, y2 in factor.cpt.items():
                    if x1[i] != x2[j]:
                        continue
                    new_cpt[x1[:i] + x2[:j] + x1[i:] + x2[j + 1:]] = y1
                        * y2
            newList = self.varList[:i] + factor.varList[:j] + self.
                varList[i:] + factor.varList[j + 1:]
        else:
            for x1, y1 in self.cpt.items():
                for x2, y2 in factor.cpt.items():
                    new_cpt[x1 + x2] = y1 * y2
            newList = self.varList + factor.varList
        new_node = Node("f" + str(newList), newList)
        new_node.setCpt(new_cpt)
        return new_node

    def sumout(self, variable):
        """function that sums out a variable given a factor"""
        #Your code here
        i = self.varList.index(variable)
        new_var_list = self.varList[:i] + self.varList[i + 1:]

        new_cpt = {}
        for x in self.cpt:
```

```
            now_key = x[:i] + x[i + 1:]
            if now_key not in new_cpt:
                new_cpt[now_key] = self.cpt[x[:i] + '0' + x[i + 1:]] +
                    self.cpt[x[:i] + '1' + x[i + 1:]]

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

    def restrict(self, variable, value):
        """function that restricts a variable to some value
        in a given factor"""
        #Your code here
        i = self.varList.index(variable)
        new_cpt = dict([(x[:i] + x[i + 1:], y) for x, y in self.cpt.
            items() if int(x[i]) == value])
        new_var_list = self.varList[:i] + self.varList[i + 1:]

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

# create nodes for Bayes Net
B = Node("B", ["B"])
E = Node("E", ["E"])
A = Node("A", ["A", "B","E"])
J = Node("J", ["J", "A"])
M = Node("M", ["M", "A"])

# Generate cpt for each node
B.setCpt({'0': 0.999, '1': 0.001})
E.setCpt({'0': 0.998, '1': 0.002})
A.setCpt({'111': 0.95, '011': 0.05, '110':0.94,'010':0.06,
```

```
'101':0.29,'001':0.71,'100':0.001,'000':0.999})
J.setCpt({'11': 0.9, '01': 0.1, '10': 0.05, '00': 0.95})
M.setCpt({'11': 0.7, '01': 0.3, '10': 0.01, '00': 0.99})

print("P(A)")
VariableElimination.inference([B,E,A,J,M], ['A'], ['B', 'E', 'J','M'],
    {})

print("P(J~M)")
VariableElimination.inference([B,E,A,J,M], ['J', '~M'], ['B', 'E', 'A'],
     {})

print("P(A | J~M)")
VariableElimination.inference([B,E,A,J,M], ['A'], ['E', 'B'], {'J':1, 'M
    ':0})

print("P(B | A)")
VariableElimination.inference([B,E,A,J,M], ['B'], ['E', 'J', 'M'], {'A'
    :1})

print("P(B | J~M)")
VariableElimination.inference([B,E,A,J,M], ['B'], ['E', 'A'], {'J':1, 'M
    ':0})

print("P(J~M | ~B)")
VariableElimination.inference([B,E,A,J,M], ['J', '~M'], ['E', 'A'], {'B'
    :0})
```

```
PS C:\Users\abc810343087\Desktop\学习\人工智能\E09> python VE.py
P(A)
RESULT:
Name = f['A']
 vars ['A']
   key: 1 val : 0.0025164420000000002
   key: 0 val : 0.997483558

P(J~M)
RESULT:
Name = f['J', 'M']
 vars ['J', 'M']
   key: 11 val : 0.002084100239
   key: 10 val : 0.050054875461
   key: 01 val : 0.009652244741
   key: 00 val : 0.938208779559

P(A | J~M)
RESULT:
Name = f['A']
 vars ['A']
   key: 1 val : 0.013573889331307631
   key: 0 val : 0.9864261106686925

P(B | A)
RESULT:
Name = f['B']
 vars ['B']
   key: 0 val : 0.626448771718164
   key: 1 val : 0.373551228281836

P(B | J~M)
RESULT:
Name = f['B']
 vars ['B']
   key: 0 val : 0.9948701418665987
   key: 1 val : 0.0051298581334013015

P(J~M | ~B)
RESULT:
Name = f['J', 'M']
 vars ['J', 'M']
   key: 11 val : 0.001493351
   key: 10 val : 0.049847948999999996
   key: 01 val : 0.009595469
   key: 00 val : 0.939063231
```

Figure 3: VE_result