

E09 Variable Elimination

20214966 Yangkai Lin 20214810 Suixin Ou

November 10, 2020

Contents

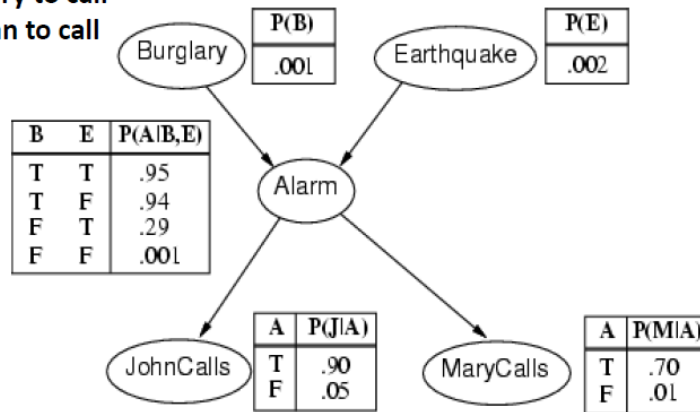
1	VE	2
2	Task	5
3	Codes and Results	5
3.1	Codes	5
3.2	Results	10

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 X_i is true.
(E.g., $\Pr(A \text{ is true} | B, E)$)
The probability that X_i 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_2020@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 $\Pr(Q|E)$

- Replace each factor $f \in F$ that mentions a variable(s) in E with its restriction $f_{E=e}$ (this might yield a "constant" factor)
- For each Z_j in the order given –eliminate $Z_j \in Z$ as follows:
 - Let f_1, f_2, \dots, f_k be the factors in F that include Z_j
 - Compute new factor $g_j = \sum_{Z_j} f_1 \times f_2 \times \dots \times f_k$
 - Remove the factors f_i from F and add new factor g_j to F
- 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 $f(\underline{X}, \underline{Y})$ & $g(\underline{Y}, \underline{Z})$ be two factors with variables \underline{Y} in common
- The **product** of f and g , denoted $h = f \times g$ (or sometimes just $h = fg$), is defined:

$$h(\underline{X}, \underline{Y}, \underline{Z}) = f(\underline{X}, \underline{Y}) \times g(\underline{Y}, \underline{Z})$$

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 $f(X, \underline{Y})$ be a factor with variable X (\underline{Y} is a set)
- We **sum out** variable X from f to produce a new factor $h = \sum_X f$, which is defined:

$$h(\underline{Y}) = \sum_{X \in \text{Dom}(X)} f(X, \underline{Y})$$

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 $f(X, \underline{Y})$ be a factor with variable X (\underline{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: $h(\underline{Y}) = f(a, \underline{Y})$

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

3.1 Codes

```
class VariableElimination:
    @staticmethod
```

```

def inference(factorList , queryVariables ,
orderedListOfHiddenVariables , evidenceList):
    for ev in evidenceList:
        #Your code here
        for i in range(len(factorList)):
            if ev in factorList[i].varList:
                factorList[i] = factorList[i].restrict(ev , evidenceList[ev])
    for var in orderedListOfHiddenVariables:
        #Your code here
        new_factor = None
        i = 0
        while True:
            while i < len(factorList):
                if var in factorList[i].varList:
                    if new_factor is None:
                        new_factor = factorList[i]
                    else:
                        new_factor = new_factor.multiply(factorList[i])
                    del factorList[i]
                    break
                i += 1
            else:
                break
        if new_factor is not None:
            factorList.append(new_factor.sumout(var))
    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
        same_var = set()
        same_var_pos = list()
        for i in range(len(self.varList)):
            for j in range(len(factor.varList)):
                if self.varList[i] == factor.varList[j]:
                    same_var.add(self.varList[i])
                    same_var_pos.append((i, j))
                break
        newList = self.varList[:]
        for var in factor.varList:

```

```

        if var not in same_var:
            newList.append(var)
new_node = Node("f" + str(newList), newList)
new_cpt = dict()
for k1, v1 in self.cpt.items():
    for k2, v2 in factor.cpt.items():
        for i, j in same_var_pos:
            if k1[i] != k2[j]:
                break
        else:
            new_k2 = list()
            for i in range(len(k2)):
                if factor.varList[i] not in same_var:
                    new_k2.append(k2[i])
            new_cpt[k1 + ''.join(new_k2)] = v1 * v2
new_node.setCpt(new_cpt)
return new_node
def sumout(self, variable):
    """function that sums out a variable given a factor"""
    #Your code here
    pos_x = self.varList.index(variable)
    new_var_list = self.varList[:pos_x] + self.varList[pos_x + 1:]
    new_node = Node("f" + str(new_var_list), new_var_list)
    new_cpt = dict()
    for k, v in self.cpt.items():
        new_k = k[:pos_x] + k[pos_x + 1:]
        if new_cpt.get(new_k) is None:
            new_cpt[new_k] = v
        else:
            new_cpt[new_k] += v
    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
pos_x = self.varList.index(variable)
new_var_list = self.varList[:pos_x] + self.varList[pos_x + 1:]
new_node = Node("f" + str(new_var_list), new_var_list)
new_cpt = dict()
for k, v in self.cpt.items():
    if k[pos_x] == str(value):
        new_cpt[k[:pos_x] + k[pos_x + 1:]] = v
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})

```

3.2 Results

```
> python2 VE.py
P(A) *****
RESULT:
Name = f['A']
vars ['A']
    key: 1 val : 0.002516442
    key: 0 val : 0.997483558

P(B | J~M) *****
RESULT:
Name = f['B']
vars ['B']
    key: 1 val : 0.0051298581334
    key: 0 val : 0.994870141867
```