An implementation of the Variable Elimination algorithm in Python

Anouk Prins
s4858956

Pleun Scholten
s4822250

27-05-2018

1 Introduction & Specification

This report describes a possible implementation of the Variable Elimination algorithm in Python. For this, the pandas library is used. The algorithm is applied on Bayesian Networks from the Bayesian Network repository.

2 Design

The algorithm that was implemented is called Variable Elimination. It is a algorithm to efficiently sum out variables in a logical order to obtain the query variable. Variable Elimination uses factors, which is a function from a tuple to a real or rational number. A factor can look like this: $f : X_1 \cdot \cdots \cdot X_k - > R$.

Below the Variable Elimination is described with the help of pseudo-code.

```
product_formula = network.probabilities #define the product formula
reduce_based_on_network(product_formula)
reduce_observed_variables(product_formula)
elimination_order = make_elimination_order(network)
for x in elimination_order:
    multiply_list = multiply_factors_containing_x(product_formula)
    new_factor = sum_out_x(multiply_list) #the multiplied factor is
        summed
    eliminate_and_add_new_factor(product_formula, new_factor) #x is
        eliminated from the product formula and the new factor is added
normalize(product_formula) #in this formula now only the query variable
    is left
```

A product formula is defined in the beginning and is updated every time it goes through the for-loop. It begins with all probability tables from the network. Then, probability tables will be reduced based on the network structure. Next, the observed variables will be reduced. For the for-loop an elimination order is needed. This elimination order is based on the network structure, leaf nodes will be eliminated first. Then, for every element in the elimination order, the factors containing this element will be multiplied. The next step is to sum these multiplied factors, so that the element is not in the factor anymore. This new factor is then added to the factor list while the factors containing that element are removed. After the algorithm has finished all elements in the elimination order, the factor is normalized and gives a real or rational number.

3 Implementation

The implementation of the algorithm is done in python. Python can represent the data in dataframes using the pandas library. This is an extensive library with functions made for data manipulation, perfect for the Variable Elimination algorithm. Thus, in this implementation of the Variable Elimination algorithm, the pandas library was used extensively.

The full code can be found in section 6.4 and section 6.5

When calculating the query variable on paper, one can write down the whole product formula with all factors in such form: $f : X_1 \cdot \cdots \cdot X_k$. However, this is not possible in python so the product formula is represented as a list of factors. The initial product formula consists of all the probability tables, that are given by the .bif file. Here it is called `probabilities`.

```
probabilities = self.network.probabilities
```

The next step would be to reduce the formula based on the network structure. However, this is done already by the framework given on Blackboard. Thus, only the observed variable needs to be reduced.

```
newProb = []
for keyO, valueO in observed.items():
    for keyP, prob in probabilities.items():
        if keyO in list(prob.columns.values):
            newProb = prob[prob.get(keyO) == valueO]
            probabilities[keyP] = newProb
```

After this, the probability tables from `probabilities` is added to `productList`. This list is used and updated throughout the whole algorithm.

```
productList = []
for key, prob in probabilities.items():
    productList.append(prob)
```

The next step will be to loop through the elimination order. The elimination order is obtained through the function found in Section 6.1.

A good elimination order consists of the leaf nodes first, which are nodes that are not parents, and then the other nodes in the network. The query does not belong in this list. First, a parent list is made. The nodes which are not in this list are not parents and thus are leaf nodes. So, the method checks which nodes are not in the parents list and adds those to a new list: `leafNodes`. Then, the leaf nodes are added to the final list `sortedList` and after that the parents from the `pList`. Lastly, the query variable is removed.

Now we loop through the elimination order. First, a new list is made which will be used for multiplying, summing and eliminating. This list is called `multList` and contains all probability tables where the to be eliminated variable `elim` occurs.

```
for elim in elim_order:
    multList = []
    for probDist in productList:
        if elim in list(probDist.columns.values):
            multList.append(probDist)
```

Next, the tables are merged on the eliminated variable `elim`. This is done by a very convenient function from the pandas library: `dataframe1.merge(dataframe2)`. To merge all the tables in the `multList`, the function `reduce()` is used. This is a useful function for performing some computation on a list and returning the result. A lambda function is normally used within this function.

```
multList = reduce(lambda x,y: x.merge(y, on = elim, suffixes=('_1',
    '_2')), multList)
```

At this moment, `multList` exists of one item. The next step is to multiply. This is done by the following (very long) line of code:

```
multList['newProb'] = multList.apply(lambda row:
    (row['prob_1']*row['prob_2'] if 'prob_1' in
    list(multList.columns.values) and 'prob_2' in
    list(multList.columns.values) else row['prob']), axis = 1)
```

A new column is made in `multList` called `newProb`. Via a lambda function, two probabilities are multiplied per row or nothing is multiplied when there is only one probability column. This result is added in `newProb`. After this, the old probabilities are removed (see Appendix 6.2).

Next is summing out the eliminated variable.

```
multList.rename({'newProb':'prob'}, axis = 1, inplace = True)

colValues = list(multList.columns.values)
colValues.remove(elim)
```

```
colValues.remove('prob')
sum_bodyoncetoldme =
    pd.DataFrame(multList.groupby(colValues).sum().reset_index())
```

First, the column `newProb` is renamed to `prob`. This is so the new probabilites can be used in a new run through the loop later. A list `colValues` is made with all names of the columns in `multList`. Then the to be eliminated variable is removed and the probability column. Next, the probabilities are grouped by the list `colValues` and summed with the help of another useful function from the pandas library: `groupby().sum()`. This is saved in a variable and is added to `productList`.

```
productList = [factor for factor in productList if elim not in
    list(factor.columns.values)]
productList.append(sum_bodyoncetoldme)
```

In this way, there is no distinct step where the to be eliminated variable is actually deleted. It happens implicitly in the summing and multiplying.

The process described above is repeated until `elim_order` is empty. Then, the only variable left to eliminate is the query variable.

```
productList = reduce(lambda x,y: x.merge(y, on = query, suffixes =
    ['_1','_2']), productList)

probFalse = 1.0
for col in productList.iloc[0]:
    if isinstance(col, float):
        probFalse = probFalse * col
probTrue = 1.0
for col in productList.iloc[1]:
    probTrue = 1.0
    if isinstance(col, float):
        probTrue = probTrue * col
productList.loc[1,'finalProb'] = probTrue
productList.loc[0,'finalProb'] = probFalse
```

The functions `reduce()` and `merge()` are used again in the same way as in the loop above. Then, `probFalse` is initialized with 1.0. A for-loop is constructed that goes through the product list in the 0^{th} column and multiplies `probFalse` with the value of that row in that column. It does this until there are no rows left anymore. The same is done for the 1^{st} column. Then, a new column is constructed called `finalProb` and it is given the just obtained probability values. After this, some cleaning up of the columns is done (see Appendix 6.3).

The very last step is to normalize.

```
normalizing = productList['finalProb'].sum()
```

```
for index, row in productList.iterrows():
    productList.loc[index,'finalProb'] = row['finalProb']/normalizing
```

A normalizing variable is initialized which is the sum of the final probabilities. Then, every final probability is divided by the normalizing variable. This is saved in `productList`.

4 Testing

4.1 Binary variables

The main limitation of our code is that the algorithm only works for binary variables. In multiple parts of our code we assume the data is binary. For example, at the very end we use the following code:

```
probFalse = 1.0
for col in productList.iloc[0]:
    if isinstance(col, float):
        probFalse = probFalse * col
probTrue = 1.0
for col in productList.iloc[1]:
    if isinstance(col, float):
        probTrue = probTrue * col
productList.loc[1,'finalProb'] = probTrue
productList.loc[0,'finalProb'] = probFalse
```

In this code we calculate the probabilities of our query variable being `True` and `False`. However, this very explicitly assumes a `True` and `False` value only.

Test:

Input:

Network	Observed	Query
`cancer.bif`	{}	`Cancer`

Expected outcome:
A normal discrete probability distribution.

Actual outcome:

Final:	Cancer	finalProb
0	False	0.98837
1	True	0.01163

Thus, for a binary Bayesian Network, the code works as intended.

Input:

Network	Observed	Query
alarm.bif	{}	HR

Expected outcome: An error, since there are non-binary variables in this Bayesian Network, e.g. `CVP: {LOW, NORMAL, HIGH}`.

Actual outcome:

```
File "variable_elim.py", line 70, in run
   multList['newProb'] = multList.apply(lambda row:
       (row['prob_1']*row['prob_2'] if 'prob_1' in
       list(multList.columns.values) and 'prob_2' in
       list(multList.columns.values) else row['prob']), axis = 1)

ValueError: Wrong number of items passed 4, placement implies 1
```

Thus, for a non-binary Bayesian Network, the code breaks as expected.

4.2 Observed variables

Our code works for any number of valid observed variables, which means that for a network of `N` variables, we can have `N-1` observed variables. The only variable that can't be observed is the query variable. This is logical, since calculating a probability distribution for an observed variable is both useless and trivially easy.

Test:

Input:

Network	Observed	Query
cancer.bif	{}	Cancer

Expected outcome:
A normal discrete probability distribution with a low probability for `Cancer = True`.

Actual outcome:

Final:	Cancer	finalProb
0	False	0.98837
1	True	0.01163

Thus, our code works with no observed variables.

Input:

Network	Observed	Query
`cancer.bif`	{'Xray':'positive','Dyspnoea':'True','Smoker':'True','Pollution':'high'}	`Cancer`

Expected outcome:
A discrete probability distribution with a high probability for `Cancer = True`.

Actual outcome:

Final:	Cancer	finalProb
0	False	0.66087
1	True	0.33913

Thus, our code works as expected with `N-1` observed variables for a Bayesian Network of `N` variables.

Input:

Network	Observed	Query
`cancer.bif`	{'Cancer':'True'}	`Cancer`

Expected outcome:
An error, since the query is contained in the observed list.

Actual outcome:

```
File "variable_elim.py", line 107, in run
  for col in productList.iloc[1]:

IndexError: single positional indexer is out-of-bounds
```

Thus, our code works as expected when the query variable is contained in the observed list.

4.3 Common column names

When multiplying, one bug in our code is that we `merge` only on the elimination variable. E.g. in the following situation:

Network	Observed	Query
`asia.bif`	{}	`either`

The last step contains the column names `either_1` and `either_2`, instead of just `either` columns. This is because in the case of multiplying a dataframe with column names {`A, B, C`} and a dataframe with column names {`A, B`}`d` where `elim = A`, we only merge on `elim`. Instead, the merge should happen on the intersection between the two lists of column names.

Test:

Input:

Network	Observed	Query
`cancer.bif`	{}	`Cancer`

Expected outcome:
A normal discrete probability distribution.

Actual outcome:

Final:	Cancer	finalProb
0	False	0.98837
1	True	0.01163

Thus, our code works when there is only one common column name for every combination of factors.

Input:

Network	Observed	Query
`asia.bif`	{}	`either`

Expected outcome:
A normal discrete probability distribution.

Actual outcome:

```
File "variable_elim.py", line 98, in run
  productList = reduce(lambda x,y: x.merge(y, on = query, suffixes =
      ['_1','_2']), productList)

KeyError: 'either'
```

This is because the `productList` at that moment looks as follows:

	either	prob
0	no	1.0
1	yes	1.0

,

	either_1	either_2	prob
0	no	no	1.0
1	no	yes	1.0
2	yes	no	1.0
3	yes	yes	1.0

At some point, we should have merged on both `either` and some `elim` variable. However, by only merging on the `elim`, we created two new variables, `either_1` and `either_2`.
Obviously, the probabilities are also wrong int this table (all values being 1.0).

Thus, our code does not work when there are more than one common column names for any combination of factors.

5 Conclusion

Our implementation of Variable Elimination works decently well. This implementation will calculate the proper discrete probability distribution for simple binary Bayesian Networks. However, the major disadvantage is that our implementation does not work when multiplying over multiple variables at once. So only simple Bayesian Networks, where multiplying over the elimination variable suffices, can be evaluated with our implementation.

6 Appendices

Appendix 1, 2 and 3 are smaller parts of code.
Appendix 4 and 5 contain the full code of the two files created by us. The `BayesNet` object imported from `read_bayesnet` in `run.py` is all taken from the framework given on Blackboard and is thus not given here.

6.1 Appendix 1

```
def sort(nodes, parents, query):
    # Make list of all parents
    pList = []
    parent = list(parents.values())
    for p1 in parent:
        for p2 in p1:
            pList.append(p2)
    # Remove duplicates
    pList = list(set(pList))
    # Add all leaf nodes.
    leafNodes = [x for x in nodes if x not n pList]

    # Add to sorted list
    sortedList = []
    sortedList.extend(leafNodes)
    sortedList.extend(pList)
    # Remove query and return
    if query in sortedList:
        sortedList.remove(query)
    return sortedList
```

6.2 Appendix 2

```
# Clean up
if 'prob' in list(multList.columns.values):
    multList.drop('prob', axis = 1, inplace = True)
```

```
if 'prob_1' in list(multList.columns.values) and 'prob_2' in
     list(multList.columns.values):
    multList.drop('prob_1', axis = 1, inplace = True)
    multList.drop('prob_2', axis = 1, inplace = True)
```

6.3 Appendix 3

```
if 'prob' in list(productList.columns.values):
    productList.drop('prob', axis = 1, inplace = True)
if 'prob_1' in list(productList.columns.values) and 'prob_2' in
     list(productList.columns.values):
    productList.drop('prob_1', axis = 1, inplace = True)
    productList.drop('prob_2', axis = 1, inplace = True)
```

6.4 Appendix 4L `run.py`

```
"""
@Author: Joris van Vugt, Moira Berens

Entry point for testing the variable elimination algorithm

"""
from read_bayesnet import BayesNet
from variable_elim import *

if __name__ == '__main__':

    def sort(nodes, parents, query):
        # Make list of all parents
        pList = []
        parent = list(parents.values())
        for p1 in parent:
            for p2 in p1:
                pList.append(p2)
        # Remove duplicates
        pList = list(set(pList))
        # Add all leafs first.
        leafNodes = [x for x in nodes if x not in pList]

        # Add to sorted list
        sortedList = []
        sortedList.extend(leafNodes)
        sortedList.extend(pList)

        # Remove query and return
        if query in sortedList:
```

```
        sortedList.remove(query)
    return sortedList

# the class BayesNet represents a Bayesian network from a .bif file
    in several variables
net = BayesNet('cancer.bif')

# Create object
ve = VariableElimination(net)

# Observed variables
observed = {}

# Query
query = 'Cancer'

# Elimination order
elim_order = sort(net.nodes, net.parents, query)

ve.run(query, observed, elim_order)
```

6.5 Appendix 5: variable_elim.py

```
"""
@Author: Joris van Vugt, Moira Berens

Implementation of the variable elimination algorithm for AISPAML
    assignment 3

"""
import pandas as pd

class VariableElimination():

    def __init__(self, network):
        self.network = network
        self.addition_steps = 0
        self.multiplication_steps = 0

    def run(self, query, observed, elim_order):
        """
        Use the variable elimination algorithm to find out the
            probability
        distribution of the query variable given the observed variables

        Input:
            query:     The query variable
```

```
    observed:  A dictionary of the observed variables {variable:
        value}
    elim_order: Either a list specifying the elimination ordering
                or a function that will determine an elimination
                    ordering
                given the network during the run

Output: A variable holding the probability distribution
        for the query variable

"""
# What is the product formula
# The reduced formula based on network structure? -- Is already
    given in the dataset
# Identify factors and reduce observed variables -- Sorta done
    needs flexibility
# Fix an elimination ordering -- Done
# For every variable in elim_order:
    # Multiply factors containing that variable -- Done
    # Sum out the variable to obtain new factor --
    # Remove the multiplied factors from the list and add the
        summed out factor
# Normalize.

# Dictionary is the reduced formula of factors.
# Reducing oberved variable = eliminate all values with
    incorrect observed value.

# Reduce observed.
probabilities = self.network.probabilities
newProb = []
for keyO, valueO in observed.items():
    for keyP, prob in probabilities.items():
        if keyO in list(prob.columns.values):
            newProb = prob[prob.get(keyO) == valueO]
            probabilities[keyP] = newProb

# Prepare a list of all the factors containing certain variables.
productList = []
for key, prob in probabilities.items():
    productList.append(prob)

for elim in elim_order:
    multList = []
    for probDist in productList:
        if elim in list(probDist.columns.values):
            multList.append(probDist)

    multList = reduce(lambda x,y: x.merge(y, on = elim,
        suffixes=('_1', '_2')), multList)
```

```
        # Multiply
        multList['newProb'] = multList.apply(lambda row:
            (row['prob_1']*row['prob_2'] if 'prob_1' in
            list(multList.columns.values) and 'prob_2' in
            list(multList.columns.values) else row['prob']), axis =
            1)
        self.multiplication_steps += 1

        # Clean up
        if 'prob' in list(multList.columns.values):
            multList.drop('prob', axis = 1, inplace = True)
        if 'prob_1' in list(multList.columns.values) and 'prob_2' in
            list(multList.columns.values):
            multList.drop('prob_1', axis = 1, inplace = True)
            multList.drop('prob_2', axis = 1, inplace = True)

        multList.rename({'newProb':'prob'}, axis = 1, inplace = True)

        # Summing
        colValues = list(multList.columns.values)
        colValues.remove(elim)
        colValues.remove('prob')
        sum_bodyoncetoldme =
            pd.DataFrame(multList.groupby(colValues).sum().reset_index())
        self.addition_steps += 1

        # Update productList
        productList = [factor for factor in productList if elim not
            in list(factor.columns.values)]
        productList.append(sum_bodyoncetoldme)

    # Last multiplication of the query variable.
    productList = reduce(lambda x,y: x.merge(y, on = query, suffixes
        = ['_1','_2']), productList)

    probFalse = 1.0
    for col in productList.iloc[0]:
        if isinstance(col, float):
            probFalse = probFalse * col
    probTrue = 1.0
    for col in productList.iloc[1]:
        if isinstance(col, float):
            probTrue = probTrue * col
    productList.loc[1,'finalProb'] = probTrue
    productList.loc[0,'finalProb'] = probFalse
    self.multiplication_steps += 1

    # Clean up
    if 'prob' in list(productList.columns.values):
```

```
    productList.drop('prob', axis = 1, inplace = True)
if 'prob_1' in list(productList.columns.values) and 'prob_2' in
     list(productList.columns.values):
    productList.drop('prob_1', axis = 1, inplace = True)
    productList.drop('prob_2', axis = 1, inplace = True)

# Normalize
normalizing = productList['finalProb'].sum()

for index, row in productList.iterrows():
    productList.loc[index,'finalProb'] =
        row['finalProb']/normalizing

#print productList

productList =
     pd.DataFrame(productList.groupby(query).sum().reset_index())

print "Final: {}\n\n Additions: {}\n Multiplications:
     {}".format(productList, self.addition_steps,
     self.multiplication_steps)
```