

Spyn 01 - Potentials

August 14, 2020

[]:

1 Spyn: Hyper-graphical Probabilistic Inference

Hyper-graphical Probabilistic Inference.

Probabilistic Inference whose model data is a set of “hyper-edges” that provide probabilistic information on subsets of random variables.

It constitutes a generalization of Bayesian Networks. The join tree of a Bayesian Network is a hyper-tree.

Sounds fancy, ay?

Well, first, know that at this point it’s powerful, yet limited in scope: for example, unless you do some wrap it in some more constructs your self, it won’t handle non-discrete variables.

But more importantly, know that there’s a simpler view to this, and you don’t need to know any Bayesian statistics at all. You know data? You’ve kicked data around a bit in the past. You’re fine.

In this tutorial, we start by counting (which you surely know how to do), and put these counts in tables (which is not too hard), and will introduce the necessary concepts of the present “building blocks” of (discrete) probability inference from there.

Enjoy.

If you want to see **spyn** in action, and get some ideas of how to bring your data to a point that you can **spyn** it, see [this \(multiple grant-ly funded\) project](#).

1.1 Install

We suggest just doing a

```
pip install spyn
```

from your terminal.

But if you’re the picky kind, you can install from [source](#).

The only requirements are **matplotlib**, **pandas**, and **numpy**, which you probably already have, since I’m guessing you’re an ML kind of person.

2 Potentials - A key data structure to Discrete Bayesian Inference

```
[1]: # %load_ext autoreload
      # %autoreload 2
```

```
[2]: from spyn.ppi.pot import Pot, _val_div_, _val_prod_
      import pandas as pd
```

A potential is basically a map between a finite discrete multi-dimensional space and numbers.

These numbers usually represent counts, probabilities, or likelihoods.

But what's more important than the data structure contents itself are the operations you perform on them, which define their actual semantics.

A potential often represents some knowledge about all possible combinations of a set of (finite discrete) variables. For example, a potential on two binary variables A and B will give you a number for each of the four combinations of values that A and B can take together. For example:

```
[3]: ab_counts = Pot(pd.DataFrame({'A': [0,0,1,1], 'B': [0,1,0,1], 'pval':
      ↪ [2,4,5,9]}))
      print(ab_counts)
```

A	B	pval
0	0	2
0	1	4
1	0	5
1	1	9

could be a “count” potential that lists the number of times you’ve observed $A = 0$ and $B = 0$ together (2 times in this case), how $A = 0$ and $B = 1$ together (3 times), etc.

2.0.1 Projecting

Perhaps you want to know only about A counts. In that case you would do this:

```
[4]: print((ab_counts['A']))
```

A	pval
0	6
1	14

`ab_counts['A']` is syntactic sugar for

```
ab_counts.project_to(var_list=['A']),
```

which could also be written

```
ab_counts >> ['A']
```

```
[5]: ab_counts.project_to(var_list=['A'])
```

```
[5]:      pval
      A
      0      6
      1     14
```

```
[6]: ab_counts >> ['A']
```

```
[6]:      pval
      A
      0      6
      1     14
```

Projecting to a variable or set of variables is equivalent to (what most people call) “marginalizing out” the other variables. You basically remove them from the table, and then group the remaining variables (according to their value combinations), summing up the pvals.

In fact, if you have some data, in the form of a (pandas) DataFrame, that has only discrete values (or you forced them to be so), you can simply add a ‘pval’ column set to 1, make it a potential, and project it to the variable list. The result will be a “count” potential for your data. For example:

```
[7]: data = pd.DataFrame({'A': [0,0,1,1,0,0,1,0], 'B': [0,1,0,1,1,0,1,1]})
      print(data)
```

```
      A  B
0  0  0
1  0  1
2  1  0
3  1  1
4  0  1
5  0  0
6  1  1
7  0  1
```

```
[8]: data['pval'] = 1
      print((Pot(data)[['A','B']]))
```

```
      A  B  pval
0  0  0      2
0  0  1      3
1  1  0      1
1  1  1      2
```

Let’s look at that ab_counts again:

```
[9]: ab_counts
```

```
[9]:      pval
      A  B
      0  0      2
```

	1	4
1 0	5	
1	9	

Let's say that you want to get the probabilities (observed ratios, really) from this count potential. You could just do this:

```
[10]: ab_probs = ab_counts / []
      print(ab_probs)
```

A	B	pval
0	0	0.10
0	1	0.20
1	0	0.25
1	1	0.45

Ah, a lot to say here! First, the expression

`ab_count / []`

is syntactic sugar for "divide `ab_count` by it's projection onto `[]`, or

`ab_count / ab_count[None]`

(python syntax doesn't allow us to say "`ab_count[]`", so we need that `None`).

We know what projection to a subset of variables is, but what about projecting on nothing? Well, if projecting onto a subset is summing up all other variables/dimensions, then it makes sense that projecting onto the empty subset is simply summing up all existing variables, leaving only a number, with no variables attached to it!

Projecting to NO VARIABLES means marginalizing out ALL VARIABLES. So you just sum up all the pvals:

```
[11]: ab_counts[None]
```

```
[11]: pval
      20
```

So doing

`abcounts / []`

simply divides each of the pvals (which are counts) by the total, thus giving you a ratio, which you can consider to be a probability if the context (or your personal frequentist beliefs about probability allows). So

`abcounts / []`

is basically

```
[12]: print((ab_counts / ab_counts[None]))
```

A	B	pval
0	0	0.10
0	1	0.20
1	0	0.25
1	1	0.45

2.0.2 Slicing

So you have a potential representing $Prob(AB)$, the joint probability of every combination of variable values of A and B . What if you wanted to get an actual probability out of it? Say, you wanted to know what $Prob(A = 0, B = 1)$. You can do that!

```
[13]: ab_probs[{'A': 0, 'B': 1}]
```

```
[13]:    pval
      0.2
```

(Note: you get a singleton (that is, without variables) Pot as an answer. If you wanted the probability as a float, you could say `ab_probs[{'A': 0, 'B': 1}].tb.pval[0]`)

Ah, projection again! Yes, but with a bit of abuse. Here we're projecting not to a whole variable space (that is, including all its possible values), but to a single point in the cartesian product space of these random variables (oops, did I say it? the big old RV word?). You could also project to a hyper-plane:

```
[14]: ab_probs[{'A': 0}]
```

```
[14]:    pval
      B
      0    0.1
      1    0.2
```

You might have expected `ab_probs[{'A': 0}]` to be the probability $Prob(A = 0)$, but it's not. What it is, is a table listing the probabilities that $A = 0$ and $B = b$ for each possible values b of B . What you're doing when you do `X['A': 0]` is asking for the hyper-plane of X that goes through $A = 0$. If you want `X['A': 0]` to give you $Prob(A = 0)$, you have to ask the right probability space X , namely the one projection (again!?) of the AB space onto A (because you don't care about the B component here:

```
[15]: ab_probs['A'][{'A': 0}]
```

```
[15]:    pval
      0.3
```

Though you could have also asked the $A = 0$ hyper-plane to marginalize out all the variables remaining (in our instance, just B), which means summing up all the probabilities in `ab_probs[{'A': 0}]`, and get the same result:

```
[16]: ab_probs[{'A': 0}][None]
```

```
[16]: pval
      0.3
```

`ab_probs[{'A': 0}]` wasn't very meaningful alone, but it becomes more meaningful if we divide it by the probability $Prob(A = 0)$, since it will then give us $Prob(B | A = 0)$ (that is to say, a table that lists $Prob(B = 0 | A = 0)$ and $Prob(B = 1 | A = 0)$). Indeed, as you may remember from Kindergarten, $Prob(B | A) = \frac{Prob(AB)}{Prob(A)}$:

```
[17]: ab_probs[{'A': 0}] / ab_probs[{'A': 0}] [None]
```

```
[17]: pval
      B
      0  0.333333
      1  0.666667
```

which you could have also written as

```
[18]: ab_probs[{'A': 0}] / []
```

```
[18]: pval
      B
      0  0.333333
      1  0.666667
```

2.0.3 Dividing, Multiplying, or adding two potentials

We've divided a potential by a number before. That was easy. But actually we were really dividing by a singleton potential.

What would the division, or multiplication, or addition of two potentials look like in general?

Well, dividing, multiplying, or adding two potentials is done in two steps: (1) join the two tables on their common variables (or take the cartesian product if no common variables) (2) apply the operation on the aligned pvals

For example, take two potentials, one on XY, and one on YZ:

```
[19]: t = Pot({'X': [0,0,1,1], 'Y': [0,1,0,1], 'pval':[2,3,5,7]})
      tt = Pot({'Y': [0,1,1], 'Z': [0,1,2], 'pval':[10,100,1000]})
      print(t)
      print("")
      print(tt)
```

X	Y	pval
0	0	2
0	1	3
1	0	5
1	1	7

Y	Z	pval
0	0	10
1	1	100
1	2	1000

0	0	10
1	1	100
1	2	1000

Now say you want to multiply them... First join the tables on the common variable Y:

```
[20]: t._merge_(tt)
```

```
[20]:
```

	X	Y	pval_x	Z	pval_y
0	0	0	2	0	10
1	1	0	5	0	10
2	0	1	3	1	100
3	0	1	3	2	1000
4	1	1	7	1	100
5	1	1	7	2	1000

... and then multiply the aligned pvals

```
[21]: _val_prod_(t._merge_(tt))
```

```
[21]:
```

	X	Y	Z	pval
0	0	0	0	20
1	1	0	0	50
2	0	1	1	300
3	0	1	2	3000
4	1	1	1	700
5	1	1	2	7000

Intuitively, what is happening when you operate on two potentials, is that you align/link the common information, and broadcast/mix the information that they don't have in common. This intuition is easier to grasp when their no variable in common (so there's only mixing (Cartesian Product) involved):

```
[22]: t = Pot({'X': [0,1], 'pval':[2,3]})
      tt = Pot({'Y': [0,1], 'pval':[10,100]})
      print((t / tt))
```

	X	Y	pval
0	0	0	0.20
0	1	0	0.02
1	0	0	0.30
1	1	0	0.03

2.0.4 But why? (growing up...)

Most laymen understand $Prob(B|A) = Prob(AB)/Prob(A)$ to mean $Prob(B = b | A = a) = Prob(A = a, B = b)/Prob(A = a)$, that is, where the probabilities are numbers representing specific events like $A = 0$ and $B = 1$. But what the (more sophisticated) Bayesians mean by *Prob*

is usually a whole probability (or likelihood, for the fanatics) space”. $Prob(A, B)$ therefore contains the information for each $Prob(A = a, B = b)$ for all possible $(a, b) \in A \times B$:

```
[23]: ab_probs
```

```
[23]:      pval
      A B
0 0  0.10
  1  0.20
1 0  0.25
  1  0.45
```

Potentials are a data structure to hold (and operate) on these very useful objects.

Note that $Prob(B|A)$ is a potential $A \times B$, just as $Prob(A, B)$ is. But their semantics are different. $Prob(B|A)$ value for (a, b) can be interpreted as “the probability that $B=b$, given that $A=a$ ”. We can get $Prob(B|A)$ in one shot by dividing $Prob(A, B)$ by its projection onto A :

```
[24]: ab_probs / ab_probs['B']
```

```
[24]:      pval
      A B
0 0  0.285714
  1 0  0.714286
0 1  0.307692
  1 1  0.692308
```

Which you could also write as

```
[25]: ab_probs / ['B']
```

```
[25]:      pval
      A B
0 0  0.285714
  1 0  0.714286
0 1  0.307692
  1 1  0.692308
```

or

```
[26]: ab_probs.normalize(['B'])
```

```
[26]:      pval
      A B
0 0  0.285714
  1 0  0.714286
0 1  0.307692
  1 1  0.692308
```


We got $Prob(B|A)$ as $Prob(AB)/Prob(A)$. Let's verify that multiplication works as expected, namely that we get $Prob(B|A) \times Prob(A) = Prob(A, B)$:

```
[27]: ab_probs.normalize(['B']) * ab_probs['B']
```

```
[27]:      pval
      A B
0 0  0.10
1 0  0.25
0 1  0.20
1 1  0.45
```

Look at what you get if you do $Prob(A) \times Prob(B)$:

```
[28]: ab_probs['A'] * ab_probs['B']
```

```
[28]:      pval
      A B
0 0  0.105
    1  0.195
1 0  0.245
    1  0.455
```

You don't get the same thing as $Prob(A, B)$, because that would only happen if A and B were independent.

2.0.5 Inference (or “integrating evidence”)

Let's come back to `probs_ab`:

```
[29]: ab_probs
```

```
[29]:      pval
      A B
0 0  0.10
    1  0.20
1 0  0.25
    1  0.45
```

If you had evidence about A (which is Bayesian for “new something about A ”), you can change your idea of B . What do you know about B now? You know (projecting your `ab_probs` information to B), that

```
[30]: ab_probs['B']
```

```
[30]:      pval
      B
0  0.35
1  0.65
```

If you knew that $A = 0$, then you should really just take the hyper-plane of `ab_probs` corresponding to that, and normalizing (which is the same as projecting to `[]` (or `[None]`) as we did earlier), you'd get

```
[31]: ab_probs[{'A': 0}].normalize([])
```

```
[31]:      pval
B
0  0.333333
1  0.666667
```

So you're $A = 0$ evidence changed your idea of B .

Evidence like $A = 0$ is called “hard evidence”. An example of “soft evidence” would be if you're not sure of the exact value of A , but you're, say, 20% sure that $A = 0$, and (by consequence), 80% sure that $A = 1$. Oh! How can we represent this? With a potential!

```
[32]: soft_evidence = Pot(pd.DataFrame({'A': [0, 1], 'pval': [0.2, 0.8]}))
soft_evidence
```

```
[32]:      pval
A
0    0.2
1    0.8
```

Now if you multiply `ab_probs` by this evidence potential, normalize, and project to B , you'll get the consequence on B of this evidence on A

```
[34]: (ab_probs * soft_evidence).normalize([])['B']
```

```
[34]:      pval
B
0  0.354839
1  0.645161
```

Mathematically, what we've done is

$$Prob(B \mid A = e) = Prob(A = e \mid B) \times Prob(B) / Prob(A = e)$$

```
[ ]:
```

3 Now, Play!

The official tutorial is over. Now you are entering the far west: A place where the code is little or not commented at all.

But that doesn't mean it's useless. The following exhibits a bunch of functionalities we haven't covered so far, or recipes you can think of as you play.

```
[36]: import pandas as pd
import numpy as np
import os
from spyn.ppi.pot import Pot
```

3.0.1 Making a few potentials from count data

```
[37]: count_data = ut.daf.get.rand(nrows=34, values_spec=[list(range(2)),
↳list(range(2)), list(range(2)), list(range(4))], columns=['A', 'B', 'C',
↳'count'])
print("\n--- count_data data ---")
print(count_data.head())
print('etc.')
abc = Pot.from_count_df_to_count(count_df=count_data[['A', 'B', 'C', 'count']],
↳count_col='count')
ab = abc[['A', 'B']]
bc = abc[['B', 'C']]
b = abc[['B']]
print("\n--- abc count potential ---")
print(abc)
print("\n--- ab count potential ---")
print(ab)
print("\n--- bc count potential ---")
print(bc)
print("\n--- b count potential ---")
print(b)
```

```
--- count_data data ---
   A  B  C  count
0  0  0  0     3
1  1  1  0     2
2  0  0  1     2
3  0  0  1     3
4  0  0  1     2
etc.
```

```
--- abc count potential ---
   A  B  C  pval
0  0  0  0    11
0  0  1     7
0  1  0     0
0  1  1     5
1  0  0    13
1  0  1     3
1  1  0    12
1  1  1     6
```

```
--- ab count potential ---
```

A	B	pval
0	0	18
0	1	5
1	0	16
1	1	18

```
--- bc count potential ---
```

B	C	pval
0	0	24
0	1	10
1	0	12
1	1	11

```
--- b count potential ---
```

B	pval
0	34
1	23

3.0.2 Making a few potentials from pts data

```
[38]: pts = ut.daf.get.rand(nrows=34, values_spec=[list(range(2)), list(range(2)), list(range(2)), list(range(2))], columns=['A', 'B', 'C'])
print("\n--- pts data ---")
print(pts.head())
print('etc.')
abc = Pot.from_points_to_count(pts[['A', 'B', 'C']])
ab = abc[['A', 'B']]
bc = abc[['B', 'C']]
b = abc[['B']]
print("\n--- abc count potential ---")
print(abc)
print("\n--- ab count potential ---")
print(ab)
print("\n--- bc count potential ---")
print(bc)
print("\n--- b count potential ---")
print(b)
```

```
--- pts data ---
```

	A	B	C
0	1	0	0
1	0	1	0
2	1	1	0
3	1	1	0
4	0	1	0

etc.

```
--- abc count potential ---
```

A	B	C	pval
0	0	0	6
0	0	1	3
0	1	0	6
0	1	1	4
1	0	0	5
1	0	1	3
1	1	0	6
1	1	1	1

```
--- ab count potential ---
```

A	B	pval
0	0	9
0	1	10
1	0	8
1	1	7

```
--- bc count potential ---
```

B	C	pval
0	0	11
0	1	6
1	0	12
1	1	5

```
--- b count potential ---
```

B	pval
0	17
1	17

3.0.3 Having a look at some operations

```
[39]: prime_list = np.array([2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97])
float_list = prime_list

idx = 0; idx_end = len(abc.tb)
abc.tb['pval'] = float_list[idx:idx_end]

idx = 0; idx_end = len(ab.tb)
ab.tb['pval'] = float_list[idx:idx_end]

idx = 0; idx_end = len(bc.tb)
bc.tb['pval'] = float_list[idx:idx_end]
```

```

idx = 0; idx_end = len(b.tb)
b.tb['pval'] = float_list[idx:idx_end]

idx = 0; idx_end = len(b.tb)
b.tb['pval'] = float_list[idx:idx_end]

a = Pot(ut.daf.ch.ch_col_names(b.tb, 'A', 'B'))
idx = 0; idx_end = len(a.tb)
a.tb['pval'] = float_list[idx:idx_end]

# idx = 0; idx_end = idx + len(abc.tb)
# abc.tb['val'] = float_list[idx:idx_end]

# idx = idx_end; idx_end = idx + len(ab.tb)
# ab.tb['val'] = float_list[idx:idx_end]

# idx = idx_end; idx_end = idx + len(bc.tb)
# bc.tb['val'] = float_list[idx:idx_end]

# idx = idx_end; idx_end = idx + len(b.tb)
# b.tb['val'] = float_list[idx:idx_end]

# idx = idx_end; idx_end = idx + len(b.tb)
# b.tb['val'] = float_list[idx:idx_end]

# a = Pot(ut.daf.ch.ch_col_names(b.tb, 'A', 'B'))
# idx = idx_end; idx_end = idx + len(a.tb)
# a.tb['val'] = float_list[idx:idx_end]

print("\n--- abc count potential ---")
print(abc)
print("\n--- ab count potential ---")
print(ab)
print("\n--- bc count potential ---")
print(bc)
print("\n--- b count potential ---")
print(b)
print("\n--- a count potential ---")
print(a)

```

```

--- abc count potential ---
  A  B  C  pval
0  0  0    2
0  0  1    3
0  1  0    5
0  1  1    7

```

1	0	0	11
1	0	1	13
1	1	0	17
1	1	1	19

--- ab count potential ---

A	B	pval
0	0	2
0	1	3
1	0	5
1	1	7

--- bc count potential ---

B	C	pval
0	0	2
0	1	3
1	0	5
1	1	7

--- b count potential ---

B	pval
0	2
1	3

--- a count potential ---

A	pval
0	2
1	3

```
[41]: print(ab * bc)
```

A	B	C	pval
0	0	0	4
0	0	1	6
1	0	0	10
1	0	1	15
0	1	0	15
0	1	1	21
1	1	0	35
1	1	1	49

```
[42]: print(ab * b)
```

A	B	pval
0	0	4
1	0	10
0	1	9
1	1	21

```
[43]: print(a * b)
```

A	B	pval
0	0	4
0	1	6
1	0	6
1	1	9

```
[44]: print(a._merge_(b))
print('-----')
print(a / b)
```

	A	pval_x	B	pval_y
0	0	2	0	2
1	0	2	1	3
2	1	3	0	2
3	1	3	1	3

A	B	pval
0	0	1.000000
0	1	0.666667
1	0	1.500000
1	1	1.000000

Marginalization (or “projection”, or “factoring in”, or “normalization”)

```
[45]: ab
```

```
[45]:      pval
A B
0 0      2
   1      3
1 0      5
   1      7
```

```
[46]: ab >> ['A']
```

```
[46]:      pval
A
0      5
1     12
```

```
[47]: ab['A']
```

```
[47]:      pval
A
0      5
1     12
```



```
[48]: ab['B']
```

```
[48]:      pval  
      B  
0      7  
1     10
```

```
[49]: # marginalization happens over the intersection of the given variable list and  
      ↪ the variables the potential actually has  
      # (other variables are ignored)  
      ab>>['A', 'variable that does not exist', 'blah blah']
```

```
[49]:      pval  
      A  
0      5  
1     12
```

```
[50]: # marginalizing over the whole set of variables doesn't do anything  
      ab[['A', 'B']]
```

```
[50]:      pval  
      A B  
0 0     2  
   1     3  
1 0     5  
   1     7
```

```
[51]: # marginalizing over nothing (the empty set) essentially normalizes the val  
      ↪ vector  
      # (divides each element by the sum of all elements)  
      ab[None]
```

```
[51]:      pval  
      17
```

Conditioning (as in $\text{Prob}(A|B)$)

```
[52]: bc
```

```
[52]:      pval  
      B C  
0 0     2  
   1     3  
1 0     5  
   1     7
```

```
[53]: # Conditioning on [] is like normalization. If you start with a count
      ↪ potential, you get a joint-probability table
      # (with probabilities badly estimated in this small sample case)
      # Note that all pvals sum to 1
      bc / []
```

```
[53]:          pval
      B C
0 0  0.117647
   1  0.176471
1 0  0.294118
   1  0.411765
```

```
[54]: # The following is like the conditional probability  $P(B|C) = P(BC)/P(B)$ 
      # Note that all pvals with same B sum up to 1
      bc / 'B'
```

```
[54]:          pval
      B C
0 0  0.400000
   1  0.600000
1 0  0.416667
   1  0.583333
```

```
[55]: abc / ['A']
```

```
[55]:          pval
      A B C
0 0 0  0.117647
   1  0.176471
   1 0  0.294118
   1  0.411765
1 0 0  0.183333
   1  0.216667
   1 0  0.283333
   1  0.316667
```

```
[56]: abc / ['A', 'B']
```

```
[56]:          pval
      A B C
0 0 0  0.400000
   1  0.600000
   1 0  0.416667
   1  0.583333
1 0 0  0.458333
   1  0.541667
```

```
1 0 0.472222
1 0 0.527778
```

```
[57]: abc / ['A', 'B', 'C']
```

```
[57]:          pval
      A B C
0 0 0 1.0
    1 1.0
    1 0 1.0
    1 1.0
1 0 0 1.0
    1 1.0
    1 0 1.0
    1 1.0
```

Slicing (taking sub-spaces defined by intercepts)

```
[58]: ab[{'A':0, 'B':1}]
```

```
[58]:          pval
      3
```

```
[59]: ab[{'A':0}]
```

```
[59]:          pval
      B
0      2
1      3
```

3.0.4 A few utils

```
[60]: ab
```

```
[60]:          pval
      A B
0 0      2
    1      3
1 0      5
    1      7
```

```
[61]: ab.order_vars('B')
```

```
[61]:          pval
      B A
0 0      2
    1      5
```

```
1 0    3
  1    7
```

```
[62]: # !!! Note order_vars returns a pot, but also changes the pot IN PLACE
#      (So if you want to not have this, you should make a copy of the pot first
→-- ab_copy = Pot(ab).order_vars('A'))
# !!! Also note that by default order_vars sorts the points (re-orders var
→values) by default
#      (you can change this with by setting the sort_pts flag to False)
ab
```

```
[62]:      pval
      B A
0 0    2
  1    5
1 0    3
  1    7
```

```
[63]: Pot(ab).order_vars('A', sort_pts=True)
```

```
[63]:      pval
      A B
0 0    2
  1    3
1 0    5
  1    7
```

```
[64]: Pot(ab).order_vars('A', sort_pts=False)
```

```
[64]:      pval
      A B
0 0    2
  1    5
0 1    3
  1    7
```

```
[ ]:
```

3.0.5 Misc

```
[65]: abc
```

```
[65]:      pval
      A B C
0 0 0    2
  1    3
  1 0    5
```

```

      1      7
1 0 0      11
      1      13
      1 0      17
      1      19

```

```
[66]: (abc/'A')
```

```
[66]:          pval
      A B C
0 0 0  0.117647
      1  0.176471
      1 0  0.294118
      1  0.411765
1 0 0  0.183333
      1  0.216667
      1 0  0.283333
      1  0.316667

```

```
[67]: (abc/'A')[{'A':1}]
```

```
[67]:          pval
      B C
0 0  0.183333
      1  0.216667
1 0  0.283333
      1  0.316667

```

```
[68]: abc/{'A':1}
```

```
[68]:          pval
      B C
0 0  0.183333
      1  0.216667
1 0  0.283333
      1  0.316667

```

```
[69]: abc/{'A':1, 'B':0}
```

```
[69]:          pval
      C
0  0.458333
1  0.541667

```

```
[33]:
```

```
[ ]:
```