probttr

April 6, 2021

[1]:
```
from probttrtypes import Type, PConstraint, BType, PType, Pred,\
Possibility, MeetType, JoinType, RecType, Fun
from utils import show, show_latex, ttrace, nottrace
from records import Rec
```

1 Judging probabilities

The witness cache in `probttr` is a pair whose first member is a list of objects and whose second member is a list of probabilities (actually probability constraints)

[2]:
```
T = Type()
print(T.witness_cache)
```

```
([], [])
```

[3]:
```
show(T.judge('a',.5))
```

[3]:
```
'0.5'
```

[4]:
```
show(T.witness_cache)
```

[4]:
```
'([a], [0.5])'
```

[5]:
```
show(T.judge('a',.6))
```

[5]:
```
'0.6'
```

[6]:
```
show(T.witness_cache)
```

[6]:
```
'([a], [0.6])'
```

Adding an additional probability argument to `judge()` gives you a minimum and maximum probability constraint.

[7]:
```
show(T.judge('a',.6,1))
```

[7]:
```
'>=0.6'
```

[8]:
```
show_latex(T.judge('a',.6,.8))
```

[8]:

$$\geq 0.6\& \leq 0.8 \tag{1}$$

[9]:
```
show_latex(T.judge('a',0,.6))
```

[9]:

$$\leq 0.6 \tag{2}$$

`judge(a,n,n)` is the same as `judge(a,n)`

[10]:
```
show_latex(T.judge('a',.6,.6))
```

[10]:

$$0.6 \tag{3}$$

Probabilities must be between 0 and 1. This can be checked by using the method `validate()` on a probability constraint.

[11]:
```
print(T.judge('a',-1).validate())
print(T.judge('a',.6,.1).validate())
```

```
-1.0 is less than 0.
False
0.6 is greater than 0.1
False
```

In contrast to non-probabilistic `pyttr` we can store a negative judgement in witness cache.

[12]:
```
T.judge('a',0)
show_latex(T.witness_cache)
```

[12]:

$$\langle [\mathrm{a}], [0.0] \rangle \tag{4}$$

For compatibility with non-probabilistic TTR: `judge(a)` is the same as `judge(a,1)` (which is the same as `judge(a,1,1)`).

[13]:
```
show(T.judge('a'))
```

[13]:
```
'1.0'
```

1.1 Non-specific judging

We can also make judgements about the probability that something belongs to a type using the method `judge_nonspec`.

[14]:
```
T_new = Type()
T_new.judge_nonspec(.3,.4)
show(T_new.prob_nonspec)
```

[14]:
```
'>=0.3&<=0.4'
```

2 Querying probabilities

2.1 Querying unconditional probabilities

If `a` is in the witness cache, `query(a)` returns the probability stored in the witness cache for `a`.

[15]:
```
show(T.query('a'))
```

[15]:
```
'1.0'
```

If a is not in the witness cache and we have no other way of computing a probability that a is a witness for T, then the probability range is [0,1], i.e. ≤ 1. This corresponds to returning an answer Don't know or Undecided.

[16]:
```
show(T.query('b'))
```

[16]:
```
'<=1.0'
```

Don't know results are not added to the witness cache. (This may or may not be a good idea.)

[17]:
```
show(T.witness_cache)
```

[17]:
```
'([a], [1.0])'
```

Witness conditions are functions which return probability constraints. Here is type Real for real numbers, implemented as floating point decimals where the witness condition gives a categorical judgement for any object, that is it returns probability 1 or 0.

[18]:
```
def RealClassifier(n):
    if isinstance(n,float):
        return PConstraint(1)
    else:
        return PConstraint(0)
Real = Type('Real')
Real.learn_witness_condition(RealClassifier)
show(Real.query(.6))
```

[18]:
```
'1.0'
```

[19]:
```
show(Real.query('a'))
```

[19]:
```
'0.0'
```

[20]:
```
show(Real.witness_cache)
```

[20]:
```
'([0.6, a], [1.0, 0.0])'
```

Here's an example with a couple of witness conditions which return probability constraints. query(a) returns the maximum obtained by the witness conditions for a. Note that this need not be identical with either of the constraints returned by the individual witness conditions since the maximum of a collection of probability constraints is defined has as minimum the maximum of all the minima and as maximum the maximum of all the maxima. (It could, of course, be done differently...)

[21]:
```
T_at = Type('T_at')
def Classifier_a(s):
    if 'a' in s:
        return PConstraint(.8,.9)
    else:
        return PConstraint(.1,.3)
def Classifier_t(s):
    if 't' in s:
```

```
        return PConstraint(.2,.95)
    else:
        return PConstraint(.15,.7)
T_at.learn_witness_condition(Classifier_a)
T_at.learn_witness_condition(Classifier_t)
show(T_at.query('a'))
```

[21]: `'>=0.8&<=0.9'`

[22]:
```
show(T_at.query('t'))
```

[22]: `'>=0.2&<=0.95'`

[23]:
```
show(T_at.query('at'))
```

[23]: `'>=0.8&<=0.95'`

[24]:
```
show(T_at.query('b'))
```

[24]: `'>=0.15&<=0.7'`

2.2 Querying conditional probabilities

Conditions are provided as a second argument to the `query()` method as a list each of whose members is *either* a tuple, `(a,T)`, where `a` is an object and `T` is a type, *or* a type. The idea is that, for example, `T.query(a,[(b,T1),T2])` queries the probability that `a` is of type `T` given that `b` is of type `T1` and there is some witness for `T2`.

Consider a query `T.query(a,Conds)`. If the probability that `a` is of type `T` does not depend on any of the conditions in `Conds` then what is returned is the same as `T.query(a)`, that is the unconditional probability. The default assumption is that probabilities are independent.

[25]:
```
T1 = Type()
T2 = Type()
T1.judge('a',.6)
show(T1.query('a',[('b',T2)]))
```

[25]: `'0.6'`

One kind of dependency between probabilities relates to subtyping in the type theory. Suppose that $T_2 \sqsubseteq T_1$. Then $p(a : T_1 | a : T_2) = 1$. A limit case of this is where we have the same type judgement in the conditions as the one we are querying: $p(a : T | a : T)$, that is, the probability that a is of type T given that a is of type T has to be 1, no matter what the unconditional probability is that a is of type T.

[26]:
```
show(T1.query('a',[('a',T1)]))
```

[26]: `'1.0'`

Let us now create a type `T3` which is a subtype of `T1`

[27]:
```
T3 = Type()
T1.learn_witness_condition(lambda x: T3.query(x))
T3.subtype_of(T1)
```

[27]: `True`

What is the probability that something is of type `T1` given that it is of type `T3`?

[28]:
```
show(T1.query('b',[('b',T3)]))
```

[28]:
```
'1.0'
```

Dependent probabilities that are not related to subtyping have to be provided by an oracle which is given as a third argument to `query()`. An oracle is a python function which takes three arguments: an object, a type and a list of conditions of the kind which is used as an argument to `query()`. For any such argument it returns either a probability constraint (e.g. `PConstraint(.6)`) or `None`. As a python function the oracle may call on resources external to `pyttr`, for example, conditional probability tables or Bayesian networks. A call `T.query(a,c,o)` where `c` does not contain `(a,TT)` where `TT` is a subtype of `T` will return `o(a,T,c)` if this is not `None`. Otherwise `T.query(a,c,o)` will return `T.query(a)` (the unconditional probability). If `c` *does* contain `(a,TT)` where `TT` is a subtype of `T`, then the oracle will be ignored and `T.query(a,c,o)` will return `PConstraint(1)`.

As an example we define a trivial oracle.

[29]:
```
def SillyOracle(a,T,c):
    if a is 'a'and T is T1 and ('b',T2) in c:
        return PConstraint(.7,.8)
    else:
        return
```

Using the oracle.

[30]:
```
show(T1.query('a',[('b',T2)],SillyOracle))
```

[30]:
```
'>=0.7&<=0.8'
```

The oracle is not defined (returns `None`) and the result is the unconditional probability.

[31]:
```
show(T1.query('a',[T2],SillyOracle))
```

[31]:
```
'0.6'
```

The oracle is defined but is ignored because of the subtyping condition.

[32]:
```
show(T1.query('a',[('b',T2),('a',T3)],SillyOracle))
```

[32]:
```
'1.0'
```

The probability that *a* is to the left of *b* is the same as the probability the *b* is to the right of *a*.

[33]:
```
Ind = BType('Ind')
Ind.judge('a')
Ind.judge('b')
left = Pred('left',[Ind,Ind])
right = Pred('right',[Ind,Ind])
left.learn_witness_fun(lambda args: PType(right,[args[1],args[0]]))
right.learn_witness_fun(lambda args: PType(left,[args[1],args[0]]))
left_a_b = PType(left,['a','b'])
right_b_a = PType(right,['b','a'])
M = Possibility('M')
right_b_a.in_poss(M).judge('s1',.6)
left_a_b.in_poss(M).judge('s2',.7)
```

```
print(show(M))
print(show(left_a_b.in_poss(M).query('s1')))
print(show(right_b_a.in_poss(M).query('s2')))
print(show(M))
```

```
M:
-------------------------------------------
right(b, a): [(s1, 0.6)]
left(a, b): [(s2, 0.7)]
-------------------------------------------

0.6
0.7

M:
-------------------------------------------
right(b, a): [(s1, 0.6), (s2, 0.7)]
left(a, b): [(s2, 0.7), (s1, 0.6)]
-------------------------------------------
```

What happens if we revise our original judgement? This shows the downside of caching the result of an inference in a dynamic environment where a premise for the inference has changed since we last checked. Below we rejudge `left(a,b)` to have probability `.3` in `s2`. However, when we query `right(b,a)` in `s2`, we still have the old value we found with the earlier judgement for `left(a,b)`.

[34]:
```
left_a_b.in_poss(M).judge('s2',.3)
print(show(M))
print(show(right_b_a.in_poss(M).query('s2')))
print(show(M))
```

```
M:
-------------------------------------------
right(b, a): [(s1, 0.6), (s2, 0.7)]
left(a, b): [(s2, 0.3), (s1, 0.6)]
-------------------------------------------

0.7

M:
-------------------------------------------
right(b, a): [(s1, 0.6), (s2, 0.7)]
left(a, b): [(s2, 0.3), (s1, 0.6)]
-------------------------------------------
```

This is clearly the wrong result for the new circumstances. A way around this is to `forget()` the probability for `s2` before calling `query()`. Of course, knowing what needs to be forgotten is a delicate problem related to work on belief revision. A good strategy is perhaps to always forget before querying when there is any possibility that something relevant might have changed. `forget()` returns the old probability it found just in case you want to save it and reinstate it later if you are unable to compute a new probability.

[35]:
```
print(show(right_b_a.in_poss(M).forget('s2')))
print(show(M))
print(show(right_b_a.in_poss(M).query('s2')))
print(show(M))
```

```
0.7

M:
-------------------------------------------
right(b, a): [(s1, 0.6)]
left(a, b): [(s2, 0.3), (s1, 0.6)]
-------------------------------------------

0.3

M:
-------------------------------------------
right(b, a): [(s1, 0.6), (s2, 0.3)]
left(a, b): [(s2, 0.3), (s1, 0.6)]
-------------------------------------------
```

2.3 Non-specific querying

The query `T.query_nonspec()` asks for the probability that there is something of type `T`, $p(T)$ in the notation of probabilistic TTR. If something has been judged to be of type `T` with probability 1, then `T.query_nonspec()` returns probability 1.

[36]:
```
show(Ind.query_nonspec())
```

[36]:
```
'1.0'
```

Alternatively, if there is nothing in the witness cache but we have made a non-specific judgement using `judge_nonspec`, then the result of that non-specific judgement is returned.

[37]:
```
show(T_new.query_nonspec())
```

[37]:
```
'>=0.3&<=0.4'
```

If no non-specific judgement has been made, then what is returned is the disjunctive probability of all the probabilities in the witness cache.

[38]:
```
print(show(M))
print(show(right_b_a.in_poss(M).query_nonspec()))
print(show(left_a_b.in_poss(M).query_nonspec()))
```

```
print(show(M))
```

```
M:
-------------------------------------------
right(b, a): [(s1, 0.6), (s2, 0.3)]
left(a, b): [(s2, 0.3), (s1, 0.6)]
-------------------------------------------

0.72
0.72

M:
-------------------------------------------
right(b, a): [(s1, 0.6), (s2, 0.3)]
left(a, b): [(s2, 0.3), (s1, 0.6)]
-------------------------------------------
```

If a non-specific judgement has been made and there is a non-empty witness cache, then what is returned is the maximum of the non-specific judgement and the disjunctive probability of the probabilities in the witness cache. Suppose, for example, that I make a judgement that there is .5 probability that there is something of type `Ind`, contrary to what is represented in the witness cache which shows objects with probability 1. Then the result from the witness cache takes precedence. If on the other hand I have made a judgement that there is certain likelihood of something being of a certain type and this exceeds the evidence represented in the witness cache, then the non-specific judgement is returned.

[39]:
```
Ind.judge_nonspec(.5)
print(show(Ind.query_nonspec()))
T_new.judge('a',.1)
print(show(T_new.query_nonspec()))
```

```
1.0
>=0.3&<=0.4
```

Non-specific queries can also be conditional. Here we show the probability that there is some situation in the type `right(b,a)` (i.e. that it is true that `b` is to the right of `a`) given that `s3` is a witness of `left(a,b)`.

[40]:
```
print(show(right_b_a.in_poss(M).query_nonspec([('s3',left_a_b.in_poss(M))])))
print(show(M))
```

```
1.0

M:
-------------------------------------------
right(b, a): [(s1, 0.6), (s2, 0.3)]
left(a, b): [(s2, 0.3), (s1, 0.6)]
```

```
-------------------------------------------
```

Below we show the probability that there is some witness for the type `right(b,a)`, given that there is some witness for the type `left(a,b)`.

[41]:
```
print(show(right_b_a.in_poss(M).query_nonspec([left_a_b.in_poss(M)])))
print(show(M))
```

```
1.0

M:
-------------------------------------------
right(b, a): [(s1, 0.6), (s2, 0.3)]
left(a, b): [(s2, 0.3), (s1, 0.6)]
-------------------------------------------
```

If the type in the conditions is not a subtype of the type being queried (in the case below because it refers to a different model), then what is returned is the unconditional probability, unless we provide a relevant oracle.

[42]:
```
M1 = Possibility('M1')
print(show(right_b_a.in_poss(M).query_nonspec([left_a_b.in_poss(M1)])))
```

```
0.72
```

2.4 Meet types

If we judge that the probability of `a` being of type `MeetType(T1,T2)` is `1`, then we also judge the probability of `a` being of `T1` and the probability of `a` being of `T2` to be `1`.

[43]:
```
Tleft = Type()
Tright = Type()
Tm = MeetType(Tleft,Tright)
Tm.judge('a')
print(show(Tleft.query('a')))
print(show(Tright.query('a')))
```

```
1.0
1.0
```

Otherwise, we do not currently draw any conclusions about the probabilities for the component types.

[44]:
```
Tleft1 = Type()
Tright1 = Type()
Tm1 = MeetType(Tleft1,Tright1)
Tm1.judge('a',.6,.8)
print(show(Tleft1.query('a')))
```

```
print(show(Tright1.query('a')))
print(show(Tm1.query('a')))
```

```
<=1.0
<=1.0
>=0.6&<=0.8
```

The user may wish to decide that the `judge` method is not to be used with meet types, only `query`, that is, that the `judge` method is restricted to basic types. However, making judgements about join types may be useful. See below.

Similar remarks hold for non-specific judgements.

[45]:
```
Tleft2 = Type()
Tright2 = Type()
Tm2 = MeetType(Tleft2,Tright2)
Tm2.judge_nonspec()
print(show(Tleft2.query_nonspec()))
print(show(Tright2.query_nonspec()))
Tleft3 = Type()
Tright3 = Type()
Tm3 = MeetType(Tleft3,Tright3)
Tm3.judge_nonspec(.6,.8)
print(show(Tleft3.query_nonspec()))
print(show(Tright3.query_nonspec()))
print(show(Tm3.query_nonspec()))
```

```
1.0
1.0
<=1.0
<=1.0
>=0.6&<=0.8
```

If an object is not in the witness cache of a meet type then the conjunctive probability of the values returned for the two components is returned.

[46]:
```
Tleft3.judge('a',.6)
print(show(Tright3.query('a')))
show(Tm3.query('a'))
```

```
<=1.0
```

[46]:
```
'<=0.6'
```

If an object is in the witness cache then the probability stored there will be returned, even though there may be conflicting evidence in the two components. In order to get the new value we need to `forget()`

[47]:
```
Tright3.judge('a',.3)
print(show(Tm3.query('a')))
Tm3.forget('a')
```

```
print(show(Tm3.query('a')))
```

```
<=0.6
0.18
```

The computation of conjunctive probability uses an adaptation of the Kolmogorov formula for conjunction: $p(a : T_1 \wedge T_2) = p(a : T_1)p(a : T_2 \mid a : T_1)$, as given in Cooper et al. (2015). This involves a conditional probability and therefore in the implementation and oracle argument creating a dependence between the two types will make a difference to the outcome when querying a meet type.

[48]:
```
def Oracle1(a,T,c):
    if a is'a'and T is Tright3 and ('a',Tleft3) in c:
        return PConstraint(.7,.8)
def Oracle2(a,T,c):
    if a is'a'and T is Tright3 and ('a',Tleft3) in c:
        return PConstraint(0)
Tm3.forget('a')
print(show(Tm3.query('a',oracle=Oracle1)))
Tm3.forget('a')
print(show(Tm3.query('a',oracle=Oracle2)))
```

```
>=0.42&<=0.48
0.0
```

Conditional probabilities can also be queried for meet types.

[49]:
```
Tm3.forget('a')
print(show(Tm3.query('a',[('a',Tleft3),('a',Tright3)])))
```

```
1.0
```

The above example shows the need for witness conditions which pass the conditions and oracle arguments to components of the type. The witness condition for meet types is, schematically, `lambda a,c,oracle: ConjProb([(a,<left>),(a,<right>)],c,oracle)`. (As in non-probabilistic TTR, meet types and other "logical types" cannot learn new witness conditions.) Note that this witness condition has three arguments so that it can pass the conditions, `c`, and the oracle to the function `ConjProb` which computes the conjunctive probability. Witness conditions in the `probttr` implementation can have one to three arguments and will be applied to the arguments provided to the `query` method from which they are called: the object, `a`, being queried for one argument, `a` and the conditions, `c`, for two arguments and `a`, `c` and the oracle for three arguments.

2.5 Join types

Join types work like meet types *mutatis mutandis*. Schematically, the witness condition is `lambda a,c,oracle: DisjProb([(a,<left>),(a,<right>)],c,oracle)` and no new witness conditions can be learnt. If we judge something to have 0 probability of being of a join type, then we judge it to have 0 probability of being of the two component types.

[50]:
```
Tleftd = Type()
Trightd = Type()
Tmd = JoinType(Tleftd,Trightd)
Tmd.judge('a',0)
print(show(Tleftd.query('a')))
print(show(Trightd.query('a')))
```

```
0.0
0.0
```

Similarly for non-specific judgements.

[51]:
```
Tleft2d = Type()
Tright2d = Type()
Tm2d = JoinType(Tleft2d,Tright2d)
Tm2d.judge_nonspec(0)
print(show(Tleft2d.query_nonspec()))
print(show(Tright2d.query_nonspec()))
Tleft3d = Type()
Tright3d = Type()
Tm3d = JoinType(Tleft3,Tright3)
Tm3d.judge_nonspec(.6,.8)
print(show(Tleft3d.query_nonspec()))
print(show(Tright3d.query_nonspec()))
print(show(Tm3d.query_nonspec()))
```

```
0.0
0.0
<=1.0
<=1.0
>=0.6&<=0.8
```

2.6 Record types

The probability that a record, r, is of a record type, T, is the conjunctive probability of the probabilities that the objects in the fields of r are of the types in the correspondingly labelled types of T. If there is a label in T which is not in r then the probability that r is of type T is 0 (also if r is not a record at all then the probability is 0).

A simple non-dependent record type.

[52]:
```
Tf1 = Type()
Tf2 = Type()
Tr1 = RecType({'l1':Tf1,'l2':Tf2})
Tf1.judge('a',.3)
Tf2.judge('b',.2)
r1 = Rec({'l1':'a','l2':'b'})
print(show(Tr1.query(r1)))
```

```
0.06
```

A simple dependent record type.

[53]:
```
dog = Pred('dog',[Ind])
a_dog = RecType({'x':Ind,'e':(Fun('v', Ind, PType(dog,['v'])),['x'])})
show_latex(a_dog)
```

[53]:

$$\left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{e} & : & \langle \lambda v : \mathit{Ind} \,.\, \mathrm{dog}(v), \langle \mathrm{x} \rangle \rangle \end{array}\right] \tag{5}$$

[54]:
```
Ind.judge('d')
PType(dog,['d']).judge('s1',.7)
r2 = Rec({'x':'d','e':'s1','z':'other_stuff'})
show(a_dog.query(r2))
```

[54]:
```
'0.7'
```

A simple record type with a path leading to another record type

[55]:
```
Tr2 = RecType({'x':a_dog})
show_latex(Tr2)
```

[55]:

$$\left[\begin{array}{lcl} \mathrm{x} & : & \left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{e} & : & \langle \lambda v : \mathit{Ind} \,.\, \mathrm{dog}(v), \langle \mathrm{x} \rangle \rangle \end{array}\right] \end{array}\right] \tag{6}$$

[56]:
```
r3 = Rec({'x':r2})
show(Tr2.query(r3))
```

[56]:
```
'0.7'
```

A slightly more complex record type.

[57]:
```
bark = Pred('bark',[Ind])
a_dog_bark = RecType({'x':a_dog,'e':(Fun('v',Ind,PType(bark,['v'])),['x.x'])})
show_latex(a_dog_bark)
```

[57]:

$$\left[\begin{array}{lcl} \mathrm{x} & : & \left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{e} & : & \langle \lambda v : \mathit{Ind} \,.\, \mathrm{dog}(v), \langle \mathrm{x} \rangle \rangle \end{array}\right] \\ \mathrm{e} & : & \langle \lambda v : \mathit{Ind} \,.\, \mathrm{bark}(v), \langle \mathrm{x.x} \rangle \rangle \end{array}\right] \tag{7}$$

[58]:
```
PType(bark,['d']).judge('s2',.3)
r4 = Rec({'x':r2,'e':'s2'})
show_latex(r4)
```

[58]:

$$\left[\begin{array}{lcl} \mathrm{x} & = & \left[\begin{array}{lcl} \mathrm{x} & = & \mathrm{d} \\ \mathrm{e} & = & \mathrm{s}_1 \\ \mathrm{z} & = & \mathrm{other}_{\mathrm{stuff}} \end{array}\right] \\ \mathrm{e} & = & \mathrm{s}_2 \end{array}\right] \tag{8}$$

[59]:
```
show(a_dog_bark.query(r4))
```

[59]:
```
'0.21'
```

Conditional probabilities.

[60]:
```
show(a_dog_bark.query(r4,[('s2',PType(bark,['d']))]))
```

[60]:
```
'0.7'
```

[61]:
```
show(a_dog_bark.query(r4,[('s1',PType(dog,['d'])),('s2',PType(bark,['d']))]))
```

[61]:
```
'1.0'
```