Efficient Inference for Pragmatic Generative Concepts

Jessica Taylor

Probabilistic models for social cognition, including language pragmatics, often require reasoning about reasoning. In the Church programming language, this can be modeled as a query expression inside a query expression, as described in *Reasoning About Reasoning*.

The most common inference algorithm for Church, a Metropolis Hastings MCMC algorithm, does not perform well on these nested queries. *Reasoning About Reasoning* describes a dynamic programming algorithm for evaluating Church programs containing nested query expressions. However, this algorithm relies on the queries having a relatively small number of possible return values.

As a result, one problem that the dynamic programming algorithm is inappropriate for is social cognition involving generative concepts. Generative concepts are concepts that are described as a probability distribution. *Learning Structured Generative Concepts* describes generative concepts for tree structures. Each generative concept specifies a probability distribution over an infinite number of trees and is represented as a probabilistic program.

Pragmatics models can be transformed to use generative concepts. In this pragmatics model, each speaker has a concept in mind and will demonstrate it to a listener. They select a tree, and the listener classifies it as one concept or another. At level 1, the speaker just samples a tree for the concept they want to demonstrate. At higher levels, the speaker is more likely to select trees that will be recognized by the listener. The listener at some level will perform Bayesian inference to determine the concept the speaker at their level was communicating through a certain tree. The model can be expressed in the following Church program:

```
(define (speaker concept depth)
  (if (= depth 1)
    (generate-tree concept)
    (query
      (define tree
        (generate-tree concept))
      (equal? concept
        (listener tree
           (- depth 1))))))

(define (listener tree depth)
  (query
    (define concept (uniform))
    (equal? tree
      (speaker concept depth)))
```

Here I present 2 different approaches to performing inference in this model in particular, and social cognitive models in general. Observe that, if the listener function can be evaluated quickly, then it is easy to evaluate speaker by rejection sampling. As long as the number of possible concepts is small, the probability that the listener will classify a new tree from a concept as the correct concept will not be unreasonably small, and so rejection sampling will be efficient.

Machine Learning For Inference

Now, the question is how to evaluate the listener function efficiently. One approach is to use machine learning. For some depth value, if we have many examples of both a concept and `(speaker concept depth)`, we can build a classifier that maps each tree to a probability distribution over concepts. The only remaining requirement is to get training data. If we can evaluate the listener function at the previous depth level efficiently, we can use this to get examples of `(speaker concept depth)` by rejection sampling. The next step is to use these examples of trees as training data for a machine learning algorithm. A nonparametric method such as a kernel-based density estimate could be appropriate.

Once the learner for one depth has been trained, it will be efficient to run the listener function at that depth. The speaker at the next depth can then run efficiently, and provide training data for the listener at the next level. This method for evaluating nested queries is significant in that it resembles the way humans learn language in society. People learn how to listen based on training data provided by speakers. Speakers, knowing how listeners will interpret what they say, will determine what they say depending on how they expect listeners to interpret it.

This technique should be applicable to a variety of social cognition inference problems. Specifically, if the program can be written so that all queries appear in functions of the form:

```
(define (querying-function
         observation)
  (query
   (define hypothesis
     (get-hypothesis))
   (get-result hypothesis)
   (equal? observation
    (get-observation hypothesis))
```

where `get-hypothesis`, `get-result`, and `get-observation` are all allowed to call previously defined querying functions, then machine learning can be applied in order to train learners for each querying function. The algorithm can be found on the next page.

```
For each querying function in order:
    Initialize a learner for the function.
    Until some termination condition:
        Let the hypothesis be the result of calling (get-hypothesis).
        Let the observation be the result of calling (get-observation hypothesis).
        Let the result be the result of calling (get-result hypothesis).
        Add (observation, result) to the learner's training data.
    Replace the function with the following:
      (define (querying-function observation)
        (predict-result-from-observation learner observation))
```

Where predict-result-from-observation uses the learner to sample from the posterior distribution over the result given the observation. If depth is limited, it should be straightforward to expand the original example to take this form for a, eliminating the depth argument by creating a separate version of each function for each depth. Since each training phase will only call previously defined querying functions, which have already been replaced with functions that call predict-result-from-observation, it will never be necessary to actually evaluate a nested query. When the algorithm has completed, each querying function has been replaced with a call to predict-result-from-observation, which should terminate in a reasonable time.

As the number of iterations per querying function increases, so should the accuracy of each learner. As long as each individual learner converges to the correct posterior, so should each querying function.

There should be nothing wrong with replacing some of the queries with rejection queries. In general, rejection queries will be more accurate than queries replaced by machine learning, because they exactly satisfy the definition of query, but they will most likely be slower. This slowdown will be acceptable if the probability of acceptance is not too low, and each individual proposal does not take too long to evaluate (for example, because it calls a querying function that has been implemented with machine learning).

Machine learning methods to perform inference in nested queries are always limited by the quality of the machine learning algorithms used. For example, in the original problem, we must be able to build a probabilistic classifier for tree data structures that will learn from examples of trees. This is not an easy problem. It would be preferable to have an analytic solution that does not heavily rely on machine learning.

Factor Graphs for Social Cognition

For this problem, we can mathematically define the model. p_n denotes the distribution over concepts and trees at level n; $p_n(t|c)$ will be the distribution for the speaker at level n, and $p_n(c|t)$ will be the distribution for the listener at level n.

$$p_n(c)=p(c)=uniform$$
$$p_1(t|c)=given$$

For n > 1:

$$p_n(t|c)=\frac{p_1(t|c)\,p_{n-1}(c|t)}{\sum_{t'} p_1(t'|c)\,p_{n-1}(c|t')}$$
$$=\frac{p_1(t|c)\,p_{n-1}(c|t)}{E_{p_1(t'|c)}[p_{n-1}(c|t')]}$$
$$p_n(c|t)\;\;\alpha\;\;p(c)\,p_n(t|c)=\frac{p_1(t|c)\,p_{n-1}(c|t)}{E_{p_1(t'|c)}[p_{n-1}(c|t')]}$$
$$\alpha\;\;\frac{p_1(c|t)\,p_{n-1}(c|t)}{E_{p_1(t'|c)}[p_{n-1}(c|t')]}$$

From these definitions it follows by induction that:

$$p_n(c|t)\;\;\alpha\;\;\frac{p_1(c|t)^n}{\prod_{i=1}^{n-1} E_{p_1(t'|c)}[p_i(c|t')]}$$

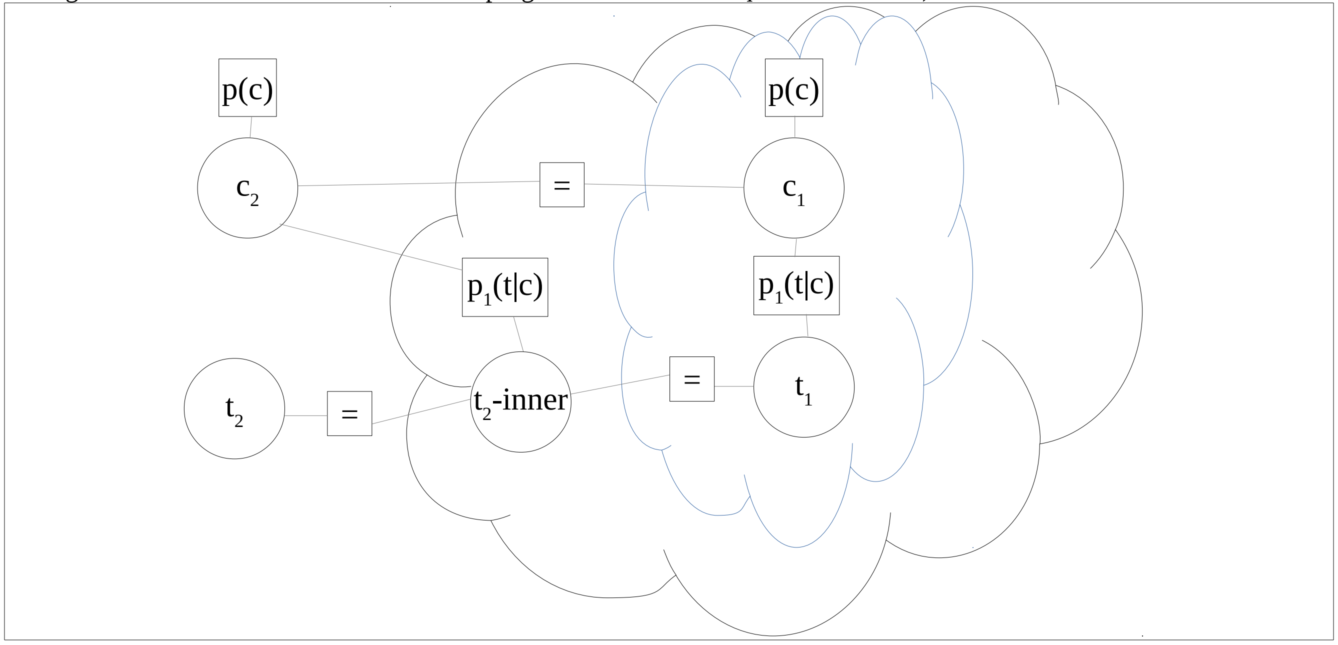
The expected values in the denominator should be simple to approximate. Each is the probability that the output of the level 1 speaker for the concept will be categorized correctly by the learner at level i. These probabilities can be approximated by Monte Carlo sampling. Once the listener's probability distribution of concept given tree can be computed, it is possible to define the speaker at the next level by rejection sampling.

It would be convenient if this technique could be extended to other social cognition models. In fact, it can. Let us extend factor graphs with additional notation. A cloud around some subset of the variables and factors denotes a nested query. Variables and factors within a cloud can be connected to variables and factors outside the cloud. However, variables in the cloud are not true variables; they cannot be conditioned on normally, and must instead be connected to outside variables that are conditioned on.

We can represent the pragmatics model at level 2 as the following model:

```
(define c2 (uniform))
(define t2
 (query
  (define t2-inner
    (generate-tree c2))
  t2-inner
  (equal? c2
   (query
    (define c1 (uniform))
    (define t1
     (generate-tree c1))
   c1
   (equal? t1 t2-inner)))))
```

We can query t2 given c2 for the speaker at level 2, or c2 given t2 for the listener at level 2. The program can be represented using the following factor graph, where p(c) denotes a uniform distribution over concepts, $p_1(t|c)$ denotes the model for generating a tree from a concept, and = denotes an equality factor (1 if all the variables in the factor are equal, 0 otherwise).



Since the program contains 2 nested queries, it is necessary to have 2 clouds, one for each query expression. Note that the factors inside a cloud are exactly those that appear in that query. For example, because `(equal? c2 (query …))` appears within the first query, the corresponding equality factor (the top one) appears within the first cloud.

I propose a rule for eliminating clouds, which serves as a definition of the cloud notation. To eliminate a cloud, create an additional factor connected to all the variables outside the cloud that are connected to factors in the cloud. The value of this factor is defined to be equal to the reciprocal of the sum, over all variables in the cloud, of the product of factors within the cloud. This factor is placed in the same context as the cloud (for example, if the cloud is in another cloud, this factor goes in the outer cloud). Finally, the cloud is removed and all variables and factors in the cloud are left intact.

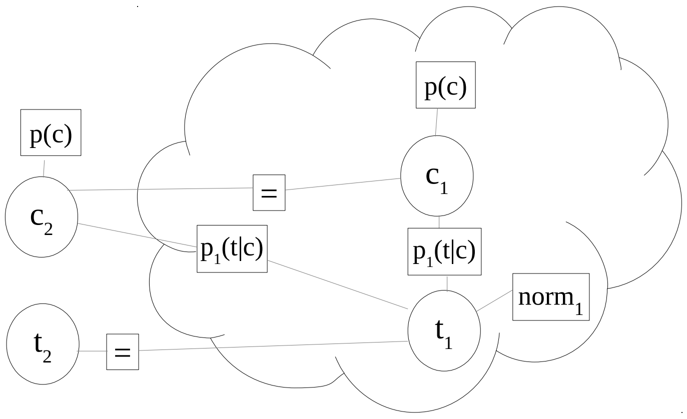
The additional factor created to eliminate the cloud is the reciprocal of the normalizing constant for that query. This normalizing constant can depend on variables connected to factors within the cloud, but does not depend on variables within the cloud or any other variables.

Let us perform the procedure on the factor graph with clouds. First, observe that the only factor in the inner cloud connected to a variable outside the inner cloud is the equality factor, which is connected to $t_{2\text{-inner}}$. Therefore, the normalizing factor will be a function of t_2-inner. Its value will equal:

$$norm_1(t_{2-inner}) = \frac{1}{\sum_{c_1}\sum_{t_1} p(c_1)\, p_1(t_1|c_1)\, 1\{t_1 = t_{2-inner}\}}$$

$$= \frac{1}{\sum_{c_1} p(c_1)\, p_1(t_{2-inner}|c_1)} = \frac{1}{p_1(t_{2-inner})}$$

After we combine $t_{2\text{-inner}}$ with t_1, dropping the equality factor, the graph will look like this:

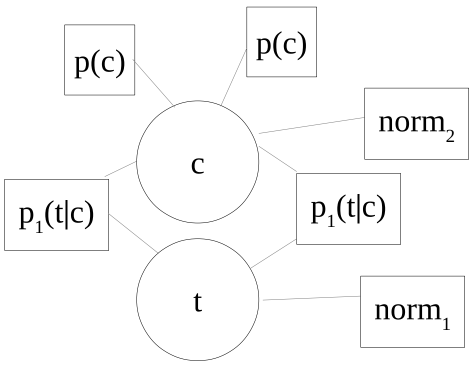


For the outer cloud, the only outer variable connected to factors in the graph is c_1. The normalizing constant for this cloud will be:

$$norm_2(c_2) = \frac{1}{\sum_{c_1} \sum_{t_1} 1\{c_1 = c_2\}\, p(t_1|c_2)\, p(c_1)\, p_1(t_1|c_1) \frac{1}{p_1(t_1)}}$$

$$= \frac{1}{\sum_{t_1} p_1(t_1|c_2)\, p_1(c_2|t_1)} = \frac{1}{E_{p_1(t_1|c_1)}[p_1(c_2|t_1)]}$$

The final graph becomes:



It is easy to check that this factor graph represents the distribution over c and t at level 2, according to the earlier mathematical analysis. If t is known, then the $norm_1$ factor is irrelevant to the distribution of c given t, so it can be ignored. Because there is a small number of possible c values, it is possible to find the relative probability of each individual concept.

On the other hand, if c is known, then inference can proceed by repeatedly sampling from $p_1(t|c)$ and assigning importance or Metropolis-Hastings weight $p_1(t|c)norm_1(t)$. This turns out to be quite similar to the rejection sampler that samples from $p_1(t|c)$ and accepts with probability $p_1(c|t)$, because $p_1(c|t)$ is proportional to $p_1(t|c)norm_1(t)$.

This approach to inference requires the normalizing factors to be efficiently computable. Here, $norm_1$ can be computed by summing $p(c)p_1(t|c)$ for each concept, and $norm_2$ can be computed by Monte Carlo approximation. The process of replacing clouds with normalizing factors can be automated, but efficiently computing the normalizing factors depends on the specific model.

Implementation

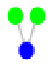
I implemented this algorithm for the pragmatic generative concept model. Each concept has associated with it a distribution over prototypes. The true distribution over trees is obtained by sampling a random prototype, and then mutating it by some rules. Mutation can produce any tree with some probability but is more likely to produce similar trees. The only difficult part of the inference is computing $p_1(t|c)$ for an arbitrary tree and concept. To compute this, I created a list of weighted prototypes for each concept, and compared the tree to each prototype for the concept to determine the probability that that prototype would mutate into the tree. Taking the weighted average of these mutation probabilities yields the $p_1(t|c)$. For more complex concepts, it would be necessary to use an algorithm that computes a normalizing constant, such as adaptive importance sampling.

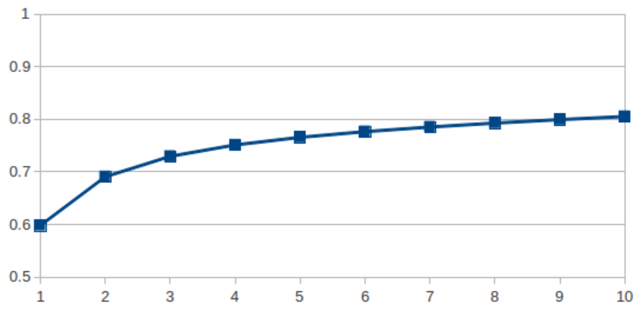
I tested the implementation on 2 types of trees: “fens” and “bargs”. Both can be described by the following generative model:

```
(define (generate trunk-color
                  branch-color
                  prob-continue)
  (if (flip prob-continue)
    (list trunk-color
      branch-color
      (generate trunk-color
        branch-color prob-continue)
      branch-color)
     (list trunk-color branch-color
        branch-color)))
```

Fens use trunk and branch colors selected independently from a uniform distribution over blue, green, and peach. Bargs use trunk and branch colors selected independently from a uniform distribution over blue and green. Fens use a prob-continue value of ½, while bargs use 2/3, so they tend to be taller.

Results can be seen on the next page. In general, at higher levels, fens became more and more peach. This makes sense, because peach trees are much more likely to be recognized as fens. The distribution over bargs did not change much. This is probably because, as fens become more and more likely to contain peach, a tree *not* containing peach will be more likely to be classified as a barg, so most bargs have a high probability of being recognized as bargs.

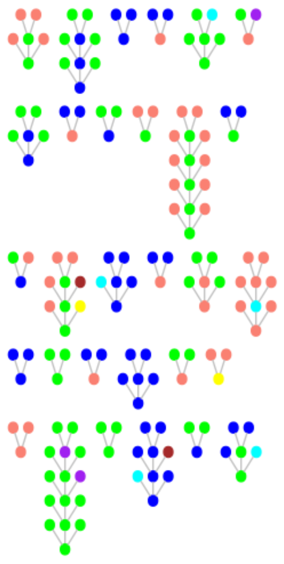
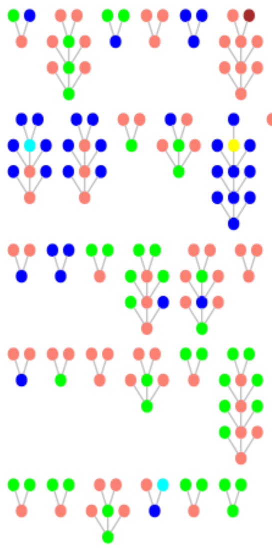
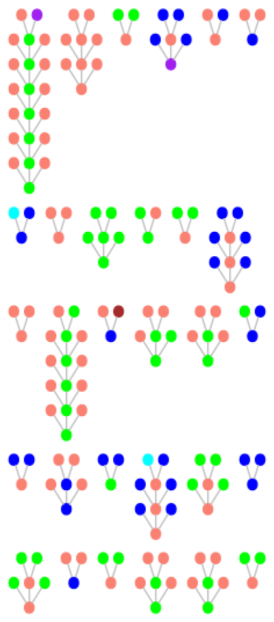
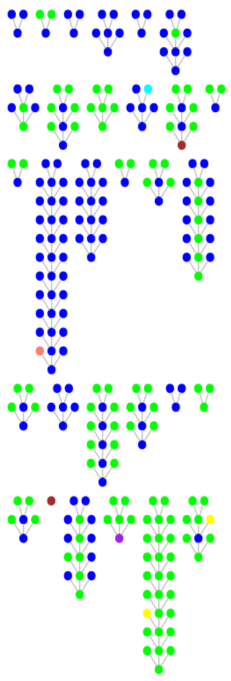
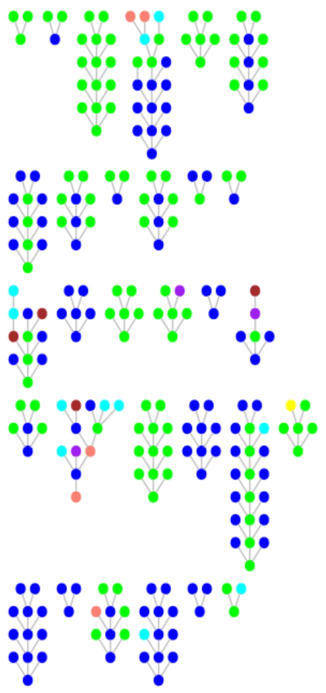
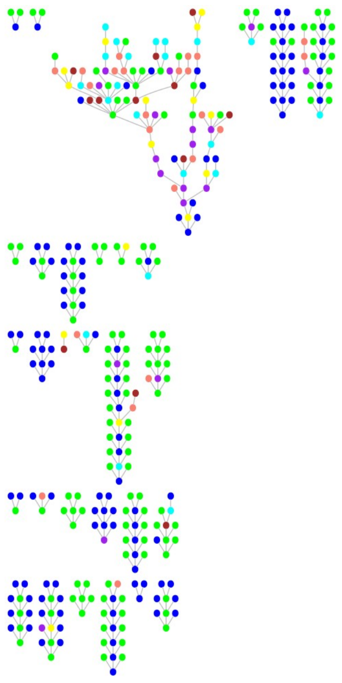
Here is a graph of the probability that the tree  will be classified as a barg over time:



The probability increases over time, as fens become more and more likely to contain peach, but does not converge to 1. Looking at the formula for $p_n(c|t)$, it would be expected that, if $p_1(c|t) > 0.5$, it should increase as the number of iterations increases. However, the product in the bottom is higher for bargs than fens, because fens generated from p_1 are more likely to be rejected by the speaker (due to the lack of peach), and as a result the probability converges to about 0.85.

Results

This table shows the result of evaluating `(speaker concept depth)` 30 times, for 2 different concepts and 3 depths.

	Level 1	Level 2	Level 3
Fens			
Bargs			

References

Stuhlmüller, Andreas and Goodman, Noah. *Reasoning About Reasoning by Nested Conditioning: Modeling Theory of Mind with Probabilistic Programs*. Accessed December 8, 2012 <https://www.stanford.edu/~ngoodman/psych204/restrictedpapers/cogsys-paper-submitted.pdf>

Stuhlmüller, Andreas; Tenenbaum, Joshua; and Goodman, Noah. *Learning Structured Generative Concepts*. Accessed December 8, 2012 <http://www.mit.edu/~ast/papers/structured-generative-concepts-cogsci2010.pdf>