PERFORMING INFERENCE IN THE SPACE OF EXPRESSIONS VIA PROBABILISTIC PROGRAMMING

FRANK WOOD

Introduction

One of the key characteristics of higher-order probabilistic programming languages equiped with `eval` is that program text both can be generated and evaluated.

In higher-order languages (Lisp, Scheme, Church, Anglican and Venture) functions are first class objects – evaluating program text that defines a valid procedure returns a procedure that can be applied to arguments.

In the context of probabilistic programming this means that we can write programs that program. We can make the computer write its own code.

In the following we set up what is essentially a regression problem and ask the computer to find a solution in the space of arithmetic *expressions* that could give rise to the observed relationship. Note very carefully that this approach stands in sharp contrast to picking a function family and looking for a parameterization of a member of this family that is optimal with respect to some cost function and observed input output values.

The following program generates simple arithmetic expression code using a probabilistic context-fee grammar (PCFG) Johnson [1998] generative model (`productions`) and then evaluates those expressions on known input values and compares the output to known outputs. It then predicts both the procedure text and the value of the procedure applied to a new argument.

```
 [assume get-int-constant
  (lambda () (uniform-discrete 0 10))]

[assume safe-div
  (lambda (x y) (if (= y 0) 0 (/ x y)))]

[assume productions
  (lambda ()
    (define expression-type (discrete (list 0.40 0.30 0.30)))
      (cond
        ((= expression-type 0) (get-int-constant))
        ((= expression-type 1) 'x)
        (else
          (list
            (nth (list (quote +) (quote -) (quote *) (quote safe-div))
```

```
                (discrete (list 0.25 0.25 0.25 0.25)))
            (productions) (productions)))))]

[assume induced-procedure-code (list 'lambda (list 'x) (productions))]
[assume induced-procedure (eval induced-procedure-code)]

[assume noise 0.00001]
[observe (normal (induced-procedure 1) noise) 5]
[observe (normal (induced-procedure 2) noise) 3]
[observe (normal (induced-procedure 3) noise) 1]

[predict induced-procedure-code]
[predict (induced-procedure 4)]
```

Questions :

(1) Change the generative model to produce longer expressions. Run the program and compare the output of the original program to the output of the program written to generate longer expressions.

(2) Try to learn a more complex relationship. For instance $y = x^3 + 7x - 12$. Generate your own training data. Experiment with varying amounts of training data and the order of data instances. Comment on the effect of adding data and data order in terms of the inference output.

(3) Introduce unary operators like `sin` and `cos` into the generative model. Try to learn a trigonometric functional expression.

Additional material related to this subject can be found in [Goodman et al., 2008, Mansinghka, 2009, Mansinghka et al., 2014].

References

Noah D Goodman, Joshua B Tenenbaum, Jacob Feldman, and Thomas L Griffiths. A rational analysis of rule-based concept learning. *Cognitive Science*, 32(1):108–154, 2008.

Mark Johnson. Pcfg models of linguistic tree representations. *Computational Linguistics*, 24(4):613–632, 1998.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

Vikash Kumar Mansinghka. *Natively probabilistic computation*. PhD thesis, Massachusetts Institute of Technology, 2009.