PERFORMING INFERENCE IN THE SPACE OF EXPRESSIONS VIA PROBABILISTIC PROGRAMMING

FRANK WOOD

Introduction

One of the key characteristics of Turing-complete probabilistic programming languages is that program text can be generated and evaluated.

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

In the context of probabilistic programming, what does this all come to? In short, it means that we can write programs that program. We can make the computer teach itself how to program.

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.

Answer Simply changing the line

```
(define expression-type (discrete (list 0.40 0.30 0.30)))
```

to something like

```
(define expression-type (discrete (list 0.20 0.30 0.50)))
```

dramatically alters the length of the inferred expressions.

Commenting out the lines

```
;[assume noise 0.0000001]
;[observe (normal (induced-procedure 3) noise) 36]
;[observe (normal (induced-procedure 4) noise) 80]
;[observe (normal (induced-procedure 5) noise) 148]
;[observe (normal (induced-procedure 10) noise) 1058]
```

allows you to directly observe the differences in generative models.

(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.

Answer
A program can be used to generate training data

Program 1. Training Data

```
[assume y (lambda (x) (- (+ (pow x 3) (* 7 x)) 12))]
[predict (y 3)]
[predict (y 4)]
[predict (y 5)]
[predict (y 6)]
[predict (y 10)]
```

which can be run with command line `anglican -s <Program 1> -n 1`
to generate

```
(y 3),36.0
(y 4),80.0
(y 5),148.0
(y 6),246.0
(y 10),1058.0
```

Program 2. Cubic

```
 [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 0.01]
[observe (normal (induced-procedure 3) noise) 36]
[observe (normal (induced-procedure 4) noise) 80]
```

```
[observe (normal (induced-procedure 5) noise) 148]
[observe (normal (induced-procedure 10) noise) 1058]

[predict (list induced-procedure-code (induced-procedure 6))]
```

A nice command line for parsing the output of this program is `anglican -s <Program 2> -P 1000 | uniq -c`

Adding `pow` to the list of operators as in the following helps quite a lot

PROGRAM 3. POW

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

[assume safe-div
  (lambda (x y) (if (< y 0.000001) 0 (/ x y)))]

[assume safe-pow
  (lambda (x y) (begin (define z (pow x y)) (if (< z .000001) 0.000001 z)))]

[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 safe-pow) (quote *) (quote saf
                 (discrete (list 0.2 0.2 0.2 0.2 0.2)))
              (productions) (productions))))))]

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

[assume noise 0.0000001]
[observe (normal (induced-procedure 3) noise) 36]
[observe (normal (induced-procedure 4) noise) 80]
[observe (normal (induced-procedure 5) noise) 148]
[observe (normal (induced-procedure 10) noise) 1058]

[predict (list induced-procedure-code (induced-procedure 6))]
```

A nice command line option for running this program is `./anglican -s <Program 3> -P 2000 -m rdb | uniq -c`

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

Answer

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

[assume safe-div
  (lambda (x y) (if (< y 0.000001) 0 (/ x y)))]

[assume productions
  (lambda ()
    (define expression-type (discrete (list 0.30 0.30 0.10 0.30)))
      (cond
        ((= expression-type 0) (get-int-constant))
        ((= expression-type 1) 'x)
        ((= expression-type 2) (list
            (nth (list (quote sin) (quote cos))
                 (discrete (list 0.5 0.5)))
               (productions)))
        (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)]

[predict (list induced-procedure-code (induced-procedure 6))]
```

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.