Overview of current work

Irvin Hwang

This document outlines current work in pattern recognition as inductive programming.

1 Overview

The following is an informal description of current work on a probabilistic approach to inductive programming with Noah Goodman and Andreas Stuhlmüller. The main idea is to view program learning as sampling from a conditional distribution over the space of programs, where one conditions on programs that probably evaluate to the observed data. An easy way to express this inference task is in the probabilistic programming language Church. Church samples from a target distribution using a Markov chain Monte Carlo (MCMC) algorithm. The problem with Church is the default MCMC algorithm is not well suited for program induction so we are working on making modifications to perform smarter sampling.

2 Church

Church is a probabilistic programming language that makes it easy to write probabilistic models and separates specification from implementation of the inference algorithm [?]. Church programs follow the general format

```
(query
  generative-model
  what-we-want-to-know
  what-we-know)
```

Query returns samples from the distribution represented by the generative model. An example being

```
(query
 ;;generative model
 (define A (if (flip) 1 0))
 (define B (if (flip) 1 0))
 (define C (if (flip) 1 0))
 (define D (+ A B C))
 ;;what we want to know (query expression)
 A
 ;;what we know (conditional expression)
 (equal? D 3)))
```

Here our model consists of three coins being flipped and computing the sum. We want to know a distribution over the possible values for `A` conditioned upon the fact that the sum is equal to three. In the case of inductive programming we might write something like the following

```
(query
  ;;generative model
  (define expr (apply (uniform-draw bool if)))
  (define bool (uniform-draw 't 'f))
  (define if (list 'if (expr) (expr) (expr)))
  (define program (expr))
  ;;what we want to know (query expression)
  program
  ;;what we know (conditional expression)
  (and (equal? (eval program) 't)))
```

Here the generative model is a grammar for a simple language that consists of if expressions and booleans, let's call this the if-boolean grammar. We sample from the distribution over programs created by the grammar, conditioned on the program evaluating to true. While this is not an interesting program induction problem, it is relatively straight-forward to see how more interesting problems can be expressed in this manner.

3 Markov Chain Monte Carlo

Church uses the Metropolis-Hastings algorithm (a MCMC method) for sampling [?]. MCMC is a technique for getting samples from a normally difficult to sample from distribution (e.g. the conditional distribution in program induction problems). The method works by starting with a Markov chain, M, over the space the target distribution is defined over. If M satisfies certain properties then it has what is called a stationary distribution i.e. if we do a random walk for long enough the frequency with which we visit states forms a fixed distribution over these states. The stationary distribution of M is generally not the target distribution, but by making a careful modification to the transition function of M we can guarantee in the limit the stationary distribution of the modified Markov chain, M', is the same as our target distribution. Performing a random walk on M' eventually gives us samples from our desired distribution. The modification to the transition function is basically a special function that says whether to accept a proposed transition made by M. When this "acceptance function" satisfies certain properties performing a random walk on M' to get samples from the target distribution is called the Metropolis-Hastings algorithm.

4 Metropolis-Hastings in Church

The main idea for implementing the Metropolis-Hastings algorithm underneath Church is to make the states a record of the random choices (e.g. `uniform-draw` or `flip` in the above examples) made during the execution of the generative model, we call this a program trace (it is easy to recover the value of the query expression and conditioning expression after each execution the way a trace is represented within the system). A proposal transition between states is made by selecting a random choice and resampling from it then letting the effects of changing that value change the execution of the program and recording the changed random choices as a new state. We accept or reject this new choice under the Metropolis-Hastings criteria where rejection occurs if the conditional expression does not hold and/or the transition probabilities and the score of the state is too low (the product of the random choices).

If we think about this in terms of program induction what is basically happening is we start with some generative grammar and somehow initialize our starting trace to produce a program from the grammar that satisfies the desired conditions. So for example in the above if-boolean grammar we might have the random choice, r1, for `expr` in `(define program (expr))` be `if` then for the first expr in `(list 'if (expr) (expr) (expr))` we randomly choose `bool`, r2, and for `bool` we randomly choose `'t`, r3, and so on until we get an expression `(if 't 't 't)`. This satisfies the condition that it evaluates to true and so we can use its trace as an initial state. Now to make a proposal transition from this initial state we select one of the random choices (say r1) and re-sample getting the value `bool` instead of `if`. This drastically changes the program essentially deleting all the old random choices since the calls to `expr` in `(list 'if (expr) (expr) (expr))` are no longer made. Instead we need to make a new random choice for `bool`, which we'll call r4 and say it takes on the value `'t`, so our new program would simply be the expression `'t` instead of `(if 't 't 't)`. Since this program requires fewer random choices it has a higher score than the initial state so we would accept this proposed transition. A less drastic change would have occurred had we chosen r3 to resample from and gotten `'f` then the proposed trace would have result in the expression `(if 'f 't 't)`, note aside from the selected random choice we keep all other non-affected random choices the same. By keeping non-affected random choices the same we are able to search through programs "close" to each other, but we can also make big jumps as was seen when r1 was re-sampled. One way to think of this algorithm is as stochastic local search, but with nice theoretical guarantees that in the limit we will not get stuck in local minima. While this algorithm is guaranteed to work in the limit it is very naive in the sense that it does not really take into account any domain-specific knowledge, e.g. patterns to how functions are created during programming. The goal of the current project is to incorporate some of this knowledge into the algorithm while maintaining the theoretical guarantees of Markov chain Monte Carlo.

5 Abstraction

We use a concept known as least general generalization (lgg) or anti-unification to formalize the notion of abstraction; this formulation of abstraction dates back to [?] and is similar to that presented in [?]. We would like to be able to think of an object in terms of its structure i.e. as a composition of primitives. A grammar is a compact way to specify how primitives can be combined so one way to capture the structure of an object is to represent it as an expression in some language. An example is the set of natural numbers where 1 is the set of primitives, + is the operator for composition, and the grammar is `N -> 1 | (+ N N)` e.g. `2=(+ 1 1)`, `4=(+ (+ 1 1) (+ 1 1))`, `6 = (+ (+ 1 1) (+ 1 (+ 1 (+ 1 1))))` or `(+ (+ 1 (+ 1 1)) (+ 1 (+ 1 1)))`.

An abstraction of a set can be thought of as the structure shared by elements of the set. Using our expression representation, we can define an abstraction for a set of expressions to be a common subexpression with variables that represent where the expressions differ. An abstraction from the set {`2,4,6`} might be `(+ x x)` where x is any natural number. It is worth noting the set defined by the abstraction can be much larger than the set the abstraction came from, e.g. `(+ x x)` represents the set of all even numbers, but the original set {`2,4,6`} is much smaller. A largest common subexpression for a set of expressions is known as a least general generalization (lgg) since the resulting abstraction contains all the expressions it was generalized from, but the lgg is also the most specific abstraction since there is no other abstraction that contains the original set that is also a subset of the lgg. The interesting thing about this notion of abstraction is it provides a natural way to rewrite a program to use modular functions.

6 Abstraction in Church

We would like to integrate abstraction into the basic church MCMC algorithm. The main idea is to essentially have another way to propose new states in addition to selecting a random choice and changing it. The new way of proposing a state is to take the program corresponding to the current trace and then find common subexpressions in this program. One common subexpression is selected and turned into a function then the original program is written in terms of this function. An example might be we start with the program `(+ (+ a b) (+ a b) (+ c d))` and two possible functions from subexpressions are `(define F1 (+ a b))` or `(define (F2 x y) (+ x y)`. The original program can then be written as `(+ F1 F1 F1)` or `(+ (F2 a b) (F2 a b) (F2 c d))`. The method used for finding common subexpressions is quite flexible and allows for higher-order functions and finding abstractions from abstractions. Here are a few examples:

```
(+ (+ a b) (+ a b) (+ a b)) =>
(define (F1) (+ a b)) (+ (F1) (F1) (F1))

(/ (* (+ a b) (+ c d)) (* (+ e f) (+ g h)) (* (+ q r) (+ t v))) =>
(define (F7 v34 v33 v32 v31) (* (+ v31 v32) (+ v33 v34)))
(/ (F7 d c b a) (F7 h g f e) (F7 v t r q)))

(+ (* (+ a b) (+ c d)) (* (+ a b) (+ c d)) (* (+ a b) (+ c d))
   (/ (- a b) (- c d)) (/ (- a b) (- c d))) =>
(define (F348 v7627 v7625) (v7625 (v7627 a b) (v7627 c d)))
(define (F369) (F348 + *))
(define (F7) (F348 - /))
(+ (F369) (F369) (F369) (F7) (F7)))
```

Once we have a new program, e.g. `(+ F1 F1 F1)` we have to determine what the series of random choices are for deriving this program from the grammar. These random choices are used to create a new state/trace and also to determine whether this proposed transition will be accepted or rejected by the Metropolis-Hastings algorithm.

The description given here is slightly inaccurate with respect to the actual implementation, but hopefully the general ideas have been properly conveyed. We have almost finished integration of abstraction into the compiler for Church and plan to test the system on the same domain as [**?**]. The main aim of the project is to incorporate more specific program transformations into a MCMC approach for searching program space. By building on top rigorously developed probabilistic techniques we gain the strong theoretical guarantees that come with such methods.