Inducing Probabilistic Programs by Bayesian Model Merging

1 Abstract

This report outlines an approach for inducing probabilistic programs within the Bayesian model merging framework. The models we use are programs that can capture context-sensitive and recursive structure in data. There are two main merge operations: the first is based on combining common subexpressions within a program and the second focuses on reducing the number of parameters for functions in the program. We demonstrate this approach in the computer graphics domain of inverse procedural modeling.

2 Introduction

There is a trade-off between the ability of a model class to capture a wide range of patterns and the feasability of finding and fitting a model in the class to interesting data [?]. Much of machine learning has focused on studying classes of models with limited expressiveness in order to develop tractable algorithms for modeling large data sets. Our investigation takes a different approach and explores how learning might proceed in a very expressive class of models with a focus on identifying patterns from very small amounts of data.

We examine a class of generative models that are represented as programs in a probabilistic programming language. These programs can have parameterized functions, recursion, and nondeterministic assignment of values to variables (this will fill a role similar to having conditional expressions), which allow for representation of many different types of patterns. We will frame searching this space of models in terms of Bayesian model merging [?] and demonstrate the class' ability to capture interesting patterns in the domain of inverse procedural modeling [?].

3 Bayesian Model Merging

4 Merge Operations

4.1 Common Subexpression Refactoring

4.1.1 Overview

The following example illustrates the common subexpression refactoring (CSR) transformation

Given the program

```
(uniform-draw
  (list
    '((list a (list a (list b) (list b)))
      (list a (list a (list c) (list c))))))
```

a possible result of this transformation would be

```
(let ()
  (define (F1 x)
    (list a (list a (list x) (list x))))
  (uniform-draw (list (F1 b) (F1 c))))
```

The first thing to note is both programs have the same behavior, in that both return either (list a (list a (list b) (list b))) or (list a (list a (list c) (list c))) with equal probability.

The transformation can be described as refactoring subexpressions that partially match in a program with a function whose body is the common parts of the matching subexpressions. In the above example the subexpressions that partially match are (list a (list a (list b) (list b))) and (list a (list a (list c) (list c))). The common subexpression is (list a (list a (list x) (list x))), the function created using this common subexpression is F1 and the original subexpressions are refactored as (F1 b) and (F1 c).

A CSR transformation can be created for each pair of subexpressions that have a partial match. In the case of (+ (+ 2 2) (- 2 5)) the following pairs of subexpressions have a partial match: ¡2,2¿, ¡(+ 2 2), (- 2 5)¿, ¡(+ 2 2), (+ (+ 2 2) (+ 2 5))¿, ¡(+ 2 5), (+ (+ 2 2) (+ 2 5))¿, the only subexpressions that do not have a partial match in this example.

4.1.2 Anti-unification

The process of finding a partial match between two expressions is called anti-unification. One way to understand the process is in terms of the syntax trees for the expressions. Every s-expression can be thought of as a tree where the lists and sublists of the s-expression make up the interior nodes and the primitive elements of the lists (e.g. symbols, numbers, etc.) are the leaves.

The following tree corresponds to the expression (+ (+ 2 2) (- 2 5))

We illustrate the process of anti-unification on the expressions (+ (+ 2 2) (- 2 5)) and (+ (- 2 3) 4) The first step is to compare the root of the trees and make sure we have lists of the same size, in this case they are both of size three so we have a matching roots and a partial match of (* * *) where the *'s are yet to be determined. Now we recursively attempt to match the three subexpressions + with +, (+ 2 2) with (- 2 3), and (- 2 5) with 4. Since + and + are both primitives that are the same they match, and our partial match is now (+ * *). Comparing (+ 2 2) and (- 2 3) we see that they are both lists of size 3 and so they match giving us a total partial match of (+ (* * *) *). Again we recursively match subexpressions of (+ 2 2) and (- 2 3) i.e. + to -, 2 to 2, and 2 to 3. Since + and - are primitives that don't match we replace them with a variable to get a total partial match of (+ (V1 * *) *)

Likewise after comparing 2 to 2 and 2 to 3 we'll get (+ (V1 2 V2) *) as our partial match. In the final comparison of (- 2 5) and 4 we check can see there is no match because one (- 2 5) is a list and 4 is a primitive so the final result of our anti-unification is (+ (V1 2 V2) V3).

In general anti-unification proceeds by recursively comparing two expressions, A and B. If A and B are the same primitive that primitive is returned. If A and B are lists $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ of the same length then a list C $(c_1, \ldots, c_n)$ where each element c_i is the anti-unification a_i and b_i is returned. For all other cases when A and B do not match a variable is returned.

4.1.3 Refactoring

Now that we have found patterns i.e. partial matches between two subexpressions of some expression, E, we can use these patterns to compress E.

In the case of (+ (+ 2 2) (- 2 5)) the partial match we would get between the subexpression ¡(+ (+ 2 2) (- 2 5)), (+ 2 2)¿ is (+ V1 V2). We refactor the original expression (+ (+ 2 2) (- 2 5)) in terms of (+ V1 V2) by creating a function (define (F1 V1 V2) (+ V1 V2)) and applying it in the expression. For example one application could be (F1 (+ 2 2) (- 2 5)) and another could be (+ (F1 2 2) (- 2 5)). In general we'll apply the function everywhere possible and get a refactored program like so...

(+ (+ 2 2) (- 2 5)) =¿ (let () (define (F1 V1 V2) (+ V1 V2)) (F1 (F1 2 2) (- 2 5)))

The input to the refactoring procedure is a function, F, created from anti-unification and an expression, E, which will be refactored in terms of F [1]. In the example above F was (define (F1 V1 V2) (+ V1 V2)), E would be (+ (+ 2 2) (- 2 5)), and the result of refactoring was (let () (define (F1 V1 V2) (+ V1 V2)) (F1 (F1 2 2) (- 2 5))).

Refactoring proceeds recursively by taking the body of F and trying to match it to the expression E. If the match holds then the application of F is returned where the arguments to F are refactored (if F has arguments). If the match does not hold then E is returned with each of its subexpressions refactored. If E is a primitive and not a match it is returned. In the example (+ (+ 2 2) (- 2 5)) matches (+ V1 V2) so an application of F1 to refactored arguments (+ 2 2) and (- 2 5) is returned resulting in (F1 (F1 2 2) (- 2 5)).

[1] The choice of separating programs into a list of definitions and a body is more of a software engineering decision. If you want to think of refactoring more generally we are taking a function F and refactoring some s-expression in terms of F (where the s-expression might be a program p like we've described earlier).

In reality at the top level we assume the expression we are refactoring is a program that has two parts: 1) a list of function definitions that are s-expressions and 2) a body that is an s-expression. See appendix for more details. The idea will be to take the body of the function F and replace any subexpressions of p that match this body. A subexpression that matches the body is replaced by an application of F. We first do this replacement for each of the functions in p and then the body, resulting in a program p'. After all the replacements are made F is inserted into the set of functions for p' and this is our refactored program.

Determining whether there is a match between a function body and an expression is known as unification. Earlier we described anti-unification which can be thought of as a process to create a pattern from two expressions, unification can be viewed as the opposite process of seeing if and how a given pattern fits onto an expression. The return value of unification is a list of assignments for the variables of the function that would make the function the same as the expression.

The example listed earlier where the function F is (define (F1 V1 V2) (+ V1 V2)) and the s-expression being refactored is (+ (+ 2 2) (- 2 5)) is used to illustrate unification. Since (+ (+ 2 2) (- 2 5)) and (+ V1 V2) are the same length unification is applied to the subexpression pairs ¡+,+¿, ¡(+ 2 2), V1¿, ¡(- 2 5), V2¿.

Unification between + and + returns nothing since neither is a variable and they match. Unification between (+ 2 2) and V1 returns the assignment of (+ 2 2) to V1 and likewise for (- 2 5) and V2. So the function F1 matches the s-expression (+ (+ 2 2) (- 2 5)) with variable assignments V1:=(+ 2 2) and V2:=(- 2 5).

If the expression being unified with F1 had been (- (+ 2 2) (- 2 5)) then unification between the outer - of the expression and the + of F1 would have returned false and the unification would have failed.

The input to unification is a function F, and an expression E. Unification occurs recursively by checking whether the body of F and E are lists of the same size. If they are the same size then unification returns the unification of each of the subexpressions. If they are not the same size or only one of them is a list unification returns false. In the case where both expressions being unified are primitives true is returned if they are equal and false otherwise. In the case where the function expression of the unification is a variable an assignment is returned i.e. the variable along with the other expression passed to unification. At the end a check is made to see if any unifications have returned false, in which case unification of F and E returns false. There is also a check that any variable that is repeated in F has the same value assigned to it for each place it appears. If this is not the case unification returns false. If unification is not false then an assignment for each unique variable of F is returned.

4.2 Argument Internalization

4.2.1 Recursion

Argument internalization is a program transformation that takes a function in a program and removes one of its arguments. The value for the argument in the body is drawn from a uniform distribution over all the instances of that argument.

We illustrate how recursive patterns can be discovered using argument internalization with the program (let () (list (list a))).

We can apply the CSR tranformation to (let () (list (list a))) to get the following program: (let () (define (F1 x) (list x)) (F1 (F1 a)))

Now we'll apply argument internalization and remove F1's argument.

The first step is to change the definition of F1 so that x is drawn from a distribution of past instances of the argument like so

(let () (define (F1 x) (let ([x (draw '(a (F1 a)))]) (list x)) (F1 (F1 a))))

Here the instances of x (i.e. what was passed into the function F1) are '(F1 a) and a and draw is a function that selects one of these for the value [2].

The final step is to remove the argument from F1 and any applications of F1 resulting in the program

(let () (define (F1) (let ([x ((uniform-draw '((lambda () a) (lambda () (F1)))))]) (list x))) (F1)))

[2] In the actual implementation we'll want to delay evaluation of (F1 a) until after the draw has been made. Having lazy evaluation is a natural solution, but there are other possible ways to deal with this issue, such as making the mixture over thunks as demonstrated in the example.

4.2.2 Context Sensitivity

4.2.3 Algorithm

In the general case we start with a program p, a function F in p, and a paramter v of the function F that we wish to "internalize." It is assumed F is applied somewhere in the program (either in the body or in another function). In the example above p was (let () (define (F1 x) (list x)) (F1 (F1 a)))

F was (define (F1 x) (list x))

and v was x. F1 was applied in the body of p.

The first step is to create a new function F' that does not use v as a parameter, but instead defines v in the body of of F' as a mixture of the instances of v. Instances of v in the example are (F1 a) and a. F' was (define (F1) (let ([x ((uniform-draw '((lambda () a) (lambda () (F1 a)))))]) (list x)))

Once the new function is created all applications of F in the program are changed so that the argument for parameter v is removed. In the example this means replacing (F1 x) with (F1). This results in the final transformed program. In the example we get (let () (define (F1) (let ([x ((uniform-draw '((lambda () a) (lambda () (F1)))))]) (list x))) (F1))

5 Inverse Procedural Modeling

6 Problem setup

6.1 Factor graphs

6.2 Computing the Posterior

7 Discussion

7.1 Open questions