1

CONSTRAINED INFERENCE - ZENNA TAVARES

We address the problem of tractably drawing exact samples from conditional probability distributions. In particular we propose a formalism for constructively incorporating constraints into generative models for probabilstic inference. The presented approach builds upon formalisms of inference as conditional execution of a program, and analysis of such programs with respect to their formal semantics.

1.1 INTRODUCTION

Probabilstic inference has established itself as a means of reasoning in domains subject to uncertainty. Although the rules of conditional probability state how to update our beliefs in hypotheses conditioned on evidence, it tells us only declaratively, and leaves us with no guidance as to how hypotheses should be constructed in the first place. In anticipation of the distinction emphasised in this essay, we describe conditional probability and in particular Bayes' Rule as the *logic* of a program, while the the *control* or *procedure* remains a topic of continued research.

Inference typically refers to finding the expectation of some function with respect to a probability distribution, computing the posterior distribution, or drawing from it. Numerous approaches exist, but our focus is on Monte Carlo methods whose strengths lie in their applicability to high dimensional distributions, where direct sampling becomes intractable. Inference by means of random perturbations to hypotheses is often inefficient however; we would prefer to not waste resources proposing candidate hypotheses which violate sensible constraints. In other words, we would like to make good guesses.

The objective of this study is to explore conceptual and practical methods of making good guesses by incorporating constraints *constructively* into generative models. We frame construction in constrast to testing; to construct is (ideally) to not test. We build upon recent interpretations of probabilstic inference in computational terms, in particular the QUERY operation, which performs universal inference in the probabilstic programming language Church [1]. We suggest a new constructive perspective CONSTRAIN, which given a generative model manifested as a stochastic *prior program* P, and a conditioning predicate C aims to construct a new program P^* which samples only values which adhere to our condition.

Universal inference [1] is analagous to notions of universality in computation. Informally it refers to the ability to perform conditional simulation of any computable probabilstic generative process.

The feasibility of this aim is vulnerable to skepticism; how can we construct only constrained proposals without testing to see if they satisfy our constraints? It is unlikely this objective can be achieved in full generality, in some cases we have no alternative than to generate and test. Furthermore we shall see that relaxations of the above maxim may lead to more pragmatic implementation strategies; to construct is to test less perhaps, or to test only partially and incrementally. Precisely what distuiginshes one constraint from another in terms of the extent to which it can be implemented constructively, is a topic of great interest and speculated on in our conclusion. Yet, we can motivate the plausibilty of constructive inference by appealling to common experience: consider the ease with one can compose a paragraph that rhymes, generate a polynomial expression, or draw an acyclic graph. Each of these examples entails a logical constraint which we satisfy without hinderance. While it is risky to draw conclusions about the computational hardness of a problem based on our subjective difficulty when solving it, each of the above examples has a known efficient algorithm. We conjecture the automatic discovery of algorithms as a means towards conditional sampling, is necessary for any general inference procedure approaching optimal efficiency.

First we will summarise QUERY, a computational theory of probabilstic inference. Then we formalise the key concept of this paper, CONSTRAIN, in terms of QUERY. Two approaches to implementing CONSTRAIN are then described in detail.

1.1.1 *Query*

QUERY formalises probabilistic inference as a *program* in a corresponding model of computation. Developed first as a primitive function in the probabilstic programming language Church [1], QUERY has been described in terms of probabilstic generalisations of the λ-calculus and Turing machine, both classical models of computation.

In its original formulation, QUERY is a higher order function which accepts as input a stochastic expression to be evaluated and a set of predicate conditions. In lisp (clojure) notation:

Notation: defn defines a function. The following term is the function name and terms inside square brackets [] are argument names. Letís used to assign names to values, conceptually similar to assignment of values to variables in imperative languages.

```
(defn query [exp pred]          ; Define function
  (let ((val (eval exp))        ; sample from model, call it val
    (if (pred val)              ; If val satisfies conditions
        val                     ; Then.. return it
        (query exp pred)))))    ; Otherwise try again
```

In [3] a formulation in terms of a probalistic turing machine (PTM) is given. A Turing machine [4] is a mathematical abstraction of a machine, which may read, write and seek access on a finite collection of infinitely long binary tapes. Prior to execution, its input is loaded onto one or more of its tapes, and the output is the content of its tapes after the machine halts. A probabilistic Turing machine (PTM) is a Turing machine equipped with a tape consisting of a sequence of independent random bits, which is accessible to the Turing machine as a read only randomness source.

QUERY is a PTM which takes two inputs, a prior program P and a conditioning predicate C. Both P and C are themselves encodingings of PTMs that take no input. QUERY generates a sample from P. Then, if C is satisfied this sample is outputted, otherwise the process is repeated.

It should be of little surprise that both these formulations are equivalent, as they both perfom the function of drawing samples from a prior distribution conditioned on C, using rejection. While rejection sampling provides an simple and intuitive understanding of the meaning of QUERY, it is of course grossly inefficient for the majority of non-trivial problems.. Much research in inference is in looking for tractable approximations and alternatives.

1.1.2 *Constrain*

The semantics of CONSTRAIN can be readily understood in terms of QUERY. Expanding on the lisp definition of QUERY given above, CONSTRAIN is a higher order function expecting a prior program P, and taking *constraint* C^*. Uncertain evidence, or equivalently soft constraints are not considered here. Hence a condition C and constraint C^* can be used almost interchangeably, but are differentiated in terms of hardness; constraints are logical and must be true. CONSTRAIN simply returns a function of no arguments which is QUERY with all its arguments evaluated.

```
(defn constrain [exp pred]        ; Define a function
  (fn [] (query expr pred)))      ; Return a 0-ary function
```

While QUERY returns a sample given a model and condition, CONSTRAIN returns a function of no arguments which calls QUERY. In deterministic programs, a function with all its arguments evaluated is simply a value which evaluates to itself, and little is typically gained from taking a functional perspective. In stochastic programs however the output of CONSTRAIN implictly defines a conditional distribution. This alone differs little from QUERY, as we have only described the semantics. We differentiate with the objective that that C^* is conditioned on constructively; we wish to refrain from applying it as a predicate to fully formed samples, i.e., testing, and instead exploit its semantics to find a more efficient means. The rest of this study is devoted into proposed means of doing this.

2 CONSTRAINED GENERATIVE MODELS

First consider a simplified instance of the problem: our generative model defines a distribution over linear transformations of a random vector of uniformly distributed real values, where the length of this vector and parameters to each random variable (both lower and upper bound) are fixed and known ahead of time.

A constraint involves the conjunction and disjunction of any number of linear equalities and inequalities on these samples. Both the generative model and the constraint are represented as programs in a limited functional language, which importantly is side-effect free and lacks general recursion.

Both a logical and geometric perspective brings some clarity; our constraint program C implicitly defines a logical formula composed of the conjunction and disjunction of a fixed number of literals, each literal representing an inequality of the form $Ax \leqslant b$ (note we can always convert an equality to two inequalities, as well as convert a greater than relation to a less than through negation). For instance the expression:

```
(if (> x1 10)                     ; If val satisfies conditions
    true                          ; then our constraint is satisfied
    (or (> x2 10) (> x1 2)))      ; Otherwise if x2>10 or x1>2
```

Defines a logical formula of the form:

$$x_1 > 10 \vee (x_1 \leqslant 10 \wedge x_2 > 10) \vee (x_1 \leqslant 10 \wedge x_1 > 2) \qquad (1)$$

The extraction of this logical expression from the program is important, and the first step of the proposed method. The formula is in disjunctive normal form: one of an infinite number of equivalent alternatives but unique in that it is the disjunction of a minimal number of clauses, where each clause is the conjunction of a number of literals. Each clause can be thought of as a set of local conditions which when all are simultaneously true, will cause the constraint to be satisfied. Geometrically, each literal is a linear inequality defininig a half-space which splits $\mathbb{R}^n$ into two, and designates one side of the split as feasible and the other infeasible. The entire clause is the intersection of a number of these half-spaces and is thus a convex polyhedron. The entire formulae in this form can then be viewed as dividing $\mathbb{R}^n$ into a number of possibly overlapping convex polyhedrons, within which our sample is permitted to lie.

Both the generative model G, and condition C define constraints on samples. These constraints are compiled into a disjunctive normal form, i.e. a set of convex polyhedra. Then we sample points within the volumes of these convex polyhedra.

2.1 A MOTIVATING EXAMPLE: MOTION PLANNING

Planning is a critical problem in robotics, computer graphics, aritificial intelligence. Purely geometric strategies have been supplanted by probabilstic methods, which broadly, rely on efficient collision detection to find feasible points, and grow path trees between these points. While geometric methods suffer from local minima, probabilstic methods can suffer from not being able to pass through small gaps.

We can take our approach to a simplified instance of the motion planning problem, in a two of three dimensional space we seek to construct a path $x_0, y_0, x_1, y_1, .., x_n, y_n$ such that the first point belongs to some start region, the last to a target region, and the path avoids all obstacles. Obstacles can be defined as polyhedra, but for simplicity and without loss of generality we will only isothetic (axis aligned) rectangles or boxes, and represented in h-space.

The obstacle avoidance constraint can be stated precisely by asserting that no edge $x_i, y_i, x_{i+1}, y_{i+1}$ may cross the boundary of an obstacle. This is a non-linear problem, since each point. TODO: Describe non-linear problem mathematically. A similar linear problem is to enforce that each vertex does not fall within an obstacle. While necessary, this alone is insufficient since the vertices of an edge may not collide with an obstacle but the path between them may cross the obstacle.

I suggest this can be overcome by incorporating two extra details. First we ensure that the maximum edge length is within some radius. This is a linear problem. Then we increase the number of points.

Effectively we have found a linear equivalent of a nonlinear problem, with the trade off that we increase the dimensionality of the problem.

Constraint can be described like this:

And in logical form

Hence the trade-off is severe. For every new point we add we will gain two new dimensions, and described in more detail in the following paragraph, you'll get b more polytopes.

Niavely scales very badly. But we can ask many questions, are all these polytopes important, perhaps some contain others. Perhaps some while logically true are geometrically impossible. Perhaps we can have a coarser representation of these convex polytopes. Perhaps there is structure in the way in which they are growing

{Algorithms

2.2 PROGRAM ANALYSIS

Program analysis if or and functions primitive recursion symbolic exection

2.3 SAMPLING

Polytope Sampling Abstraction Our primary results are methods in ensuring the correctness , we analyse these points in detail after a first motivating example.

We then analyse P to extract further constraints on the samples. These constraints will define regions of convex polyhedra We sample from these regions

Theorem: Polytope sampling is correct.

Theorem: we can sample from disjoint polytopes by first selecting a box and then sampling within that box.

Difficulties arise primarily from two sources, first is avoiding bias and second is handling complexity.

2.4 HANDLING COMPLEXITY

Problem Extraction of convex polyhedra from program - number of convex polyhedra typically exponential in number of variables - We need tocounter act this, first pass: – Subsumption/redundancy, some expressions are redundant – infeasible, some expressions when in a conjunction become infeasible — These sets of expressions appear in sets of regularity

2.4.1 *Inconsistency*

Logical inconsistency denotes infeasibility of the conjunction. We can determine this efficiently using linear programming.

2.4.2 *Subsumption*

Handling this logically Geometrically

2.5 ABSTRACT INTERPRETATION

- Previous methods may help in removing redundant clauses but number of polytopes may and likely will still be exponential - Joining means placing an abstraction around some terms

2.5.1 *Exploiting Regularities*

- In many problems the convex polyhedra are positioned with some regularity - Can we exploit this - parameterised boxes

2.6 OVERLAPPING POLYHEDRA

Covering of these convex polyhedra Accounting for overlapping convex polyhedra - Overlapping polyhedra cause 2 problems – 1. Bias, if not accounted for – 2. Inefficiencies when covering

2.6.1 *Removing overlap*

First solution is to find intersection between two convex polyhedra. Problem 1. This would require conversion to vertex representation which can be exponential 2. Aand Bwould not necessarily be convex any more.

Hence we are more interesting in dissecting the abstraction

Dissecting exactly The problem then can be stated as given a set of overlapping intervals in the domain, we wish to disect these into a set of non-overlaping intervals. It's not clear how hard this problem is, and it is little studied in higher than two or three dimensions.

It should be clear that must be at least as hard as Klee's problem, which is to fine the volume of the union of a set of boxes.

Naive approach is to form a grid of every bound for every box in every dimension. This results in aworse case peroramnce, which is exponential in d.

There are two areas in which we could relax this problem. First we can permit some overlap, in accodinace with preceeding paragraph. If an exact solution is infeasible we would like to be able to tune how much overlap, such that it can be managed. 2. We can tolerate some overextension into free space because 1) Such space may be removed in our divisive stage 2) anypoint landingin free space will be a rejected sample. Undesirable, but not changing the distribution

Our output sensitive growth algorithm.

Problem 1: In order to be able to disect boxes we ned to know which boxes are overlapping (Do we?), how ca we do this efficiently.

Problem 2: How can we make the expansion of the algorithm constant time

2.6.2 *Permitting overlap*

A simple way to permit overlap is to assign every element of the set to a single abstraction. Numerous reasonable ways exist to do this, one obious way is to enforce a total order on the abstraction, such that any point is assigned to the most dominant abstraction which encapulates that point

If a sampled point as not selected by first selecting its dominant abstraction it is discard. The problem with this is that we are wasting points, perhaps a large percentalge of points. What we can do instead is have a cache for each abstraction

PROBLEMTOSOLVE: Does This method work with abstractions as it does with polytopes.

PROBLEMTOSOLVE: How can we find a good assignment

2.7 RESULTS

2.8 NON-LINEAR CASE

2.9 CONCLUSION

Here the probabilstic inference framework QUERY was given a constructive perspective with CONSTRAIN. We proposed two methods for implementing CONSTRAIN based on program transformations and symbolic evaluation. We suspect there will be many difficulties with such an implementation, but preliminary evidence suggests even crude approaches to generating more constrained proposals can have dramatic effects on inference performance.

Discussion has been limited to a single constraint C^*, since multiple constraints can be found conjoining each individual constraint together. However, there may be important practical implications in both of the above methods for the order of the conjunction, e.g $A \wedge B$ vs $B \wedge A$. There may for instance be a constraint, only visible with a semantic understanding of the programs which represent A and B, such that $A \implies B$. Moreover, A and B may have an internal structure such that there is a more efficient compiliaton of the two together than just considering each in order.

The central idea of this paper, constructivism, is still open to interpretation. How formally can we differentiate between constructive and non-constructive inference. An appealing possibility is to frame constructivism as the synthesis of an optimal algorithm to sample from the conditional distribution. The cause of variation in the difficulty in which constraints can be adhered to constructively can then be framed in this light; a constraint which is difficult to implement constructively is one where there does not exist (or it is difficult to find) an algorithm which implements it efficiently.

But optimality must be defined with respect to some criterion, and there exist many. An asymtotically optimal algorithm for instance is one which for large inputs performs at worst a constant factor worse than the best possible algorithm. But constant factors can be important, and the worst case is not always indicative of real world performance. Hence, often times even if such an algorithm is known, it is not used regularly.

In spite of these issues the synthesis perspective on inference is appealing, and seeks to appeal to the intuition that whether explicitly or otherwise, almost all approaches to probabilstic inference are implemented on computers as probabilistic programs. The synergy between the fields of program semantics and probability is likely to yield continued insight.

BIBLIOGRAPHY

[1] N Goodman, V Mansinghka, D Roy, K Bonawitz *Church: a language for generative models* 2012.

[2] RA Paige, O Danvy *Automatic program development: A tribute to Robert Paige* 2008:

[3] CE Freer, DM Roy, JB Tenenbaum *Towards common-sense reasoning via conditional simulation: legacies of Turing in Artificial Intelligence* 2012 Arxiv:

[4] A Turing *On computable numbers, with an application to the Entscheidungsproblem* 1936: Proceedings of the London mathematical society.