Notes

Probabilistic Programming

@May 9, 2023

Section 1: Introduction

One would say that is Programming that behaves randomly. A PPL is not made to write a full-fledged software (just as Prolog); it is instead a tool for statistical modeling, which uses the world of programming languages instead of Maths; there's no need for Advanced Mathematical Knowledge about stats/probabilities to get interesting results/models. To sum up: the main goal is analysis, not execution.

Typical Objectives of Statistical Modeling

1. Develop a new probabilistic (generative) model, with unknown parameters
2. Design an inference schema for the model and its parameters, given some observations
3. Using the previous scheme, fit the model to the observed data and deduce the parameters

Typical Objectives of Statistical Modeling in PPL

1. Program a new probabilistic (generative) model, with unknown parameters
2. Using a generic inference algorithm of the language, fit the model to the observed data, and deduce the parameters.

We have several PPL with different objectives. They can differ in the way they write generative models and the expressivity of those models, and in the way inference algorithms are implemented. The tradeoff is that less expressive models usually means more optimized inference algorithms). They usually take a base language and add probabilistic modeling and inference algorithms to them. We use WebPPL → there's a Web Version, easy access.

Section 2: WebPPL

The base language is the Functional Core of Javascript, so we have similarities with what we already know about JS, but major differences. It is derived from Church, which is based on Scheme/LISP. The pros are that it is easy to learn, intuitive, has a good online support for learning, available in both a web version and local. Models are functional, which can be

useful for some behaviors (Cognitive Models). The con is that it is very generic, and thus it can be inefficient for specific models.

The documentation keeps only what's added and removed from the original JS functional core.

Programming with WebPPL

We mainly use these types in WebPPL:

- String
- Number
- Boolean
- Object
  - Array
  - (Probabilistic) Distribution
- Function
- Thunk Random Function

We have the Basic Arithmetics, with some automatic cast. As for Booleans and Equality, we have Abstract Equality (==) which allows the typecast, a Strict Equality (===) which requires the types to be the same, and the classic operators `and` (&&), `or` (||), `not` (!). We have the `if`…`else` construct, and the equivalent ternary operator. We need to declare every variable with the keyword `var` (differently from Javascript, we cannot use `let`, `const`…). Objects contain data, usually indexed by a string or an integer. Arrays are objects which have only integer indexes. Also functions are declared as variables. The `return` keyword is optional, but practical for readability.

We can make comments in WebPPL using classic `//` or `/**/`. The end of line is optional. The online editor automatically pretty-print (e.g. different color for keywords); one could also use a random JS IDE and program locally.

Consider always that WebPPL is a functional language. So, we have that some features of JS are unavailable. Every data is immutable. The only thing you can still mutate is `push` and `pop` on arrays. Also, we have no looping/iterative constructs: we need to think in a functional way and use recursion.

We inherit from classical JS some Standard Methods, e.g. for Strings, Mathematical functions, the so-called "Lodash methods" (the ones you invoke using `_.`), Arrays

(sometimes with new implementations, this is stated in the docs). Also, we have Higher Order Functions, i.e. functions that take functions as an argument (e.g. `map` and `repeat`)

Section 3: Functional Programming

Well studied paradigm for programming languages. The main principle is that flow control comes from the use of function calls and recursion (“Functions are First-Class Citizens”). In modern languages, the border with imperative languages got actually less clear (”map” and “lambdas” for imperative languages, and imperative operators for functional languages).

The main advantages of functional programming are tests and verification. The strong typing procedure implies that it is easy to test the correctness of software produced. Some programs, moreover, are easier to write in an imperative language. But it is also inefficient: we need to remember a stack of contexts that originate from a stack of function calls due to the recursive paradigm (there is often a built-in optimization, but still fares worse than standard imperative procedures); plus, the recursive paradigm can be source of many errors and we miss the mutability of variables and objects. We can find functional programming in parsers, parallelization of pure functions and programs in environments in which verification is more important than efficiency.

Recursive Paradigms

```
var factorial = function(n){
  if (n==1){
    return 1
  }
  else{
    return n*factorial(n-1)
  }
}
```

Tips and tricks:

- Don’t forget the base case: usually implemented by using a conditional structure (it is the equivalent of the while condition)
- It is a good habit (though not always possible) to take an incremental/decremental argument, to easily see what our recursive calls will be.

We can replicate iteration, but it’s less natural.

```
var factorial = function(n) {
  var factorial_iter = function(i,r){
    if (i>n) {return r} else {factorial_iter(i+1, i*r)}
```

```
  }
  factorial_iter(1,1)
}
factorial(5)
```

Another, more natural option, would be to use tail-recursion. A tail-recursive function is any function that ends with a recursion call alone. They are more efficient in practice, and for some inference algorithms, this is the form in which WebPPL actually transforms the program.

```
var factorial = function(n){
  var factorialTR = function(n,r){
    (n==0) ? (return r) : factorialTR(n-1, n*r)
  }
  factorialTR(n,1)
}
```

WebPPL (but also functional programming in general) also supports nested recursion, in which we define a recursive function inside a recursive call, or mutual recursion - functions that are defined in terms of each other (circular call sequence)

Higher-Order Functions

Functions that take other functions as an argument.

```
var myMap = function(l, f) {
  if (l.length == 0){
    return []
  } else {
    var hd = l.pop()
    var tl = myMap(l,f)
    tl.push(f(hd))
    return tl
  }
}

myMap(_.range(1,11), function(x){x*x})
```

Some notes to consider: WebPPL equality (but in general JS) for objects compare the references. Thus two objects with the same data are considered different if they do not have the same pointer; in the last call to map, we use a lambda, which is something now common in many programming languages.

Higher-Order functions allow a higher-level of abstractions. Some direct implementations in WebPPL are:

- `map` → applies a function to all elements of an array, one by one. Variants: `map2`, `mapN`, `mapIndexed`
- `reduce` → iterates a function over each element of the array, starting from an initial point.
- in general, functions that ask for a predicate, e.g. `sort` (comparison function), `all`, `any`, `filter`, `find`…

Forward Models in WebPPL

Section 1: Introduction

We call Forward (or Generative) Model a representation of how the world should work (or seems to work). The uncertainty is modeled by the use of probabilities. In the context of PPL, a generative model is some kind of a program with a random behavior, which we can observe in some way.

Some examples:

- Poll → how the poll works. We can make strong assumptions: we are able to choose n people uniformly in the set of all people at random and all selected people are independent from each other (in the probabilistic sense). Mathematically, is a binomial distribution. We need to model the parameter p (the probability of answering "yes"). The observation is actually doing the poll.
- Linear Regression → what are the laws that can generate the data. We can maybe state that we have a linear function $f(x) = Ax + B$, with A and B unknown, and my data is generated with noise on the output, which can be represented by a normal distribution centered and the actual output. We need to decide a model on how we generate A and B. The observation is the actual generated data from the experiment.
- Board Games → we can generate some states using a random procedure. We can approximate the results by computing a generative model in a PPL and approximate the probability by testing the model a huge number of times.
- Physics Models → Physics here is mainly deterministic, but we have randomness on the initial state of the world. We could also imagine that our models do not simulate perfectly reality and thus the data we generated should be understood with additional noise.

Section 2: Probabilistic Constructors in WebPPL: Flipping Coins

To construct a PPL we need:

- An expressive set of operators to control data and computation flow (a functional core).
- A set of distributions to sample from
- A way to inform the program about the observed data
- Inference procedures.

For our generative model, we need only the first two items.

The “Hello World” of Probability: How to Flip a Coin?

`flip()`

The outcome is probabilistic (you can see the random seed if you want). Plus, what we see there (`flip()`) is a boolean - the sampled data. The function `flip` describes the random procedure (thunk function)

WebPPL allows to easily repeat and visualize a big number of samples.

`viz(repeat(1000, flip))`

The repeat takes as a second argument a thunk function (the signature is `repeat(n, f)` → repeat n times the computation $f()$). Plus, viz is a method from the WebPPL-viz library, which contains functions to visualize the data and distributions.

What if we want to use more coins?

`flip() + flip() + flip()`

And to repeat? A thunk function:

```
var sumFlips = function(){
  flip() + flip() + flip()
}
repeat(1000, sumFlips)
```

And if we want more and more? Use recursion

```
var moreFlips = function(n){
  (n==0) ? 0 : flip() + moreFlips(n-1)
}
print(moreFlips(10))
viz(repeat(1000, function(){moreFlips(10)}))
```

or, by using a Thunk function (`moreFlips` is not a thunk function as we cannot repeat it):

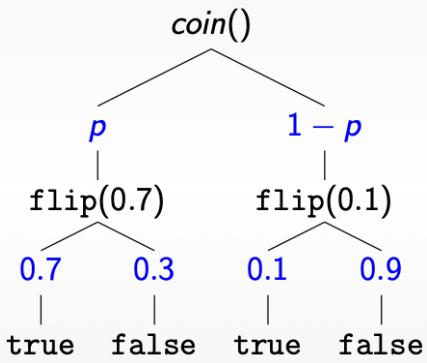
```
var moreFlips = function(n){
  return function(){
    (n==0) ? 0 : flip() + moreFlips(n-1)()
  }
}
viz(repeat(1000, moreFlips(10)))
```

Note that WebPPL user-friendliness for errors is not so great, so pay attention to the notions.

Now I want to bend a coin (modify it). In WebPPL, we take as an input a thunk function and return another thunk function

```
var bendCoin = function(coin) {
  return function(){
    (coin()) ? flip(0.7) : flip(0.1)
  }
}
```

What is the probability obtained from bending a coin with initial probability p of returning `true` ?



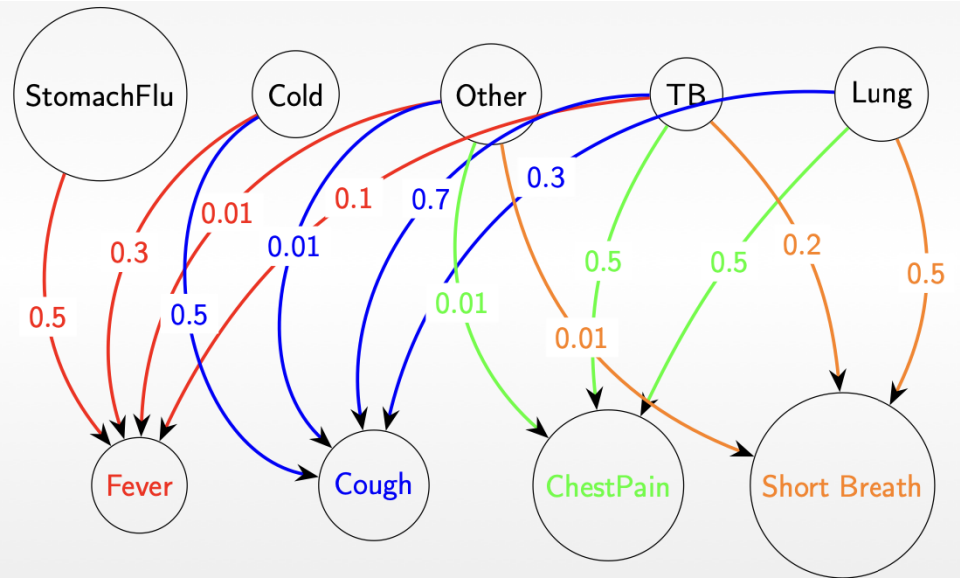
$Bend(p) = 0.7p + 0.1(1-p) = 0.1 + 0.6p$ Thus, $Bend(0.5) = 0.4$ and $Bend(0.95) = 0.67$

Section 2.5: Causal Models and Bayesian Networks by an Example

This will be a simple model for Medical Diagnosis. Here, what you can observe at first glance are the patients' symptoms, which come from diseases; you though can't say which. Thus:

- Generative Model → probability of having the diseases and logical functions deriving symptoms from disease (which are noisy logical functions built from "and" "or" and "flip").

- Data → set of symptoms. We're interested in a function that computes the symptoms



The generic case for this graph would be a Bayesian Network, as a DAG such that has each edge represent a causality relation between vertices. We can easily represent causality in WebPPL.

Section 3: Sampled Data, Random Functions and Distributions

@May 15, 2023

In webPPL a distribution is a specific kind of object, with two methods:

- score → gives a representation of the probabilities
- sample → sampling a distribution gives back a possible value according to the probabilities.

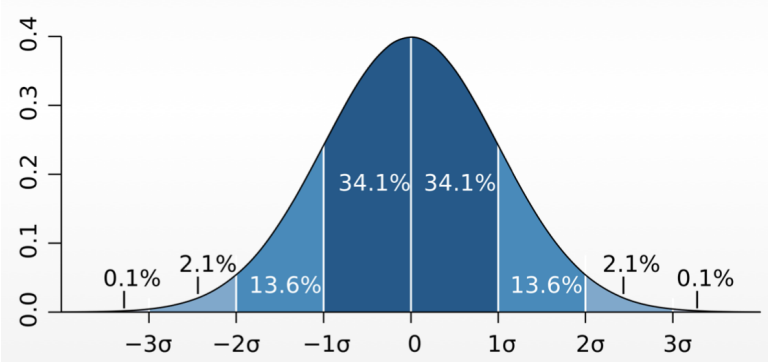
```
var b = Bernoulli({p:0.5})
print(sample(b))
print(b.score(true))
viz(b)
```

We need to call the `sample` operator to work on a distribution, `b` is not a function. We can visualize the distribution using `viz` (visualizing a function is not possible)

We can find a complete list of available distributions in the documentation, these are the main ones for forward models:

- `Binomial({p,n})` → number of success of `n` independent trials, where each trial has a probability `p` of success
- `Categorical({ps,vs})` → sampling in the array `vs` according to the probabilities in `ps`
- `Discrete({ps})` → sampling in integers from 0 to `ps.length-1` according to the probabilities in `ps`
- `Gaussian({mu,sigma})` → normal law centered on `mu` with standard deviation `sigma`.
- `RandomInteger({n})` → uniform distribution on 0,1,…,n-1

A probability density function can be understood as a function associating to each value a relative likelihood that the distribution would be equal to this value after sampling.



Gaussian pdf

In webPPL, Distribution objects and thunk functions are thus different.

- A Distribution is not a procedure: sampling from a distribution is easy as we have perfect information on each probability. Similarly, having access to this, it is easy to output a graphical representation without executing anything.
- A thunk function representing a distribution may actually need to compute its results. Moreover, they can represent complex generative models for which we have no idea about the actual probability of each value

There's though a duality: we can recover a function from a distribution and approximate a distribution given a function. In webPPL:

- from a distribution to a thunk function: take a sample

```
var b = Bernoulli({p:0.5})
var sampleFunction = function(){
  sample(b)
}
repeat(1000, sampleFunction)
```

- the other way: run the function for a lot and then approximate: optimized method `Infer`

```
var sumFlips = function(){
  flip()+flip()+flip()
}
var d = Infer(sumFlips)
```

The function Infer is the central point of webPPL and all PPL have a similar function optimized in some way. Some notes:

- Infer takes as a mandatory argument a thunk function, but there are a lot of optional tuning arguments (when not specified, they are chosen automatically)
- Infer is able to calculate exact probabilities for some cases of forward models; otherwise it gives an approximation
- It can also take into account observed data
- The distribution it computes is usually called marginal distribution
- In itself, it can be a complex function, taking a lot of time. However, once done, you have access to an efficient (maybe imprecise) representation as a distribution object.

As for the notion of score, we can see that probabilities are recorded as their logarithms. This happens for several reasons:

- Logarithm is bijective, thus we can always recover the actual probability.
- Products of probabilities are transformed into sums, which are more efficient
- We can represent small probabilities without losing much precision due to the floating point.

Section 4: Persistent Randomness

Memoization is an optimization technique in which you store results of all the previous function calls, so that you don’t need to execute them again when they are called on the same input. Useful for complex recursive functions in which some recursive calls may be the same through different paths of recursion.

When we implement memoization in the deterministic case, we don’t change the semantics of the program, we just increase its efficiency. In functional languages, it is standard to have access to an higher order function `mem` that automatically implements memoization.

In webPPL we have so-called Stochastic Memoization. the idea is the same; however, in the case of a random function, this means that the same sample is returned on a call with the same argument. This time there IS a semantic meaning.

This is useful in cases where you want to draw randomly some parameters or characteristics, but then remember them automatically, without having to manually stock information.

Backward Reasoning in WebPPL

Section 1: Conditioning

In practice, the generative model can depend on some initial parameters of events, and we observe the consequences of the initial parameter. So I would ask myself: “assuming I have observed those consequences, what must the initial parameters have been?”. E.g. in the medical field, we observe the symptoms, but what we want is to find the disease that is behind those symptoms. We call this backward reasoning. The act of observing some data is called conditioning. In WebPPL is done using a list of operators.

In the study of natural languages, conditional inference has an important role: given beliefs about the structure of a language and an observed sentence, what is the actual syntactic structure of the sentence? We call this parsing. There is also the same kind of reasoning in the field of semantics of a natural language.

The `Infer` operator we’ve seen to transform a forward model into a distribution can be used to compute the marginal distribution under some assumptions that give us the backward reasoning

```
var model() = function(){
  var A = flip()
  var B = flip()
  var C = flip()
  condition(A+B+C==3)
  return {'A':A}
}
var dist = Infer(model)
viz(dist)
```

The `condition` operator takes as an argument a boolean expression. As a good habit, we use complex conditioning instead of introducing new variables, so that we can distinguish the forward model and the observations. It is important to understand that conditioning only makes sense if you then infer the marginal distribution, otherwise the operator does not have any semantic value (WebPPL would then forbid it).

Other operators:

- `observe(dist,v)` assumes that we observed the value `v` from a sample from the continuous distribution `dist`. It is semantically equivalent to `condition(sample(dist)==v)`. However, it is more efficient for the majority of inference procedures to use.
- `factor(score)` formally adds `score` to the log probability of the current execution. It can be used to increase/decrease the weight of a particular execution, to guide heuristics, to skew toward observations without enforcing equality.

The `factor` operator is very general. However, it is always better to write models using the more specialized form. In fact, it is common to distinguish between directed and undirected generative models:

- Directed models → you only use the `observe` and `condition` operators without any `factor`
- Undirected models → `factor` is used (often it is the only one used)

Formal definition of Conditional Probability in Mathematics:

$$P(A=a|B=b)=\frac{P(A=a,B=b)}{P(B=b)}$$

Section 2: Inference

You can ask WebPPL to try to compute mathematically conditional probability using the `enumerate` method. The `enumerate` method tries to enumerate all possibilities to compute the conditional probabilities, as expected.

```
var model = function(){
  var A = flip()
  var B = flip()
  var C = flip()
  var D = A + B + C
  condition(D>=2)
  return A
}
var dist = Infer({method:'enumerate'},model)
viz(dist)
```

Enumeration can quickly become unfeasible if the number of probabilistic choices becomes too big, but WebPPL provides ways to tune the enumeration procedure to approximate the marginal distribution:

- Using “maxExecutions” one can change the number of explored executions
- One can change the exploration strategy (“strategy”) by using either ‘likelyFirst’, ‘depthFirst’ or ‘breadthFirst’.

Pros:

1. Gives very precise values for simple problems that can be enumerable
2. Has an heuristic to gain efficiency (consider only likely path)
3. Can be optimized by using judiciously factors and observations as soon as possible

Cons:

1. Not feasible for a large number of samples (bad scalability)
2. Does not work for continuous distributions
3. Can fail to capture some paths
4. Guiding the heuristics can be complex, the expected answer must be well understood to do this

The mathematical enumeration can be good at times, but for complex problems is not feasible. What can we do in the case of conditioning and backward reasoning?

Rejection Sampling

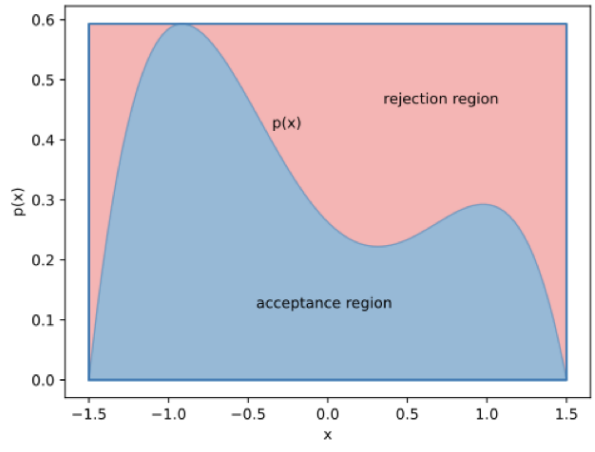


Figure: Rejection Sampling (Armavica, CC BY-SA 4.0, via Wikimedia Commons)

Suppose that we can sample from the square. We reject samples from the red region and accept samples from the blue region. This is actually efficient in some cases, especially

problems in a low dimension. In a WebPPL program we sample from the forward model and we define the acceptance and rejection regions using the `condition` operator: if the sample satisfies all the conditions it is accepted.

For simple examples, we can implement rejection sampling ourself.

```
var takeSample = function(){
  var A = flip()
  var B = flip()
  var C = flip()
  var D = A + B + C
  return D >= 2 ? A : takeSample()
}
viz(repeat(1000, takeSample))
```

Also, we can write this using `Infer`

```
var model = function(){
  var A = flip()
  var B = flip()
  var C = flip()
  var D = A + B + C
  condition(D >= 2)
  return A
}
var dist = Infer({method:'rejection', samples:1000}, model)
viz(dist)
```

This method corresponds mathematically to computing the conditional probability. Indeed, suppose that we make N_{total} samples and we accept those that satisfies $B = \text{true}$, denoted by $N_{B=\text{true}}$. Then, what we compute with rejection sampling is:

$$P(A = a, B = \text{true}) \sim \frac{N_{A=a,B=\text{true}}}{N_{B=\text{true}}} = \frac{\frac{N_{A=a,B=\text{true}}}{N_{total}}}{\frac{N_{B=\text{true}}}{N_{total}}} \sim \frac{P(A = a, B = \text{true})}{P(B = \text{true})}$$

Pros:

1. Simple method, but efficient for some problems
2. Gives a direct approximation of the marginal distribution

Cons:

1. Does not work well with very rare events (the majority of the samples are rejected…)
2. In general, it does not scale well with high dimensions

The Infer operator can be used with several different methods. A very easy one is “forward”. This method ignores all conditioning and can only compute the actual forward model without any backward reasoning. You can give the number of samples also using the “sample” option, which is practical to avoid some copy/cut, for example if you want to compare the forward distribution with the marginal distribution obtained with conditioning without having to rewrite the same program without conditioning.

WebPPL provides also more involved algorithms:

- MCMC (Markov Chain Monte Carlo) → informally, to sample from a distribution ψ, it constructs a random object called Markov Chain, which generates a distribution close to ψ. We have several variants available in WebPPL (MH: Metropolis-Hastings, HMC: Hamiltonian Monte Carlo…)
- SMC (Sequential Monte Carlo) or Particle Filtering → informally, it tries to enumerate only the important paths, similarly to using enumeration with “likelyFirst”. It makes use of resampling each time we make an observation. Works well in cases when you can actually “alternate” between sampling and factoring,
- Variational Inference (`Optimize`) → it tries to approximate the wanted distribution in a family of guide distributions.

Section 4: Bayesian Data Analysis

The approach of Bayesian Data Analysis (BDA) is an approach to making sense of some observed data, and it is a particular case of backward reasoning. A BDA model is a generative model based on some parameters, which are generally not observable directly, so we must infer them from some data. In order to actually give a generative model, we need to make some assumptions on which values the parameters could take.

Typically, for a poll, you have an unknown parameter which is the actual probability p that a random person will vote for candidate A. Thus, you can, for example, assume that it is uniformly distributed between 0 and 1. Then, you can actually run a poll and observe the results. You can then update your beliefs on this probability p: it is not uniformly distributed anymore.

```
var k = 1
var n = 20
var model = function () {
  var p = uniform(0, 1);
  observe(Binomial({p : p, n: n}), k);
  var posteriorPredictive = binomial(p, n);
  recreate model structure , without observe
  var prior_p = uniform (0 , 1);
```

```
  var priorPredictive = binomial(prior_p , n);
  return {
    prior : prior_p , priorPredictive : priorPredictive ,
    posterior : p, posteriorPredictive : posteriorPredictive
  };
}
var posterior = Infer(model);
viz.marginals(posterior)
```

For a given Bayesian Model, along with some data, we want to analyze four important parameters:

- For parameters:
  - Prior Distribution → Initial hypotheses on the parameters before seeing the data
  - Posterior Distribution → The updated beliefs about those parameters, after seeing the data
- For the forward model, distributions on possible data:
  - Prior Predictive Distribution → what data to expect, given the initial belief on parameters
  - Posterior Predictive Distribution → what data to expect, on the same model, but with the updated beliefs after already observing some data.

This is not the “standard” way of doing statistics. In particular, for BDA we need to assume some prior distribution over the parameters, it is essential to the reasoning (which is not the case in frequentist statistics). This form of reasoning is also associated to learning: with BDA, we actually have a belief that we update according to some data, which is the main starting point of machine learning.

Bayes’ Rule

$$P(h|d) = \frac{P(d|h) \times P(h)}{P(d)}$$

$$P(h|d) = \frac{P(d|h) \times P(h)}{P(d|h) \times P(h) + P(d|\neg h) \times P(\neg h)}$$

where d is the data and h the hypotheses over our parameters

Essentially, to compute the posterior distribution over parameters $P(h|d)$, you only need to have a prior distribution over your hypotheses ($P(h)$ and $P(\neg h)$) and you need to be able

to compute the prior predictive distribution ($P(d|h)$ and $P(d|\neg h)$).

Markov Chain Monte Carlo

Section 1: Enumeration, Rejection Sampling and Importance Sampling

We reconsider our example of the biased coin

```
var baserate = 0.1
// if the baserate is too low, samples will be rejected
var infModel = function(){
  Infer({method: 'rejection', samples:100}, function(){
    var A = flip(baserate)
    var B = flip(baserate)
    var C = flip(baserate)
    condition(A+B+C>=2)
    return A})
}
```

If the base rate is too low, more samples will be rejected, thus rejection sampling will take more time. However, given enough time, rejection sampling still has a good precision.

Rejection Sampling loses efficiency with complex models, but not precision.

```
var baserate = 0.1
var numFlips = 3
var infModel = function(){
  Infer({method:'enumerate'}, function(){
    var choices = repeat(numFlips,
      function(){flip(baserate)})
    condition(sum(choices) >= 2)
    return choices[0]})
}
```

For the enumeration method, the problem does not come from the probability of each path, but from the number of possible paths. If you increase the number of flips, then each flip multiply by 2 the number of paths, and you quickly become unable to enumerate all paths.

With complex models, enumeration is impossible. Restricting the number of samples would allow a fast algorithm, but very imprecise.

Markov Chain Monte Carlo (MCMC) is an algorithm that can give you a tradeoff between execution time and precision of the marginal distribution. If you lower the base rate in the MCMC approach, the algorithm will not slow down but it will become less stable. Also, the

number of paths does not increase exponentially the computation time of the MCMC. The imprecision should not be too big.

In practice, different algorithms should be use for different problems, and finding the right algorithm is an important question in itself (considering also that each algorithm can be given a set of parameters that may also change behavior).

Heuristic implemented in webPPL for `Infer` when no method is specified:

- `Enumerate`? If it works, it is the best choice. Doesn't work if one has continuous choices or especially huge discrete state spaces.
- Rejection Sampling? Try with one sample and see how long it takes. If it's reasonable, it is the next best thing.
- Want inference to be fast and can tolerate bias? Variational Inference. Start with the mean field, then make fancier guide families as you understand the model better.
- If your model has observations interleaved with sampling (or you can rewrite it like this), give SMC a shot, but look out for filter collapse.
- MCMC is a good fall back (if you run it long enough). HMC tends to be better when the model has continuous variables.

Another algorithm which is not implemented in WebPPL (no practical use against the other methods) is Importance Sampling.

The main problem of rejection sampling is that in high-dimensional spaces, it rejects the majority of samples, thus it takes too much time to get a reasonable amount of samples. However, it doesn't lose precision given enough samples (which means given enough time)

Importance Sampling have the dual problem: it will always be relatively quick, but for complex problems, the inferred distribution could be far from the actual marginal distribution.

Importance Sampling

Given a generative model f with conditioning, we sample N times from the model f. For each sample, we compute its importance, aka the actual probability that this sample was taken. If we have a conditioning, we modify the weight accordingly. Finally, we approximate the marginal distribution by computing the weighted distribution

$$Pr(x_j) = \frac{w(x_j)}{\sum_{1 \leq i \leq n} w(x_i)}$$

The thing here is, we actually only consider a fixed number of samples as if it represents the whole distribution space. This is a “sampling” version of enumeration with a fixed number of paths.

Concrete Implementation of Rejection Sampling

```
var takeSample = function(){
  var A = flip()
  var B = flip()
  var C = flip()
  var D = A + B + C
  return D>=2 ? A : takeSample()
}
viz(repeat(1000, takeSample))
```

The algorithm is optimized for conditioning, informally:

- For `condition(b)` we can immediately reject if `b` is false in the current sample and resample again. Otherwise we continue.
- For `observe(d,v)`, instead of sampling `x` from `d` and rejecting if `x` $\neq$ `v`, the sample is guided toward taking directly the value `c`. That is why it is strictly better than `condition(sample(d)==v)`.
- For `factor(s)`, if `s` is negative we increase the probability of rejection of this sample, otherwise we decrease the probability of rejection (and if it is lower than 0, we need to take the bias into account in the computation of the marginal distribution).

Importance Sampling is relatively easy to formalize in WebPPL. For each random path of the program, we compute a weight:

- For `sample(d)`, we sample a value `v`, and update the weight by adding `d.score(v)`
- For `condition(b)`, if `b` is false, the weight becomes $-\infty$, otherwise we do nothing
- For `observe(d,v)`, we force the sample of `d` to be equal to `v`, and we update the weight by adding `d.score(v)`
- For `factor(s)`, we update the weight by adding `s`.

To obtain the marginal distribution, we repeat this N times and do the normalization described previously.

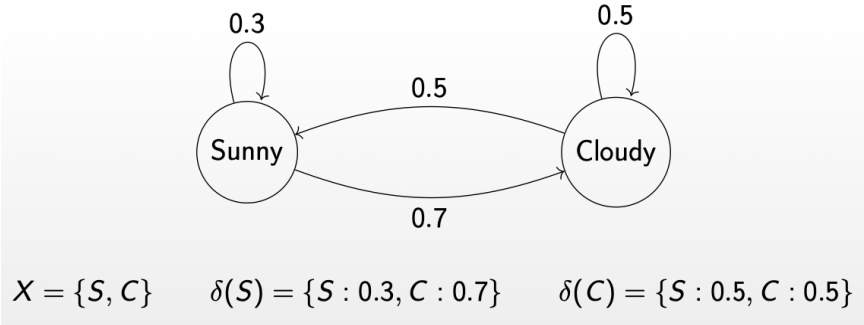
The point is to keep track of some information about the current path of the program in an actual implementation (this is what we call a trace).

Informally, the Metropolis-Hastings algorithm will use both the concept of rejecting some samples, and it will use the information about the weight of the current trace.

Section 2: Markov Chains

A Markov Chain is intuitively a probabilistic automata, or a probabilistic transition system. It consists of:

- A state space X
- A transition function $\delta : X \to \mathcal{D}(X)$, where $\mathcal{D}(X)$ denotes the set of distributions over X



In WebPPL we have

```
var states = ['S', 'C']
var transitionProbs = {S:[.3,.7], C:[.5, .5]}
var transition = function(state){
  var vs = states
  var ps = transitionProbs[state]
  return categorical({vs,ps})
}

// we repeat n times the transition function
var chain = function(state, n){
  return (n==0 ? state : chain(transition(state), n-1))
}

print('State after 10 steps')
viz.hist(repeat(1000, function(){chain('S', 10)}))
viz.hist(repeat(1000, function(){chain('C', 10)}))
```

A Stochastic Matrix of dimension n is a $n \times n$ matrix P whose entries belong to $[0, 1]$ and such that each row vector sums to 1

$$\forall i, \sum_j P(i, j) = 1$$

It can be useful to see Markov Chain as matrices since computing a chain is equivalent to matrix multiplication.

For a given Markov Chain P, we say that ψ is a stationary distribution $\iff \psi \times P = \psi$, which means that ψ is a fixpoint for the transition function of the Markov Chain.

Under some conditions, we can show that for a Markov Chain P:

- P has a stationary distribution ψ
- This stationary distribution is unique
- Any initial state eventually converges to ψ after a sufficient number of transitions.

Thus, we found a way to approximate any distribution ψ: this distribution can be approximated by running a Markov Chain that has ψ as a stationary distribution. The main idea is that even if ψ can be very hard to sample from, we may be able to actually construct a Markov Chain that has ψ as a stationary distribution. Thus, we would have a way to approximately sample from ψ, which is the principle behind MCMC.

Section 3: Metropolis-Hastings

Metropolis Hastings needs some information about ψ, we need to be able to compute ψ up to a normalization factor, which means we need to be able to compute f s.t.

$$\exists K, \forall x, \psi(x) = K \cdot f(x)$$

Obviously, K must be the same for all x and it is actually the hard thing to compute.

Back to Bayes' Rule

$$P(H = h|D = d) = \frac{P(D = d|H = h) \times P(H = h)}{P(D = d)}$$

d is the fixed observed data (or conditioning). The posterior distribution over parameters $P(H = h|D = d)$ is the distribution we want to approximate for all h. Let us call $\psi(h) = P(H = h|D = d)$. $P(H = h)$ is given by the prior distribution, that we can sample from. $P(D = d|H = h)$ consists in running the generative model with a fixed parameter h and comparing to the observed data. $P(D = d)$ consists in running the generative model over all possible hypotheses and compare with data. This is the actual hard part: if you could sample from that directly, you should have used rejection sampling anyway. We call $K = \frac{1}{P(D=d)}$

To sum up, we have

$$\psi(h) = K \times P(D = d|H = h) \times P(H = h)$$

where the function $f : h \mapsto P(D = d|H = h) \times P(H = h)$ is not so hard to compute, or at least it is far easier than K.

We only have an estimation of the function f in the general case, but the point is that approximating f on a given input h is easier than approximating K in both time and precision. Thus, we can say we're in a case where we can consider that we know ψ up to a normalization factor and so we can apply the MH algorithm.

First of all, let us give a sufficient condition for a distribution to be stationary. We say that a distribution ψ satisfies the Detailed Balance Condition for the Markov Chain P if:

$$\forall x, y. \psi(x) \cdot P(x, y) = \psi(y) \cdot P(y, x)$$

The case when $x = y$ is obvious.

We want to show that if a distribution ψ satisfies the DBC, then ψ is stationary.

We want $\psi \times P = \psi$. By definition of matrix multiplication, this means $\forall x, \psi(x) = \sum_i \psi(i) \cdot P(i, x)$

Using the DBC, we obtain:

$$\forall x, \sum_i \psi(i) \cdot P(i, x) = \sum_i \psi(x) \cdot P(x, i)$$

We extract the multiplicative factor $\psi(x)$ that does not depend on i:

$$= \psi(x) \sum_i P(x, i)$$

Since P is a stochastic matrix, we obtain:

$$\forall x, \sum_i \psi(i) \cdot P(i, x) = \psi(x)$$

The principle of the transition function of the Markov Chain constructed by the MH algorithm can be described in two steps:

1. Given a state x, we propose a next state y according to an arbitrary (but fixed) probability distribution $g(x \to y)$. For example, in a continuous state space, we could take a Gaussian around x, in the state of natural numbers, we could propose a state close to x, etc.

2. Once we have a candidate, we decide whether to accept it or not according to an accepting rate $\alpha(x \to y)$. If we refuse this next state, we stay on the same state.

This Markov Chain is easy to simulate as long as both α and g are easy to compute. For g we can take a simple distribution, but for α we need something that will give us the stationary distribution we want.

Let us take this Markov Chain P defined previously. We know that if a distribution ψ satisfies the DBC on P, then ψ is a stationary distribution. We use this condition to define α

$$\forall x, y. \psi(x) \cdot P(x, y) = \psi(y) \cdot P(y, x)$$

$P(x, y)$? We go from x to y iff y is sampled from g and that this new state is accepted with rate α. So we want:

$$\forall x, y. \psi(x) \cdot g(x \to y) \cdot \alpha(x \to y) = \psi(y) \cdot g(y \to x) \cdot \alpha(y \to x)$$

We can rewrite this to isolate the α:

$$\frac{\alpha(x \to y)}{\alpha(y \to x)} = \frac{\psi(y) \cdot g(y \to x)}{\psi(x) \cdot g(x \to y)}$$

Consequently, given that $\psi(.) = K \cdot f(.)$:

$$\frac{\alpha(x \to y)}{\alpha(y \to x)} = \frac{f(y) \cdot g(y \to x)}{f(x) \cdot g(x \to y)}$$

This is what we want and this would be sufficient to have ψ as a stationary distribution. We can thus choose:

$$\alpha(x \to y) = \min(1, \frac{f(y) \cdot g(y \to x)}{f(x) \cdot g(x \to y)})$$

This way, the quotient between the two αs will always satisfy the equality given above.

```
t <- 0
initialize array x of size n
x[0] <- xinit
while t < n do:
  x <- x[t]
  propose y according to g(x->y)
  compute alpha(x->y) = min(1, (f(y)*g(y->x)/(f(x)*g(x->y)))
  with probability alpha do:
    x[t+1]<- y
```

```
    t++
  otherwise:
    x[t+1]<- x
    t++
```

The algorithm can indeed sample from the Markov Chain with stationary distribution ψ, but there are obvious disadvantages compared to an ideal sampling:

- Samples are correlated, a sample $x[t+1]$ depends strongly from the previous sample $x[t]$
- The first samples depends strongly on x_{init}. By the Theorem of Markov Chain, we eventually sample from the stationary distribution, but at the beginning of the algorithm, this is not the case!

However, as usual, you can try to get around these problems at the expense of some efficiency, this is detailed by the options available in webPPL for the MCMC algorithm:

- `samples` → gives the number of samples, the more the better the precision
- `lag` → allows to do several steps between samples (instead of only 1). This way, samples are less correlated with each other. Especially useful if the accepting rate can be small, as MC tends to stagnate.
- `burn` → allows to do some initial steps before the first sampling. This way, the samples depend less on the initial state that was chosen
- `kernel` → describes the transition relation of the MC that is constructed by the algorithm. It can be MH or HMC.
- `verbose` → gives the acceptance ratio seen during the computation.

Section 4: Metropolis-Hastings, Concretely

We define the trace of an execution of a probabilistic program as the list of all results of samples that happened during the execution. To implement concretely MH in WebPPL, we will remember the following information when executing a generative model:

- The trace of this execution
- For each sample point, the whole context of the program (the continuation). This allows in particular backtracking (Functional programming is very practical)
- For each sample point, the weight of the choice done in the trace
- The current weight of the execution (the score, computed as importance sampling)

Concretely, MH is implemented as:

1. We start with an empty trace, and a score 0
2. We run the generative model, keeping in memory the trace, the score and the continuations.
3. This first trace is taken as our initial state. We can then start computing the Markov Chain:
    a. We choose uniformly at random a point in the trace (an index of the list, denoted r). We backtrack entirely an execution from this starting point.
    b. We run again the generative model from this point, obtaining a new trace, a new score and new continuations.
    c. We compute the acceptance rate of the new trace from the old trace. We accept the new trace according to this acceptance rate.
    d. We do this until we have a sufficiently large number of traces (and thus, samples)

How can we compute the acceptance rate given the information stocked in the memory?

Recall that to compute the acceptance rate from the old trace t to the new trace u, we need to compute:

$$\frac{f(u) \times g(u \to t)}{f(t) \times g(t \to u)}$$

We condition on the value of r, the point of backtracking in the algorithm (we call R the random variable representing this point:

$$\frac{f(u) \times g(u \to t | R = r) \times Pr(R = r | currentTrace = u)}{f(t) \times g(t \to u | R = r) \times Pr(R = r | currentTrace = t)}$$

This means that to compute the probability to obtain the new trace t from the old trace u, we can look at the case for which the backtracking point selected in the previous algorithm was r. From this we can compute:

$$\frac{Pr(R = r | currentTrace = u)}{Pr(R = r | currentTrace = t)} = \frac{|t|}{|u|}$$

where $|t|$ denotes the size of t. It corresponds to choosing uniformly an index, and this probability for the trace t is $\frac{1}{|t|}$ and similarly for u.

$$\frac{f(u)}{g(t \to u | R = r)}$$

$f(u)$ corresponds to the new score, because the new execution follows the trace u, and thus the new score is the unnormalized weight of the trace u. Moreover, when executing this new trace u, we recalled all the weight from all the samples we saw during the execution. If we take all those weights starting from index r, and we add them, we obtain the probability that the generative model obtain the new trace u from a rollback of the old trace t assuming that the rollback point was r.

Thus, all the information from the execution of u allows to compute

$$\frac{f(u)}{g(t \to u | R = r)} = \frac{newScore}{\sum_{r \leq i \leq |u|} u[i].weight}$$

Similarly, we can take the other way round, as going from u to t and going from t to u is symmetric

$$\frac{f(t)}{g(u \to t | R = r)} = \frac{oldScore}{\sum_{r \leq i \leq |u|} t[i].weight}$$

In the score of f you take conditioning into account, but in the computation of g we do not, we only look at the weight of samples (we only look at the generative model). Thus, using all the information stocked in the trace, we can actually compute the acceptance ratio and run the MH algorithm.

The use of functional programming allows the generative model to be transformed easily for MH: having no mutable variables simplifies the capture of the continuation and backtracking and when transforming a program to remember the trace/continuation/weights, we only have to redefine the implementation of the functions `sample`, `factor`, `condition`, `observe`.

Causality and Correlation

Causality is a notion that can be tricky to define precisely, but informally, we can say that it means that “the event A causes the event B”. We present the Bayesian approach of causality. We will actually define causal dependence, which is a bit more precise.

In the programming language point of view, we can say that a variable B causally depends from a variable A if it is necessary to evaluate A in order to evaluate B.

In a program, there is an easy way to detect causal dependence. We can say that B is causally dependent from A if we can change the distribution over values of B by changing the value

of A

```
var ChangeValue = function(AValue){
  return Infer(function(){
    var A = AValue
    var B = flip()
    var C = A ? flip(0.1) : flip(0.4)
    return C
})}
viz(ChangeValue(true))
viz(ChangeValue(false))
```

However, we should not confuse causal dependence and statistical dependence. Getting information about the value of C will indeed change our belief on A

```
Infer(function(){
  var A = flip()
  var B = flip()
  var C = A ? flip(0.1) : flip(0.4)
  condition(C)
  return A
})
```

This notion of statistical dependence is what we usually call correlation: from an observation of C, we can deduce something for the value of A. In a probabilistic program, it means that conditioning on C changes the distribution of A.

One main difference between correlation and causal dependence is that correlation is a symmetric relation: if A is statistically dependent from B, then B is statistically dependent from A. Mathematically, we can say that A is correlated with C if and only if $P(A|C) \neq P(A)$. But of course, if we have this then we obtain:

$$P(C|A) = \frac{P(A\&C)}{P(A)} = \frac{P(A|C)P(C)}{P(A)}$$

and this is different from $P(C)$ when $P(A|C)$ is different from $P(A)$.

Note: if A is causally dependent from C, then A is correlated with C. Intuitively, it works. And in a programming language, this is because conditioning can in particular set a variable to a given value, and thus conditioning on the value of C will change the distribution for C. But correlation is more than just the symmetric version of causality.

In the previous definition, we had access to a generative model written as a program, and we were able to detect causality and correlation by setting or conditioning. However, in the real

world, causal dependency is typically hard to observe, whereas correlation can be observed from data, by definition. That is why, when you have no model and you can only make observations, a typical mistake could be assuming that C is causally dependent from A, try to run a naive experiment, observe that C is statistically dependent from A and conclude that your assumption was correct.

In a world of machine learning, this means that it can be hard to actually find the causal model. However, if you have a causal model, you may be interested in analyzing it.