Statistical Learning Methods
Turing.jl

Tuesday 27th April, 2021

Agenda

Bayesian vs. frequentist statistics

Frequentist statistics
If the probability of getting heads in a single coin toss is 0.5, that means after a single coin toss we can expect to get one-half of a head of a coin (with two tosses we can expect to get one head, which makes more sense).
Bayesian statistics
Concerned with how probabilities represent how uncertain we are about a piece of information. In Bayesian terms, if the probability of getting heads in a coin toss is 0.5, that means we are equally unsure about whether we'll get heads or tails.

When you're quantifying your belief that your favorite candidate will win the next election, the Bayesian interpretation makes much more sense (Kurt 2019).

Bayes' Theorem

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

A - belief
B - observations

Turing - What?

Turing.jl is a probabilistic programming language library for Julia. *Turing* components are available under:

1. `https://github.com/TuringLang/Turing.jl` (source code)
2. `https://turing.ml` (documentation)
3. `https://github.com/TuringLang/` (Turing environment landing page)

Turing - Who?

Created and managed by Hong Ge (University of Cambridge, Cambridge Machine Learning Group)

Important contributor is Zoubin Ghahramani (University of Cambridge, Uber, Alan Turing Institute)

Library creation was supported by Alan Turing Institute and donations from Google and Microsoft Research.Since Turing is an open source project, anyone can participate and provide contribution (63 contributors)

Turing - Why? p.1

Remark

Traditional machine learning can be summarised as a collection of thousands of learning algorithms. In contrast, model-based machine learning approach seeks to create solution (model) tailored to each new problem.

Model-based machine learning framework emerged from three key ideas:

- the adoption of a Bayesian viewpoint,
- the use of probabilistic graphical model,
- application of efficient inference algorithms.

Turing - Why? p.2

The core idea is that problem is defined in generic form of a *model* - set of assumptions about the world expressed in a probabilistic graphical format with all the parameters and variables expressed as random component.

Example

Joint probability $P(\mu, X)$ over the whole model in Figure 1 is factorized as:

$$P(\mu, X) = P(\mu)P(X|\mu)$$

Where μ is the model parameter and X are the set of observed variables

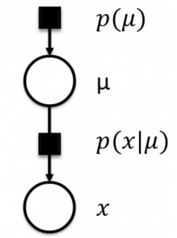

Figure: Probabilistic factor graph

Turing - Why? p.3

Alert

However, for each new application it is necessary to derive the inference method, e.g. in the form of variational or Markov chain Monte Carlo (MCMC) algorithm, and then implement it in application-specific code.

Probabilistic programming languages (Wood, Meent, and Mansinghka 2015; Goodman et al. 2008) aim to fill this gap by providing a flexible framework for defining probabilistic models and automating the model learning process using *generic* inference engines.
This enables researchers to focus on designing a suitable model using their insight and expert knowledge, and accelerates the iterative process of model modification.

Macros

Turing.jl PPL main extension to the Julia language consist of two macros:

1. `@model` macro for defining probabilistic models. It can use arbitrary Julia code, but to ensure correctness of inference it should not have external effects or modify global state. To help avoid bugs for mutable heap-allocated objects, Turing provide safe datatype `TArray`.
2. $\sim$ macro specify distributions of random variables such as `x` $\sim \mathrm{distr}$ where `x` is a symbol and `distr` is a distribution. Turing also supports vectorized variables with following syntax: `rv = Vector(10); rv` $\sim [\mathrm{Normal}(0, 1)]$

Example model

Listing 1: Model and sampling defined in Turing PPL

```julia
# Define a simple Normal model with unknown mean and variance.
@model gdemo(x, y) = begin
  s ~ InverseGamma(2, 3)
  m ~ Normal(0, sqrt(s))
  x ~ Normal(m, sqrt(s))
  y ~ Normal(m, sqrt(s))
end
# Particle Gibbs sampler
c1 = sample(gdemo(1.5, 2), PG(10), 1000)
# Hamiltonian Monte Carlo sampler
c2 = sample(gdemo(1.5, 2), HMC(0.1, 5), 1000)
```

Sampling from The Prior

Turing allows to sample from a declared model's prior by calling the model without specifying inputs or a sampler.

Listing 2: Unconditional distribution sampling

```
# Samples from p(x,y)
g_prior_sample = gdemo()
g_prior_sample()
Output: (0.685690547873451, -1.1972706455914328)
```

Sampling from The Posterior

Values that are *missing* are treated as parameters to be estimated. This can be useful when simulating draws for that parameter, or while sampling from a conditional distribution.

Listing 3: Conditional distribution sampling

```
# Equivalent to sampling p( x1 | x2 = 1.5).
model = gdemo(missing, 1.5)
c = sample(model, HMC(0.01, 5), 500)
```

Generative models

Turing models can also be treated as generative model.

Listing 4: Generative model

```julia
# Declare a model with a default value.
@model generative(x = Vector{Real}(undef, 10)) = begin
    s ~ InverseGamma(2, 3)
    m ~ Normal(0, sqrt(s))
    for i in 1:length(x)
        x[i] ~ Normal(m, sqrt(s))
    end
    return s, m
end
# Generate a vector of 10 values every sampler iteration.
generated = sample(generative(), HMC(0.01, 5), 1000)
xs = generated[:x]
```

Requirements for custom distributions

Turing supports custom distributions while given requirements are met:

1. Custom distribution is a subtype of a corresponding distribution type in the `Distributions.jl` package
2. Sampling and Evaluation of the log-pdf is implemented for the distribution
3. Functions for domain transformation are required by multivariate or matrix-variate distributions
4. Appropriate function for vectorization support should be defined. Vectorized syntax requires `rand` and `logpdf` to be called on multiple data points at once.

Implementation of custom distribution

1.

```
struct Flat <: ContinuousUnivariateDistribution
...
end
```

2.

```
rand(rng::AbstractRNG, d::Flat) = rand(rng)
logpdf(d::Flat, x::Real) = zero(x)
```

3.

```
Distributions.minimum(d::Flat) = -Inf
Distributions.maximum(d::Flat) = +Inf
```

4.

```
logpdf(d::Flat, x::AbstractVector{<:Real}) = zero(x)
```

MCMC Introduction

Turing inference engine is based on multiple Markov chain Monte Carlo (MCMC) methods which comprise a class of algorithms for **sampling** from a probability distribution.
By constructing a Markov chain that has the desired distribution as its equilibrium distribution, one can obtain a sample of the desired distribution by recording states from the chain. The more steps that are included, the more closely the distribution of the sample matches the actual desired distribution.

Example

Various algorithms exist for constructing the Markov chain e.g the *Metropolis–Hastings* algorithm.

Metropolis–Hastings algorithm p.1

Initialization: Choose an arbitrary point x_0 to be the first sample, and choose an arbitrary probability density $g(x|y)$ $(Q(x|y))$ that suggests a candidate for the next sample value x, given the previous sample value y. The function g is referred to as the *proposal density* or *jumping distribution*.

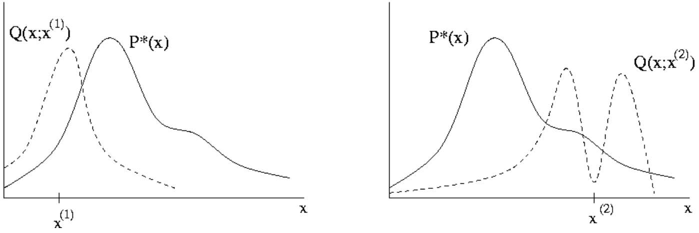

Figure: Picking the next point from distribution Q to which the random walk might move.

Metropolis–Hastings algorithm p.2

For each iteration t:

- **Generate** : Generate a candidate x' for the next sample by picking from the distribution $g(x'|x_t)$.
- **Calculate** : Calculate the *acceptance ratio* $\alpha = f(x')/f(x_t)$, which will be used to decide whether to accept or reject the candidate. Because f is proportional to the density of P, we have that $\alpha = f(x')/f(x_t) = P(x')/P(x_t)$.
- **Accept or Reject**:
    1. Generate a uniform random number u on $[0,1]$.
    2. If $u \leq \alpha$ *accept* the candidate by setting $x_{t+1} = x'$,
    3. If $u > \alpha$ *reject* the candidate and set $x_{t+1} = x_t$, instead.

Integral evaluation

MCMC samplers can be used to evaluate an integral over given variable, as its expected value or variance.

Remark

Equation 1 presents conditional probability formula based on Bayes Theorem, where D is observed data and θ represents model parameters. Monte Carlo methods are often used to solve the integral.

$$P(x|D) = \int_{\theta} P(x|D)P(\theta|D)d\theta \tag{1}$$

Correlation problem

Alert

Whereas the random samples of the integrand used in a conventional Monte Carlo integration are statistically independent, those used in Markov chain Monte Carlo methods are autocorrelated.

Correlations of samples introduces the need to use the *Markov chain central limit theorem* when estimating the error of mean values.

Curse of dimensionality

Alert

While MCMC methods were created to address multi-dimensional problems better than simple Monte Carlo algorithms, when the number of dimensions rises they too tend to suffer the *curse of dimensionality*: the regions of higher probability tend to stretch and get lost in an increasing volume of space that gives little contribution to the desired integral.

One way to address this problem could be shortening the steps of the walker, so that it doesn't continuously try to exit the highest probability region, though this way the process would be highly autocorrelated and quite ineffective (i.e. many steps would be required for an accurate result).

Computation graph based inference

One challenge of performing inference for probabilistic programs is about how to obtain the computation graph between model variables.

Example

For certain probabilistic programs, it is possible to construct the computation graph between variables through static analysis e.g. in BUGS language (Lunn et al. 2000).

Alert

Computation graph underlying a probabilistic program needs to be *fixed* during inference time. For programs involving *stochastic branches*, this requirement may not be satisfied.

Hamiltonian Monte Carlo based inference p.1

For the family of models whose log probability is **pointwise computable and differentiable**, Hamiltonian Monte Carlo (Neal 2012) algorithm (extending Metropolis algorithm with Hamiltonian dynamics) may be used to produce *minimally correlated* proposals.

Within HMC, the slow exploration of the state space, originating from the difiusive behavior of MH's random-walk proposals, is avoided by augmenting the state space of the target distribution $p(\theta)$ with a d-dimensional vector r. The resulting joint distribution is as follows:

$$p(\theta, r) = p(\theta)p(r), \quad p(r) = \mathcal{N}(0, I_D) \tag{2}$$

Hamiltonian Monte Carlo based inference p.2

HMC operates through alternating between two types of proposals. The first, randomizes r (momentum variable). The second changes both θ and r using simulated Hamiltonian dynamics as define by

$$H(\theta, r) = E(\theta) + K(r) \tag{3}$$

where $E(\theta) = -logp(\theta)$ and $K(r)$ is a 'kinetic energy' such as $K(r) = r^T r/2$. These two proposals would produce samples from the joint distribution $p(\theta, r)$ which is separable to the desired distribution $p(\theta)$.

Alert

HMC cannot sample from distributions that are not differentiable or involving discrete variables.

Blocked Gibbs sampling

Turing provide *composable* sampling interface that can combine many sampling algorithms for each variable/parameter separately.

Blocked Gibbs algorithm `Gibbs()` ensemble of both Particle Gibbs and Hamiltonian Monte Carlo samplers:

```
sample(model(par1, par2,...), Gibbs(1000,PG(10,2,:z),HMC
    (2,0.1,5,:ϕ,:θ)))
```

Model variables were split into two subsets and assigned to two different samplers. Differentiable variables θ and ϕ were assigned to HMC engine and categorical z to PG engine.

Available samplers

Turing supports a wide range of sampling algorithms including Hamiltonian Monte Carlo, particle Gibbs, No-U-Turns, sequential Monte Carlo, and interactive particle MCMC (Table 1).

Sampler	**Discrete variables?**	**Require gradients?**	**Support universal programs?**	**Composable?**
HMC	No	Yes	No	Yes
NUTS	No	Yes	No	Yes
IS	Yes	No	Yes	No
SMC	Yes	No	Yes	No
PG	Yes	No	Yes	Yes
PMMH	Yes	No	Yes	Yes
IPMCMC	Yes	No	Yes	Yes

Table: Supported Monte Carlo algorithms in Turing

Dependencies

Turing rely on multiple utility packages from Turing environment (`TuringLang`). Some of them includes:

- `MCMCChains.jl` - utility functions for MCMC simulations
- `Bijectors.jl` - library for transforming distributions and constrained random variables
- `AdvancedHMC.jl` - implementation of HMC algorithms
- `AdvancedMH.jl` - library for Metropolis-Hastings algorithms
- `Libtask.jl` - task copying for Turing
- `DistributionsAD.jl` - enable automatic differentiation (AD) of the logpdf function from `Distributions.jl` using the packages `Tracker.jl` and `ForwardDiff.jl`.

Related packages

Turing can be combined with Julia's machine learning packages e.g. `Flux.jl` to add bayesian flavor to standard models available in the packages.

Particular Julia libraries depends on Turing in their calculations.

Example

`DiffEqBayes.jl` library (`DifferentialEquations.jl` family) use Turing to estimate parameters in differential equations.

Automatic differentiation

Simulating Hamiltonian dynamics for the Hamiltonian Monte Carlo sampler step requires the gradient of $logp(\theta|\mathbf{z}_{1:N}, \gamma)$. These gradients can be obtained through automatic differentiation (AD) techniques (Baydin et al. 2015).

MCMCChains.jl

In Turing, the following statistics for each MCMC chain can be calculated by calling the `describe` function:

- mean,
- standard deviation,
- naive standard error,
- Monte Carlo standard error (MCSE),
- effective sample size (ESS),
- quantiles of 2.5%, 25.0%, 50.0%, 75.0% and 97.5%.

In addition, highest posterior density intervals can be computed by `hpd`, cross-correlations by `cor`, lag-autocorrelations by `autocor`, state space change rate (per iteration) by `changerate` and deviance information criterion by `dic`.

References I

Lunn, David J. et al. (Oct. 2000). "WinBUGS - A Bayesian modelling framework: Concepts, structure, and extensibility". In: *Statistics and Computing* 10.4, pp. 325–337. ISSN: 1573-1375.

Goodman, Noah et al. (Jan. 2008). "Church: A language for generative models". In: vol. 2008, pp. 220–229.

Neal, Radford M. (2012). *MCMC using Hamiltonian dynamics*. arXiv: 1206.1901 [stat.CO].

Baydin, Atilim Gunes et al. (2015). *Automatic differentiation in machine learning: a survey*. arXiv: 1502.05767 [cs.SC].

Wood, Frank, Jan Willem van de Meent, and Vikash Mansinghka (2015). *A New Approach to Probabilistic Programming Inference*. arXiv: 1507.00996 [stat.ML].

References II

Kurt, Will (June 2019). *Bayesian Statistics the Fun Way*. No Starch Press.