Probabilistic Programming

Investigating optimisation-based inference methods

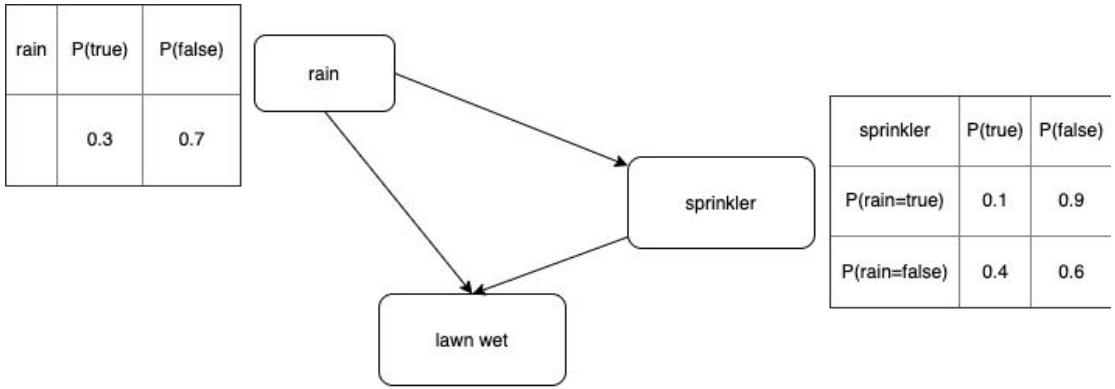
Karl-Oskar Rikås D16

Probabilistic Programming Languages (PPLs)

- Current methodology: describe a model in mathematical notation, then implement it from scratch. Model and inference algorithm ends up being tightly coupled.
- Goal of PPLs: take ideas from Computer Science (Software Engineering) to make it faster and easier to develop these models.
- How? Decouple the model description from the inference algorithm.

- Model is described as code in a custom language or embedded DSL, inference is implemented as an interpreter or compiler.

Example of a model

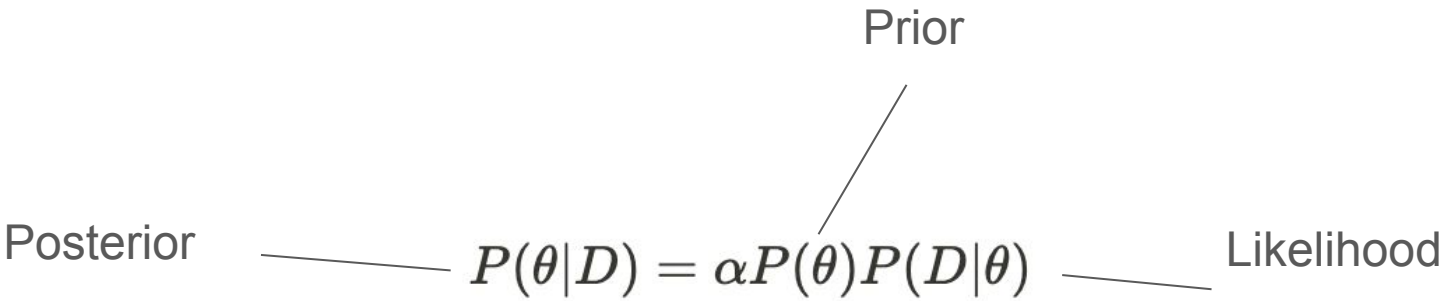


rain	P(true)	P(false)
	0.3	0.7

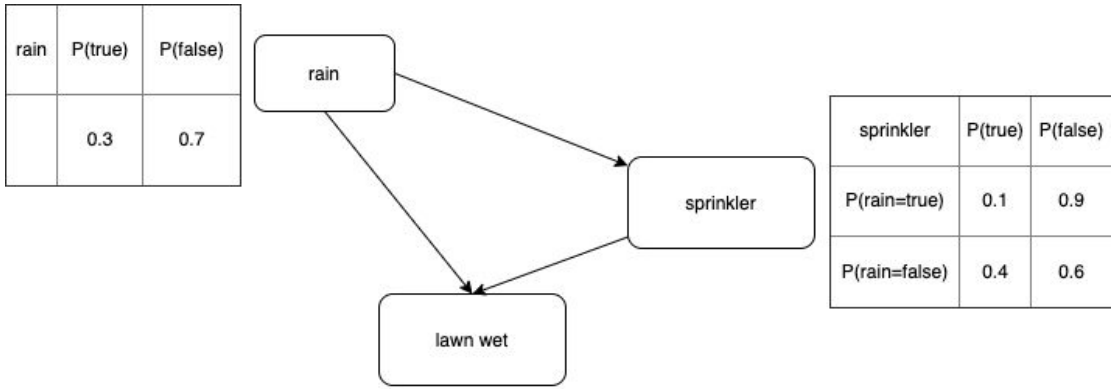
sprinkler	P(true)	P(false)
P(rain=true)	0.1	0.9
P(rain=false)	0.4	0.6

lawn wet	P(true)	P(false)
P(rain=true, sprinkler=true)	0.99	0.01
P(rain=false, sprinkler=false)	0.70	0.30
P(rain=false, sprinkler=true)	0.90	0.10
P(rain=false, sprinkler=false)	0.01	0.99

Bayes' theorem



Given that the lawn is dry, did it rain?



rain	P(true)	P(false)
	0.3	0.7

sprinkler	P(true)	P(false)
P(rain=true)	0.1	0.9
P(rain=false)	0.4	0.6

lawn wet	P(true)	P(false)
P(rain=true, sprinkler=true)	0.99	0.01
P(rain=false, sprinkler=false)	0.70	0.30
P(rain=false, sprinkler=true)	0.90	0.10
P(rain=false, sprinkler=false)	0.01	0.99

P(rain=true | lawn wet=false)

```haskell
-- has it rained? given the lawn is not wet
-- bernoulli p: true (1-p): false
hasRained :: (MonadSample m, MonadInfer m) => m Bool
hasRained = do
  rain <- bernoulli 0.3       -- Prior
  sprinkler <- bernoulli $
    if rain then 0.1 else 0.4
  lawnWet <- bernoulli $      -- Likelihood
    case (rain, sprinkler) of
      (True, True) -> 0.99
      (True, False) -> 0.70
      (False, True) -> 0.90
      (False, False) -> 0.01
  condition $ not lawnWet     -- re-scale prior based on likelihood
  return rain
```

Based on example from Ścibior, A., Kammar, O. and Ghahramani, Z., 2018. Functional programming for modular Bayesian inference. *Proceedings of the ACM on Programming Languages*, *2*(ICFP), pp.1-29.

```haskell
-- draw 1000 samples using Metropolis-Hasting (MCMC)
samples :: IO [Bool]
samples = sampleIOfixed . prior $ mh 1000 hasRained

demo :: IO ()
demo = do
  xs <- samples
  let ys = [if x then 1 else 0 | x <- xs]
  putStrLn "How likely to have rained?"
  putStrLn $ "mean\t" ++ (show $ mean ys)
  putStrLn $ "std \t" ++ (show $ std ys)

mean :: [Double] -> Double
mean xs = …

std :: [Double] -> Double
std xs = …
```

```
Presentation> demo
How likely to have rained?
mean    0.1838161838161838
std     0.38752783406849745
```

More advanced models

- Bayesian Linear and Logistic Regression (regression, classification)
- Hidden Markov Model (robot localisation and more, as. 3 in the AI course)
- Topic models (used for NLP to categorise collection of documents)
- Gaussian process models (time-series prediction)
- And many more...

Functional Programming for Modular Bayesian Inference

ADAM ŚCIBIOR, University of Cambridge, UK and MPI for Intelligent Systems, Germany
OHAD KAMMAR, University of Oxford, UK
ZOUBIN GHAHRAMANI, University of Cambridge, UK and Uber AI Labs, USA

Why a PPL in a FP language?

- Leverage same benefits of FP for developing models.
- Modularity
  - In ‘monad-bayes’ some inference algorithms are simply compositions of simpler ones. For example Particle Independent Metropolis Hastings is composed by Sequential Monte Carlo and Metropolis Hasting. I.e. potentially make it easier to implement inference methods.
  - Also harder, because most literature on algorithms assumes imperative programming style.
- Correctness
  - “Correct by construction” by leveraging static typing with an advanced type-system.
  - Very recent research shows it is possible to check certain statistical properties statically using “Trace Types” (i.e. at compile time). Not yet implemented in ‘monad-bayes’.

Lew, Alexander K., Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. “Trace Types and Denotational Semantics for Sound Programmable Inference in Probabilistic Languages.” Proceedings of the ACM on Programming Languages 4, no. POPL (December 20, 2019): 19:1–19:32.

Project

- Future work in the paper: integrate and implement gradient-based inference methods, PyTorch suggested for AD.

- Goal: investigate AD, implement HMC or VI and integrate into the library ‘monad-bayes’.
- Result: investigated AD, PyTorch bindings seemed appropriate, partial implementation of VI in Haskell.

Bayesian inference

- Compute the posterior.
- For complicated models (most real applications) computing the posterior is intractable (exponential time complexity).
- Instead we use algorithms that approximate the posterior.
- Usually done with some kind of Monte Carlo algorithm.

$$P(\theta|D) = \alpha P(\theta)P(D|\theta)$$

Automatic differentiation (AD)

- Used to compute gradients for the inference algorithms: Hamiltonian Monte Carlo and **Variational Inference**.
- Not the same as numeric or symbolic differentiation, computes the exact derivative at a given point.

Dual numbers

$$\epsilon^2 = 0$$

$$f(x) = x^2 + x + 1$$

$$f(2) = 7,\ f'(2) = 5$$

$$f(2+\epsilon) = (2+\epsilon)^2 + (2+\epsilon) + 1 = 2^2 + 4\epsilon + e^2 + 3 + \epsilon = 7 + 5\epsilon$$

```haskell
data Dual = Dual Double Double
    deriving (Show, Eq)

instance Num Dual where
  (+) (Dual a a') (Dual b b') = Dual (a + b) (a' + b')

instance Floating Dual where
  (**) (Dual a a') (Dual b b')
    = Dual (a ** b) (b * a**(b - 1))

instance Fractional Dual where
  fromRational n              = Dual (fromRational n) 0

dual :: Double -> Dual
dual x = Dual x 1

value :: Dual -> Double
value (Dual x _) = x

deriv :: Dual -> Double
deriv (Dual _ x) = x
```

Based on tutorial https://www.danielbrice.net/blog/automatic-differentiation-is-trivial-in-haskell/

```haskell
f :: Floating a => a -> a
f x = (x**2) + x + 1

f' :: Double -> Double
f' x = deriv . f $ dual x

Presentation> f 2
7.0
Presentation> f' 2
5.0
```

AD implementations

- Forward-mode is easy to implement in a language that supports overloaded operators (ad-hoc polymorphism).
- Too slow in for real world applications, linear time complexity with respect to model parameters.
- In practice a more advanced algorithm called reverse-mode AD is used, usually implemented in C/C++ and called using bindings from higher-level languages.

- Side note: AD is used in neural network libraries like PyTorch and Tensorflow to compute gradients.

Variational inference (VI)

- Turns Bayesian inference into an optimisation problem. Normally VI requires manually deriving an optimisation routine for a given model.
- Using AD this routine can be derived automatically given a differentiable model.

1. Transform the model into a representation that is possible to optimise.
2. Sample variables from normal distributions using an existing inference method.
3. Optimise the resulting function using Stochastic Gradient Descent (or something more advanced like Adam).

Demo: Bayesian Linear Regression

Not finished on time.

Future work

- Finish the implementation and test it.
- Benchmark the implementation and see if it actually improves performance.
- Generalize the ADVI implementation and look into integration into ‘monad-bayes’.
- Look into applications.

- Even further in the future: look into “Trace Types” paper mentioned earlier in the presentation, promises higher correctness guarantees and improved performance for ‘monad-bayes’.

Thanks for listening