Probabilistic programming

Jure Demšar

Problems of the two approaches

- The two approaches (analytical, brute force) do not scale!
- Useful only for the simplest models.
- Solution: **MCMC** and **probabilistic programming**.
- We only have one very minor limitation: likelihood and prior must be easily calculated (up to a multiplicative constant).

Imperative programming

- Under the term programming languages, we generally think about imperative (general purpose) programming languages (e.g., c++, python, R ...).

- In imperative programming we are given a set of inputs (input parameters or input data).

- Our goal is to develop an algorithm (define a step-by-step procedure) that uses given inputs to produce desired outputs

Probabilistic programming

Wikipedia: “*Probabilistic programming is a programming paradigm in which probabilistic models are specified and inference for these models is performed automatically.*”

Probabilistic programming language (PPL)

- PPLs are programming languages designed specifically for developing probabilistic models and inferring from them.
- They enable us to focus on the modelling part while they take care of the mathematical and computational problems required for statistical inference.
- Under the hood, PPLs are developed via traditional, imperative programming.

Imperative vs. probabilistic programming

- With imperative (traditional) programming your task is to develop an algorithm (a set of steps) that generates some outputs based on given inputs. In other words, we are developing a process that generates some data.

- With probabilistic programming we are going the other way, we are given the “output” (some data) and our task is to “reverse engineer” the process (or algorithm) that generated the data.

Imperative vs. probabilistic programming

```r
# Generate n Bernoulli variables
bernoulli <- function(n, p) {
  # Create and empty vector
  x <- c()

  # Loop n times
  for (i in 1:n) {
    # Draw from uniform(1) and append 0 or 1
    if (runif(1) > p) {
      x <- c(x, 0)
    } else {
      x <- c(x, 1)
    }
  }

  # return the vector
  return(x)
}
```

$$y_1, \ldots, y_n \qquad y_i \in 0, 1.$$

$$y_1, \ldots, y_n | \theta \sim_{iid} Bernoulli(\theta).$$

$$\theta \sim Beta(1, 1).$$

Imperative vs. probabilistic programming

- We need both!
- We need imperative programing to load and prepare the input data for our probabilistic program.
- We need PPLs to design the model and do the statistical inference.
- After the inference is completed, we again need imperative programing to diagnose the outputs, analyze results, prepare visualizations ...

All models are wrong, but some are useful

- Problems we are solving with probabilistic programming are in its essence unsolvable. There are infinite ways to generate the given data and no mechanism to prove which of them is correct.

- We are not developing a perfect model, but a useful model!

- Probabilistic model development is an iterative process guided by knowledge, experience and intuition.

- Do not get discouraged if your initial models fail in a spectacular manner.

Example

```
# Generate a sequence of 0, 1 values
01_sequence <- function(n) {
  # Create and empty vector
  x <- c()

  # Loop n times
  for (i in 1:n) {
    # get temperature in LJ
    t_lj <- get_lj_temp()

    if (t_lj > 15) {
      # output 0
      x <- c(x, 0)
    } else {
      # get air pressure in LJ
      p_lj <- get_air_pressure_lj()
      if (p_lj < 1000) {
        # flip a coin for output 0 or 1
          if (runif(1) > 0.5) {
          x <- c(x, 0)
        } else {
          x <- c(x, 1)
        }
      } else {
        # output 1
        x <- c(x, 1)
      }
    }
  }

  # return the vector
  return(x)
}
```

Stan

- Stan is a state-of-the-art platform for statistical modelling and high-performance statistical computation.
- It is one of the most popular probabilistic programming languages as thousands of users rely on it for statistical modelling in many domains. It has an excellent support forum and great documentation.
- Stan's main sampling algorithm is called the No-U-Turn Sampler (NUTS), a variant of Hamiltonian Monte Carlo.
- Pairs well with many popular programming languages (e.g. R, python ...).

Basic variable types in Stan

```
int n; // integer number

real r; // real number

// we can specify lower and upper bounds for almost all types
real<lower=0> sigma;
real<lower=0,upper=1> success_rate;

// array, Stan starts indexing from 1
int a[n];
real b[n];
```

Basic variable types in Stan

```
// 2D array matrix [rows, columns]
int A[n, n];

// vector (real number)
// computationly vectors are more efficient then arrays
vector[n] v;

// matrix (real number)
// just like vectors, matrices are more efficient then arrays
matrix[n, n] M;
```

Structure of Stan models

- Stan models are usually stored in `.stan` files and are composed of blocks.

- There are 7 types of blocks:
  - functions,
  - **data**,
  - transformed data,
  - **parameters**,
  - transformed parameters,
  - **model**,
  - generated quantities.

The data block

```
data {
  // input data definition
  ... declarations ...
}
```

- Declaration of variables which represent the input data.

- Putting constraints on the variables by specifying the lower and upper bounds (`<lower=0,upper=1>`) is a safety net for us so we do not provide Stan with erroneous data.

The parameters block

```
parameters {
  // definition of model's parameters
  ... declarations ...
}
```

- Declaration of parameters we are interested in estimating.
- From a user's perspective, the parameters in the program block are the parameters being sampled by Stan. Their values will be passed back to the interface programming language.
- Lower and upper constraints define the minimal and maximal value which can be accredited to the parameter.

The model block

```
model {
  // model definition
  ... declarations ...
  ... statements ...
}
```

- Variable declaration is optional, defined variables are local and are not part of the output.
- Statements in this block define the probabilistic model.
- We use probability distributions as the core building blocks in our modelling process.

Distributions

- Bernoulli distribution (`y ~ bernoulli(theta);`), where **`y`** is a vector of successes (1) and failures (0) and **`theta`** is the probability of a success.
- Beta distribution (`y ~ beta(alpha, beta);`), where **`y`** is a vector of real numbers between 0 and 1 and **`alpha`** and **`beta`** are the parameters of the beta distribution.
- Normal distribution (`y ~ normal(mu, sigma);`), where **`y`** is a vector of real numbers, **`mu`** is the mean and **`sigma`** the standard deviation.
- Gamma distribution (`y ~ gamma(alpha, beta);`), where **`y`** is a vector of positive real numbers and **`alpha`** and **`beta`** are the parameters of the beta distribution.
- Obviously, Stan supports many other probability distribution, for a complete list consult Stan's documentation.

Additional variable types in Stan

```
// simplex is a vector of positive real numbers that sum up to 1
simplex[n] s;

// unit vector - a vector of real values with Euclidian length of one
unit_vector[n] uv;

// ordered vectors contains ascendingly ordered real numbers
ordered[n] o;

// positive ordered vector contains ordered positive real numbers
positive_ordered[n] po;
```

Additional variable types in Stan

```
// row vector
row_vector[n] rv;

// covariance matrices - symmetric and positive definite
cov_matrix[K] Omega;

// correlation matrices - symmetric and positive definite
// has entries between −1 and 1 and has a unit diagonal
corr_matrix[3] Sigma;
```

Additional variable types in Stan

```
// Cholesky factors of covariance matrices
cholesky_factor_cov[4] L;

// Cholesky factors of correlation matrices
cholesky_factor_corr[K] L;
```

Conditional statements

```
if (condition1)
  statement1
else if (condition2)
  statement2
// a comment
else if (conditionN-1)
  statementN-1
/*
a comment block
*/
else
  statementN
```

For loops

```
// this is a for loop
for (n in 1:N) {
  y[n] ~ bernoulli(theta);
}

// Stan is quite smart, as you saw in our first example
// the above can be simplified (presuming y is a vector or an array)
y ~ bernoulli(theta);
```

“Foreach” loops

```
// if items is a container (vector, row vector, matrix, or array)
for (i in items) {
  ... do something with i ...
}

// for 2D arrays we require a double loop
// we can traverse matrices in a single “foreach” loop
// elements in a matrix are visited in a column-major order
real array[2, 3];
for (i in array)
  for (j in i)
    ... do something with y ...
```

“Foreach” loops

```
// if items is a container (vector, row vector, matrix, or array)
for (i in items) {
  ... do something with i ...
}

// for 2D arrays we require a double loop
// we can traverse matrices in a single “foreach” loop
// elements in a matrix are visited in a column-major order
real array[2, 3];
for (i in array)
  for (j in i)
    ... do something with y ...
```

While statements, break and continue

```
while (condition) {
  if (x[n] >= 0) continue;
  // if above holds the line below will be skipped due to continue
  sum += x[n];
}

while (condition) {
  if (x[n] >= 0) break;
  // if above is true, while loop will exit
  sum += x[n];
}
```

While statements, break and continue

```
while (condition) {
  if (x[n] >= 0) continue;
  // if above holds the line below will be skipped due to continue
  sum += x[n];
}

while (condition) {
  if (x[n] >= 0) break;
  // if above is true, while loop will exit
  sum += x[n];
}
```

The functions block

```
functions {
  // custom user-defined functions
  ... function declarations and definitions ...
}
```

- This block is placed at the top of your Stan program.

- In it you can code up any custom functions that will help you in your work.

The transformed data block

```
transformed data {
  // mathematical transformations of the data
  ... declarations ...
  ... statements ...
}
```

- This block is used to perform any transformation on the input data from the `data` block.
- After transformations (e.g., normalization, standardization ...) are completed, Stan checks if transformed data meets the boundaries, we set on declared variables. If we violated our own constraints Stan errors out.

The transformed parameters block

```
transformed parameters {
  // mathematical transformations of model's parameters
  ... declarations ...
  ... statements ...
}
```

- This block servers a similar purpose as the **`transformed data`** block, except that this time we are transforming model's parameters.
- Transformed parameters are part of model's output.
- After statements are executed, Stan check the constraints we provided. If the parameter does not meet the constraints its value will be rejected.

The generated quantities block

```
generated quantities {
  // additional generated outputs
  ... declarations ...
  ... statements ...
}
```

- This block is rather different then other blocks as nothing in it affects the sampled parameter values.
- This block is executed after a sample is generated, in it we can use model's variables to generate custom quantities we are interested in (e.g. posterior event probabilities, comparisons between parameters, log likelihoods ...).

Log posterior probability

- In the Bayesian setting a probabilistic program is a description of how to compute the posterior distribution. The essence of computation in Stan is dealing with the logarithm of the posterior probability density.

- In practice `y[i] ~ normal(mu, sigma);` multiplies the current posterior probability by the density of the `normal` distribution at `y[i]`.

- This is the same as an increment of current log-probability by the log-density of the normal distribution at `y[i]`.

- Indeed, we can replace `y[i] ~ normal(mu, sigma);` with `target += normal_lpdf(y | mu, sigma)`.

Hands on examples

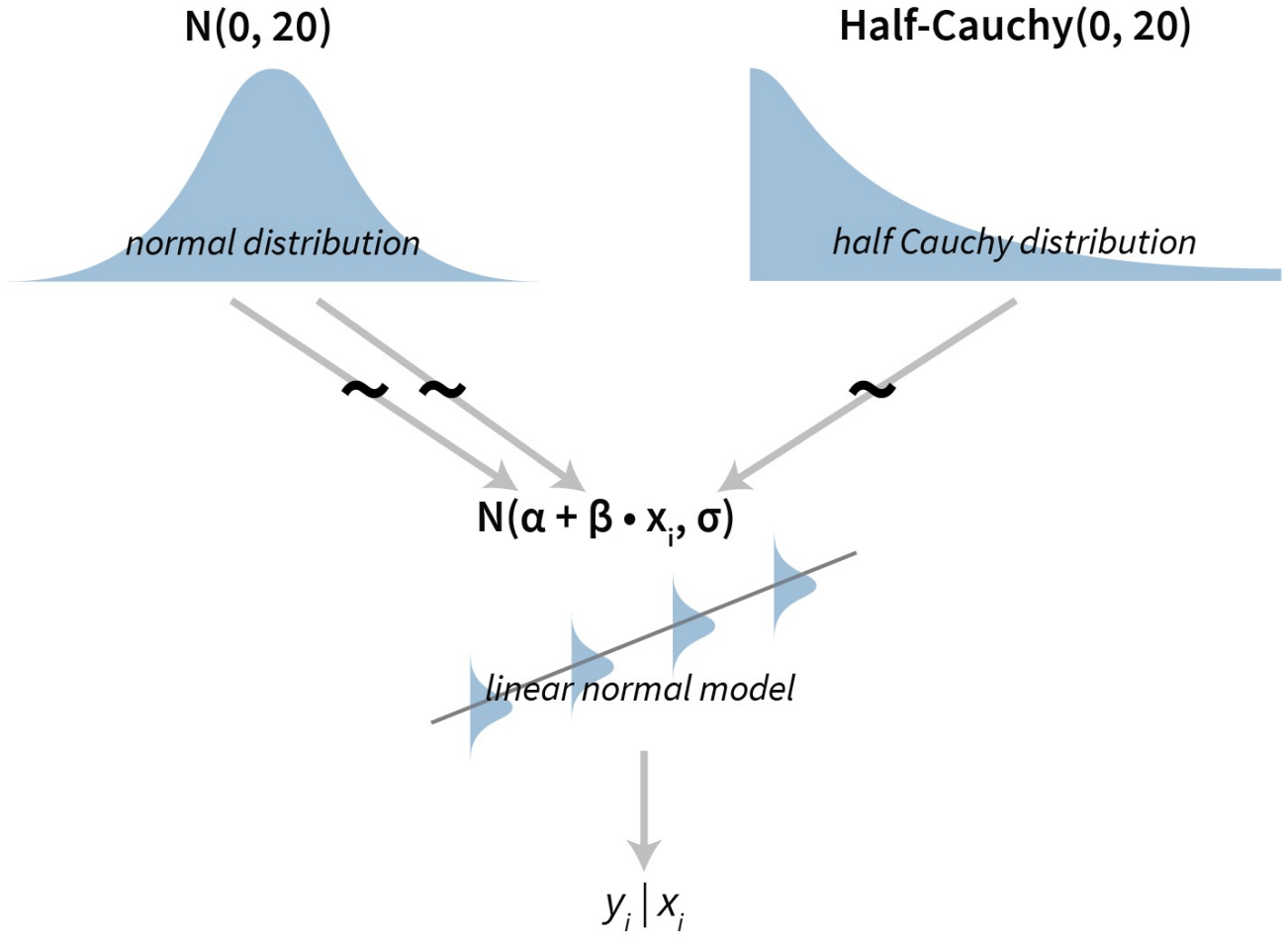
Presenting models

$$y_i \sim N(a + b \cdot x_i, \sigma),$$

$$a \sim N(0, 20),$$

$$b \sim N(0, 20),$$

$$\sigma \sim \text{Half-Cauchy}(0, 20).$$


N(0, 20)
Half-Cauchy(0, 20)
normal distribution
half Cauchy distribution
~ ~
~
N(α + β • x_i, σ)
linear normal model
y_i | x_i

Questions, problems …

jure.demsar@fri.uni-lj.si