OpenPPL Documentation

Release 0.1.0

All contributors

October 25, 2015

Contents

This document proposes the design of *OpenPPL*, an open source probabilistic programming framework implemented as a domain-specific language on top of Julia. This document discusses the design of basic language constructs for model specification, query, as well as directives to control algorithmic choices in inference.

Contents:

CHAPTER 1

Basics

OpenPPL is a domain-specific language built on top of Julia using macros. The language consists of two parts: *model specification* and *query*. In particular, a *model specification* formalizes a probabilistic model, which involves declaring variables and specifying relations between them; while a *query* specifies *what is given* and *what is to be inferred*.

1.1 Terminologies

Here is a list of terminologies that would be involved in the description.

Variable A variable generally refers to an entity that can take a value of certain type. It can be a random variable directly associated with a distribution, a deterministic transformation of another variable, or just some value given by the user. The value of a variable can be given or unknown.

Constant A value that is fixed when a query is constructed and fixed throughout the inference procedure. A constant is typically used to represent vector dimensions, model sizes, and hyper-parameters etc. Note that model parameters are typically considered as variables instead of constants. For example, a learning process can be formulated as a query that solves the parameter given a set of observations, where the parameter values can be iteratively updated.

Domain The domain of a variable refers to the set of possible values that it may take. Any Julia type (e.g. `Int`, `Float64`) can be considered as a domain that contains any value of that type. This package also supports array domains and restrictive domains that contain only a subset of a specific type. Here are some examples:

```
1..K              # integers between 1 and K
0.0..Inf          # all non-negative real values
Float64^n         # n-dimensional real vectors
Float64^(m,n)     # matrices of size (m, n)
(0.0..1.0)^m      # m-dimensional real vectors with all components in [0., 1.]
```

Distribution From a programmatic standpoint, a distribution can be considered as a *stochastic* function that yields a random value in some domain. A distribution can accept zero or more arguments. A distribution should be *stochastically pure*, meaning that it always outputs the same value given the same arguments and the same state of the random number generator. Such purity makes it possible to reason about program structure and transform the model from one form to another.

Factor A factor is a pure real-valued function. Here, "pure" means that it always output the same value given the same inputs. Factors are the core building blocks of a probabilistic model. A complex distribution is typically formulated as a set of variables connected by factors. Even a simple distribution (e.g. normal distribution) consists of a factor that connects between generated variables and parameters (which may be considered as a variable with fixed value).

1.2 Getting Started: Gaussian Mixture Model

Here, we take the *Gaussian Mixture Model* as an example to illustrate how we can specify a probabilistic model using this Julia domain-specific language (DSL). A *Gaussian Mixture Model* is a generative model that combines several Gaussian components to approximate complex distributions (e.g. those with multiple modals.). A Gaussian mixture model is characterized by a prior distribution π and a set of Gaussian component parameters $(\mu_1, \Sigma_1), \ldots, (\mu_K, \Sigma_K)$. The generative process is described as follows:

$$\begin{aligned} z_i &\sim \pi \\ x_i | z_i &\sim \mathcal{N}(\mu_{z_i}, \Sigma_{z_i}) \end{aligned}$$

1.2.1 Model Specification

The model specification is given by

```
@model GaussianMixtureModel begin
    # constant declaration
    @constant d::Int    # vector dimension
    @constant n::Int    # number of observations
    @hyperparam K::Int    # number of components

    # parameter declaration
    @param pi :: (0.0..1.0)^K     # prior proportions
    for k in 1 : K
        @param mu[k] :: Float64^d          # component mean
        @param sig[k] :: Float64^(d, d)    # component covariance
    end

    # sample generation process
    for i in 1 : n
        z[i] ~ Categorical(pi)
        x[i] ~ MultivariateNormal(mu[z[i]], sig[z[i]])
    end
end
```

This model specification should be self-explanatory. However, it is still worth clarifying several aspects:

- The macro `@model` defines a model type named `GaussianMixtureModel`, and creates an environment (delimited by `begin` and `end`) for model formulation. All model types created by `@model` is a sub type of `AbstractModel`.
- The macro `@constant` declares `d`, `K`, and `n` as constants. The values of these constants need not be given in the specification. Instead, they are needed upon query. Particularly, to construct a model, one can write

```
mdl = GaussianMixtureModel()
```

  One can *optionally* fix the value of constants through keyword arguments in model construction, as below

```
mdl = GaussianMixtureModel(d = 2, K = 5)
```

  Note: fixing constants upon model construction is generally unnecessary. However, it might be useful to fix them under certain circumstances to to simplify queries or restrict its use. Once a constant is fixed, it need not be specified again in the query.

- The macro `@hyperparam` declares hyper parameters. Hyper parameters are similar to constant technically, except that they typically refer to model configurations that may be changed during cross validation.

- Variables can be defined using the syntax as `variable-name :: domain`. A for-loop can be used to declare multiple variables in the same domain. When the variable domain is clear from the context (e.g. the domain of `z` and `x` can be inferred from where they are drawn), the declaration can be omitted.
- The macro `@param` tags certain variables to be parameters. The information will be used in the learning algorithm to determine which variables are the parameters to estimate.
- The statement `variable-name ~ distribution` introduces a conditional distribution over variables, which will be translated into a factor during model compilation.

1.3 Generic Specification: Finite Mixture Model

The Gaussian mixture model can be considered as a special case in a generic family called *Finite mixture model*. Generally, the components of a finite mixture model can be arbitrary distributions. To capture the concept of *generic distribution family*, we introduce *generic specification* (or *parametric specification*), which can take type arguments.

The specification of the generic finite mixture model is given by

```
@model FiniteMixtureModel{G, ParamTypes} begin
    # constant declaration
    @hyperparam K::Int
    @constant n::Int

    # parameter declaration
    @param pi :: (0.0..1.0)^K    # prior proportions
    for k = 1 : K
        for j = 1 : length(ParamTypes)
            @param theta[k][j] :: ParamTypes[j]
        end
    end

    # sample generation process
    for i in 1 : n
        z[i] ~ Categorical(pi)
        x[i] ~ G(theta[z[i]]...)
    end
end
```

One may consider a generic specification above as a specification template. To obtain a Gaussian mixture model specification, we can use the `@modelalias` macro, as below:

```
@modelalias GaussianMixtureModel FiniteMixtureModel{G, ParamTypes} begin
    @constant d::Int
    @with G = MultivariateNormal
    @with ParamTypes[1] = Float64^d          # component mean
    @with ParamTypes[2] = Float64^(d, d)     # component covariance
end

mdl = GaussianMixtureModel()
```

The `@modelalias` macro allows introducing new constants and specializing the type parameters.

1.4 Queries

In machine learning, the most common queries that people would make include

- learning: estimate model parameters
- prediction: predict the value or marginal distribution over unknown variables, given a learned model and observed variables.
- evaluation: evaluate log-likelihood of observations with a given model
- sampling: draw a set of samples of certain variables

To simplify these common queries, we provide several functions.

1.4.1 Query

Query refers to the task of inferring the value or marginal distributions of unknown variables, given a set of known variables.

```
function query(rmdl::AbstractModel, knowns::Associative, qlist::Array, options)
    set_variables!(rmdl, knowns)
    q = compile_query(rmdl, qlist, options) # this returns a query function q
    return q()  # runs the query function and returns the results
end

function query(rmdl::AbstractModel, knowns::Associative, q)
    infer(rmdl, knowns, q, default_options(rmdl))
end
```

`qlist` is a list of variables or functions over variables that you want to infer. The function `compile_query` actually performs model compilation, analyzing model structure, choosing appropriate inference algorithms, and generating a closure `q`, which, when executed, actually performs the inference.

This `query` function here is very flexible. One can use it for prediction and sampling, etc.

```
# let rmdl be a learned model

# predict the value of z given observation x
z_values = query(rmdl, {:x=>columns(data)}, :z)

# infer the posterior marginal distributions over z given x
z_marginal = query(rmdl, {:x=>columns(data)}, :(marginal(z)))

# you can simultaneously infer only selected variables in a flexible way
r = query(rmdl, {:x=>columns(data)}, {:(z[1]), :(z[2]), :(marginal(z[3]))})

# draw 100 samples of z
samples = query(rmdl, {x:=>columns(data)}, :(samples(z, 100)))
```

Note that inputs to the function are symbols like `:z` or expressions like `:(marginal(z))`, which indicate *what we want to query*. It is incorrect to pass `z` or `marginal(z)` – the value of `z` or `marginal(z)` is unavailable before the inference.

1.4.2 Learning

Learning refers to the task of estimating model parameters given observed data. This can be considered as a special kind of query, which infers the values of model parameters, given observed data.

```
function learn_model(mdl::AbstractModel, data::Associative, options)
    rmdl = copy(mdl)
    set_variables!(rmdl, data)
```

```
    q = compile_query(rmdl, parameters(rmdl), options)
    set_variables!(rmdl, q())
    return rmdl
end

function learn_model(mdl::AbstractModel, data)
    learn_model(mdl, data, default_options(mdl))
end

# learn a GMM, a simple wrapper of learn_model
# suppose data is a d-by-n matrix
rmdl = learn_model(
    GaussianMixtureModel(K = 3, d = size(data,1), n = size(data,2)),
    {:x => columns(data)})
```

In the function `learn_model`, `parameters(rmdl)` returns a list of parameters as the query list. Then the statement `q = compile_query(rmdl, parameters(rmdl), options)` returns a query function `q`, such that `q()` executes the estimation procedure and returns the estimated model parameters. The following example shows how we can use this function to learn a Gaussian mixture model.

```
function learn_gmm(data::Matrix{Float64}, K::Int)
    learn_model(
        GaussianMixtureModel(K = K, d = size(data,1), n = size(data,2)),
        {:x => columns(data)})
end

rmdl_K3 = learn_gmm(data, 3)
rmdl_K5 = learn_gmm(data, 5)
```

Here, `learn_gmm` is a light-weight wrapper of `learn_model`.

1.4.3 Evaluation

Evaluation refers to the task of evaluating log-pdf of samples with respect to a learned model.

```
# evaluate the logpdf of x with respect to a GMM
lp = query(rmdl, {:x=>columns(data)}, :(logpdf(x)))
```

1.4.4 Options

The compilation options that control how the query is compiled can be specified through the `options` argument in the `query` or `learn_model` function. The following is some examples

```
rmdl = learn_model(mdl, data, {:method=>"variational_em", :max_iter=>100, :tolerance=1.0e-6})
```

For sampling, we may use a different set of options

```
options = {
    :method=>"gibbs_sampling",  # choose to use Gibbs sampling
    :burnin=>5000,              # the number of burn-in iterations
    :lag=>100}                  # the interval between two samples to retain

samples = query(rmdl, {x:=>columns(data)}, :(samples(z, 100)), options)
```

1.4.5 Query Functions with Arguments

It is often desirable in practice that a query function can be applied to different data sets without being re-compiled. For this purpose, we introduce a function `make_query_function`. The following example illustrates its use:

```
# suppose rmdl is a learned model

q = make_query_function(rmdl,
    (:data,),  # indicates that the function q would take one argument data
    {:x=>:(columns(data))},  # indicates how the argument is set to the model as a known value
    {:z},      # specifies what to query
    options)   # compilation options

# q can be repeatedly use for different datasets (without being re-compiled)
z1 = q(x1)
z2 = q(x2)
```

Note that `q` is a closure that holds reference to the learned model, so you don't have to pass the model as an argument into `q`. The following code use this mechanism to generate a sampler:

```
q = make_query_function(rmdl,
    (:data, :n), # q would take two arguments, the observed data and the number of samples
    {:x=>:(columns(data))},
    {:sample(z, n)},
    options)

# draw 100 samples of z
zs1 = q(x, 100)

# draw another 200 samples of z
zs2 = q(x, 200)
```

CHAPTER 2

More Examples

This chapter shows several common examples to illusrate how the model specification DSL works in practice.

2.1 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is a probabilistic model for topic modeling. The basic idea is to use a set of topics to describe documents. Key aspects of this model is summarized below:

1. Each document is considered as a *bag of words*, which means that only the frequencies of words matter, while the sequential order is ignored. In practice, each document is summarized by a histogram vector h.
2. The model comprises a set of topics. We denote the number of topics by K. Each topic is characterized by a distribution over vocabulary, denoted by β_k. It is a common practice to place a Dirichlet prior over these distributions.
3. Each document is associated with a topic proportion vector θ, generated from a Dirichlet prior.
4. Each word in a document is generated independently. Specifically, a word is generated as follows

$$\begin{aligned} z_i &\sim \theta \\ w_i | z_i &\sim \beta_{z_i} \end{aligned}$$

   Here, z_i indicates the topic associated with the word w_i.

The model specification is then given by

```
@model LatentDirichletAllocation begin
    # constants
    @constant m::Int      # vocabulary size
    @constant n::Int      # number of documents
    @constant nw::Vector{Int}    # numbers of words in documents

    @hyperparam K::Int      # number of topics
    @hyperparam a::Float64  # Dirichlet hyper-prior for beta

    # parameters
    @param alpha :: (0.0 .. Inf)^K
    for k in 1 : K
        @param beta[k] ~ Dirichlet(a)
    end

```

```
    # documents
    for i in 1 : n
        theta[i] ~ Dirichlet(alpha)
        let p = sum(beta[k] * theta[i][k] for k in 1 : K)
            h[i] ~ Multinomial(nw[i], p)
        end
    end
end
```

Here, the keyword `let` introduces a local value `p` to denote the sum vector. It is important to note that `p` has a local scope (thus it is not visible outside the loop), and the value of `p` can be different for different `i`.

The following function learns an LDA model from word histograms of training documents.

```
function learn_lda(h::Matrix{Double}, K::Int, alpha::Float64)
    learn_model(LatentDirichletAllocation(
        m = size(h, 1), n = size(h, 2), nw = sum(h, 1), K = K, a = alpha),
        {:h => columns(h)})
end

mdl = learn_lda(training_hists, K, alpha)
```

The following statement infers topic proportions of testing documents, given a learned LDA model.

```
# suppose mdl is a learned LDA model
theta = query(mdl, {:h => columns(testing_corpus)}, :theta)
```

2.2 Hidden Markov Model

Hidden Markov Model (HMM) is a popular model to describe dynamic processes. It assumes that the observed process is driven by a latent Markov chain, and the observation at each time step is independently generated conditioned on the latent states at the same time. A time-homogeneous Markov model is characterized by an initial distribution of states, denoted by π, a transition probability matrix, denoted by T, and an observation model that generates observations based on latent states. Each sample of a Hidden Markov model is a sequence $x = (x_0, \ldots, x_n)$, where the observation x_t is associated with a latent state s_t. The joint distribution over both the observations and the states is given by

$$p(x, s) = \pi(x_0) \prod_{t=1}^{n} T(x_{t-1}, x_t) \prod_{t=0}^{n} p(x_t | s_t; \theta).$$

Here, θ denotes the parameter of the observation model.

Generally, the component models associated with the observations can be any distributions. Therefore, HMM is actually a family of distributions that can be specified using a generic specification, as below:

```
@model HiddenMarkovModel{G, ParamTypes} begin
    @constant n::Int     # number of sequences
    @constant len::Vector{Int}    # the sequence lengths
    @hyperparam K::Int     # the size of latent state space

    # parameters
    @param pi :: (0.0 .. 1.0)^K         # initial distribution
    @param T :: (0.0 .. 1.0)^(K, K)     # transition probability matrix

    for k in 1 : K
```

```
        for j in 1 : length(ParamTypes)
            @param theta[k][j] :: ParamTypes[j]
        end
    end

    # sequences
    for i in 1 : n
        z[i][1] ~ Categorical(pi)
        for t = 2 : len[i]
            z[i][t] ~ Categorical(T[z[t-1], :])
        end

        for t in 1 : len[i]
            let s = z[i][t]
                x[i][t] ~ G(theta[s]...)
            end
        end
    end
end
```

To construct an HMM with K Gaussian components, one can write:

```
@modelalias HiddenMarkovGaussModel HiddenMarkovModel{G, Params} begin
    @constant d::Int    # vector space dimension
    @with G = MultivariateNormal
    @with Params[1] = Float64^d
    @with Params[2] = Float64^(d, d)
end
```

The following query function learns a HMM (with Gaussian components):

```
function learn_hmm(seqs, K::Int)
    # seqs is a collection of observed sequences
    # K is the number of latent states

    learn_model(HiddenMarkovGaussModel(
        n = length(seqs), len = map(length, seqs), K = K),
        {:x => seqs})
end

mdl = learn_hmm(seqs, K)
```

The following query draws samples of the latent state sequences, given a learned HMM model and a sequences of observations.

```
function hmm_sample(mdl::HiddenMarkovGaussModel, obs::Matrix{Float64}, ns::Int)
    # obs is a sequence of observed features (each column for a time step)
    # ns is the number of samples to d

    query(mdl, {:x => obs}, :(sample(z, ns)))
end

# run the function to draw 100 sample state-sequendes for x
x = rand(3, 100)
y = hmm_sample(mdl, x, 100)
```

2.3 Markov Random Fields

Unlike Bayesian networks, which can be factorized into a product of (conditional) distributions, Markov random fields are typically formulated in terms of potentials. Generally, a MRF formulation consists of two parts: identifying relevant cliques (small subsets of directly related variables) and assigning potential functions to them. In computer vision, Markov random fields are widely used in low level vision tasks, such as image recovery and segmentation. Deep Boltzmann machines, which become increasingly popular in recent years, are actually a special form of Markov random field. Here, we use a simple MRF model in the context of image denoising to demonstrate how one can use the model specification to describe an MRF.

From a probabilistic modeling standpoint, the task of image denoising can be considered as an inference problem based on an image model combined with an observation model. An image model captures the prior knowledge as to what an *clean* image may look like, while the observation model describes how the observed image is generated through a noisy imaging process. Here, we consider a simple setting: Gaussian MRF prior + white noise. A classical formulation of Gaussian MRF for image modeling is given below

$$p(x) = \frac{1}{Z} \exp\left(-E(x;\theta)\right).$$

Here, the distribution is formulated in the form of a *Gibbs distribution*, and $E(x;\theta)$ is the energy function, which is controlled by a parameter $theta$. The energy function $E(x; theta)$ can be devised in different ways. A typical design would encourage smoothness, that is, assign low energy value when the intensity values of neighboring pixels are close to each other. For example, a classical formulation uses the following energy function

$$E(x;\theta) = \theta \sum_{\{u,v\} \in \mathcal{C}} (x(u) - x(v))^2$$

Here, u and v are indices of pixels, and the clique set $cset$ contains all edges between neighboring pixels. With the white noise assumption, the observed pixel values are given by

$$y(u) = x(u) + \varepsilon(u), \quad \text{with } \varepsilon(u) \sim \mathcal{N}(0, \sigma^2).$$

Below is the specification of the joint model:

```
@model SimpleMRF begin
    @constant nimgs::Int    # the number of images
    @constant imgsizes::Vector{(Int, Int)}

    # parameters
    @param theta::Float64      # the Gaussian MRF parameter
    @param sig::Float64        # the variance of white noise

    for t in 1 : nimgs
        let m = imgsizes[t][1], n = imgsizes[t][2]
            x[t]::Float64^(m, n)    # the true image
            y[t]::Float64^(m, n)    # the observed noisy image

            let xt = x[t], yt = y[t]
                # the image prior (Gaussian MRF)
                for i in 2 : m-1, j in 2 : n-1
                    @fac exp(-theta * (xt[i,j] - xt[i,j-1])^2)
```

```
                    @fac exp(-theta * (xt[i,j] - xt[i,j+1])^2)
                    @fac exp(-theta * (xt[i,j] - xt[i-1,j])^2)
                    @fac exp(-theta * (xt[i,j] - xt[i+1,j])^2)
                end

                # the observation model
                for i in 1 : m, j in 1 : n
                    yt[i,j] ~ Normal(xt[i,j], sig)
                end
            end
        end
    end
end
```

The following statement learns the model from a set of uncorrupted images

```
# suppose imgs is an array of images
mdl = learn_model(SimpleMRF(nimgs=length(imgs), imgsizes=map(size, imgs)), {:x=>imgs})
```

In this specification, four potentials are used to connect a pixel to its left, right, upper, and lower neighbors. This approach would become quite cumbersome as the neighborhood grows. Many state-of-the-art denoising algorithms use mucher larger neighborhood (e.g. `5 x 5`, `9 x 9`, etc) to capture high order texture structure. A representative example is the *Field of Experts*, where the MRF prior is defined using a set of filters as follows:

$$p(x) = \frac{1}{Z} \exp\left(\sum_{k=1}^{K} \sum_{c \in \mathcal{C}} \rho(J_k^T x_c, \alpha_k)\right), \quad \text{with } \rho(v, \alpha) := -\alpha \log(1 + v^2).$$

Here, $\mathcal{C}$ is the set of all patches of certain size (say $5 times 5$), and x_c is the pixel values over a small patch c. Here, K filters $J_1, \ldots, J_K$ are used, and $J_k^T x_c$ is the filter response at patch c. ρ is a robust potential function that maps the filter responses to potential values, controlled by a parameter α. The specification below describes this more sophisticated model, where local functions and local variables are used to simplify the specification.

```
@model FieldOfExperts begin
    @constant K::Int          # the number of filters
    @constant w::Int          # patch size (w = 5 for 5 x 5 patches)
    @constant ew::Int = (w - 1) / 2  # half patch dimension
    @constant nimgs::Int      # the number of images
    @constant imgsizes::Vector{(Int, Int)}
    @constant sig :: Float64   # variance of white noise

    # parameters
    for k = 1 : K
        @param J[k] :: Float64^(w, w)    # filter kernel
        @param alpha[k] :: Float64       # filter coefficient
    end

    # the robust potential function
    rho(v, a) = -a * log(1 + v * v)

    for t in 1 : nimgs
        let m = imgsizes[t][1], n = imgsizes[t][2]
            x[t]::Float64^(m, n)    # the true image
            y[t]::Float64^(m, n)    # the observed noisy image

            let xt = x[t], yt = y[t]
                # the image prior
```

```
                for k in 1 : K, i in 1+ew : m-ew, j in 1+ew : n-ew
                    let c = vec(xt(i-ew:i+ew, j-ew:j+ew))
                        @fac exp(rho(dot(J[k], c), alpha[k]))
                    end
                end

                # the observation model
                for i in 1 : m, j in 1 : n
                    yt[i,j] ~ Normal(xt[i,j], sig)
                end
            end
        end
    end
end
```

Below is a query function that learns a field-of-experts model.

```
function learn_foe(imgs, w::Int, K::Int)
    # imgs: an array of images
    # w: the patch dimension
    # K: the number of filters

    mdl = FieldOfExpers(
            K = K, w = w, nimgs = length(imgs),
            imgsize = map(size, imgs))

    learn_model(mdl, {:x=>imgs})
end
```

Given a learned model, the following query function performs image denosing.

```
function foe_denoise(mdl::FieldOfExperts, sig::Float64, noisy_im::Matrix{Float64})
    # sig:  the noise variance (which is typically given in denoising tasks)
    # noisy_im:  the observed noisy image

    query(mdl, {:sig=>sig, :y=>[noisy_im]}, :x)
end

denoised_im = foe_denoise(mdl, 0.1, noisy_im)
```

2.4 Conditional Random Fields

Structured prediction, which exploits the statistical dependencies between multiple entities within an instance, has become an important area in machine learning and related fields. Conditional random field is a popular model in this area. Here, I consider a simple application of CRF in computer vision. A visual scene usually comprises multiple objects, and there exist statistical dependencies between the scene category and the objects therein. For example, a bed is more likely in the bedroom than in a forest. A conditional random field that takes advantage of such relations can be formulated as follows

$$p(s, o|x, y) = \frac{1}{Z(\alpha, \beta, \theta)} \exp\left(\psi_s(s, x; \alpha) + \sum_{i=1}^{n} \psi_o(o_i, y_i; \beta) + \sum_{i=1}^{n} \varphi(s, o_i; \theta)\right)$$

This formulation contains three potentials:

- $\psi_s(s, x; \alpha) := \alpha_s^T x$ connects the scene class s to the observed scene feature x,
- $\psi_o(o_i, y_i; \beta) := \beta_o^T y_i$ connects the object label o_i to the corresponding object feature y_i,
- $\varphi(s, o_i; \theta) := \theta(s, o_i)$ captures the statistical dependencies between scene classes and object classes.

In addition, Z is the normalization constant, whose value depends on the parameters $\alpha, \beta,$ and θ. Below is the model specification:

```
@model SceneObjectCRF begin
    @constant M::Int     # the number of scene classes
    @constant N::Int     # the number of object classes
    @constant p::Int     # the scene feature dimension
    @constant q::Int     # the object feature dimension
    @constant nscenes    # the number of scenes
    @constant nobjs::Vector{Int}  # numbers of objects in each scene

    for k in 1 : M
        @param alpha[k] :: Float64^p
    end
    for k in 1 : N
        @param beta[k] :: Float64^q
    end
    @param theta :: Float64^(p, q)

    for i in 1 : nscenes
        let n = nobjs[i]
            s[i] :: 1 .. M         # the scene class label
            o[i] :: (1 .. N)^n     # the object class labels
            x[i] :: Float64^p      # the scene feature vector

            for j in 1 : n
                y[i][j] :: Float64^q     # the object features
            end

            @fac dot(alpha[s[i]], x)
            for j in 1 : n
                let k = o[i][j]
                    @expfac dot(beta[k], y[i][j])
                    @expfac theta[s[i], k]
                end
            end
        end
    end
end
```

Note here that `@expfac f(x)` is equivalent to `@fac exp(f(x))`. The introduction of `@expfac` is to simplify the syntax in cases where factors are specified in log-scale.

2.5 Deep Boltzmann Machines

A *Boltzmann machine (BM)* is a generative probabilistic model that describes data through hidden layers. In particular, a *deep belief network* and a *deep Boltzmann machine*, which becomes increasingly popular in machine learning and its application domains, can be constructed by stacking multiple layers of BMs. In a generic Boltzmann machine, the joint distributions over both hidden units $\mathbf{h}$ and visible units $\mathbf{v}$ are given by

$$p(\mathbf{v}, \mathbf{h}; \theta) = \frac{1}{Z(\theta)} \exp\left(\frac{1}{2}\mathbf{v}^T\mathbf{L}\mathbf{v} + \frac{1}{2}\mathbf{h}^T\mathbf{J}\mathbf{h} + \mathbf{v}^T\mathbf{W}\mathbf{h}\right)$$

When $\mathbf{L}$ and $\mathbf{J}$ are zero matrices, this reduces to a *restricted Boltzmann machine*. By stakcing multiple layers of BMs, we obtain a *deep Boltzmann machine* as follows

$$p(\mathbf{v}, \mathbf{h}; \theta) = \frac{1}{Z} \exp\left(\frac{1}{2}\mathbf{v}^T\mathbf{L}\mathbf{v} + \mathbf{v}^T\mathbf{W}_0\mathbf{h}_1 + \sum_{l=1}^{L}\mathbf{h}_l^T\mathbf{J}_l\mathbf{h}_l + \sum_{l=1}^{L-1}\mathbf{h}_l^T\mathbf{W}_l\mathbf{h}_{l+1}\right)$$

This probabilistic network, despite its complex internal structure, can be readily specified using the DSL as below

```
@model DeepBoltzmannMachine begin
    @hyperparam L::Int    # the number of latent layers
    @hyperparam nnodes::Vector{Int}   # the number of nodes in each layer
    @constant d::Int    # the dimension of observed sample
    @constant n::Int    # the number of observed samples

    # declare coefficient matrices
    @param L  :: Float64^(d, d)
    @param W0 :: Float64^(d, nnodes[1])

    for k = 1 : L
        @param J[k] :: Float64^(nnodes[k], nnodes[k])
    end

    for k = 1 : L-1
        @param W[k] :: Float64^(nnodes[k], nnodes[k+1])
    end

    # declare of variables
    for i = 1 : n
        obs[i] :: Float64^d
        for k = 1 : L
            latent[i][k] :: Float64^(nnodes[k])
        end
    end

    # samples
    for i = 1 : n
        let h = samples[i], v = obs[i]
            # intra-layer connections
            @expfac v' * L * v

            for k = 1 : L
                @expfac h[k]' * J[k] * h[k]
            end

            # inter-layer connections
            @expfac v' * W0 * h[1]

            for k = 1 : L-1
                @expfac h[k]' * W[k] * h[k+1]
            end
        end
```

```
    end
end
```

To learn this model from a set of samples, one can write

```
function learn_deepbm(x::Matrix{Float64}, nnodes::Vector{Int})
    # each column of x is a sample
    # nnodes specifies the number of nodes at each layer

    learn_model(DeepBoltzmannMachine(L = length(nnodes), nnodes=nnodes,
        d=size(x,1), n=size(x,2)), {:obs=>columns(x)})
end

mdl = learn_deepbm(x, [100, 50, 20])
```

CHAPTER 3

Nonparametric Models

Bayesian nonparametric models, such as DP mixture models, provide a flexible approach in which model structure (e.g. the number of components) can be adapted to data. The capability and effectiveness of such models have been proven in many applications.

This flexibility, however, leads to new questions to probabilistic programming – how to express models whose size/structure can vary. The Julia macro system makes it possible to address this problem in an elegant way due to is lazy evaluation nature.

3.1 Dirichlet Process Mixture Model

Dirichlet process mixture model (DPMM) is one of the most widely used in Bayesian nonparametrics. The formulation of DPMM is given by

$$\begin{aligned} D &\sim DP(\alpha B) \\ \theta_i &\sim D, \;\; x_i \sim G(\theta_i), \quad \forall i = 1, 2, \ldots, n \end{aligned}$$

Here, α is the concentration paramater, B is the base measure, and G denotes the component models that generate the observed samples. The model specification for a DPMM is given as below. It has been shown that D is almost surely discrete, and can be expressed in the followin form:

$$D = \sum_{k=1}^{\infty} \pi_k \delta_{\phi_k}$$

Hence, there exists positive probability that some of the components will be repeatedly sampled, and thus the number of distinct components is usually smaller than the number of samples. In practice, it is useful to construct a pool of components $\phi_1, \phi_2, \ldots$, and introduce an indicator z_i for each sample x_i, such that $\theta_i = \phi_{z_i}$. To make this explicit, we can reformulate the model as below

$$\begin{aligned} \pi &\sim StickBreak(\alpha) \\ \phi_k &\sim B, \quad \forall k = 1, 2, \ldots \\ z_i &\sim \pi, \;\; x_i \sim G(\phi_{z_i}), \quad \forall i = 1, 2, \ldots, n \end{aligned}$$

Here, `\pi`, a sample from the stick breaking process, is an infinite sequence that sums to unity (i.e. $\sum_{i=1}^{\infty} \pi_i = 1$). The values of pi are defined as

$$
\begin{aligned}
v_k &\sim Beta(1, \alpha), \quad \forall k = 1, 2, \ldots \\
\pi_1 = v1, \;\; \pi_k &= v_k \prod_{l=1}^{k-1} (1 - v_l), \quad \forall k = 1, 2, \ldots
\end{aligned}
$$

The model specification for this formulation is given below

```
@model DPMixtureModel{B, G} begin
    @constant n::Int            # the number of observed samples
    @hyperparam alpha::Float64  # the concentration parameter

    pi ~ StickBreak(alpha)
    for k = 1 : Inf
        phi[k] ~ B
    end

    # samples
    for i = 1 : n
        z[i] ~ pi
        x[i] ~ G(phi[z[i]])
    end
end
```

Remarks:

- The parametric setting makes it possible to use arbitrary base distribution B and component G here, with the same generic formulation.
- In theory, π is an infinite sequence, and therefore it is not feasible to completely instantiate a sample path of π in computer. This variable may be marginalized out in the inference, and thus directly querying π is not allowed.
- ϕ has infinitely many components. However, only a finite number of them are needed in inference. The compiler should generate a lazy data structure that only memorizes the subset of components needed during the inference. In particular, `phi[k]` is constructed and memorized when there is an `i` such that `k = z[i]`. Some efforts (e.g. the *LazySequences.jl*) have demonstrated that lazy data structure can be implemented efficiently in Julia.

3.2 Hierarchical Dirichlet Processes

HDP is an extension of the DP mixture models, which allows groups of data to be modeled by different DPs that share components. The formulation of HDP is given below

$$
\begin{aligned}
D_0 &\sim DP(\alpha B) \\
D_k &\sim DP(\gamma_k D_0) \\
\theta_{ki} &\sim D_k, \quad x_{ki} \sim G(\theta_{ki}), \quad \forall k = 1, \ldots, m, \; i = 1, \ldots, n_k
\end{aligned}
$$

Using D_0 as a base measure for the DP associated with each group, all groups share components in D_0 while allowing potentially infinite number of components. This formulation can be re-written (equivalently) using Pitman-Yor process, as follows

$$
\begin{aligned}
\pi_0 &\sim StickBreak(\alpha) \\
\psi_j &\sim B, \quad \forall j = 1, 2, \ldots
\end{aligned}
$$

Then for each $k = 1, \ldots, m$,

$$\begin{aligned} \pi_k &\sim StickBreak(\gamma_k) \\ u_{kt} &\sim \pi_0, \quad \phi_{kt} = \psi_{u_{kt}}, \quad t = 1, 2, \ldots \\ z_{ki} &\sim \pi_k, \quad x_{ki} \sim G(\phi_{kz_{ki}}), \quad i = 1, 2, \ldots, n_k \end{aligned}$$

Here is a brief description of this procedure:

1. To generate D_0, we first draw an *infinite* multinomial distribution π_0 from a Pitman-Yor process with concentration parameter α, and draw each component ψ_j from B. Then $D_0 = \sum_{j=1}^{\infty} \pi_j \psi_j$.
2. Then for each group (say the k-th one), we draw π_k from a stick breaking process and draw each component from D_0. Note that drawing a component ϕ_{kt} from D_0 is equivalent to choosing one of the atoms in D_0, which can be done in two steps: draw u_{kt} from π_0 and then set $\phi_{kt} = \psi_{u_{kt}}$. In other words, the t-th component in the k-th group is identical to the u_{kt}-th component in D_0.
3. Finally, to generate the i-th sample in the k-th group, denoted by x_{ki}, we first draw z_{ki} from π_k and use the corresponding component $\phi_{kz_{ki}}$ to generate the sample.

This formulation can be expressed using the DSL as below:

```
@model HierarchicalDP{B, G} begin
    @constant m::Int            # the number of groups
    @constant ns::Vector{Int}   # the number of samples in each group
    @hyperparam alpha::Float64    # the base concentration
    @hyperparam gamma::Float64    # the group specific concentration

    # for D0
    pi0 ~ StickBreak(alpha)
    for j = 1 : Inf
        psi[j] ~ B
    end

    # each group
    for k = 1 : m
        pi[k] ~ StickBreak(gamma)

        # Dk
        for t = 1 : Inf
            u[k][t] ~ pi0
            phi[k][t] = psi[u[k][t]]
        end

        # samples
        for i = 1 : ns[k]
            z[k][t] ~ pi[k]
            x[k][i] ~ G(phi[z[k][t]])
        end
    end
end
```

3.3 Gaussian Processes

Gaussian process (GP) is another important stochastic process that is widely used in Bayesian modeling. Formally, a Gaussian process is defined to be a a function-valued distribution $X_t : t \in T$, where T can be arbitrary domain, such

that any finite subset of values in X_t is normally distributed. A Gaussian process is characterized by a mean function $\mu : T \to R$ and a positive definite covariance function $\kappa : T \times T \to R$. The covariance function is typically given in a parametric form. The following is one that is widely used

$$\kappa(s, t; \theta) = \theta_0 \delta_{s,t} + \theta_1 \exp(-\theta_2 (s - t)^2)$$

In many application, the GP is considered to be hidden, and observations are a noisy transformation of the samples generated from the GP, as

$$\begin{aligned} g &\sim GP(\mu, \kappa) \\ x_i &\sim B, \quad y_i \sim F(g(x_i)), \qquad i = 1, \ldots, n \end{aligned}$$

The following model specification describes this model.

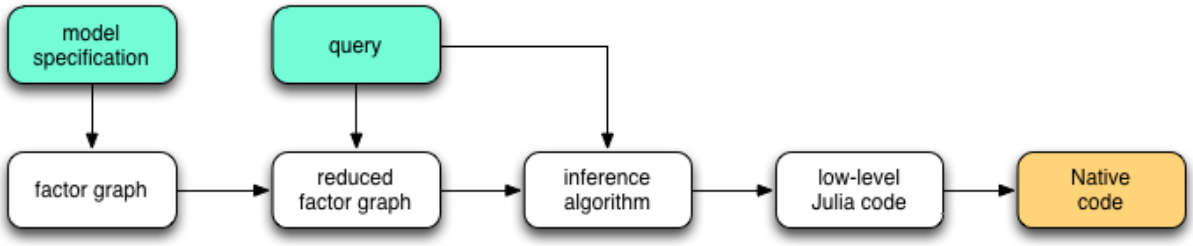
```
@model TransformedGaussProcess{B, F} begin
    @constant n::Int        # the number of observed samples

    # define mean and covariance function
    @param theta::Float^3
    mu(x) = 0.
    kappa(x, y) = theta[1] * delta(x, y) + theta[2] * exp(- theta[3] * abs2(x - y))

    # GP
    g ~ GaussianProcess(mu, kappa)

    # samples
    for i = 1 : n
        x[i] ~ B
        y[i] ~ F(g[x[i]])
    end
end
```

The `GaussianProcess` distribution here is a high-order stochastic function, which takes into two function arguments and generates another function. This is readily implementable in Julia, where functions are first-class citizens like in many functional programming languages.

CHAPTER 4

The Inference Framework

The diagram below outlines the overall architecture of the inference framework.



The entire process of generating inference codes from model and query specification consists of four stages:

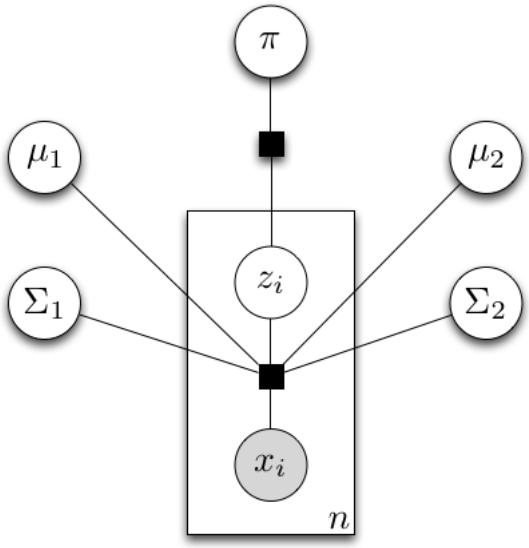
1. The `@model` macro will convert a model specification into a model class, of which the internal representation is a factor graph.
2. The `@query` macro will reduce the factor graph based on the query. The reduction may involve following steps:
   (a) Simplify factors by absorbing known variables. For example, a second-order factor (i.e. a factor with two arguments) $f(x, y)$ can be reduced to a first-order factor if the value of one variable (say y) is given.
   (b) Eliminate irrelevant variables and factors: variables and factors that do not influence the conditional distribution of the queried variables can be safely removed. For example, consider a joint distribution $p(x, y, z) = p_1(x|z)p_2(y|z)$. When the value of y is given, the variable y is conditionally independent from x. Therefore, the variable y can be ignored in the inference for x.

   The macro `@query` also generates a high-level description of the inference algorithm.
3. An *inference compiler* will compiles the inference algorithm into low-level Julia codes, taking into account the computational architecture of the target platform (e.g. CPU cores, GPU, cluster, cloud, etc).
4. Finally, the Julia compiler will emit LLVM instructions, which will then be compiled into native codes by the LLVM compiler.

Here, we focus on the first two steps, that is, compilation of model and query specifications into inference algorithms.

4.1 Gaussian Mixture Model Revisited

To illustrate the procedure of model compilation, let's revisit the Gaussian mixture model (GMM). Given a model specification, the `@model` macro creates a *factor graph*, which is a hyper-graph with factors connecting between variables. The following diagram is the factor graph that represents a GMM with two components.



In this graph, each sample is associated with two factors, a `mixture factor` that connects between observed samples, components, and component indicator z_i, and a factor that specifies the prior of each component indicator. These two factors were directly specified in the model specification. In the *model learning query*, the data `x` are given. Therefore the order of each mixture factor is reduced from $2K + 2$ to $2K + 1$.

The most widely used learning algorithm for this purpose is *Expectation Maximization*, which comprises three updating steps, as outlined below.

- Update the poterior probabilities of z_i conditioned on prior π, component parameters (μ_k, Σ_k), and the corresponding observed sample x_i, as

$$\begin{aligned} q_i(k) := p(z_i = k|\pi, \theta, x_i) &\propto \pi(k)\mathcal{N}(x_i; \mu_k, \Sigma_k) \\ &= \exp\left(\log \pi(k) + \log \mathcal{N}(x_i; \mu_k, \Sigma_k)\right) \end{aligned}$$

  This can be interpreted as an integration of a message from the prior factor (that is, $\log \pi(k)$) and a message from the corresponding mixture factor (that is, $\log \mathcal{N}(x_i; \mu_k, \Sigma_k)$). Here, $\log \mathcal{N}(x_i; \mu_k, \Sigma_k)$ denotes the pdf at x_i with respect to a Gaussian component.

- Update the maximum likelihood estimation of π, as

$$\pi(k) = \frac{1}{n}\sum_{i=1}^{n} p(z_i = k) = \frac{1}{n}\sum_{i=1}^{n} \exp(\log q_i(k))$$

  This can be interpreted as a computational process that combines messages from each of z_i (that is, $\log q_i(k)$).

- Update the maximum likelihood estimation of component parameters μ_k and Σ_k, as

$$\begin{aligned} \mu_k &= \left(\sum_{i=1}^{n} q_i(k)x_i\right)\left(\sum_{i=1}^{n} q_i(k)\right)^{-1} \\ \Sigma_k &= \left(\sum_{i=1}^{n} q_i(k)(x_i - \mu_k)(x_i - \mu_k)^T\right)\left(\sum_{i=1}^{n} q_i(k)\right)^{-1} \end{aligned}$$

Again, this is also a specific way that combines messages from each of the mixture factors associated with them.

The analysis above shows that the E-M algorithm can be viewed as a message-passing scheme that iteratively update the states associated with each variable by exchanging messages between factors and variables. Actually, the message-passing scheme is a very general paradigm. Many widely used inference algorithms, including *mean-field based variational inference*, *belief propagation*, *expectation propagation*, *Gibbs sampling*, and *Hamiltonian Monte Carlo*, can be implemented as certain forms of message passing schemes.

4.2 Inference via Message Passing

Generally, a message passing procedure consists of several stages:

1. Initialize the states associated with each variable. Depending on the chosen algorithm, the states associated with a variable can be in differen forms. For example, for a discrete variable, its associated state can be a value (in Gibbs sampling or maximum-likelihood estimation), variational distribution (in variational inference), or a marginal distribution (in belief propagation or expectation propagation).
2. Iteratively invoke the following steps until convergence or other termination criteria are met.
   (a) update the message from a factor to some of its incident variables (based on updated status of other incident variables).
   (b) update variable states based on incoming messages.
3. Compute queried quantities based on variable states. For example, if a Gibbs sampling algorithm is used, the expectation of a variable can be approximated by the sample mean.

It is possible to generate such an inference procedure according to the structure of the factor graph. However, there remains several challenges to be addressed:

1. Some intermediate quantities may be used in the computation of several different messages. For example, $\sum_{i=1}^{n} q_i(k)$ appears in multiple updating formulas for GMM. It is desirable to identify these quantities and avoid unnecessary re-computation of the same value.
2. Each message depends on the states of several variables, while the states of a variable may depend on several messages. A message/variable state only needs to be updated when its depending values have changed. To identify whether a variable/message needs to be updated, a natural idea is to build a dependency graph, where each node corresponds to either a variable state or a message. By time-stamping each node, it is not difficult to see whether a node can be updated by looking at the time-stamps of each neighboring nodes.
3. Updating steps can be scheduled in numerous ways. Poor scheduling may result in slow convergence. Therefore, deriving a reasonable schedule is also important to achieve high efficiency.