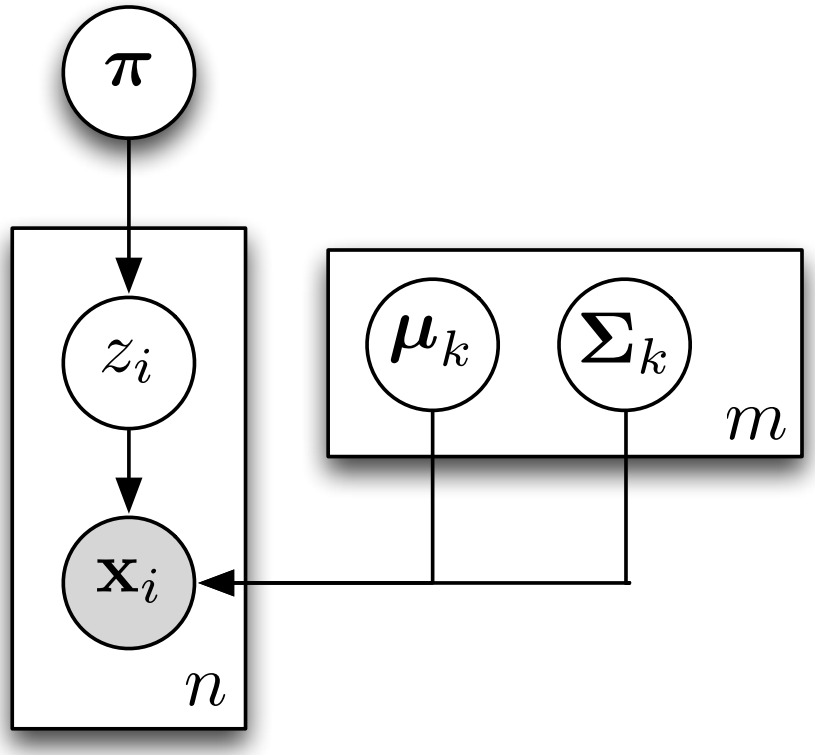
A Julia Framework for Bayesian Inference

Dahua Lin

Motivation

- Writing procedure to perform estimation or inference over complex graphical models is often tedious and error-prone
- Existing tools are unsatisfactory
  - C/C++/Fortran: low productivity
  - Research codes that come with papers: buggy and difficult to extend/generalize
  - Generic engines (*WinBUGs*, *VIBES*, etc): overly slow
- Our goal: greatly simplify the process of writing Bayesian inference procedures, while maintaining competitive performance

Example: Gaussian Mixture Model



$$z_i \sim \boldsymbol{\pi}$$
$$\mathbf{x}_i \sim \mathcal{N}(\boldsymbol{\mu}_{z_i}, \boldsymbol{\Sigma}_{z_i})$$

Estimating a GMM using EM

- Optimize a variational objective function:

$$L(\boldsymbol{\theta}, \mathbf{x}, q) = \sum_{i=1}^{n} \mathrm{E}_{z_i \sim \mathbf{q}_i} \left[\log p(\mathbf{x}_i | \boldsymbol{\mu}, \boldsymbol{\Sigma}) + \log p(z_i | \boldsymbol{\pi}) \right] + \sum_{i=1}^{n} H_{\mathbf{q}_i}(\mathbf{q}_i)$$

with

$$E_{z_i \sim \mathbf{q}_i} \log p(\mathbf{x}_i | \boldsymbol{\theta}_{z_i}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{k=1}^{m} q_{ik} \log \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

$$E_{z_i \sim \mathbf{q}_i} \log p(z_i | \boldsymbol{\pi}) = \sum_{k=1}^{m} \log q_{ik} \pi_k$$

$$H_{\mathbf{q}_i}(\mathbf{q}_i) = -\sum_{k=1}^{m} q_{ik} \log(q_{ik})$$

Estimating a GMM using EM

The optimization procedure alternates the updates between

M-steps:

$$w_k^{(t)} = \sum_{i=1}^{n} q_{ik}^{(t-1)}$$

$$\boldsymbol{\mu}_k^{(t)} = \frac{1}{w_k^{(t)}} \sum_{i=1}^{n} q_{ik}^{(t-1)} \mathbf{x}_i$$

$$\boldsymbol{\Sigma}_k^{(t)} = \frac{1}{w_k^{(t)}} \sum_{i=1}^{n} q_{ik}^{(t-1)} (\mathbf{x}_i - \boldsymbol{\mu}_k^{(t)})(\mathbf{x}_i - \boldsymbol{\mu}_k^{(t)})^T$$

$$\pi_k^{(t)} = w_k^{(t)} / n$$

E-steps:

$$q_{ik}^{(t)} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{(t)})}{\sum_{l=1}^{m} \pi_l \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_l^{(t)}, \boldsymbol{\Sigma}_l^{(t)})}$$

Available choices

- Use an existing GMM tools (*e.g.* MATLAB has one):
  - many are implemented in a way that runs 100x slower than it should
  - an experiment that should take two days to run now takes half a year – miss an important conference deadline
- Implement it yourself (refer to my implementation in PLI-toolbox)
  - several hundred lines of code to prepare the infrastructure
  - several hundred lines of code for the main procedure

What if you want to extend the model

- In pratice, you may want to
  - change the component from Gauss to something else
  - add an indicator to filter out outliers
  - add a prior to component parameters
  - add a MRF to connect between labels
  - some combination of the above, ...

- Every time I want to adapt/extend the model to a new application, I ended up
  - Re-deriving part or all of the updating formulas
  - Re-writing the main inference procedure
  - Going through again the debugging cycles

A New Framework for Bayesian Inference

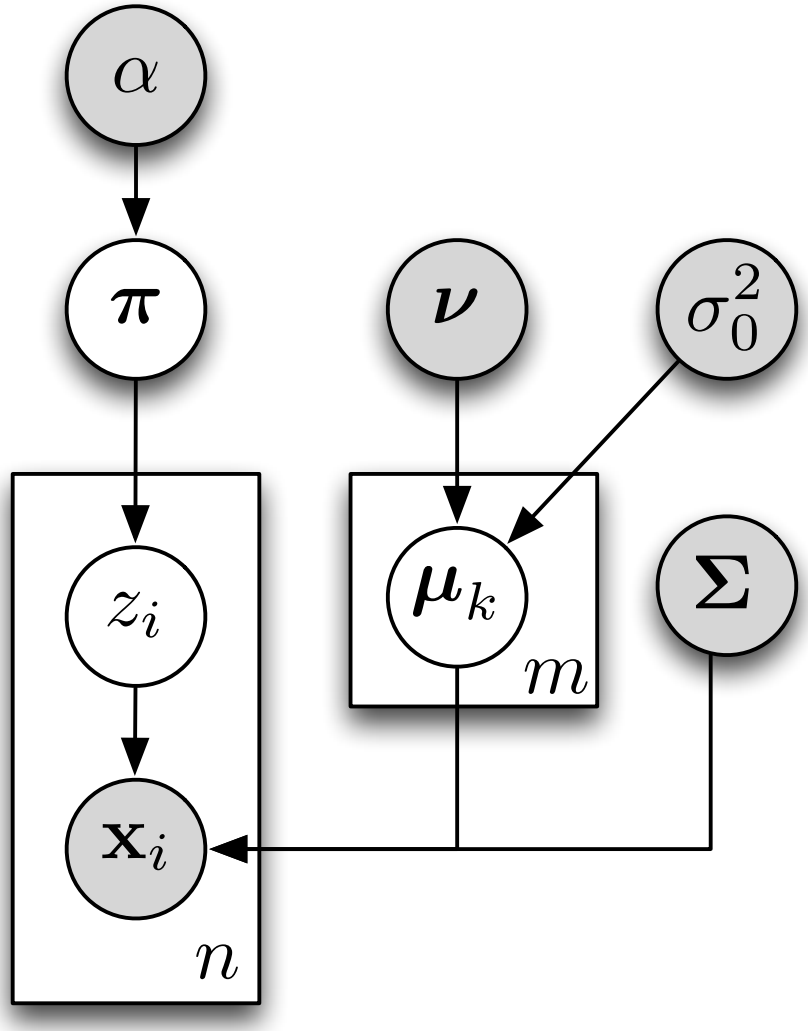
Key motivation: I am so tired of such tedious cycles, and decided to do something to make my (and perhaps many others') life easier.

- I had unsuccessful tries of implementing this framework in MATLAB and Python.
  - Such a framework inevitably requires lots of abstractions.
  - The overhead of introducing abstractions is so large that it often takes up over 90% of the run-time.

- I decided to resume this project after I found and were impressed by **Julia**
  - **Julia** is a very young language (being developed at MIT)
  - It is the best combination of *elegance* and *performance* I have ever seen.
  - It is as easy to use as MATLAB, but with a much more powerful type system and much lower cost of introducing abstractions.

Overview

I use the following modified version of GMM to illustrate what it would look like.

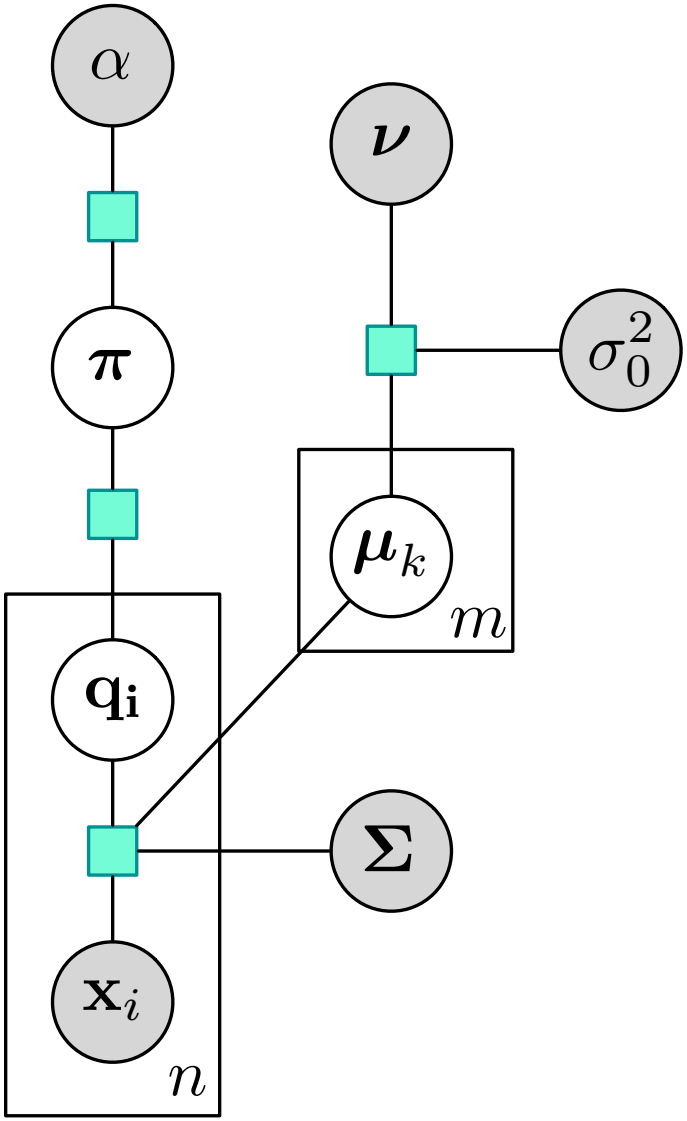


$$\boldsymbol{\pi} \sim \text{Dirichlet}(\alpha)$$
$$\boldsymbol{\mu}_k \sim \mathcal{N}(\boldsymbol{\nu}, \sigma_0^2)$$
$$z_i \sim \boldsymbol{\pi}$$
$$\mathbf{x}_i \sim \mathcal{N}(\boldsymbol{\mu}_{z_i}, \boldsymbol{\Sigma})$$

This is a simple and reasonable modification, but most GMM packages out there simply cannot handle it.

Factor Graph

We use a generic notion *factor* to represent the relations between variables.

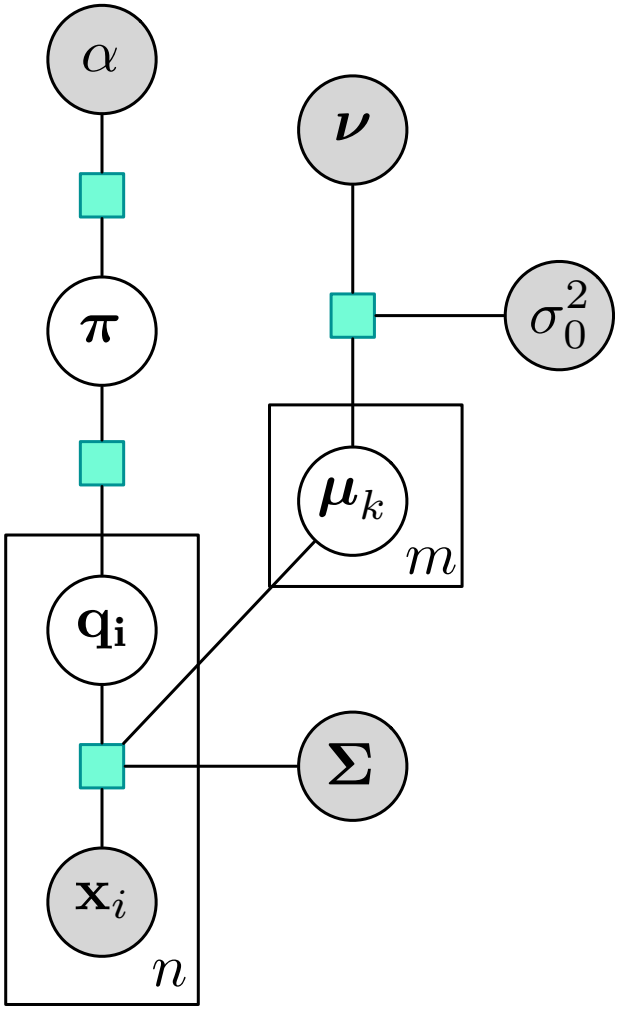


Convert to a *factor graph*

using mean-field approximation, z_i is replaced by a variational distribution $\mathbf{q}_i$ here

- Dirichlet factor: $\alpha, \boldsymbol{\pi}$
- Categorical factor: $\boldsymbol{\pi}, \mathbf{q}_i$
- Gaussian factor: $\boldsymbol{\nu}, \sigma_0^2, \boldsymbol{\mu}_k$
- Mixture factor: $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}, \mathbf{q}_i, \mathbf{x}_i$
- Entropy factor (hidden) for $\mathbf{q}_i$

Codes: add variables

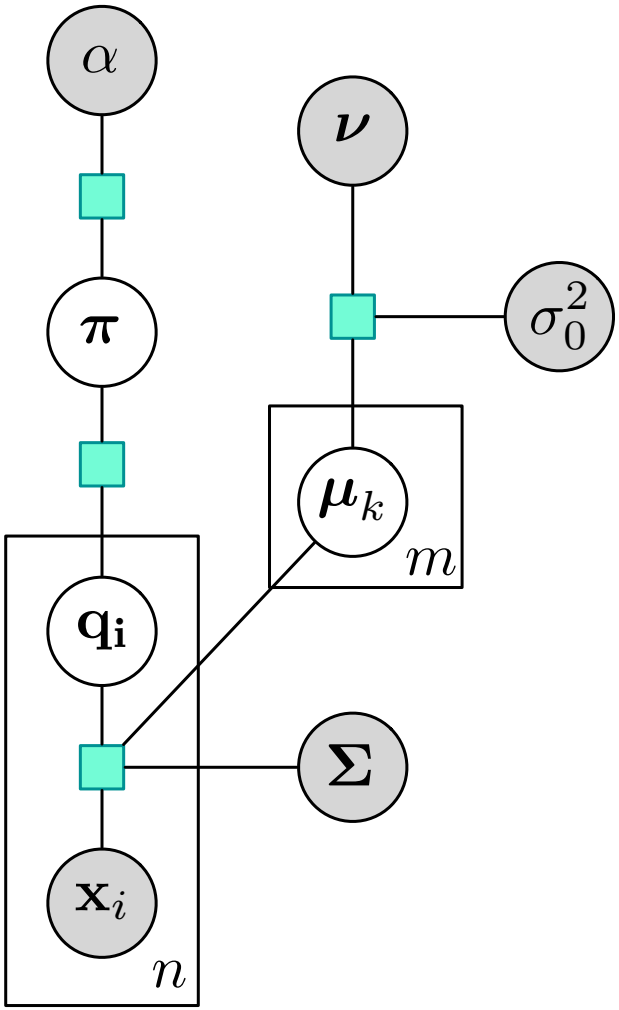



```
using BayesInference

model = ModelTemplate("MyGMM")
@add_consts model begin
    n : Int
    m : Int
    d : Int
end

@add_vars model begin
    alpha : RealVar
    pi    : RealVecVar(m)
    q     : RealMatVar(m, n)
    x     : RealMatVar(d, n)
    nu    : RealVecVar(d)
    s0    : RealVar
    mu    : RealMatVar(d, m)
    sig   : RealMatVar(d, d)
end
```

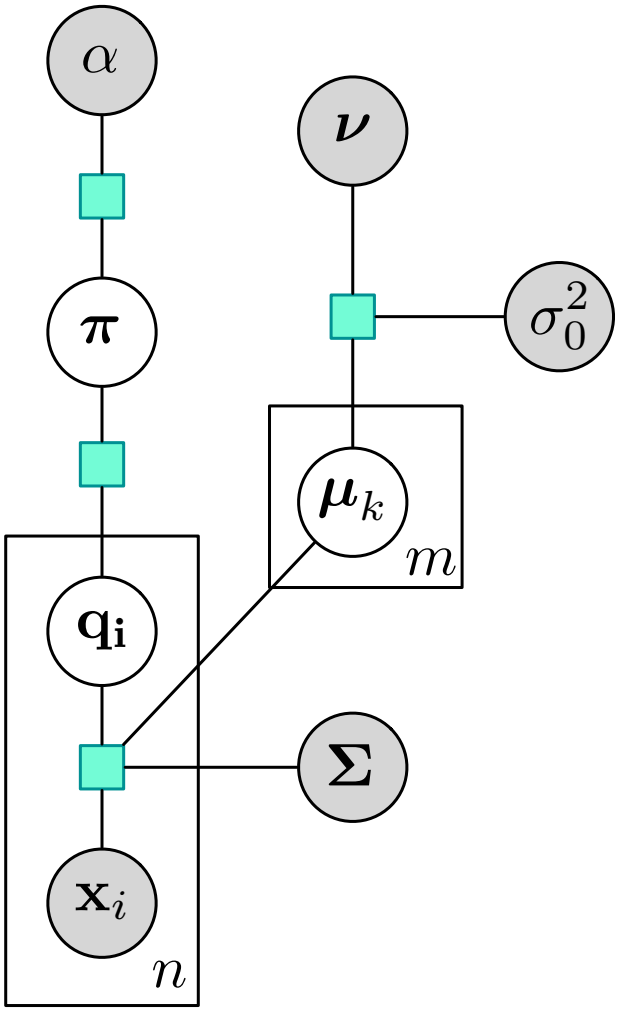
Codes: add factors




```
@add_factors model begin
    pi_fac :  DirichletFactor(alpha, pi)
    q_fac  :  CategoricalFactor(pi, q)
    mu_fac :  GaussFactor(nu, s0, mu)
    mix    :  Mixture{GaussFactor}
              ((mu, sig), q, x)
    q_ent  :  CategoricalEntropy(q)
end
model = compile(model)
```

Thanks to Julia's multiple dispatch mechanism, a factor type can handle different types of variables (e.g. single variable, an array of variables, or a variational distribution over them, etc)

Codes: Inference planning



An iteration comprises a sequences of message passing steps

```
init = auto_init_scheme(model,
        :x, :sig, :alpha, :nu, :s0)

# manual planning
updates = @set_iteration model begin
    # M-steps
    pi << (pi_fac, q_fac)
    mu << (mu_fac, mix)

    # E-steps
    q << (q_fac, mix, q_ent)
end
```

Codes: executing the inference/estimation

```
instance = instantiate(model,
    {:x=>x, :sig=>xsig, :alpha=>a0, :nu=nu, :s0=s0})

init(instance)
opts = @options max_iter=50 tol=1.0e-6 display=:iter
iterative_update!(instance, updates, opts)
```

- Every message passing step will be delegated to a specialized `send_message` function, which actually did the job.
- Each factor is associated with a `evaluate` function to compute the factor value. The `iterative_update!` function combines these values to get the objective and determine convergence.
- Each factor may maintain some internal states for inference as well as references to its incident variables.
  - Each time a variable is updated, it will notify all its neighboring factors, which may then make according changes to their internal states.

Design consideration

- Using *factor graph representation* instead of the original model greatly simplifies the back-end implementation.
  - Consider a factor/clique relating four variables a, b, c, d, then without explicitly using factor, you have to implement a large number of inference routines (*e.g.*, $a, b \to c, d$; $a, b, c \to d$, etc) – this number grow exponentially.

- Using *factor graph representation* allows non-casual (i.e. undirected) relations (e.g. those in Markov random fields)

- With this framework, the design of updating steps and the planning of iterations are decoupled.
  - In these ways, the updating functions can be easily reused under different model settings
  - These two aspects are usually coupled in typical implementations, making reuse of such codes difficult.

Philosophy

- This framework is for people who has basic understanding of machine learning, graphical models, and probabilistic inference.
- Our primary goal is to develop a domain-specific language to help people to express their model and algorithm.
- We give the users all control of how the inference is actually performed, instead of encapsulating it into a magical blackbox.
  - There are some "inference engines" that claim to be able to automatically perform the inference with only a model description – the result is usually a sub-optimal procedure that runs much slower.
- Users can design their own updating steps and incorporate them easily by wrapping them into a specialized `send_message` function.