ForneyLab.jl
a Julia Toolbox for Factor Graph-based Probabilistic Programming

JuliaCon 2018
Example Dataset
Iterative Model Design

Automation

Manual

Build Model

Automatable

Data

Infer Quantities

Criticize Model

Apply Model

Repeat
Data → Build Model → Quantities → Infer Quantities → Apply Model → Criticize Model → Message Passing → Factor Graphs → Repeat
ForneyLab

Data

Factor Graphs

Build Model

Message Passing

Infer Quantities

Apply Model

Repeat

Criticize Model

Free Energy
ForneyLab

Julia package for automatically generating Bayesian inference algorithms through message passing on Forney-style factor graphs.

Add topics

1,466 commits 1 branch 12 releases 6 contributors

Branches: master

Latest commit 42c9a7c a day ago

https://github.com/biaslab/ForneyLab.jl
Model Specification

Prior: \( x_0 \sim \mathcal{N}_p(0, 0.04) \)

State transition model: \( x_t \sim \mathcal{N}_p(x_{t-1}, 100) \)

Observation model: \( y_t \sim \mathcal{N}_p(x_t, 10) \)

```plaintext
@RV x_0 \sim \text{GaussianMeanPrecision}(0.0, 0.04) \ # State prior

x_t_{min} = x_0

for t=1:T
    @RV x[t] \sim \text{GaussianMeanPrecision}(x_t_{min}, 100.0) \ # State transition model
    @RV y[t] \sim \text{GaussianMeanPrecision}(x[t], 10.0) \ # Observation model
    placeholder(y[t], :y, index=t) \ # Placeholder for data

    x_t_{min} = x[t] \ # Reset state for next section
end
```
Factor Graph Representation


Inference Specification

q = RecognitionFactorization([x_0; x], ...) # Specify a recognition distribution
algo = variationalAlgorithm(q) # Construct the inference algorithm

Variational Message Passing

q = RecognitionFactorization([x_0; x], ...) # Specify a recognition distribution
algo = variationalAlgorithm(q) # Construct the inference algorithm
Automated Algorithm Generation

```plaintext
q = RecognitionFactorization([x_0; x], ...) # Specify a recognition distribution
algo = variationalAlgorithm(q) # Construct the inference algorithm

function step!(data::Dict, marginals::Dict=Dict(),
              messages::Vector{Message}=Array{Message}(499))

    messages[1] = ruleVBGaussianMeanPrecisionM(ProbabilityDistribution(Univariate,
                                                                     PointMass, m=data[:y][50]), nothing, ProbabilityDistribution(Univariate,
                                                                     PointMass, m=10.0))
    ...
    messages[499] = ruleSVBGateauinMeanPrecisionMGVD(messages[498], nothing, ProbabilityDistribution(Univariate, PointMass, m=100.0))

    marginals[:, x_0] = messages[3].dist * messages[499].dist
    ...
    return marginals
end
```
Inference Results

Infer Quantities

Estimated state (x)

Observed position (y)
Model Performance

```
algo_F = freeEnergyAlgorithm(q)  # Construct a performance evaluation metric
```

Automated Performance Evaluation

algo_F = freeEnergyAlgorithm(q)  # Construct a performance evaluation metric

```python
function freeEnergy(data::Dict, marginals::Dict)
    F = 0.0

    F += averageEnergy(GaussianMeanPrecision, marginals[:x_1_x_0],
                        ProbabilityDistribution(Univariate, PointMass, m=100.0))
    F += averageEnergy(GaussianMeanPrecision, ProbabilityDistribution(Univariate,
                                                                      PointMass, m=data[:y][44]), marginals[:x_44],
                        ProbabilityDistribution(Univariate, PointMass, m=10.0))
    ...
    F -= differentialEntrophy(marginals[:x_0])

    return F
end
```
Model Comparison

Evaluate free energy (less is better)

\[ F_m = 294 \text{ [dB]} \]
Model Adaptation

Data

Build Model → Infer Quantities → Apply Model → Criticize Model → Build Model

Repeat
Model Adaptation

**Prior:** \( x_0 \sim \mathcal{N}_p(0, 0.04) \)

**State transition model:** \( x_t \sim \mathcal{N}_p(Ax_{t-1}, 100) \)

**Observation model:** \( y_t \sim \mathcal{N}_p(b^T x_t, 10) \)

```plaintext
@RV x_0 ~ GaussianMeanPrecision(zeros(2), 0.04*eye(2))  # State prior

x_t_min = x_0
for t=1:T
    @RV x[t] ~ GaussianMeanPrecision(A*x_t_min, 100.0*eye(2))  # Transition model
    @RV y[t] ~ GaussianMeanPrecision(dot(b, x[t]), 10.0*eye(2))  # Obs. model
    placeholder(y[t], :y, index=t)  # Placeholder for data

    x_t_min = x[t]  # Reset state for next section
end
```
Model Adaptation

\[ x_{t-1} \rightarrow A \rightarrow \mathcal{N}_p \rightarrow = \rightarrow x_t \rightarrow \cdots \]

\[ \cdots \rightarrow \mathcal{N}_p \rightarrow b^T \rightarrow \mathcal{N}_p \rightarrow \gamma_t \rightarrow \cdots \]
Inference Results

- Estimated state (x) with (m)
- Estimated state (x) with (m')
- Observed position (y)

Time (t)
Model Comparison

Evaluate free energy (less is better)

\[ F_{m'} = 205 \ [dB]\]

\[ F_m = 294 \ [dB]\]
Model Comparison

Evaluate free energy (less is better)

\[ F_{m'} = 205 \text{ [dB]} \]

\[ F_m = 294 \text{ [dB]} \]
ForneyLab

• Enhances the **probabilistic model design cycle**
• Is a Julia program that **writes Julia programs**
• Is **available** on GitHub
Thanks

Ivan Bocharov
Anouk van Diepen
Joris Kraak

Ismail Senoz
Tjalling Tjalkens
Wouter van Roosmalen