5SSD0

Bayesian Machine Learning and Information Processing

Probabilistic Programming

Bert de Vries (Flux 7.101)

Wouter Kouw (Flux 7.060)

Magnus T. Koudahl (Flux 7.060)

Probabilistic Programming 1: Introduction to Bayesian Machine Learning

Goal

- Familiarize yourself with basic concepts from Bayesian inference such as prior and posterior distributions.
- Familiarize yourself with Jupyter notebooks and the basics of the Julia programming language.

Materials

- Mandatory
  - This notebook
  - Lecture notes on Probability Theory
  - Lecture notes on Bayesian Machine Learning
- Optional
  - Course installation guide
  - Jupyter notebook tutorial
  - Intro to programming in Julia.
  - Differences between Julia and Matlab / Python.
  - Beer Tasting Experiment
  - Savage-Dickey ratios

In 1937, one of the founders of the field of statistics, Ronald Fisher, published a story of how he explained inference to a friend. This story, called the "Lady Tasting Tea", has been re-told many times in different forms. In this notebook, we will re-tell one of its modern variants and introduce you to some important concepts along the way. Note that none of the material used below is new; you have all heard this in the theory lectures. The point of the Probabilistic Programming sessions is to solve practical problems so that concepts from theory become less abstract and you develop an intuition for them.

First, let's get started with activating a Julia workspace and importing some modules.

```
using Pkg
Pkg.activate("./workspace/")
Pkg.instantiate();
```

Code notes:

- The code cell above activates a specific Julia workspace (a virtual environment) that lists all packages you will need for the Probabilistic Programming sessions. The first time you run this cell, it will download and install all packages automatically.

```julia
using Distributions
using Plots
pyplot();
```

Code notes:

- `using` is how you import libraries and modules in Julia. Here we have imported a library of probability distributions called Distributions.jl and a library of plotting utilities called Plots.jl.

Beer Tasting Experiment

In the summer of 2017, students of the University of Amsterdam participated in a "Beer Tasting Experiment" (Doorn et al., 2019). Each participant was given two cups and were told that the cups contained Hefeweissbier, one with alcohol and one without. The participants had to taste each beer and guess which of the two contained alcohol.

We are going to do a statistical analysis of the tasting experiment. We want to know to what degree participants are able to discriminate between the alcoholic and alcohol-free beers. The Bayesian approach is about 3 core steps: (1) specifying a model, (2) absorbing the data through inference (parameter estimation), and (3) evaluating the model. We are going to walk through these steps in detail below.

1. Model Specification

Model specification consists of two parts: a likelihood function and a prior distribution.

Likelihood

A likelihood function is a function of parameters given observed data.

Here, we have an event variable X that indicates that the choice was either "correct", which we will assign the number 1, or "incorrect", which we will assign the number 0. We can model this choice with what's known as a Bernoulli distribution. The Bernoulli distribution is a formula to compute the probability of a binary event. It has a "rate parameter" θ, a number between 0 and 1, which governs the probability of the two events. If $\theta = 1$, then the participant will always choose the right cup ("always" = "with probability 1") and if $\theta = 0$, then the participant will never choose the right cup ("never" = "with probability 0"). Choosing at random, i.e. getting as many correct choices as incorrect choices, corresponds to $\theta = 0.5$.

As stated above, we are using the Bernoulli distribution in our tasting experiment. As the Bernoulli distribution's rate parameter θ increases, the event $X = 1$, i.e. the participant correctly guesses the alcoholic beverage, becomes more probable. The formula for the Bernoulli distribution is:

$$\begin{aligned} p(X = x \mid \theta) &= \text{Bernoulli}(x \mid \theta) \\ &= \theta^x (1-\theta)^{1-x} \end{aligned}$$

If $X = 1$, then the formula simplifies to $p(X = 1 \mid \theta) = \theta^1 (1-\theta)^{1-1} = \theta$. For $X = 0$, it simplifies to $p(X = 0 \mid \theta) = \theta^0 (1-\theta)^{1-0} = 1 - \theta$. If you have multiple independent observations, e.g. a data set $\mathcal{D} = \{X_1, X_2, X_3\}$, you can get the probability of all observations by taking the product of individual probabilities:

$$p(\mathcal{D} \mid \theta) = \prod_{i=1}^{N} p(X_i \mid \theta)$$

As an example, suppose the first two participants have correctly guessed the beverage and a third one incorrectly guessed it. Then, the probability under $\theta = 0.8$ is

$$p(\mathcal{D} = \{1, 1, 0\} \mid \theta = 0.8) = 0.8 \cdot 0.8 \cdot 0.2 = 0.128\,.$$

That is larger than the probability under $\theta = 0.4$, which is

$$p(\mathcal{D} = \{1, 1, 0\} \mid \theta = 0.4) = 0.4 \cdot 0.4 \cdot 0.6 = 0.096\,.$$

But it is not as large as the probability under $\theta = 0.6$, which is

$$p(\mathcal{D} = \{1, 1, 0\} \mid \theta = 0.6) = 0.6 \cdot 0.6 \cdot 0.4 = 0.144\,.$$

As you can see, the likelihood function tells us how well each value of the parameter fits the observed data. In short, how "likely" each parameter value is.

Prior Distribution

In Bayesian inference, it is important to think about what kind of prior knowledge you have about your problem. In our tasting experiment, this corresponds to what you think the probability is that a participant will correctly choose the cup. In other words, you have some thoughts about what value θ is in this scenario. You might think that the participants' choices are all going to be roughly random. Or, given that you have tasted other types of alcohol-free beers before, you might think that the participants are going to choose the right cup most of the time. This intuition, this "prior knowledge", needs to be quantified. We do that by specifying another probability distribution for it, in this case the Beta distribution:

$$\begin{aligned} p(\theta) &= \text{Beta}(\theta \mid \alpha, \beta) \\ &= \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1-\theta)^{\beta-1} \, . \end{aligned}$$

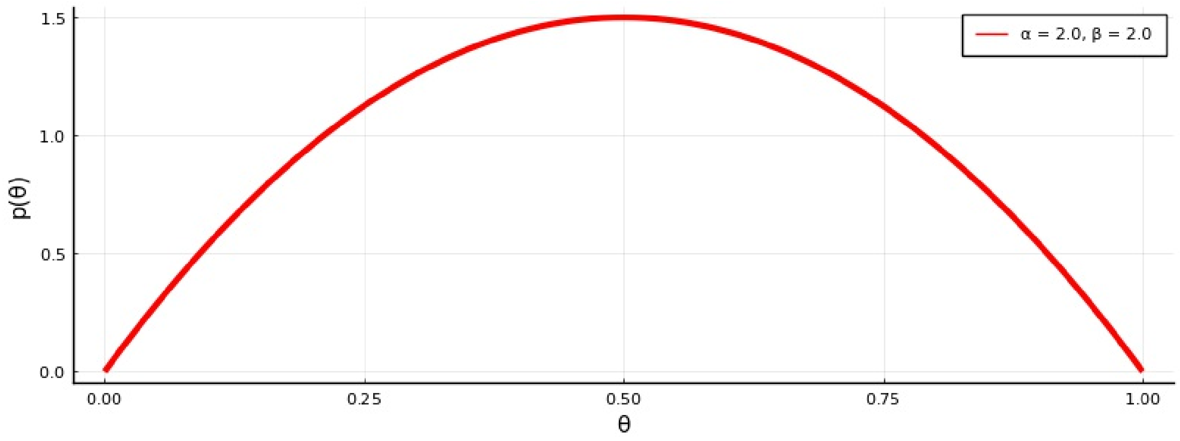
We use a Beta distribution to describe our state of knowledge about appropriate values for θ.

The Beta distribution computes the probability of an outcome in the interval $[0, 1]$. Like any other other distribution, it has parameters: α and β. Both are "shape parameters", meaning the distribution has a different shape for each value of the parameters. Let's visualise this!

```
x = [1 2 3;
     4 5 6]
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

```
# Define shape parameters
α = 2.0
β = 2.0

# Define probability distribution
pθ = Beta(α, β)

# Define range of values for θ
θ = range(0.0, step=0.01, stop=1.0)

# Visualize probability distribution function
plot(θ, pdf.(pθ, θ),
     linewidth=3,
     color="red",
     label="α = "*string(α)*", β = "*string(β),
     xlabel="θ",
     ylabel="p(θ)",
     size=(800,300))
```

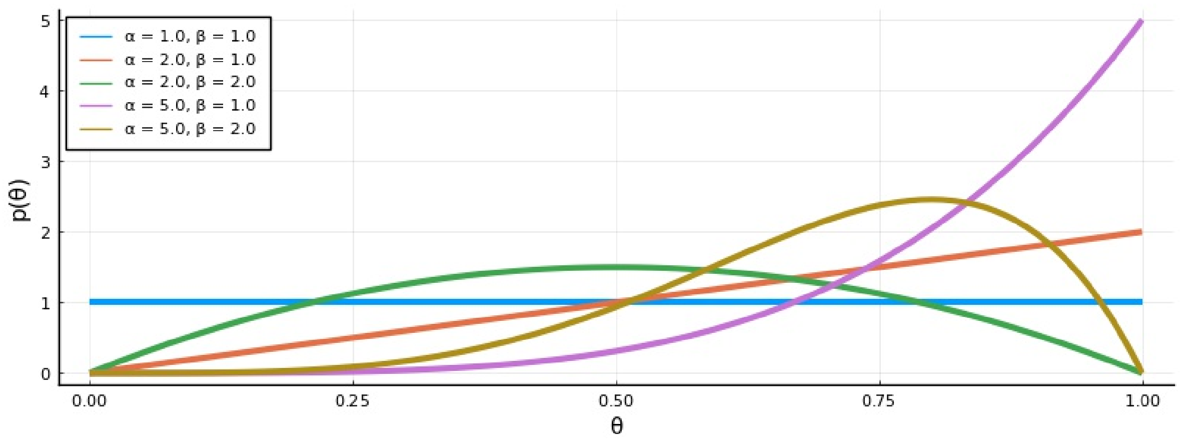


Code notes:

- You can use greek letters as variables (write them like in latex, e.g. \alpha, and press `tab`)
- Ranges of numbers work just like they do in Matlab (e.g. `0.0:0.1:1.0`) and Python (e.g. `range(0.0, stop=100., length=100)`). Note that Julia is strict about types, e.g. using integers vs floats.
- There is a `.` after the command `pdf`. This refers to "broadcasting": the function is applied to each element of a list or array. Here we use the `pdf` command to compute the probability for each value of θ in the array.
- Many of the keyword arguments in the `plot` command should be familiar to you if you've worked with Matplotlib (Python's plotting library).
- In the `label=` argument to plots, we have performed "string concatenation". In Julia, you write a string with double-quote characters and concatenate two strings by "multiplying", i.e. using `*`.

Note for the keen observers among you: since this is a continuous distribution, we are not actually plotting "probability", but rather "probability density" (probability densities can be larger than 1).

```julia
# Define shape parameters
α = [2.0, 5.0]
β = [1.0, 2.0]

# Define initial distribution
pθ = Beta(1.0, 1.0)

# Start initial plot
plot(θ, pdf.(pθ, θ), linewidth=3, label="α = 1.0, β = 1.0", xlabel="θ", ylabel="p(θ)", legend=:top
left)

# Loop over shape parameters
for a in α
    for b in β
        plot!(θ, pdf.(Beta(a, b), θ), linewidth=3, label="α = "*string(a)*", β = "*string(b))
    end
end
plot!(size=(800,300))
```



Code notes:

- Square brackets around numbers automatically creates an Array (in Python, they create lists).
- The `:` in `:topleft` indicates a `Symbol` type. It has many uses, but here it is used synonymously with a string option (e.g. legend="topleft").
- `for` loops can be done by using a range, such as `for i = 1:10` (like in Matlab), or using a variable that iteratively takes a value in an array, such as `for i in [1,2.3]` (like in Python). More here.
- The `!` at the end of the plot command means the function is performed "in-place". In other words, it changes its input arguments. Here, we change the plot by adding lines.
- The final `plot!` is there to ensure Jupyter actually plots the figure. If you end a cell on an `end` command, Jupyter will remain silent.

As you can see, the Beta distribution is quite flexible and can capture your belief about how often participants will correctly detect the alcoholic beverage. For example, the purple line indicates that you believe that it is very probable that participants will always get it right (peak lies on $\theta = 1.0$), but you still think there is some probability that the participants will guess at random ($p(\theta = 1/2) \approx 0.3$). The yellow-brown line indicates you believe that it is nearly impossible that the participants will always get it right ($p(\theta = 1) \approx 0.0$), but you still believe that they will get it right more often than not (peak lies around $\theta \approx 0.8$).

In summary: a prior distribution $p(\theta)$ reflects our beliefs about good values for parameter θ before data is observed.

---

* Try for yourself

I want you to pick values for the shape parameters α and β that reflect how often you think the participants will get it right.

---

2. Parameter estimation

Now that we have specified our generative model, it is time to estimate unknown variables. We'll first look at the data and then the inference part.

Data

The data of the participants in Amsterdam is available online at the Open Science Foundation. We'll start by reading it in.

```julia
using DataFrames
using CSV
```

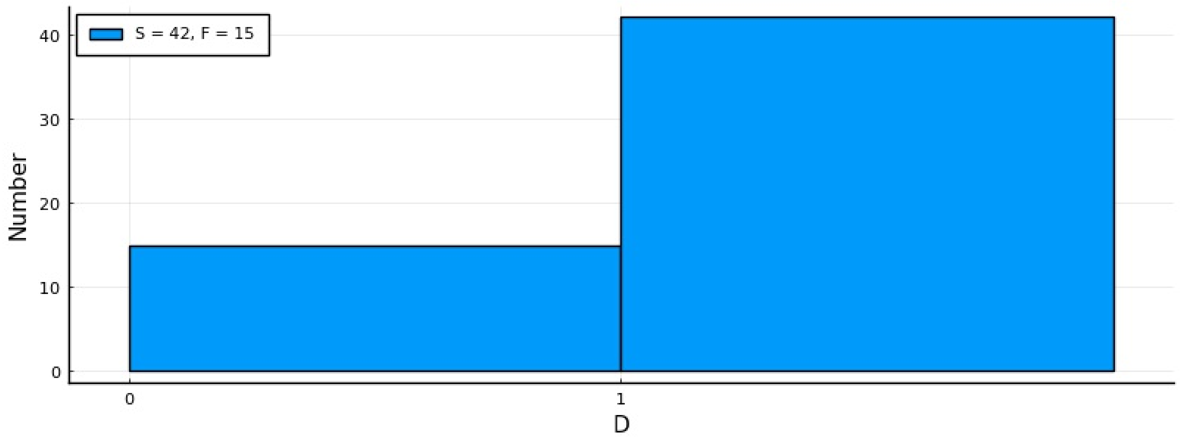
Code notes:

- CSV.jl is a library for reading in data stored in tables.
- DataFrames.jl manipulates table data (like `pandas` in Python).

```
# Read data from CSV file
data = DataFrame(CSV.File("../datasets/TastingBeerResults.csv"))

# Extract variable indicating correctness of guess
D = data[!, :CorrectIdentify];
println("D = ", D)
```

```
D = [1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0,
 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1]
```

```
# Number of successes and failures
S = sum(D .== 1)
F = sum(D .== 0)

# Visualize frequencies
histogram(D, bins=[0,1,2], label="S = "*string(S)*", F = "*string(F), xlabel="D", xticks=[0,1], yl
abel="Number", legend=:topleft, size=(800,300))
```

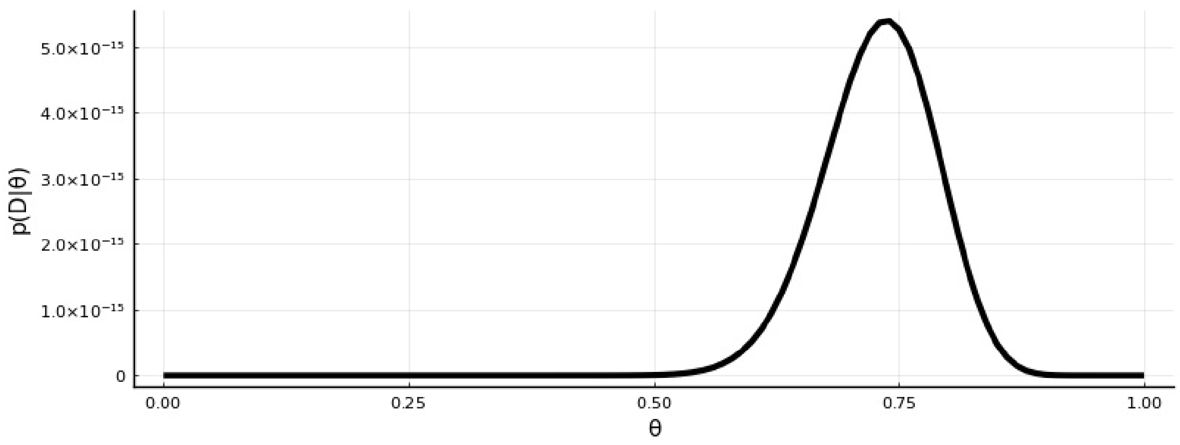


Code notes:

- The `!` in `data[!,` is specific to the DataFrames syntax.
- The `.==` checks for each element of the array D whether it is equal.

Let's visualize the likelihood of these observations.

```
# Define the Bernoulli likelihood function
likelihood(θ) = prod([θ^X_i * (1-θ)^(1-X_i) for X_i in D])

# Plot likelihood
plot(θ, likelihood.(θ), linewidth=3, color="black", label="", xlabel="θ", ylabel="p(D|θ)", size=(8
00,300))
```



The likelihood has somewhat of a bell shape, peaking just below $\theta = 0.75$. Note that the y-axis is very small. Indeed, the likelihood is not a proper probability distribution, because it doesn't integrate / sum to 1.

Inference

Using our generative model, we can estimate parameters for unknown variables. Remember Bayes' rule:

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta) p(\theta)}{p(\mathcal{D})} \, .$$

The posterior $p(\theta \mid \mathcal{D})$ equals the likelihood $p(\mathcal{D} \mid \theta)$ times the prior $p(\theta)$ divided by the evidence $p(\mathcal{D})$. In our tasting experiment, we have a special thing going on: conjugacy. The Beta distribution is "conjugate" to the Bernoulli likelihood, meaning that the posterior distribution is also going to be a Beta distribution. Specifically with the Beta-Bernoulli combination, it is easy to see what conjugacy actually means. Recall the formula for the Beta distribution:

$$p(\theta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha - 1} (1 - \theta)^{\beta - 1} \, .$$

The term $\Gamma(\alpha + \beta) / \left(\Gamma(\alpha)\Gamma(\beta)\right)$ normalises this distribution. If you ignore that and multiply it with the likelihood, you get something that simplifies beautifully:

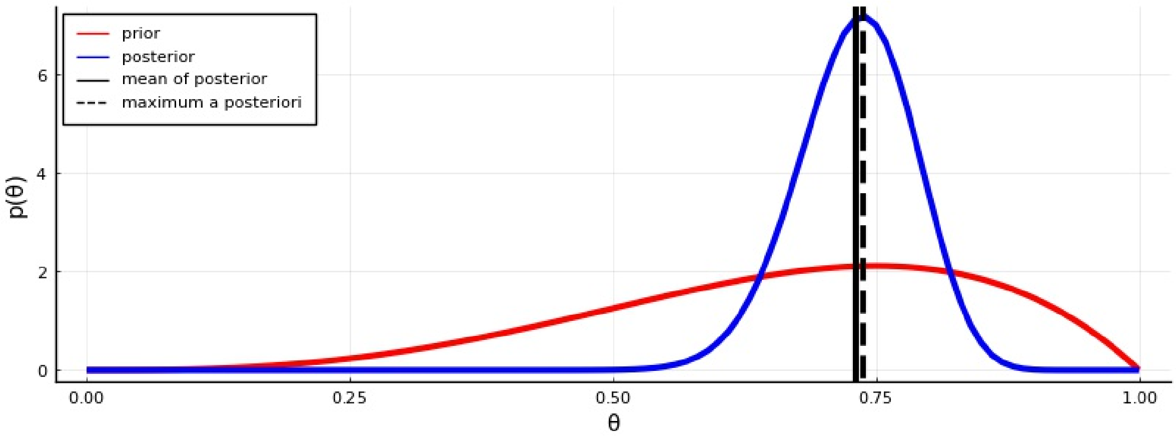
$$\begin{aligned} p(\mathcal{D} \mid \theta) p(\theta) &\propto \prod_{i=1}^{N} \left[\theta^{X_i} (1 - \theta)^{1 - X_i}\right] \cdot \theta^{\alpha - 1} (1 - \theta)^{\beta - 1} \\ &= \theta^{\sum_{i=1}^{N} X_i} (1 - \theta)^{\sum_{i=1}^{N} 1 - X_i} \cdot \theta^{\alpha - 1} (1 - \theta)^{\beta - 1} \\ &= \theta^{S} (1 - \theta)^{F} \cdot \theta^{\alpha - 1} (1 - \theta)^{\beta - 1} \\ &= \theta^{S + \alpha - 1} (1 - \theta)^{F + \beta - 1} \, , \end{aligned}$$

where $S = \sum_{i=1}^{N} X_i$ is the number of successes (correct guesses) and $F = \sum_{i=1}^{N} 1 - X_i$ is the number of failures (incorrect guesses).

This last line is again the formula for the Beta distribution (except for a proper normalisation) but with different parameters ($S + \alpha$ instead of α and $F + \beta$ instead of β). This is what we mean by conjugacy: applying Bayes rule to a conjugate prior and likelihood pair yields a posterior distribution of the same family as the prior, which in this case is a Beta distribution.

Let's now visualise the posterior after observing the data from Amsterdam.

```
# Define shape parameters of prior distribution
α0 = 4.0
β0 = 2.0

# Define prior distribution
pθ = Beta(α0, β0)

# Update parameters for the posterior
αN = α0 + sum(D .== 1)
βN = β0 + sum(D .== 0)

# Define posterior distribution
pθD = Beta(αN, βN)

# Mean of posterior
mean_post = αN / (αN + βN)
mode_post = (αN - 1) / (αN + βN - 2)

# Visualize probability distribution function
plot(θ, pdf.(pθ, θ), linewidth=3, color="red", label="prior", xlabel="θ", ylabel="p(θ)")
plot!(θ, pdf.(pθD, θ), linewidth=3, color="blue", label="posterior", size=(800,300))
vline!([mean_post], color="black", linewidth=3, label="mean of posterior", legend=:topleft)
vline!([mode_post], color="black", linewidth=3, linestyle=:dash, label="maximum a posteriori", leg
end=:topleft)
```



Code notes:

- `vline` draw a vertical line in the plot, at the specified point on the x-axis.
- `mode()` extracts the mode) of the supplied distribution, i.e. the point with the largest probability.

That looks great! We have updated our belief from a very broad prior to a much sharper posterior.

The posterior contains a lot of information: it tells us something about every value for θ. Sometimes, we are interested in a point estimate, i.e. the probability of a single value for θ under the posterior. Two well-known point estimators are the mean of the posterior and the mode (the value for θ with the highest probability). I have plotted both point estimates in the figure above. In this case, they are nearly equal.

* Try for yourself

Plug the shape parameters of your prior into a copy of the cell above and see how your posterior differs.

3. Model Evaluation

Given our model assumptions and a posterior for theta, we can now make quantitative predictions about how well we think people can recognize alcoholic from non-alcoholic hefeweizen. But suppose you meet someone else who is absolutely sure that people can't tell the difference. Can you say something about the probability of his belief, given the experiment?

Technically, this is a question about comparing the performance of different models. Model comparison is also known in the statistical literature as "hypothesis testing".

In hypothesis testing, you start with a null hypothesis $\mathcal{H}_0$, which is a particular choice for the detection parameter θ. In the question above, the other person's belief corresponds to $\theta = 0.5$. We then have an alternative hypothesis $\mathcal{H}_1$, namely that his belief is wrong, i.e. $\theta \neq 0.5$. From a Bayesian perspective, hypothesis testing is just about comparing the posterior beliefs about these two hypotheses:

$$\underbrace{\frac{p(\mathcal{H}_1|\mathcal{D})}{p(\mathcal{H}_0|\mathcal{D})}}_{\text{Posterior belief over hypotheses}} = \underbrace{\frac{p(\mathcal{H}_1)}{p(\mathcal{H}_0)}}_{\text{Prior belief over hypotheses}} \times \underbrace{\frac{p(\mathcal{D}|\mathcal{H}_1)}{p(\mathcal{D}|\mathcal{H}_0)}}_{\text{Likelihood of hypotheses}} \, .$$

Note that the evidence term $p(\mathcal{D})$ is missing, because it appears in the posterior for both hypotheses and therefore cancels out. The hypothesis likelihood ratio is also called the Bayes factor. Bayes factors can be hard to compute, but in some cases we can simplify it: if the null hypothesis is a specific value of interest, for instance $\theta = 0.5$, and the alternative hypothesis is not that specific value, e.g. $\theta \neq 0.5$, then the factor reduces to what's known as a Savage-Dickey Ratio (see Appendix A of Wagemakers et al., 2010):

$$\frac{p(\mathcal{D}|\mathcal{H}_1)}{p(\mathcal{D}|\mathcal{H}_0)} = \frac{p(\theta = 0.5)}{p(\theta = 0.5 \mid \mathcal{D})} \, .$$

This compares the probability of $\theta = 0.5$ under the prior versus $\theta = 0.5$ under the posterior. It effectively tells you how much your belief changes after observing the data. Let's compute the Savage-Dickey ratio for our experiment:

```
BF_10 = pdf(pθ, 0.5) / pdf(pθD, 0.5)
println("The Bayes factor for H1 versus H0 = "*string(BF_10))
```

```
The Bayes factor for H1 versus H0 = 229.23178145896927
```

So, in the experiment, the alternative hypothesis "students can discriminate alcoholic from non-alcoholic Hefeweissbier" is more than 200 times more probable than the null hypothesis that "students cannot discriminate alcoholic from non-alcoholic Hefeweissbier".

* Try for yourself

Compute the Bayes factor for your prior and posterior distribution. How many times is the alternative hypothesis more probable than the null hypothesis?

Probabilistic Programming 2: Message Passing & Analytical Solutions

Goal

- Understand when and how analytical solutions to Bayesian inference can be obtained.
- Understand how to perform message passing in a Forney-style factor graph.

Materials

- Mandatory
  - This notebook
  - Lecture notes on factor graphs
  - Lecture notes on continuous data
  - Lecture notes on discrete data
- Optional
  - Chapters 2 and 3 of Model-Based Machine Learning.
  - Differences between Julia and Matlab / Python.

Note that none of the material below is new. The point of the Probabilistic Programming sessions is to solve practical problems so that the concepts from Bert's lectures become less abstract.

```
using Pkg
Pkg.activate("./workspace/")
Pkg.instantiate();
```

```
using LinearAlgebra
using SpecialFunctions
using ForneyLab
using PyCall
using Plots
pyplot();
```

We'll be using the toolbox ForneyLab.jl to visualize factor graphs and compute messages passed within the graph.

Problem: A Job Interview

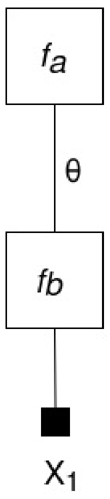
After you finish your master's degree, you will need to start looking for jobs. You will get one or more job interviews and some will be fun while others will be frustrating. The company you applied at wants a talented and skilled employee, but measuring a person's skill is tricky. Even a highly-skilled person makes mistakes and people with few skills can get lucky. In this session, we will look at various ways to assess skills using questions and test assignments. Along the way, you will gain experience with message passing, factor graphs and working with discrete vs continuous data.

1: Right or wrong

Suppose you head to a job interview for a machine learning engineer position. The company is interested in someone who knows Julia and has set up a test with syntax questions. We will first look at a single question, which we treat as an outcome variable X_1. You can either get this question right or wrong, which means we're dealing with a Bernoulli likelihood. The company assumes you have a skill level, denoted θ, and the higher the skill, the more likely you are to get the question right. Since the company doesn't know anything about you, they chose an uninformative prior distribution: the Beta(1,1). We can write the generative model for answering this question as follows:

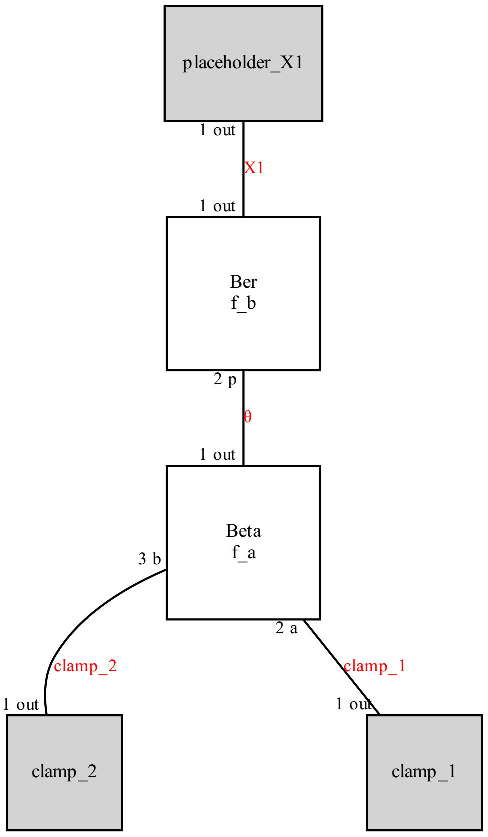
$$\begin{aligned} p(X_1, \theta) &= p(X_1 \mid \theta) \cdot p(\theta) \\ &= \text{Bernoulli}(X_1 \mid \theta) \cdot \text{Beta}(\theta \mid \alpha = 1, \beta = 1)\,. \end{aligned}$$

The factor graph for this model is:



where $f_b(X_1, \theta) \triangleq \text{Bernoulli}(X_1 \mid \theta)$ and $f_a(\theta) \triangleq \text{Beta}(\theta \mid 1, 1)$. We are now going to construct this factor graph using the toolbox ForneyLab.

```julia
# Start building a model by setting up a FactorGraph structure
factor_graph1 = FactorGraph()

# Add the prior over
@RV θ ~ Beta(1.0, 1.0, id=:f_a)

# Add the question correctness likelihood
@RV X1 ~ Bernoulli(θ, id=:f_b)

# The outcome X1 is going to be observed, so we set up a placeholder for the data entry
placeholder(X1, :X1)

# Visualize the graph
ForneyLab.draw(factor_graph1)
```



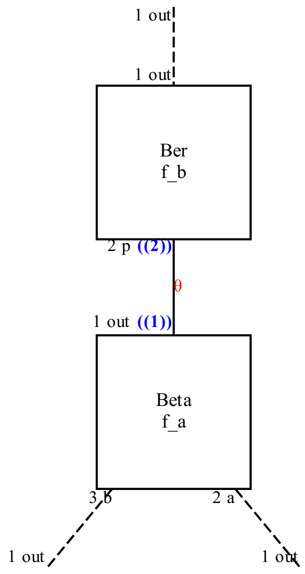
Code notes:

- @RV is a macro that lets you add Random Variables as nodes to your factor graph.
- The symbol ~ means "is distributed as". For example, $\theta \sim \text{Beta}(1,1)$ should be read as "θ is distributed according to a Beta(θ | a=1, b=1) probability distribution".

Above you can see the factor graph that ForneyLab has generated. It is not as clean as the ones in the theory lectures. For example, ForneyLab generates nodes for the clamped parameters of the Beta prior ($\alpha = 1$ and $\beta = 1$), while we ignore these in the manually constructed graphs. Nonetheless, ForneyLab's version is very useful for debugging later on.

We are now going to tell ForneyLab to generate a message passing procedure for us.

```julia
# Indicate which variables you want posteriors for
q = PosteriorFactorization(θ, ids=[:θ])

# Generate a message passing inference algorithm
algorithm = messagePassingAlgorithm(θ, q)

# Compile algorithm code
source_code = algorithmSourceCode(algorithm)

# Bring compiled code into current scope
eval(Meta.parse(source_code))

# Visualize message passing schedule
pfθ = q.posterior_factors[:θ]
ForneyLab.draw(pfθ, schedule=pfθ.schedule);
```



Code notes:

- ForneyLab.jl compiles the specified model and inference procedure into a string. This string is human-readable and portable across devices. The functions `eval(Meta.parse())` are used to bring that string into the current scope, so the generated code can be used.
- In `ForneyLab.draw()`, only the edge of interest is shown with the two connecting nodes and their inputs. All other parts of the graph are ignored.

ForneyLab's visualization of the message passing procedure for a specific variable isolates that variable in the graph and shows where the incoming messages come from. In this case, we are interested in θ (your skill level), which receives message ((2)) from the likelihood node (the "Ber" node above θ) and message ((1)) from the prior node (the "Beta" node below θ).

In the message passing framework, the combination of these two messages produces the "marginal" distribution for θ. We are using message passing to do Bayesian inference, so note that the "marginal" for θ corresponds to the posterior distribution $p(\theta \mid X_1)$.

Let's inspect these messages.

```
# Initialize data structure for messages
messages = Array{Message}(undef, 2)

# Initalize data structure for marginal distributions
marginals = Dict()

# Suppose you got question 1 correct
data = Dict(:X1 => 1)

# Update coefficients
stepθ!(data, marginals, messages);

# Print messages
print("\nMessage ((1)) = "*string(messages[1].dist))
println("Message ((2)) = "*string(messages[2].dist))
```

```
Message ((1)) = Beta(a=1.00, b=1.00)
Message ((2)) = Beta(a=2.00, b=1.00)
```

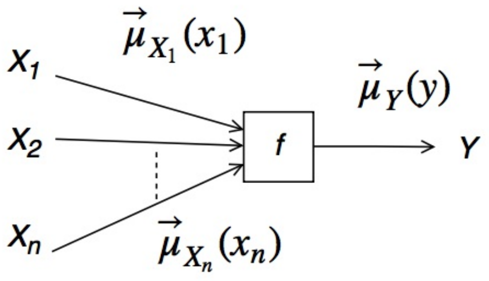
Code notes:

- A `Dict` is a dictionary data structure. In the `marginals` dictionary we only have one entry: the key is the variable `θ` (as a Symbol, i.e. as `:θ`) and the value is a `ProbabilityDistribution` object. It is the initial distribution for that variable. In the `data` dictionary, we also only have one entry: the key is the variable `X1` and the value is a Float. This is because `X1` is observed. We know its value without uncertainty.
- The `stepθ!` function comes from the algorithm compilation.

Alright. So, they are both Beta distributions. Do they actually make sense? Where do these parameters come from?

Recall from the lecture notes that the formula for messages sent by factor nodes is:

$$\boxed{\underbrace{\overrightarrow{\mu}_Y(y)}_{\substack{\text{outgoing}\\ \text{message}}} = \sum_{x_1,\ldots,x_n} \underbrace{\overrightarrow{\mu}_{X_1}(x_1)\cdots \overrightarrow{\mu}_{X_n}(x_n)}_{\substack{\text{incoming}\\ \text{messages}}} \cdot \underbrace{f(y,x_1,\ldots,x_n)}_{\substack{\text{node}\\ \text{function}}}}$$



The prior node is not connected to any other unknown variables and so does not receive incoming messages. Its outgoing message is therefore:

$$\begin{aligned}\overrightarrow{\mu}(\theta) &= f(\theta) & (1)\\ &= \text{Beta}(\theta \mid 1, 1)\,. & (2)\end{aligned}$$

So that confirms the correctness of Message ((1)).

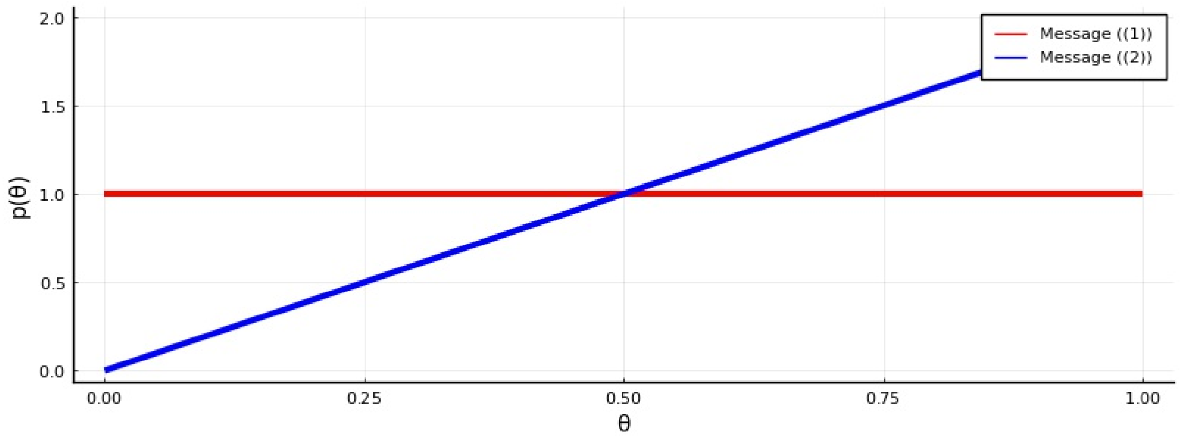
Similarly, we can also derive the message from the likelihood node by hand. For this, we need to know that the message coming from the observation $\overleftarrow{\mu}(x)$ is a delta function, which, if you gave the right answer ($X_1 = 1$), has the form $\delta(X_1 - 1)$. The "node function" is the Bernoulli likelihood $\text{Bernoulli}(X_1 \mid \theta)$. Another thing to note is that this is essentially a convolution with respect to a delta function and that its sifting property holds: $\int_{X_1} \delta(X_1 - x)\ f(X_1, \theta)\mathrm{d}X_1 = f(x, \theta)$. The fact that X_1 is a discrete variable instead of a continuous one, does not negate this. Using these facts, we can perform the message computation by hand:

$$\begin{aligned}\overleftarrow{\mu}(\theta) &= \sum_{X_1} \overleftarrow{\mu}(X_1)\ f(X_1, \theta) & (3)\\ &= \sum_{X_1} \delta(X_1 - 1)\ \text{Bernoulli}(X_1 \mid \theta) & (4)\\ &= \sum_{X_1} \delta(X_1 - 1)\ \theta^{X_1}(1-\theta)^{1-X_1} & (5)\\ &= \theta^1(1-\theta)^{1-1}\,. & (6)\end{aligned}$$

Remember that the pdf of a Beta distribution is proportional to $\theta^{\alpha-1}(1-\theta)^{\beta-1}$. So, if you read the second-to-last line above as $\theta^{2-1}(1-\theta)^{1-1}$, then the outgoing message $\overleftarrow{\mu}(\theta)$ is proportional to a Beta distribution with $\alpha = 2$ and $\beta = 1$. So, our manual derivation verifies ForneyLab's Message ((2)).

Let's now look at these messages visually.

```
# Probability density function of a Beta distribution
Beta(θ, α, β) = 1/beta(α,β) * θ^(α-1) * (1-θ)^(β-1)

# Extract parameters from message ((1))
α1 = messages[1].dist.params[:a]
β1 = messages[1].dist.params[:b]

# Extract parameters from message ((2))
α2 = messages[2].dist.params[:a]
β2 = messages[2].dist.params[:b]

# Plot messages
θ_range = range(0, step=0.01, stop=1.0)
plot(θ_range, Beta.(θ_range, α1, β1), color="red", linewidth=3, label="Message ((1))", xlabel="θ",
 ylabel="p(θ)")
plot!(θ_range, Beta.(θ_range, α2, β2), color="blue", linewidth=3, label="Message ((2))", size=(800
,300))
```



The marginal distribution for θ, representing the posterior $p(\theta \mid X_1)$, is obtained by taking the product (followed by normalization) of the two messages: $\overrightarrow{\mu}(\theta) \cdot \overleftarrow{\mu}(\theta)$. Multiplying two Beta distributions produces another Beta distribution with parameter:

$$\alpha \leftarrow \alpha_1 + \alpha_2 - 1 \tag{7}$$
$$\beta \leftarrow \beta_1 + \beta_2 - 1\,, \tag{8}$$

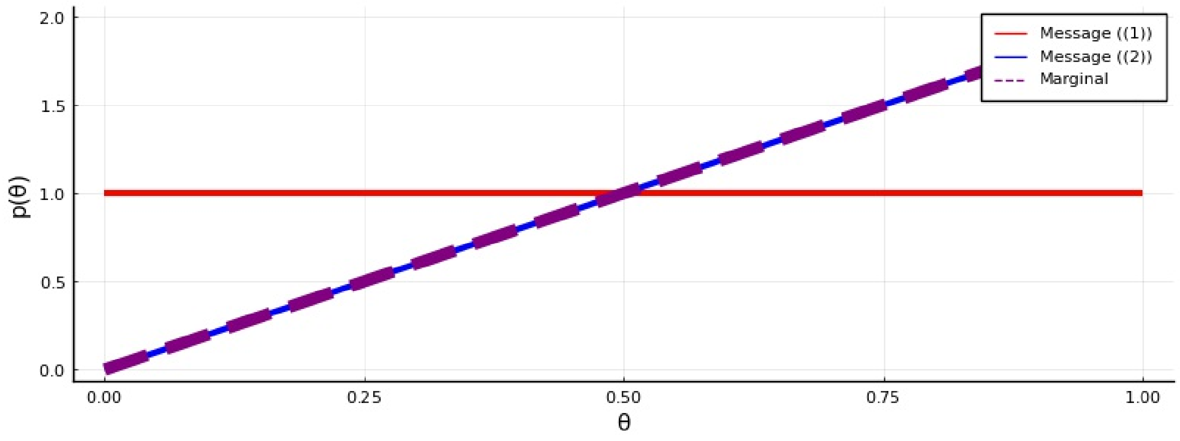
In our case, the new parameters would be $\alpha = 1 + 2 - 1 = 2$ and $\beta = 1 + 1 - 1 = 1$. Let's check with ForneyLab what it computed.

```
marginals[:θ]
```

```
    Beta(a=2.00, b=1.00)
```

Again, ForneyLab matches our manual derivations. Let's visualize the messages as well as the marginal.

```
# Extract marginal's parameters
α_marg = marginals[:θ].params[:a]
β_marg = marginals[:θ].params[:b]

# Plot messages
θ_range = range(0, step=0.01, stop=1.0)
plot(θ_range, Beta.(θ_range, α1, β1), color="red", linewidth=3, label="Message ((1))", xlabel="θ",
 ylabel="p(θ)")
plot!(θ_range, Beta.(θ_range, α2, β2), color="blue", linewidth=3, label="Message ((2))", size=(800
,300))
plot!(θ_range, Beta.(θ_range, α_marg, β_marg), color="purple", linewidth=6, linestyle=:dash, label
="Marginal")
```



The pdf of the marginal distribution lies on top of the pdf of Message ((2)). That's not always going to be the case; the Beta(1,1) distribution is special in that when you multiply Beta(1,1) with a general Beta(a,b) the result will always be Beta(a,b), kinda like multiplying by 1. We call prior distributions that have this special effect "non-informative priors".

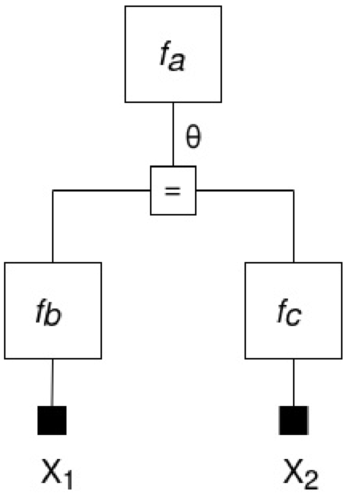
Multiple questions

Of course, you won't be evaluated on just a single question: it's still possible for you to get one question wrong even if you have a high skill level. You would consider it unfair to be rejected based on only one question. So, we are going to add another question. We're also going to change the prior: the company now assumes that you must have some skill if you applied for the position. This is reflected in a prior Beta distributions with $\alpha = 3.0$ and $\beta = 2.0$.

For now, the second question is also a right-or-wrong question. The outcome of this new question is denoted with variable X_2. With this addition, the generative model becomes

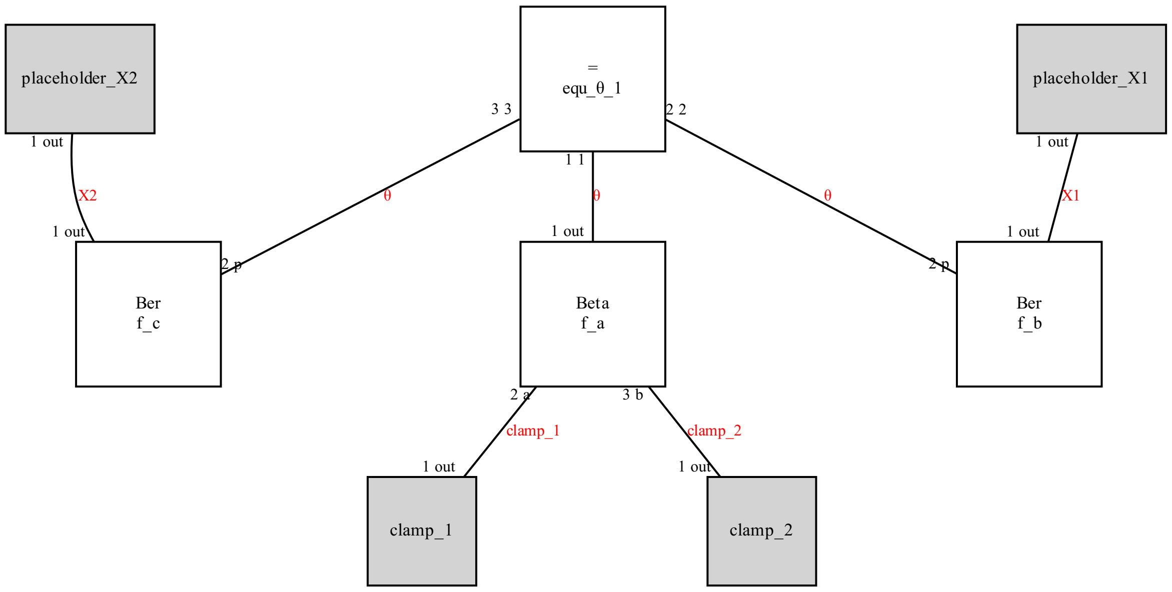
$$p(X_1, X_2, \theta) = p(X_1 \mid \theta) p(X_2 \mid \theta) p(\theta) \, ,$$

with the accompanying factor graph



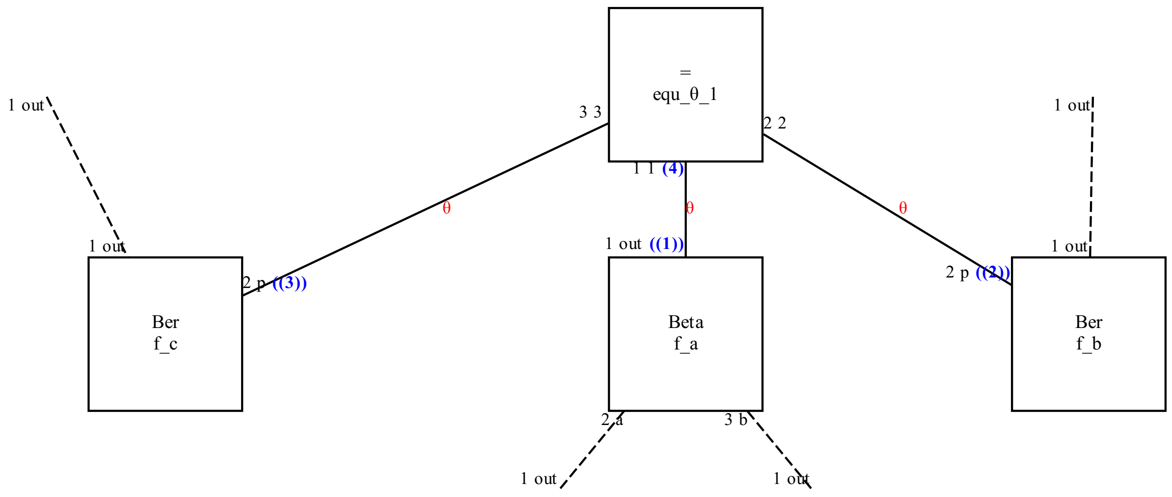
where $f_c \triangleq \text{Bernoulli}(X_2 \mid \theta)$ and f_a, f_b are still the same. Notice that we now have an equality node as well. That is because the variable θ is used in three factor nodes. ForneyLab automatically generates the same factor graph:

```
# Start building a model
factor_graph2 = FactorGraph()

# Add the prior
@RV θ ~ Beta(3.0, 2.0, id=:f_a)

# Add question 1 correctness likelihood
@RV X1 ~ Bernoulli(θ, id=:f_b)

# Add question 2 correctness likelihood
@RV X2 ~ Bernoulli(θ, id=:f_c)

# The question outcomes are going to be observed
placeholder(X1, :X1)
placeholder(X2, :X2)

# Visualize the graph
ForneyLab.draw(factor_graph2)
```



We will go through the message passing operations below. First, we generate an algorithm and visualize where all the messages for θ come from.

```
# Indicate which variables you want posteriors for
q = PosteriorFactorization(θ, ids=[:θ])

# Generate a message passing inference algorithm
algorithm = messagePassingAlgorithm(θ, q)

# Compile algorithm code
source_code = algorithmSourceCode(algorithm)

# Bring compiled code into current scope
eval(Meta.parse(source_code))

# Visualize message passing schedule
pfθ = q.posterior_factors[:θ]
ForneyLab.draw(pfθ, schedule=pfθ.schedule);
```



There are 4 messages, one from the prior ((1)), one from the first likelihood ((2)), one from the second likelihood ((3)) and one from the equality node ((4)). ForneyLab essentially combines messages 2 and 3 into message 4 and then multiplies messages 1 and 4 to produce the marginal. We can see this if we look in the source code:

```
println(source_code)
```

```
begin

function stepθ!(data::Dict, marginals::Dict=Dict(), messages::Vector{Message}=Array{Message
}(undef, 4))

messages[1] = ruleVBBetaOut(nothing, ProbabilityDistribution(Univariate, PointMass, m=3.0),
 ProbabilityDistribution(Univariate, PointMass, m=2.0))
messages[2] = ruleVBBernoulliIn1(ProbabilityDistribution(Univariate, PointMass, m=data[:X1]
), nothing)
messages[3] = ruleVBBernoulliIn1(ProbabilityDistribution(Univariate, PointMass, m=data[:X2]
), nothing)
messages[4] = ruleSPEqualityBeta(nothing, messages[2], messages[3])

marginals[:θ] = messages[1].dist * messages[4].dist

return marginals

end

end # block
```

You can see that `messages[4]` is a function of `messages[2]` and `messages[3]` and that `marginals[:θ]` is the product of `messages[1]` and `messages[4]`.

Suppose you got the first question right and the second question wrong. Let's execute the message passing procedure and take a look at the functional form of the messages.

```
# Initialize a message data structure
messages = Array{Message}(undef, 4)

# Initalize marginal distributions data structure
marginals = Dict()

# Suppose you got question 1 right and question 2 wrong
data = Dict(:X1 => 1,
            :X2 => 0)

# Update coefficients
stepθ!(data, marginals, messages);

# Print messages
print("\nMessage ((1)) = "*string(messages[1].dist))
print("Message ((2)) = "*string(messages[2].dist))
print("Message ((3)) = "*string(messages[3].dist))
println("Message ((4)) = "*string(messages[4].dist))
```

```
    Message ((1)) = Beta(a=3.00, b=2.00)
    Message ((2)) = Beta(a=2.00, b=1.00)
    Message ((3)) = Beta(a=1.00, b=2.00)
    Message ((4)) = Beta(a=2.00, b=2.00)
```

Messages ((1)) and ((2)) are clear, but Message ((3)) and Message ((4)) are new.

---

* Try for yourself

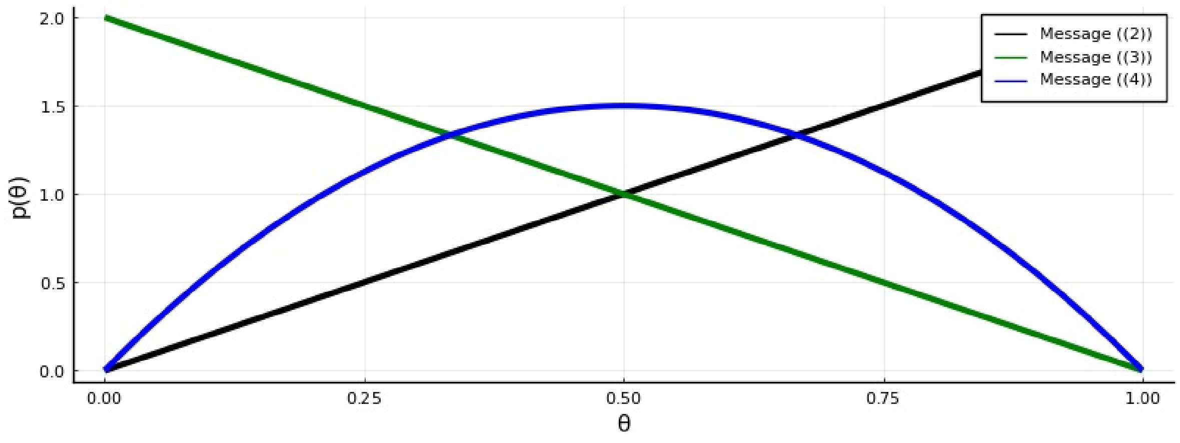
Try deriving the functional form of Message ((3)) for yourself. Tip: the derivation is very similar to that of Message ((2)). The most important change is to use $\delta(X_2 - 0)$ instead of $\delta(X_1 - 1)$.

Message ((4)) is the result of the standard message computation formula for the case of an equality node:

$$\begin{aligned}
\downarrow \mu(\theta) &= \sum_{\theta', \theta''} \overrightarrow{\mu}(\theta'')\ f_{=}(\theta, \theta', \theta'')\ \overleftarrow{\mu}(\theta') && (9)\\
&= \overrightarrow{\mu}(\theta) \cdot \overleftarrow{\mu}(\theta) && (10)\\
&= \text{Beta}(\theta \mid 2, 1) \cdot \text{Beta}(\theta \mid 1, 2) && (11)\\
&= \text{Beta}(\theta \mid 2, 2) \quad . && (12)
\end{aligned}$$

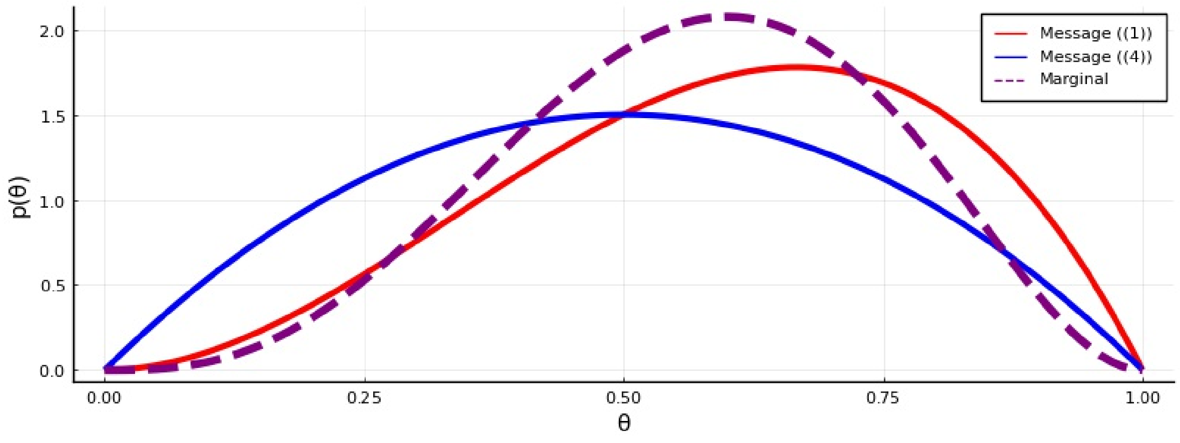
Let's visualize the messages and the marginal again.

```
# Extract parameters from message ((2))
α2 = messages[2].dist.params[:a]
β2 = messages[2].dist.params[:b]

# Extract parameters from message ((3))
α3 = messages[3].dist.params[:a]
β3 = messages[3].dist.params[:b]

# Extract parameters from message ((4))
α4 = messages[4].dist.params[:a]
β4 = messages[4].dist.params[:b]

plot(θ_range, Beta.(θ_range, α2, β2), color="black", linewidth=3, label="Message ((2))", xlabel="θ
", ylabel="p(θ)")
plot!(θ_range, Beta.(θ_range, α3, β3), color="green", linewidth=3, label="Message ((3))", size=(80
0,300))
plot!(θ_range, Beta.(θ_range, α4, β4), color="blue", linewidth=3, label="Message ((4))")
```



Message ((2)) and Message ((3)) are direct opposites: ((2)) increases the estimate and ((3)) decreases the estimate of your skill level. Message ((4)) end up being centered on 0.5. With one question right and one question wrong, you have essentially been guessing at random.

```
# Extract parameters from message ((1))
α1 = messages[1].dist.params[:a]
β1 = messages[1].dist.params[:b]

# Extract parameters from message ((4))
α4 = messages[4].dist.params[:a]
β4 = messages[4].dist.params[:b]

# Extract parameters
α_marg = marginals[:θ].params[:a]
β_marg = marginals[:θ].params[:b]

plot(θ_range, Beta.(θ_range, α1, β1), color="red", linewidth=3, label="Message ((1))", xlabel="θ",
 ylabel="p(θ)")
plot!(θ_range, Beta.(θ_range, α4, β4), color="blue", linewidth=3, label="Message ((4))", size=(800
,300))
plot!(θ_range, Beta.(θ_range, α_marg, β_marg), color="purple", linewidth=4, linestyle=:dash, label
="Marginal")
```



If we now combine the prior (Message ((1)) in red above) with the combined message from both likelihood terms (Message ((4)) in blue), we get the new marginal (purple dotted line). The mean of the marginal lies above 0.5, which is due to the prior assumption that you must have some skill if you applied.

2. Score questions

So far, the models we have been looking at have been quite simple; they are Beta-Bernoulli combinations which is exactly what we did for the Beer Tasting Experiment. We will now move on to more complicated distributions. These will enrich your toolbox and allow you to do much more.

Suppose you are not tested on a right-or-wrong question, but on a score question. For instance, you have to complete a piece of code for which you get a score. If all of it was wrong you get a score of 0, if some of it was correct you get a score of 1 and if all of it was correct you get a score 2. That means we have a likelihood with three outcomes: $X_1 = \{0, 1, 2\}$. Suppose we once again ask two questions, X_1 and X_2. The order in which we ask these questions does not matter, so that means we choose Categorical distributions for these likelihood functions: $X_1, X_2 \sim \text{Categorical}(\theta)$. The parameter θ is no longer a single parameter, indicating the probability of getting the question right, but a vector of three parameters: $\theta = (\theta_1, \theta_2, \theta_3)$. Each θ_k indicates the probability of getting the k-th outcome. In other words, θ_1 indicates the probability of getting 0 points, θ_2 of getting 1 point and θ_3 of getting two points. A highly-skilled applicant will have a parameter vector of $(0.05, 0.1, 0.85)$, for example. The prior distribution conjugate to the Categorical distribution is the Dirichlet distribution.

Let's look at the generative model:

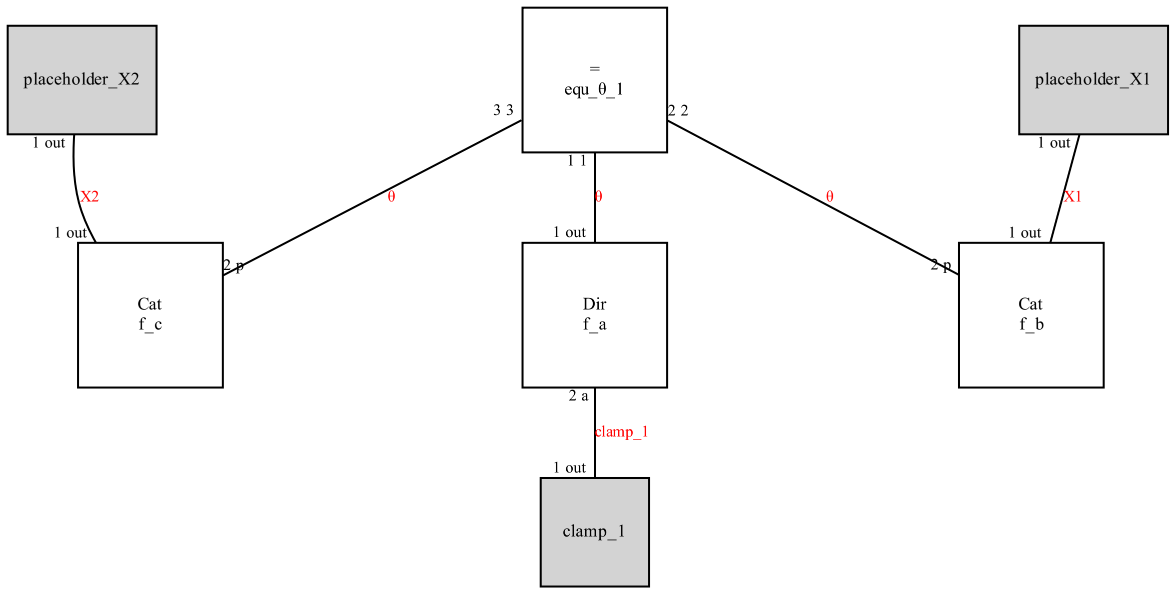
$$p(X_1, X_2, \theta) = p(X_1 \mid \theta)p(X_2 \mid \theta)p(\theta)\,.$$

It's the same as before. The only difference is that:

$$\begin{aligned} p(X_1 \mid \theta) &= \text{Categorical}(X_1 \mid \theta) && (13) \\ p(X_2 \mid \theta) &= \text{Categorical}(X_2 \mid \theta) && (14) \\ p(\theta) &= \text{Dirichlet}(\theta) && (15) \end{aligned}$$

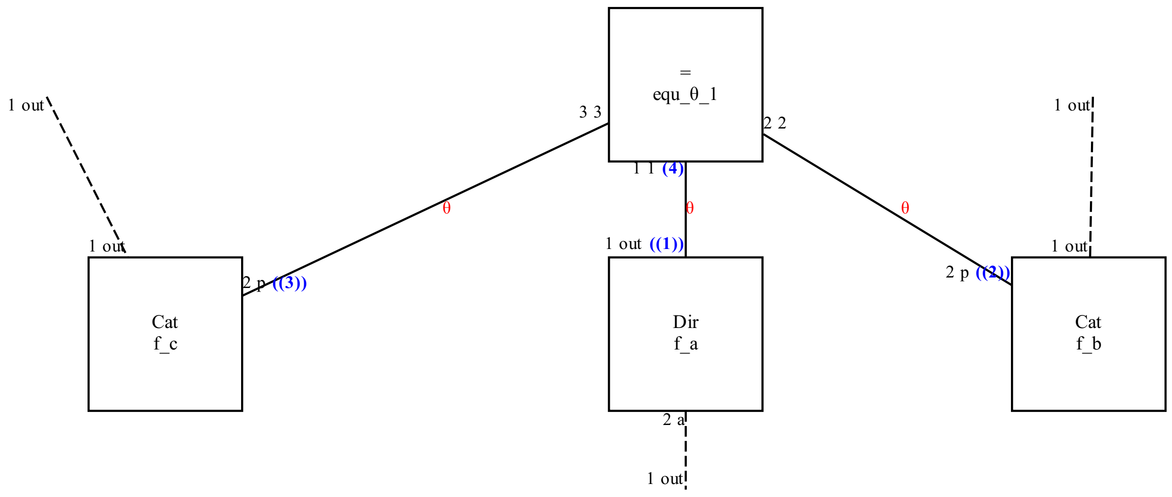
The factor graph has the same structure as before. The only change is that the factor nodes f_a, f_b, f_c are now parameterized differently.

```
# Start building a model
factor_graph3 = FactorGraph()

# Add the prior
@RV θ ~ Dirichlet([1.0, 3.0, 2.0], id=:f_a)

# Add question 1 correctness likelihood
@RV X1 ~ Categorical(θ, id=:f_b)

# Add question 2 correctness likelihood
@RV X2 ~ Categorical(θ, id=:f_c)

# The question outcomes are going to be observed
placeholder(X1, dims=(3,), :X1)
placeholder(X2, dims=(3,), :X2)

# Visualize the graph
ForneyLab.draw(factor_graph3)
```



The only difference with the previous graph is the fact that the node called "prior" is a 'Dir', short for Dirichlet, and that the two nodes called "likelihood1" and "likelihood2" are 'Cat' types, short for Categorical. Let's look at the message passing schedule:

```
# Indicate which variables you want posteriors for
q = PosteriorFactorization(θ, ids=[:θ])

# Generate a message passing inference algorithm
algorithm = messagePassingAlgorithm(θ, q)

# Compile algorithm code
source_code = algorithmSourceCode(algorithm)

# Bring compiled code into current scope
eval(Meta.parse(source_code))

# Visualize message passing schedule
pfθ = q.posterior_factors[:θ]
ForneyLab.draw(pfθ, schedule=pfθ.schedule);
```



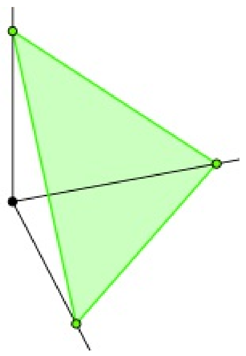
That's the same as before as well: 2 messages from the likelihoods, 1 combined likelihood message from the equality node and 1 message from the prior.

If we now setup the message passing procedure, we have to be a little bit more careful. We cannot feed the scores $\{0, 1, 2\}$ as outcomes directly. We have to encode them in one-hot vectors (see Bert's lecture notes on discrete distributions). Suppose you had a score of 1 for the first question and a score of 2 for the second one. That translates into a vector $[0, 1, 0]$ and $[0, 0, 1]$, respectively. These we enter into the `data` dictionary:

```
# Initialize a message data structure
messages = Array{Message}(undef, 4)

# Initalize marginal distributions data structure
marginals = Dict()

# Enter the observed outcomes in the placeholders
data = Dict(:X1 => [0, 1, 0],
            :X2 => [0, 0, 1])

# Update coefficients
stepθ!(data, marginals, messages);

# Print messages
print("\nMessage ((1)) = "*string(messages[1].dist))
print("Message ((2)) = "*string(messages[2].dist))
print("Message ((3)) = "*string(messages[3].dist))
print("Message ((4)) = "*string(messages[4].dist))
println("Marginal of θ = "*string(marginals[:θ]))
```

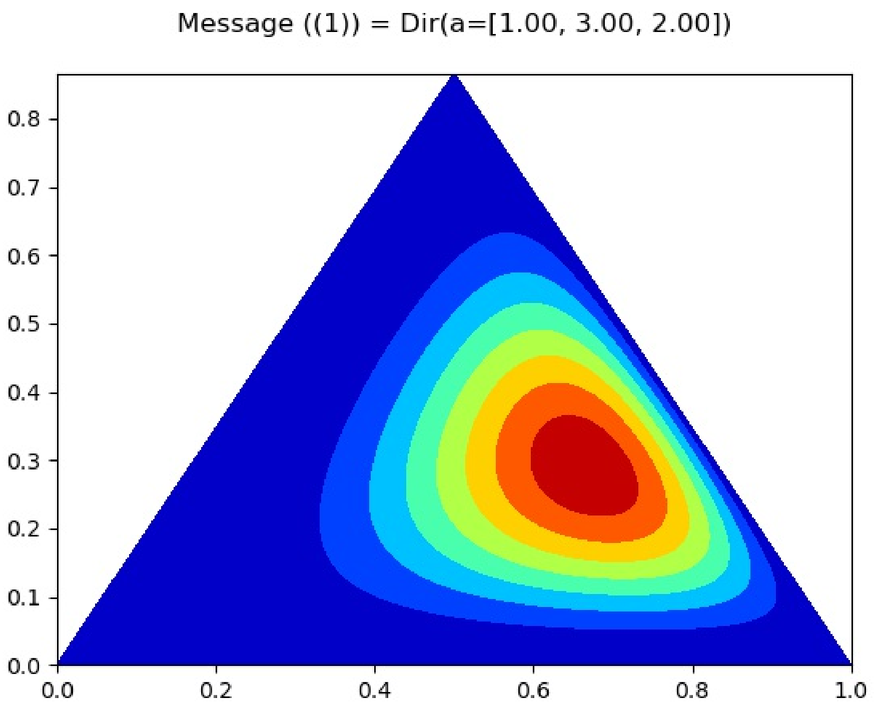
```
Message ((1)) = Dir(a=[1.00, 3.00, 2.00])
Message ((2)) = Dir(a=[1.00, 2.00, 1.00])
Message ((3)) = Dir(a=[1.00, 1.00, 2.00])
Message ((4)) = Dir(a=[1.00, 2.00, 2.00])
Marginal of θ = Dir(a=[1.00, 4.00, 3.00])
```

Visualizing a Dirichlet distribution is a bit tricky. In the special case of 3 parameters, we can plot the probabilities on a simplex. As a reminder, a simplex in 3-dimensions is the triangle between the coordinates $[0, 0, 1]$, $[0, 1, 0]$ and $[1, 0, 0]$:



Each vector θ is a point on that triangle and its elements sum to 1. Since the triangle is 2-dimensional, we can plot the Dirichlet's probability density over it.

```
# Import matplotlib
plt = pyimport("matplotlib.pyplot")

# Include helper function
include("../scripts/dirichlet_simplex.jl")

# Extract parameters of Message ((1))
α1 = messages[1].dist.params[:a]

# Compute pdf contour lines on the simplex
trimesh, pvals = pdf_contours_simplex(α1)

# Plot using matplotlib's tricontour
plt.tricontourf(trimesh, pvals, nlevels=200, cmap="jet");
plt.title("Message ((1)) = "*string(messages[1].dist));
```

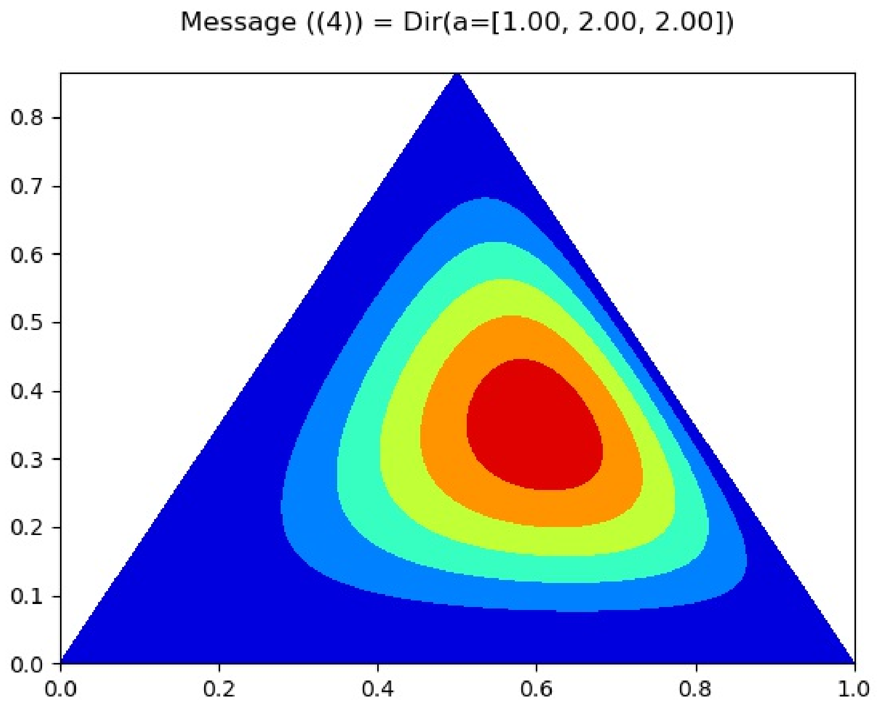


Code notes:

- `pyimport` allows you to import Python modules.
- When you `include()` another julia file, it is as if you wrote it at that point in your script.
- `tricontourf` is a function from Matplolib, where you create a contour plot over a triangulated mesh (here, the simplex).

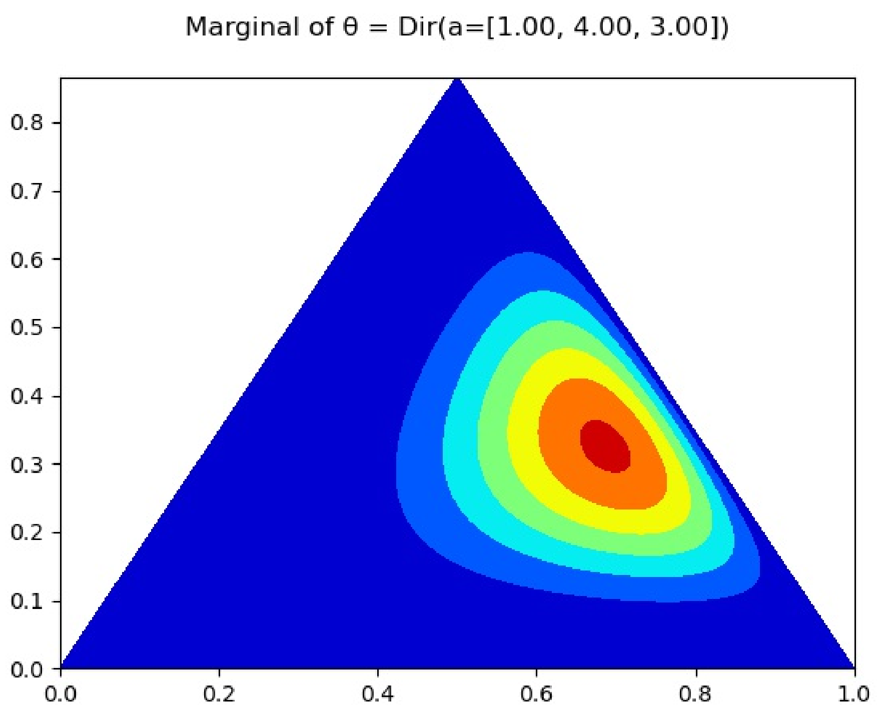
The red spot is the area of high probability, with the contours around indicating increasing uncertainty. The prior, with concentration parameters $[1, 3, 2]$, reflects the belief that applicants are least likely to get the question completely wrong (α_1 = 1, score = 0), most likely to get the question partly right (α_2 = 3, score = 1) and moderately likely to get the question completely correct (α_3 = 2, score = 2).

```
# Extract parameters
α4 = messages[4].dist.params[:a]

# Compute pdf contour lines on the simplex
trimesh, pvals = pdf_contours_simplex(α4)

# Plot using matplotlib's tricontour
plt.tricontourf(trimesh, pvals, nlevels=200, cmap="jet")
plt.title("Message ((4)) = "*string(messages[4].dist));
```



Since we got scores $X_1 = 1$ and $X_2 = 2$, the combined message from both likelihoods has concentration parameters $[1, 2, 2]$.

```
# Extract parameters
α_marg = marginals[:θ].params[:a]

# Compute pdf contour lines on the simplex
trimesh, pvals = pdf_contours_simplex(α_marg)

# Plot using matplotlib's tricontour
plt.tricontourf(trimesh, pvals, nlevels=200, cmap="jet")
plt.title("Marginal of θ = "*string(marginals[:θ]));
```



The posterior is the combination of Messages ((1)) and ((4)) and focuses much more strongly in the area where the two messages overlap.

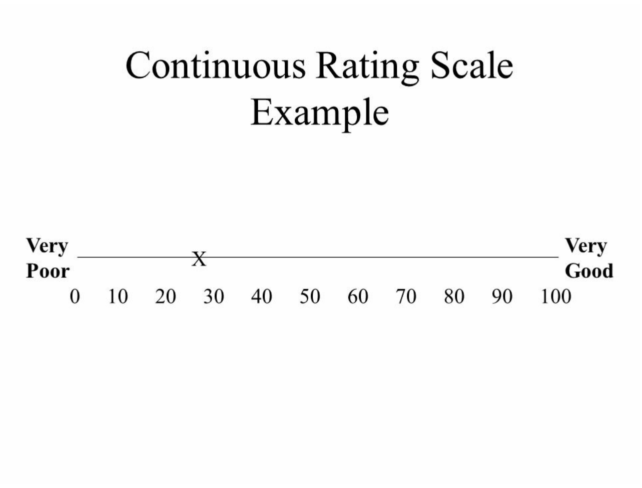
∗ Try for yourself

Play around with the prior parameters and your responses to the questions. See how they change your posterior.

---

3. Rating scale

You might want to evaluate someone by an even finer metric. For example, in oral exams you need to provide a score based on a conversation which is hard to quantify. You could do this by taking away the discrete set of responses and replacing it with a continuous response variable. For example, rating scales are forms of continuous response models. You would mark the applicant's performance on a question as a cross on a line:



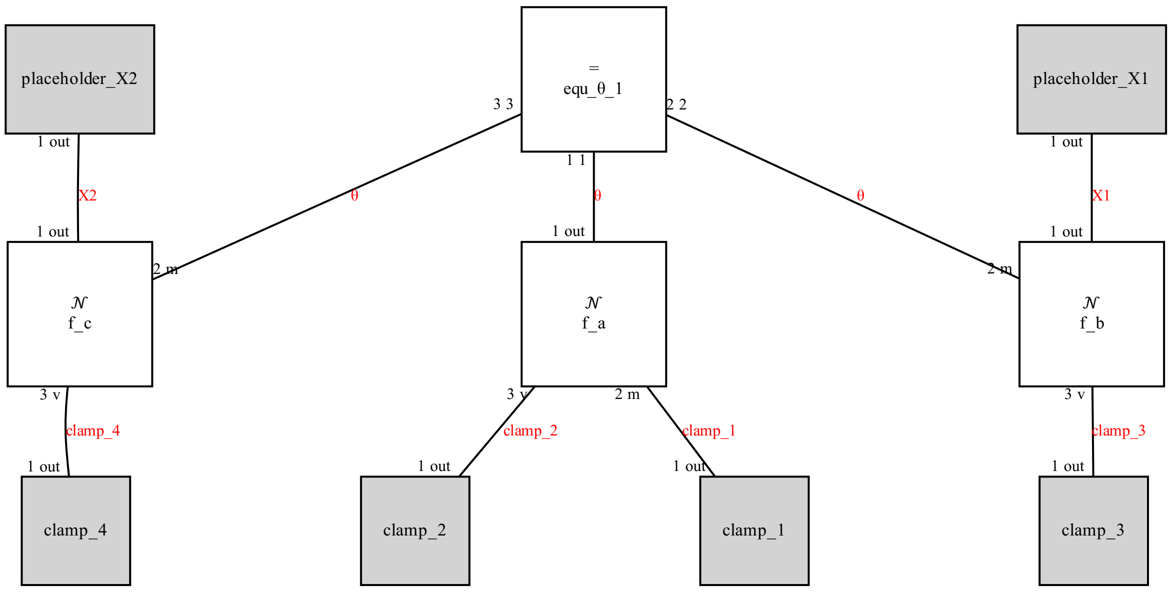
It is still the case that there is some underlying level of skill, that we'll call θ, and that the performance on each question is a noisy measurement of that skill, that we'll call X. We argue that performance noise is symmetric: the probability of performing a little better than their skill level is equal to performing a little worse. We will therefore use Gaussian, or Normal, likelihood functions: $p(X \mid \theta) = \text{Normal}(X \mid \theta, \sigma^2)$. The conjugate prior to the mean in Gaussian likelihoods is another Gaussian distribution: $p(\theta) = \text{Normal}(\theta \mid 60, 20)$. Say that we rate performance on a scale from 0 to 100, then it makes sense to use a mean of 60 and a variance of 20 for the prior.

We'll keep the same generative model as before, with new definitions for each distribution:

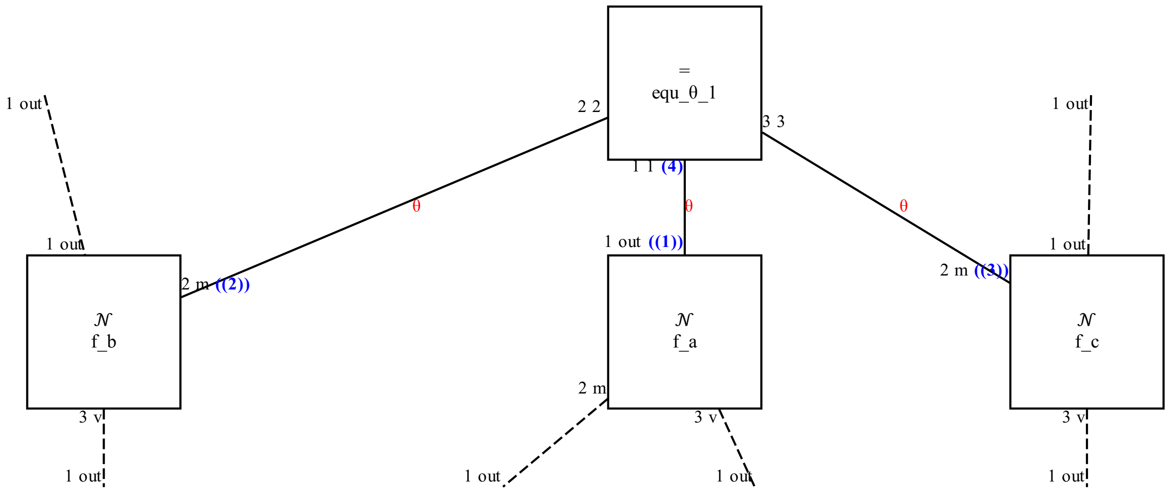
$$\begin{aligned} p(X_1 \mid \theta) &= \text{Normal}(X_1 \mid \theta, 10) & (16) \\ p(X_2 \mid \theta) &= \text{Normal}(X_2 \mid \theta, 15) & (17) \\ p(\theta) &= \text{Normal}(\theta \mid 60, 20) & (18) \end{aligned}$$

The factor graph will again be the same, but with different parameterizations of factor nodes:

```
# Start building a model
factor_graph4 = FactorGraph()

# Add the prior
@RV θ ~ GaussianMeanVariance(60, 20, id=:f_a)

# Add question 1 likelihood
@RV X1 ~ GaussianMeanVariance(θ, 10, id=:f_b)

# Add question 2 likelihood
@RV X2 ~ GaussianMeanVariance(θ, 15, id=:f_c)

# Outcomes are going to be observed
placeholder(X1, :X1)
placeholder(X2, :X2)

# Visualize the graph
ForneyLab.draw(factor_graph4)
```



```
# Indicate which variables you want posteriors for
q = PosteriorFactorization(θ, ids=[:θ])

# Generate a message passing inference algorithm
algorithm = messagePassingAlgorithm(θ, q)

# Compile algorithm code
source_code = algorithmSourceCode(algorithm)

# Bring compiled code into current scope
eval(Meta.parse(source_code))

# Visualize message passing schedule
pfθ = q.posterior_factors[:θ]
ForneyLab.draw(pfθ, schedule=pfθ.schedule);
```



The message passing schedule is still exactly the same.

```
# Initialize a message data structure
messages = Array{Message}(undef, 4)

# Initalize marginal distributions data structure
marginals = Dict()

# Enter the scores in the data dictionary
data = Dict(:X1 => 61.5,
            :X2 => 72)

# Update coefficients
stepθ!(data, marginals, messages);

# Print messages
print("\nMessage ((1)) = "*string(messages[1].dist))
print("Message ((2)) = "*string(messages[2].dist))
print("Message ((3)) = "*string(messages[3].dist))
print("Message ((4)) = "*string(messages[4].dist))
print("Marginal of θ = "*string(marginals[:θ]))
```

```
    Message ((1)) = 𝒩(m=60, v=20)
    Message ((2)) = 𝒩(m=61.50, v=10)
    Message ((3)) = 𝒩(m=72, v=15)
    Message ((4)) = 𝒩(xi=10.95, w=0.17)
    Marginal of θ = 𝒩(xi=13.95, w=0.22)
```

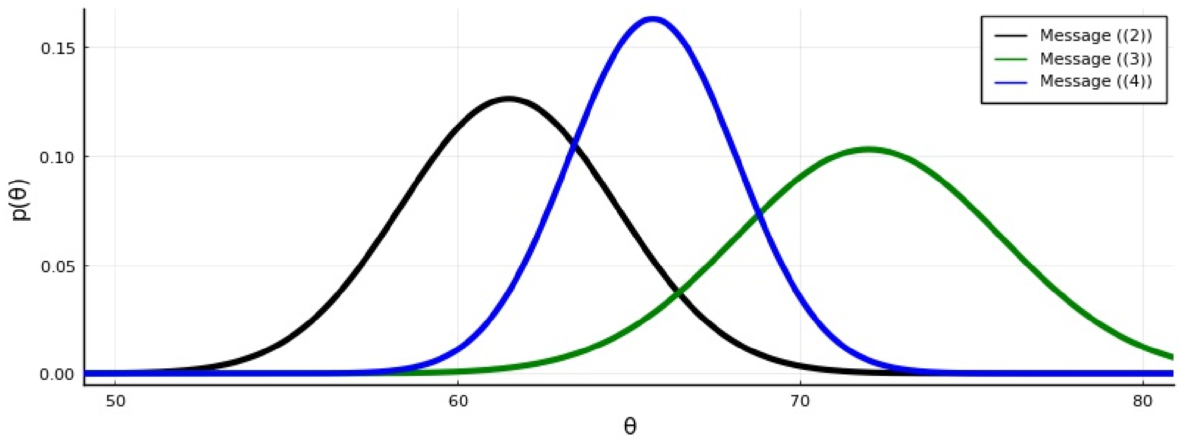
Message ((4)) has a somewhat unusual form in that uses `xi` as a parameter instead of `m`. When you take the product of Messages ((2)) and ((3)), the resulting mean is the sum of the precision-weighted means of Messages ((2)) and ((3)), normalized by the total precision (see Bert's lecture). `xi` represents the sum of precision-weighted means of the two messages. Let's look for the mean:

```
# Extract parameters from message ((4))
m4 = mean(messages[4].dist)
v4 = var(messages[4].dist)
println("Mean of Message ((4)) = "*string(m4))
println("Variance of Message ((4)) = "*string(v4))
```

```
    Mean of Message ((4)) = 65.69999999999999
    Variance of Message ((4)) = 5.999999999999999
```

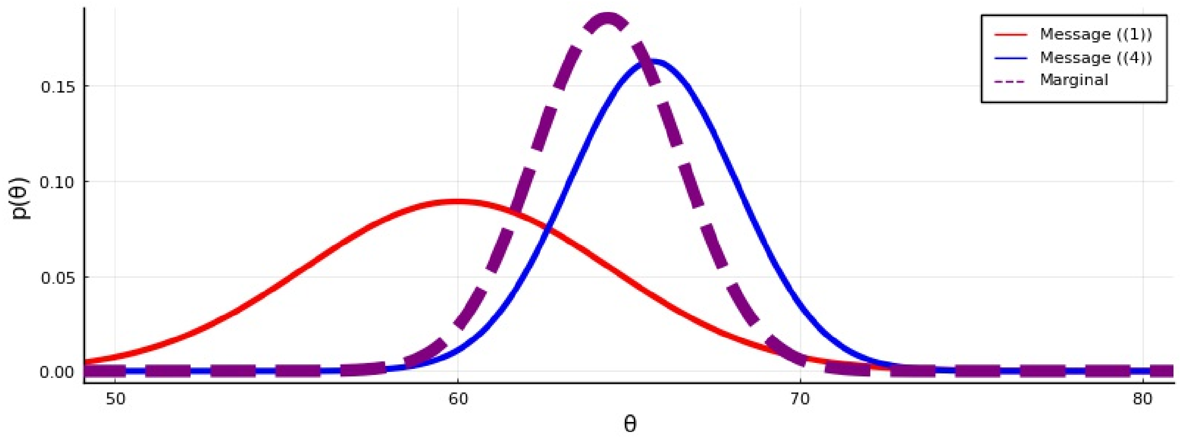
As you can see, the mean of Message ((4)) lies in between the means of Messages ((2)) and ((3)). Note that the variance is much lower than that of Messages ((2)) or ((3)).

```
# Define probability density function for Gaussian distribution
pdf_Normal(θ, m, v) = 1/sqrt(2*π*v) * exp( -(θ - m)^2/(2*v))

# Extract parameters from message ((2))
m2 = messages[2].dist.params[:m]
v2 = messages[2].dist.params[:v]

# Extract parameters from message ((3))
m3 = messages[3].dist.params[:m]
v3 = messages[3].dist.params[:v]

# Extract parameters from message ((4))
m4 = mean(messages[4].dist)
v4 = var(messages[4].dist)

# Define new range for skill level θ
θ_range = range(0.0, step=0.1, stop=100.0)
plot(θ_range, pdf_Normal.(θ_range, m2, v2), color="black", linewidth=3, label="Message ((2))", xla
bel="θ", ylabel="p(θ)")
plot!(θ_range, pdf_Normal.(θ_range, m3, v3), color="green", linewidth=3, label="Message ((3))", si
ze=(800,300))
plot!(θ_range, pdf_Normal.(θ_range, m4, v4), color="blue", linewidth=3, label="Message ((4))", xli
ms=[50., 80.])
```



Message ((4)) is really a weighted average of Messages ((2)) and ((3)).

```
# Extract parameters from message ((1))
m1 = messages[1].dist.params[:m]
v1 = messages[1].dist.params[:v]

# Extract parameters from message ((4))
m4 = mean(messages[4].dist)
v4 = var(messages[4].dist)

# Extract parameters from marginal
m_marg = mean(marginals[:θ])
v_marg = var(marginals[:θ])

# Define new range for skill level θ
plot(θ_range, pdf_Normal.(θ_range, m1, v1), color="red", linewidth=3, label="Message ((1))", xlabe
l="θ", ylabel="p(θ)")
plot!(θ_range, pdf_Normal.(θ_range, m4, v4), color="blue", linewidth=3, label="Message ((4))", siz
e=(800,300))
plot!(θ_range, pdf_Normal.(θ_range, m_marg, v_marg), color="purple", linewidth=6, linestyle=:dash,
 label="Marginal", xlims=[50., 80.])
```



The posterior is also a weighted average of two incoming messages. Notice that it is much closer to Message ((4)) than Message ((1)). That is because the variance of Message ((1)) (the prior) is much higher than that of Message ((4)) (the combination of likelihoods). The prior has a smaller weight in the weighted average.

---

* Try for yourself

Play around with different values for the prior's variance and the variance of the likelihoods. What happens when you make the variance of $p(X_1 \mid \theta)$ different from that of $p(X_2 \mid \theta)$?

Probabilistic Programming 3: Regression & Classification

Goal

- Understand how to estimate regression parameters using Bayesian inference.
- Understand how to estimate classification parameters using Bayesian inference.

Materials

- Mandatory
  - This notebook.
  - Lecture notes on regression.
  - Lecture notes on discriminative classification.
- Optional
  - Bayesian linear regression (Section 3.3 Bishop)
  - Bayesian logistic regression (Section 4.5 Bishop)
  - Local Variational Methods (Section 10.5 Bishop)
  - Cheatsheets: how does Julia differ from Matlab / Python.

```
using Pkg
Pkg.activate("workspace")
Pkg.instantiate();
```

Problem: Economic growth

In 2008, the credit crisis sparked a recession in the US, which spread to other countries in the ensuing years. It took most countries a couple of years to recover. Now, the year is 2011. The Turkish government is asking you to estimate whether Turkey is out of the recession. You decide to look at the data of the national stock exchange to see if there's a positive trend.

```
using DataFrames
using CSV
using ProgressMeter
using Plots
pyplot();
```

Data

We are going to start with loading in a data set. We have daily measurements from Istanbul, from the 5th of January 2009 until 22nd of February 2011. The dataset comes from an online resource for machine learning data sets: the UCI ML Repository.

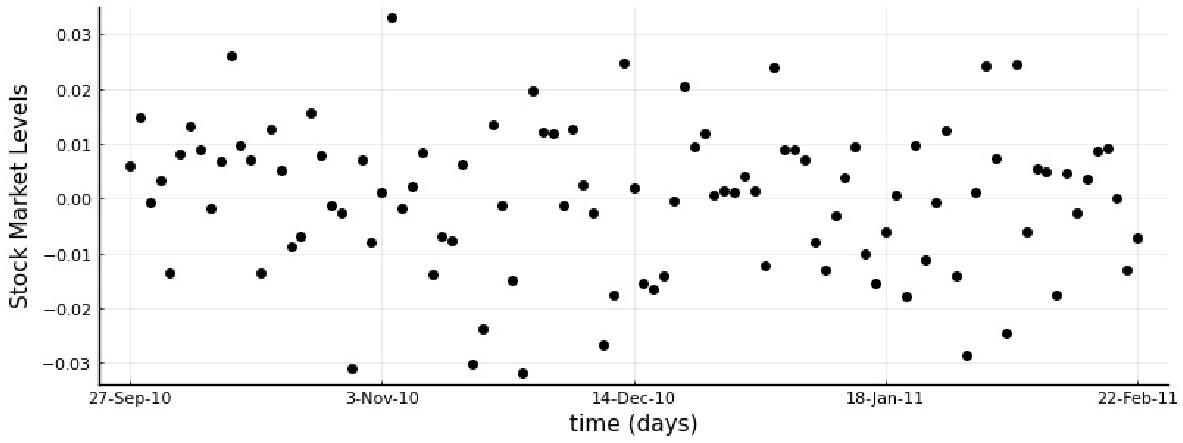
```
# Read CSV file
df = DataFrame(CSV.File("../datasets/istanbul_stockexchange.csv"))
```

536 rows × 2 columns

	date	ISE
	String	Float64
1	5-Jan-09	0.0357537
2	6-Jan-09	0.0254259
3	7-Jan-09	-0.0288617
4	8-Jan-09	-0.0622081
5	9-Jan-09	0.00985991
6	12-Jan-09	-0.029191
7	13-Jan-09	0.0154453
8	14-Jan-09	-0.0411676
9	15-Jan-09	0.000661905
10	16-Jan-09	0.0220373
11	19-Jan-09	-0.0226925
12	20-Jan-09	-0.0137087
13	21-Jan-09	0.000864697
14	22-Jan-09	-0.00381506
15	23-Jan-09	0.00566126
16	26-Jan-09	0.0468313
17	27-Jan-09	-0.00663498
18	28-Jan-09	0.034567
19	29-Jan-09	-0.0205282
20	30-Jan-09	-0.0087767
21	2-Feb-09	-0.0259191
22	3-Feb-09	0.0152795
23	4-Feb-09	0.0185778
24	5-Feb-09	-0.0141329
25	6-Feb-09	0.036607
26	9-Feb-09	0.0113532
27	10-Feb-09	-0.040542
28	11-Feb-09	-0.0221056
29	12-Feb-09	-0.0148884
30	13-Feb-09	0.00702675
&vellip;	&vellip;	&vellip;

We can plot the evolution of the stock market values over time.

```
# Count number of samples
time_period = 436:536
num_samples = length(time_period)

# Extract columns
dates_num = 1:num_samples
dates_str = df[time_period,1]
stock_val = df[time_period,2]

# Set xticks
xtick_points = Int64.(round.(range(1, stop=num_samples, length=5)))

# Scatter exchange levels
scatter(dates_num,
        stock_val,
        color="black",
        label="",
        ylabel="Stock Market Levels",
        xlabel="time (days)",
        xticks=(xtick_points, [dates_str[i] for i in xtick_points]),
        size=(800,300))
```



Model specification

We have dates X, referred to as "covariates", and stock exchange levels Y, referred to as "responses". A regression model has parameters θ, used to predict Y from X. We are looking for a posterior distribution for the parameters θ:

$$\underbrace{p(\theta \mid Y, X)}_{\text{posterior}} \propto \underbrace{p(Y \mid X, \theta)}_{\text{likelihood}} \cdot \underbrace{p(\theta)}_{\text{prior}}$$

We assume each observation Y_i is generated via:

$$Y_i = f_\theta(X_i) + e$$

where e is white noise, $e \sim \mathcal{N}(0, \sigma_Y^2)$, and the regression function f_θ is linear: $f_\theta(X) = X\theta_1 + \theta_2$. The parameters consist of a slope coefficient θ_1 and an intercept θ_2, which are summarized into a parameter vector, $\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$. In practice, we augment the data matrix X with a 1, i.e., $\begin{bmatrix} X \\ 1 \end{bmatrix}$, so that we may define $f_\theta(X) = \theta^\top X$.

If we integrate out the noise e, then we obtain a Gaussian likelihood function centered on $f_\theta(X)$ with variance σ_Y^2:

$$Y_i \sim \mathcal{N}(f_\theta(X_i), \sigma_Y^2) \ .$$

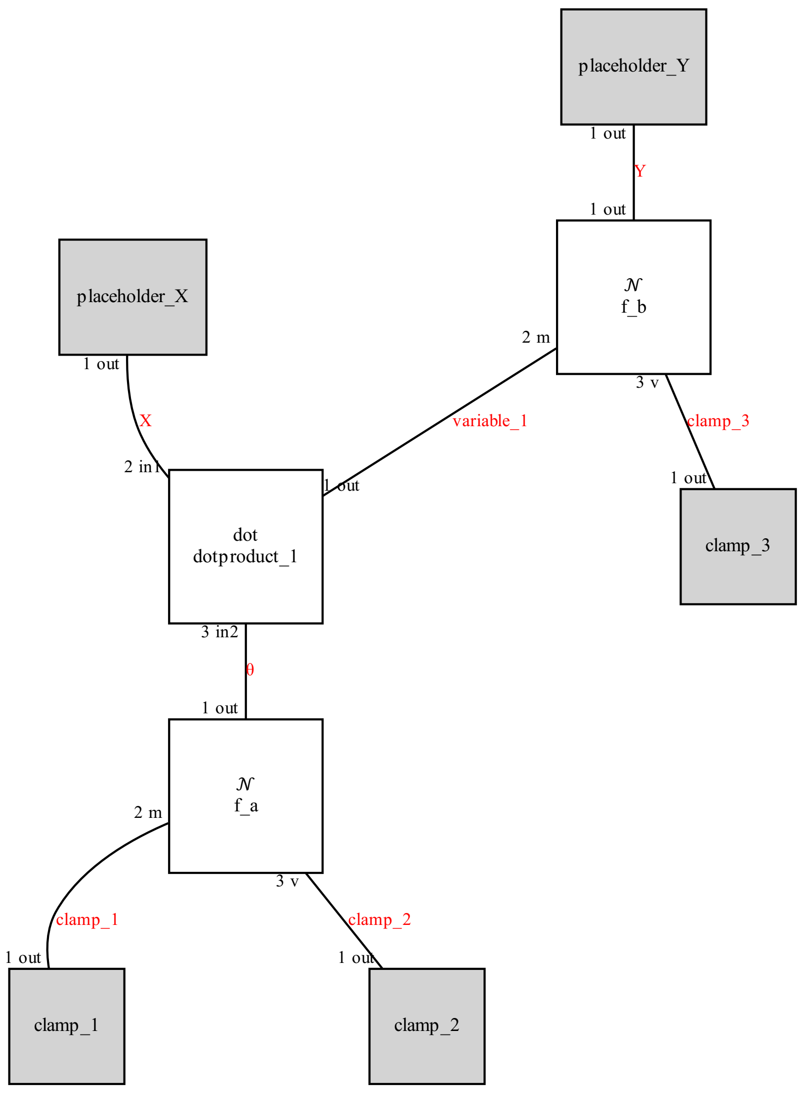
Note that this corresponds to $p(Y \mid X, \theta)$. We know that the weights are real numbers and that they can be negative. That motivates us to use a Gaussian prior:

$$\theta \sim \mathcal{N}(\mu_\theta, \Sigma_\theta) \,.$$

Note that this corresponds to $p(\theta)$. For now, this is all we need. We're going to specify these two equations with ForneyLab. Our prior will become the first factor node: $f_a(\theta) = \mathcal{N}(\theta \mid \mu_\theta, \Sigma_\theta)$ and our likelihood becomes the second factor node $f_b(\theta) = \mathcal{N}(f_\theta(X_i), \sigma_Y^2)$. Note that θ is the only unknown variable, since X and Y are observed and μ_θ, Σ_θ and σ_Y^2 are clamped.

```
using ForneyLab
```

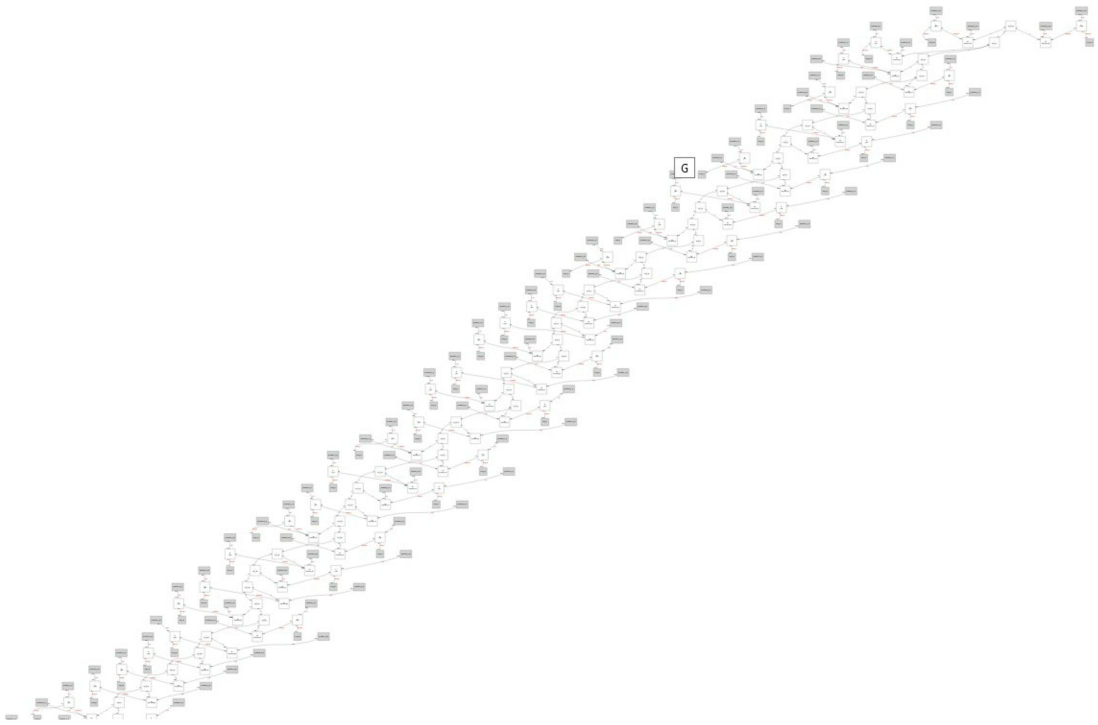
```
# Start factor graph
graph = FactorGraph();

# Prior weight parameters
μ_θ = [0., 0.]
Σ_θ = [1. 0.; 0. 1.]

# Noise variance
σ2_Y = 1.

# Add weight prior to graph
@RV θ ~ GaussianMeanVariance(μ_θ, Σ_θ, id=:f_a)

# Define covariates
@RV X

# Define likelihood
@RV Y ~ GaussianMeanVariance(dot(θ,X), σ2_Y, id=:f_b)

# Designate observed variables
placeholder(X, :X, dims=(2,))
placeholder(Y, :Y)

# Visualise the graph
ForneyLab.draw(graph)
```



But this is the graph for a single observation. In reality, we have multiple observations. If we want to consume all this information at once, the likelihood part of the above graph will need to be repeated.

```
# Start factor graph
graph2 = FactorGraph();

# Prior weight parameters
μ_θ = [0., 0.]
Σ_θ = [1. 0.; 0. 1.]

# Noise variance
σ2_Y = 1.

# Add weight prior to graph
@RV θ ~ GaussianMeanVariance(μ_θ, Σ_θ, id=:f_a)

# Pre-define vectors for storing latent and observed variables
X = Vector{Variable}(undef, num_samples)
Y = Vector{Variable}(undef, num_samples)

for i = 1:num_samples

    # Define i-th covariate
    @RV X[i]

    # Define likelihood of i-th response
    @RV Y[i] ~ GaussianMeanVariance(dot(θ,X[i]), σ2_Y, id=Symbol("f_b"*string(i)))

    # Designate observed variables
    placeholder(X[i], :X, index=i, dims=(2,))
    placeholder(Y[i], :Y, index=i);

end
```

I've generated the factor graph and embedded a screenshot below. It continues on for a while.



If you'd like to see the full thing, uncomment the line below.

```
# ForneyLab.draw(graph2)
```

It's hard to tell, but each of these likelihood nodes f_{bi} is connected to the prior node f_a via an equality node.

Now that we have our model, it is time to infer parameters.

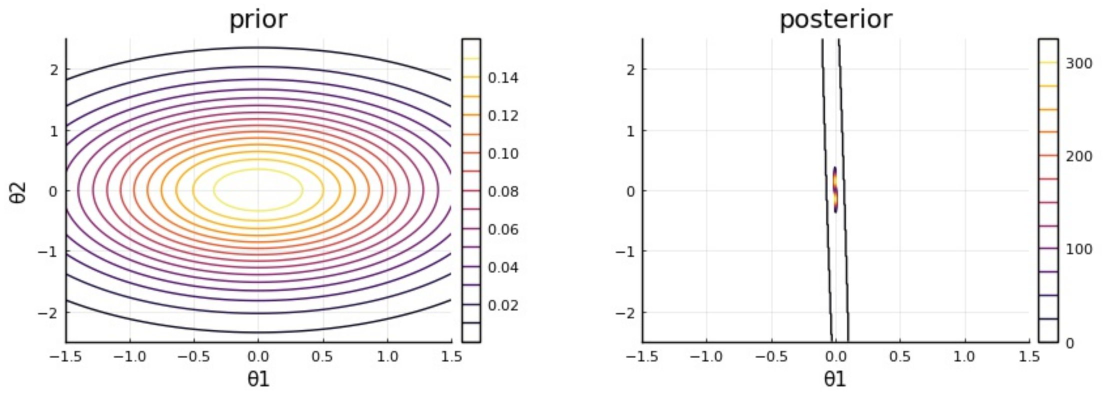
```
# Define and compile the algorithm
algorithm = messagePassingAlgorithm(θ)
source_code = algorithmSourceCode(algorithm)

# Evaluate the generated code to get the step! function
eval(Meta.parse(source_code));
# println(source_code)
```

Now, we iterate over time, feeding our data as it comes in and updating our posterior distribution for the parameters.

```
# Initialize posterior
posterior = Dict()

# Load data
data = Dict(:X => [[dates_num[i], 1] for i = 1:num_samples],
            :Y => stock_val)

# Update posterior for θ
step!(data, posterior);
```

Let's visualize the resulting posterior.

```
import ForneyLab: logPdf

# Define ranges for plot
x1 = range(-1.5, length=500, stop=1.5)
x2 = range(-2.5, length=500, stop=2.5)

# Draw contour plots of distributions
prior = ProbabilityDistribution(Multivariate, GaussianMeanVariance, m=μ_θ, v=Σ_θ)
p1a = contour(x1, x2, (x1,x2) -> exp(logPdf(prior, [x1,x2])), xlabel="θ1", ylabel="θ2", title="pri
or", label="")
p1b = contour(x1, x2, (x1,x2) -> exp(logPdf(posterior[:θ], [x1,x2])), xlabel="θ1", title="posterio
r", label="")
plot(p1a, p1b, size=(900,300))
```



It has become quite sharply peaked in a small area of parameter space.

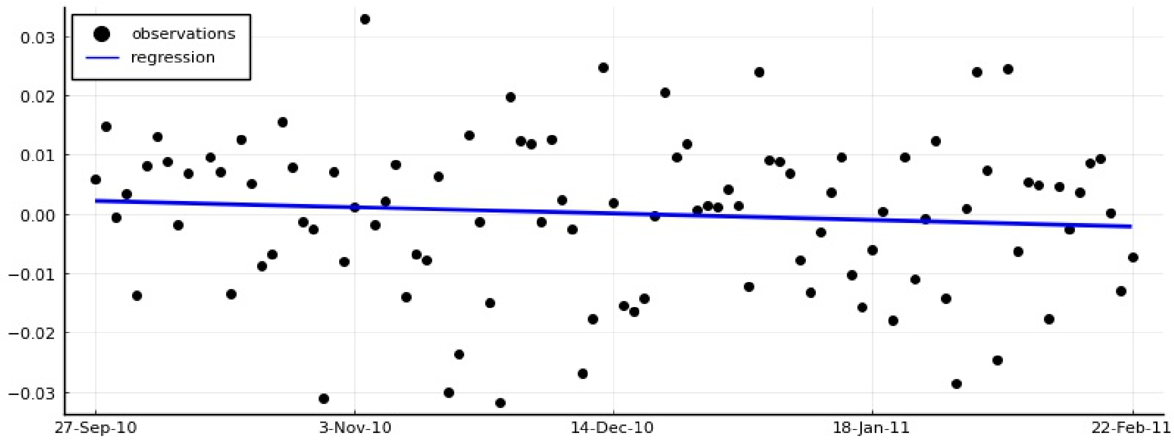
We can use the MAP point estimate to compute and visualize the regression function f_θ. The full predictive distribution is left for the PP Assignment.

```
# Extract estimated weights
θ_MAP = mode(posterior[:θ])

# Report results
println("Slope coefficient = "*string(θ_MAP[1]))
println("Intercept coefficient = "*string(θ_MAP[2]))

# Make predictions
regression_estimated = dates_num * θ_MAP[1] .+ θ_MAP[2];
```

```
Slope coefficient = -4.320468927288263e-5
Intercept coefficient = 0.0022453122200430577
```

Let's visualize it.

```
# Visualize observations
scatter(dates_num, stock_val, color="black", xticks=(xtick_points, [dates_str[i] for i in xtick_po
ints]), label="observations", legend=:topleft)

# Overlay regression function
plot!(dates_num, regression_estimated, color="blue", label="regression", linewidth=2, size=(800,30
0))
```



The slope coefficient θ_1 is negative and the plot shows a decreasing line. The ISE experienced a negative linear trend from October 2010 up to March 2011. Assuming the stock market is an indicator of economic growth, then we may conclude that in March 2011 the Turkish economy is still in recession.

*** Try for yourself**

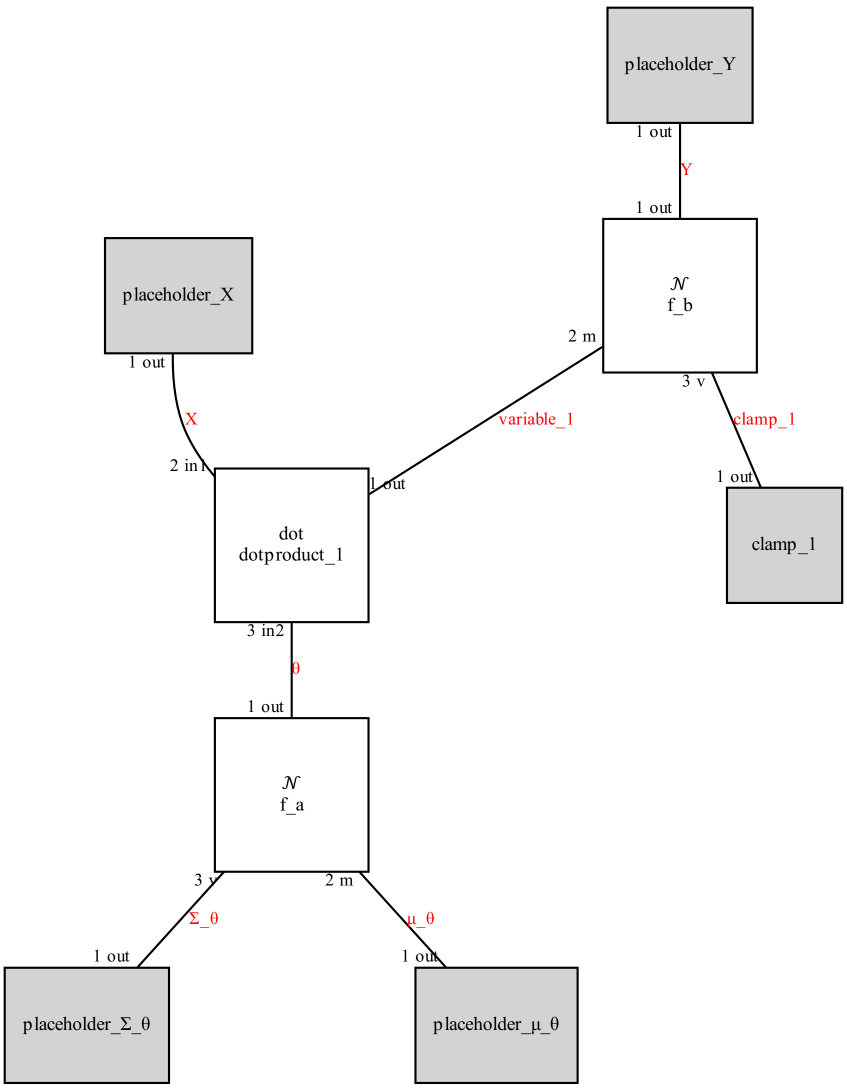
Change the `time period` variable. Re-run the regression and see how the results change.

Recursive estimation

Our graph is quite large, which means our inference algorithm is slow. But as I already said earlier, the graph is essentially a repetition of the same structure. In this case, we don't need to generate such a large graph; we can recursively estimate the classification parameters. To do this, we essentially estimate parameters for a single observation, and make the resulting posterior our prior for the next observation.

Let's first re-define the subgraph for a single observation.

```
# Start factor graph
graph = FactorGraph();

# Noise variance
σ2_Y = 1.

# Add weight prior to graph
@RV θ ~ GaussianMeanVariance(placeholder(:μ_θ, dims=(2,)),
                             placeholder(:Σ_θ, dims=(2,2)), id=:f_a)

# Define covariates
@RV X

# Define likelihood
@RV Y ~ GaussianMeanVariance(dot(θ,X), σ2_Y, id=:f_b)

# Designate observed variables
placeholder(X, :X, dims=(2,))
placeholder(Y, :Y)

# Visualise the graph
ForneyLab.draw(graph)
```



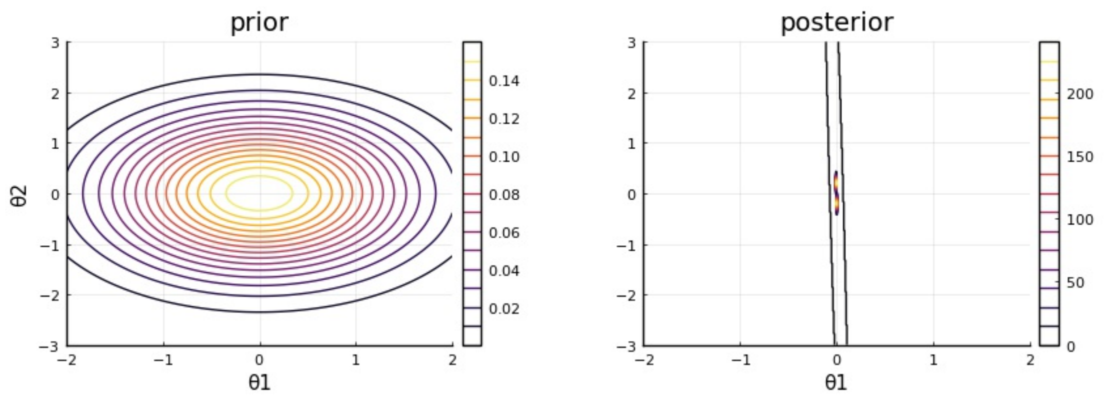
```
# Define and compile the algorithm
algorithm = messagePassingAlgorithm(θ)
source_code = algorithmSourceCode(algorithm)

# Evaluate the generated code to get the step! function
eval(Meta.parse(source_code));
```

The syntax for compiling the inference algorithm is the same as before, but now we execute it differently. We feed in each sample and perform a step update for the posterior.

```
# Initialize posteriors dictionary
posterior = Dict()
posterior[:θ] = ProbabilityDistribution(Multivariate, GaussianMeanVariance, m=[0.,0.], v=[1. 0.;0.
 1.])

@showprogress for i = 1:num_samples

    # Load i-th data point
    data = Dict(:X => [dates_num[i], 1],
                :Y => stock_val[i],
                :μ_θ => mean(posterior[:θ]),
                :Σ_θ => cov(posterior[:θ]))

    # Update posterior for θ
    step!(data, posterior)
end
```

This is much faster. If we now visualize the resulting posterior, we see that it is not exactly the same. Recursive estimation is not mathematically equivalent to non-recursive estimation (it's the difference between filtering and smoothing, for those familiar). In this case, it produces a less sharply peaked posterior (note the y-axis between this plot and the previous posterior plot).

```
import ForneyLab: logPdf

# Define ranges for plot
x1 = range(-2, length=500, stop=2)
x2 = range(-3, length=500, stop=3)

# Draw contour plots of distributions
prior = ProbabilityDistribution(Multivariate, GaussianMeanVariance, m=μ_θ, v=Σ_θ)
p1a = contour(x1, x2, (x1,x2) -> exp(logPdf(prior, [x1,x2])), xlabel="θ1", ylabel="θ2", title="pri
or", label="")
p1b = contour(x1, x2, (x1,x2) -> exp(logPdf(posterior[:θ], [x1,x2])), xlabel="θ1", title="posterio
r", label="")
plot(p1a, p1b, size=(900,300))
```



Problem: Credit Assignment

We will now look at a classification problem. Suppose you are a bank and that you have to decide whether you will grant credit, e.g. a mortgage or a small business loan, to a customer. You have a historic data set where your experts have assigned credit to hundreds of people. You have asked them to report on what aspects of the problem are important. You hope to automate this decision process by training a classifier on the data set.

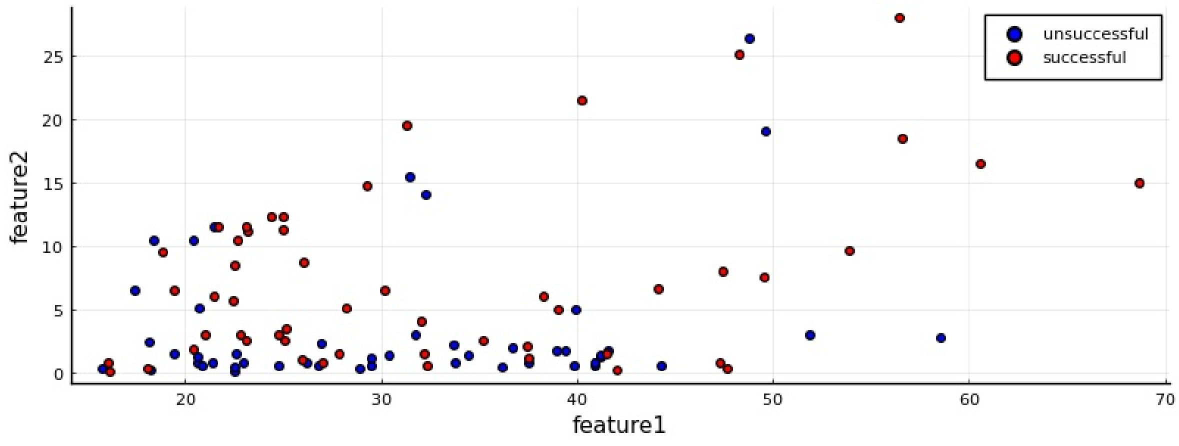
Data

The data set we are going to use actually comes from the UCI ML Repository. It consists of past credit assignments, labeled as successful (=1) or unsuccessful (=0). Many of the features have been anonymized for privacy concerns.

```
# Read CSV file
df = DataFrame(CSV.File("../datasets/credit_train.csv"))

# Split dataframe into features and labels
features_train = Matrix(df[:,1:7])
labels_train = Vector(df[:,8])

# Store number of features
num_features = size(features_train,2)

# Number of training samples
num_train = size(features_train,1);
```

Let's visualize the data and see if we can make sense of it.

```
scatter(features_train[labels_train .== 0, 1], features_train[labels_train .== 0, 2], color="blue"
, label="unsuccessful", xlabel="feature1", ylabel="feature2")
scatter!(features_train[labels_train .== 1, 1], features_train[labels_train .== 1, 2], color="red"
, label="successful", size=(800,300))
```



Mmhh, it doesn't look like the samples can easily be separated. This will be challenging.

```
* Try for yourself
```

The plot above shows features 1 and 2. Have a look at the other combinations of features.

Model specification

We have features X, labels Y and parameters θ. Same as with regression, we are looking for a posterior distribution of the classification parameters:

$$\underbrace{p(\theta \mid Y, X)}_{\text{posterior}} \propto \underbrace{p(Y \mid X, \theta)}_{\text{likelihood}} \cdot \underbrace{p(\theta)}_{\text{prior}}$$

The likelihood in this case will be of a Logit form:

$$p(Y \mid X, \theta) = \prod_{i=1}^{N} \text{Logit}(Y_i \mid f_\theta(X_i), \xi_i) \, .$$

A "Logit" is short for a Bernoulli distribution with a sigmoid transfer function: $\sigma(x) = 1/(1 + \exp(-x))$. The sigmoid maps the input to the interval $(0, 1)$ so that the result acts as a rate parameter to the Bernoulli. Check Bert's lecture on discriminative classification for more information.

We are not using a Laplace approximation to the posterior, but rather a "local variational method" (see Section 10.5 of Bishop). We won't go into how that works here. All you need to know in this implementation is that there is a second parameter to the Logit, ξ (the "local variational parameter"), which has to be estimated just as the classification parameters θ (but doesn't need a prior).

We will use a Gaussian prior distribution for the classification parameters θ:

$$p(\theta) = \mathcal{N}(\theta \mid \mu_\theta, \Sigma_\theta) \, .$$

Note that the true posterior is still approximated with a Gaussian distribution.

```
import LinearAlgebra: I
```

```julia
# Start factor graph
graph = FactorGraph();

# Parameters for priors
μ_θ = zeros(num_features+1,)
Σ_θ = Matrix{Float64}(I, num_features+1, num_features+1)

# Define a prior over the weights
@RV θ ~ GaussianMeanVariance(μ_θ, Σ_θ)

X = Vector{Variable}(undef, num_train)
ξ = Vector{Variable}(undef, num_train)
Y = Vector{Variable}(undef, num_train)

for i = 1:num_train

    # Features
    @RV X[i]

    # Local variational parameter
    @RV ξ[i]

    # Logit likelihood
    @RV Y[i] ~ Logit(dot(θ, X[i]), ξ[i])

    # Observed
    placeholder(X[i], :X, index=i, dims=(num_features+1,))
    placeholder(Y[i], :Y, index=i)

end
```

We will now compile an inference algorithm for this model. Since we now have two unknown variables, θ and ξ, we need to define a `PosteriorFactorization()`. With this function, we are basically telling ForneyLab that these variables need to be estimated separately (i.e., generate two `step!` functions).

```julia
# We specify a recognition distribution
q = PosteriorFactorization(θ, ξ, ids=[:θ, :ξ])

# Define and compile the algorithm
algorithm = messagePassingAlgorithm()
source_code = algorithmSourceCode(algorithm)

# Bring the generated source code into scope
eval(Meta.parse(source_code));
```

Now that we have compiled the algorithm, we are going to iteratively update the classification parameters and the local variational parameter.

```
# Initialize posteriors
posteriors = Dict()
for i = 1:num_train
    posteriors[:ξ_*i] = ProbabilityDistribution(Function, mode=1.0)
end

# Load data
data = Dict(:X => [[features_train[i,:]; 1] for i in 1:num_train],
            :Y => labels_train)

# Iterate updates
@showprogress for i = 1:10

    # Update classification parameters
    stepθ!(data, posteriors)

    # Update local variational parameters
    stepξ!(data, posteriors)
end
```

Predict test data

The bank has some test data for you as well.

```
# Read CSV file
df = DataFrame(CSV.File("../datasets/credit_test.csv"))

# Split dataframe into features and labels
features_test = Matrix(df[:,1:7])
labels_test = Vector(df[:,8])

# Number of test samples
num_test = size(features_test,1);
```

You can classify test samples by taking the MAP for the classification parameters, computing the linear function f_θ and rounding the result to obtain the most probable label.

```
import ForneyLab: unsafeMode
```

```
# Extract MAP estimate of classification parameters
θ_MAP = unsafeMode(posteriors[:θ])

# Compute dot product between parameters and test data
fθ_pred = [features_test ones(num_test,)] * θ_MAP

# Predict labels
labels_pred = round.(1 ./(1 .+ exp.( -fθ_pred)));

# Compute classification accuracy of test data
accuracy_test = mean(labels_test .== labels_pred)

# Report result
println("Test Accuracy = "*string(accuracy_test*100)*"%")
```

```
    Test Accuracy = 63.0%
```

Mmmhh... If you were a bank, you might decide that you don't want to automatically assign credit to your customers.

Probabilistic Programming 4: Latent Variable & Dynamic Models

Goal

- Understand how to estimate latent variables in models.
- Understand how to estimate states in dynamical models.

Materials

- Mandatory
  - This notebook
  - Lecture notes on latent variable models
  - Lecture notes on dynamical models
- Optional
  - Review of latent variable models
  - Bayesian Filtering & Smoothing
  - Differences between Julia and Matlab / Python.

Note that none of the material below is new. The point of the Probabilistic Programming sessions is to solve practical problems so that the concepts from Bert's lectures become less abstract.

```julia
using Pkg
Pkg.activate("./workspace")
Pkg.instantiate();
```

```julia
using JLD
using Statistics
using StatsBase
using LinearAlgebra
using ProgressMeter
using ColorSchemes
using LaTeXStrings
using ForneyLab
using Plots
pyplot();

import LinearAlgebra: I
import ForneyLab: unsafeMean
include("../scripts/clusters.jl");
include("../scripts/filters.jl");
```

Problem: Stone Tools

Archeologists have asked for your help in analyzing data on stone tools. It is believed that primitive humans created tools by striking stones with others. During this process, the stone loses flakes, which have been preserved. The archeologists have recovered these flakes from various locations and time periods and want to know whether this stone tool shaping process has improved over the centuries.

Data

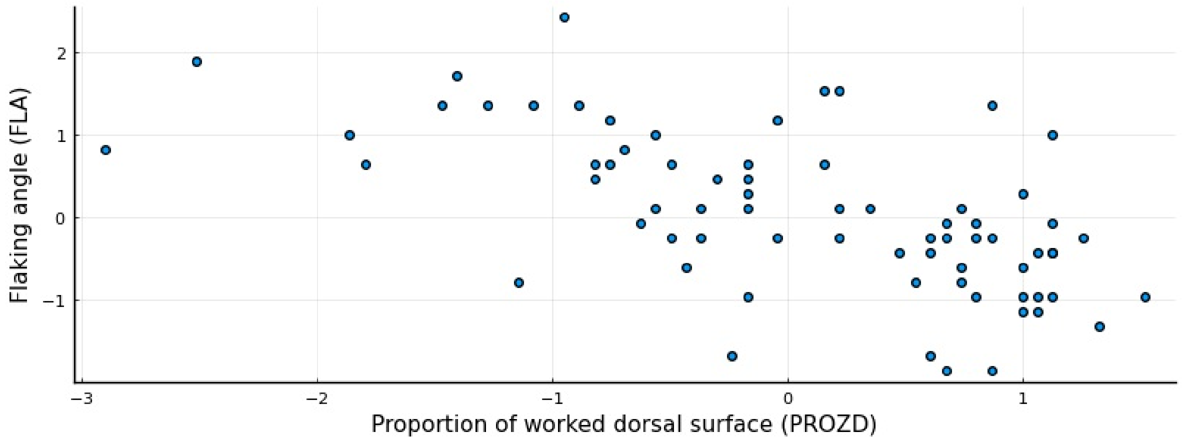
The data is available from the UCI Machine Learning Repository. Each instance represents summary information of the stone flakes for a particular site. We will be using the attributes flaking angle (FLA) and the proportion of the dorsal surface worked (PROZD) for now.

```
dataset = load("../datasets/stoneflakes.jld");
```

I've done some pre-processing on the data set, namely z-scoring and removing two outliers. This reduces the scale of the attributes which helps numerical stability during optimization.

Now let's visualize the data with a scatterplot.

```
scatter(dataset["data"][:,1],
        dataset["data"][:,2],
        label="",
        xlabel="Proportion of worked dorsal surface (PROZD)",
        ylabel="Flaking angle (FLA)",
        size=(800,300))
```



Model specification

We will be clustering this data with a Gaussian mixture model, to see if we can identify clear types of stone tools. The generative model for a Gaussian mixture consists of:

$$p(X, z, \phi, \mu, \Lambda) = \underbrace{p(X \mid z, \mu, \Lambda)}_{\text{likelihood}} \times \underbrace{p(z \mid \phi)}_{\text{prior latent variables}} \times \underbrace{p(\mu \mid \Lambda)\; p(\Lambda)\; p(\phi)}_{\text{prior parameters}}$$

with the likelihood of observation X_i being a Gaussian raised to the power of the latent assignment variables z

$$p(X_i \mid z, \mu, \Lambda) = \prod_{k=1}^{K} \mathcal{N}(X_i \mid \mu_k, \Lambda_k^{-1})^{z_i = k}$$

the prior for each latent variable z_i being a Categorical distribution

$$p(z_i \mid \phi) = \text{Categorical}(z_i \mid \phi)$$

and priors for the parameters being

$$\begin{aligned} p(\mu_k \mid \Lambda_k) &= \mathcal{N}(\mu_k \mid m_0, l_0^{-1}\Lambda_k^{-1}) \qquad && \text{for all } k \\ p(\Lambda_k) &= \text{Wishart}(\Lambda_k \mid V_0, n_0) && \text{for all } k \\ p(\phi) &= \text{Dirichlet}(\phi \mid a_0)\,, \end{aligned}$$

We will be implementing this model directly in ForneyLab. If you're unfamiliar with these distributions or with the Gaussian mixture model, have another look at Bert's lectures.

First, we will do a bit of bookkeeping.

```
# Data dimensionality
num_features = size(dataset["data"],2)

# Sample size
num_samples = size(dataset["data"],1)

# Number of mixture components
num_components = 3;

# Identity matrix (convenience variable)
Id = Matrix{Float64}(I, num_features, num_features);
```

Mixture models can be sensitive to initialization, so we are going to specify the prior parameters explicitly.

```
# Prior means
m0 = [ 1.0 0.0 -1.0;
      -1.0 0.0  1.0];

# Prior scale matrices
V0 = cat(Id, Id, Id, dims=3)

# Prior degrees of freedom
n0 = num_features

# Prior concentration parameters
a0 = ones(num_components);
```

Now to start the factor graph.

```julia
# Start a graph
graph1 = FactorGraph()

# Initialize vector variables
z = Vector{Variable}(undef, num_samples)
X = Vector{Variable}(undef, num_samples)
Λ = Vector{Variable}(undef, num_components)
μ = Vector{Variable}(undef, num_components)

# Mixture weights are drawn from a Dirichlet distribution
@RV ϕ ~ Dirichlet(a0)

θ = []
for k = 1:num_components

    # Parameters of k-th component
    @RV Λ[k] ~ Wishart(V0[:,:,k], n0)
    @RV μ[k] ~ GaussianMeanPrecision(m0[:,k], Λ[k])

    push!(θ, μ[k], Λ[k])
end

for i = 1:num_samples

    # Assignment variable
    @RV z[i] ~ Categorical(ϕ)

    # Gaussian mixture component
    @RV X[i] ~ GaussianMixture(z[i], θ...)

    # Add data
    placeholder(X[i], :X, dims=(num_features,), index=i)
end
```

This is another iid setting, which means the graph will be too large to visualize.

The next step is to compile an inference algorithm.

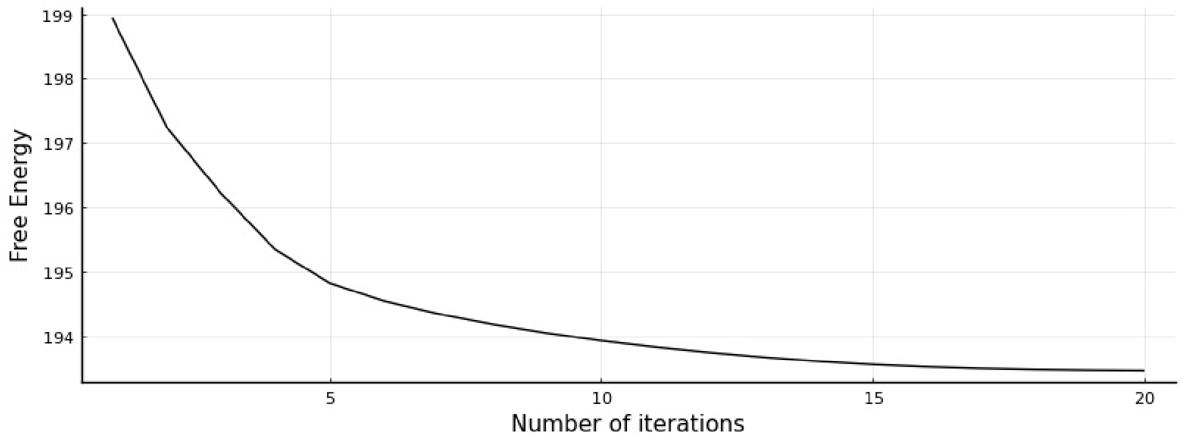
```julia
# Specify recognition factorization (mean-field)
q = PosteriorFactorization(ϕ, μ[1], Λ[1], μ[2], Λ[2], μ[3], Λ[3], z,
                           ids=[:ϕ, :μ_1, :Λ_1, :μ_2, :Λ_2, :μ_3, :Λ_3, :z])

# Generate the algorithm
algorithm = messagePassingAlgorithm(free_energy=true)
source_code = algorithmSourceCode(algorithm, free_energy=true);
eval(Meta.parse(source_code));
```

After that, we feed in data, initialize recognition factors and run the inference procedure.

```
# Convert data to a format suited to ForneyLab
observations = [dataset["data"][i,:] for i in 1:num_samples]

# Add to data dictionary
data = Dict(:X => observations)

# Prepare recognition distributions
marginals = Dict()
marginals[:ϕ] = ProbabilityDistribution(Dirichlet, a=ones(num_components,))
for k = 1:num_components
    marginals[:μ_*k] = ProbabilityDistribution(Multivariate, GaussianMeanPrecision, m=m0[:,k], w=I
d)
    marginals[:Λ_*k] = ProbabilityDistribution(Wishart, v=Id, nu=num_features)
end
for i = 1:num_samples
    marginals[:z_*i] = ProbabilityDistribution(Categorical, p=ones(num_components,)./num_component
s)
end

# Number of iterations
num_iterations = 20

# Preallocate free energy tracking array
F = Float64[]

# Execute algorithm
@showprogress for i = 1:num_iterations

    # Update assignments
    stepz!(data, marginals)

    # Update parameters
    stepϕ!(data, marginals)
    stepμ_1!(data, marginals)
    stepΛ_1!(data, marginals)
    stepμ_2!(data, marginals)
    stepΛ_2!(data, marginals)
    stepμ_3!(data, marginals)
    stepΛ_3!(data, marginals)

    # Store variational free energy for visualization
    push!(F, freeEnergy(data, marginals))
end
```

Alright, we're done. Let's track the evolution of free energy.

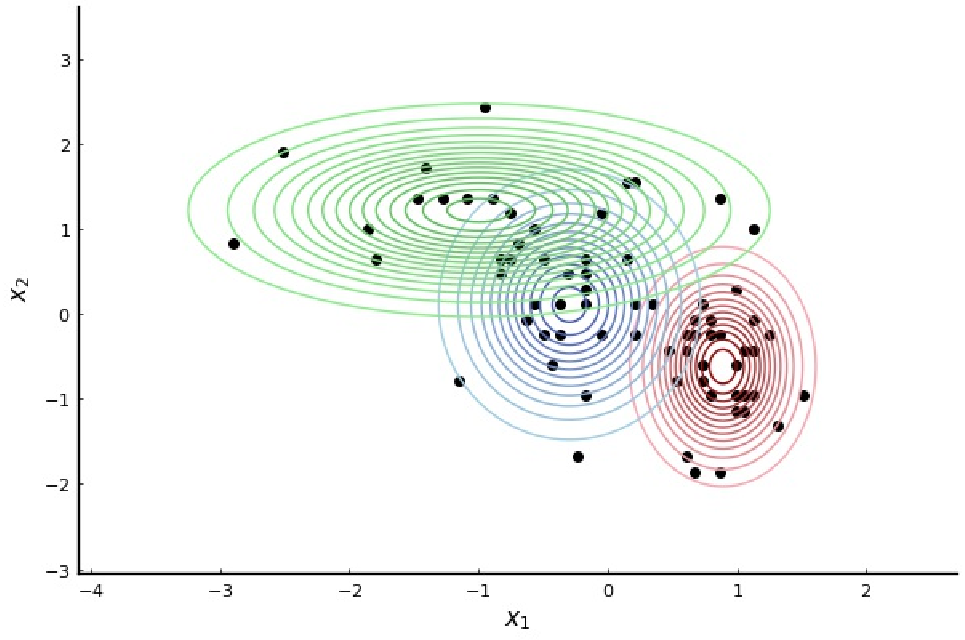
```
# Plot free energy to check for convergence
plot(1:num_iterations, F, color="black", label="", xlabel="Number of iterations", ylabel="Free Ene
rgy", size=(800,300))
```



That looks like it is nicely decreasing. We might want to increase the number of iterations a bit more.

Let's now visualize the cluster on top of the observations.

```
# Estimated means (unsafeMean retrieves parameters directly)
μ1_estimated = unsafeMean(marginals[:μ_1])
μ2_estimated = unsafeMean(marginals[:μ_2])
μ3_estimated = unsafeMean(marginals[:μ_3])

# Estimated precisions
Λ1_estimated = unsafeMean(marginals[:Λ_1])
Λ2_estimated = unsafeMean(marginals[:Λ_2])
Λ3_estimated = unsafeMean(marginals[:Λ_3])

# Invert to covariances
Σ1_estimated = inv(Λ1_estimated)
Σ2_estimated = inv(Λ2_estimated)
Σ3_estimated = inv(Λ3_estimated)

# Select dimensions to plot
dims_plot = [1, 2]
dim_limsx = [minimum(dataset["data"][:,dims_plot[1]])-1, maximum(dataset["data"][:,dims_plot[1]])+
1]
dim_limsy = [minimum(dataset["data"][:,dims_plot[2]])-1, maximum(dataset["data"][:,dims_plot[2]])+
1]

# Plot data and overlay estimated posterior probabilities
plot_clusters(dataset["data"][:, dims_plot],
              μ=[μ1_estimated[dims_plot], μ2_estimated[dims_plot], μ3_estimated[dims_plot]],
              Σ=[Σ1_estimated[dims_plot,dims_plot], Σ2_estimated[dims_plot,dims_plot], Σ3_estimate
d[dims_plot,dims_plot]],
              x1=range(dim_limsx[1], step=0.01, stop=dim_limsx[2]),
              x2=range(dim_limsy[1], step=0.01, stop=dim_limsy[2]),
              colorlist=[:reds, :blues, :greens],
              size=(600,400))
```



That doesn't look bad. The three Gaussians nicely cover all samples.

*** Try for yourself**

Play around with the number of components. Can you get an equally good coverage with just 2 components? What if you had 4?

Problem: Alpine Railways

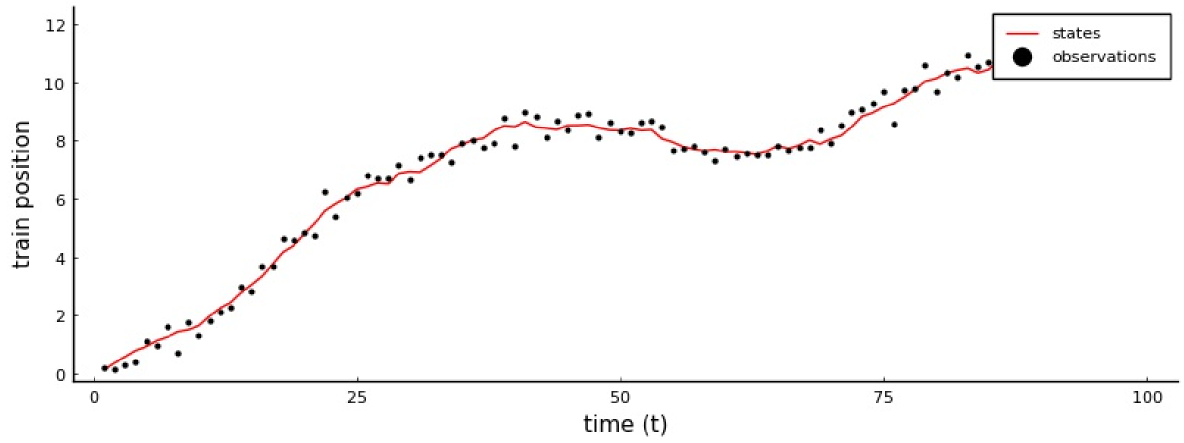
The Swiss Federal Railways company operates a series of mountain railways bringing hikers (in the summer) and skiers (in the winter) up the Alps. They are setting up a new fallback security system where they intend to track trains through cameras and remote sensors. They want you to design a system to keep track of the trains' positions.

Data

A train going uphill updates its position according to: new position = old position + velocity x length of time-step + noise. The noise represents the train randomly slipping and sliding back down. We observe the train through a remote sensor, producing noisy observations of its position.

You receive a data set with past recordings. Your job is to set up an online filtering system, which can be deployed later on to process the incoming signal in real-time.

```
signal = load("../datasets/alpinerails_filtering.jld");

# Unpack data
states = signal["X"]
observations = signal["Y"]
transition = signal["A"]
emission = signal["C"]
process_noise = signal["Q"]
measurement_noise = signal["R"]
T = signal["T"]
Δt = signal["Δt"];

# Size
M = size(states,1)
N = size(observations,1)

# Visualize
plot(1:T, states[1,:], color="red", label="states", grid=false, xlabel="time (t)", ylabel="train p
osition")
scatter!(1:T, observations[1,:], markersize=2, color="black", label="observations", size=(800,300)
)
```



Model specification

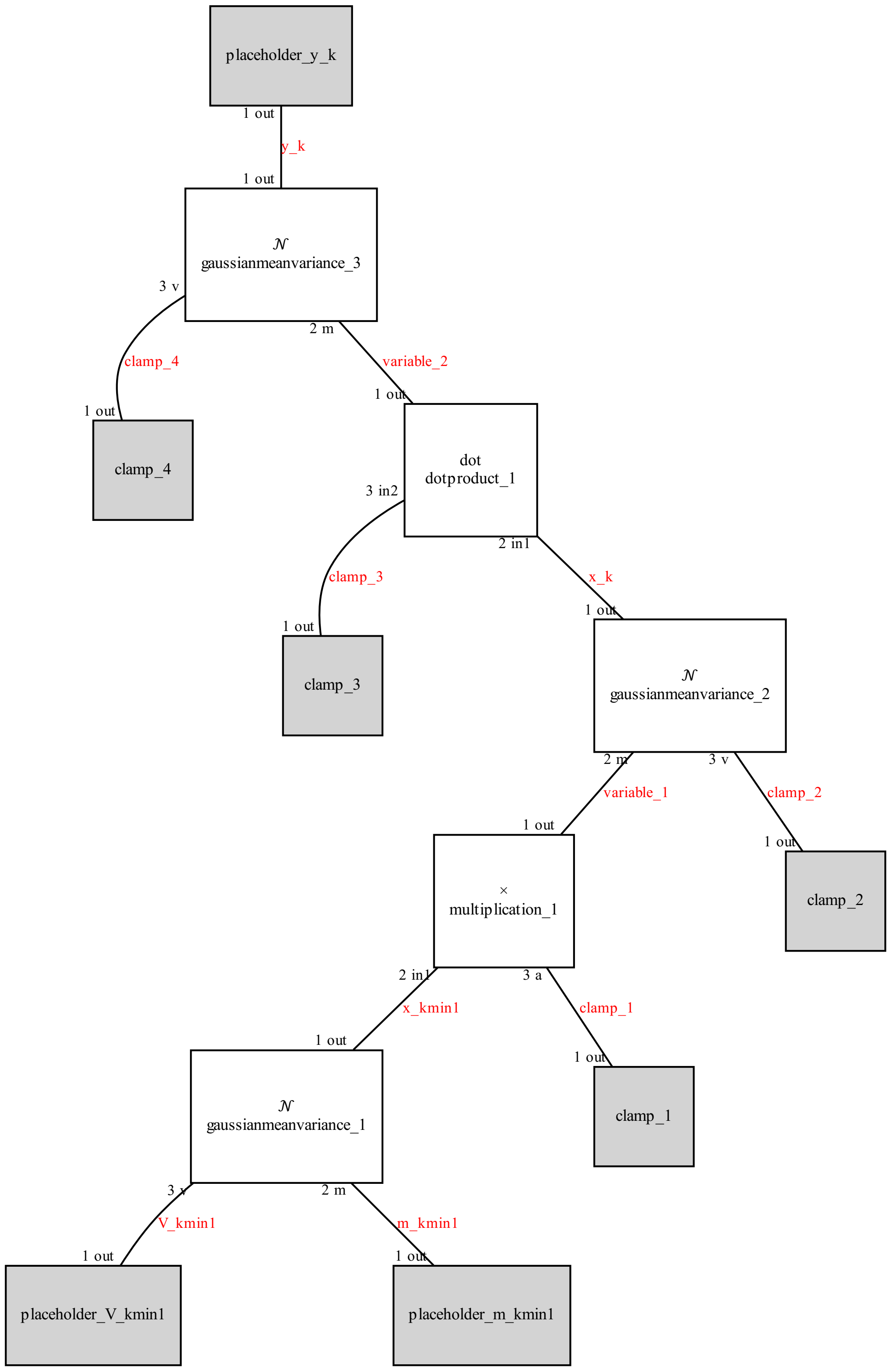
We define noisy observations $y_k \in \mathbb{R}^1$ with latent states $x_k \in \mathbb{R}^2$. Observations y_k are generated through a Gaussian likelihood centered on an emission matrix C times the current state x_k perturbed by measurement noise with covariance matrix R. State transitions follow a Gaussian distribution centered on a transition matrix A times the previous state perturbed by process noise with covariance matrix Q. In equation form, these are:

$$p(x_k \mid x_{k-1}) = \mathcal{N}(x_k \mid Ax_{k-1}, Q) \tag{19}$$

$$p(y_k \mid x_k) = \mathcal{N}(y_k \mid Cx_k, R)\,. \tag{20}$$

We have a prior for the previous state $x_{k-1} \sim \mathcal{N}(m_{k-1}, V_{k-1})$. In filtering problems, we feed the estimates of the current states as the parameters for the previous state in the next time-step.

```
# Initialize a graph
graph2 = FactorGraph()

# Define initial state prior
@RV x_kmin1 ~ GaussianMeanVariance(placeholder(:m_kmin1, dims=(M,)),
                                   placeholder(:V_kmin1, dims=(M,M)))

# State transition
@RV x_k ~ GaussianMeanVariance(transition * x_kmin1, process_noise)

# Observation likelihood
@RV y_k ~ GaussianMeanVariance(dot(emission, x_k), measurement_noise)

# Tell FL that y is observed
placeholder(y_k, :y_k);

# Visualize subgraph
ForneyLab.draw(graph2)
```



We'll now construct the algorithm and infer results.

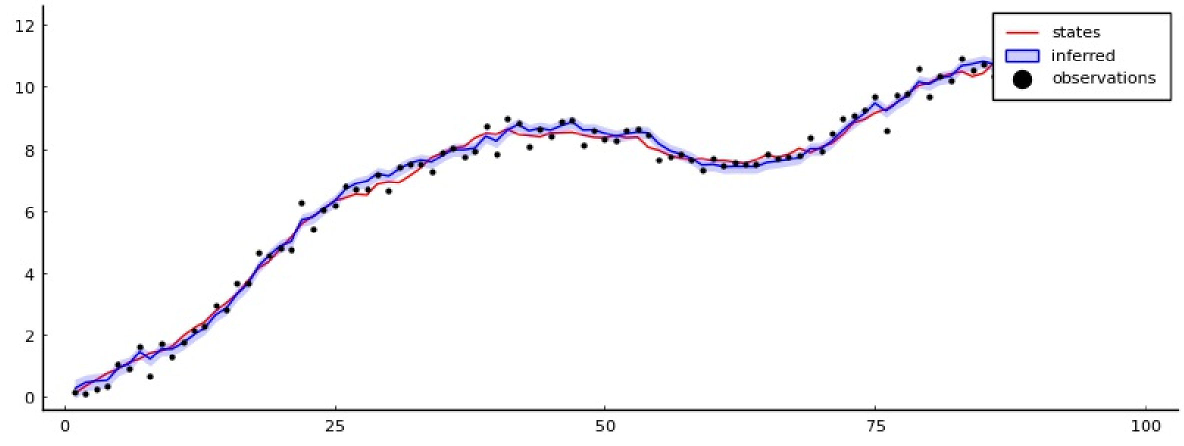
```
# Generate inference algorithm
algorithm = messagePassingAlgorithm(x_k)
source_code = algorithmSourceCode(algorithm)
eval(Meta.parse(source_code));
```

For filtering, we use the same graph in each time-step.

```
# Initialize recognition distribution marginals
marginals = Dict(:x_k => vague(GaussianMeanVariance, M))

# Initialize message array
messages = Array{Message}(undef, 5)

# Keep track of estimates
m_x = 10*ones(M,T+1)
V_x = repeat(10*Matrix{Float64}(I,M,M), outer=(1,1,T+1))

@showprogress for k = 1:T

    # Initialize data
    data = Dict(:y_k => observations[k],
                :m_kmin1 => m_x[:,k],
                :V_kmin1 => V_x[:,:,k])

    # Update states
    step!(data, marginals, messages)

    # Store estimates
    m_x[:,k+1] = mean(marginals[:x_k])
    V_x[:,:,k+1] = cov(marginals[:x_k])

end
```

Let's check whether everything went ok. We'll visualize the state estimations.

```
# Visualization
plot(1:T, states[1,:], color="red", label="states", grid=false)
plot!(1:T, m_x[1,2:end], ribbon=[sqrt.(V_x[1,1,2:end]), sqrt.(V_x[1,1,2:end])], fillalpha=0.2, col
or="blue", label="inferred")
scatter!(1:T, observations[1,:], markersize=2, color="black", label="observations", size=(800,300)
)
```



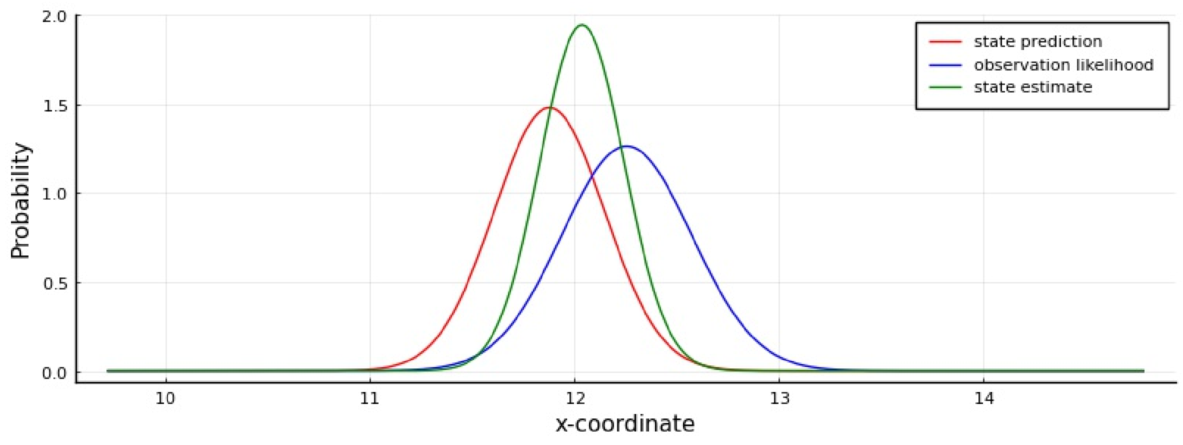
We're going to inspect some messages. Let's open up the algorithm and look up the marginal computation for the final x_k. It will be the multiplication of two messages, one consisting of the state transition prediction and the other consisting of the measurement likelihood.

```
println(source_code)
```

```
begin

function step!(data::Dict, marginals::Dict=Dict(), messages::Vector{Message}=Array{Message}
(undef, 5))

messages[1] = ruleSPGaussianMeanVarianceOutNPP(nothing, Message(Multivariate, PointMass, m=
data[:m_kmin1]), Message(MatrixVariate, PointMass, m=data[:V_kmin1]))
messages[2] = ruleSPMultiplicationOutNGP(nothing, messages[1], Message(MatrixVariate, Point
Mass, m=[1.0 0.1; 0.0 1.0]))
messages[3] = ruleSPGaussianMeanVarianceOutNGP(nothing, messages[2], Message(MatrixVariate,
 PointMass, m=[0.01 0.0; 0.0 0.1]))
messages[4] = ruleSPGaussianMeanVarianceMPNP(Message(Univariate, PointMass, m=data[:y_k]),
nothing, Message(Univariate, PointMass, m=0.1))
messages[5] = ruleSPDotProductIn1GNP(messages[4], nothing, Message(Multivariate, PointMass,
 m=[1.0, 0.0]))

marginals[:x_k] = messages[3].dist * messages[5].dist

return marginals

end

end # block
```

Alright, we need messages 3 and 5. Let's visualize them along with the state marginal.

```
# Collect state transition prediction
m_statepred = convert(ProbabilityDistribution{Multivariate, GaussianMeanVariance}, messages[3].dis
t)

# Collect observation likelihood
m_likelihood = convert(ProbabilityDistribution{Multivariate, GaussianMeanVariance}, messages[5].di
st)

# Collect corrected prediction
state_marginal = convert(ProbabilityDistribution{Multivariate, GaussianMeanVariance}, marginals[:x
_k])

# # Extract x-coordinates
m_statepred_x = ProbabilityDistribution(Univariate, GaussianMeanVariance, m=m_statepred.params[:m]
[1], v=m_statepred.params[:v][1,1])
m_likelihood_x = ProbabilityDistribution(Univariate, GaussianMeanVariance, m=m_likelihood.params[:
m][1], v=m_likelihood.params[:v][1,1])
state_marginal_x = ProbabilityDistribution(Univariate, GaussianMeanVariance, m=state_marginal.para
ms[:m][1], v=state_marginal.params[:v][1,1])

# Plot of the prediction, noisy measurement, and corrected prediction for x-coordinate
plot_messages(m_statepred_x, m_likelihood_x, state_marginal_x, size=(800,300))
```



As you can see, the state estimate is a combination of the state prediction, produced by the message from the state transition node, and the observation likelihood, produced by the message from the likelihood.

---

* Try for yourself

Re-run the inference procedure and stop at an earlier time-step, for example $\boldsymbol{k}$=2. How does the balance between the state prediction and the observation likelihood differ?
