```julia
begin
    import Pkg
    # activate the shared project environment
    Pkg.activate(Base.current_project())
    using Omega, Distributions, UnicodePlots, OmegaExamples
    using Images, Plots
end
```

```
Activating project at `~/Documents/GitHub/Omega.jl/OmegaExamples`
```

In the chapter on Hierarchical Models, we saw the power of probabilistic inference in learning about the latent structure underlying different kinds of observations: the mixture of colors in different bags of marbles, or the prototypical features of categories of animals. In that discussion we always assumed that we knew what kind each observation belonged to—the bag that each marble came from, or the subordinate, basic, and superordinate category of each object. Knowing this allowed us to pool the information from each observation for the appropriate latent variables. What if we don't know a *priori* how to divide up our observations? In this chapter we explore the problem of simultaneously discovering kinds and their properties – this can be done using *mixture models*.

Learning Categories

Imagine a child who enters the world and begins to see objects. She can't begin to learn the typical features of cats or mice directly, because she doesn't yet know that there are such kinds of objects as cats and mice. Yet she may quickly notice that some of the objects all tend to purr and have claws, while other objects are small and run fast—she can *cluster* the objects together on the basis of common features and thus form categories (such as cats and mice), whose typical features she can then learn.

To formalize this learning problem, we begin by adapting the bags-of-marbles examples from the Hierarchical Models chapter. However, we now assume that the bag that each marble is drawn from is *unobserved* and must be inferred.

colours = [:blue, :green, :red]

```
colours = [:blue, :green, :red]
```

ϕ = Dirichlet([1.0, 1.0, 1.0])

```
ϕ = OmegaExamples.Dirichlet(3, 1)
```

α = 0.1

```
α = 0.1
```

prototype (generic function with 1 method)

```
prototype(i, ω) = (i ~ ϕ)(ω) .* α
```

colour_probs (generic function with 1 method)

```
colour_probs(i, ω) = (i ~ OmegaExamples.Dirichlet(prototype(i, ω)))(ω)
```

make_bag (generic function with 1 method)

```
make_bag(i, ω) = (pget(colours) ∘ (i ~ Categorical((i ~ colour_probs)(ω))))(ω)
```

obs_to_bag = Distributions.DiscreteUniform(a=1, b=3)

```
obs_to_bag = DiscreteUniform(1, 3)
```

obs_fn (generic function with 1 method)

```
obs_fn(data) =
    Variable(ω -> all(map(i -> make_bag(obs_to_bag(i, ω), ω) == data[i],
    1:length(data))))
```

false

```
randsample(obs_fn(obs))
```

obs = [:red, :red, :blue, :blue, :red, :blue]

```
obs = [:red, :red, :blue, :blue, :red, :blue]
```

```
obs123 (generic function with 1 method)
```

```julia
obs123(ω::Ω) = (obs1=(1~obs_to_bag)(ω), obs2=(2~obs_to_bag)(ω), obs3=(3~obs_to_bag)
(ω))
```

```
samples123 =
  [(obs1 = 2, obs2 = 2, obs3 = 1), (obs1 = 2, obs2 = 2, obs3 = 1), (obs1 = 1, obs2 = 1, obs3 = 3),
```

```julia
samples123 = randsample(obs123 |ᶜ obs_fn(obs), 1000)
```

```
BitVector: [true, true, true, false, true, true, true, true, true, false, true, true, true, tr
```

```julia
same_bag_1and2::Vector{Bool} =
        getfield.(samples123, :obs1) .== getfield.(samples123, :obs2)
```

```
BitVector: [false, false, false, false, false, false, false, false, false, false, false, fals
```

```julia
same_bag_1and3::Vector{Bool} =
        getfield.(samples123, :obs1) .== getfield.(samples123, :obs3)
```

```
false ┤■■■■■■■■■■ 228
 true ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 772
```

```julia
viz(same_bag_1and2)
```

```
false ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 1 000
```

```julia
viz(same_bag_1and3)
```

We see that it is likely that observations `1` and `2` came from the same bag, but quite unlikely that `3` did. Why? Notice that we have set `α` small, indicating a belief that the marbles in a bag will tend to all be the same color. How do the results change if you make `α` larger? Why? Note that we have queried on whether observed marbles came out of the same bag, instead of directly querying on the bag number that an observation came from. This is because the bag number by itself is meaningless—it is only useful in its role of determining which objects have similar properties. Formally, the model we have defined above is symmetric in the bag labels (if you permute all the labels you get a new state with the same probability).

Instead of assuming that a marble is equally likely to come from each bag, we could instead learn a distribution over bags where each bag has a different probability. This is called a *mixture distribution* over the bags:

```
bag_mixture =   Dirichlet([1.0, 1.0, 1.0])
```

```julia
# the probability that an observation will come from each bag:
bag_mixture = OmegaExamples.Dirichlet(3, 1)
```

```
obs_to_bag_mix (generic function with 1 method)
```

```julia
obs_to_bag_mix(i, ω) = (i~ Categorical((i~ bag_mixture)(ω)))(ω)
```

```
obs_fn_mix (generic function with 1 method)
```

```julia
obs_fn_mix(data) =
    Variable(ω -> all(map(i -> make_bag(obs_to_bag_mix(i, ω), ω) == data[i],
    1:length(data))))
```

```
[(obs1 = 3, obs2 = 1, obs3 = 2), (obs1 = 2, obs2 = 2, obs3 = 3), (obs1 = 3, obs2 = 3, obs3 = 1),
```

```julia
begin
    obs123_mix(ω::Ω) = (obs1=(1~obs_to_bag_mix)(ω), obs2=(2~obs_to_bag_mix)(ω), obs3=
    (3~obs_to_bag_mix)(ω))
    samples123_mix = randsample(obs123_mix |ᶜ obs_fn_mix(obs), 1000)
end
```

```
BitVector: [false, true, true, true, true, true, true, true, true, false, true, false, true, t
```

```julia
same_bag_1and2_mix::Vector{Bool} =
        getfield.(samples123_mix, :obs1) .== getfield.(samples123_mix, :obs2)
```

```
BitVector: [false, false, false, false, false, false, false, false, false, false, false, fals
```

```julia
same_bag_1and3_mix::Vector{Bool} =
        getfield.(samples123_mix, :obs1) .== getfield.(samples123_mix, :obs3)
```

```
        ┌                                        ┐
  false ┤■■■■■■■■■■■■■■■■ 305
   true ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 695
        └                                        ┘
```

```julia
viz(same_bag_1and2_mix)
```

```
        ┌                                        ┐
  false ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 1 000
        └                                        ┘
```

```julia
viz(same_bag_1and3_mix)
```

Models of this kind are called **mixture models** because the observations are a “mixture” of several categories. Mixture models are widely used in modern probabilistic modeling because they describe how to learn the unobservable categories which underlie observable properties in the world.

The observation distribution associated with each mixture *component* (i.e., kind or category) can be any distribution we like. For example, here is a mixture model with *Normal* components:

```
data =
  [(x = 1.53439, y = 2.34609), (x = 1.18101, y = 1.44715), (x = 1.33595, y = 0.59791), (x = 1.7
```

```julia
data = [(x = 1.5343898902525506, y = 2.3460878867298494),
    (x = 1.1810142951204246, y = 1.4471493362364427),
    (x = 1.3359476185854833, y = 0.5979097803077312),
    (x = 1.7461500236610696, y = 0.07441351219375836),
    (x = 1.1644280209698559, y = 0.5504283671279169),
    (x = 0.5383179421667954, y = 0.36076578484371535),
    (x = 1.5884794217838352, y = 1.2379018386693668),
    (x = 0.633910148716343, y = 1.21804947961078),
    (x = 1.3591395983859944, y = 1.2056207607743645),
    (x = 1.5497995798191613, y = 1.555239222467223),
    (x = -1.7103539324754713, y = -1.178368516925668),
    (x = -0.49690324128135566, y = -1.4482931166889297),
    (x = -1.0191455290951414, y = -0.4103273022785636),
    (x = -1.6127046244033036, y = -1.198330563419632),
    (x = -0.8146486481025548, y = -0.33650743701348906),
    (x = -1.2570582864922166, y = -0.7744102418371701),
    (x = -1.2635542813354101, y = -0.9202555846522052),
    (x = -1.3169953429184593, y = -0.40784942495184096),
    (x = -0.7409787028330914, y = -0.6105091049436135),
    (x = -0.7683709878962971, y = -1.0457286452094976)]
```

```
cat_mixture = -991692059603514511@OmegaExamples.Dirichlet{Vector{Float64}}([1.0, 1.0])
```

```julia
cat_mixture = @~ OmegaExamples.Dirichlet(2, 1)
```

```
obs_to_cat (generic function with 1 method)
```

```julia
obs_to_cat(i, ω) = (i ~ Categorical(cat_mixture(ω)))(ω)
```

```
x_mean (generic function with 1 method)
```

```julia
x_mean(i, ω) = ((@uid, i) ~ StdNormal{Float64}())(ω)
```

```
y_mean (generic function with 1 method)
```

```julia
y_mean(i, ω) = ((@uid, i) ~ StdNormal{Float64}())(ω)
```

```
cat_to_mean (generic function with 1 method)
```

```julia
cat_to_mean(i, ω) = (x_mean = x_mean(i, ω), y_mean = y_mean(i, ω))
```

```
predictives (generic function with 1 method)
```

```julia
function predictives(data, ω)
    for (i, d) in enumerate(data)
        mus = cat_to_mean(obs_to_cat(i, ω), ω)
        cond!(ω, (i ~ Normal(mus.x_mean, 0.01))(ω) == d.x)
        cond!(ω, ((@uid, i) ~ Normal(mus.y_mean, 0.01))(ω) == d.y)
    end
    return cat_to_mean(obs_to_cat(@uid, ω), ω)
end
```

```
  [(x_mean = -0.793278, y_mean = -0.705885), (x_mean = -0.793278, y_mean = -0.705885), (x_mean =
```

```julia
randsample(ω -> predictives(data, ω), 100, alg = MH)
```

Example: Categorical Perception of Speech Sounds

This example is adapted from Feldman et al. (2009).

Human perception is often skewed by our expectations. A common example of this is called *categorical perception* – when we perceive objects as being more similar to the category prototype than they really are. In phonology this is been particularly important and is called the perceptual magnet effect: Hearers regularize a speech sound into the category that they think it corresponds to. Of course this category isn't known a priori, so a hearer must be doing a simultaneous inference of what category the speech sound corresponded to, and what the sound must have been. In the below code we model this as a mixture model over the latent categories of sounds, combined with a noisy observation process.

```
prototype_1 = 0
```

```
prototype_1 = 0
```

```
prototype_2 = 5
```

```
prototype_2 = 5
```

```
stimuli = 0.0:0.2:5.0
```

```
stimuli = prototype_1 : 0.2 : prototype_2
```

```
vowel_1 = 1590624885156798913@Distributions.Normal{Float64}(μ=0.0, σ=1.0)
```

```
vowel_1 = @~ Normal(prototype_1, 1)
```

```
vowel_2 = -7743763858335340153@Distributions.Normal{Float64}(μ=5.0, σ=1.0)
```

```
vowel_2 = @~ Normal(prototype_2, 1)
```

```
category = -1113466418018157009@Distributions.Bernoulli{Float64}(p=0.5)
```

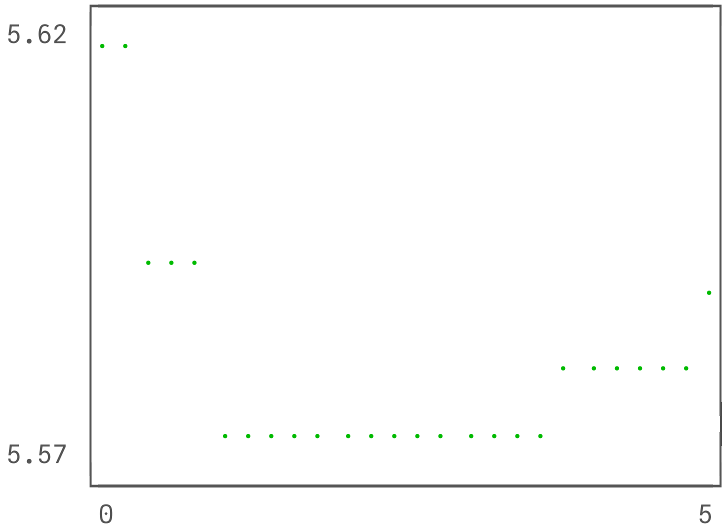
```
category = @~ Bernoulli()
```

```
value =
ifelseₚ(-1113466418018157009@Distributions.Bernoulli{Float64}(p=0.5), 1590624885156798913@D
```

```
value = ifelse.(category, vowel_1, vowel_2)
```

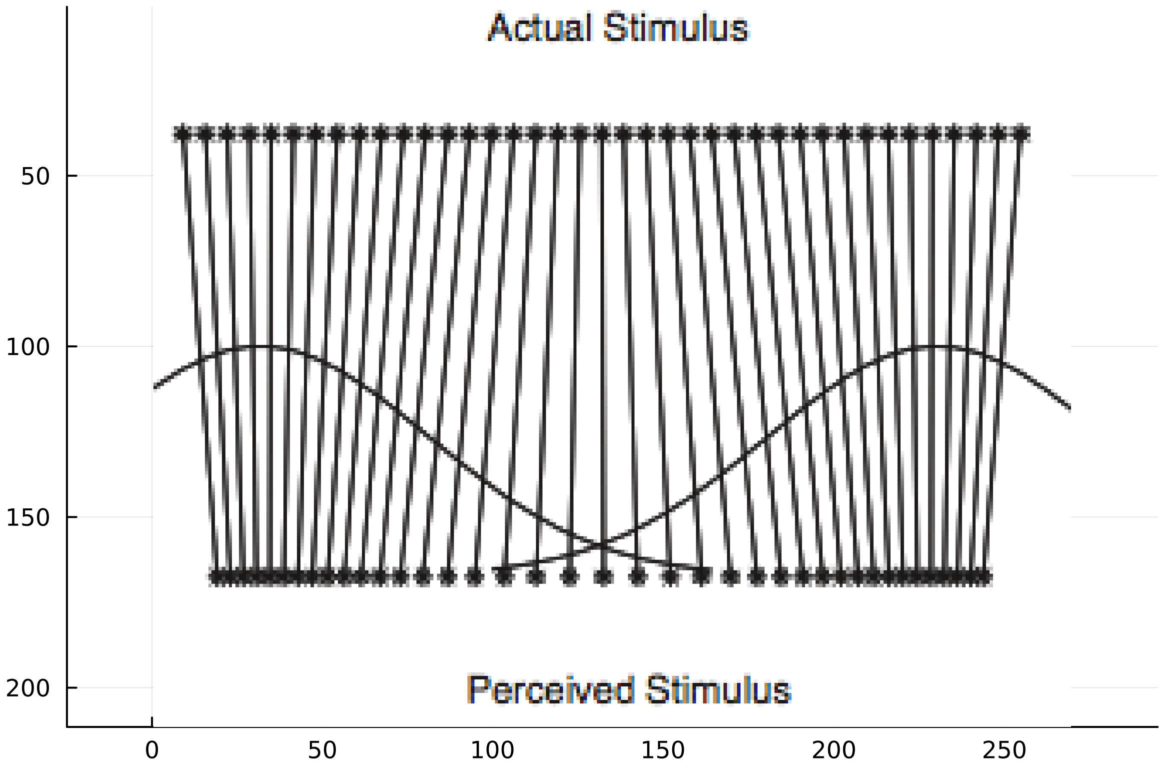
```
perceived_value =
  Conditional(ifelseₚ(-1113466418018157009@Distributions.Bernoulli{Float64}(p=0.5), 15906248
```

```
perceived_value =
    value |ᶜ pw(==, manynth(Normal(value, 1), 1:length(stimuli)), stimuli)
```




```
scatterplot(stimuli, randsample(perceived_value, length(stimuli), alg = MH))
```

Notice that the perceived distances between input sounds are skewed relative to the actual acoustic distances – that is they are attracted towards the category centers.



Example: Topic Models

One very popular class of mixture-based approaches are *topic models*, which are used for document classification, clustering, and retrieval. The simplest kind of topic models make the assumption that documents can be represented as *bags of words* — unordered collections of the words that the document contains. In topic models, each document is associated with a mixture over *topics*, each of which is itself a distribution over words. (Sometimes models like this, where the observations are a mixture of different mixtures is called an *admixture* model.)

One popular kind of bag-of-words topic model is known as *Latent Dirichlet Allocation* (LDA, Blei et al. (2003)). The generative process for this model can be described as follows. For each document, mixture weights over a set of K topics are drawn from a Dirichlet prior. Then N topics are sampled for the document—one for each word. Each topic itself is associated with a distribution over words, and this distribution is drawn from a Dirichlet prior. For each of the N topics drawn for the document, a word is sampled from the corresponding multinomial distribution. This is shown in the code below.

```
vocabulary = ["DNA", "evolution", "parsing", "phonology"]
```

```julia
vocabulary = ["DNA", "evolution", "parsing", "phonology"]
```

```
η = [1, 1, 1, 1]
```

```julia
η = ones(Int64, length(vocabulary))
```

```
num_topics = 2
```

```julia
num_topics = 2
```

```
alpha = [1, 1]
```

```julia
alpha = ones(Int64, num_topics)
```

```
corpus =
 [["DNA", "evolution", "DNA", "evolution", "DNA", "evolution", "DNA", "evolution", "DNA", "ev
```

```julia
corpus = split.([
    "DNA evolution DNA evolution DNA evolution DNA evolution DNA evolution",
  "DNA evolution DNA evolution DNA evolution DNA evolution DNA evolution",
  "DNA evolution DNA evolution DNA evolution DNA evolution DNA evolution",
  "parsing phonology parsing phonology parsing phonology parsing phonology parsing phonology",
  "parsing phonology parsing phonology parsing phonology parsing phonology parsing phonology",
  "parsing phonology parsing phonology parsing phonology parsing phonology parsing phonology"
])
```

```
topics = [ᵛ#22, ᵛ#22]
```

```julia
topics = [Variable(ω -> (i~ OmegaExamples.Dirichlet(η))(ω)) for i in 1:num_topics]
```

```
topic_dist (generic function with 1 method)
```

```julia
topic_dist(i, ω) = (i~ OmegaExamples.Dirichlet(alpha))(ω)
```

```
z (generic function with 1 method)
```

```julia
z(i, ω) = (i~ Categorical((i~topic_dist)(ω)))(ω)
```

```
topic (generic function with 1 method)
```

```julia
topic(i, ω) = topics[(i~z)(ω)](ω)
```

```
rand_doc (generic function with 1 method)
```

```julia
rand_doc(doc) =
manynth((i, ω) -> (pget(vocabulary) ∘ (i~Categorical(topic(i, ω))))(ω), 1:length(doc))
```

```
evidence (generic function with 1 method)
```

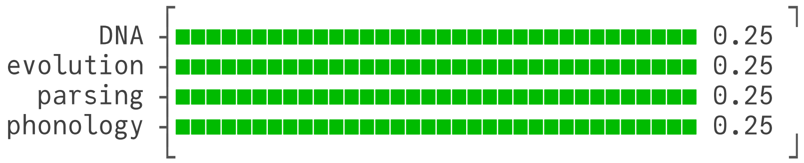
```julia
evidence(ω) = map(doc -> (rand_doc(doc) .== doc)(ω), corpus)
```

```
model =
  Conditional(#15 (generic function with 1 method), ᵛall ∘ Main.var"workspace#175".evidence)
```

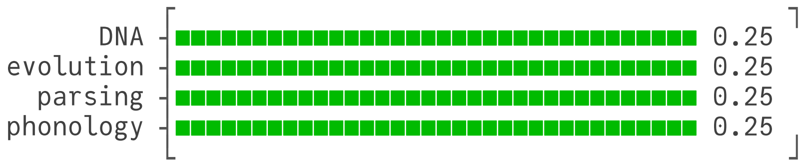
```julia
model = (ω -> mapf(ω, topics)) |ᶜ Variable(all ∘ evidence)
```

```
results =
  Vector{Vector{Float64}}[
     1:   [[0.0416587, 0.562912, 0.338243, 0.0571865], [0.172645, 0.367853, 0.33185, 0.127652
     2:   [[0.0476088, 0.643313, 0.243724, 0.0653545], [0.172645, 0.367853, 0.33185, 0.127652
     3:   [[0.0561696, 0.579175, 0.287549, 0.0771063], [0.172645, 0.367853, 0.33185, 0.127652
     4:   [[0.0561696, 0.579175, 0.287549, 0.0771063], [0.142257, 0.303107, 0.449452, 0.10518
     5:   [[0.0561696, 0.579175, 0.287549, 0.0771063], [0.155468, 0.331254, 0.491188, 0.02209
     6:   [[0.0544909, 0.561865, 0.278955, 0.104689], [0.155468, 0.331254, 0.491188, 0.022090
     7:   [[0.0544909, 0.561865, 0.278955, 0.104689], [0.0581395, 0.369429, 0.547795, 0.02463
     8:   [[0.0544909, 0.561865, 0.278955, 0.104689], [0.0547182, 0.347689, 0.515559, 0.08203
     9:   [[0.0387367, 0.571227, 0.283603, 0.106434], [0.0547182, 0.347689, 0.515559, 0.08203
     10:   [[0.0387367, 0.571227, 0.283603, 0.106434], [0.0590294, 0.296294, 0.55618, 0.08849
     11:   [[0.0387367, 0.571227, 0.283603, 0.106434], [0.069172, 0.347205, 0.479921, 0.10370
     12:   [[0.0387367, 0.571227, 0.283603, 0.106434], [0.0819579, 0.226541, 0.56863, 0.12287
     13:   [[0.0387367, 0.571227, 0.283603, 0.106434], [0.0965369, 0.0889547, 0.66978, 0.1447
     14:   [[0.0373595, 0.586471, 0.27352, 0.102649], [0.0965369, 0.0889547, 0.66978, 0.14472
     15:   [[0.0351737, 0.552159, 0.316024, 0.0966438], [0.0965369, 0.0889547, 0.66978, 0.144
     16:   [[0.0351737, 0.552159, 0.316024, 0.0966438], [0.0922188, 0.0849758, 0.639821, 0.18
     17:   [[0.0305243, 0.55482, 0.317546, 0.0971095], [0.0922188, 0.0849758, 0.639821, 0.182
     18:   [[0.0305243, 0.55482, 0.317546, 0.0971095], [0.00195795, 0.0934249, 0.703439, 0.20
     19:   [[0.0, NaN, 0.0, 0.0], [0.00195795, 0.0934249, 0.703439, 0.201178]]
     20:   [[0.0347967, 0.492509, 0.361993, 0.110702], [0.00195795, 0.0934249, 0.703439, 0.20

       more

     991:   [[0.0, 0.764916, 0.152288, 0.0827967], [0.0848098, 0.36646, 0.452711, 0.0960199]]
     992:   [[0.0, 0.686647, 0.239028, 0.0743248], [0.0848098, 0.36646, 0.452711, 0.0960199]]
     993:   [[0.0, 0.577632, 0.201079, 0.221289], [0.0848098, 0.36646, 0.452711, 0.0960199]]
     994:   [[0.0, 0.577632, 0.201079, 0.221289], [0.136275, 0.345852, 0.427253, 0.0906202]]
     995:   [[0.0, 0.577632, 0.201079, 0.221289], [0.235096, 0.306282, 0.37837, 0.0802521]]
     996:   [[0.0, 0.569947, 0.211709, 0.218345], [0.235096, 0.306282, 0.37837, 0.0802521]]
     997:   [[0.0, 0.723015, 0.0, 0.276985], [0.235096, 0.306282, 0.37837, 0.0802521]]
     998:   [[0.0, 0.723015, 0.0, 0.276985], [0.24666, 0.321348, 0.347792, 0.0841996]]
     999:   [[0.0, 0.723015, 0.0, 0.276985], [0.252897, 0.329473, 0.356586, 0.0610442]]
     1000:   [[0.0, 0.723015, 0.0, 0.276985], [0.252897, 0.329473, 0.356586, 0.0610442]]
]
```

```julia
results = randsample(model, 1000, alg = MH)
```




```julia
barplot(vocabulary, map(x -> mean(first.(results)[x]), 1:4))
```




```julia
barplot(vocabulary, map(x -> mean(last.(results)[x]), 1:4))
```

In this simple example, there are two topics `1` and `2`, and four words. These words are deliberately chosen to represent one of two possible subjects that a document can be about: One can be thought of as 'biology' (i.e., `DNA` and `evolution`), and the other can be thought of as 'linguistics' (i.e., `parsing` and `syntax`).

The documents consist of lists of individual words from one or the other topic. Based on the co-occurrence of words within individual documents, the model is able to learn that one of the topics should put high probability on the biological words and the other topic should put high probability on the linguistic words. It is able to learn this because different kinds of documents represent stable mixture of different kinds of topics which in turn represent stable distributions over words.

Unknown Numbers of Categories

The models above describe how a learner can simultaneously learn which category each object belongs to, the typical properties of objects in that category, and even global parameters about kinds of objects in general. However, it suffers from a serious flaw: the number of categories was fixed. This is as if a learner, after finding out there are cats, dogs, and mice, must force an elephant into one of these categories, for want of more categories to work with.

The simplest way to address this problem, which we call *unbounded* models, is to simply place uncertainty on the number of categories in the form of a hierarchical prior. Let's warm up with a simple example of this: inferring whether one or two coins were responsible for a set of outcomes (i.e. imagine a friend is shouting each outcome from the next room–"heads, heads, tails…"–is she using a fair coin, or two biased coins?).

```
observed_data =  [true, true, true, true, false, false, false, false]
```

```
observed_data = [true, true, true, true, false, false, false, false]
# observed_data = [true, true, true, true, true, true, true, true]
```

```
coins = ᵛ#33
```

```
coins = Variable(ω -> ((@~Bernoulli())(ω) ? 1 : [1, 2]))
```

```
coin_to_weight =  StdUniform()
```

```
coin_to_weight = StdUniform{Float64}()
```

```
rand_obs (generic function with 1 method)
```

```
rand_obs(i, ω) =
    (i~Bernoulli(((i ~ UniformDraw(coins(ω)))(ω) ~ coin_to_weight)(ω)))(ω)
```

```
coins_cond =
 Conditional(length ∘ ᵛ#33, ==ₚ(Mv{UnitRange{Int64}, typeof(rand_obs)}(1:8, Main.var"workspa
```

```
coins_cond =
    (length ∘ coins) |ᶜ pw(==, manynth(rand_obs, 1:length(observed_data)),
    observed_data)
```

```
# viz(randsample(coins_cond, 1000))
```

How does the inferred number of coins change as the amount of data grows? Why? (Note that we have used the `RejectionSampling` inference method. Inference in unbounded mixture models can be very tricky because a different choice of dimensionality leads to different options for the category of each observation. For instance, in the case of Metropolis-Hastings this means that proposals to reduce the number of categories are almost always rejected – mixing is impossibly slow.)

We could extend this model by allowing it to infer that there are more than two coins. However, no evidence requires us to posit three or more coins (we can always explain the data as “a heads coin and a tails coin”). Instead, let us apply the same idea to the marbles examples above:

```
observed_marbles =  [:red, :red, :blue, :blue, :red, :blue]
```

```
observed_marbles = [:red, :red, :blue, :blue, :red, :blue]
```

```
num_bags (generic function with 1 method)
```

```
num_bags(ω) = (1 + (@~ Poisson(1))(ω))
```

```
bags (generic function with 1 method)
```

```
bags(i, ω) = (i~ UniformDraw(1:num_bags(ω)))(ω)
```

```
num_bags_cond =
 Conditional(num_bags (generic function with 1 method), ==ₚ(Mv{UnitRange{Int64}, typeof(bag
```

```
num_bags_cond =
    num_bags |ᶜ pw(==, manynth(bags, 1:length(observed_marbles)), observed_marbles)
```

```
# viz(randsample(num_bags_cond, 1000))
```

Vary the amount of evidence and see how the inferred number of bags changes.

For the prior on `num_bags` we used the *[Poisson distribution](https://en.wikipedia.org/wiki/Poisson_distribution)* which is a distribution on non-negative integers. It is convenient, though implies strong prior knowledge (perhaps too strong for this example).

Infinite mixtures

Unbounded models give a straightforward way to represent uncertainty over the number of categories in the world. However, inference in these models often presents difficulties. An alternative is to use *infinite* mixture models. In an unbounded model, there are a finite number of categories whose number is drawn from an unbounded prior distribution, such as the Poisson prior that we just examined. In an infinite model we assume an *infinite number* of categories (most not yet observed).

To understand how we can work with an infinite set of categories in a finite computer, let’s first revisit the Discrete distribution.

In Omega the categorical distribution is a random class, e.g. `Categorical([0.2, 0.3, 0.1, 0.4])`. If it wasn’t built-in and the only primitive random class you could use was `Bernoulli`, how could you sample from it? One solution is to recursively walk down the list of probabilities, deciding whether to stop on each step. For instance, in `Categorical([0.2, 0.3, 0.1, 0.4])` there is a 0.2 probability of stopping on the first trial, a $0.3/0.8$ probability of stopping on the second (given that we didn’t stop on the first), and so on. We can start by turning the list of probabilities into a list of residual probabilities—the probability we will stop on each step, given that we haven’t stopped yet:

```
categorical (generic function with 1 method)
```

```julia
function categorical(probs, ω)
    @assert (sum(probs) ≈ 1) "probs is not a probability vector"
    residuals = [probs[1]]
    for i in 2:length(probs)
        push!(residuals, probs[i]/(1 - sum(probs[1:i-1])))
    end
    residuals[end] = 1
    b(ω, k) = (k ~ Bernoulli(residuals[k]))(ω)
    for k in 1:length(probs)
        return b(ω, k) ? k : continue
    end
end
```

```
              ┌                                        ┐
[1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■ 199
[1.5, 2.0) ┤ 0
[2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 309
[2.5, 3.0) ┤ 0
[3.0, 3.5) ┤■■■■■■■■ 80
[3.5, 4.0) ┤ 0
[4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 412
              └                                        ┘
                            Frequency
```

```julia
viz(randsample(ω -> categorical([0.2, 0.3, 0.1, 0.4], ω), 1000))
```

In the above mixture model examples, we generally expressed uncertainty about the probability of each category by putting a Dirichlet prior on the probabilities passed to a Categorical distribution:

```
probs_dirichlet =
-2561981913411928605@OmegaExamples.Dirichlet{Vector{Float64}}([1.0, 1.0, 1.0, 1.0])
```

```julia
probs_dirichlet = @~ OmegaExamples.Dirichlet(4, 1)
```

```
              ┌                                        ┐
[1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 255
[1.5, 2.0) ┤ 0
[2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 258
[2.5, 3.0) ┤ 0
[3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 236
[3.5, 4.0) ┤ 0
[4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 251
              └                                        ┘
                            Frequency
```

```julia
viz(randsample(ω -> categorical(probs_dirichlet(ω), ω), 1000))
```

It makes sense to sample the residuals directly. Since we know that the residual probability is simply a number between 0 and 1, we could do something like:

```
residuals (generic function with 1 method)
```

```julia
residuals(ω) = vcat(manynth(Beta(1, 1), 1:3)(ω), 1)
```

```
categorical_ (generic function with 2 methods)
```

```julia
categorical_(resid, ω, i = 1) =
    (i~ Bernoulli(resid[i]))(ω) ? i : categorical_(resid, ω, i + 1)
```

```
[1.0, 1.5) ┤■■ 45
[1.5, 2.0) ┤ 0
[2.0, 2.5) ┤■■■ 61
[2.5, 3.0) ┤ 0
[3.0, 3.5) ┤■■ 46
[3.5, 4.0) ┤ 0
[4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 848

                    Frequency
```

```
viz(randsample(ω -> categorical_(residuals(ω), ω), 1000))
```

Notice that we have added a final residual probability of 1 to the array of residuals. This is to make sure we stop at the end! It is kind of ugly, though, and breaks the symmetry between the final number and the ones before. After staring at the above code you might have an idea: why bother stopping? If we had an infinite set of residual probs we could still call `categorical_`, and we would eventually stop each time. We can get the effect of an infinite set by using independent `Beta` distributions to only construct a particular value when we need it:

```
residuals_ = 2152978606906721840@Distributions.Beta{Float64}(α=1.0, β=1.0)
```

```
residuals_ = (@~ Beta(1, 1))
```

```
categorical_inf (generic function with 2 methods)
```

```
categorical_inf(ω, i = 1) =
    (i~ Bernoulli(residuals_(ω)))(ω) ? i : categorical_inf(ω, i+1)
```

```
itr =  randsample(categorical_inf, 1000);
```

```
525 ┤ 0
526 ┤ 0
527 ┤ 0
528 ┤ 0
529 ┤ 0
530 ┤ 0
531 ┤ 0
532 ┤ 0
533 ┤ 0
534 ┤ 0
535 ┤ 0
536 ┤ 0
537 ┤ 0
538 ┤ 0
539 ┤ 0
540 ┤ 0
541 ┤ 0
542 ┤ 0
543 ┤ 0
544 ┤ 0
545 ┤ 0
546 ┤ 0
547 ┤ 0
548 ┤ 0
549 ┤ 0
550 ┤ 0
551 ┤ 0
552 ┤ 0
553 ┤ 0
554 ┤ 0
555 ┤ 0
556 ┤ 0
557 ┤ 0
558 ┤ 0
```

```
barplot(1:maximum(itr), map(k -> count(x -> x == k, itr), 1:maximum(itr)))
```

We've just constructed an infinite analog of the Dirichlet-Discrete pattern, it is called a *Dirichlet Process* (DP, more technically this is a GEM: a DP over integers). We have derived the DP by generalizing the Discrete distribution, but we've arrived at something that also looks like a Geometric distribution with heterogeneous stopping probabilities, which is an alternative derivation.

We can use the DP to construct an *infinite mixture model*:

```
colors = [:blue, :red, :green]
```

```
colors = [:blue, :red, :green]
```

```
obs_marbles = [:red, :blue, :red, :blue, :red, :blue]
```

```
obs_marbles = [:red, :blue, :red, :blue, :red, :blue]
```

```
phi = Dirichlet([1.0, 1.0, 1.0])
```

```
phi = OmegaExamples.Dirichlet(3, 1)
```

```
α_ = 0.1
```

```
α_ = 0.1
```

```
prototype_inf (generic function with 1 method)
```

```
prototype_inf(i, ω) = α_ .* phi(i, ω)
```

```
make_bag_inf (generic function with 1 method)
```

```julia
function make_bag_inf(i, ω)
    colour_probs = (i~ OmegaExamples.Dirichlet(prototype_inf(i, ω)))(ω)
    (pget(colors) ∘ (i~ Categorical(colour_probs)))(ω)
end
```

```
obs_to_bag_inf (generic function with 2 methods)
```

```julia
obs_to_bag_inf(i, ω, k = 1) =
    ((i, k) ~ Bernoulli(residuals_(ω)))(ω) ? k : obs_to_bag_inf(i, ω, k+1)
```

```
obs_fn_inf (generic function with 1 method)
```

```julia
obs_fn_inf(data) =
    Variable(ω -> all(map(i -> make_bag_inf(obs_to_bag_inf(i, ω), ω) == data[i],
    1:length(data))))
```

```
[(obs1 = 2, obs2 = 3, obs3 = 2), (obs1 = 7, obs2 = 1, obs3 = 4), (obs1 = 2, obs2 = 1, obs3 = 2),
```

```julia
begin
    obs123_inf(ω::Ω) = (obs1=(1~obs_to_bag_inf)(ω), obs2=(2~obs_to_bag_inf)(ω), obs3=
    (3~obs_to_bag_inf)(ω))
    samples123_inf = randsample(obs123_inf |ᶜ obs_fn_inf(obs_marbles), 100)
end
```

```
BitVector: [false, false, false, false, false, false, false, false, false, false, false, fals
```

```julia
same_bag_12_inf::Vector{Bool} =
    getfield.(samples123_inf, :obs1) .== getfield.(samples123_inf, :obs2)
```

```
BitVector: [true, false, true, false, true, true, true, false, true, false, true, true, false
```

```julia
same_bag_13_inf::Vector{Bool} =
    getfield.(samples123_inf, :obs1) .== getfield.(samples123_inf, :obs3)
```

```
        ┌                                        ┐
  false ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 100
        └                                        ┘
```

```julia
viz(same_bag_12_inf)
```

```
        ┌                                        ┐
  false ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■ 42
   true ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 58
        └                                        ┘
```

```julia
viz(same_bag_13_inf)
```

Like the unbounded mixture above, there are an infinite set of possible catgories (here, bags). Unlike the unbounded mixture model the number of bags is never explicitly constructed. Instead, the set of categories is thought of as an infinite set; because they are constructed as needed only a finite number will ever be explicitly constructed. (Technically, these models are called infinite because the expected number of categories used goes to infinity as the number of observations goes to infinity.)