Hierarchical models

Exercise 1: Pseudocounts and the Dirichlet distribution

In the Bayesian Data Analysis exercises, we explored the Beta distribution by varying its parameters. The Dirichlet is a generalization of the Beta distribution to more than two categories (see Appendix (https://probmods.org/chapters/appendix-useful-distributions.html)). Instead of Beta parameters (a, b) governing the probabilities of two categories $(false / true)$(false/true), the Dirichlet parameter $\alpha = [\alpha_1, \alpha_2, ..., \alpha_n]$ α=[α1,α2,...,αn] controls the probabilities over categories $[A_1, A_2, ..., A_n]$[A1,A2,...,An]. In other words, different choices of αα correspond to different ways of distributing the prior probability mass over the $N - 1$N−1 simplex.

In this exercise, we will explore a particularly intuitive way of understanding the $alpha$ parameter as pseudocounts, or virtual observations. That is, if $\alpha = [2, 2, 1]$α=[2,2,1], that is the equivalent of having already observed the first category and second category twice each, and the third category one time only.

这个和我原来学6.431x的感觉是一致的，对于一些prior，它们表达的就是“预先观测到了一个值”

Complete the code below to prove that setting $\alpha = [2, 3, 1, 1, 1]$α=[2,3,1,1,1] is equivalent to setting $\alpha = [1, 1, 1, 1, 1]$α=[1,1,1,1,1] and then observing the first category once and the second category twice:

```
x
```

```
var colors = ['black', 'blue', 'green', 'orange', 'red'];
```

```

```

```
var observedData = [
```

```
{bag: 'bag1', draw: 'blue'},
```

```
{bag: 'bag1', draw: 'blue'},
```

```
{bag: 'bag1', draw: 'black'}]
```

```

```

```
// first model: set alpha = [1, 1, 1, 1, 1] and observe `observedData`

var observed = Infer({method: 'MCMC', samples: 20000}, function(){

  var makeBag = mem(function(bag){

    var colorProbs = dirichlet(ones([colors.length, 1]))

    return Categorical({vs: colors, ps: colorProbs})

  })



  var obsFn = function(datum){

    observe(makeBag(datum.bag), datum.draw)

  }



  mapData({data: observedData}, obsFn)



  return {bag1: sample(makeBag('bag1'))}

})



viz.marginals(observed)



// second model. Set alpha = [2, 3, 1, 1, 1]

var usealpha = Infer(



```

```
    // your code here
```

```

```

```
)
```

```

```

```
viz.marginals(usealpha) // should roughly match first figure
```

run ▼

Exercise 2: Rotten apples

On any given day, a given grocery store has some number of apples for sale. Some of these apples may be mushy or even rotten. The probability that each apple is rotten is not independent: a ripening fruit emits chemicals that encourages other fruit to ripen as well. As they say, one rotten apple spoils the whole barrel (https://idiomation.wordpress.com/2013/03/27/one-bad-apple-spoils-the-whole-barrel/).

For each apple in a barrel, assume the probability that the apple is rotten is `flip(p)` where `p` is drawn from some prior. An appropriate prior distribution is Beta. Recall that the Beta distribution is just a Dirichlet that returns a vector of length one. So it, too, is defined based on pseudocounts `[a, b]`. `Beta({a: 10, b: 2})` returns the equivalent of a Beta distribution conditioned on having previously seen 10 heads and 2 tails, while `[a,b]` values less than 1 concentrate mass at the endpoints. A `Beta({a: .1, b: .2})` prior nicely captures our expectations about rotten apples: most of the time, the probability of a rotten apple is quite low. The rest of the time, the probability is very high. Middling probabilities are rare.

a)

Write a function `makeBarrel` that returns a function (a 'barrel') that takes a single argument *N* and returns an array representing the rottenness of *N* apples from that barrel (where `true` is rotten and `false` is not rotten). That is, the following code:

```
var abarrel = makeBarrel('b')
abarrel(5)
```

should return something like `[true, true, true, false, true]`.

Complete the following codebox:

```

```

```
// your code here
```

```

```

```
// Do not edit this function: it tests your code
```

```
var post = Infer({method: 'forward'}, function(){

  var abarrel = makeBarrel('b')

  return Math.sum(abarrel(10))

})

viz(post)
```

run ▼

b)

Some grocery stores have fresher produce than others. So let's create a function `makeStore` that returns a `makeBarrel` function, which works as it did in part (a). Importantly, each store has its own Beta parameters `[a, b]` drawn from some prior.

HINT: In order to maintain the likelihood that in a given barrel, either most of the apples are rotten or few are, you need to ensure that `a < 1` and `b < 1`. However, if `a` is much larger than `b` (or vice versa), you will get extreme results with *every* apple being rotten or *every* apple being good.

NOTE: No need to be overly fancy with this prior. Pick something simple that you know will give you what you want: stores that tend to have bad barrels and stores that tend to have good barrels.

```
var makeStore = // your code here



// Following code inspects your functions

viz(Infer({method: 'forward', samples:10000}, function(){

  var S = makeStore('S')

  var B1 = S('B1')

  var B2 = S('B2')

  return Math.abs(Math.sum(B1(10))-Math.sum(B2(10)))

})) // should generally be little difference
```

```
viz(Infer({method: 'forward', samples:10000}, function(){
  var S1 = makeStore('S1')
  var S2 = makeStore('S2')
  var B1 = S1('B1')
  var B2 = S2('B2')
  return Math.abs(Math.sum(B1(10))-Math.sum(B2(10)))
})) // difference should be larger on average
```

run ▼

c)

We can keep going. Some cities are located in apple country and thus have more access to fresh apples. Most stores in those cities are going to mostly have good barrels with good apples. Other cities have less access to fresh apples, and so more of their stores will have bad barrels with rotten apples.

In the code block below, create a `makeCity` function, which returns a `makeStore` function, which works as in (b). In (b), each store had a prior on `[a, b]`. Let's put a prior on *that* prior, such that cities either tend to have good stores or tend to have bad stores.

HINT: Again, it is not necesary to have an overly fancy prior here. If you are spending hours trying to find just the right prior distribution, you are over-thinking it.

```
var makeCity = // your code here
```

```
//Make sure the following code runs:
var C1 = makeCity("C1")
var S1 = C1("S1")
var B1 = S1("B1")
```

```
viz(Infer({method: 'forward'}, function(){
    return Math.sum(B1(10))
}))
//repeat to see different kinds of cities
```

run ▼

d)

Suppose you go to a store in a city. The store has a barrel of 10 apples, 7 of which are rotten. You leave and go to another store in the same city. It also has has a barrel with 10 apples. Using your code above, how many of these apples are likely to be rotten?

```
// your code here
```

run ▼

Exercise 3: Hierarchical models for BDA

Imagine that you have done an experiment on word reading times to test the hypothesis that words starting with vowels take longer to read. Each data point includes which condition the word is from ("vowel" vs. "consonant"), the word itself, the participant id, and the response time you measured ("rt"). A simple data anlysis model attempts to infer the mean reading time for each word group, and returns the difference between the groups (a sort of Bayesian version of a t-test). Note that there is no cognitive model inside this BDA; it is directly modeling the data.

```
var data = [{group: "vowel", word: "abacus", id: 1, rt: 210},
            {group: "vowel", word: "abacus", id: 2, rt: 212},
            {group: "vowel", word: "abacus", id: 3, rt: 209},
            {group: "vowel", word: "aardvark", id: 1, rt: 200},
            {group: "vowel", word: "aardvark", id: 2, rt: 201},
            {group: "vowel", word: "aardvark", id: 3, rt: 198},
```

```
            {group: "vowel", word: "ellipse", id: 1, rt: 220},
            {group: "vowel", word: "ellipse", id: 2, rt: 222},
            {group: "vowel", word: "ellipse", id: 3, rt: 219},

            {group: "consonant", word: "proton", id: 1, rt: 190},
            {group: "consonant", word: "proton", id: 2, rt: 191},
            {group: "consonant", word: "proton", id: 3, rt: 189},
            {group: "consonant", word: "folder", id: 1, rt: 180},
            {group: "consonant", word: "folder", id: 2, rt: 182},
            {group: "consonant", word: "folder", id: 3, rt: 178},
            {group: "consonant", word: "fedora", id: 1, rt: 230},
            {group: "consonant", word: "fedora", id: 2, rt: 231},
            {group: "consonant", word: "fedora", id: 3, rt: 228},
            {group: "consonant", word: "fedora", id: 1, rt: 231},
            {group: "consonant", word: "fedora", id: 2, rt: 233},
            {group: "consonant", word: "fedora", id: 3, rt: 230},
            {group: "consonant", word: "fedora", id: 1, rt: 230},
            {group: "consonant", word: "fedora", id: 2, rt: 232},
            {group: "consonant", word: "fedora", id: 3, rt: 228}]

var post = Infer({method: "MCMC",  kernel: {HMC: {steps: 10, stepSize: 1}}, samples: 1000}, fu
```

```
  var groupMeans = {vowel: gaussian(200, 100), consonant: gaussian(200, 100)}


  var obsFn = function(d){
    //assume response times (rt) depend on group means with a small fixed noise:
    observe(Gaussian({mu: groupMeans[d.group], sigma: 10}), d.rt)
  }

  mapData({data: data}, obsFn)

  //explore the difference in means:
  return groupMeans['vowel']-groupMeans['consonant']
})

print("vowel - consonant reading time:")
viz(post)
print(expectation(post))
```

run ▼

As you see, this model concludes that consonants actually take significantly longer to read! But if you look at the data carefully you may not trust this conclusion: it seems to be driven by a single outlier, the word “fedora”!

a)

Adjust the model to allow each word to have its own mean reading time, that depends on the `groupMean` but needn’t be the same. This is called a hierachical data analysis model.

```
var data = [{group: "vowel", word: "abacus", id: 1, rt: 210},
```

```
        {group: "vowel", word: "abacus", id: 2, rt: 212},
        {group: "vowel", word: "abacus", id: 3, rt: 209},
        {group: "vowel", word: "aardvark", id: 1, rt: 200},
        {group: "vowel", word: "aardvark", id: 2, rt: 201},
        {group: "vowel", word: "aardvark", id: 3, rt: 198},
        {group: "vowel", word: "ellipse", id: 1, rt: 220},
        {group: "vowel", word: "ellipse", id: 2, rt: 222},
        {group: "vowel", word: "ellipse", id: 3, rt: 219},

        {group: "consonant", word: "proton", id: 1, rt: 190},
        {group: "consonant", word: "proton", id: 2, rt: 191},
        {group: "consonant", word: "proton", id: 3, rt: 189},
        {group: "consonant", word: "folder", id: 1, rt: 180},
        {group: "consonant", word: "folder", id: 2, rt: 182},
        {group: "consonant", word: "folder", id: 3, rt: 178},
        {group: "consonant", word: "fedora", id: 1, rt: 230},
        {group: "consonant", word: "fedora", id: 2, rt: 231},
        {group: "consonant", word: "fedora", id: 3, rt: 228},
        {group: "consonant", word: "fedora", id: 1, rt: 231},
        {group: "consonant", word: "fedora", id: 2, rt: 233},
        {group: "consonant", word: "fedora", id: 3, rt: 230},
```

```
            {group: "consonant", word: "fedora", id: 1, rt: 230},
            {group: "consonant", word: "fedora", id: 2, rt: 232},
            {group: "consonant", word: "fedora", id: 3, rt: 228}]

var post = Infer({method: "MCMC",  kernel: {HMC: {steps: 10, stepSize: 1}}, samples: 1000}, fu
  var groupMeans = {vowel: gaussian(200, 100), consonant: gaussian(200, 100)}

  //...your code here

  var obsFn = function(d){
    //assume response times (rt) depend on group means with a small fixed noise:
    observe(Gaussian({mu: //..your code here ,
      sigma: 10}), d.rt)
  }

  mapData({data: data}, obsFn)

  //explore the difference in means:
  return groupMeans['vowel']-groupMeans['consonant']
})
```

```
print("vowel - consonant reading time:")
```

```
viz(post)
```

```
print(expectation(post))
```

run ▼

What do you conclude about vowel words vs. consonant words now?

Hint Consider how the model is sensitive to the different assumed variances (e.g. the fixed noise in the observe function we assume sigma=10). In particular, think about how this should affect how you choose a sigma for your word-level effects.

The individual word means that you have introduced are called *random effects* – in a BDA they are random variables (usually at the individual item or person level) that are not of interest by themselves.

b)

If you stare at the data further, you might notice that some of the participants in your experiment read slightly faster overall. Extend your model to include an additional random effect of participant id, that is, an unknown (and not of interest) influence on reading time of the particular person. Does this make your conclusion different? Stronger or weaker?