‘Assignment 11’

There is nothing here to hand in. These questions are only for your practice, and come with my solutions.

1. A binomial experiment with 8 trials produces the following results: success, failure, success, success, failure, success, success, success. (Each result is therefore a Bernoulli trial.) The person who gave you the data says that the success probability is most likely somewhere near 0.5, but might be near 0 or 1. The aim of this question is to estimate the success probability using Bayesian methods.

   In this question, use `rstan` as in class, or alternatively use `cmdstanr` (see this site for instructions). If you have trouble with installing `rstan`, try `cmdstanr`. I will do it both ways. Documentation for Stan, either way, is here. You will probably want to be running R on your own computer.

   (a) Write a Stan program that will estimate the success probability p. To do this, start with the likelihood (Stan has a function `bernoulli` that takes one parameter, the success probability). The data, as 1s and 0s, will be in a vector `x`. Use a beta distribution with unknown parameters as a prior for `p`. (We will worry later what those parameters should be.)

Solution:

File, New and Stan. Leave the template program there if you like, as a reminder of what to do. In the `model` section is where the likelihood goes, like this:[1]

```
model {
  // likelihood
  x ~ bernoulli(p);
}
```

The right one here is `bernoulli` since your data are Bernoulli trials (successes and failures, coded as 1s and 0s). If you had a summarized total number of successes and a number of trials, then that would be binomial. It actually doesn’t make any difference which way you do it, but it’s probably easier to think about it this way because it’s more like the Poisson one in lecture.

Thinking ahead, `x` is going to be data, and `p` is a parameter, so `p` will need a prior distribution. The standard one for a Bernoulli success probability is a beta distribution. This is actually the conjugate prior, if you have learned about those: if `p` has a beta prior and the likelihood is Bernoulli, then the posterior is also beta. Back in the days when algebra was your only hope for this kind of thing, conjugate priors were very helpful, but now that we can sample from any posterior, the fact that a prior is conjugate is neither here nor there. Having said that, the beta distribution is a nice choice for a prior for this, because it is restricted to $[0, 1]$ the same way that a Bernoulli `p` is.

I’m going leave the prior parameters for `p` unknown for now; we’ll just call them `a` and `b`.[2] Here’s our completed `model` section:

```
model {
  // prior
  p ~ beta(a, b);
  // likelihood
  x ~ bernoulli(p);
}
```

a and b are not parameters; they are some numbers that we will supply, so they will be part of the data section. Leaving them unspecified like this, rather than hard-coding them, is good coding practice, since the code we finish with can be used for any Bernoulli estimation problem, not just the one we happen to have.

There is only one parameter, p, so the parameters section is short:

```
parameters {
  real<lower=0,upper=1> p;
}
```

We know that p must be between 0 and 1, so we specify that here so that the sampler doesn't stray into impossible values for p.

That goes before the model section. Everything else is data. We also want to avoid hard-coding the number of observations, so we will also have an n as data, which we declare first, so we can declare the array of values x to be of length n:

```
data {
  int<lower=0> n;
  real a;
  real b;
  int<lower=0, upper=1> x[n];
}
```

x is an integer array of length n. This is how you declare one of those: the type is first, along with any limits, and then the length of the array is appended in square brackets to the name of the array.

Arrange your code in a file with extension .stan, with data first, parameters second, and model third. I called mine bernoulli.stan.

Extra: there are two ways to declare a *real*-valued array y: as real y[n], or as vector[n] y. Sometimes it matters which way you do it (and I don't have a clear sense of when it matters). The two ways differ in what you can do with them.

(b) Compile your code, correcting any errors until it compiles properly.

Solution:

If you are going with rstan, that looks like this:

```
m <- stan_model(file = "bernoulli.stan")
```

```
m
```

```
## S4 class stanmodel 'bernoulli' coded as follows:
## data {
##   int<lower=0> n;
##   real a;
##   real b;
##   int<lower=0, upper=1> x[n];
## }
##
## parameters {
##   real<lower=0,upper=1> p;
## }
##
## model {
##   // prior
##   p ~ beta(a, b);
##   // likelihood
##   x ~ bernoulli(p);
## }
##
```

If you are using `conflicted`, you may have a number of things to resolve. The only new one is that we will be using the `rstan` `extract` later, rather than the `tidyr` one that we have used before.

This one gave me a warning when I compiled it, but displaying the compiled model looked as I expected, so I am happy.

The other way to go is to use `cmdstanr`, like this:

```
m2 <- cmdstan_model("bernoulli.stan")
```

```
m2
```

```
## data {
##   int<lower=0> n;
##   real a;
##   real b;
##   int<lower=0, upper=1> x[n];
## }
##
## parameters {
##   real<lower=0,upper=1> p;
## }
##
## model {
##   // prior
##   p ~ beta(a, b);
##   // likelihood
##   x ~ bernoulli(p);
## }
```

If it doesn't compile, you have some fixing up to do. The likely first problem is that you have missed a semicolon somewhere. The error message will at least give you a hint about where the problem is. Fix any errors you see and try again. If you end up with a different error message, that at least is progress.

(c) The person who brought you the data told you that the success probability `p` should be somewhere near 0.5 (and is less likely to be close to 0 or 1). Use this information to pick a prior distribution for `p`. (The exact answer you get doesn't really matter, but try to interpret the statement in some kind of sensible way.)

Solution:

I don't know how much intuition you have for what beta distributions look like, so let's play around a bit. Let's imagine we have a random variable Y that has a beta distribution. This distribution has two parameters, usually called a and b. Let's draw some pictures and see if we can find something that would serve as a prior. R has `dbeta` that is the beta distribution density function.

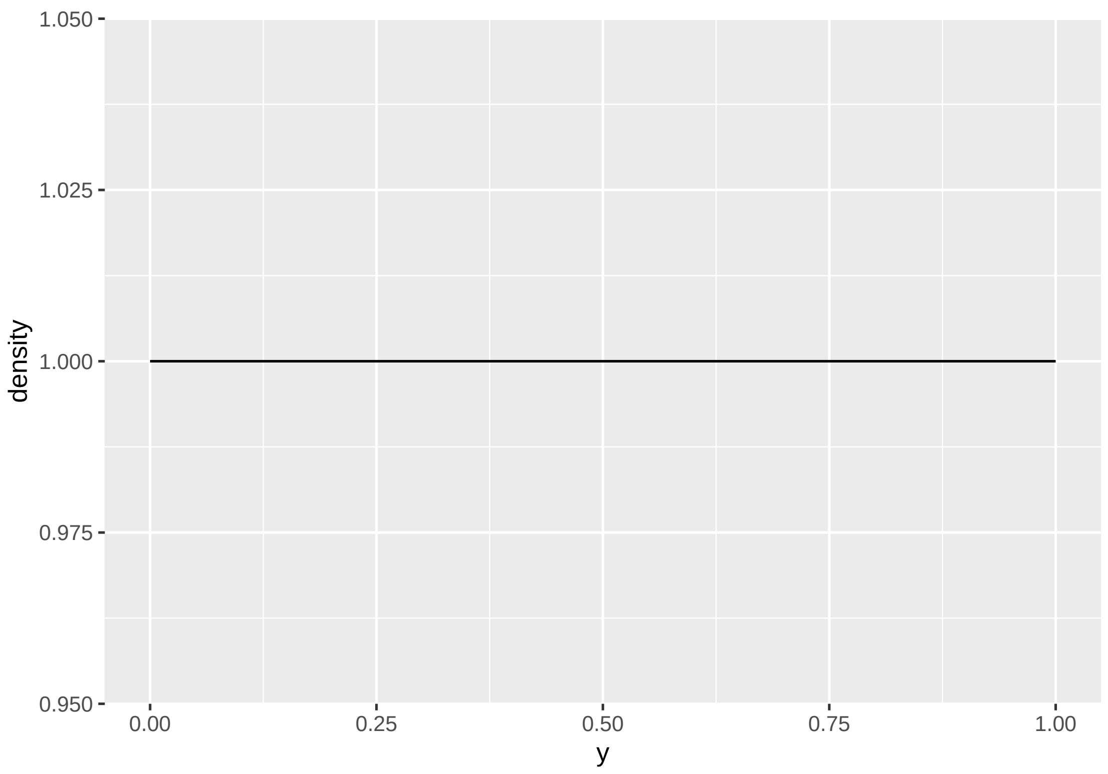
Start by choosing some values for Y:

```r
y <- seq(0, 1, 0.01)
y
```

```
##   [1] 0.00 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14
##  [16] 0.15 0.16 0.17 0.18 0.19 0.20 0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29
##  [31] 0.30 0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.40 0.41 0.42 0.43 0.44
##  [46] 0.45 0.46 0.47 0.48 0.49 0.50 0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59
##  [61] 0.60 0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69 0.70 0.71 0.72 0.73 0.74
##  [76] 0.75 0.76 0.77 0.78 0.79 0.80 0.81 0.82 0.83 0.84 0.85 0.86 0.87 0.88 0.89
##  [91] 0.90 0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99 1.00
```

then work out `dbeta` of these for your choice of parameters, then plot it. I'm going straight to a function for this, since I anticipate doing it several times. This `y` and the two parameters should be input to the function:

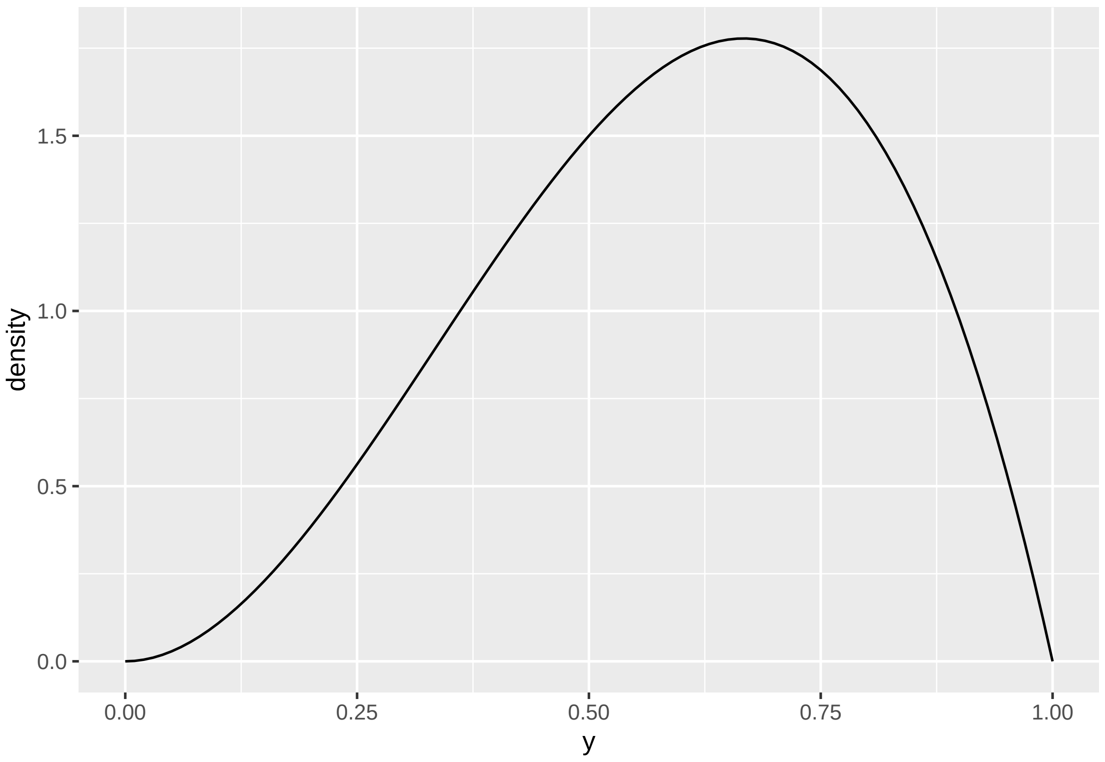
```r
plot_beta <- function(y, a, b) {
  tibble(y=y) %>%
    mutate(density = dbeta(y, a, b)) %>%
    ggplot(aes(x = y, y = density)) + geom_line()
}
plot_beta(y, 1, 1)
```



The beta with parameters 1 and 1 is a uniform distribution. (If you look up the beta density function, you'll see why that is.)

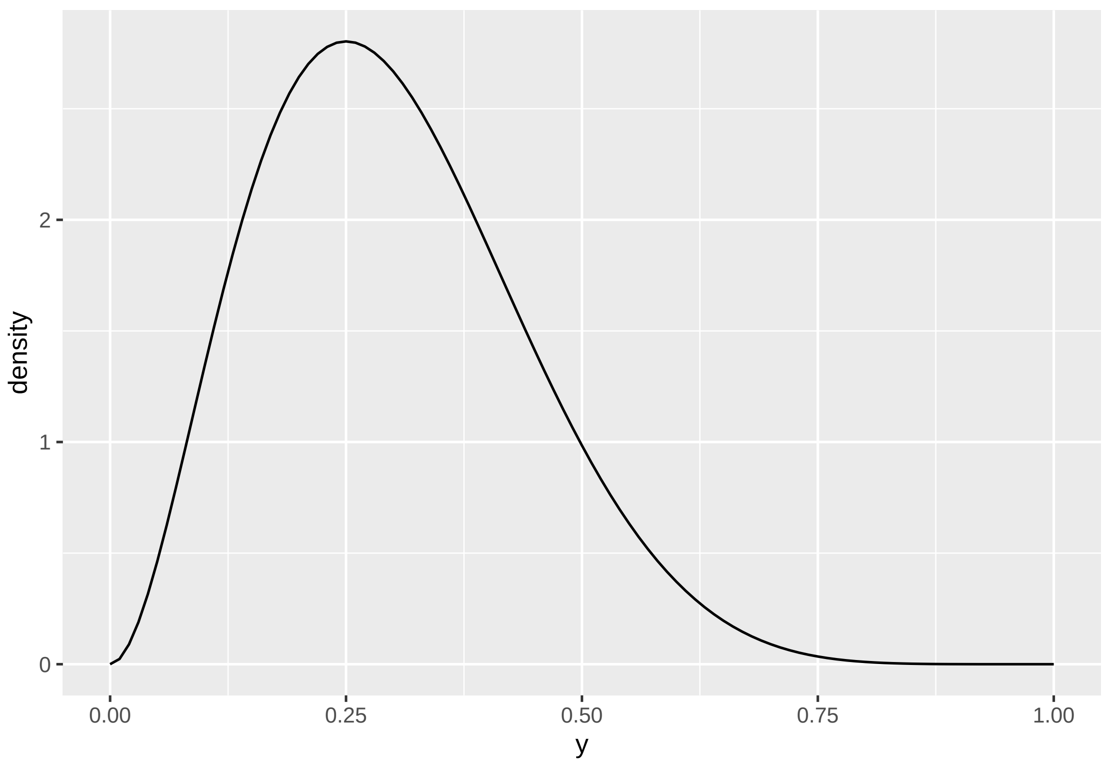
Let's try another:

```
plot_beta(y, 3, 2)
```



This one is skewed to the left. You might guess that having the second parameter bigger would make it skewed to the right:

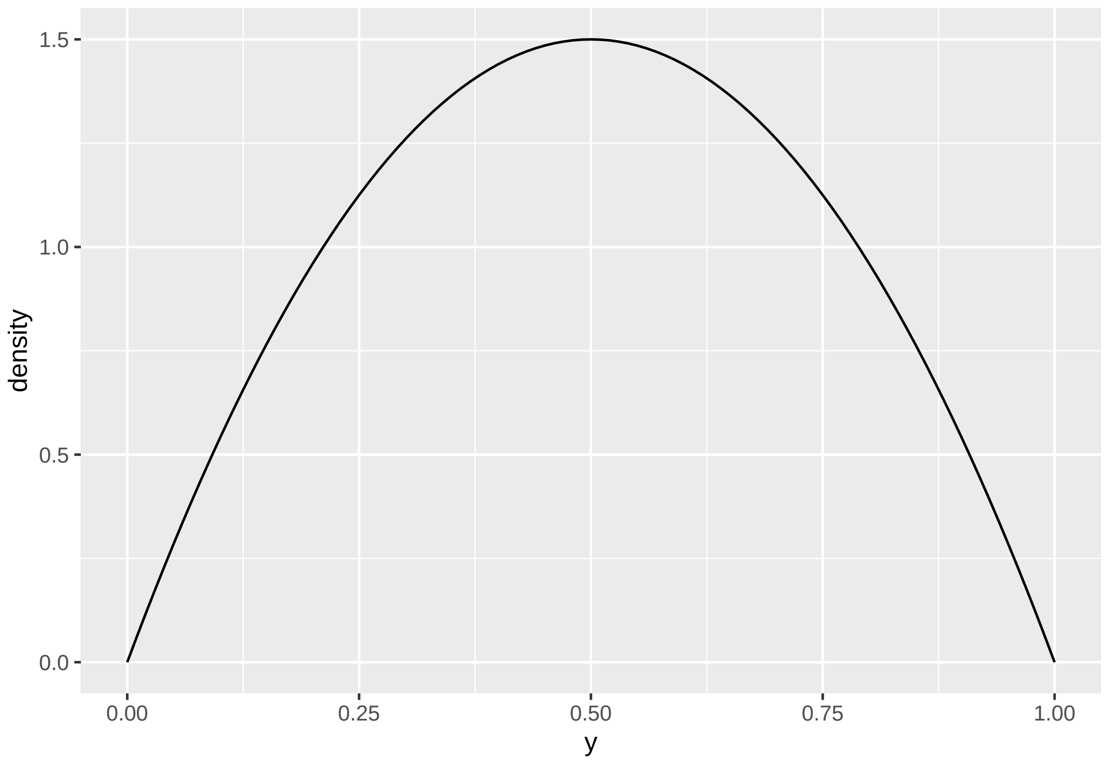
```
plot_beta(y, 3, 7)
```



which indeed is the case. If you try some other values, you'll see that this pattern with the skewness continues to hold. Furthermore, the right-skewed distributions have their peak to the *left* of 0.5, and the left-skewed ones have their peak to the *right* of 0.5.

Therefore, you would think, having the two parameters the same would give a symmetric distribution:

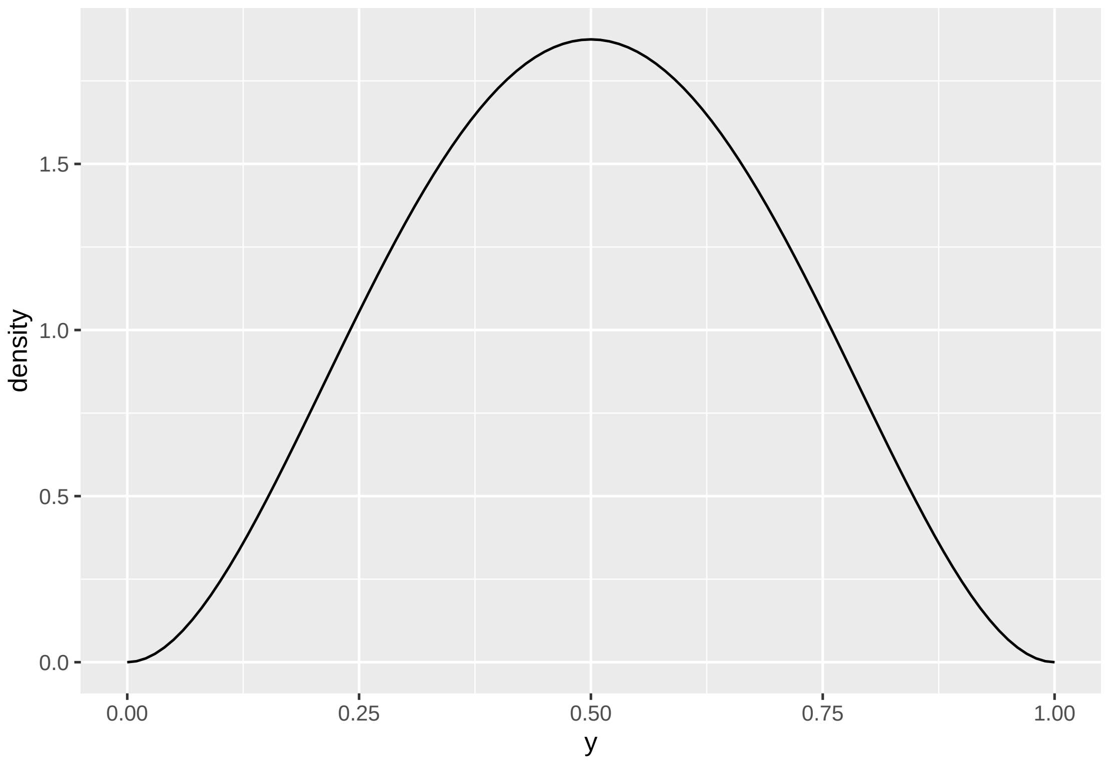
```
plot_beta(y, 2, 2)
```



Note that the peak is now at 0.5, which is what we wanted.

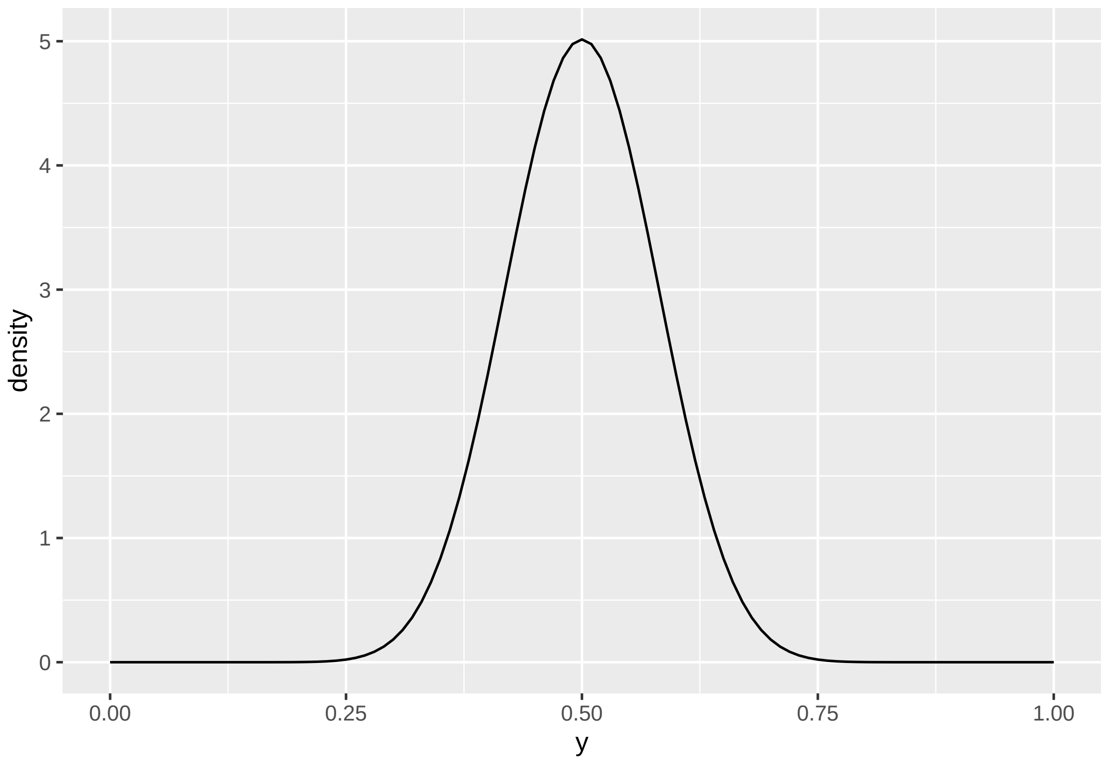
The question called for a prior distribution of values "somewhere near 0.5", and you could reasonably say that this does the job. What does 3 and 3 look like?

```
plot_beta(y, 3, 3)
```



This is more concentrated around 0.5, and as you increase the two parameter values while keeping them equal, it gets more concentrated still:

```
plot_beta(y, 20, 20)
```



For our purposes, this is undoubtedly too concentrated around 0.5; there is no chance of y being outside $[0.25, 0.75]$. I would go with parameters 2 and 2 or maybe 3 and 3. As I said, pretty much any choice of parameter values that are both the same is at least somewhat justifiable.

If you don't want to go through all of this, find some pictures of beta distributions with different parameters, and pick one you like. The Wikipedia page is one place (from which you would probably pick 2 and 2). Here is another, from which you might pick 5 and 5.

In practice, you would have some back-and-forth with the person who brought you the data, and try to match what they are willing to say about `p` to what you know or can find out about the beta distribution. This process is called "prior elicitation".

Extra: if you have obtained the posterior distribution in this case by algebra, you might recall that the effect of the prior distribution is to add some "fake" Bernoulli trials to the data. With `a = b = 2`, for example, you imagine four fake trials, with two successes and two failures, to add to the data. This brings the estimate of `p` closer to 0.5 than it would be with just plain maximum likelihood.

(d) Create an R `list` that contains all your `data` for your Stan model. Remember that Stan expects the data in `x` to be 0s and 1s.

Solution:

Turn those successes and failures in the question into a vector of 0 and 1 values: they were success, failure, success, success, failure, success, success, success.

```
x <- c(1, 0, 1, 1, 0, 1, 1, 1)
x
```

```
## [1] 1 0 1 1 0 1 1 1
```

Then make a "named list" of inputs to your Stan program, including the parameter values for the prior distribution (I went with 2 and 2):

```
stan_data <- list(
  n = 8,
  a = 2,
  b = 2,
  x = x
)
stan_data
```

```
## $n
## [1] 8
##
## $a
## [1] 2
##
## $b
## [1] 2
##
## $x
## [1] 1 0 1 1 0 1 1 1
```

(e) Run your Stan model to obtain a simulated posterior distribution, using all the other defaults.

Solution:

This depends on whether you are using `rstan` or `cmdstanr`. With `rstan`:

```
fit <- sampling(m, data = stan_data)
```

```
##
## SAMPLING FOR MODEL 'bernoulli' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 5e-06 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.05 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 1: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 1: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 1: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 1: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 1: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 1: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.005712 seconds (Warm-up)
## Chain 1:                0.005466 seconds (Sampling)
## Chain 1:                0.011178 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'bernoulli' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 3e-06 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.03 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 2: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 2: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 2: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 2: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 2: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 2: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 2: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 2: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 0.005528 seconds (Warm-up)
## Chain 2:                0.005415 seconds (Sampling)
## Chain 2:                0.010943 seconds (Total)
## Chain 2:
##
## SAMPLING FOR MODEL 'bernoulli' NOW (CHAIN 3).
## Chain 3:
## Chain 3: Gradient evaluation took 2e-06 seconds
## Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0.02 seconds.
## Chain 3: Adjust your expectations accordingly!
## Chain 3:
## Chain 3:
## Chain 3: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 3: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 3: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 3: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 3: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 3: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 3: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 3: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3:
```

```
## Chain 3:  Elapsed Time: 0.005574 seconds (Warm-up)
## Chain 3:                0.00597 seconds (Sampling)
## Chain 3:                0.011544 seconds (Total)
## Chain 3:
##
## SAMPLING FOR MODEL 'bernoulli' NOW (CHAIN 4).
## Chain 4:
## Chain 4: Gradient evaluation took 4e-06 seconds
## Chain 4: 1000 transitions using 10 leapfrog steps per transition would take 0.04 seconds.
## Chain 4: Adjust your expectations accordingly!
## Chain 4:
## Chain 4:
## Chain 4: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 4: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 4: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 4: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 4: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 4: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 4: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 4: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 4: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 4: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4:
## Chain 4:  Elapsed Time: 0.005629 seconds (Warm-up)
## Chain 4:                0.005424 seconds (Sampling)
## Chain 4:                0.011053 seconds (Total)
## Chain 4:
```

```
fit
```

```
## Inference for Stan model: bernoulli.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##       mean se_mean   sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
## p     0.66    0.00 0.13   0.38  0.58  0.67  0.76  0.90  1302    1
## lp__ -8.19    0.02 0.80 -10.45 -8.35 -7.88 -7.69 -7.64  1295    1
##
## Samples were drawn using NUTS(diag_e) at Thu Apr  1 23:22:47 2021.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

The posterior mean is 0.66, with a 95% posterior interval from 0.38 to 0.90: not very informative, since there was very little data.

The `cmdstanr` way is slightly different:

```
fit2 <- m2$sample(data = stan_data)
```

```
## Running MCMC with 4 sequential chains...
##
```

```
## Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 1 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 1 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 1 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 1 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 1 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 1 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 1 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 1 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 1 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 1 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 1 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 1 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 1 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 1 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 1 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1 finished in 0.0 seconds.
## Chain 2 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 2 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 2 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 2 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 2 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 2 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 2 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 2 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 2 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 2 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 2 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 2 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 2 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 2 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 2 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 2 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 2 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 2 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 2 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2 finished in 0.0 seconds.
## Chain 3 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 3 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 3 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 3 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 3 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 3 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 3 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 3 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 3 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 3 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 3 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 3 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 3 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 3 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 3 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 3 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 3 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 3 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3 finished in 0.0 seconds.
## Chain 4 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4 Iteration:  100 / 2000 [  5%]  (Warmup)
## Chain 4 Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 4 Iteration:  300 / 2000 [ 15%]  (Warmup)
## Chain 4 Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 4 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 4 Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 4 Iteration:  700 / 2000 [ 35%]  (Warmup)
## Chain 4 Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 4 Iteration:  900 / 2000 [ 45%]  (Warmup)
## Chain 4 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 4 Iteration: 1100 / 2000 [ 55%]  (Sampling)
## Chain 4 Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 4 Iteration: 1300 / 2000 [ 65%]  (Sampling)
## Chain 4 Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 4 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 4 Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 4 Iteration: 1700 / 2000 [ 85%]  (Sampling)
## Chain 4 Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 4 Iteration: 1900 / 2000 [ 95%]  (Sampling)
## Chain 4 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4 finished in 0.0 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 0.0 seconds.
## Total execution time: 0.6 seconds.
```

```
fit2
```

```
##  variable  mean median   sd  mad    q5   q95 rhat ess_bulk ess_tail
##      lp__ -8.15  -7.87 0.70 0.31 -9.58 -7.64 1.00     1899     2254
##      p     0.66   0.67 0.13 0.13  0.43  0.86 1.01     1487     1899
```

This one gives you a 90% posterior interval instead of a 95% one, but the posterior mean is 0.66, as before, and the interval once again says that `p` could be almost anything; the data did not narrow it down much.

(f) Make a plot of the posterior distribution of the probability of success.

Solution:

This means extracting the sampled values of p first. The `rstan` way is this:

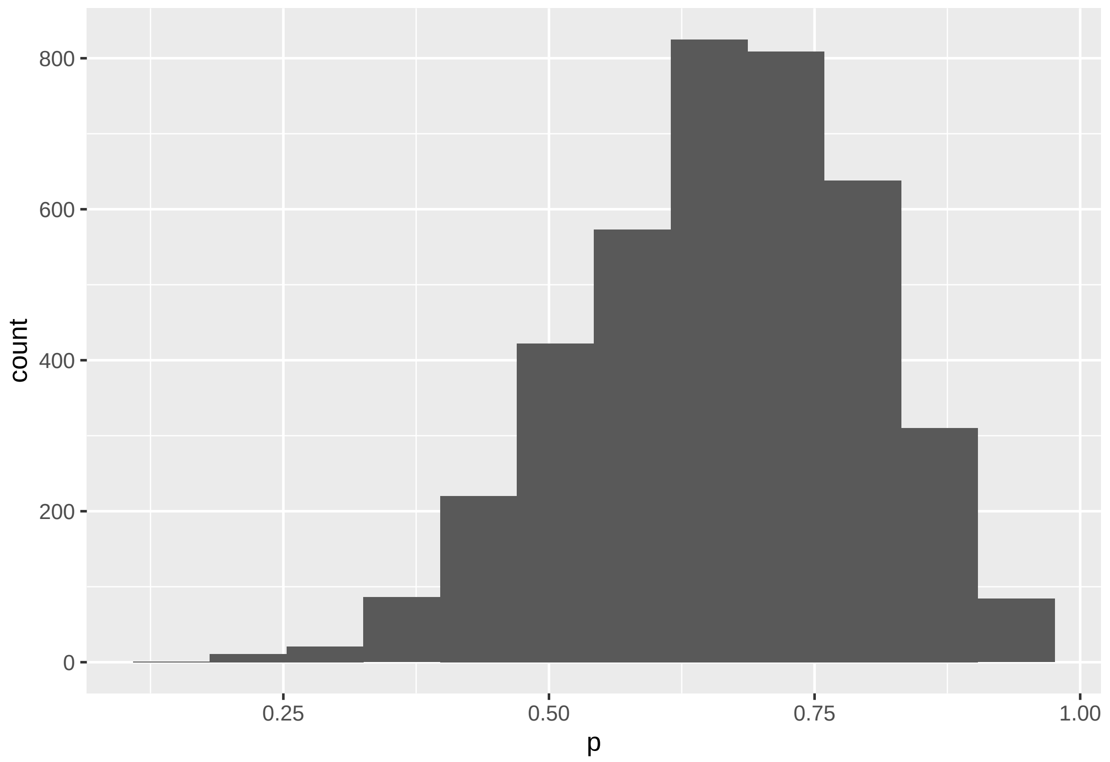
```
bern.1 <- extract(fit)
glimpse(bern.1)
```

```
## List of 2
##  $ p   : num [1:4000(1d)] 0.678 0.792 0.85 0.457 0.536 ...
##   ..- attr(*, "dimnames")=List of 1
##   .. ..$ iterations: NULL
##  $ lp__: num [1:4000(1d)] -7.64 -8.14 -8.88 -8.71 -8.06 ...
##   ..- attr(*, "dimnames")=List of 1
##   .. ..$ iterations: NULL
```

This contains a list of two things: 4000 values of `p`, and 4000 corresponding values of the log-posterior distribution, should you need them for anything (I usually don't).

The plot is then a histogram, as you would expect. With 4000 sampled values of p, you can use more bins than usual; Sturges' rule says 12, since $2^{12} = 4096$. More would also be fine, most likely:

```
tibble(p = bern.1$p) %>%
  ggplot(aes(x=p)) + geom_histogram(bins = 12)
```



This is skewed to the left. The reason for the skewness here is that the upper limit for p is 1, and there is a reasonable chance of p being close to 1, so the distribution is skewed in the opposite direction. There is basically no chance of p being close to zero. If we had had more data, it is more likely that the values of p near 0 and 1 would be ruled out, and then we might have ended up with something more symmetric.

The `cmdstanr` way is a bit less convenient, at least at first:

```
bern.2a <- fit2$draws()
str(bern.2a)
```

```
##  'draws_array' num [1:1000, 1:4, 1:2] -7.65 -7.72 -8.03 -7.79 -7.91 ...
##  - attr(*, "dimnames")=List of 3
##   ..$ iteration: chr [1:1000] "1" "2" "3" "4" ...
##   ..$ chain    : chr [1:4] "1" "2" "3" "4"
##   ..$ variable : chr [1:2] "lp__" "p"
```

This is a 3-dimensional array (sample by chain by variable). For plotting and so on, we really want this as a dataframe. At this point, I would use the `posterior` and `bayesplot` packages, which you should install following the instructions for `cmdstanr` at the top of this page. Put the names of the extra two packages in place of the `cmdstanr` that you see there.

```
library(posterior)
```

```
## This is posterior version 0.1.4
```

```
library(bayesplot)
```

```
## This is bayesplot version 1.8.0
## - Online documentation and vignettes at mc-stan.org/bayesplot
```

```
## - bayesplot theme set to bayesplot::theme_default()
```

```
##    * Does _not_ affect other ggplot2 plots
```

```
##    * See ?bayesplot_theme_set for details on theme setting
```

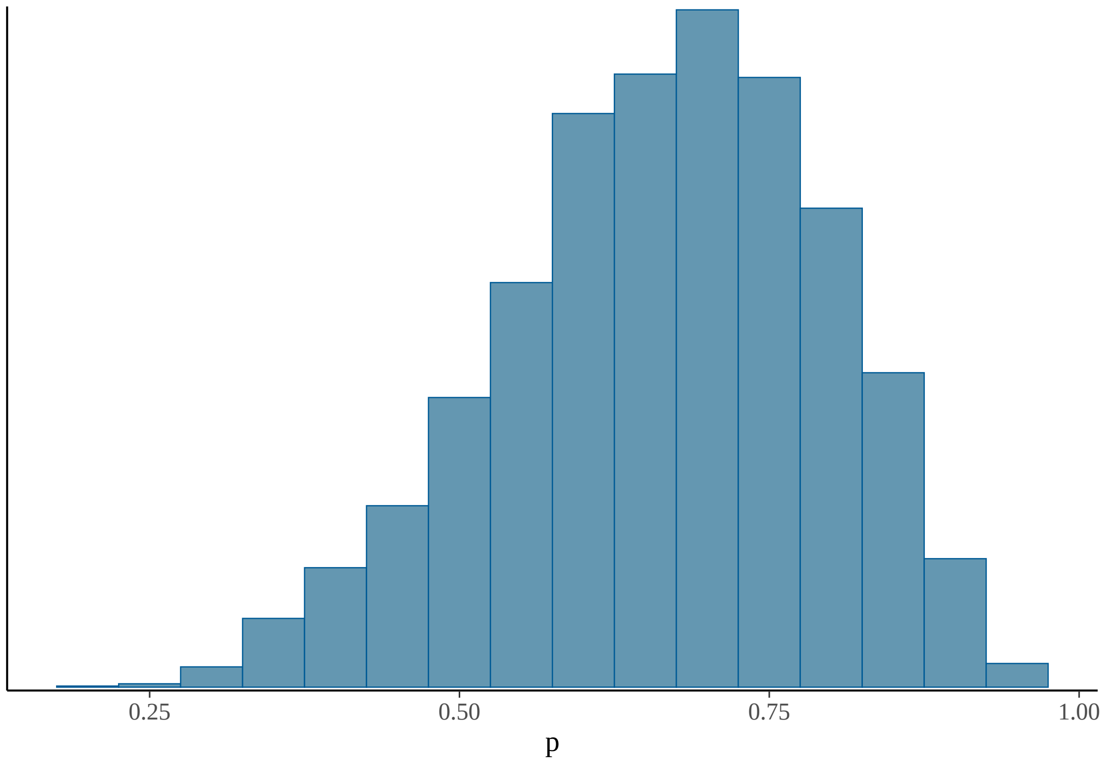
To get the samples as a dataframe:

```r
bern.2 <- as_draws_df(fit2$draws())
bern.2
```

```
## # A draws_df: 1000 iterations, 4 chains, and 2 variables
##      lp__    p
## 1    -7.6 0.68
## 2    -7.7 0.61
## 3    -8.0 0.54
## 4    -7.8 0.74
## 5    -7.9 0.76
## 6   -10.0 0.36
## 7    -8.5 0.83
## 8    -8.1 0.79
## 9    -7.9 0.75
## 10   -8.3 0.50
## # ... with 3990 more draws
## # ... hidden reserved variables {'.chain', '.iteration', '.draw'}
```

You don't even need to go this far to make a plot of the posterior distribution, because `bayesplot` does it automatically:

```r
mcmc_hist(fit2$draws("p"), binwidth =  0.05)
```



Rather annoyingly, this plot function passes `binwidth` on to `geom_histogram`, but not `bins`! The picture, though, is very similar to what we got from `fit` (out of `rstan`).

(g) The posterior predictive distribution is rather odd here: the only possible values that can be observed are 0 and 1. Nonetheless, obtain the posterior predictive distribution for these data, and explain briefly why it is not surprising that it came out as it did.

Solution:

The way to obtain the (sampled) posterior predictive distribution is to get the posterior distribution of values of p in a dataframe, and make a new column as random values from the data-generating mechanism (here Bernoulli). This is easier to do and then talk about:

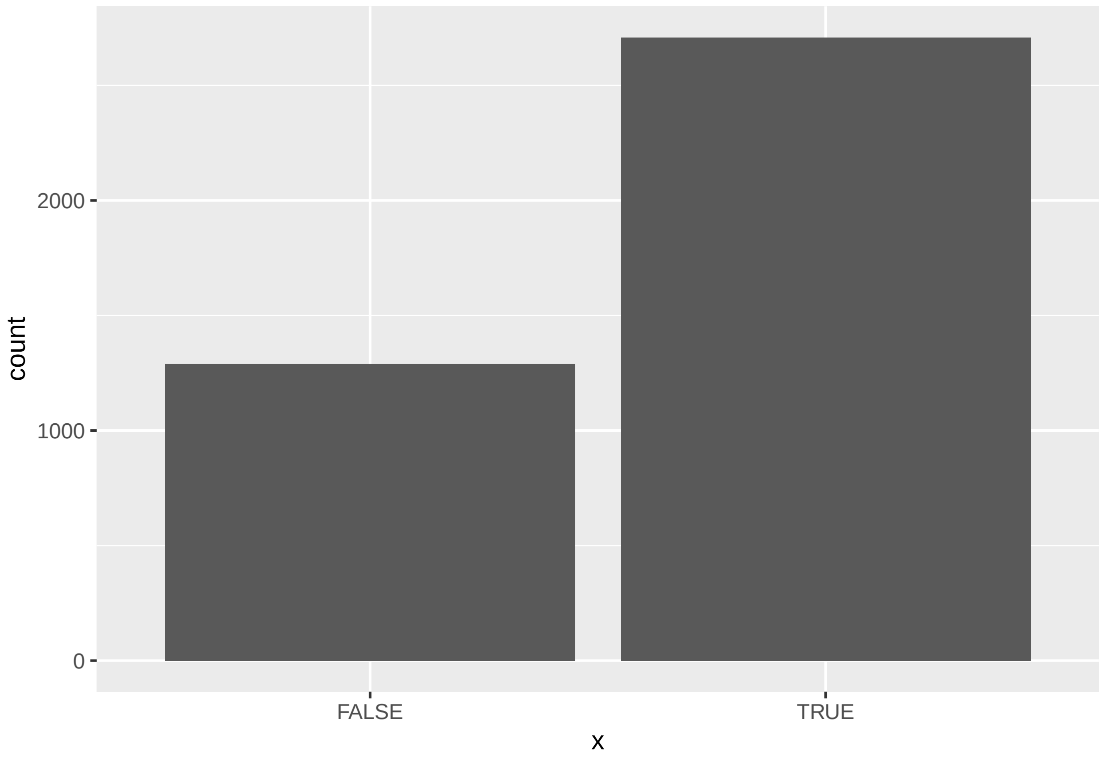
```
bern.1$p %>% enframe(value = "p") %>%
  mutate(x = rbernoulli(4000, p)) -> ppd
ppd
```

```
## # A tibble: 4,000 x 3
##     name     p x
##    <int> <dbl> <lgl>
##  1     1 0.678 TRUE
##  2     2 0.792 TRUE
##  3     3 0.850 TRUE
##  4     4 0.457 FALSE
##  5     5 0.536 TRUE
##  6     6 0.574 TRUE
##  7     7 0.641 TRUE
##  8      8 0.705 TRUE
##  9      9 0.596 TRUE
## 10     10 0.790 FALSE
## # ... with 3,990 more rows
```

The values of `x` in the last column are TRUE for success and FALSE for failure (they could have been 1 and 0). Thus, the first `x` is a Bernoulli trial with success probability the first value of `p`, the second one uses the second value of `p`, and so on. Most of the success probabilities are bigger than 0.5, so most of the posterior predictive distribution is successes.

A bar chart would be an appropriate plot (you can either think of `x` as categorical, or as a discrete 0 or 1):

```
ggplot(ppd, aes(x=x)) + geom_bar()
```



which shows the majority of successes in the posterior predictive distribution. The idea is that the data and the posterior predictive distribution ought to be similar, and we did indeed have a majority of successes in our data as well.

You might have been perplexed by the 4000 in the code above. `bernoulli` is vectorized, meaning that if you give it a vector of values for `p`, it will generate Bernoulli trials for each one in turn, and the whole result should be 4000 values long altogether. We'll see a way around that in a moment, but you could also do this using `rbinom` (random binomials) if you do it right:

```
bern.1$p %>% enframe(value = "p") %>%
  mutate(x = rbinom(4000, 1, p))
```

```
## # A tibble: 4,000 x 3
##     name     p     x
##    <int> <dbl> <int>
##  1     1 0.678     1
##  2     2 0.792     1
##  3     3 0.850     1
##  4     4 0.457     0
##  5     5 0.536     0
##  6     6 0.574     0
##  7     7 0.641     1
##  8     8 0.705     1
##  9     9 0.596     0
## 10    10 0.790     0
## # ... with 3,990 more rows
```

There are 4000 random binomials altogether, and *each one* has one trial. This is confusing, and a less confusing way around this is to work one row at time with `rowwise`:

```
bern.1$p %>% enframe(value = "p") %>%
  rowwise() %>%
  mutate(x = rbernoulli(1, p))
```

```
## # A tibble: 4,000 x 3
## # Rowwise:
##     name     p x
##    <int> <dbl> <lgl>
##  1     1 0.678 TRUE
##  2     2 0.792 TRUE
##  3     3 0.850 TRUE
##  4     4 0.457 TRUE
##  5     5 0.536 FALSE
##  6     6 0.574 FALSE
##  7     7 0.641 TRUE
##  8     8 0.705 TRUE
##  9     9 0.596 FALSE
## 10    10 0.790 FALSE
## # ... with 3,990 more rows
```

or

```
bern.1$p %>% enframe(value = "p") %>%
  rowwise() %>%
  mutate(x = rbinom(1, 1, p))
```

```
## # A tibble: 4,000 x 3
## # Rowwise:
##     name     p     x
##    <int> <dbl> <int>
##  1     1 0.678     1
##  2     2 0.792     1
##  3     3 0.850     0
##  4     4 0.457     0
##  5     5 0.536     0
##  6     6 0.574     1
##  7     7 0.641     0
##  8     8 0.705     1
##  9      9 0.596     0
## 10     10 0.790     1
## # ... with 3,990 more rows
```

If you used `cmdstanr`, your choices here are exactly the same, but starting from what I called `bern.2`.[3]

Extra: I'd also like to put in a plug for the `tidybayes` package. This works best with `rstan`, though it will work with `cmdstanr` also. The first thing it will help you with is setting up the data:

```
library(tidybayes)
tibble(x) %>% compose_data()
```

```
## $x
## [1] 1 0 1 1 0 1 1 1
##
## $n
## [1] 8
```

Starting from a dataframe of data (our `x`), this returns you a list that you can submit as `data =` to `sampling`. Note that it counts how many observations you have, on the basis that you'll be sending this to Stan as well (we did).

Another thing that this will do is to handle categorical variables. Say you had something like this, with `g` being a group label:

```
d <- tribble(
  ~g, ~y,
  "a", 10,
  "a", 11,
  "a", 12,
  "b", 13,
  "b", 14,
  "b", 15
)
compose_data(d)
```

```
## $g
## [1] 1 1 1 2 2 2
##
## $n_g
## [1] 2
##
## $y
## [1] 10 11 12 13 14 15
##
## $n
## [1] 6
```

Knowing that Stan only has `real` and `int`, it labels the groups with numbers, and keeps track of how many groups there are as well as how many observations. These are all things that Stan needs to know. See slides 32 and 34 of my lecture notes, where I prepare to fit an ANOVA model. The `tidybayes` way is, I have to say, much cleaner than the way I did it in the lecture notes. After you have fitted the model, `tidybayes` lets you go back and re-associate the real group names with the ones Stan used, so that you could get a posterior mean and interval for each of the two groups.

After obtaining the posterior distribution, `tidybayes` also helps in understanding it. This is how you get hold of the sampled values:

```
fit %>% spread_draws(p)
```

```
## # A tibble: 4,000 x 4
##    .chain .iteration .draw     p
##     <int>      <int> <int> <dbl>
##  1      1          1     1 0.429
##  2      1          2     2 0.416
##  3      1          3     3 0.328
##  4      1          4     4 0.501
##  5      1          5     5 0.588
##  6      1          6     6 0.626
##  7      1          7     7 0.707
##  8      1          8     8 0.685
##  9      1          9     9 0.565
## 10      1         10    10 0.479
## # ... with 3,990 more rows
```

which you can then summarize:

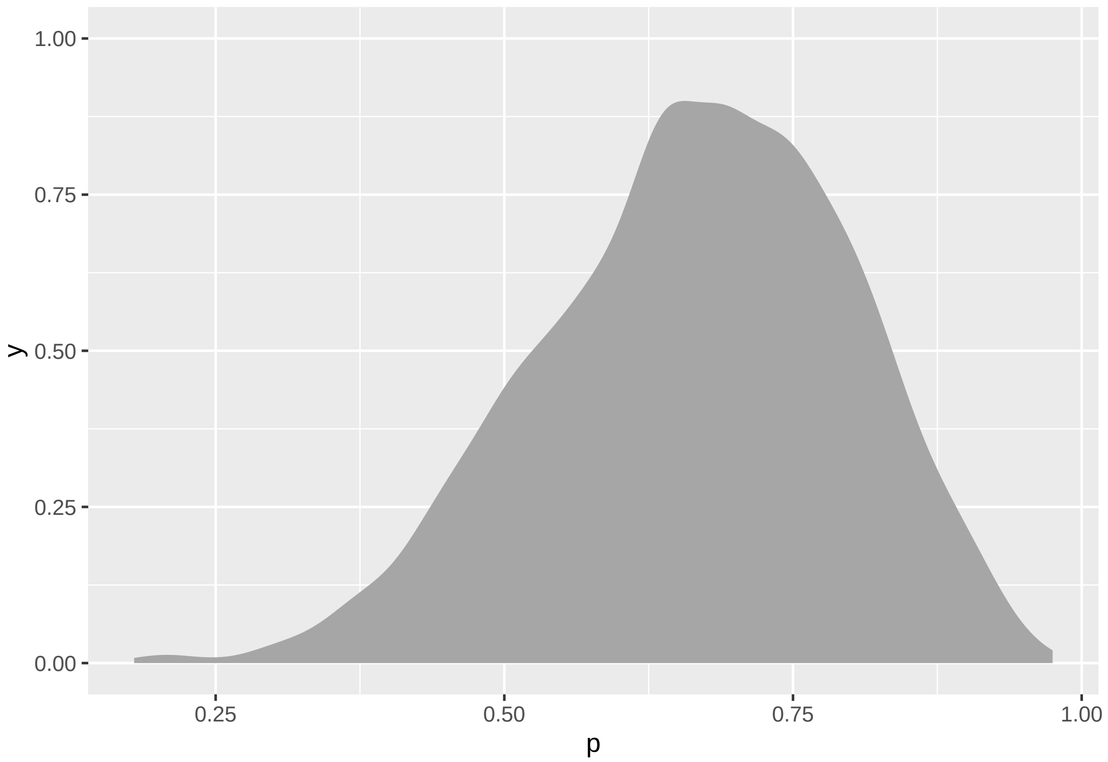
```
fit %>% spread_draws(p) %>%
  median_hdi()
```

```
## # A tibble: 1 x 6
##       p .lower .upper .width .point .interval
##   <dbl>  <dbl>  <dbl>  <dbl> <chr>  <chr>
## 1 0.674  0.417  0.916   0.95 median hdi
```

The median of the posterior distribution, along with a 95% Bayesian posterior interval based on the highest posterior density. There are other possibilities.

Or you can plot it:

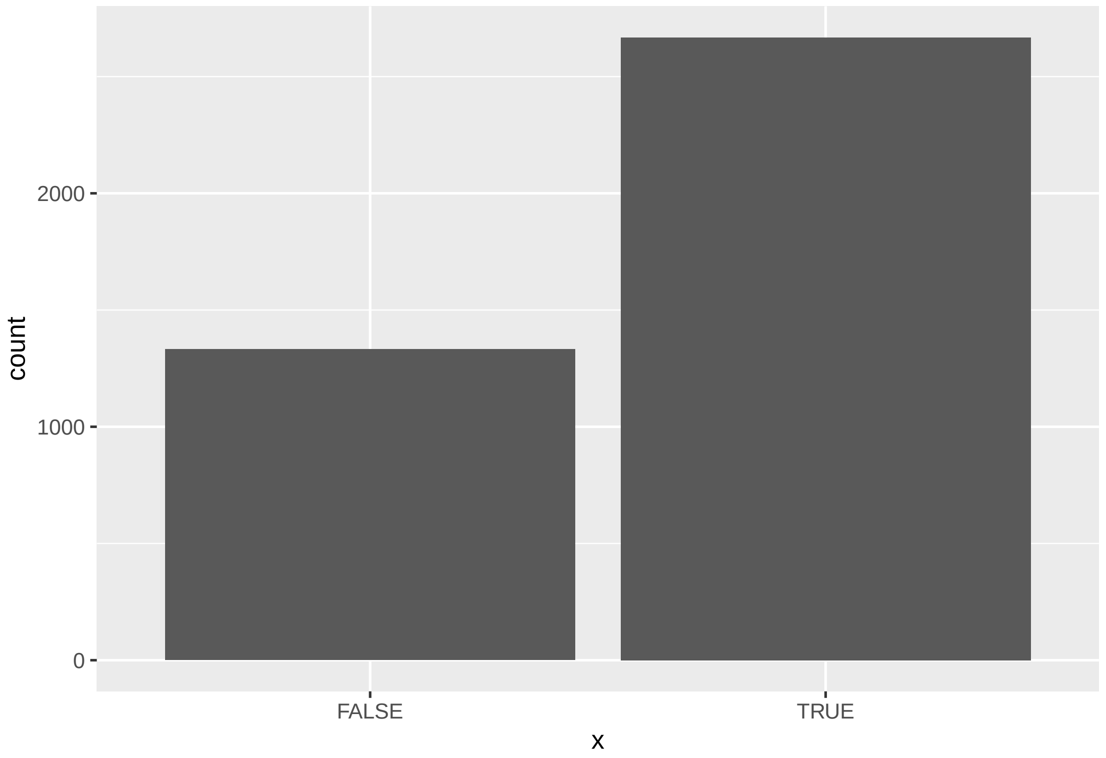
```
fit %>% spread_draws(p) %>%
  ggplot(aes(x = p)) + stat_slab()
```



(a density plot)

or, posterior predictive distribution:

```
fit %>% spread_draws(p) %>%
  rowwise() %>%
  mutate(x = rbernoulli(1, p)) %>%
  ggplot(aes(x = x)) +
  geom_bar()
```



This is a nice introduction to `tidybayes`, with a running example.

Notes

1. The comment line, with two slashes on the front, is optional but will help you keep track of what's what.

2. We'll come back later to the question of what `a` and `b` should be for our situation.

3. I am writing this a couple of days after the Ever Given was freed from blocking the Suez Canal. One of the memes I saw about this was actually a meme-upon-a-meme: on the picture of the tiny tractor and the huge ship, someone had superimposed that picture of Bernie Sanders sitting on his chair. Feel the `bern.2`.