A Modern Statistical Workflow

Jim Savage

9 April 2018

Why think about workflow?

Most of this book is about implementing common econometric models using Bayesian techniques. These work-horse models are really just the starting point for the sorts of models you might start to implement. As you become more creative in your model design, it will help to have a rigorous workflow. Focusing on this workflow will help you build better understanding of what is going on in your complex models, as well as catching bugs or improving the computational efficiency in the implementation.

The workflow is fairly simple:

1. Prepare and visualize your data
2. Come up with a generative model for the data. The first time, it should be as simple as possible.
3. Choose some priors for the model's unknowns. Draw some values from these priors
4. Simulate some "fake data" from your model given a draw from these priors
5. Fit your model to the fake data and check for the quality of the fit
6. Check that you were able to capture these "known unknowns". Possibly repeat 4-6, to get an understanding of bias or other issues in your fitting procedure
7. Fit the model to your real data, check the fit
8. Argue about the results with your friends and colleagues
9. Go back to 2. with a slightly richer model. Repeat.
10. Think carefully about what decisions will be made from the analysis, encode a loss function, and perform statistical decision analysis.

Each of these steps can be somewhat involved. This chapter described each step in some detail, with worked examples. Where more depth may be required, we encourage you to read the referenced resources. We start by providing some background on basic data manipulation and plotting in R. Though this is not a book about these "core data science" skills, the functions described here are used throughout the book, and knowing them will be necessary for the exercises throught the book. next we cover random variables, their distributions, the likelihood function, priors, and Bayes' rule. These concepts form the basis of building probabilitic generative models. Next, we focus on how to choose priors, and why this depends on your model and your data. Then we go through the workflow above in details. Finally, we work through the full workflow for two simple models.

Some required knowledge

This chapter deals with building probabilistic models of our data. Building such models requires that you understand at least the basics of probability, including how we think about random variables and their distributions, and a collection of commonly-used parametric distributions. It also requires that you can handle, manipulate and visualize data, which we'll call "core data science skills". This section provides a brief introduction to this required knowledge at a level that will help you understand what's going on later, but not much deeper. It is certainly worth investing in improving your understanding of core data science skills and probability theory, but we leave a lot of the details to some other resources. In particular, we recommend

Core data science skills

- Hadley Wickham's data science in R

Probability theory

- This introduction from Michael Betancourt
- This book for a deeper view (Jayne?)

Preparing and visualizing your data

Preparing and visualizing your data should be the first step to your analysis. It will help you discover potential issues far sooner in the analysis, and will help you to spot relationships that ought to be modeled. For data visualization in R, we strongly recommend you become acquainted with the `ggplot2` package used within the `tidyverse` family of packages. These allow for a swift exploration of your data along various facets. The `tidyverse` family of packages do have a relatively steep learning curve, particularly for those experienced in Excel or Stata. Yet they are certainly worth investing in.

The authoritative resource for this part of the workflow is the Data Science in R book mentioned above. It is available online for free, and is coauthored by the primary developer of the `tidyverse` family of packages. This section describes the main functions within this package which we'll use throughout the book.

Tidy data

One data format that makes both data preparation and modeling easier is the so-called "tidy data" format. The idea is to use data for which the the following hold

- Each variable is a column
- Each observation is a row

This is often *not* how data is provided to economists. For example, if you download the state accounts for Australia you will have data that look something like the following:

Date	NSW	VIC	QLD	
1990-06-30	276695	181556	111877	...
1991-06-30	277964	176618	112035	...
1992-06-30	278424	174085	115589	...
1993-06-30	286267	182347	123096	...
1994-06-30	296913	188796	129213	...
...	...	...	...	...

You will see here that each column corresponds to a state, and each entry corresponds to the state-year pair.

Or take the default output from the World Bank's data portal. Here, GDP per capita:

Country Name	2012	2013	2014	2015	
Aruba	NA	NA	NA	NA	...
Afghanistan	669	639	629	570	...
Angola	4598	4805	4709	3696	...
Albania	4248	4413	4579	3935	...
Andorra	38391	40620	42295	36038	...
...	...	...	...	...	...

Here we see that each row corresponds to a country, each column to a year, and each entry to a country-year pair.

What does a tidy data representation of these datasets look like? First let's take the state accounts. In order to perform the following operations, you should have already installed the `tidyverse` and `readxl` packages in R. You can do this by running

```
install.packages(c("tidyverse", "readxl"))
```

To convert these datasets into a tidy form, we'll use the `gather` function. Let's run it, to see what the output looks like, then explain the function.

```
library(tidyverse); library(readxl)
# read data
state_accounts <- read_excel("data/state_accounts.xlsx")[1:26,]

# "Gather" into long or tidy form
state_accounts_tidy <- gather(state_accounts, State, `Gross state product`, -Date)
```

Which gives us

Date	State	Gross state product
1990-06-30	NSW	276695
1991-06-30	NSW	277964
1992-06-30	NSW	278424
1993-06-30	NSW	286267
1994-06-30	NSW	296913
...	...	...
1990-06-30	ACT	18383
1991-06-30	ACT	18794
1992-06-30	ACT	18846
1993-06-30	ACT	19502
1994-06-30	ACT	20210
...	...	...

So what did we just do? The `gather` function takes the following arguments

```
gather(data, key, value, ...)
```

The `data` argument is the data frame we wish to convert. In our case, it is the wide/non-tidy gross state product. The `key` argument is the name we want to give to the new column that will be formed by the values which were column names in the wide dataset. The old dataset has abbreviated Australian state names, so we'll use `State` as the key. The `value` argument is the name we want to give to the new column formed by the values which will be "stacked" in the tidy format. Here, we want it to say `Gross state product`. Finally, the `...` argument lets us select the columns we wish to stack together. By default, all will be used. Any columns which we don't want to use as keys we should exclude. We do that using `-Date`. Run it again without excluding that column and see what happens. We can convert from "tidy" data to a wide format using the `spread()` function, for which we provide the key and value columns which will be converted into column headers and the values within each column.

```
# There are four meaningless rows at the top of the file, which we exclude
gdp_pc <- read_csv("data/API_NY.GDP.PCAP.CD_DS2_en_csv_v2.csv", skip = 4)

# Convert to tidy format and arrange by country and year
gdp_pc_tidy <- gather(gdp_pc, Year, `GDP per capita`,
                      -`Country Name`:-`Indicator Code`) %>%
  arrange(`Country Name`, Year)
```

Note three new things we've done here. The first is that we've excluded several columns. Because in the raw data these columns are adjacent, we use the notation `-`Country Name`:-`Indicator Code`` to "de-select" columns with names `` `Country Name` `` through `` `Indicator Code` ``. We've also used the "pipe" notation `%>%`, which takes the output from the preceding function and "pipes" it into the first argument of whatever

function comes next. Finally, we've used the `arrange()` function, which sorts the resulting data frame by whatever columns we specify; in this case, the country name and year columns.

The split-apply-combine idea

What was the point of putting data in this tidy format?

We almost always want to deal with data in this "tidy" format. The reason is that it allows us to use many extremely powerful data manipulation and visualization methods. These are primarily based on the "split-apply-combine idea", in which we can **split** our dataset up into sub-groups defined by one or more columns, **apply** a function to each subgroup, and combine the result into a new data frame. The package `dplyr` within the `tidyverse` family of packages provides support for operations of this flavour. Let's look at two: `mutate` and `summarise`.

`mutate` is one of a few functions we use for the "apply" part of "split-apply-combine"; it simply adds a new column to the data frame based on existing columns and, possibly, the grouping. For instance, in our example we might want to convert State Domestic Product into an index that is 1 in the year 2000, to allow for comparisons in State growth using that point in time as the base. We'd do that like so:

```
state_accounts_index <- state_accounts_tidy %>%
  mutate(Date = as.Date(Date)) %>%
  group_by(State) %>%
  mutate(`GSP in 2000` = `Gross state product`[Date=="2000-06-30"],
         Index = `Gross state product`/`GSP in 2000`) %>%
  group_by(Date) %>%
  mutate(`Weighted mean index` = weighted.mean(Index, `Gross state product`)) %>%
  ungroup
```

Here, we've added two new columns, the first, `GSP in 2000` to show how we can pull out a single value of the `Gross state product` column using a logical test, and a second, `Index`, to show that we can use previously-defined columns in defining a new one. Note that in order to use the logical test in the second `mutate` call, we need to first convert `Date` to a `Date` class.

What did `group_by(State)` do? In the "split-apply-combine" mantra, it is the "split"—we separate the data so that any functions used after the `group_by()` call *only refer to observations in that group*. To further illustrate this, we've performed two groupings—the first by `State`, the second by `Date`. You can also group by multiple columns at the same time.

If `mutate` allows us to add columns while refering to variables only within a group, then how can we perform group summaries? This is where the `summarise()` (or `summarize()` if you prefer) function comes in. `summarise()` takes a grouped data frame, and allows you to specify "summaries"—values of length 1—for each group. For example, say each state has GSP Y_t in period t and it grows at a constant annual growth rate of g per year. Let Y_0 be the initial GSP. Then

$$Y_t = Y_0(1+g)^t$$

Rearranging this gives us the average growth rate between periods 0 and t as

$$g = \left(\frac{Y_t}{Y_0}\right)^{\frac{1}{t}} - 1$$

Let s by the standard deviation of the actual annual growth rates around this "constant" rate. Let's produce summaries that describe these two statistics for each state, using the tidy data

```
state_growth <- state_accounts_tidy %>%
  group_by(State) %>%
```

```
  mutate(`Annual growth %` = `Gross state product`/lag(`Gross state product`)) %>%
  summarise(`Constant growth rate %` = 100*((last(`Gross state product`)/
                                               first(`Gross state product`))^(1/n()) - 1),
            `SD growth %` = 100*sd(`Annual growth %` - `Constant growth rate %`, na.rm = T))
```

Which gives

State	Constant growth rate %	SD growth %
ACT	2.49	1.44
NSW	2.36	1.22
NT	3.44	4.23
QLD	3.87	2.07
SA	2.16	1.69
TAS	1.84	1.61
VIC	2.62	1.97
WA	4.55	1.84

What have we done here? We've "grouped by" `State` (essentially splitting the tidy data frame into data frames for each state), then "summarised", reducing each state's data frame to a single row, where each column is defined within the `summarise()` function. The "combine" step of "split-apply-combine" happens automatically, after `summarise()` or `mutate()` is called.

Note that we've used a few functions that come packaged within the `dplyr`/`tidyverse` packages: `lag()`, `last()` and `first()`. `lag()` (and its counterpart, `lead`) returns the previous observation within a group; you can also ask for further lags by specifying the lag order `n`, `lag(x, n)`. The `last()` and `first()` functions return the last and first observations respectively. There is another function `nth()` which allows you to specify the position of the element you want to return—so `nth(c(1,3,5,7), 3) = 5`.

Plotting

> The most powerful econometric tool is the eye

Once our data are in a tidy format we can use the `ggplot2` package—also included in the `tidyverse` family, to explore our data. This section provides a very brief introduction to this package.

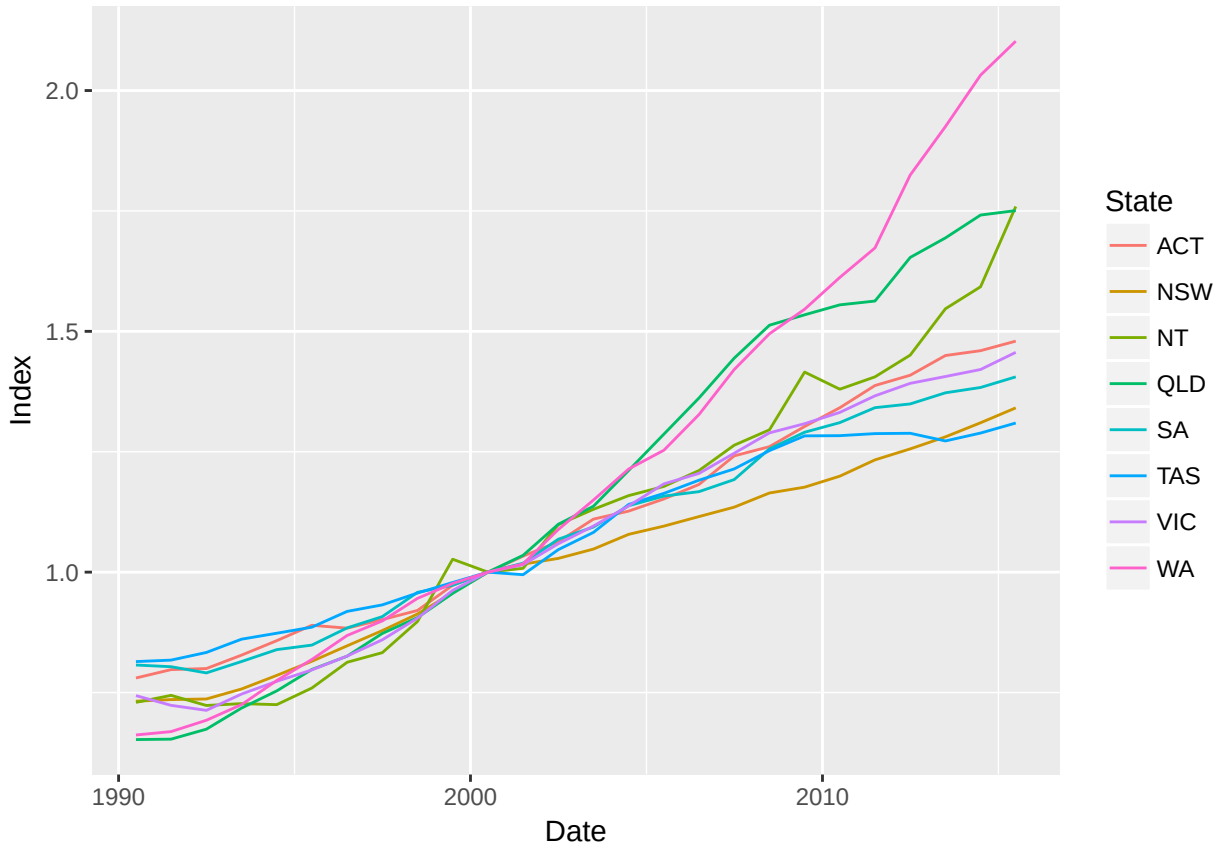
The central idea in `ggplot2` is to build up a plot, element by element. The basis of a ggplot object is a blank plot, created with `ggplot()`. On top of this, we can add more layers, each a "geom". For instance, a histogram is a type of geom; a set of lines is another; a density plot another still. The aesthetics of the geom, for istance the position, colour, size etc., are given by a mapping from variables (columns in your tidy data) to "aesthetics". There are several new terms there—let's illustrate with some examples.

We can start with a blank plot, given by piping our tidy data into ggplot. As you can see, if we don't add any geoms, we don't have a very interesting plot.

```
state_accounts_index %>%
  ggplot()
```

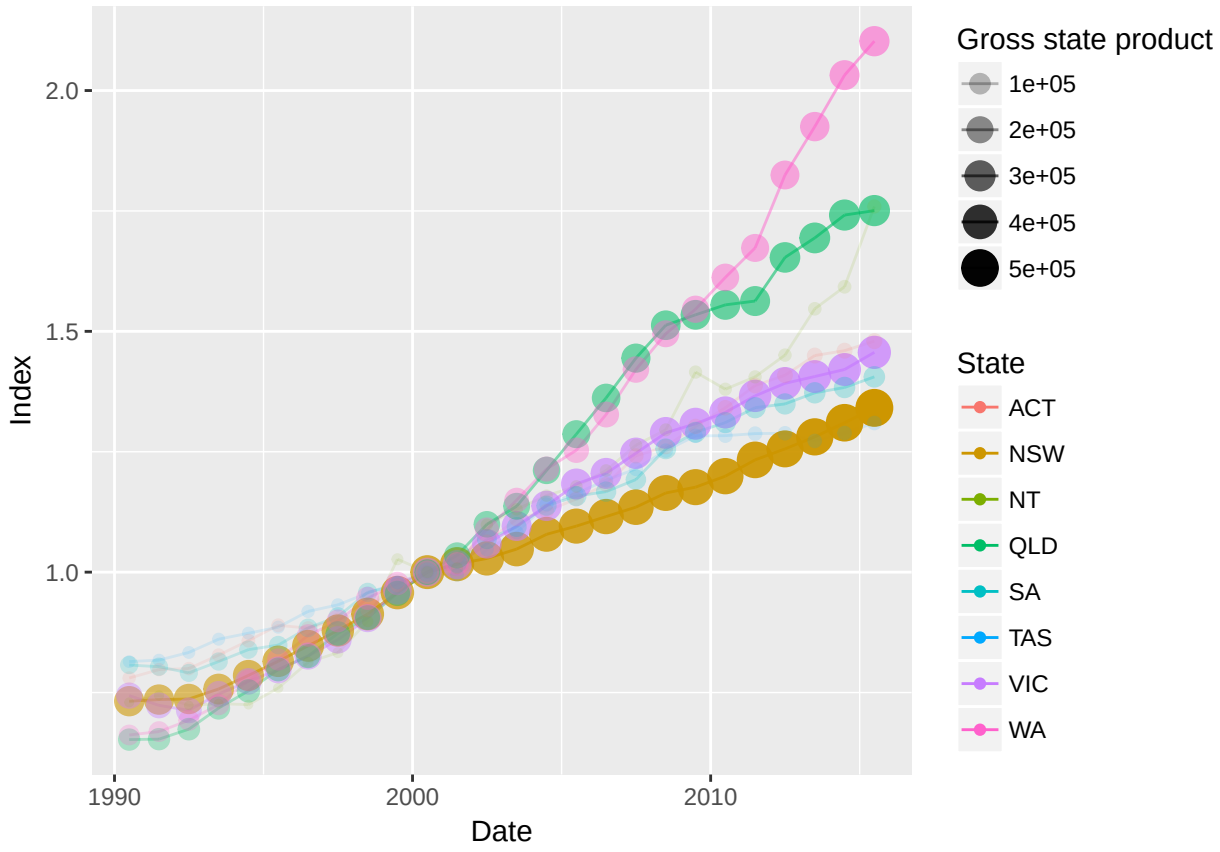
We might want to look at what has happened to our index of gross state product. To do that, we can add `geom_line()`, which has two mandatory aesthetics (the x and y mapping) and some optional ones, like group, colour (for when you have a column that defines which lines correspond to different colours), size, etc. For example:

```
state_accounts_index %>%
  ggplot() +
  geom_line(aes(x = Date, y = Index, colour = State))
```



This does a good job at illustrating the fact that Western Australia grew extremely quickly during the mining boom, and NSW and Tasmania did not. Yet the economies of NSW and Victoria are far larger than other states. How might we illustrate this? One approach is to create more mappings between the variables in our data and the acceptable aesthetics.

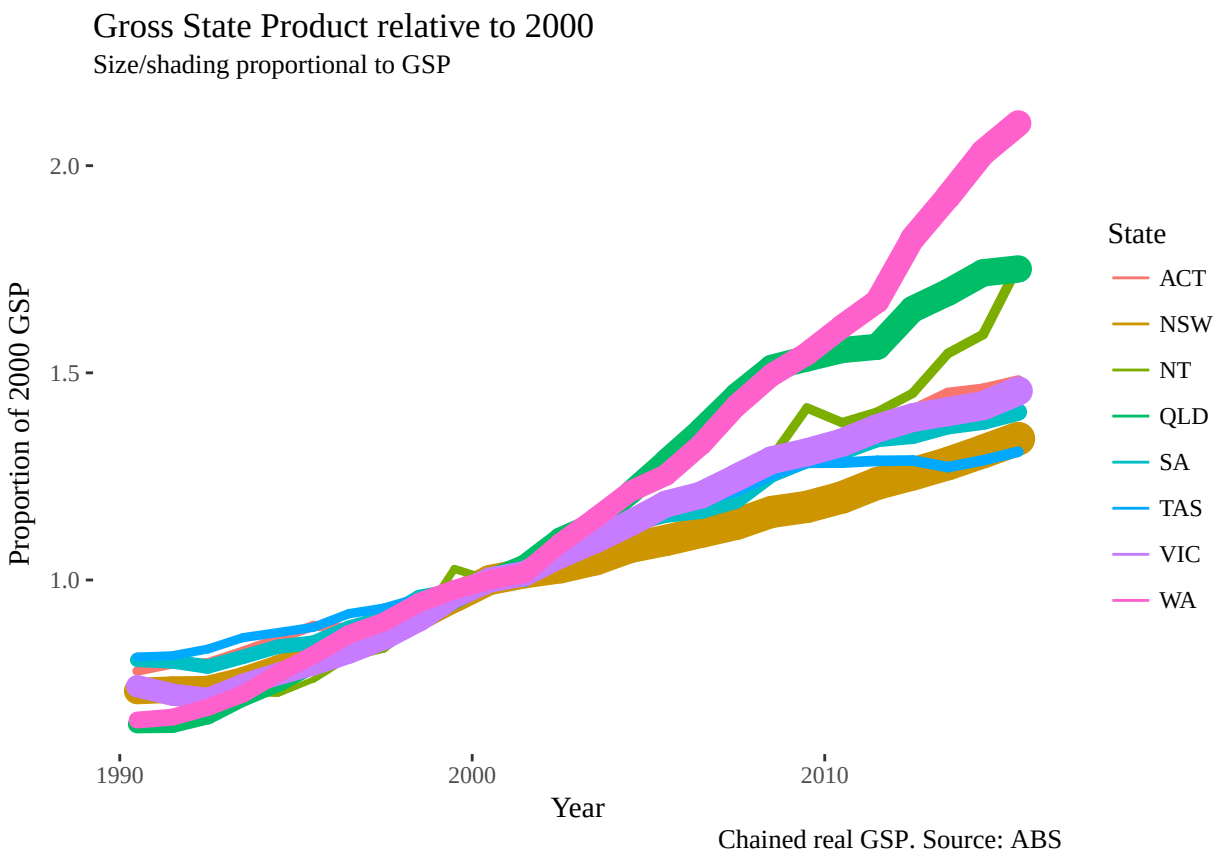
```
state_accounts_index %>%
  ggplot() +
  geom_line(aes(x = Date, y = Index,
                colour = State, alpha  = `Gross state product`)) +
  geom_point(aes(x = Date, y = Index,
                 colour = State, size = `Gross state product`,
                 alpha = `Gross state product`))
```



Now we can see that the two biggest states have grown fairly slowly relative to resource rich Western Australia and Queensland. The Northern Territory barely figures, given how few people live there.

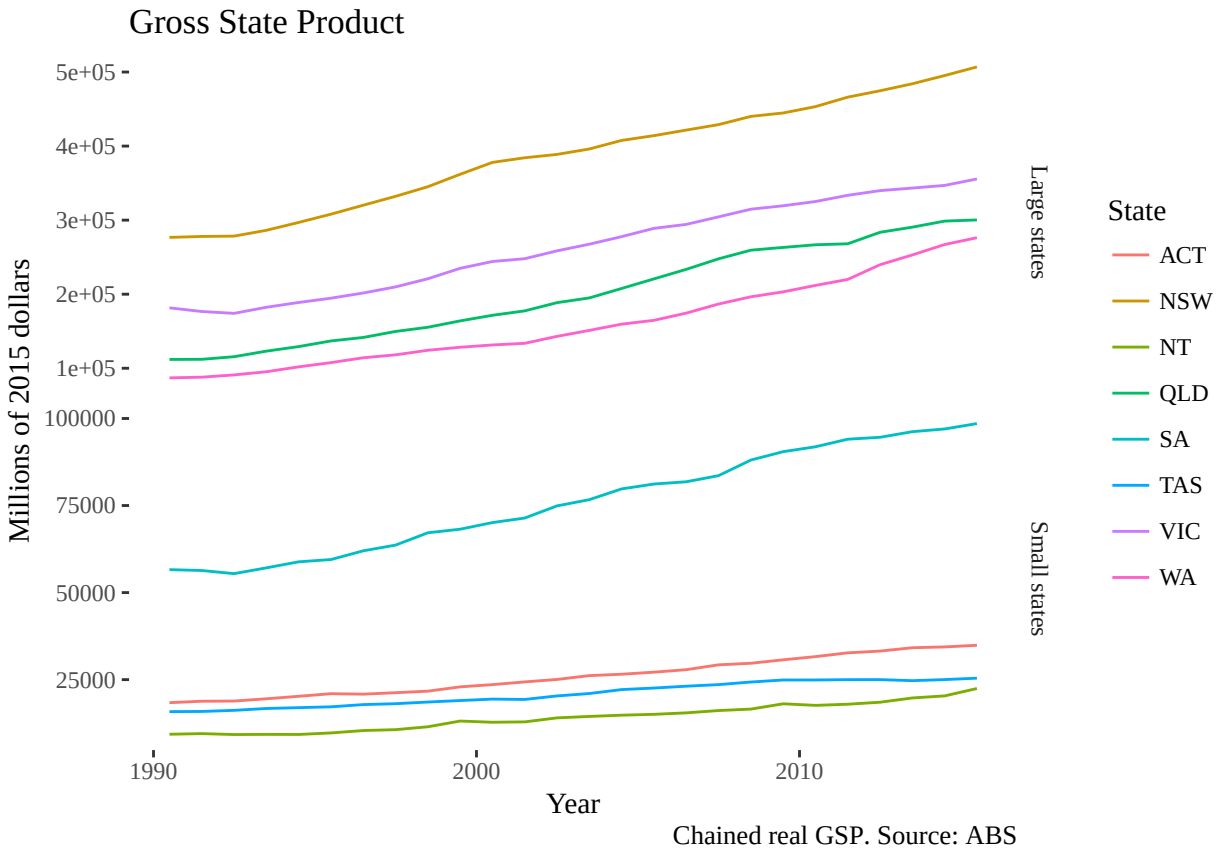
This plot is starting to look ok. We still need to add some trimmings. Let's use a more attractive theme (The "Tufte" theme, available in the `ggthemes`) package, and give it some new labels. Also, the alpha/size legend (a `guide`) isn't very helpful, so we will remove it.

```
state_accounts_index %>%
  ggplot() +
  geom_path(aes(x = Date, y = Index,
                colour = State,
                size = `Gross state product`), lineend = "round") +
  ggthemes::theme_tufte() +
  labs(x = "Year",
       y = "Proportion of 2000 GSP",
       title = "Gross State Product relative to 2000",
       subtitle = "Size/shading proportional to GSP",
       caption = "Chained real GSP. Source: ABS") +
  guides(size = F, alpha = F)
```



Now let's say we take this to a meeting and we receive feedback that the size/shading of the points wasn't really a very good way of illustrating the difference in economic activity across the states. instead, they want to see raw GSP, but have the chart split in two (one for big states, one for small ones). We can do something like this very easily:

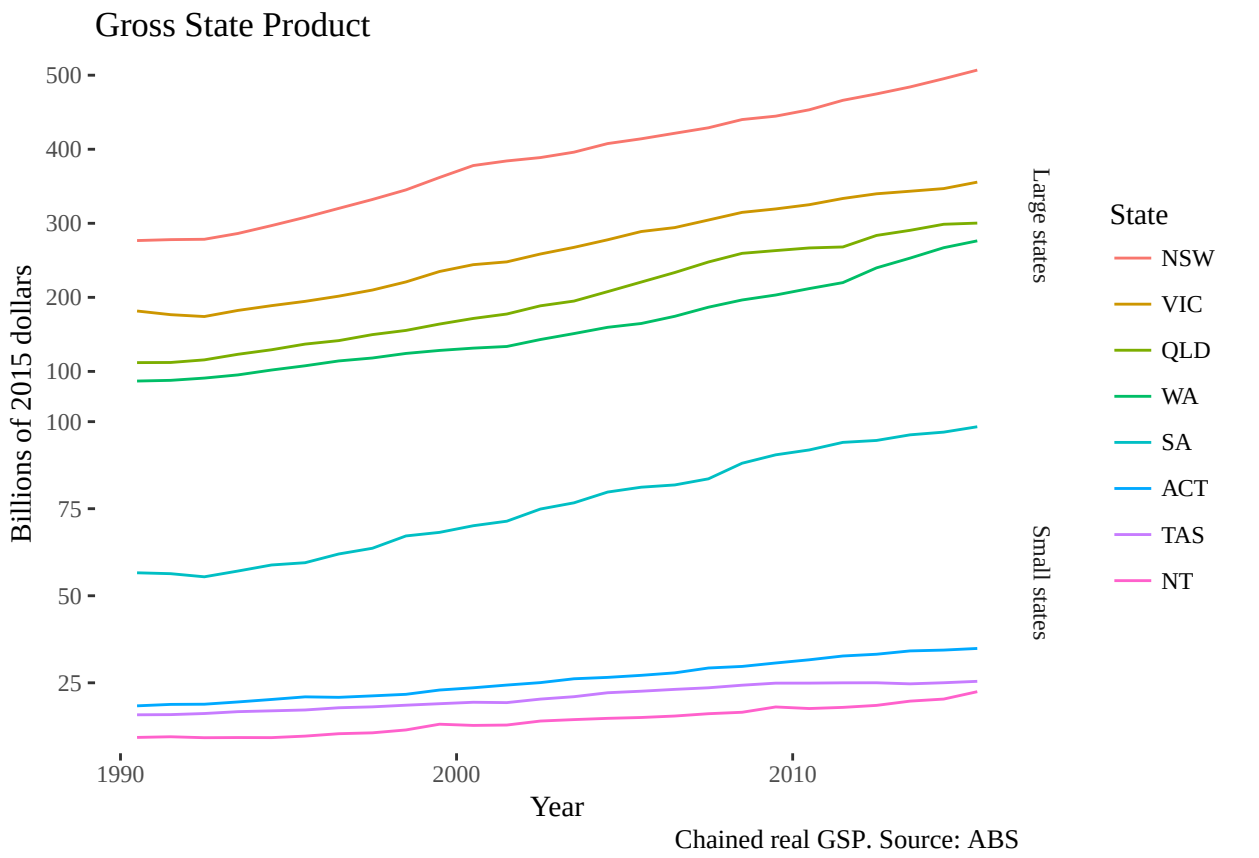
```
state_accounts_index %>%
  mutate(`State type` = if_else(State %in% c("NSW", "WA", "QLD", "VIC"),
                                "Large states", "Small states")) %>%
  ggplot() +
  geom_line(aes(x = Date, y = `Gross state product`,
                colour = State)) +
  ggthemes::theme_tufte() +
  labs(x = "Year",
       y = "Millions of 2015 dollars",
       title = "Gross State Product",
       caption = "Chained real GSP. Source: ABS") +
  guides(size = F, alpha = F) +
  facet_grid(`State type` ~., scales = "free_y")
```



What have we done here? We added two new things: the first mutates our source data to add a new column, which is `"Large states"` for the bigger states and `"Small states"` otherwise. We then use `facet_grid()` to split the plot into two smaller plots, each with their own y axis.

There are still a couple of slightly annoying thing about the plots above. The legend does not line up very nicely with the order of the lines on the plot. This is because by default, the discrete colour variable is converted into a factor where the order is given alphabetically. We want it to be ordered by the final GSP. So let's do this. Also, the scientific notation on the y-axis has to go. So let's rescale in terms of billions of dollars.

```
state_accounts_index %>%
  # Get the last GSP for each state
  group_by(State) %>%
  mutate(final_gsp = last(`Gross state product`)) %>%
  ungroup %>%
  # Reorder the State variable by decreasing order of final GSP
  mutate(State = reorder(State, -final_gsp),
         `State type` = if_else(State %in% c("NSW", "WA", "QLD", "VIC"),
                                "Large states", "Small states")) %>%
  ggplot() +
  geom_line(aes(x = Date, y = `Gross state product`/1000,
                colour = State)) +
  ggthemes::theme_tufte() +
  labs(x = "Year",
       y = "Billions of 2015 dollars",
       title = "Gross State Product",
       caption = "Chained real GSP. Source: ABS") +
  guides(size = F, alpha = F) +
  facet_grid(`State type` ~., scales = "free_y")
```



This concludes the very brief introduction to data manipulation and plotting using the `tidyverse` family of packages. This aspect of the workflow is not especially academic, but is extremely important. The rest of the chapter focuses on the more academic aspects of model building.

Random variables and their distributions

The Bayesian approach to model building is centered around estimating the distribution of random variables. So what is a random variable? And how do Bayesians think about random variables in their models?

Here are two questions for you. Do you know how much you are going to earn over the next three years? And are you aware how much reading through this book will affect those earnings? These sorts of questions get to the nub of much economic modeling: we want informed (probabilistic) forecasts and estimates of variables that might take on many possible values (and might do so because of random causes).

Random variables are variables like those alluded to in these two questions. The first question asks us to make a *forecast* about a variable (our income) that may take on many values (what if there is a recession, or if you change jobs, etc?). This is a random variable. The second question asks us for a *causal estimate* of this book on your earnings. In reality, this "treatment effect" is probably fixed, but our understanding of it will never be, and so we want to treat our inference of this value as a random variable.

What does it mean to "treat something as a random variable"? In practical terms, this means that we are up-front in saying that we do not know its true value *out of sample*. When we say "out of sample" we mean "for the yet-unobserved data for which we want to say something useful." This might be a broader population, or a specific population (future customers, say), or time periods in the future, etc. The entire point of building models is to be able to say something meaningful about the probable values of these random variables in a useful context.

Two more questions: do you know what sex you will be in two years' time? How about what day it will be in ten days' time? These are similar "forecasting" questions, but most people would be able to answer them with something close to certainty. If we were to be using the day of the week or a person's sex in a model to help predict some (random) outcome, we would not typically need to treat them as random variables. When we use these variables in a model, we typically call them *covariates*, *predictors* or in the machine-learning world, *features*.

All random variables have distributions

The point of using Bayesian methods is to understand the *joint distribution* of random variables. Let's set aside the *joint* part of this and briefly define a probability distribution. A probability distribution, also known as a *probability density* (for continuous random variables) or a *probability mass* (for discrete random variables) describes the relative probability of observing various values of a random variable. If we know the distribution of a random variable with certainty, then we can also make probabilistic statements about the random variable falling within a set of values. We typically do not know the distribution of the random variable, and have to infer what it might be.

As an example, let's say we're interested in the heights of men. The below comes from the US government's Census bureau:

From this distribution, we can make probabilistic statements, like "The probability of a randomly chosen male between 30-39 being less than 5'4" is 3.1%." The distribution above is known as an *empirical distribution*, the actual distribution of some data—in this case, from a population. When we bucket the data and present it like this it is is also known as a *histogram*.

Such a distribution is said to be *nonparametric*, in that we aren't using a parameterized analytical function to describe the relative probabilities of various heights. Note we can always draw a histogram of our data, indeed doing so is an important part of the modeling process.

In this example, we can plot the "true" empirical distribution of the data. First, human heights are an *observable* random variable. Whenever a random variable is observable, we can calculate its distribution directly, at least for the sample of data that we have at hand. (There is undoubtedly some measurment error, but we'll ignore that for now). Second, this is data from the census, so the distribution of heights that we observe is approximately equal to the true distribution of heights in the population. In most applied modeling work, few of these properties are satisfied; we often don't observe what we are truly interested in, and don't observe data at the population level. And so we need to make some assumptions.

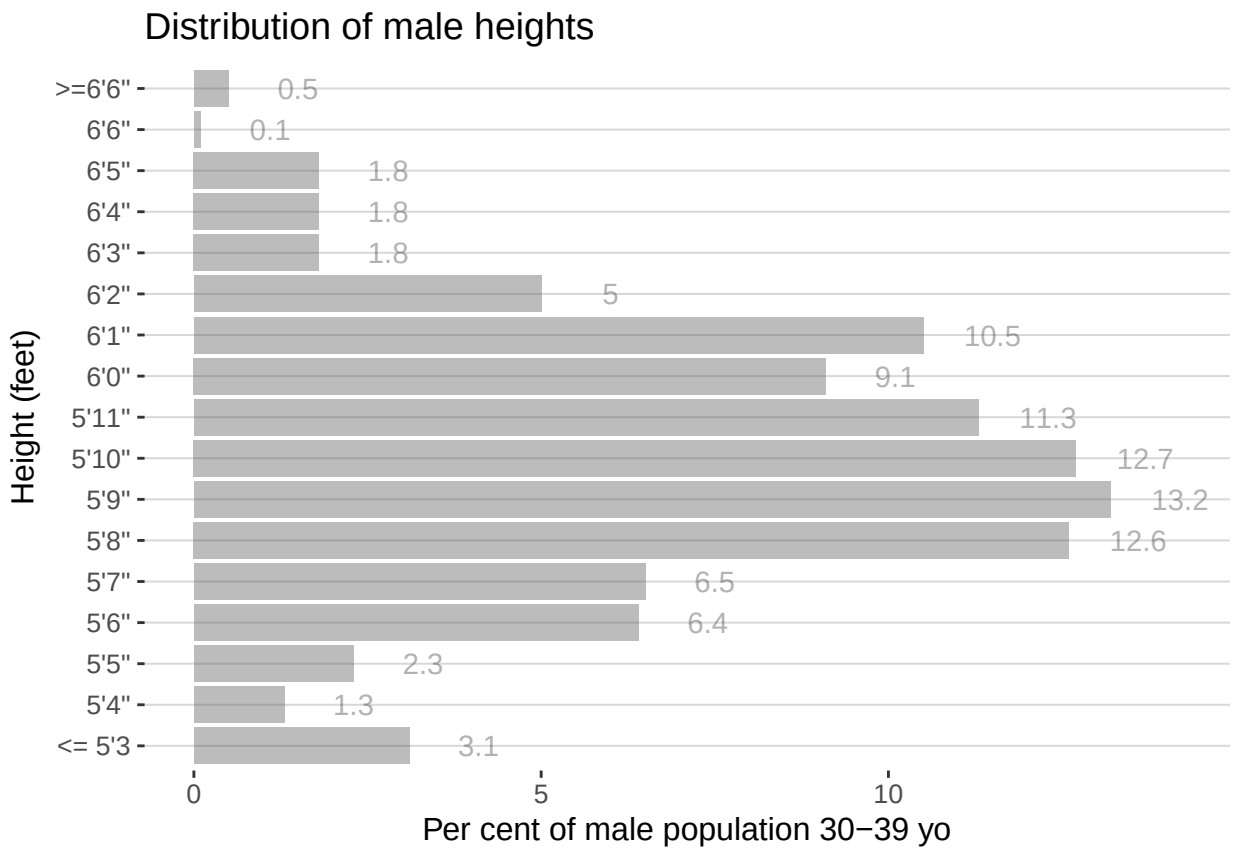


Figure 1: Heights of males in the US, 30-39y

When we build economic models, we're almost always interested in unobservables. If we are performing causal inference on a public policy experiment, we're interested in *treatment effects*—how much a policy impacts various stakeholders— which we never truly observe. A macroeconomist might want to know the impact of some fiscal policy on unemployment in a counterfactual state—again, unobserved. And engaging in a forecasting exercise involves making inference about an unknown, the future values of some series.

Just as applied economists aren't estimating the distributions of observed variables, applied economists rarely observe population-level data, relying on samples drawn from potentially biased sources. This isn't always the case; there is an exciting literature that makes use of population-level data from taxation agencies, for instance, Chetty (bunch of papers) and Kleven (some more). Yet most of the time we're being asked to draw inference for some new population (or the current population in the future).

When it comes to unobserved random variables, we're left with a sad reality: we cannot see them! So we can't calculate their empirical distribution. In order to make the analysis tractable, we need to make some more assumptions. In classical econometrics, these assumptions tend to center on the limiting distribution of random variables as the sample size approaches infinity. In many cases, for instance the tax studies by Chetty and Kleven cited above, these assumptions are very likely to be satisfied. Yet when our data are of a more limited size, or if the number of unknowns is high relative to the sample size, then we will need to make some further assumptions in order to say anything meaningful about the random variables in our model. That's where Bayesian modeling comes in.

Analytical (parametric) distributions

In most Bayesian applications, we make liberal use of parametric distribution functions to help us define and fit models. The type of distributions we use depends on the random variable we are modeling. Any function that satisfies the following can be used as a distribution function:

1. Must give non-negative values to values of the random variable;
2. Must integrate to 1 over the domain of the random variable;
3. The value of the function at x, $p(x)$, defines the relative probability of observing the value of x.

There are a huge number of possible parametric distributions that we can use, but the ones we choose will typically be decided by the nature (and constraints) of our data. Each parametric distribution is characterised by *parameters*. Typically, different values of these parameters imply different distributions of the random variables.

Below we list the sorts of things we need to check (typically by data visualization) in our data to help us choose the right distributions, as well as describing some commonly-used distributions.

Probability mass functions

Probability mass functions assign *probabilities* to various values of a discrete random variable. There are many types of discrete random variables and things we should look out for:

- **Binary/dichotemous or logical data**
  - In the canonical case, this is the toss of a weighted coin in which each face comes up with some probability. In industrial applications this might be whether or not a customer makes a purchase/choice, whether or not the economy is classified by the NBER as being in default, etc. This sort of data has a *Bernoulli distribution*.
- **Ordinal data**
  - A type of discrete data when the actual value is not observed, but the rank ordering is. When a survey asks us to rank a set of alternatives, the result is ordinal data. It does not measure *how much* we value one alternative relative to another, but does constrain the possible cardinal values. We typically use a generalization of the Bernoulli distribution to model ordinal data.
- **Categorical data**

  - When our outcome variable is an unordered discrete variable (for instance, which product you choose choose from a set of alternatives, or a person's ID code, then the data is "categorical". When modeling this sort of data we use the *categorical distribution*, which can be thought of as rolling a weighted dice. If our data is counts of categorically distributed data then we use the *multinomial distribution*. The categorical distribution is a special case of the multinomial distribution (one with a single count).
- **Count data**
  - Often our data are counts of an event, for example the a discrete number of purchases. This sort of data can be unbounded or unbounded. An example of bounded counts would be "given 100 customers were in the story today, how many made a purchase?" Such count data is normally modeled using the Binomial distribution. Unbounded count data, especially that referring to the number of events that occurs over a fixed interval of time, is often modeled using the Poisson distribution. For example, "how many goals will Everton kick in a game?"
  - Count data as in these examples often have a couple of characteristics that these distributions not fit the data particularly well. The first is *zero inflation*—we observe more 0 counts than the distribution would suggest is probable. The second is *over-dispersion*, where we observe a wider range of counts than would be expected by the Poisson distribution. It's fairly straightforward to include these characteristics in a parametric mass function.

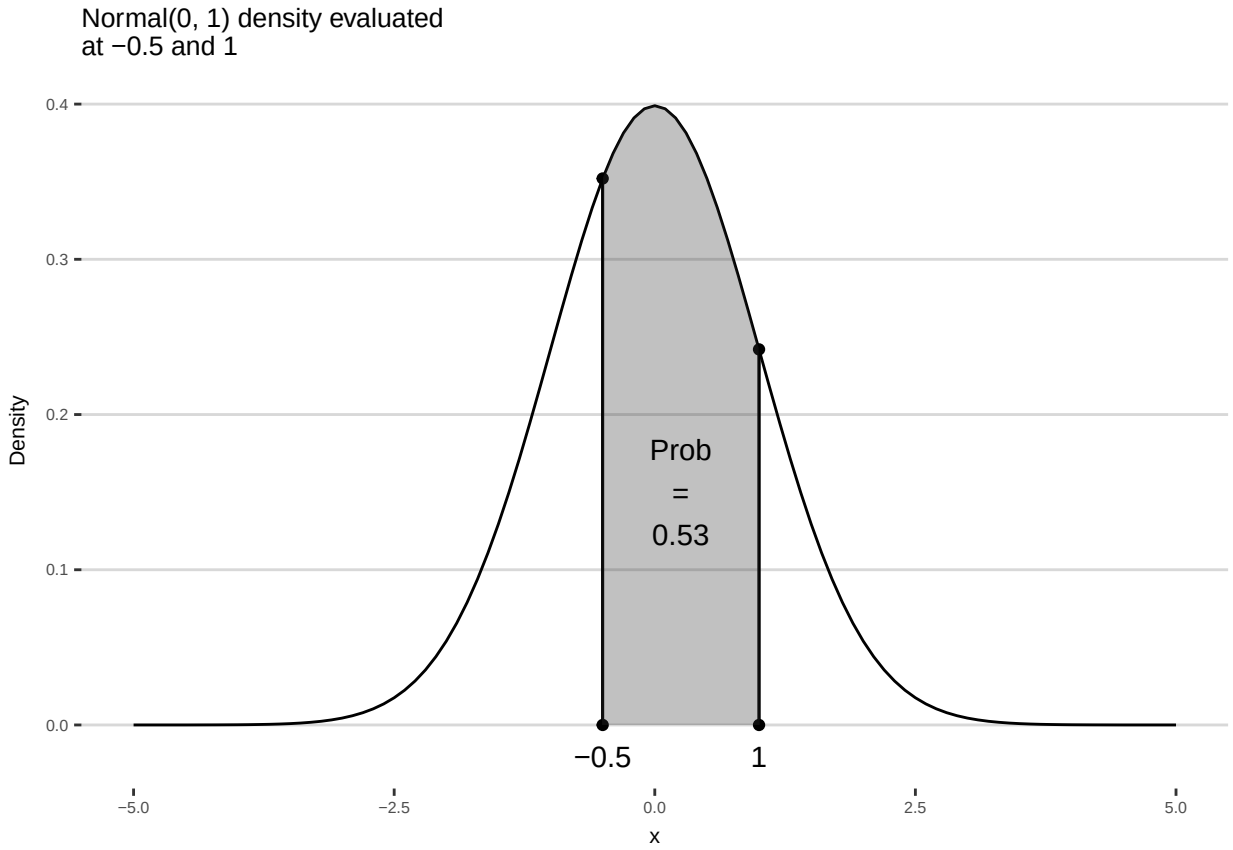
Probability density functions

When our random variable is continuous, we use *probability density functions* to describe its probabilistic distribution. These functions describe the relative likelihood of observing various values of a random variable. They also describe the probability of observing values of a random variable within a range. For example, take a standard normal density (that is, a normal density with mean $\mu = 0$ and scale $\sigma = 1$. To get the probability that a random variable with this distribution falls in the interval $(-0.5, 1)$ all we need to do is integrate the density between those values

$$\text{Given } x \sim \text{Normal}(0, 1)$$

$$p(-0.5 < x < 1) = \int_{-0.5}^{1} \text{Normal}(x|\, 0, 1) dx$$

This is illustrated below



Let's quickly discuss the notation used above. Above, we said "Given $x \sim \text{Normal}(0, 1)$". This can be read as "given that x (a random variable) is distributed with a normal density with location parameter 0 and scale parameter 1". Often we use $\sim$ to denote a "sampling". For instance, you could generate draws of x by drawing from a standard normal distribution. If we wanted 1000 draws, we could do this in R like so:

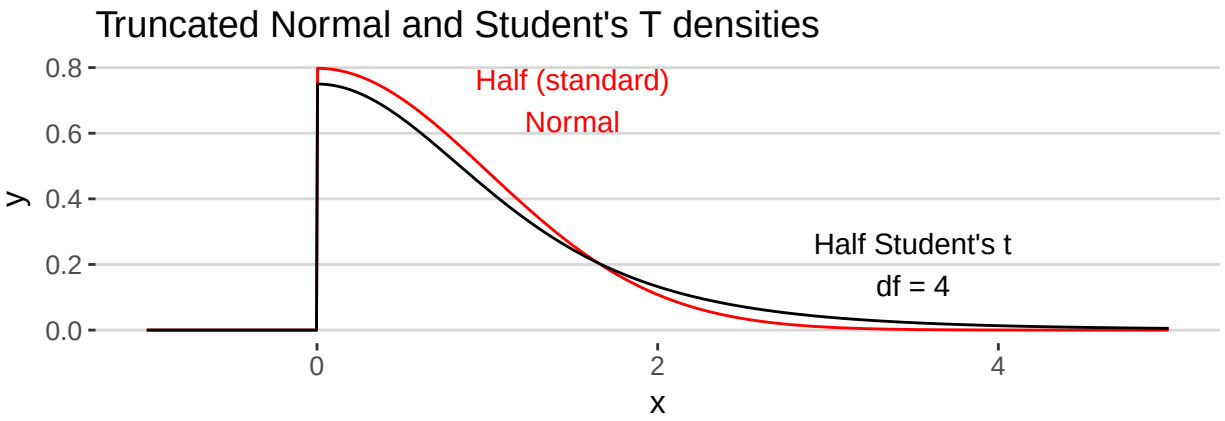
```
x <- rnorm(1000, mean = 0, sd = 1)
```

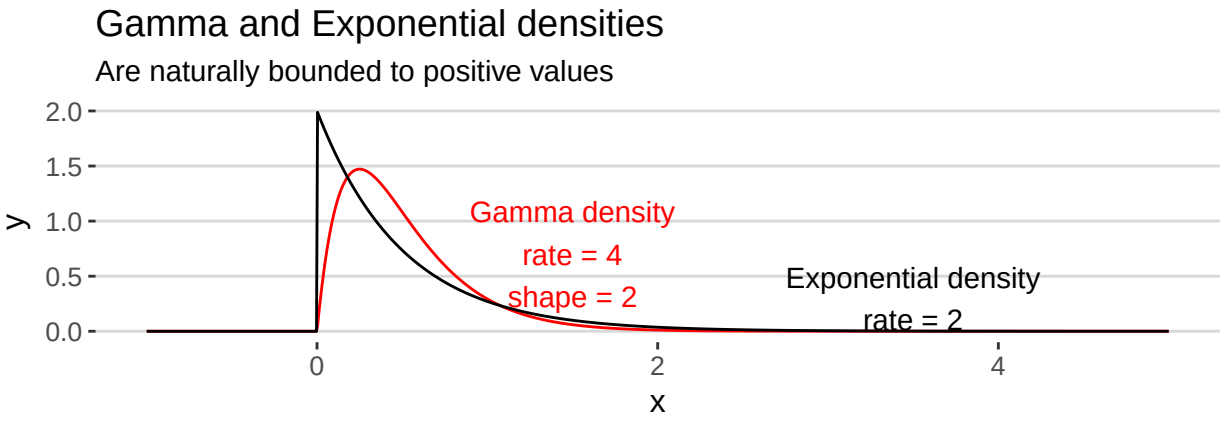
or just

```
x <- rnorm(1000) # by default it's a standard normal
```

Just as with discrete random variables, we typically choose the type of continuous density based on the nature of the random variable we're modeling. The "default" is almost always a normal distribution, yet it often should not be. A few things we have to look out for:

- **Is the distribution bounded?**
  - Often random variable has a natural bounding; for example, many prices should be positive, and for normal goods, we expect price elasticies of demand to be negative. Others are unbounded: GDP growth or inflation could be positive or negative.
  - When a random variable is unbounded, we use unbounded distributions. Commonly, the Normal distribution, the Cauchy, Student's T, Double Exponential, and Gumbel distributions are used.
  - When the random variable is bounded, we use either truncated versions of the unbounded distributions above (these give no probability to values outside the bounding), or distributions that have a natural bounding. Examples of the latter include the Gamma/inverse Gamma distribution, the exponential, and the lognormal distribution. Four examples are illustrated below.





- **Does the random variable have "fat tails"?**
  - Often we want to allow for "fat tails"—values of the random variable that are a long way from the bulk of the distribution. Common use-cases are in modeling financial markets, where big positive and negative jumps in asset prices happen with some frequency. We also often use fat-tailed prior distributions (discussed later in this chapter) to allow for the possibility that the true value of the unknown lies a long way from our prior.
  - Many densities have fat tails; for unbounded densities we often use Student's T, Cauchy and Gumbel densities; for positively-constrained variables we often use Exponential and Lognormal densities.
- **Top and bottom coding**
  - Top and bottom coding involve a random variable having a limit, and being clustered at the limit.
  - TKTK

Joint distributions

We are often interested in the *joint* behaviour of random variables.

- Joint distributions
- Correlation
- Covariance

Why parametric distributions?

In Bayesian modeling, we use parametric distributions for three main purposes. First, to describe the distribution of the observable variables that we are modeling (given the unknowns in the model, and the "knowns" that we're not modeling). For example, if we're modeling the joint distribution of unemployment, wages growth and GDP growth, then we would choose a parametric distribution for their joint behaviour given the model unknowns and things that we don't want to model (say, fiscal policy). A wide variety of

parametric distributions, and even "mixtures" of them, can approximate many data that we see in the wild.

The second use is to allow us to summarise our prior information about model unknowns before fitting the model. We discuss priors later in this chapter, but all you need to know for now is that we almost always use parametric distributions as priors.

Finally, we use parametric distributions to help us evaluate whether the model unknowns we propose are good or bad. In this sense, parametric distributions can act like *loss functions* (to borrow a term from machine learning), returning larger values when "better" unknowns are proposed. We discuss this more thoroughly in the Likelihood section of the chapter.

Parametric distributions have parameters

The defining characteristic of parametric distributions is that their shape is characterized by a set of parameters. For instance, the normal distribution has a location (mean), often μ, and scale (standard deviation), often σ. Sometimes you will see the normal distribution "parameterized" in a different way, for instance with a location μ and variance σ^2. Or perhaps using the location μ and "precision" $\tau = 1/\sigma^2$. Often it can be convenient to reparameterize models like this to aid in interpretability of estimates, or for computational reasons.

When we write

$$y_i \sim \text{Normal}(\mu, \sigma)$$

we read this as "each observation i of the the random variable y is distributed normally with location μ and scale σ". Given this information, we can calculate any statistic we want about the distribution of yet-unseen values of y. For instance, the expected value $E[y] = \mu$, the standard deviation is σ, and so the variance σ^2 and so on.

The entire game of model building using parametric distributions is to replace the parameters with functions which may contain unknowns. For example, the normal linear model

$$y_i = X_i\beta + \epsilon_i \text{ with } \epsilon_i \sim \text{Normal}(0, \sigma)$$

can be written

$$y_i \sim \text{Normal}(X_i\beta, \sigma)$$

Note what we've done—we've simply replaced the location μ with a function describing the conditional mean of observation i, $X_i\beta$. In this model, each observation can be centered around a different point ($\mu_i = X_i\beta$), but all have the same error scale around this point σ. This is a simple linear function, but there's nothing stopping us from replacing it with something that contains more information—for example, it might be a function that finds a Nash Equilibrium of some game (we'll see these in chapter 4). Just as easily, we could let each observation have their own error scale, $\sigma_i = \exp(X_i\gamma)$

$$y_i \sim \text{Normal}(X_i\beta, \exp(X_i\gamma))$$

Most commonly, scale parameters are modeled explicitly in the finance and macroeconomics literature—we cover several such models in chapter 5. A number of common distributions and their parameter names are given on the following page.

The following section describes how we can use parametric distributions—chosen with respect to our data—to evaluate the likelihood function of the data given a proposal of some unknowns. This is a step towards fitting models using Bayesian techniques.

The likelihood function

In order to fit Bayesian models we need to construct a function that tells us when certain values of model unknowns are good or bad. For example, in image @ref(goodbaddensity) below, we plot a histogram of values of some random variable X. We want to fit a density function to this histogram so as to be able to make probabilistic statements about the likely distribution of yet-unseen observations. To gauge which proposed density is a good one, we would like a function that gives a higher value for the proposed model unknowns that lead to the distribution in the bottom panel and lower values for the proposed model unknowns that lead to distribution in the top panel.

There are many functions that we could use to determine whether some proposed model unknowns result in a "better" fit than some other unknowns. Likelihood functions are one approach, based on the assumption that the proposed *generative distribution* gave rise to the observed data. This is most easily motivated with an example.

Example: the binomial likelihood of loan defaults

Let's say we wanted to estimate the probability of loan defaults. We have a small sample—only five loans—and one has defaulted. If we encode defaulted = 1 and not defaulted = 0, then our data look like

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

The likelihood function answers the question "what is the probability of observing this sequence if the probability of defaulting is θ?" In the case that the outcomes are independent of each other, then this is the product of the *likelihood contributions*—the probability of observing each observation. The probability of observing a non-default is $(1 - \theta)$, and so the probability of observing a default is θ. For example, let's say we think the probability of defaulting is 2%. Then our likelihood (the product of the likelihood contributions) would be

$$p(y = (0, 0, 1, 0, 0)' | \, \theta = 0.02) = 0.98 \times 0.98 \times 0.02 \times 0.98 \times 0.98 = 0.0184$$

The above is the specific case to this example. In general, the Bernoilli likelihood—the likelihood function we use for binary outcomes—is

$$p(y | \, \theta) = \prod_{i=1}^{N} \theta^{y_i} \times (1 - \theta)^{1 - y_i}$$

Note that when $y_i = 0$—no default—$\theta^{y_i} = 1$. Now, is the proposed probability of defaulting a good one? That's what we can use the likelihood function to perhaps determine. We plot the values of the likelihood function for this data evaluated over the possible values of θ below.

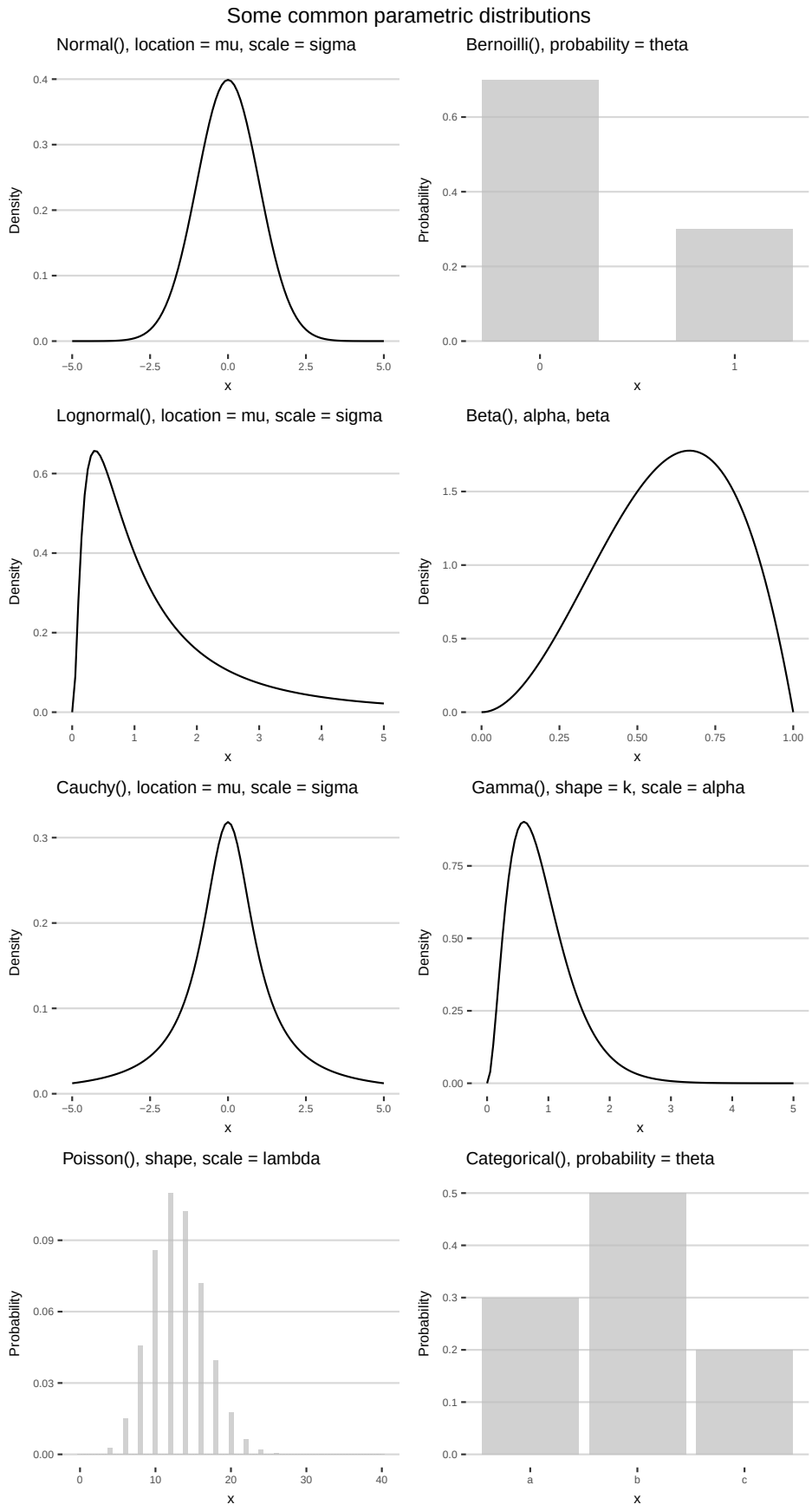


Figure 2: A collection of commonly-used distributions and their parameter names

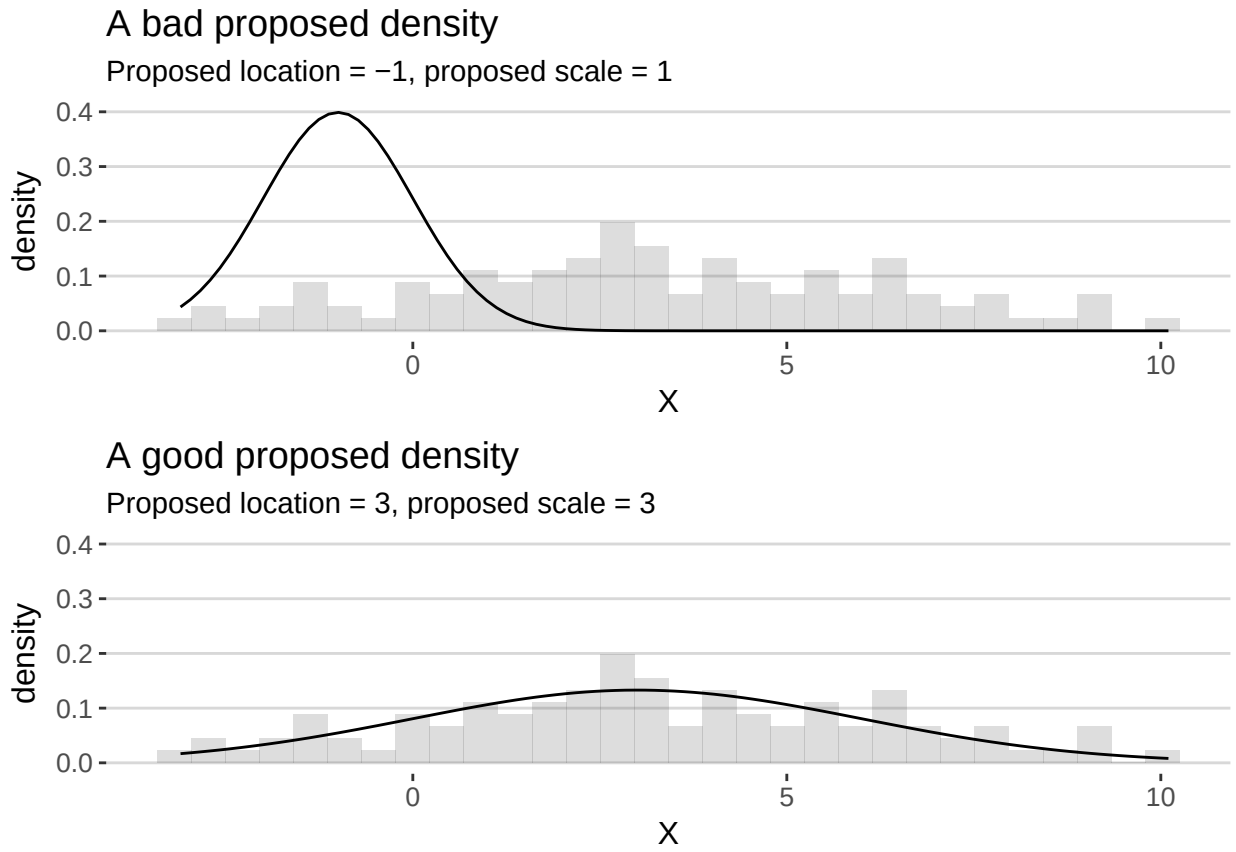
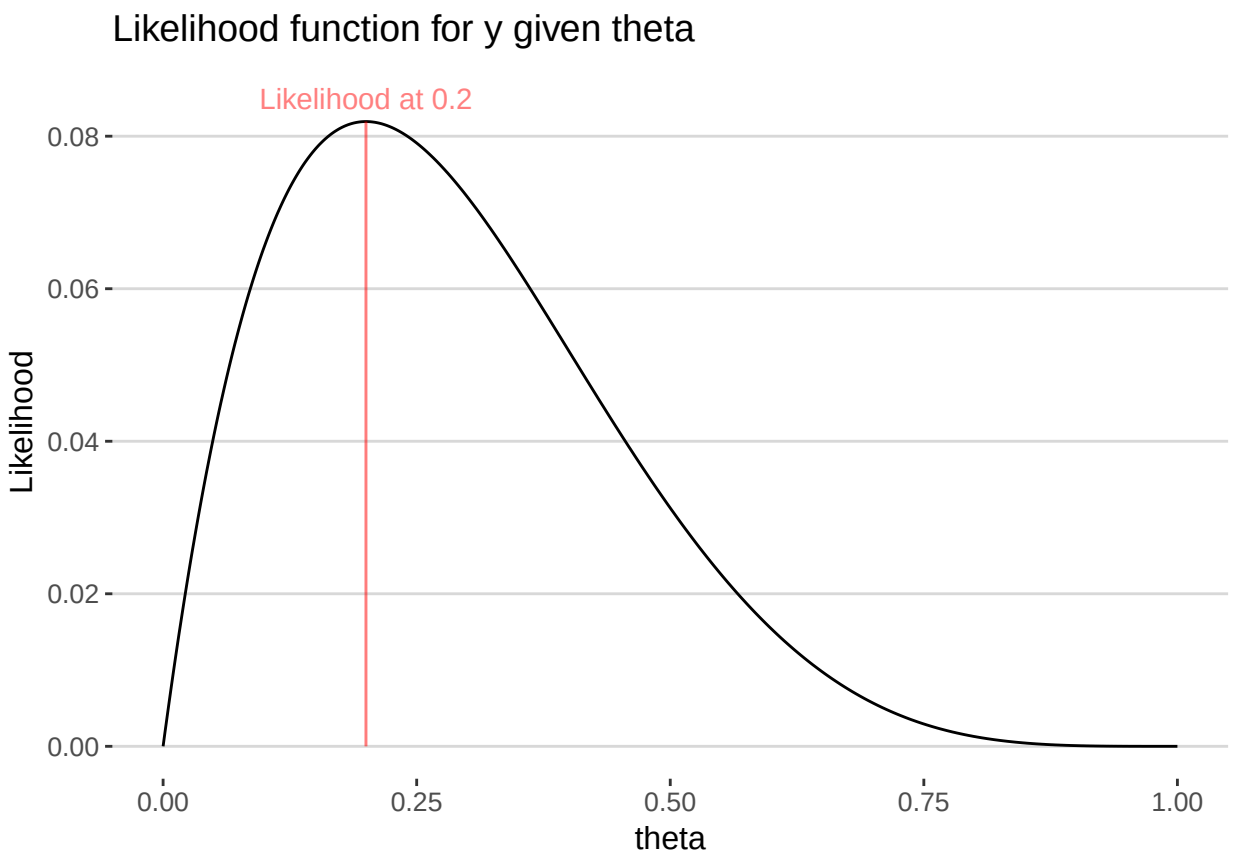


Figure 3: A likelihood function would return a higher value for the proposed density on the bottom than the proposed density on top.



What do we notice about this likelihood? First, it's *not* a probability distribution in θ—it does not integrate to 1. This is a common source of confusion, firstly because it looks like many of the distributions we're used to working with. But also because we write it as Likelihood $= p(y|\,\theta)$. It *does* integrate to 1 over the possible values of y, but not over the possible values of θ. Second, we notice that in this case it is maximized at 0.2, which happens to be the proportion of loans in our very small sample that have defaulted. The maximum point of the likelihood function is known as the *maximum likelihood estimate* (or MLE) of our parameter θ given our data. Formally,

$$\hat{\theta}_{MLE} = \text{argmax}_{\theta}(p(y|\,\theta))$$

The plot illustrates something potentially dangerous about using the Maximum likelihood estimate with small samples. In this case, before we did the analysis, we had reason to believe that the default rate was 2%. This is what we might call *prior information*; information that exists from sources other than the sample of data we're analyzing. Unfortunately our maximum likelihood estimate does not reflect this prior knowledge at all, and we're left with a probability of default simply equal to the sample frequency. Later in this chapter we'll discuss how to encode this outside information in our estimate.

The Bernoulli log likelihood

In this simple example, in which we have only one unknown and very few data points, we have no problem evaluating the likelihood function directly. Yet you might have noticed that as we provide more observations of y to the function, it will get closer to 0—in the discrete case, the likelihood contribution of each observation is between 0 and 1, so multiplying more and more of them together results in a rapidly shrinking likelihood value. If you had a large number of observations for example, your computer will simply not be able to handle that number of leading 0s.

```
# draw 100k 0-1 coin flips
y <- sample(0:1, 1e5, replace = T)
```

```r
# A bernoulli likelihood function
bernoulli_l <- function(theta, y) {
  prod(theta^y * (1 - theta)^(1 - y))
}
# evaluate likelihood at the correct value
bernoulli_l(0.5, y)
```

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

In order to handle these issues with numerical precision, we use the *log likelihood function*. This is, as the name suggests, the log of the likelihood function. In the Bernoulli case, it's

$$\log(p(y|\,\theta)) = \log\left(\prod_{n=1}^{N} \theta^{y_i} \times (1-\theta)^{1-y_i}\right) = \sum_{i=1}^{N}(y_i \times \log(\theta) + (1-y_i) \times \log(1-\theta))$$

Which makes use of the fact that $\log(ab) = log(a) + log(b)$ and $\log(a^x) = x\log(a)$. This representation of the likelihood is far easier for us to work with than the raw likelihood. For one, it is order preserving—the values of the unknowns that maximize the log likelihood are the same as those that maximize the likelihood—and yet we sum the log likelihood contributions, so small probabilities don't send the value towards 0. Second, in some cases (for instance, with a binomial or categorical likelihood), you can evaluate it with linear algebra. If we want the likelihood of a vector $y = (0, 0, 1, 0, 0, \dots)'$ evaluated with the probability vector $\theta = (\theta_1, \theta_2, \dots)'$ denoting the probability of each observation, then the log likelihood is just

$$\log p(y|\,\theta) = y' \log(\theta) + (1-y)' \log(1-\theta)$$

We use this formulation extensively in Chapter 4.

Now that we have a basic understanding of the log likelihood function, let's take a look at the log likelihood function for a model with a continuous outcome and three unknowns.

Log likelihood of a simple normal linear model

Let's say we have a very simple linear model of y with a single covariate x. α, β and σ are the model's parameters.

$$y_i = \alpha + \beta x_i + \epsilon_i$$

with $\epsilon_i \sim \mathrm{N}(0, \sigma)$. This is the same as saying that

$$y_i \sim \mathrm{N}(\alpha + \beta x_i, \sigma)$$

This is the model we are saying *generates* the data—$\mathrm{N}(\alpha + \beta x_i, \sigma)$ is the *generative distribution*. It has three unknowns, α, β and σ. If we knew what these values were for sure, we could make probabilistic statements about what values y_i might take if we knew the value of x_i.

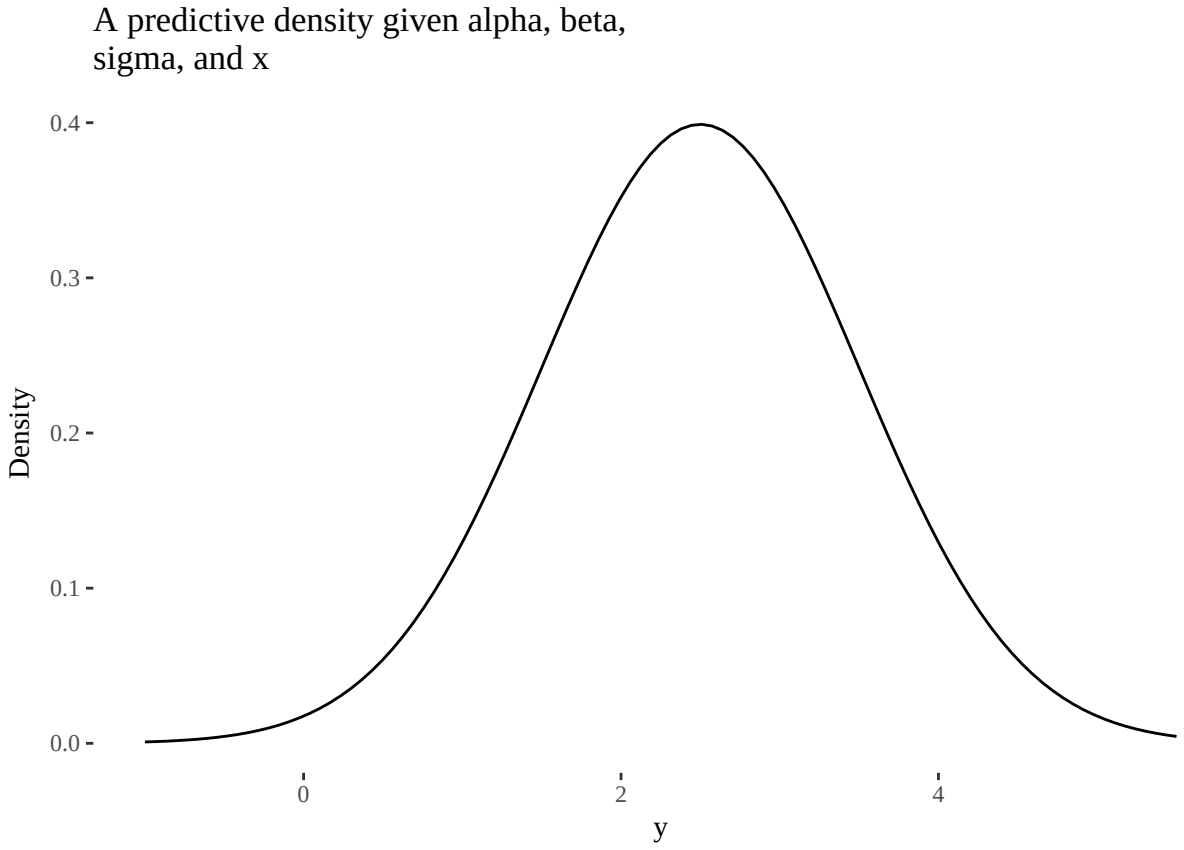
But that's not how data analysis normally works. Normally we're given values of x and y (or many xs and many ys) and have to infer the relationship between them, often with the help of some model. Our task is to estimate the values of the unknowns—in this case, α, β and σ.

So how do we do this by maximum likelihood? One thing that computers are relatively good at is choosing values of some unknowns in order to optimize (normally minimize) the value of some function. Here we use the normal log likelihood function as a way of scoring various combinations of α, β and σ so that the score is high when the model describes the data well, and low when it does not.

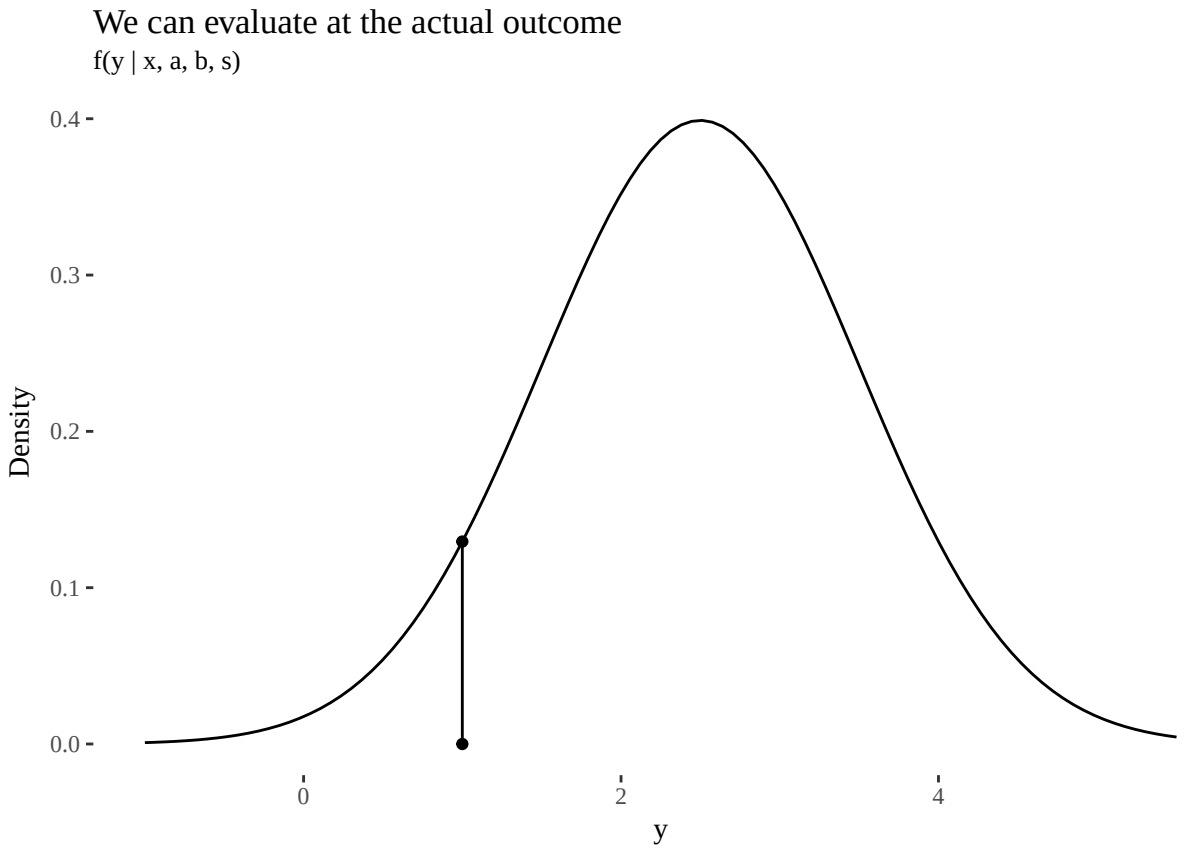
Let's say we propose a set of parameters, $\alpha = 2$, $\beta = .5$ and $\sigma = 1$ and we have an obervation $y_i = 1$, $x_i = 1$. Given the parameters and x_i we know that the outcomes y should be distributed as

$$y_i \sim \mathrm{N}(2 + .5 \times 1, 1) = \mathrm{N}(2.5, 1)$$

which might look like this:



Now we ask: what was the density at the *actual* outcome y_i?



In R, we could go ahead and write a function that returns the density for a given observation. Because we'll be optimizing this, we want to give it the unknowns as a single vector.

```
density_1 <- function(theta, # the unknown parameters
                      x, y) {
  # This function returns the density of a normal with mean theta[1] + theta[2]*x
  # and standard deviation exp(theta[3])
  # evaluated at y.
  dnorm(y, theta[1] + theta[2]*x, exp(theta[3]))
}
```

Now let's evaluate the density at $x = 1$, $y = 1$ for the given likelihood values. Note that we've included the scale (which must be positive) as $\sigma = \exp(\theta_3)$, as it must be positive. If we know θ_3, we can get back to σ by taking its log.

```
density_1(theta = c(2, .5, log(1)),
          x = 1, y = 1)
```

```
## [1] 0.1295176
```

Just as in the binary outcome case, the value of the density at an observed outcome is the *likelihood contribution* $f(y_i|x_i, \alpha, \beta, \sigma)$ of a single datapoint. Unlike the binary case, this has no direct probabilistic interpretation except in relative terms. If the continous density at a given value is twice the density at another, then the relative probability of observing the first value is twice the second. Also, the density needn't be less than one; when the likelihood is very narrow, we routinely see values of the likelihood greater than 1. You'll be able to see that this likelihood contribution is maximized (at infinity) when it imposes a spike on the single observation we observe, ie, $\alpha + \beta \times 1 = 1$ and $\sigma = 0$).

```r
# log(0) -> -Inf
density_1(c(0, 1, -Inf) , 1, 1)
```

```
## [1] Inf
```

But we have more than one observation

We have one observation, three unknowns—of course this is an unidentified model. In reality we have many observations. So how do we use this principle of likelihood to get estimates of the many model unknowns?

If we assume (conditional) independence of the draws–that is, the value of one observation's ϵ_i has no influence on another's ϵ_j, the sample likelihood of your data vector y is the *product* of all the individual likelihoods $\prod_{i=1}^{N} f(y_i|x_i, \alpha, \beta, \sigma)$. This is identical to the binary outcomes likelihood abve.

You can probably see the problem here—for reasonably large spreads of the data, the value of the density for a datapoint is typically less than 1, and the product of many such datapoints will be an extremely small number. And so just as in the binary case, we take the log likelihood of the data.

$$\log\left(\prod_{i=1}^{N} f(y_i|x_i, a, b, s)\right) = \sum_{i=1}^{N} \log(f(y_i|x_i, a, b, s))$$

So what does the likelihood mean?

Log likelihood is a confusing concept to beginners as the values of the numbers aren't easily interpretable. You might run the analysis and get a number of -3477 and ask "is that good? Bad? What does it mean?" These numbers are more meaningful when conducting model comparison, that is, we get the out of sample log likelihood for two different models. Then we can compare them. The usefulness of this approach is best illustrated with an example.

Imagine that we have an outlier—rather than $y_j = 1$ as in the image above, it is -10. Under the proposed model, we're essentially saying that the such an outcome is all but impossible. The probability of observing an observation that low or lower is 3.73256429887771e-36. The density of such a point would be 4.69519535797515e-35—a very small number. But the log of its density is large in absolute value: -79.0439385332047. Compare that with the log likelihood of the original $y_j = 1$: -2.04393853320467. An outlier penalizes the log likelihood far more than a point at the mode, and consequently our maximum likelihood estimate will try to make sense of the outlier by dragging the entire distribution that way.

This idea helps us do good modeling: we want to give positive weight to outcomes that happen, and no weight to impossible outcomes. If in the data visualization step we notice some of these outliers (and they are real data rather than miscoded entries—which are common) then we will want to use distributions that make these possible. Use of these fat-tailed distributions is covered in Chapter 3.

Maximum likelihood

Maximum likelihood estimators are simple: if the (log) likelihood is a unidimensional score of how well the data fit the model for a (potentially large) number of parameters, then we can simply run an optimizer that attempts to maximize this score by varying the values of the parameters. If the optimizer is able to find us the mode of the likelihood, it gives us the maximum likelihood estimator (MLE). This mode needn't be a good estimate of what we want to know. If we have good prior information, the MLE won't capture any of that. And in some cases (as with the hierarchical models illustrated in the second worked example in this chapter) the MLE returns an objectively terrible estimate.

Let's get some practice at writing this out in R. Going through this exercise will help you understand what's going on under the hood in Stan.

Suppose we have a a thousand observations. x is drawn from a standard normal, $\alpha = 1$, $\beta = 2$ and $\sigma = 3$. $y = \alpha + \beta x + \epsilon$, where ϵ has a zero-centered normal distribution with standard deviation (scale) $= \sigma$.

```r
alpha <- 1; beta <- 2; sigma <- 3
x <- rnorm(1000)
y <- rnorm(1000, alpha + beta * x, sigma)
```

Now that we have some fake data, let's write out the log likelihood function, as before. Note that because the optimization function we use *minimizes* the function, we need to return the *negative* of the log likelihood (minimizing the negative is the same as maximizing the likelihood).

```r
negative_log_likelihood <- function(theta, # the unknown parameters
                      x, y) {
  # This function returns the log likelihood of a normal with mean theta[1] + theta[2]*x
  # and standard deviation exp(theta[3])
  # evaluated at a vector y.
  -sum(log(dnorm(y, theta[1] + theta[2]*x, exp(theta[3]))))
}
```

Now we optimize

```r
estimates <- optim(par = c(1,1,1), negative_log_likelihood, x = x, y =y)

estimated_alpha <- estimates$par[1]
estimated_beta <- estimates$par[2]
estimated_sigma <- exp(estimates$par[3])

paste("estimate of alpha is", round(estimated_alpha, 2), "true value is", alpha)
```

```
## [1] "estimate of alpha is 1.12 true value is 1"
```

```r
paste("estimate of beta is", round(estimated_beta, 2), "true value is", beta)
```

```
## [1] "estimate of beta is 2.1 true value is 2"
```

```r
paste("estimate of sigma is", round(estimated_sigma, 2), "true value is", sigma)
```

```
## [1] "estimate of sigma is 2.92 true value is 3"
```

Hey presto! We just estimated a model by maximizing the likelihood by varying the parameters of our model *keeping the data fixed*. You should note that this method can be applied more generally in much larger, more interesting models.

Prior distributions

Prior distributions summarize our information about the values of parameters *before* seeing the data. For the uninitiated, this is the scary bit of building a Bayesian model, often because of fears of being biased, or having a poor understanding of how exactly the parameter influences the model. Such fears are often revealed by the choice of extremely diffuse priors, for instance $\beta \sim \text{Normal}(0, 1000)$.

Don't be afraid of using priors. You almost always have high quality information about the values parameters might take on. For example:

- Estimates from previous studies
- Some unknowns have sign restrictions (fixed costs or discount rates probably aren't negative; price elasticities of demand probably aren't positive)
- The knowledge that regularization can help prevent over-fitting
- The scale of effects are probably known. Going to college probably won't increase your income 100000%.

Prior distributions should be such that they put positive probabilistic weight on possible outcomes, and no weight on impossible outcomes. In the example above, the standard deviation of the residuals, σ must be

positive. And so its prior should not have probabilistic weight below 0. That might guide the choice to a distribution like a half-Normal, half-Cauchy, half-Student's t, inverse Gamma, lognormal etc.

Bayes rule

Bayes rule gives us a method for combining the information from our data (the likelihood) and our priors. It says that the (joint) probability density of our parameter vector θ is

$$\mathrm{p}(\theta|\, y) = \frac{\mathrm{p}(y|\, \theta)\, \mathrm{p}(\theta)}{\mathrm{p}(y)}$$

where $\mathrm{p}(y|\, \theta)$ is the likelihood, and $\mathrm{p}(\theta)$ is the prior. We call $\mathrm{p}(\theta|\, y)$ the posterior. Because $\mathrm{p}(y)$ doesn't depend on the vector of unknowns, θ, we often express the posterior up to a constant of proportionality

$$\mathrm{p}(\theta|\, y) \propto \mathrm{p}(y|\, \theta)\, \mathrm{p}(\theta)$$

What can you take away from this? A few things:

- If the prior or likelihood is equal to zero for a given value of θ, so too will be the posterior
- If the prior is very peaked, the posterior well be drawn towards the prior
- If the likelihood is very peaked (as tends to happen when you have many observations per unknown), the posterior will be drawn towards the likelihood estimate.

Bayes' rule also tells us that in order to obtain a posterior, we need to have a prior and a likelihood.

Using penalized likelihood estimation

Just as we can estimate a model using maximum likelihood, we can also estimate the *mode* of the posterior using *penalized likelihood* estimation. This is a really great way to start thinking about Bayesian estimation.

To estimate a model using penalized likelihood, we simply need to recognize that if

$$\mathrm{p}(\theta|\, y) \propto \mathrm{p}(y|\, \theta)\, \mathrm{p}(\theta)$$

then

$$\log(\mathrm{p}(\theta|\, y)) \propto log(\mathrm{p}(y|\, \theta)) + \log(\mathrm{p}(\theta))$$

That is, our log posterior density is proportional to the log likelihood plus the log prior density evaluated at a given value of θ. So we can choose θ to maximize our posterior by choosing a θ that maximizes the right hand side.

We do this in the function below. I've been verbose in my implementation to make it very clear what's going on. The `target` is proportional to the accumulated log posterior density. To maximize the log posterior density, we minimize the negative target. We'll choose priors

$$\alpha,\, \beta \sim \mathrm{N}(0, 1) \text{ and } \sigma \sim \mathrm{N}_{+}(0, 1)$$

```
negative_penalized_log_likelihood <- function(theta, # the unknown parameters
                      x, y) {
  # This function returns a value proportional to the
  # log posterior density of a normal with mean theta[1] + theta[2]*x
  # and standard deviation exp(theta[3])
```

```
  # evaluated at a vector y.

  # Initialize the target
  target <- 0

  # Add the density of the priors to the accumulator
  # Add the log density of the scale prior at the value (truncated normal)
  target <- target + log(truncnorm::dtruncnorm(exp(theta[3]), a = 0))
  # Add the log density of the intercept prior at the parameter value
  target <- target + log(dnorm(theta[1]))
  # Add the log density of the slope prior at the parameter value
  target <- target + log(dnorm(theta[2]))
  # Add the Log likelihood
  target <- target + sum(log(dnorm(y, theta[1] + theta[2]*x, exp(theta[3]))))

  # Return the negative target
  -target
}
```

Now we optimize

```
estimates <- optim(par = c(1,1,1), negative_penalized_log_likelihood, x = x, y =y)

estimated_alpha <- estimates$par[1]
estimated_beta <- estimates$par[2]
estimated_sigma <- exp(estimates$par[3])

paste("estimate of alpha is", round(estimated_alpha, 2), "true value is", alpha)
```

```
## [1] "estimate of alpha is 1.11 true value is 1"
```

```
paste("estimate of beta is", round(estimated_beta, 2), "true value is", beta)
```

```
## [1] "estimate of beta is 2.09 true value is 2"
```

```
paste("estimate of sigma is", round(estimated_sigma, 2), "true value is", sigma)
```

```
## [1] "estimate of sigma is 2.9 true value is 3"
```

And there you go! You've estimated your very first (very simple) model using penalized likelihood.

Before you go on and get too enthusiastic about estimating everything with penalized likelihood, be warned: the modal estimator will do fine in fairly low dimensions, and with many fairly simple models. But as your models become more complex you'll find that it does not do a good job. This is partly because the mode of a high-dimensional distribution can be a long way from its expected value (which is what you really care about). But also because you often want to generate predictions that take into account the uncertainty you have in your model's unknowns when making predictions. For this, you'll need to *draw samples* from your posterior.

We'll not get there just yet—that's for another post. But just first, let's take a quick look at what a posterior really looks like in practical terms.

What does a posterior look like?

In Bayesian inference we do not get point estimates for the unknowns in our models; we estimate their posterior distributions. Ideally, we'd want to be able to make posterior inference by asking questions like "what is the 5th percentile of the marginal posterior of θ_1?". This would require that we can analytically

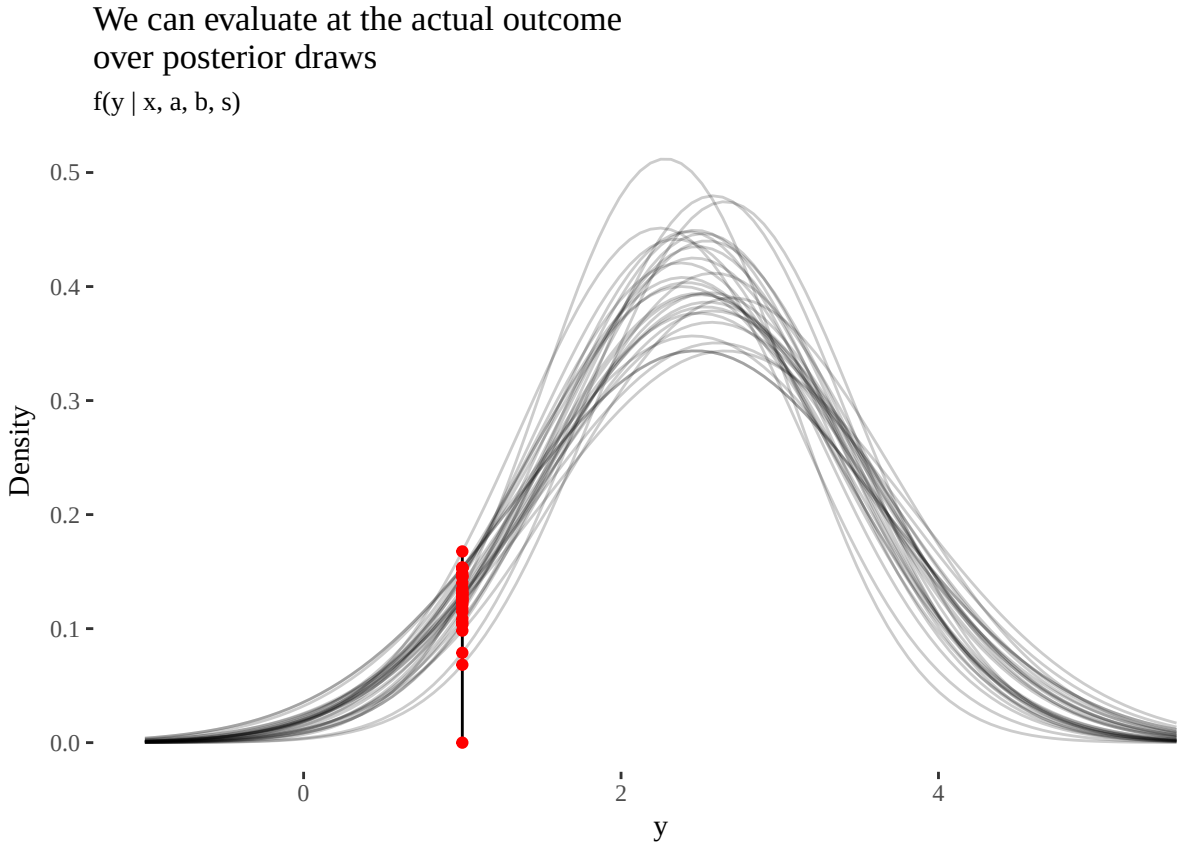
derive these statistics from the posterior. Sadly, most posteriors do not have a closed form, and so we need to approximate. The two common types of approximation are:

1. To approximate the posterior with a joint density for which we can analytically evaluate quantiles, expected values etc. This is the approach in Variational Bayes and penalized likelihood.
2. To obtain many independent draws from the posterior, and evaluate quantiles, expected values of those draws. This is the approach in MCMC and ABC.

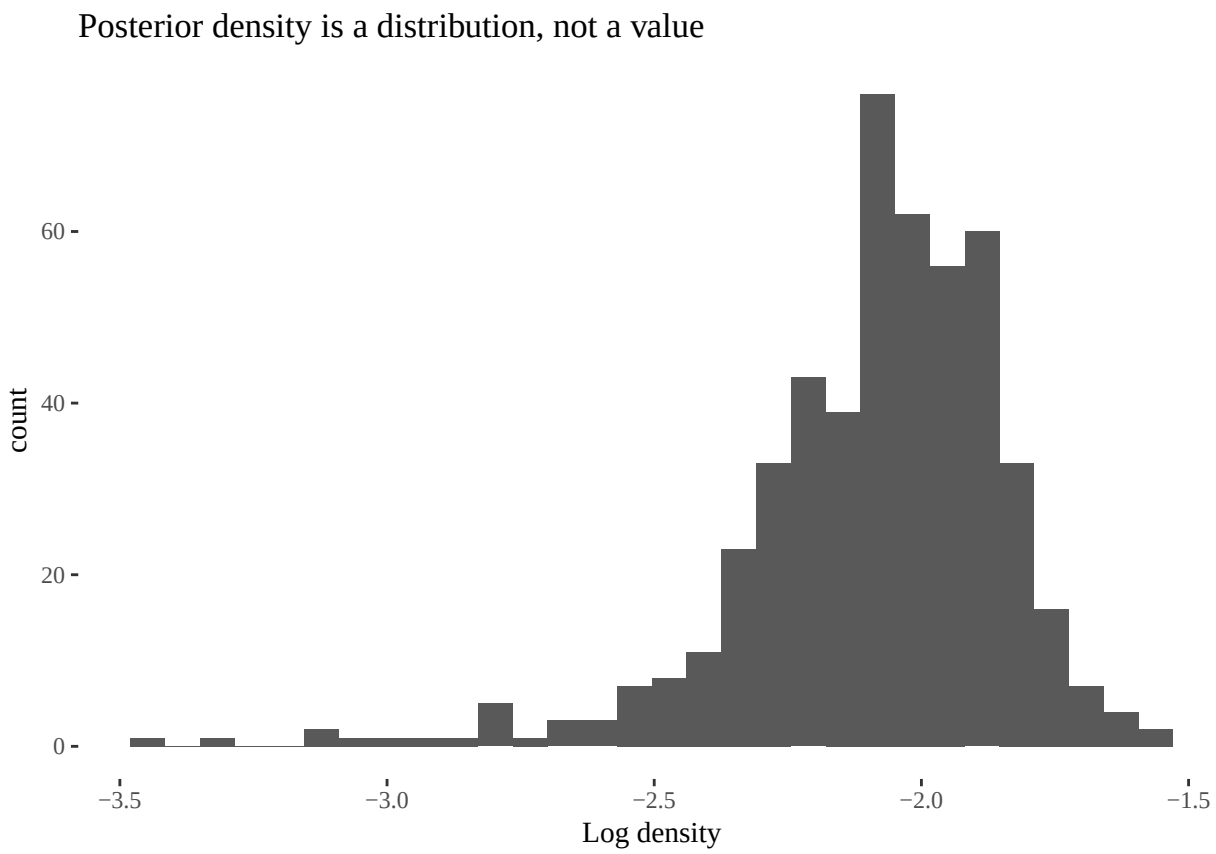
In theory, our posterior $p(\theta|y)$ is an abstract distribution; in practice (when using MCMC), it's a matrix of data, where each column corresponds to a parameter, and each row a draw from $p(\theta|y)$. For instance, the first five rows of our posterior matrix for our linear model above might look like:

a	b	sigma
2.03	0.50	0.84
2.09	0.49	0.96
1.95	0.51	1.17
2.00	0.47	0.92
2.11	0.59	0.93

Each of these parameter combinations implies a different predictive distribution. Consequently, they also imply different predictive likelihoods for our out of sample datapoint $y_j = 1, x_j = 1$. This is illustrated for 30 posterior replicates below.



The density of the data across posterior draws is known as the *log posterior density* of a datapoint. Unlike the likelihood, we are uncertain of its value and need to perform inference. To illustrate this, let's look at a histogram of the log posterior density in the above example (but with 500 replicates, not 30):



Note that the log posterior density can be evaluated *for each datapoint*; if we have many datapoints for which we want to evaluate the LPD we simply need to evaluate it for each, and sum across all datapoints.

Bayes rule

- What is Bayes Rule
- Approximating a distribution with draws
- Markov Chain Monte Carlo
- The Monte Carlo standard error
- Fitting a model with MCMC in Stan
- Fitting a model with penalized likelihood in Stan
- What do posterior draws look like? What can I do with them?

Choosing priors

- Priors help us to encode information about population-level knowledge that is absent from our sample-level data
- They can help to regularize richly-parameterized models
- Hierarchical models (or the "partial pooling" approach) lets us fit models that would otherwise be extremely noisy and get far more precise estimates in practice

Prior predictive analysis

- In order to choose priors; take our data
- Draw from the prior
- Draw from the distribution corresponding to the likelihood

- Are the outcomes at all plausible?
- Are they explosive?
- Do they assign probabilistic weight to impossible outcomes?
- Do they imply economically implausible outcomes? Such as market shares of 1, or negative prices?
- Do they interact with the scale of the input data?

A Modern Statistical Workflow

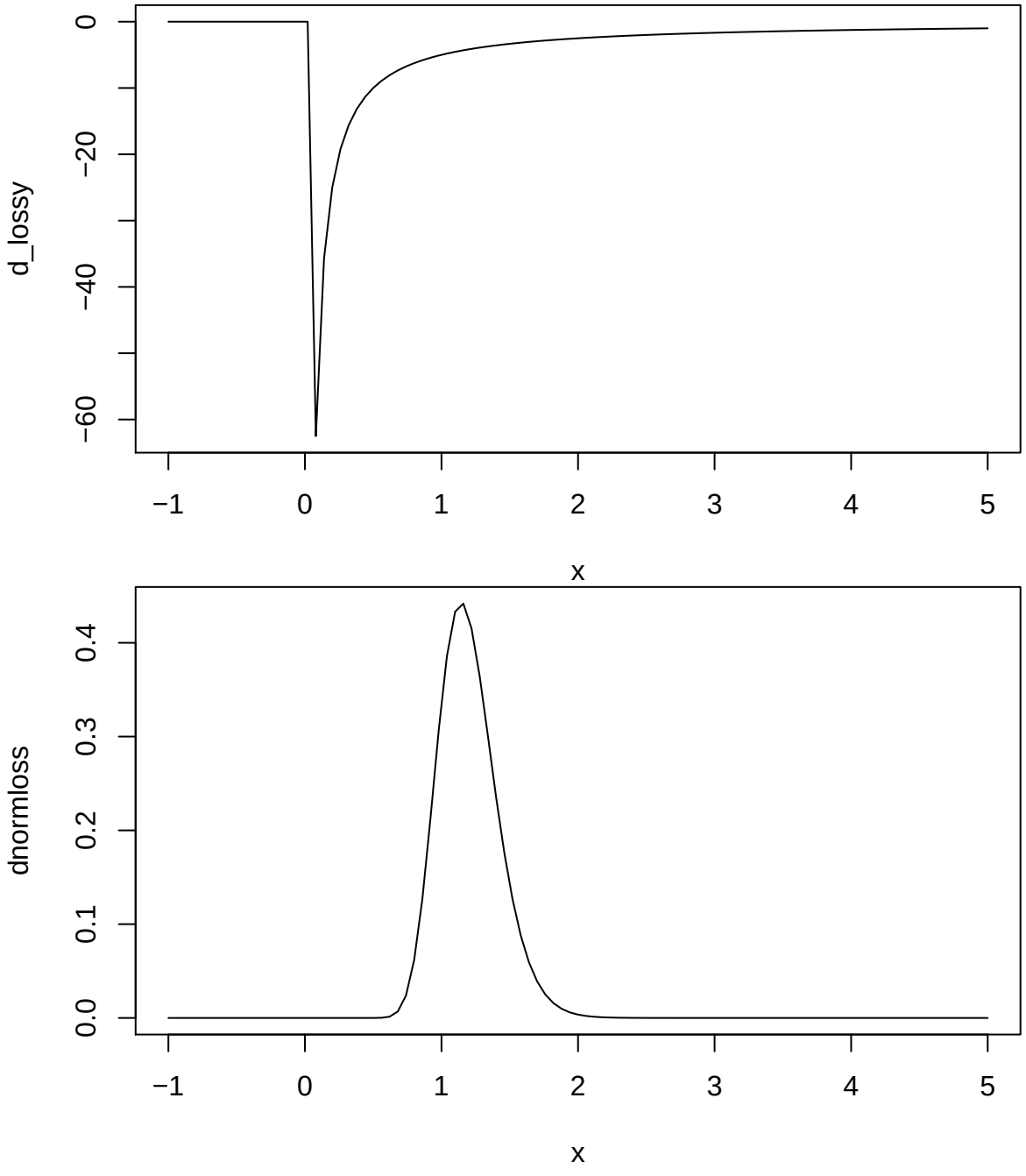
Why use this workflow? Use when

- You are building complex models and need to make sure that your software is correct
- You want to learn about the model (and especially the impact of priors) more deeply

The workflow

- Plot your data and make notes of it
- Outcome variables
- Identify missing values
- Correlations between explanators
- Non-linearities
- Interactions
- Outliers
- Scale
- Write down the likelihood
- Choose based on conditional distribution of the data and theory
- Choose priors for the unknowns
- We want priors that a) encode outside information, improve the geometry of the posterior
- In practice this means choosing priors that look like normal(0, 1) and reparameterizing the model so that this makes sense
- Draw from the priors (with example of how to do this in Stan)
- Simulate fake data given your "knowns"
- Fit the model to the fake data
- Check the fit
- Chains converged
- Rhat close to 1?
- Divergent transitions
- Energy
- Shinystan
- Reparameterize the model if you need to, and fit again
- Evaluate fit: check parameter estimates against those used to generate the data
- Scatter plots of known vs estimates
- Coverage

- Order statistics
- Take to real data
- Evaluate fit
- Posterior predictive checking
- Evaluate qualitative findings
- Discuss with a critical audience
- Make the model richer and repeat
- Apply decision theory to the estimate



```
## [1] -1.249996
```

```
## [1] 0.2445242
```