STA 578 - Statistical Computing Notes

Derek Sonderegger

2017-09-02

Contents

Preface

This is some stuff to say first.

Chapter 1

Data Manipulation

Most of the time, our data is in the form of a data frame and we are interested in exploring the relationships. This chapter explores how to manipulate data frames and methods.

1.1 Classical functions for summarizing rows and columns

1.1.1 `summary()`

The first method is to calculate some basic summary statistics (minimum, 25th, 50th, 75th percentiles, maximum and mean) of each column. If a column is categorical, the summary function will return the number of observations in each category.

```
# use the iris data set which has both numerical and categorical variables
data( iris )
str(iris)    # recall what columns we have
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
# display the summary for each column
summary( iris )
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
```

##

1.1.2 apply()

The summary function is convenient, but we want the ability to pick another function to apply to each column and possibly to each row. To demonstrate this, suppose we have data frame that contains students grades over the semester.

```
# make up some data
grades <- data.frame(
  l.name = c('Cox', 'Dorian', 'Kelso', 'Turk'),
  Exam1 = c(93, 89, 80, 70),
  Exam2 = c(98, 70, 82, 85),
  Final = c(96, 85, 81, 92) )
```

The `apply()` function will apply an arbitrary function to each row (or column) of a matrix or a data frame and then aggregate the results into a vector.

```
# Because I can't take the mean of the last names column,
# remove the name column
scores <- grades[,-1]
scores
```

```
##   Exam1 Exam2 Final
## 1    93    98    96
## 2    89    70    85
## 3    80    82    81
## 4    70    85    92
```

```
# Summarize each column by calculating the mean.
apply( scores,      # what object do I want to apply the function to
       MARGIN=2,    # rows = 1, columns = 2, (same order as [rows, cols]
       FUN=mean     # what function do we want to apply
     )
```

```
## Exam1 Exam2 Final
## 83.00 83.75 88.50
```

To apply a function to the rows, we just change which margin we want. We might want to calculate the average exam score for person.

```
apply( scores,      # what object do I want to apply the function to
       MARGIN=1,    # rows = 1, columns = 2, (same order as [rows, cols]
       FUN=mean     # what function do we want to apply
     )
```

```
## [1] 95.66667 81.33333 81.00000 82.33333
```

This is useful, but it would be more useful to concatenate this as a new column in my grades data frame.

```
average <- apply(
  scores,      # what object do I want to apply the function to
  MARGIN=1,    # rows = 1, columns = 2, (same order as [rows, cols]
  FUN=mean     # what function do we want to apply
)
grades <- cbind( grades, average ) # squish together
grades
```

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

There are several variants of the `apply()` function, and the variant I use most often is the function `sapply()`, which will apply a function to each element of a list or vector and returns a corresponding list or vector of results.

1.2 Package `dplyr`

```
library(dplyr)   # load the dplyr package!
```

Many of the tools to manipulate data frames in R were written without a consistent syntax and are difficult use together. To remedy this, Hadley Wickham (the writer of `ggplot2`) introduced a package called plyr which was quite useful. As with many projects, his first version was good but not great and he introduced an improved version that works exclusively with data.frames called `dplyr` which we will investigate. The package `dplyr` strives to provide a convenient and consistent set of functions to handle the most common data frame manipulations and a mechanism for chaining these operations together to perform complex tasks.

The Dr Wickham has put together a very nice introduction to the package that explains in more detail how the various pieces work and I encourage you to read it at some point. [http://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html].

One of the aspects about the `data.frame` object is that R does some simplification for you, but it does not do it in a consistent manner. Somewhat obnoxiously character strings are always converted to factors and subsetting might return a `data.frame` or a `vector` or a `scalar`. This is fine at the command line, but can be problematic when programming. Furthermore, many operations are pretty slow using `data.frame`. To get around this, Dr Wickham introduced a modified version of the `data.frame` called a `tibble`. A `tibble` is a `data.frame` but with a few extra bits. For now we can ignore the differences.

The pipe command `%>%` allows for very readable code. The idea is that the `%>%` operator works by translating the command `a %>% f(b)` to the expression `f(a,b)`. This operator works on any function and was introduced in the `magrittr` package. The beauty of this comes when you have a suite of functions that takes input arguments of the same type as their output.

For example if we wanted to start with `x`, and first apply function `f()`, then `g()`, and then `h()`, the usual R command would be `h(g(f(x)))` which is hard to read because you have to start reading at the innermost set of parentheses. Using the pipe command `%>%`, this sequence of operations becomes `x %>% f() %>% g() %>% h()`.

Dr Wickham gave the following example of readability:

```
bopping(
  scooping_up(
    hopping_through(foo_foo),
    field_mice),
  head)
```

is more readably written:

```
foo_foo %>%
  hopping_through(forest) %>%
  scooping_up( field_mice) %>%
  bopping( head )
```

In `dplyr`, all the functions below take a data set as its first argument and outputs an appropriately modified data set. This will allow me to chain together commands in a readable fashion.

1.2.1 Verbs

The foundational operations to perform on a data set are:

- Subsetting - Returns a with only particular columns or rows
  - `select` - Selecting a subset of columns by name or column number.
  - `filter` - Selecting a subset of rows from a data frame based on logical expressions.
  - `slice` - Selecting a subset of rows by row number.
- `arrange` - Re-ordering the rows of a data frame.
- `mutate` - Add a new column that is some function of other columns.
- `summarise` - calculate some summary statistic of a column of data. This collapses a set of rows into a single row.

Each of these operations is a function in the package `dplyr`. These functions all have a similar calling syntax, the first argument is a data set, subsequent arguments describe what to do with the input data frame and you can refer to the columns without using the `df$column` notation. All of these functions will return a data set.

1.2.1.1 Subsetting with `select`, `filter`, and `slice`

These function allows you select certain columns and rows of a data frame.

1.2.1.1.1 `select()`

Often you only want to work with a small number of columns of a data frame. It is relatively easy to do this using the standard `[,col.name]` notation, but is often pretty tedious.

```
# recall what the grades are
grades
```

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

I could select the columns Exam columns by hand, or by using an extension of the `:` operator

```
grades %>% select( Exam1, Exam2 )   # Exam1 and Exam2
```

```
##   Exam1 Exam2
## 1    93    98
## 2    89    70
## 3    80    82
## 4    70    85
```

```
grades %>% select( Exam1:Final )    # Columns Exam1 through Final
```

```
##   Exam1 Exam2 Final
## 1    93    98    96
## 2    89    70    85
## 3    80    82    81
## 4    70    85    92
```

```
grades %>% select( -Exam1 )          # Negative indexing by name works
```

```
##   l.name Exam2 Final  average
## 1    Cox    98    96 95.66667
## 2 Dorian    70    85 81.33333
## 3  Kelso    82    81 81.00000
## 4   Turk    85    92 82.33333
```

```
grades %>% select( 1:2 )             # Can select column by column position
```

```
##   l.name Exam1
## 1    Cox    93
## 2 Dorian    89
## 3  Kelso    80
## 4   Turk    70
```

The `select()` command has a few other tricks. There are functional calls that describe the columns you wish to select that take advantage of pattern matching. I generally can get by with `starts_with()`, `ends_with()`, and `contains()`, but there is a final operator `matches()` that takes a regular expression.

```
grades %>% select( starts_with('Exam') )    # Exam1 and Exam2
```

```
##   Exam1 Exam2
## 1    93    98
## 2    89    70
## 3    80    82
## 4    70    85
```

1.2.1.1.2 `filter()`

It is common to want to select particular rows where we have some logically expression to pick the rows.

```
# select students with Final grades greater than 90
grades %>% filter(Final > 90)
```

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2   Turk    70    85    92 82.33333
```

You can have multiple logical expressions to select rows and they will be logically combined so that only rows that satisfy all of the conditions are selected. The logicals are joined together using `&` (and) operator or the `|` (or) operator and you may explicitly use other logicals. For example a factor column type might be used to select rows where type is either one or two via the following: `type==1 | type==2`.

```
# select students with Final grades above 90 and
# average score also above 90
grades %>% filter(Final > 90, average > 90)
```

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
```

```
# we could also use an "and" condition
grades %>% filter(Final > 90 & average > 90)
```

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
```

1.2.1.1.3 slice()

When you want to filter rows based on row number, this is called slicing.

```
# grab the first 2 rows
grades %>% slice(1:2)
```

```
## # A tibble: 2 x 5
##   l.name Exam1 Exam2 Final  average
##   <fctr> <dbl> <dbl> <dbl>    <dbl>
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
```

1.2.1.2 arrange()

We often need to re-order the rows of a data frame. For example, we might wish to take our grade book and sort the rows by the average score, or perhaps alphabetically. The `arrange()` function does exactly that. The first argument is the data frame to re-order, and the subsequent arguments are the columns to sort on. The order of the sorting column determines the precedent… the first sorting column is first used and the second sorting column is only used to break ties.

```
grades %>% arrange(l.name)
```

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

The default sorting is in ascending order, so to sort the grades with the highest scoring person in the first row, we must tell arrange to do it in descending order using `desc(column.name)`.

```
grades %>% arrange(desc(Final))
```

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2   Turk    70    85    92 82.33333
## 3 Dorian    89    70    85 81.33333
## 4  Kelso    80    82    81 81.00000
```

In a more complicated example, consider the following data and we want to order it first by Treatment Level and secondarily by the y-value. I want the Treatment level in the default ascending order (Low, Medium, High), but the y variable in descending order.

```
# make some data
dd <- data.frame(
  Trt = factor(c("High", "Med", "High", "Low"),
               levels = c("Low", "Med", "High")),
  y = c(8, 3, 9, 9),
  z = c(1, 1, 1, 2))
dd
```

```
##    Trt y z
## 1 High 8 1
```

```
## 2  Med 3 1
## 3 High 9 1
## 4  Low 9 2
```

```
# arrange the rows first by treatment, and then by y (y in descending order)
dd %>% arrange(Trt, desc(y))
```

```
##    Trt y z
## 1  Low 9 2
## 2  Med 3 1
## 3 High 9 1
## 4 High 8 1
```

1.2.1.3 mutate()

I often need to create a new column that is some function of the old columns. This was often cumbersome. Consider code to calculate the average grade in my grade book example.

```
grades$average <- (grades$Exam1 + grades$Exam2 + grades$Final) / 3
```

Instead, we could use the `mutate()` function and avoid all the `grades$` nonsense.

```
grades %>% mutate( average = (Exam1 + Exam2 + Final)/3 )
```

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

You can do multiple calculations within the same `mutate()` command, and you can even refer to columns that were created in the same `mutate()` command.

```
grades %>% mutate(
  average = (Exam1 + Exam2 + Final)/3,
  grade = cut(average, c(0, 60, 70, 80, 90, 100),  # cut takes numeric variable
                       c( 'F','D','C','B','A')) )  # and makes a factor
```

```
##   l.name Exam1 Exam2 Final  average grade
## 1    Cox    93    98    96 95.66667     A
## 2 Dorian    89    70    85 81.33333     B
## 3  Kelso    80    82    81 81.00000     B
## 4   Turk    70    85    92 82.33333     B
```

1.2.1.4 summarise()

By itself, this function is quite boring, but will become useful later on. Its purpose is to calculate summary statistics using any or all of the data columns. Notice that we get to chose the name of the new column. The way to think about this is that we are collapsing information stored in multiple rows into a single row of values.

```
# calculate the mean of exam 1
grades %>% summarise( mean.E1=mean(Exam1))
```

```
##   mean.E1
## 1      83
```

We could calculate multiple summary statistics if we like.

```
# calculate the mean and standard deviation
grades %>% summarise( mean.E1=mean(Exam1), stddev.E1=sd(Exam2) )
```

```
##   mean.E1 stddev.E1
## 1      83      11.5
```

If we want to apply the same statistic to each column, we use the `summarise_each()` command. We have to be a little careful here because the function you use has to work on every column (that isn't part of the grouping structure (see `group_by()`)).

```
# calculate the mean and stddev of each column - Cannot do this to Names!
grades %>%
  select( Exam1:Final ) %>%
  summarise_each( funs(mean, sd) )
```

```
## `summarise_each()` is deprecated.
## Use `summarise_all()`, `summarise_at()` or `summarise_if()` instead.
## To map `funs` over all variables, use `summarise_all()`
```

```
##   Exam1_mean Exam2_mean Final_mean Exam1_sd Exam2_sd Final_sd
## 1         83      83.75       88.5 10.23067     11.5 6.757712
```

1.2.1.5 Miscellaneous functions

There are some more function that are useful but aren't as commonly used. For sampling the functions `sample_n()` and `sample_frac()` will take a subsample of either n rows or of a fraction of the data set. The function `n()` returns the number of rows in the data set. Finally `rename()` will rename a selected column.

1.2.2 Split, apply, combine

Aside from unifying the syntax behind the common operations, the major strength of the `dplyr` package is the ability to split a data frame into a bunch of sub-dataframes, apply a sequence of one or more of the operations we just described, and then combine results back together. We'll consider data from an experiment from spinning wool into yarn. This experiment considered two different types of wool (A or B) and three different levels of tension on the thread. The response variable is the number of breaks in the resulting yarn. For each of the 6 `wool:tension` combinations, there are 9 replicated observations per `wool:tension` level.

```
data(warpbreaks)
str(warpbreaks)
```

```
## 'data.frame':    54 obs. of  3 variables:
##  $ breaks : num  26 30 54 25 70 52 51 26 67 18 ...
##  $ wool   : Factor w/ 2 levels "A","B": 1 1 1 1 1 1 1 1 1 1 ...
##  $ tension: Factor w/ 3 levels "L","M","H": 1 1 1 1 1 1 1 1 1 2 ...
```

The first we must do is to create a data frame with additional information about how to break the data into sub-dataframes. In this case, I want to break the data up into the 6 wool-by-tension combinations. Initially we will just figure out how many rows are in each wool-by-tension combination.

```
# group_by:  what variable(s) shall we group one
# n() is a function that returns how many rows are in the
#   currently selected sub-dataframe
warpbreaks %>%
  group_by( wool, tension) %>%    # grouping
  summarise(n = n() )             # how many in each group
```

```
## # A tibble: 6 x 3
## # Groups:   wool [?]
##     wool tension     n
##   <fctr>  <fctr> <int>
## 1      A       L     9
## 2      A       M     9
## 3      A       H     9
## 4      B       L     9
## 5      B       M     9
## 6      B       H     9
```

The `group_by` function takes a data.frame and returns the same data.frame, but with some extra information so that any subsequent function acts on each unique combination defined in the `group_by`. If you wish to remove this behavior, use `group_by()` to reset the grouping to have no grouping variable.

Using the same `summarise` function, we could calculate the group mean and standard deviation for each wool-by-tension group.

```
warpbreaks %>%
  group_by(wool, tension) %>%
  summarise( n           = n(),             # I added some formatting to show the
             mean.breaks = mean(breaks),    # reader I am calculating several
             sd.breaks   = sd(breaks))      # statistics.
```

```
## # A tibble: 6 x 5
## # Groups:   wool [?]
##     wool tension     n mean.breaks sd.breaks
##   <fctr>  <fctr> <int>       <dbl>     <dbl>
## 1      A       L     9    44.55556 18.097729
## 2      A       M     9    24.00000  8.660254
## 3      A       H     9    24.55556 10.272671
## 4      B       L     9    28.22222  9.858724
## 5      B       M     9    28.77778  9.431036
## 6      B       H     9    18.77778  4.893306
```

If instead of summarizing each split, we might want to just do some calculation and the output should have the same number of rows as the input data frame. In this case I'll tell `dplyr` that we are mutating the data frame instead of summarizing it. For example, suppose that I want to calculate the residual value

$$e_{ijk} = y_{ijk} - \bar{y}_{ij\cdot}$$

where $\bar{y}_{ij\cdot}$ is the mean of each `wool:tension` combination.

```
warpbreaks %>%
   group_by(wool, tension) %>%                 # group by wool:tension
   mutate(resid = breaks - mean(breaks)) %>%   # mean(breaks) of the group!
   head(  )                                    # show the first couple of rows
```

```
## # A tibble: 6 x 4
## # Groups:   wool, tension [1]
##   breaks   wool tension      resid
##    <dbl> <fctr>  <fctr>      <dbl>
## 1     26      A       L -18.555556
## 2     30      A       L -14.555556
## 3     54      A       L   9.444444
## 4     25      A       L -19.555556
## 5     70      A       L  25.444444
## 6     52      A       L   7.444444
```

1.2.3 Chaining commands together

In the previous examples we have used the %>% operator to make the code more readable but to really appreciate this, we should examine the alternative.

Suppose we have the results of a small 5K race. The data given to us is in the order that the runners signed up but we want to calculate the results for each gender, calculate the placings, and the sort the data frame by gender and then place. We can think of this process as having three steps:

1. Splitting
2. Ranking
3. Re-arranging.

```
# input the initial data
race.results <- data.frame(
  name=c('Bob', 'Jeff', 'Rachel', 'Bonnie', 'Derek', 'April','Elise','David'),
  time=c(21.23, 19.51, 19.82, 23.45, 20.23, 24.22, 28.83, 15.73),
  gender=c('M','M','F','F','M','F','F','M'))
```

We could run all the commands together using the following code:

```
arrange(
  mutate(
    group_by(
      race.results,          # using race.results
      gender),               # group by gender
    place = rank( time )),   # mutate to calculate the place column
  gender, place)             # arrange the result by gender and place
```

```
## # A tibble: 8 x 4
## # Groups:   gender [2]
##     name  time gender place
##   <fctr> <dbl> <fctr> <dbl>
## 1 Rachel 19.82      F     1
## 2 Bonnie 23.45      F     2
## 3  April 24.22      F     3
## 4  Elise 28.83      F     4
## 5  David 15.73      M     1
## 6   Jeff 19.51      M     2
## 7  Derek 20.23      M     3
## 8    Bob 21.23      M     4
```

This is very difficult to read because you have to read the code *from the inside out.*

Another (and slightly more readable) way to complete our task is to save each intermediate step of our process and then use that in the next step:

```
temp.df0 <- race.results %>% group_by( gender)
temp.df1 <- temp.df0 %>% mutate( place = rank(time) )
temp.df2 <- temp.df1 %>% arrange( gender, place )
```

It would be nice if I didn't have to save all these intermediate results because keeping track of temp1 and temp2 gets pretty annoying if I keep changing the order of how things or calculated or add/subtract steps. This is exactly what %>% does for me.

```
race.results %>%
  group_by( gender ) %>%
  mutate( place = rank(time)) %>%
  arrange( gender, place )
```

```
## # A tibble: 8 x 4
## # Groups:   gender [2]
##     name  time gender place
##   <fctr> <dbl> <fctr> <dbl>
## 1 Rachel 19.82      F     1
## 2 Bonnie 23.45      F     2
## 3  April 24.22      F     3
## 4  Elise 28.83      F     4
## 5  David 15.73      M     1
## 6   Jeff 19.51      M     2
## 7  Derek 20.23      M     3
## 8    Bob 21.23      M     4
```

1.3 Exercises

1. The dataset `ChickWeight` tracks the weights of 48 baby chickens (chicks) feed four different diets.
   a. Load the dataset using

   ```
   data(ChickWeight)
   ```

   b. Look at the help files for the description of the columns.
   c) Remove all the observations except for the weights on day 10 and day 20.
   d) Calculate the mean and standard deviation for each diet group on days 10 and 20.
2. The OpenIntro textbook on statistics includes a data set on body dimensions.
   a) Load the file using

   ```
   Body <- read.csv('http://www.openintro.org/stat/data/bdims.csv')
   ```

   b) The column sex is coded as a 1 if the individual is male and 0 if female. This is a non-intuitive labeling system. Create a new column `sex.MF` that uses labels Male and Female.
   c) The columns wgt and hgt measure weight and height in kilograms and centimeters (respectively). Use these to calculate the Body Mass Index (BMI) for each individual where

$$BMI = \frac{Weight\,(kg)}{[Height\,(m)]^2}$$

   d) Double check that your calculated BMI column is correct by examining the summary statistics of the column. BMI values should be between 18 to 40 or so. Did you make an error in your calculation?
   e) The function `cut` takes a vector of continuous numerical data and creates a factor based on your give cut-points.

   ```
   # Define a continuous vector to convert to a factor
   x <- 1:10

   # divide range of x into three groups of equal length
   cut(x, breaks=3)
   ```

   ```
   ##  [1] (0.991,4] (0.991,4] (0.991,4] (0.991,4] (4,7]     (4,7]     (4,7]
   ##  [8] (7,10]    (7,10]    (7,10]
   ## Levels: (0.991,4] (4,7] (7,10]
   ```

   ```
   # divide x into four groups, where I specify all 5 break points
   cut(x, breaks = c(0, 2.5, 5.0, 7.5, 10))
   ```

   ```
   ##  [1] (0,2.5]  (0,2.5]  (2.5,5]  (2.5,5]  (2.5,5]  (5,7.5]  (5,7.5]
   ##  [8] (7.5,10] (7.5,10] (7.5,10]
   ```

```
## Levels: (0,2.5] (2.5,5] (5,7.5] (7.5,10]
```

```
# (0,2.5] (2.5,5] means 2.5 is included in first group
# right=FALSE changes this to make 2.5 included in the second

# divide x into 3 groups, but give them a nicer
# set of group names
cut(x, breaks=3, labels=c('Low','Medium','High'))
```

```
##  [1] Low    Low    Low    Low    Medium Medium Medium High   High   High
## Levels: Low Medium High
```

Create a new column of in the data frame that divides the age into decades (10-19, 20-29, 30-39, etc). Notice the oldest person in the study is 67.

```
Body <- Body %>%
  mutate( Age.Grp = cut(age,
                        breaks=c(10,20,30,40,50,60,70),
                        right=FALSE))
```

f) Find the average BMI for each Sex and Age group.

Chapter 2

Data Reshaping

```
library(tidyr)   # for the gather/spread commands
library(dplyr)   # for the join stuff
```

Most of the time, our data is in the form of a data frame and we are interested in exploring the relationships. However most procedures in R expect the data to show up in a 'long' format where each row is an observation and each column is a covariate. In practice, the data is often not stored like that and the data comes to us with repeated observations included on a single row. This is often done as a memory saving technique or because there is some structure in the data that makes the 'wide' format attractive. As a result, we need a way to convert data from 'wide' to 'long' and vice-versa.

2.1 `tidyr`

There is a common issue with obtaining data with many columns that you wish were organized as rows. For example, I might have data in a grade book that has several homework scores and I'd like to produce a nice graph that has assignment number on the x-axis and score on the y-axis. Unfortunately this is incredibly hard to do when the data is arranged in the following way:

```
grade.book <- rbind(
  data.frame(name='Alison',  HW.1=8, HW.2=5, HW.3=8),
  data.frame(name='Brandon', HW.1=5, HW.2=3, HW.3=6),
  data.frame(name='Charles', HW.1=9, HW.2=7, HW.3=9))
grade.book
```

```
##      name HW.1 HW.2 HW.3
## 1  Alison    8    5    8
## 2 Brandon    5    3    6
## 3 Charles    9    7    9
```

What we want to do is turn this data frame from a *wide* data frame into a *long* data frame. In MS Excel this is called pivoting. Essentially I'd like to create a data frame with three columns: `name`, `assignment`, and `score`. That is to say that each homework datum really has three pieces of information: who it came from, which homework it was, and what the score was. It doesn't conceptually matter if I store it as 3 columns or 3 rows so long as there is a way to identify how a student scored on a particular homework. So we want to reshape the HW1 to HW3 columns into two columns (assignment and score).

This package was built by the sample people that created dplyr and ggplot2 and there is a nice introduction at: [http://blog.rstudio.org/2014/07/22/introducing-tidyr/]

2.1.1 Verbs

As with the dplyr package, there are two main verbs to remember:

1. `gather` - Gather multiple columns that are related into two columns that contain the original column name and the value. For example for columns HW1, HW2, HW3 we would gather them into two column HomeworkNumber and Score. In this case, we refer to HomeworkNumber as the key column and Score as the value column. So for any key:value pair you know everything you need.
2. `spread` - This is the opposite of gather. This takes a key column (or columns) and a results column and forms a new column for each level of the key column(s).

```
# first we gather the score columns into columns we'll name Assesment and Score
tidy.scores <- grade.book %>%
  gather( key=Assessement,  # What should I call the key column
          value=Score,      # What should I call the values column
          HW.1:HW.3         # which columns to apply this to
          )
tidy.scores
```

```
##      name Assessement Score
## 1  Alison        HW.1     8
## 2 Brandon        HW.1     5
## 3 Charles        HW.1     9
## 4  Alison        HW.2     5
## 5 Brandon        HW.2     3
## 6 Charles        HW.2     7
## 7  Alison        HW.3     8
## 8 Brandon        HW.3     6
## 9 Charles        HW.3     9
```

To spread the key:value pairs out into a matrix, we use the `spread()` command.

```
# Turn the Assessment/Score pair of columns into one column per factor level of Assessment
tidy.scores %>% spread( key=Assessement, value=Score )
```

```
##      name HW.1 HW.2 HW.3
## 1  Alison    8    5    8
## 2 Brandon    5    3    6
## 3 Charles    9    7    9
```

One way to keep straight which is the `key` column is that the key is the category, while `value` is the numerical value or response.

2.2 Storing Data in Multiple Tables

In many datasets it is common to store data across multiple tables, usually with the goal of minimizing memory used as well as providing minimal duplication of information so any change that must be made is only made in a single place.

To see the rational why we might do this, consider building a data set of blood donations by a variety of donors across several years. For each blood donation, we will perform some assay and measure certain qualities about the blood and the patients health at the donation.

```
##   Donor Hemoglobin Systolic Diastolic
## 1 Derek       17.4      121        80
## 2  Jeff       16.9      145       101
```

But now we have to ask, what happens when we have a donor that has given blood multiple times? In this case we should just have multiple rows per person along with a date column to uniquely identify a particular donation.

```
donations
```

```
##   Donor       Date Hemoglobin Systolic Diastolic
## 1 Derek 2017-04-14       17.4      120        79
## 2 Derek 2017-06-20       16.5      121        80
## 3  Jeff 2017-08-14       16.9      145       101
```

I would like to include additional information about the donor where that infomation doesn't change overtime. For example we might want to have information about the donar's birthdate, sex, blood type. However, I don't want that information in *every single donation line*. Otherwise if I mistype a birthday and have to correct it, I would have to correct it *everywhere*. For information about the donor, should live in a `donors` table, while information about a particular donation should live in the `donations` table.

Furthermore, there are many Jeffs and Dereks in the world and to maintain a unique identifier (without using Social Security numbers) I will just create a `Donor_ID` code that will uniquely identify a person. Similarly I will create a `Donation_ID` that will uniquely identify a dontation.

```
donors
```

```
##   Donor_ID F_Name L_Name B_Type      Birth       Street      City State
## 1  Donor_1  Derek    Lee     O+ 1976-09-17 7392 Willard Flagstaff    AZ
## 2  Donor_2   Jeff  Smith      A 1974-06-23     873 Vine   Bozeman    MT
```

```
donations
```

```
##   Donation_ID Donor_ID       Date Hemoglobin Systolic Diastolic
## 1  Donation_1  Donor_1 2017-04-14       17.4      120        79
## 2  Donation_2  Donor_1 2017-06-20       16.5      121        80
## 3  Donation_3  Donor_2 2017-08-14       16.9      145       101
```

If we have a new donor walk in and give blood, then we'll have to create a new entry in the `donors` table as well as a new entry in the `donations` table. If an experienced donor gives again, we just have to create a new entry in the donations table.

```
donors
```

```
##   Donor_ID F_Name L_Name B_Type      Birth       Street      City State
## 1  Donor_1  Derek    Lee     O+ 1976-09-17 7392 Willard Flagstaff    AZ
## 2  Donor_2   Jeff  Smith      A 1974-06-23     873 Vine   Bozeman    MT
## 3  Donor_3 Aubrey    Lee     O+ 1980-12-15 7392 Willard Flagstaff    AZ
```

```
donations
```

```
##   Donation_ID Donor_ID       Date Hemoglobin Systolic Diastolic
## 1  Donation_1  Donor_1 2017-04-14       17.4      120        79
## 2  Donation_2  Donor_1 2017-06-20       16.5      121        80
## 3  Donation_3  Donor_2 2017-08-14       16.9      145       101
## 4  Donation_4  Donor_1 2017-08-26       17.6      120        79
## 5  Donation_5  Donor_4 2017-08-26       16.1      137        90
```

This data storage set-up might be flexible enough for us. However what happens if somebody moves? If we don't want to keep the historical information, then we could just change the person's `Street_Address`, `City`, and `State` values. If we do want to keep that, then we could create `donor_addresses` table that contains a `Start_Date` and `End_Date` that denotes the period of time that the address was valid.

```
donor_addresses
```

```
##   Donor_ID       Street      City State Start_Date   End_Date
```

```
## 1  Donor_1 346 Treeline   Pullman    WA 2015-01-26 2016-06-27
## 2  Donor_1     645 Main Flagstsff    AZ 2016-06-28 2017-07-02
## 3  Donor_1 7392 Willard Flagstaff    AZ 2017-07-03       <NA>
## 4  Donor_2     873 Vine   Bozeman    MT 2015-03-17       <NA>
## 5  Donor_3 7392 Willard Flagstaff    AZ 2017-06-01       <NA>
```

Given this data structure, we can now easily create new donations as well as store donor information. In the event that we need to change something about a donor, there is only *one* place to make that change.

However, having data spread across multiple tables is challenging because I often want that information squished back together. For example, the blood donations services might want to find all 'O' or 'O+' donors in Flagstaff and their current mailing address and send them some notification about blood supplies being low. So we need someway to join the `donors` and `donor_addresses` tables together in a sensible manner.

2.3 Table Joins

Often we need to squish together two data frames but they do not have the same number of rows. Consider the case where we have a data frame of observations of fish and a separate data frame that contains information about lake (perhaps surface area, max depth, pH, etc). I want to store them as two separate tables so that when I have to record a lake level observation, I only input it *one* place. This decreases the chance that I make a copy/paste error.

To illustrate the different types of table joins, we'll consider two different tables.

```
# tibbles are just data.frames that print a bit nicer and don't automaticall
# convert character columns into factors.  They behave a bit more consistently
# in a wide variety of situations compared to data.frames.
Fish.Data <- tibble(
  Lake_ID = c('A','A','B','B','C','C'),
  Fish.Weight=rnorm(6, mean=260, sd=25) ) # make up some data
Lake.Data <- tibble(
  Lake_ID = c(    'B','C','D'),
  Lake_Name = c('Lake Elaine', 'Mormon Lake', 'Lake Mary'),
  pH=c(6.5, 6.3, 6.1),
  area = c(40, 210, 240),
  avg_depth = c(8, 10, 38))
```

```
Fish.Data
```

```
## # A tibble: 6 x 2
##   Lake_ID Fish.Weight
##     <chr>       <dbl>
## 1       A    237.8033
## 2       A    197.9190
## 3       B    236.6522
## 4       B    285.1902
## 5       C    264.5087
## 6       C    265.3231
```

```
Lake.Data
```

```
## # A tibble: 3 x 5
##   Lake_ID   Lake_Name    pH  area avg_depth
##     <chr>       <chr> <dbl> <dbl>     <dbl>
## 1       B Lake Elaine   6.5    40         8
## 2       C Mormon Lake   6.3   210        10
```

```
## 3       D   Lake Mary   6.1   240        38
```

Notice that each of these tables has a column labled `Lake_ID`. When we join these two tables, the row that describes lake `A` should be duplicated for each row in the `Fish.Data` that corresponds with fish caught from lake `A`.

```
full_join(Fish.Data, Lake.Data)
```

```
## Joining, by = "Lake_ID"
```

```
## # A tibble: 7 x 6
##   Lake_ID Fish.Weight   Lake_Name    pH  area avg_depth
##     <chr>       <dbl>       <chr> <dbl> <dbl>     <dbl>
## 1       A    237.8033        <NA>    NA    NA        NA
## 2       A    197.9190        <NA>    NA    NA        NA
## 3       B    236.6522 Lake Elaine   6.5    40         8
## 4       B    285.1902 Lake Elaine   6.5    40         8
## 5       C    264.5087 Mormon Lake   6.3   210        10
## 6       C    265.3231 Mormon Lake   6.3   210        10
## 7       D          NA   Lake Mary   6.1   240        38
```

Notice that because we didn't have any fish caught in lake `D` and we don't have any Lake information about lake `A`, when we join these two tables, we end up introducing missing observations into the resulting data frame.

The other types of joins govern the behavor or these missing data.

`left_join(A, B)` For each row in A, match with a row in B, but don't create any more rows than what was already in A.

`inner_join(A,B)` Only match row values where both data frames have a value.

```
left_join(Fish.Data, Lake.Data)
```

```
## Joining, by = "Lake_ID"
```

```
## # A tibble: 6 x 6
##   Lake_ID Fish.Weight   Lake_Name    pH  area avg_depth
##     <chr>       <dbl>       <chr> <dbl> <dbl>     <dbl>
## 1       A    237.8033        <NA>    NA    NA        NA
## 2       A    197.9190        <NA>    NA    NA        NA
## 3       B    236.6522 Lake Elaine   6.5    40         8
## 4       B    285.1902 Lake Elaine   6.5    40         8
## 5       C    264.5087 Mormon Lake   6.3   210        10
## 6       C    265.3231 Mormon Lake   6.3   210        10
```

```
inner_join(Fish.Data, Lake.Data)
```

```
## Joining, by = "Lake_ID"
```

```
## # A tibble: 4 x 6
##   Lake_ID Fish.Weight   Lake_Name    pH  area avg_depth
##     <chr>       <dbl>       <chr> <dbl> <dbl>     <dbl>
## 1       B    236.6522 Lake Elaine   6.5    40         8
## 2       B    285.1902 Lake Elaine   6.5    40         8
## 3       C    264.5087 Mormon Lake   6.3   210        10
## 4       C    265.3231 Mormon Lake   6.3   210        10
```

The above examples assumed that the column used to join the two tables was named the same in both tables. This is good practice to try to do, but sometimes you have to work with data where that isn't the case. In

that situation you can use the `by=c("ColName.A"="ColName.B")` syntax where `ColName.A` represents the name of the column in the first data frame and `ColName.B` is the equivalent column in the second data frame.

2.4 Exercises

1. Suppose we are given information about the maximum daily temperature from a weather station in Flagstaff, AZ. The file is available at the GitHub site that this book is hosted on.

```
FlagTemp <-  read.csv(
  'https://github.com/dereksonderegger/570L/raw/master/data-raw/FlagMaxTemp.csv',
   header=TRUE, sep=',')
```

   This file is in a wide format, where each row represents a month and the columns X1, X2, ..., X31 represent the day of the month the observation was made.

   a. Convert data set to the long format where the data has only four columns: `Year`, `Month`, `Day`, `Tmax`.

   b. Calculate the average monthly maximum temperature for each Month in the dataset (So there will be 365 mean maximum temperatures). *You'll probably have some issues taking the mean because there are a number of values that are missing and by default R refuses to take means and sums when there is missing data. The argument* ***`na.rm=TRUE`*** *to* ***`mean()`*** *allows you to force R to remove the missing observations before calculating the mean.*

   c. Convert the average month maximums back to a wide data format where each line represents a year and there are 12 columns of temperature data (one for each month) along with a column for the year. *There will be a couple of months that still have missing data because the weather station was out of commision for those months and there was NO data for the entire month.*

2. A common task is to take a set of data that has multiple categorical variables and create a table of the number of cases for each combination. An introductory statistics textbook contains a dataset summarizing student surveys from several sections of an intro class. The two variables of interest for us are `Gender` and `Year` which are the students gender and year in college.

   a. Download the dataset and correctly order the `Year` variable using the following:

```
Survey <- read.csv('http://www.lock5stat.com/datasets/StudentSurvey.csv', na.strings=c('',' '))
   mutate(Year = factor(Year, levels=c('FirstYear','Sophomore','Junior','Senior')))
```

   b. Using some combination of `dplyr` functions, produce a data set with eight rows that contains the number of responses for each gender:year combination. *Notice there are two females that neglected to give their Year and you should remove them first. The function* ***`is.na(Year)`*** *will return logical values indicating if the Year value was missing and you can flip those values using the negation operator* ***`!`****. So you might consider using* ***`!is.na(Year)`*** *as the argument to a filter() command. Alternatively you sort on Year and remove the first two rows using* ***`slice(-2:-1)`****. Next you'll want to summarize each Year/Gender group using the* ***`n()`*** *function which gives the number of rows in a data set.*

   c. Using `tidyr` commands, produce a table of the number of responses in the following form:

Gender	First Year	Sophmore	Junior	Senior
Female				
Male				

3. The package `nycflights` contains information about all the flights that arrived in or left from New

York City in 2013. This package contains five data tables, but there are three data tables we will work with. The data table `flights` gives information about a particular flight, `airports` gives information about a particular airport, and `airlines` gives information about each airline. Create a table of all the flights on February 14th by Virgin America that has columns for the carrier, destination, departure time, and flight duration. Join this table with the airports information for the destination. Notice that because the column for the destination airport code doesn't match up between `flights` and `airports`, you'll have to use the `by=c("TableA.Col"="TableB.Col")` argument where you insert the correct names for `TableA.Col` and `TableB.Col`.

Chapter 3

Markov Chain Monte Carlo

```
library(ggplot2)
library(dplyr)
library(devtools)
install_github('dereksonderegger/STA578') # Some routines I created for this
library(STA578)
```

Modern statistical methods often rely on being able to produce random numbers from an arbitrary distribution. In this chapter we will explore how this is done.

3.1 Generating $U \sim Uniform(0,1)$

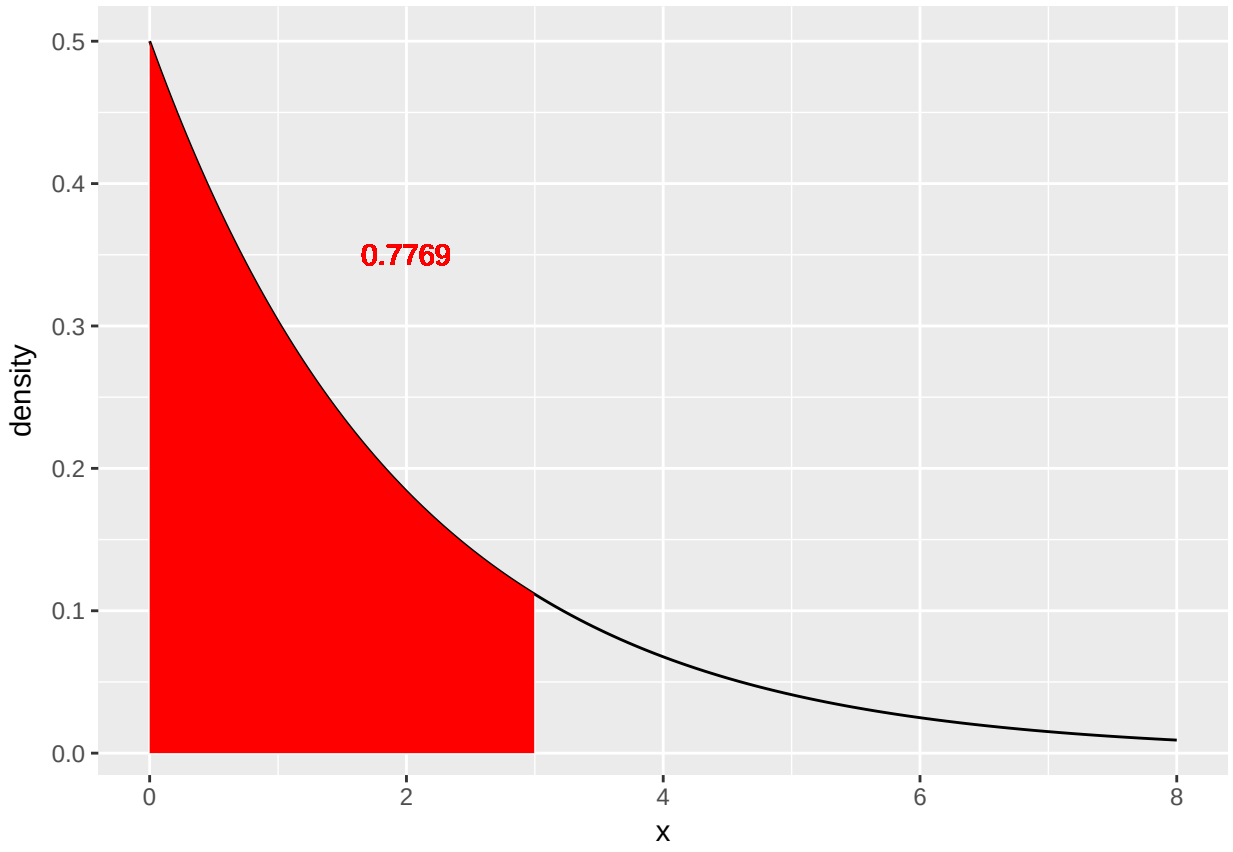
It is extremely difficult to generate actual random numbers but it is possible to generate pseudo-random numbers. That is, we'll generate a sequence of digits, say 1,000,000,000 long such that any sub-sequence looks like we are just drawing digits [0-9] randomly. Then given this sequence, we chose a starting point (perhaps something based off the clock time of the computer we are using). From the starting point, we generate $U \sim Uniform(0,1)$ numbers by using just reading off successive digits.

In practice there are many details of the above algorithm that are quite tricky, but we will ignore those issues and assume we have some method for producing random samples from $Uniform\,(0,1)$ distribution.

3.2 Inverse CDF Method

Suppose that we wish to generate a random sample from a given distribution, say, $X \sim Exp\,(\lambda = 1/2)$. This distribution is pretty well understood and it is easy to calculate various probabilities. The density function is $f\,(x) = \lambda \cdot e^{-x\lambda}$ and we can easily calculate probabilities such as

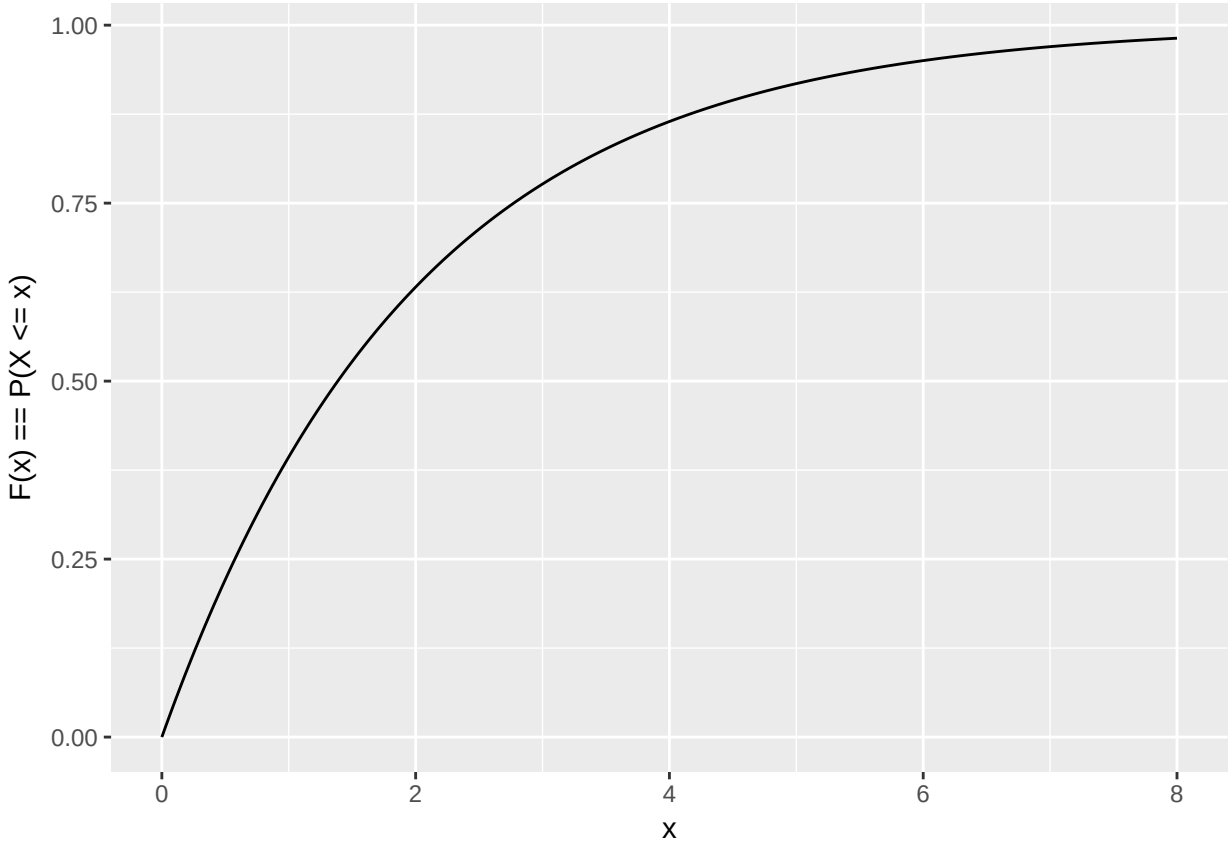
$$\begin{aligned} P\,(X \leq 3) &= \int_0^3 \lambda e^{-x\lambda}\,dx \\ &= e^{-0\lambda} - e^{-3\lambda} \\ &= 1 - e^{-3\lambda} \\ &= 0.7769 \end{aligned}$$



Given this result, it is possible to figure out $P(X \leq x)$ for any value of x. (Here the capital X represents the random variable and the lower case x represents a particular value that this variable can take on.) Now thinking of these probabilities as a function, we define the cumulative distribution function (CDF) as

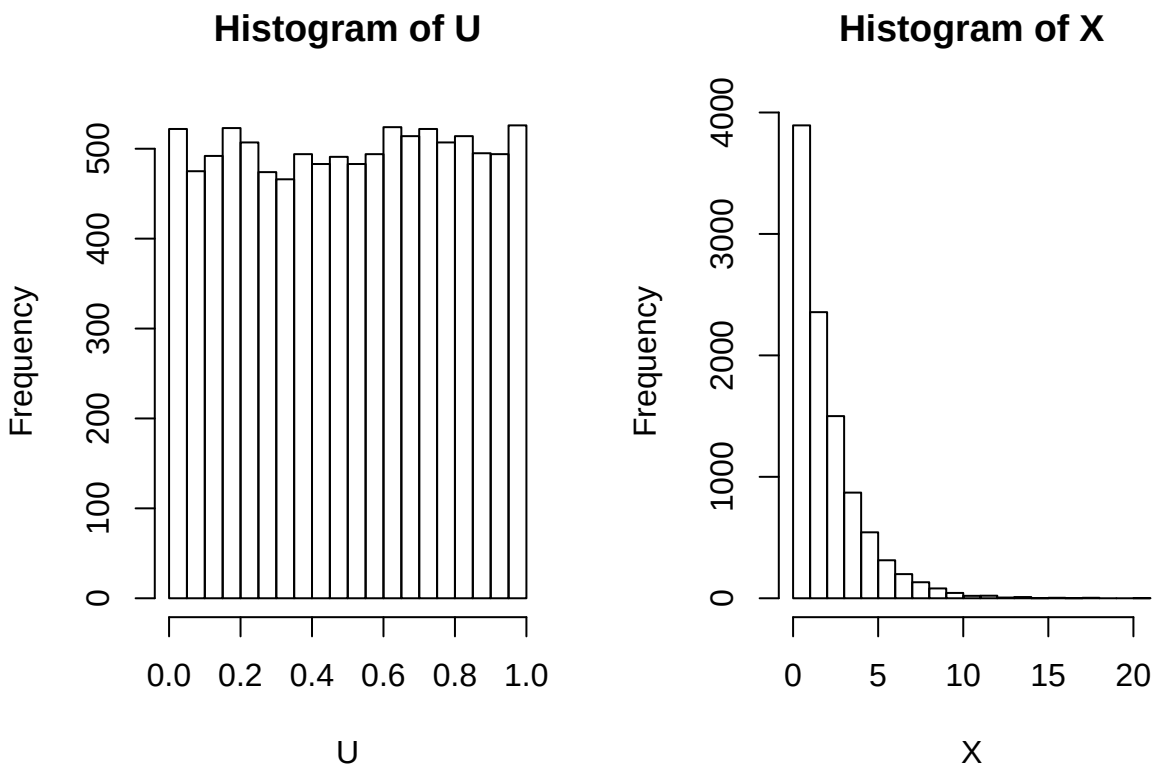
$$F(x) = P(X \leq x) = 1 - e^{-x\lambda}$$

If we make a graph of this function we have



Given this CDF, we if we can generate a $U \sim Uniform(0,1)$, we can just use the CDF function in reverse (i.e the inverse CDF) and transform the U to be an $X \sim Exp(\lambda)$ random variable. In R, most of the common distributions have a function that calculates the inverse CDF, in the exponential distribution it is `qexp(x, rate)` and for the normal it would be `qnorm()`, etc.

```
U <- runif(10000, min=0, max=1)  # 10,000 Uniform(0,1) values
X <- qexp(U, rate=1/2)
par(mfrow=c(1,2))
hist(U)
hist(X)
```

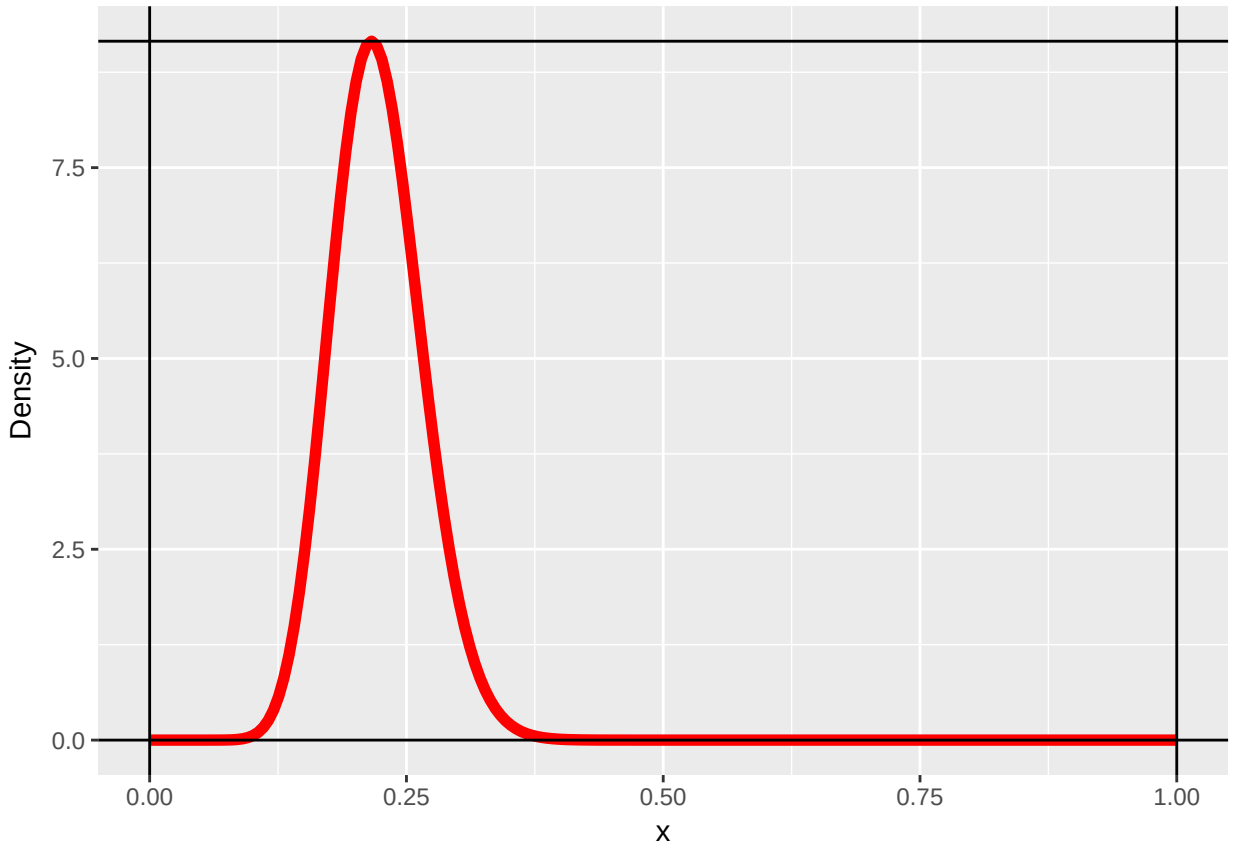


This is the type of trick that Statistics students might learn in a probability course, but this is hardly interesting from a computationally intensive perspective, so if you didn't follow the calculus, don't fret.

3.3 Accept/Reject Algorithm

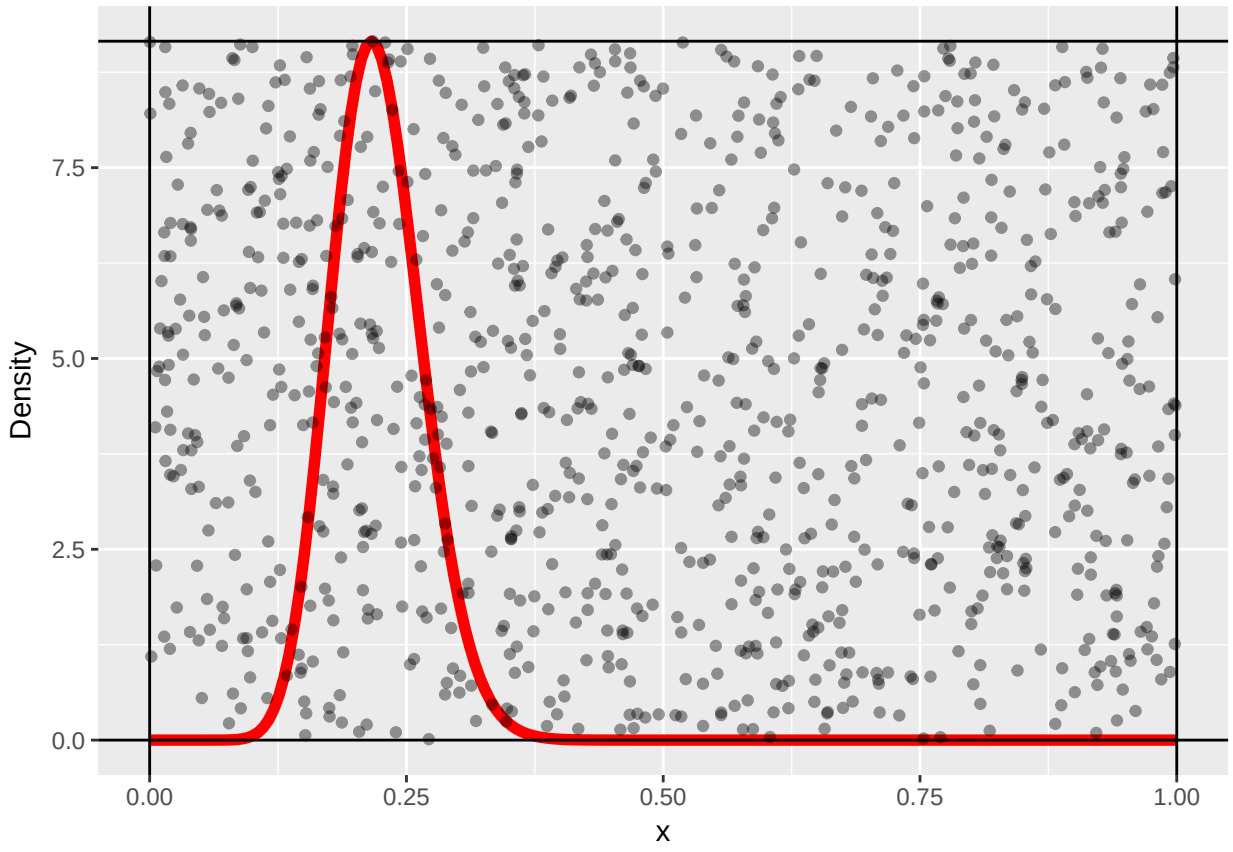
We now consider a case where we don't know the CDF (or it is really hard to work with). Let the random variable X which can take on values from $0 \leq X \leq 1$ and has probability density function $f(x)$. Furthermore, suppose that we know what the maximum value of the density function is, which we'll denote $M = \max f(x)$.

```
x <- seq(0,1, length=200)
y <- dbeta(x, 20, 70)
M <- max(y)
data.line <- data.frame(x=x, y=y)
p <- ggplot(data.line, aes(x=x, y=y)) + geom_line(color='red', size=2) +
  labs(y='Density') +
  geom_vline(xintercept=c(0,1)) +
  geom_hline(yintercept=c(0, max(y)))
p
```



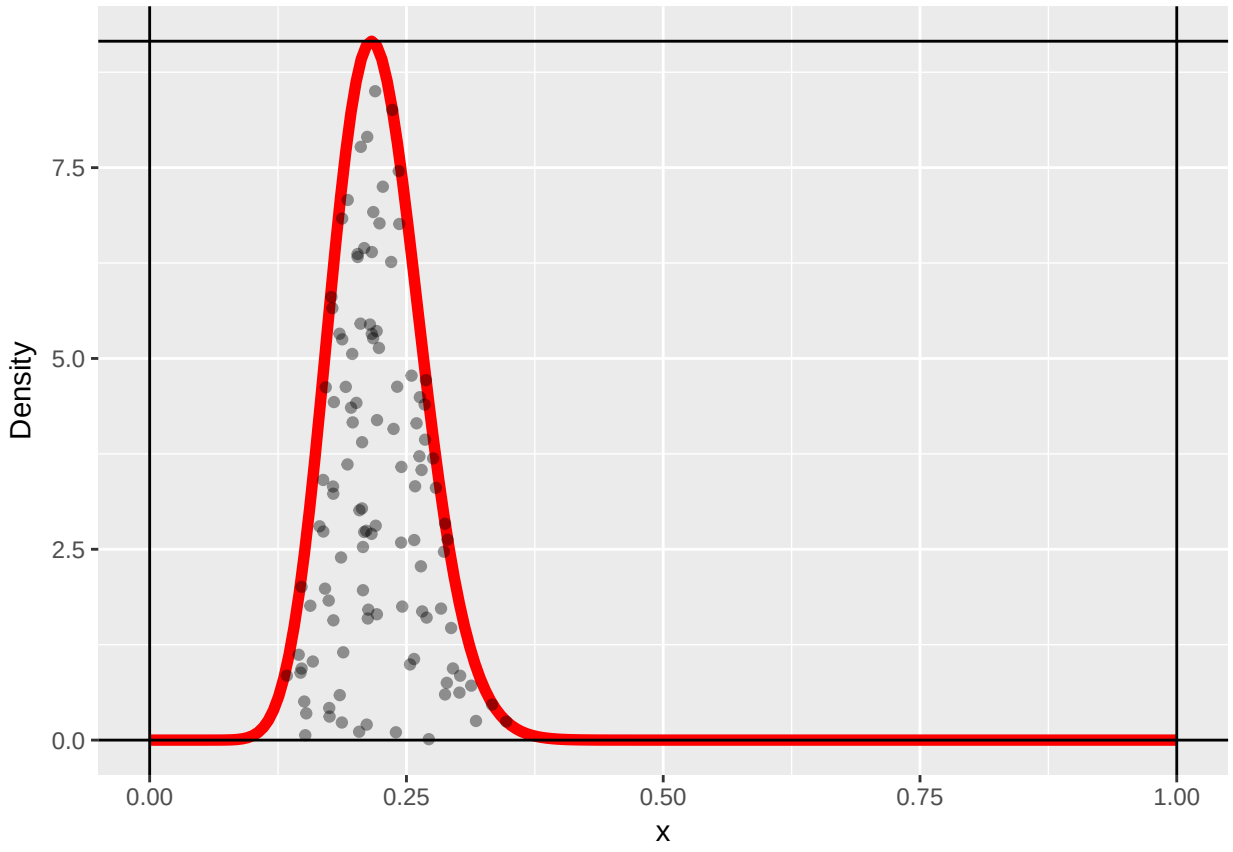
It is trivial to generate points that are uniformly distributed in the rectangle by randomly selecting points (x_i, y_i) by letting x_i be randomly sampled from a $Uniform(0, 1)$ distribution and y_i be sampled from a $Uniform(0, M)$ distribution. Below we sample a thousand points.

```
N <- 1000
x <- runif(N, 0,1)
y <- runif(N, 0, M)
proposed <- data.frame(x=x, y=y)
p + geom_point(data=proposed, alpha=.4)
```



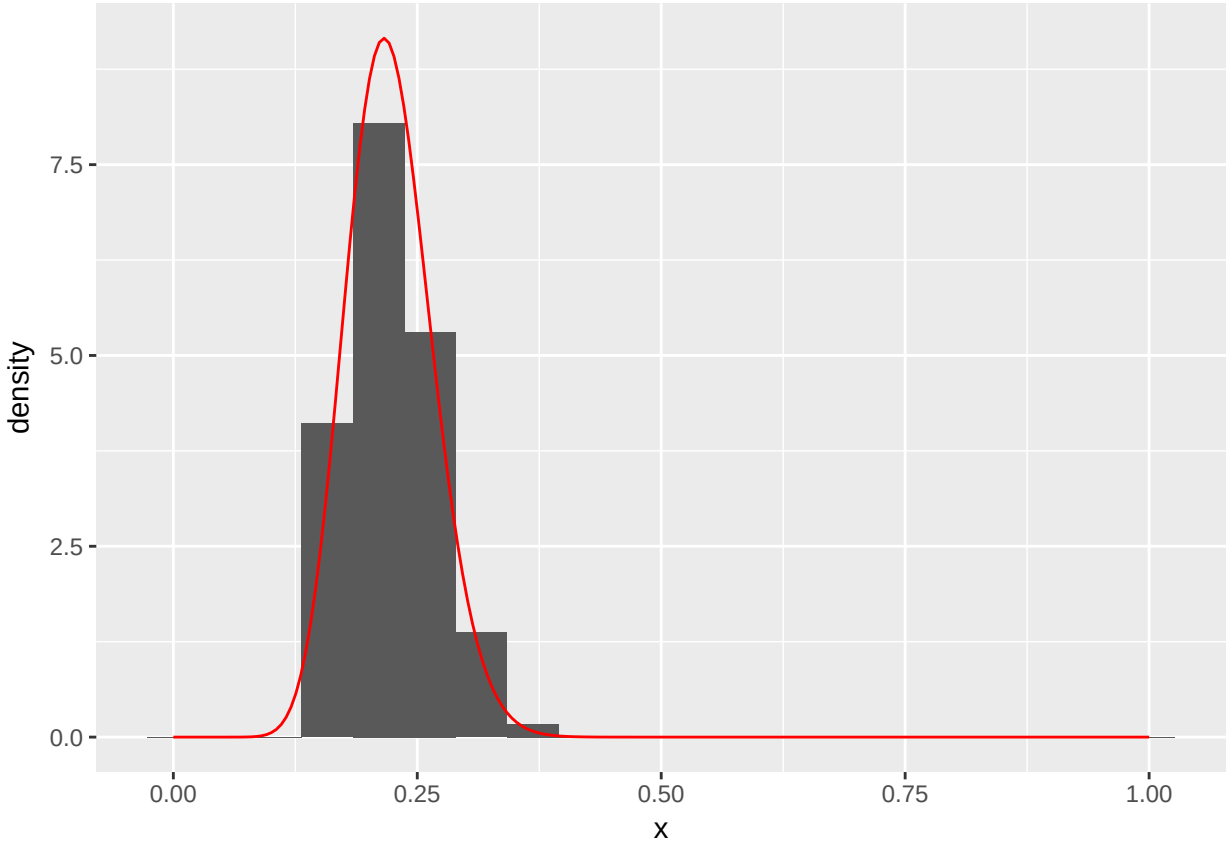
Since we want to select a random sample from the curved distribution (and not uniformly from the box), I will reject pairs (x_i, y_i) if $y_i \geq f(x_i)$. This leaves us with the following points.

```
accepted <- proposed %>%
  filter( y <= dbeta(x, 20,70) )
p + geom_point(data=accepted, alpha=.4)
```



We can now regard those x_i values as a random sample from the distribution $f(x)$ and the histogram of those values is a good approximation to the distribution $f(x)$.

```
ggplot(data=accepted, aes(x=x)) +
  geom_histogram(aes(y=..density..), bins=20) +
  geom_line(data=data.line, aes(x=x,y=y), color='red')
```



Clearly this is a fairly inefficient way to generate points from a distribution, but it contains a key concept

$$x_i \text{ is accepted if } u_i < \frac{f(x_i)}{M}$$

where $y_i = u_i \cdot M$ is the height of the point and u_i is a random observation from the $Uniform(0,1)$ distribution.

Pros/Cons

- Pro: This technique is very simple to program.
- Cons: Need to know the maximum height of the distribution.
- Cons: This can be a very inefficient algorithm as most of the proposed values are thrown away.

3.4 MCMC algorithm

Suppose that we have one observation, x_1, from the distribution $f(x)$ and we wish to create a whole sequence of observations $x_2, \ldots, x_n$ that are also from that distribution. The following algorithm will generate x_{i+1} given the value of x_i.

1. Generate a proposed x^*_{i+1} from a distribution that is symmetric about x_i. For example $x^*_{i+1} \sim N(x_i, \sigma = 1)$.
2. Generate a random variable u_{i+1} from a $Uniform(0,1)$ distribution.
3. If $u_{i+1} \leq \frac{f(x^*_{i+1})}{f(x_i)}$ then we accept x^*_{i+1} and define $x_{i+1} = x^*_{i+1}$, otherwise reject x^*_{i+1} and define $x_{i+1} = x_i$

The idea of this algorithm is that if we propose an x_{i+1}^* value that has a higher density than x_i, we should always accept that value as the next observation in our chain. If we propose a value that has lower density, then we should *sometimes* accept it, and the correct probability of accepting it is the ratio of the probability density function. If we propose a value that has a much lower density, then we should rarely accept it, but if we propose a value that is has only a slightly lower density, then we should usually accept it.

There is theory that show that a large sample drawn as described will have the desired proportions that match the distribution of interest. However, the question of "how large is large enough" is a very difficult question.

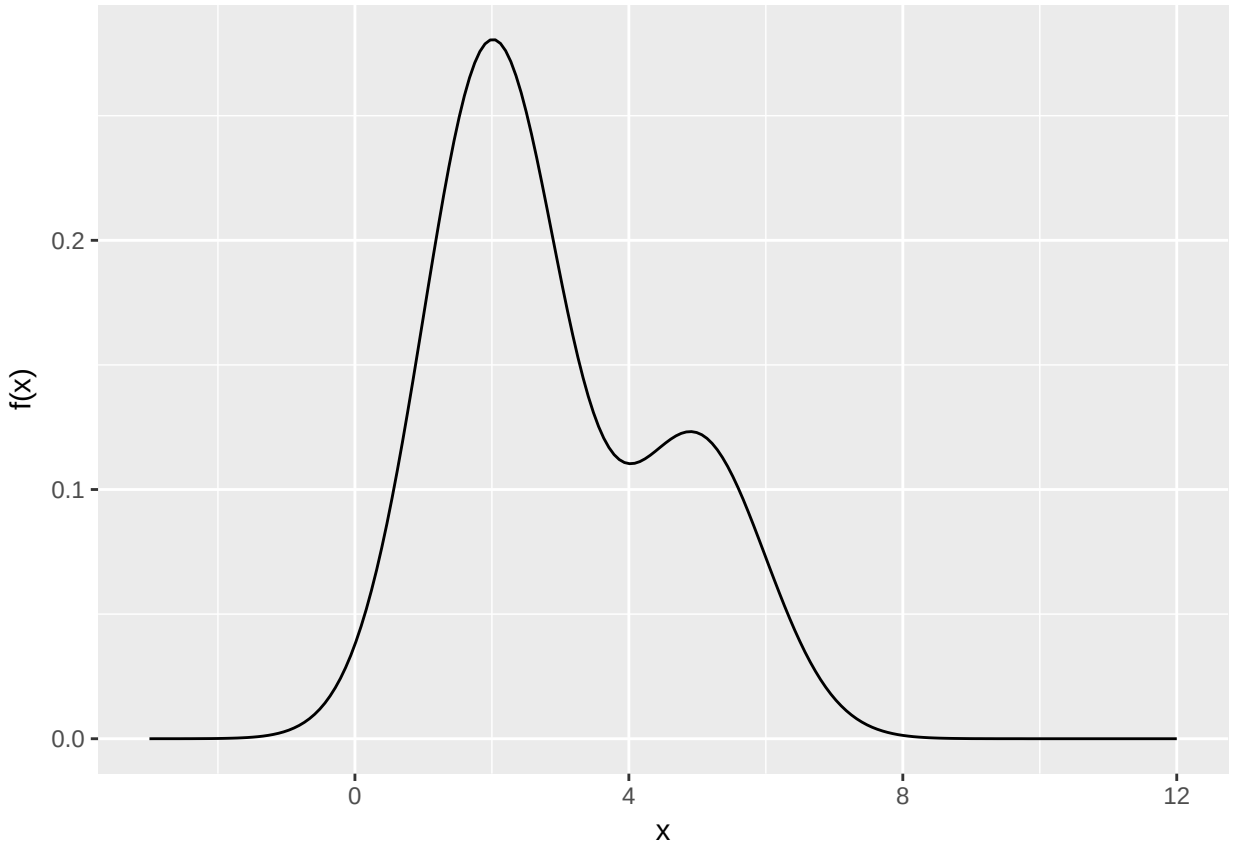
Actually we only need to know the distribution up to a scaling constant. Because we'll look at the ratio $f(x^*)/f(x)$ any constants that don't depend on x will cancel. For Bayesians, this is a crucial point.

3.4.1 Mixture of normals

We consider the problem of generating a random sample from a mixture of normal distributions. We first define our distribution $f(x)$

```
df <- function(x){
  return(.7*dnorm(x, mean=2, sd=1) + .3*dnorm(x, mean=5, sd=1))
}
```

```
x <- seq(-3,12, length=200)
density.data <- data.frame(x=x, y=df(x))
density <- ggplot( density.data, aes(x=x, y=y)) +
  geom_line() +
  labs(y='f(x)', x='x')
density
```



Next we define our proposal distribution, which will be $Uniform\left(x_i - 2,\, x_i + 2\right)$.

```
rproposal <- function(x.i){
  out <- x.i + runif(1, -2, 2)  # x.i + 1 observation from Uniform(-2, 2)
  return(out)
}
```

Starting from $x_1 = 3$, we will randomly propose a new value.

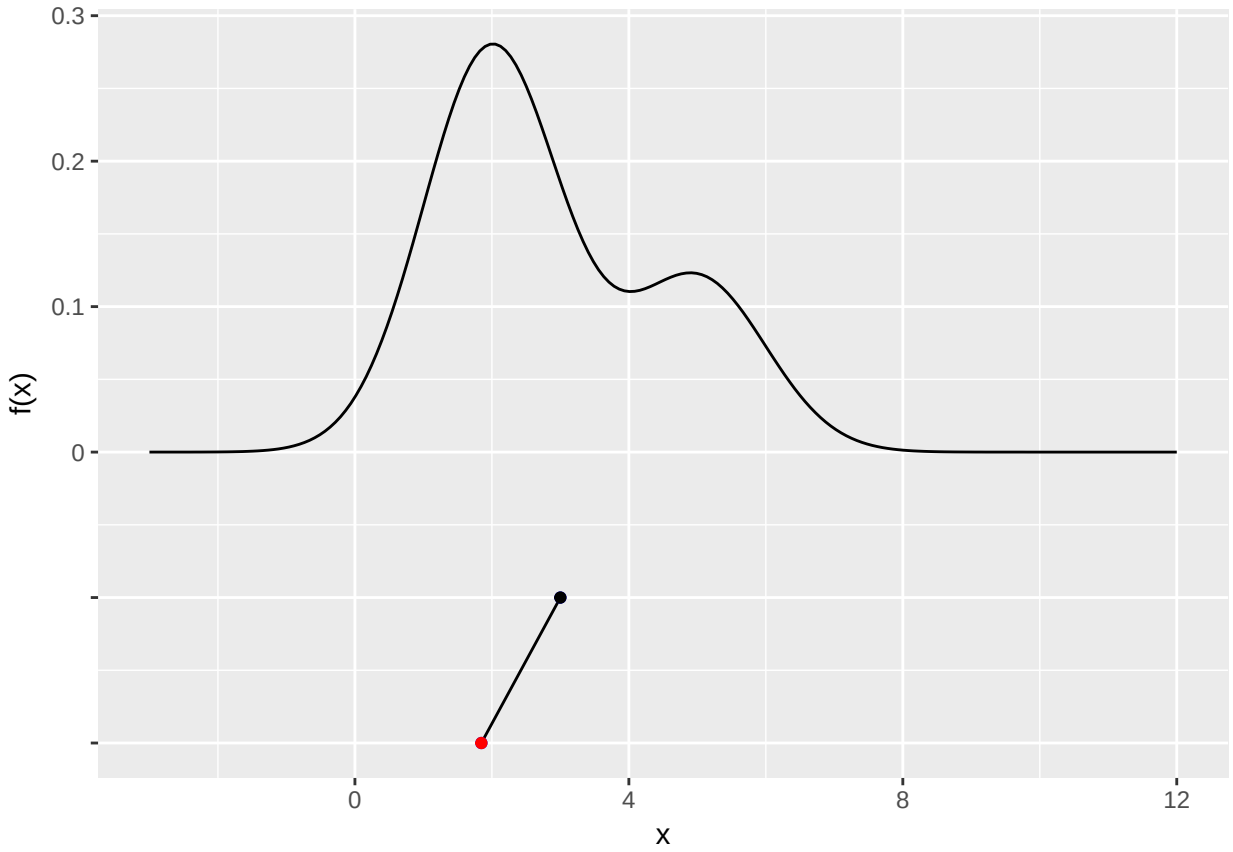
```
x <- 3;       # starting value for the chain
x.star <- 3   # initialize the vector of proposal values
x.star[2] <- rproposal( x[1] )
x.star[2]
```

```
## [1] 1.846296
```

```
if( df(x.star[2]) / df(x[1])  >  runif(1) ){
  x[2] <- x.star[2]
}else{
  x[2] <- x[1]
}
```

We proposed a value of $x = 1.846$ and because $f\left(1.846\right) \geq f\left(3\right)$ then this proposed value must be accepted (because the ratio $f\left(x_2^*\right)/f\left(x_1\right) \geq 1$ and is therefore greater than $u_2 \in [0, 1]$. We plot the chain below the density plot to help visualize the process.

```
STA578::plot_chain(density, x, x.star)
```

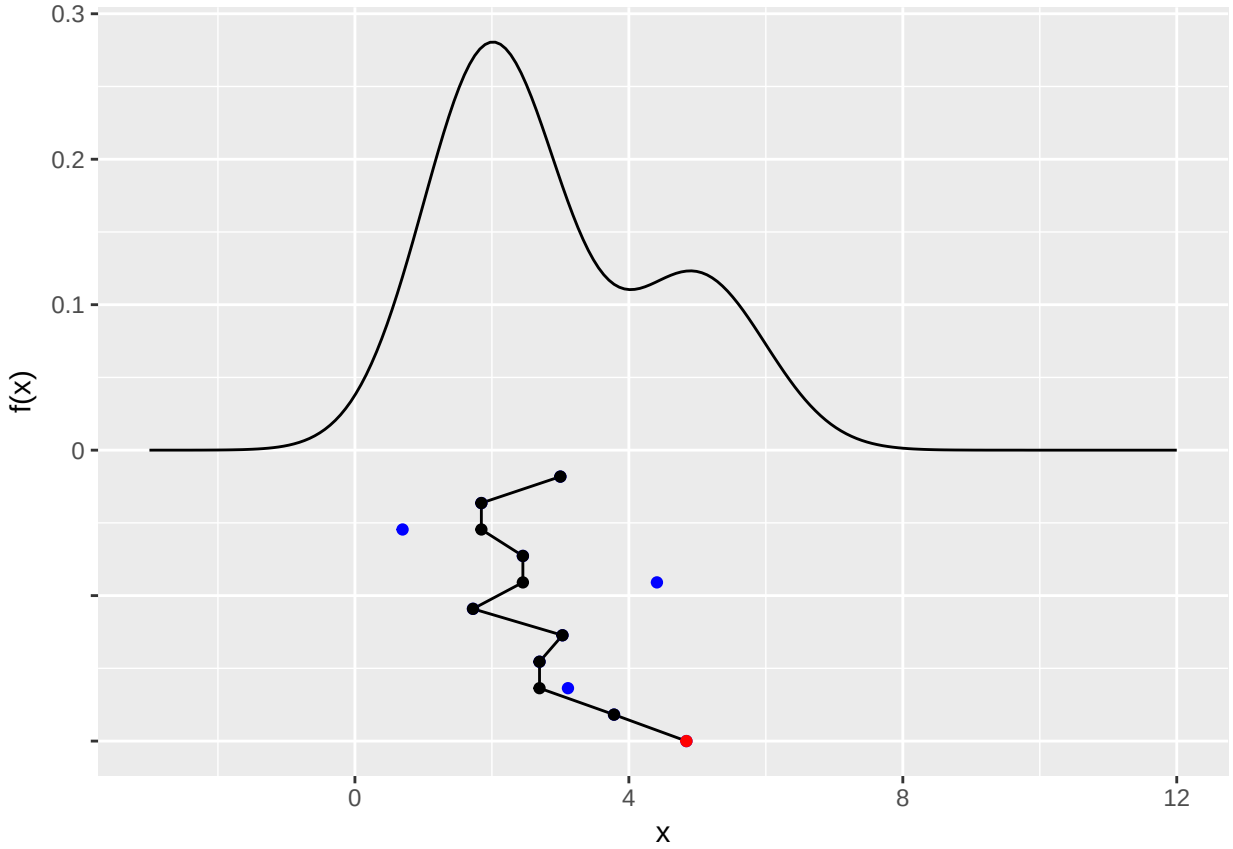


Next we will repeat the process for ten iterations. If a proposed point is rejected, it will be shown in blue and not connected to the chain. The connected line will be called a traceplot and is usually seen rotated

counter-clockwise by 90 degrees.

```
for( i in 2:10 ){
  x.star[i+1] <- rproposal( x[i] )
  if( df(x.star[i+1]) / df(x[i]) > runif(1) ){
    x[i+1] <- x.star[i+1]
  }else{
    x[i+1] <- x[i]
  }
}
```

```
STA578::plot_chain(density, x, x.star)
```



We see that the third step in the chain, we proposed a value near $x = 1$ and it was rejected. Similarly steps 5 and 9 were rejected because $f\left(x^*_{i+1}\right)$ of the proposed value was very small compared to $f\left(x_i\right)$ and thus the proposed value was unlikely to be accepted.

At this point, we should make one final adjustment to the algorithm. In general, it is not necessary for the proposal distribution to by symmetric about x_i. Let $g\left(x^*_{i+1}|x_i\right)$ be the density function of the proposal distribution, then the appropriate accept/reject criteria is modified to be

- If

$$u_{i+1} \leq \frac{f\left(x^*_{i+1}\right)}{f\left(x_i\right)} \frac{g\left(x_i|x^*_{i+1}\right)}{g\left(x^*_{i+1}|x_i\right)}$$

then we accept x^*_{i+1} and define $x_{i+1} = x^*_{i+1}$, otherwise reject x^*_{i+1} and define $x_{i+1} = x_i$

Notice that if $g\left(\right)$ is symmetric about x_i then $g\left(x_i|x^*_{i+1}\right) = g\left(x^*_{i+1}|x_i\right)$ and the second fraction is simply 1.

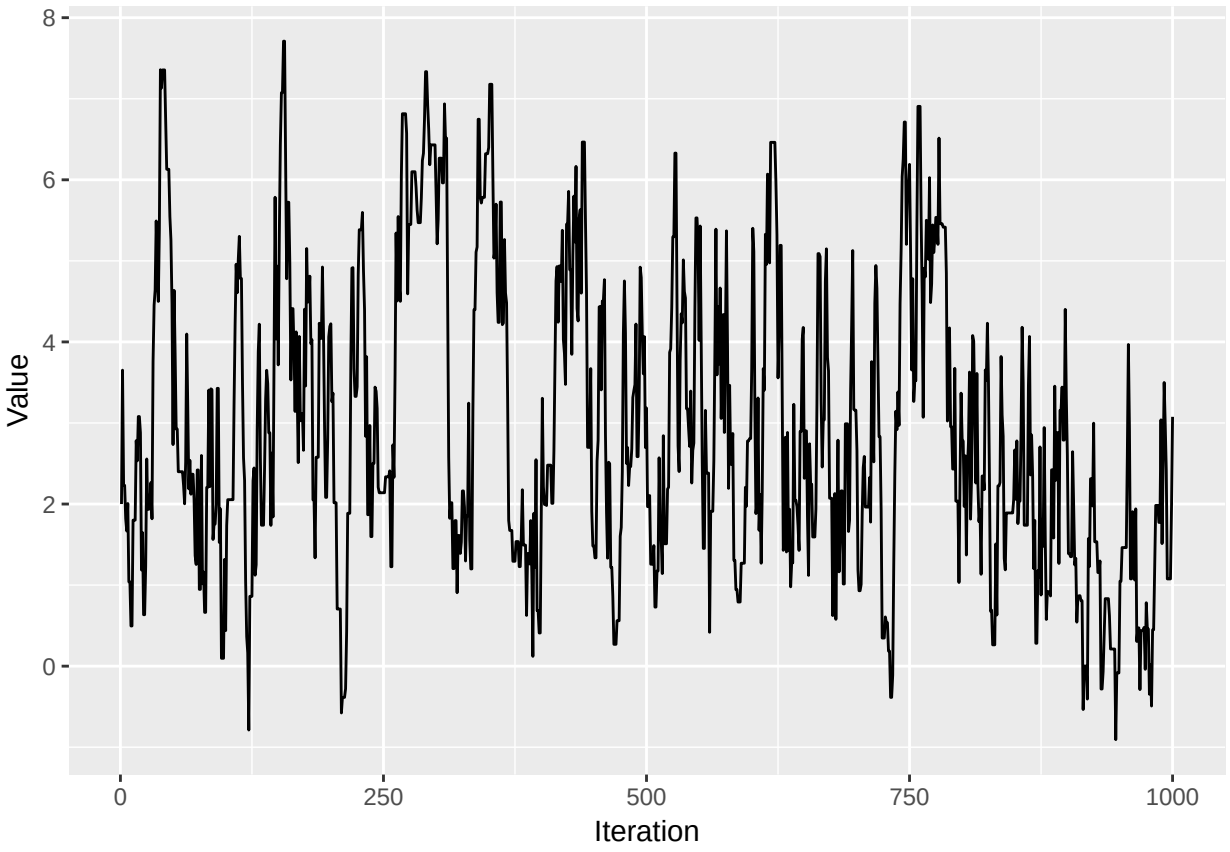
It will be convenient to hide the looping part of the MCMC, so we'll create a simple function to handle the details. We'll also make a program to make a traceplot of the chain.

```
?MCMC
# You can look at the source code of many functions by just typing the name.
# This trick unfortunately doesn't work with functions that are just wrappers
# to C or C++ programs or are hidden due to Object Oriented inheritance.
MCMC
```

```
## function (df, start, rprop, dprop = NULL, N = 1000)
## {
##     if (is.null(dprop)) {
##         dprop <- function(to, from) {
##             1
##         }
##     }
##     chain <- rep(NA, N)
##     chain[1] <- start
##     for (i in 1:(N - 1)) {
##         x.star <- rprop(chain[i])
##         r1 <- df(x.star)/df(chain[i])
##         r2 <- dprop(chain[i], x.star)/dprop(x.star, chain[i])
##         if (r1 * r2 > runif(1)) {
##             chain[i + 1] <- x.star
##         }
##         else {
##             chain[i + 1] <- chain[i]
##         }
##     }
##     return(chain)
## }
## <environment: namespace:STA578>
```

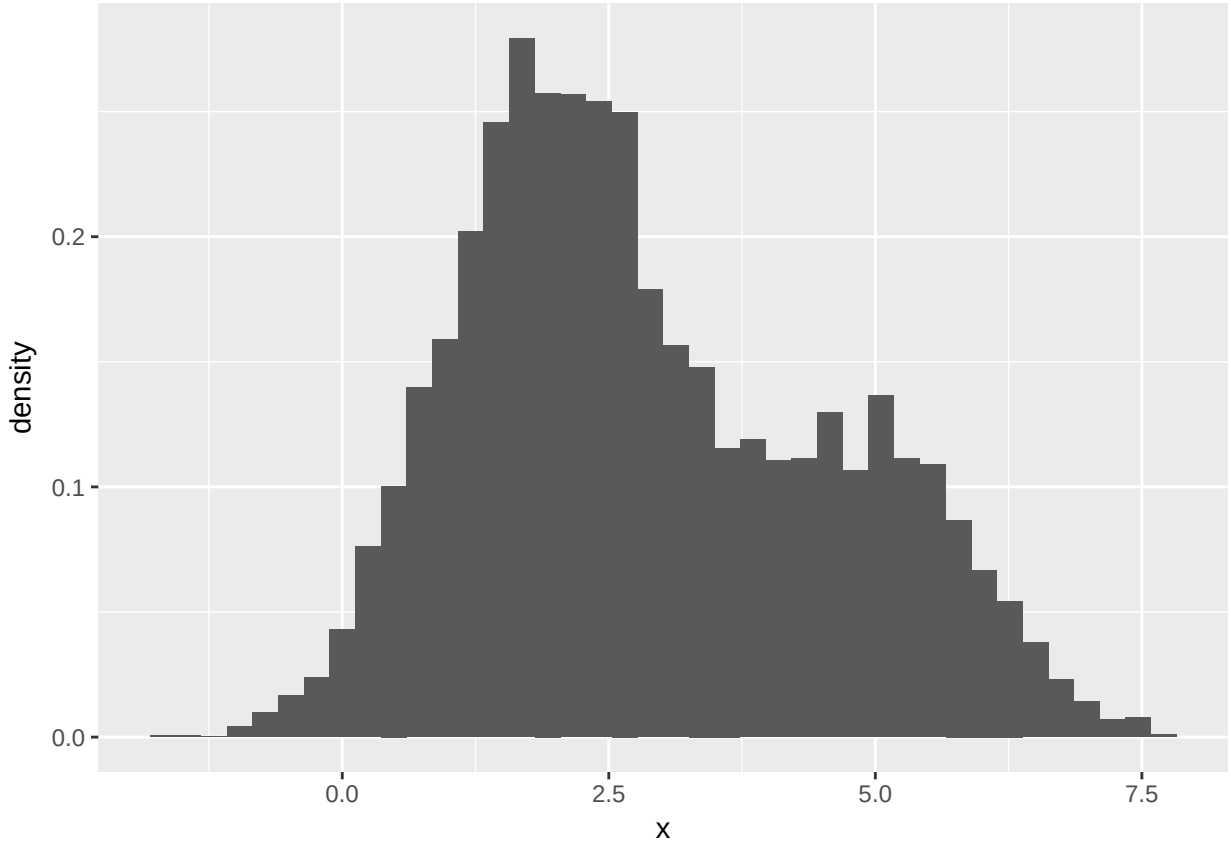
We should let this process run for a long time, and we will just look the traceplot.

```
chain <- STA578::MCMC(df, 2, rproposal, N=1000) # do this new!
STA578::trace_plot(chain)
```



This seems like it is working as the chain has occasional excursions out tho the tails of the distribution, but is mostly concentrated in the peaks of the distribution. We'll run it longer and then examine a histogram of the resulting chain.

```
chain2 <- MCMC(df, chain[1000], rproposal, N=10000)
long.chain <- c(chain, chain2)
ggplot( data.frame(x=long.chain), aes(x=x, y=..density..)) +
  geom_histogram(bins=40)
```



All in all this looks quite good.

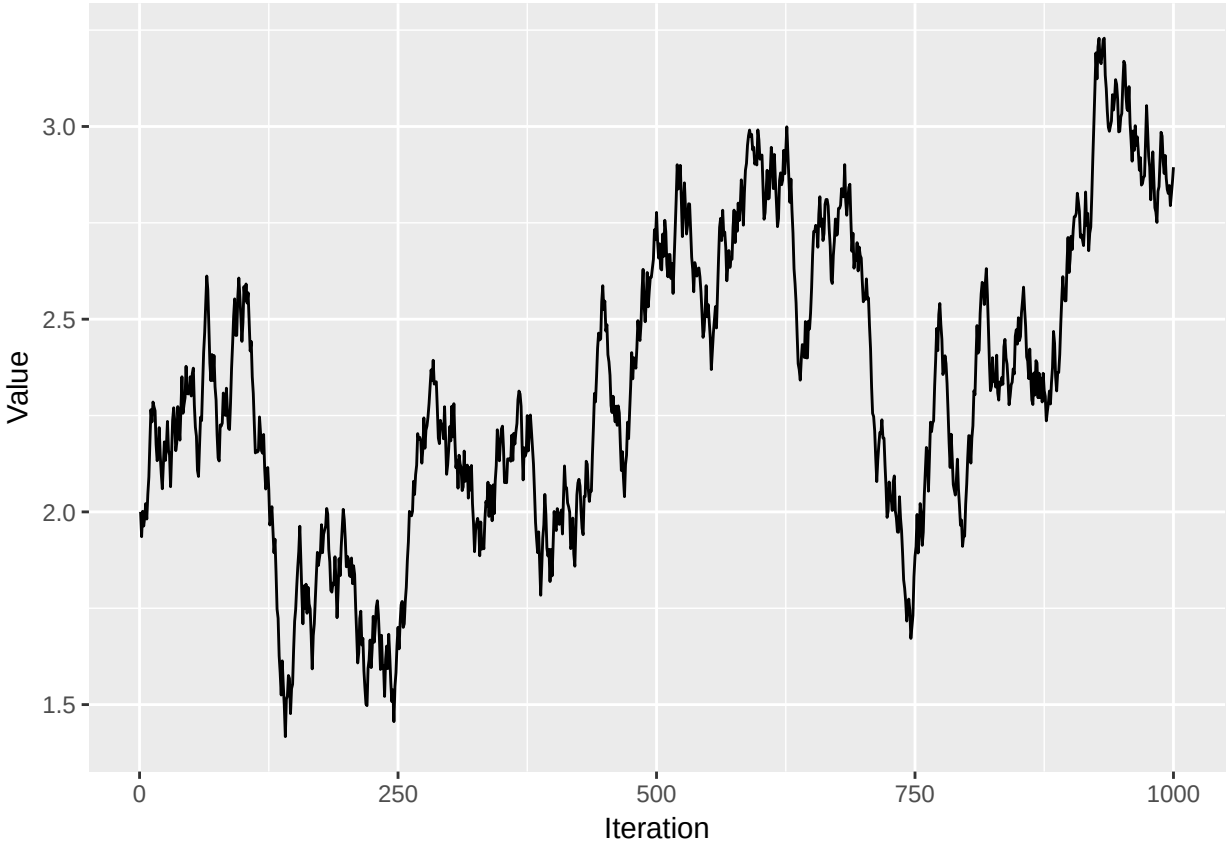
3.4.2 Common problems

Proposal Distribution

The proposal distribution plays a critical role in how well the MCMC simulation explores the distribution of interest. If the variance of the proposal distribution is too small, then the number of steps to required to move a significant distance across the distribution becomes large. If the variance is too large, then a large percentage of the proposed values will be far from the center of the distribution and will be rejected.

First, we create a proposal distribution with a small variance and examine its behavior.

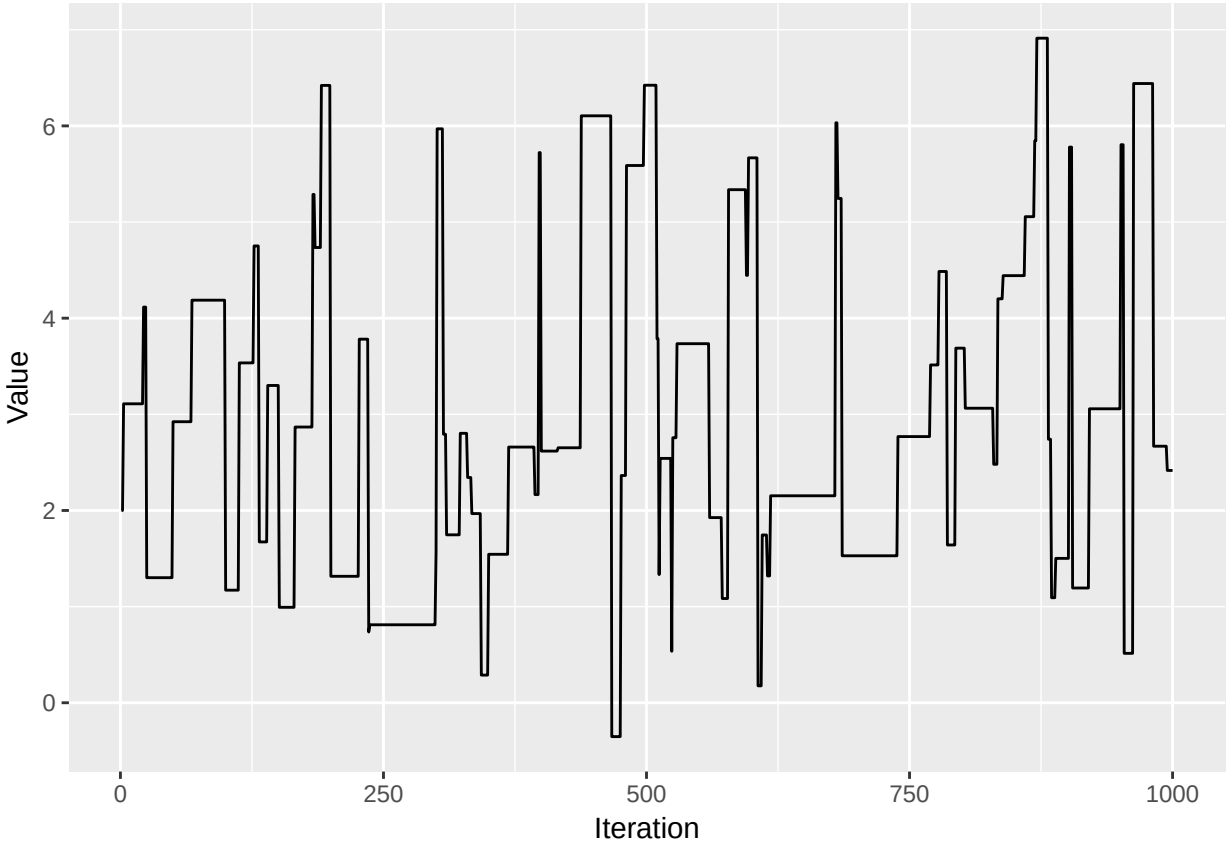
```
p.small <- function(x){
  return( x + runif(1, -.1, +.1) )
}
chain <- MCMC(df, 2, p.small, N=1000)
trace_plot(chain)
```



In these 1000 steps, the chain has not gone smaller than 1 or larger than 3.5 and hasn't explored the second "hump" of the distribution yet. If the valley between the two humps was deeper, it is possible that the chain would never manage to cross the valley and find the second hump.

The effect of too large of variance is also troublesome.

```
p.large <- function(x){
  return( x + runif(1, -30, +30) )
}
chain <- MCMC(df, 2, p.large, N=1000)
trace_plot(chain)
```



This chain is not good because it stays in place for too long before finding another suitable value. It will take a very long time to suitably explore the distribution.

Given these two examples, it is clear that the choice of the variance parameter is a critical choice. A commonly used rule of thumb that balances these two issues is that the variance parameter should be chosen such that approximately $40 - 60\%$ of the proposed values are accepted. To go about finding that variance parameter, we will have a initialization period where we tweak the variance parameter until we are satisfied with the acceptance rate.

Burn-in

Another problem with the MCMC algorithm is that it requires that we already have a sample drawn from the distribution in question. This is clearly not possible, so what we do is use a random start and hope that the chain eventually finds the center of the distribution.

<<fig.height=3, fig.width=5>>=

Given this example, it seems reasonable to start the chain and then disregard the initial "burn-in" period. However, how can we tell the difference between a proposal distribution with too small of variance, and a starting value that is far from the center of the distribution? One solution is to create multiple chains with different starting points. When all the chains become well mixed and indistinguishable, we'll use the remaining observations in the chains as the sample.

<<fig.height=3, fig.width=5>>=

1.4.2 Assessing Chain Convergence

Perhaps the most convenient way to asses convergence is just looking at the traceplots, but it is also good to derive some quantitative measurements of convergence. One idea from ANOVA is that the variance of the distribution we are sampling from $\left(\sigma^2\right)$ could be estimated using the within chain variability or it could be estimated using the variance between the chains.

Let m be the number of chains and n be the number of samples per chain. We define W be the average of the within chain sample variances. Formally we let s_i^2 be the sample variance of the i^{th} chain and $W = \frac{1}{m}\sum_{i=1}^{m} s_i^2$

We can interpret W as the amount of wiggle within each chain.

Next we recognize that under perfect sampling and large sample sizes, the chains should be right on top of each other, and the mean of each chain should be very very close to the mean of all the other chains. Mathematically, we'd say the mean of each chain has a distribution $\bar{x}_i \sim N\left(\mu, \frac{\sigma^2}{n}\right)$ where μ is the mean of the distribution we are approximating. Doing some probability, this implies that $n\bar{x}_i \sim N\left(n\mu, \sigma^2\right)$. Next, using the m chain means, we could estimate σ^2 by looking at the variances of the chain means! Let $\bar{x}_{..}$ be the mean of all observation and $\bar{x}_{i\cdot}$ be the mean of the i^{th} chain. Let

$$B = \frac{n}{m-1}\sum_{i=1}^{m}(\bar{x}_{i\cdot} - \bar{x}_{..})^2$$

Notice that both W and B are estimating σ^2, but if the MCMC has not converged (i.e. the chains don't overlap), the B will be much larger than σ^2. Likewise, W will be much smaller than σ^2 because the chains won't have fully explored the distribution. We'll put these two estimators together using a weighted average and define.$\hat{\sigma}^2 = \frac{n-1}{n}W + \frac{1}{n}$ B Notice that if the chains have not yet converged, then $\hat{\sigma}^2$ should overestimate σ^2 because the between chain variance is too large. By itself W will underestimate σ^2 because the chains haven't fully explored the distribution. This suggests

$$\hat{R} = \sqrt{\frac{\hat{\sigma}^2}{W}}$$

is a useful ratio which decreases to 1 as $n \to \infty$. This ratio (Gelman and Rubin, 1992) can be interpreted as the potential reduction in the estimate of $\hat{\sigma}^2$ if we continued sampling.

A closely related is the number of effective samples from the posterior distribution $n_{eff} = mn\frac{\hat{\sigma}^2}{B}$

To demonstrate these, we'll calculate $\hat{R}$ for the above chains

A useful rule-of-thumb for assessing convergence is if the $\hat{R}$ value(s) are less than 1.05, then we are close enough. However, this does not imply that we have enough independent draws from the sample to trust our results. In the above example, even though we have $4 \cdot 800 = 3200$ observations, because of the Markov property of our chain, we actually only have about 35 independent draws from the distribution. This suggests that it takes around 100 MCMC steps before we get independence between two observations.

To further explore this idea, lets look at what happens when the chains don't mix as well.

Bibliography