MTH 3270 Notes 8

13 Simulation (13)

13.1 Introduction

- Data are generated by natural or man-made systems or processes. We can sometimes gain understanding about those systems or processes by ***simulating*** data.
- Two ways this is done:
  - Conditional inference – simulating data after tweaking parameters of the system or process, thereby forming "what-if" scenarios, to determine the effect of the tweak on the (simulated) data.
  - "Winnowing" out hypotheses – simulating several data sets, each consistent with a different hypothesis about the system or process, and comparing them to real data to rule out one (or more) of the hypotheses.

13.2 Random Number Generation

13.2.1 Sampling Elements from a Vector

- **Sampling** and **resampling** (**bootstrap**) were introduced in Class Notes 5.
- Now, to generate a random sample from the elements of a *vector*, we'll use:

```
sample()      # Generate a random sample from the elements of a vector.
```

- `sample()` takes arguments `x`, a vector, and `size`, the desired sample size. It can be used to:
  - **Sample** – By default `sample()` returns a sample drawn **without replacement** from the elements of `x`.
  - **Resample** (e.g. as in **bootstrap**) – An optional argument `replace` can be set to `TRUE` to sample **with replacement**.
  - **Shuffle** – Setting `size = length(x)` **rearranges** the values from `x` so that they're in a **random order**.
- For example, to generate a random **sample** of **ten** numbers from the set of numbers **1, 2, . . . , 100** (i.e. from the *vector* `1:100`), type:

```
sample(x = 1:100, size = 10)                    # 1:100 is 1, 2, 3, ..., 100

##  [1] 50 78  1 47 52 70 84 99  7 91
```

and to randomly select **one** of the **26 letters** of the alphabet, using the built-in `letters` `"character"` vector, type:

```
sample(x = letters, size = 1)                    # letters are "a", "b", "c", ..., "z"

## [1] "j"
```

- To **resample** the numbers **1, 2, 3, 4, 5** (i.e. sample from them **with replacement**), set `replace = TRUE` in `sample()`:

```
sample(x = 1:5, replace = TRUE)                    # 1:5 is 1, 2, 3, 4, 5

## [1] 1 3 3 1 4
```

and to **shuffle** them (i.e. place them in a **random order**), set `size = 5` in `sample()`:

```
sample(x = 1:5, size = 5)                    # 1:5 is 1, 2, 3, 4, 5

## [1] 5 3 1 4 2
```

13.2.2 Duplicating a Random Sample Using `set.seed()`

- This function enables us to *reproduce* a set of random numbers later:

```
set.seed()      # Set the random seed, which determines which numbers
                # will be generated by R's random number generator.
```

- `set.seed()` takes a positive integer argument `seed` (any value will do) that determines which numbers will be generated by R's random number generator.
- For example, the second call to `sample()` below generates the *same* ten random numbers as the first one:

```
set.seed(15)
sample(x = 1:100, size = 10)

##  [1] 37 34 38 49  5 89 76 84 65 12
```

```
set.seed(15)
sample(x = 1:100, size = 10)

##  [1] 37 34 38 49  5 89 76 84 65 12
```

13.2.3 Generating Uniform and Normal Random Numbers

- In simulations, we induce **random variation** in our *simulated data* so that it mimics *real data*.
- Two types of **random numbers** are widely used:
  - A ***uniform(a, b) random number*** is one that's **equally likely** to fall anywhere in the interval (a, b).
  - A ***normal(μ, σ) random number*** is one from the ***normal distribution*** (bell curve) with **mean μ** and **standard deviation σ**:

    For a **normal** distribution:
    - μ determines where the curve is centered.
    - σ determines how spread out the curve is: The curve extends about three σs to the left of μ and about three σs to the right of μ (total spread = six σs).

    The value of a **normal** random variable is **more likely** to fall near the **center** of the bell curve and **less likely** to fall in one of its **tails**.
- To generate **uniform** or **normal** random numbers, use:

```
runif()         # Generate a random sample of size n from a uniform
                # probability distribution.
rnorm()         # Generate a random sample of size n from a normal
                # probability distribution.
```

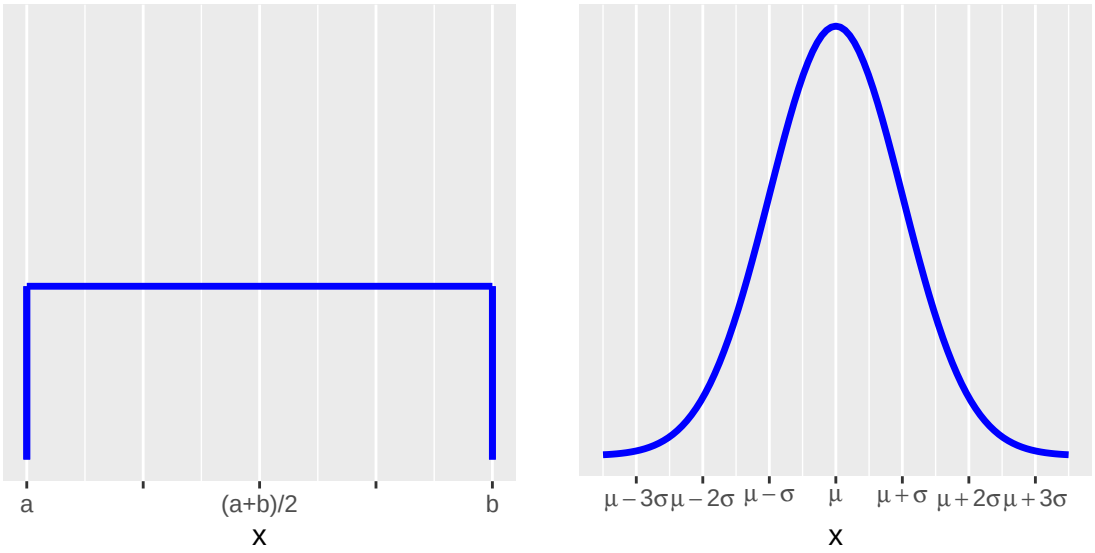

Figure 1: Uniform probability distribution (left); Normal probability distribution (right).

- For example, to generate $n = 3$ **uniform(0, 1)** values, type:

```
runif(n = 3, min = 0, max = 1)

## [1] 0.4161184 0.6947637 0.1488006
```

- And to generate $n = 3$ **normal(μ, σ)** values, with $\mu = 4$ and $\sigma = 2$, we type:

```
rnorm(n = 3, mean = 4, sd = 2)

## [1] 7.814325 6.289754 2.470939
```

13.2.4 Functions for Other Probability Distributions

- Functions analogous to `runif()` and `rnorm()` are available for other common ***probability distributions***. Here are some of them:

```
runif()         # Uniform random numbers
rnorm()         # Normal random numbers
rexp()          # Exponential random numbers
rgamma()        # Gamma random numbers
rlnorm()        # Lognormal random numbers
rt()            # T random numbers
rf()            # F random numbers
rchisq()        # Chi-squared random numbers
rbinom()        # Binomial (and Bernoulli) random numbers
rpois()         # Poisson random numbers
rnbinom()       # Negative binomial random numbers
rgeom()         # Geometric random numbers
rhyper()        # Hypergeometric random numbers
```

- For example, `rbinom()` can be used to to generate a sequence of **dichotomous** observations that are **1** with probability `prob` and **0** otherwise:

```
rbinom(n = 10, prob = 0.5, size = 1)

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

The output above shows outcomes of **ten** (`n = 10`) **"flips"** of a **fair "coin"** (`prob = 0.5`). Specifying `size = 1` specifies that only **one "coin"** should be tossed at a time (thereby giving **dichotomous** outcomes).

`prob` can be a *vector* of probabilities. For example, below, outcomes in the sequence are increasingly likely be **1** (i.e. the **"coin flips"** become increasingly likely to land **"heads"**):

```
my.probs <- seq(from = 0.05, to = 0.95, by = 0.1)
my.probs

##  [1] 0.05 0.15 0.25 0.35 0.45 0.55 0.65 0.75 0.85 0.95

rbinom(n = 10, prob = my.probs, size = 1)

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

- ***Cumulative probabilities***, ***percentiles*** (i.e. ***quantiles***), and ***probability density*** or ***mass*** values can be obtained from these probability distributions using functions whose names are prefaced by `p`, `q`, and `d` in place of `r` (e.g. `punif()`, `qunif()`, and `dunif()`).

Section 13.2 Exercises

Exercise 1 Recall that `set.seed()` is used to reproduce a set of random values later.

a) Use `set.seed()` to set the "seed" to a value (any positive integer will do). Then use `sample()` to generate $n = 5$ random numbers from **1, 2, ..., 100**. Report your R commands.

b) Now set the "seed" again (to the same value), then use `sample()` again to produce $n = 5$ random numbers from **1, 2, ..., 100**. Confirm that this *regenerates* the *same* five values you got in part *a*.

c) What would've happened in part *b* if you hadn't set the "seed" prior to the call to `sample()`? Try it.

Exercise 2 This problem concerns **uniform** random numbers.

a) Use `runif()` to generate $n = 1,000$ random values between **0** and **1**. Save them in a vector `x`. Report your R command(s).

b) Produce a **histogram** of the simulated data:

```
ggplot(data.frame(x = x), mapping = aes(x = x)) +
  geom_histogram(binwidth = 0.1, boundary = 0.0, fill = "blue")
```

Are the simulated values fairly evenly spread over the interval from **0** to **1**?

Exercise 3 This problem concerns **normal** random numbers.

a) Use `rnorm()` to generate $n = 1,000$ random values from the **normal**(μ, σ) curve, with with $\mu = 0$ and $\sigma = 1$. Save them in a vector `x`. Report your R command(s).

b) Produce a **histogram** of the simulated data:

```
ggplot(data.frame(x = x), mapping = aes(x = x)) +
  geom_histogram(binwidth = 0.5, boundary = -3.5, fill = "blue")
```

Do the simulated values follow a (approximately) bell-shaped pattern from **-3** to **3**?

Exercise 4 This problem concerns **dichotomous** random numbers.

a) Use `rbinom()` to generate a **dichotomous** (**0** or **1**) sequence of **ten** (`n = 10`) **"flips"** of a **biased "coin"** that lands **"heads"** (i.e. results in a **1**) with probability **0.7** (`prob = 0.7`) and **"tails"** (**0**) otherwise. Report your R command(s).

b) Now use `rbinom()` to generate a sequence of **ten** (`n = 10`) **dichotomous** outcomes that are *decreasingly* likely be **1** according to the following **probabilities**:

```
my.probs <- seq(from = 0.95, to = 0.05, by = -0.1)
```

Report your R command(s).

13.3 Simulating Variability

- We'll simulate data from a **logistic regression model** to investigate the performance of **parameter estimates**, the (estimated) intercept $\boldsymbol{b_0}$ and slope $\boldsymbol{b_1}$.

  Recall, in a **logistic regression model**, the response variable $\boldsymbol{Y}$ is **dichotomous**, which we code as **0** and **1**.

  Defining a function $\boldsymbol{p(X)}$ as

  $$p(X) \;=\; P(Y = 1 \text{ when the value of the explanatory variable is } X)\,,$$

  the **fitted logistic regression model** has the form:

  $$p(X) \;=\; \frac{e^{b_0 + b_1 X}}{1 + e^{b_0 + b_1 X}}$$

  (were $\boldsymbol{e}$ is the *exponential constant*, $e = 2.718282...$).

  Below, we generate $n = 5,000$ *dichotomous* observations from a **logistic regression model**, with (**true**) **parameter values** intercept $\boldsymbol{\beta_0 = -2}$ and slope $\boldsymbol{\beta_1 = 0.8}$, at values of the explanatory variable (predictor) generated from a normal(μ, σ) distribution with mean $\mu = 4$ and standard deviation $\sigma = 2$.

```
# Set the seed (to allow regenerating the simulated values later):
set.seed(50)

# Generate 5,000 values of the explanatory variable X:
x <- rnorm(n = 5000, mean = 4, sd = 2)

# Generate the 5,000 corresponding probabilities p(X):
true_probs <- exp(-2 + 0.8 * x) / (1 + exp(-2 + 0.8 * x))

# Simulate the 5,000 corresponding dichotomous response values Y:
y <- rbinom(n = 5000, size = 1, prob = true_probs)

# Make data frame of simulated data:
sim.data <- data.frame(X = x, Y = y)
```

```
head(sim.data)

##           X Y
## 1 5.0993398 1
## 2 2.3167925 1
## 3 4.0659959 1
## 4 5.0482994 1
## 5 0.5447918 0
## 6 3.4442709 1
```

The **simulated** data (shown above) and **fitted logistic regression model** are plotted below.

```
ggplot(data = sim.data, mapping = aes(x = X, y = Y)) +
  geom_point() +
  labs(title = "Y Versus X, Simulated Data", x = "X", y = "Y") +
  geom_smooth(method = "glm",
              method.args = list(family = "binomial"),
              se = FALSE) +
  scale_y_continuous(breaks = seq(-0.25, 1.25, 0.25),
                     labels = seq(-0.25, 1.25, 0.25),
                     limits = c(-0.25, 1.25))
```

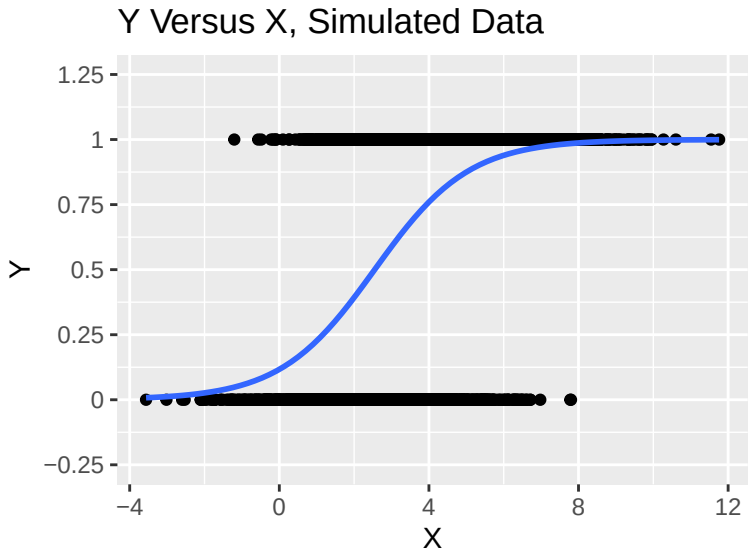

Figure 2

Upon **fitting** a **logistic regression model** to the **simulated data**, the resulting **parameter estimates** are in the output of `summary()` below.

```
my.logreg <- glm(Y ~ X, data = sim.data, family = "binomial")

summary(my.logreg)

##
## Call:
## glm(formula = Y ~ X, family = "binomial", data = sim.data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.8920  -0.7101   0.4013   0.7290   2.4610
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.02278    0.09083  -22.27   <2e-16 ***
## X            0.79369    0.02528   31.40   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 6194.2  on 4999  degrees of freedom
## Residual deviance: 4619.1  on 4998  degrees of freedom
## AIC: 4623.1
##
## Number of Fisher Scoring iterations: 5
```

The **estimates**, $b_0 = -2.023$ and $b_1 = 0.794$, are close to the **true parameter values** $\beta_0 = -2$ and $\beta_1 = 0.8$.

Section 13.3 Exercises

Exercise 5 This exercise involves simulating data from a **logistic regression model** to investigate the performance of **parameter estimates**, the (estimated) intercept b_0 and slope b_1.

a) After setting the seed to **57** (so everyone gets the same results):

```
# Set the seed (so everyone gets the same results):
set.seed(57)
```

generate $n = 1,000$ *dichotomous* observations from a **logistic regression model**, with (**true**) **parameter values** intercept $\beta_0 = 4$ and slope $\beta_1 = -1$, at values of the explanatory variable generated from a **uniform(0, 10)** distribution. Report your R commands.

You should end up with this:

```
head(sim.data)

##           X Y
## 1 2.4391435 0
## 2 5.1294954 0
## 3 0.3862843 1
## 4 1.6617658 1
## 5 7.3320525 0
## 6 6.6280162 0
```

b) **Fit** a **logistic regression model** to your **simulated data**, and report the resulting **parameter estimates** b_0 and b_1 from the output of `summary()`. Are they close to the **true parameter values** $\beta_0 = 4$ and $\beta_1 = -1$?