Computational Skills for Biostatistics I: Lecture 4

Amy Willis, Biostatistics, UW

24 April, 2019

Critical pieces in developing a statistical method

- Define the problem
- Come up with a solution
- Investigate the performance of the solution
    - Compare to existing methods, if they exist
- Explore the properties of the solution
- Describe the method and your results

–> –> –> –> –>

Focus of today

- Define the problem
- Come up with a solution
- **Investigate the performance of the solution**
  - **Compare to existing methods, if they exist**
- Explore the properties of the solution
- Describe the method and your results

Simulation studies

Investigating the performance of a solution typically involves the following

- Generate data...
  - ...according to a model...
  - ...with some parameters
- apply your estimator/prediction, and competitors
- evaluate the performance

Can also be done theoretically – see your other classes!

Exercise (5 minutes)

In groups, investigate the performance of the least-squares estimate of a regression slope.

In five minutes, I'm going to ask you to tell the class what approach you were implementing.

Exercise (5 minutes)

- How did you design your study?
- What parameters did you vary?

My approach

(released after class)

My approach

```
nsims <- 100
n <- 10
beta <- 5
sigma_squared <- 10
betahat <- vector("numeric", nsims)
for (i in 1:nsims) {
  x <- runif(n, -1, 1)
  y <- rnorm(n, beta*x, sigma_squared)

  my_lm_fit <- lm(y ~ x - 1)
  betahat[i] <- my_lm_fit$coef
}
mean((betahat-beta)^2)
```

```
## [1] 32.48183
```

My approach

```r
library(tidyverse)
library(magrittr)
nsims <- 100
n <- 10
betas <- seq(from = 5, to = 50, by = 5)
sigma_squared <- 10
results <- tibble("iteration" = NA,
                  "sigma_squared" = NA,
                  "betahat"= NA,
                  "beta" = NA)
```

My approach

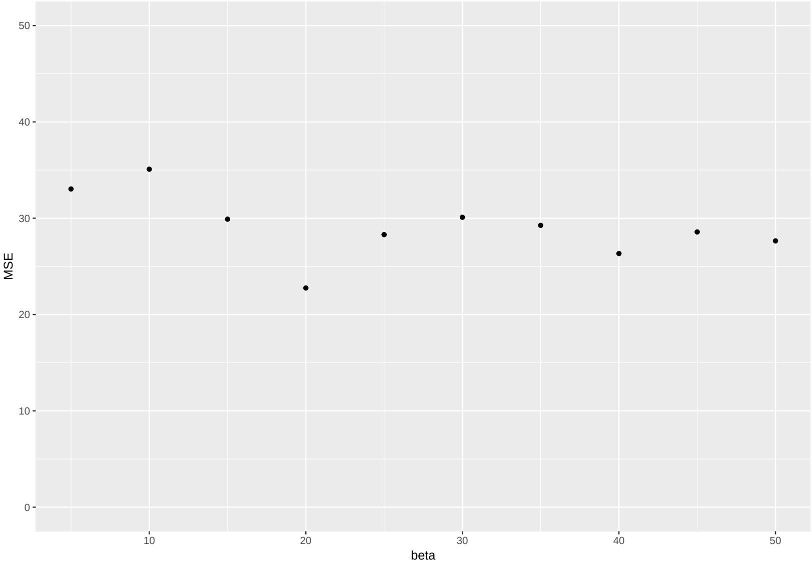
```
for (k in 1:length(betas)) {
  beta <- betas[k]
  for (i in 1:nsims) {
    x <- runif(n, -1, 1)
    y <- rnorm(n, beta*x, sigma_squared)

    my_lm_fit <- lm(y ~ x - 1)
    results %<>% add_row("iteration" = i,
                          "sigma_squared" = sigma_squared,
                          "betahat" =  my_lm_fit$coef,
                          "beta" = beta)
  }
}
results %<>% filter(!is.na(iteration))
```

My approach

```
results %>%
  group_by(beta) %>%
  mutate(sq_error = (betahat - beta)^2) %>%
  summarise(MSE = mean(sq_error)) %>%
  ggplot(aes(x = beta, y = MSE)) +
  geom_point() +
  ylim(0, 50)
```

My approach



Does this surprise anyone?

Expanding the simulation

- What happens when I want `n` to vary?
- What happens when I want `sigma_squared` to vary?
- The distribution of the `x`'s?
- What happens when I want to compare to other estimates? (e.g., ridge regression)

Welcome to reality: always more

There are always more simulations that you need to do!

- change more parameters
- change the data generating process
- someone publishes a method! Compare to theirs
- Reviewer asks for their favourite evaluation criterion...

Welcome to reality: coding for succession

- How would you write simulation code if you were planning to share it with someone else?
  - co-author sharing (e.g., advisor, other students)
  - What if that person were going to build on your simulation?
  - interested readership (future competitor/extension method)
  - future forgetful self

Key Observation

Simulations

- are highly formulaic in nature
- reuse *a lot of code*
- formulaic means this can be **standardized**

How do we minimise the time spent rewriting and reorganising simulations?

Introducing… THE `SIMULATOR`

`simulator` is an amazing R package

```
library(simulator)
```

Great things about the simulator

- easy to
  - add more parameters
  - change the data generating process
  - change/adapt your method
  - add a comparison method
  - add an evaluation criterion

Great things about the simulator

- reproducible
- parallelisable
- prevents mistakes
  - errors from copying and pasting code
  - accidentally using parameters in your estimates

The simulator can help you understand the methods development process: it forces you to think through `what` your method needs and `how` you will evaulate it

simulator: Running a simulation

```
first_sim <- new_simulation("least-squares-estimates",
                            "What's up with LSEs") %>%
  generate_model(linear_model,
                 n = 10,
                 sigma_sq = as.list(seq(5, 15, by = 5)),
                 x_width = 1,
                 beta = 1,
                 vary_along = "sigma_sq") %>%
  simulate_from_model(nsim = 10) %>%
  run_method(list(lse)) %>%
  evaluate(list(squared_error))
```

```
## ..Created model and saved in slm/beta_1/n_10/sigma_sq_5/
## ..Created model and saved in slm/beta_1/n_10/sigma_sq_10
## ..Created model and saved in slm/beta_1/n_10/sigma_sq_15
## ..Simulated 10 draws in 0 sec and saved in slm/beta_1/n_
## ..Simulated 10 draws in 0 sec and saved in slm/beta_1/n_
## ..Simulated 10 draws in 0 sec and saved in slm/beta_1/n
```

simulator: Running a simulation

Let's dive into where those pieces all come from!

simulator: Defining models

```
linear_model <- function(n, beta, x_width, sigma_sq) {
  new_model(
    name = "slm",
    label = sprintf("n = %s, beta = %s,
                             x_width = %s, sigma_sq = %s",
                     n, beta, x_width, sigma_sq),
    params = list(beta = beta, x_width = x_width,
                   sigma_sq = sigma_sq, n = n),
    simulate = function(n, beta, x_width, sigma_sq, nsim){
      sim_list <- list()
      for (i in 1:nsim) {
        x <- runif(n, -x_width, x_width)
        y <- beta*x + rnorm(n, 0, sigma_sq)
        sim_list[[i]] <- list("x" = x,
                               "y" = y)
      }
      return(sim_list)
    })}
```

simulator: Defining methods

```r
lse <- new_method("lse", "LSE",
                  method = function(model, draw) {
                    yy <- draw$y
                    xx <- draw$x
                    fit <- lm(yy ~ xx - 1)
                    list(betahat = fit$coef)
                  })
```

simulator: Defining methods

```
squared_error <-
  new_metric("squared_error",
             "Squared error",
             metric = function(model, out) {
               (out$betahat - model$beta)^2
            })
```

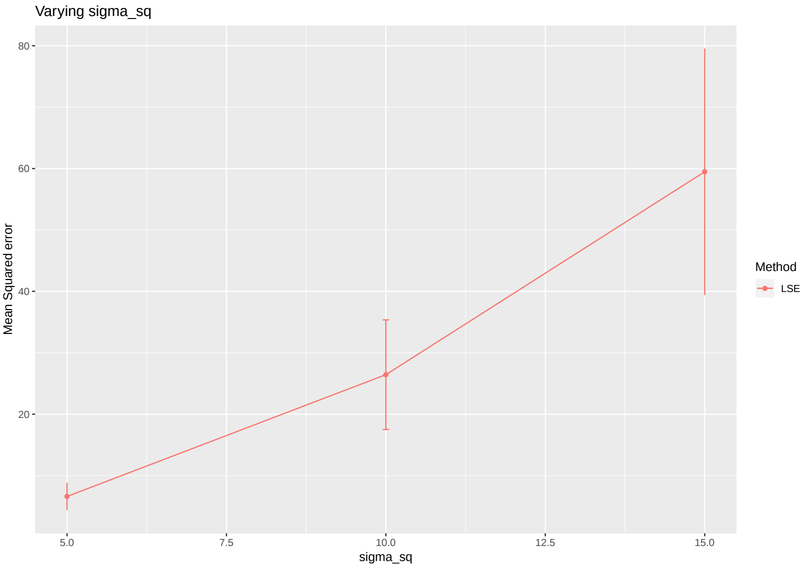
simulator: Running a simulation

```
first_sim <- new_simulation("least-squares-estimates",
                            "What's up with LSEs") %>%
  generate_model(linear_model,
                 n = 10,
                 sigma_sq = as.list(seq(5, 15, by = 5)),
                 x_width = 1,
                 beta = 1,
                 vary_along = "sigma_sq") %>%
  simulate_from_model(nsim = 10) %>%
  run_method(list(lse)) %>%
  evaluate(list(squared_error))
```

```
## ..Created model and saved in slm/beta_1/n_10/sigma_sq_5/
## ..Created model and saved in slm/beta_1/n_10/sigma_sq_10
## ..Created model and saved in slm/beta_1/n_10/sigma_sq_15
## ..Simulated 10 draws in 0 sec and saved in slm/beta_1/n_
## ..Simulated 10 draws in 0 sec and saved in slm/beta_1/n_
## ..Simulated 10 draws in 0 sec and saved in slm/beta_1/n
```

simulator: Plotting the results

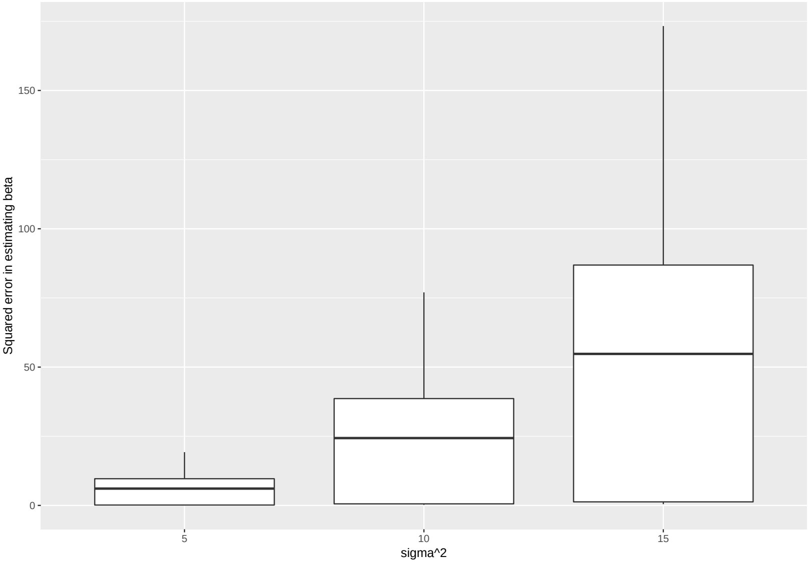
```
plot_eval_by(first_sim, "squared_error", varying = "sigma_s
```



simulator: Plotting the results (more nicely)

```
ev_df <- first_sim %>% evals %>% as.data.frame
model_df <- first_sim %>% model %>% as.data.frame
right_join(model_df, ev_df, by = c("name" = "Model")) %>%
  as_tibble %>%
  mutate(sigma_sq_f = factor(sigma_sq)) %>%
  ggplot(aes(x = sigma_sq_f, y = squared_error)) +
  geom_boxplot() +
  xlab("sigma^2") +
  ylab("Squared error in estimating beta")
```

simulator: Plotting the results (more nicely)



Oh no! Your adviser wants to see how the results change with x_width

```
simulated_data <- new_simulation(
  "lses-x",
  "How do LSEs change with the range of x?") %>%
  generate_model(linear_model,
                 n = 10,
                 beta = 5,
                 x_width = as.list(seq(2, 10, by = 2)),
                 sigma_sq = 1,
                 vary_along = "x_width") %>%
  simulate_from_model(nsim = 10)
```

```
## ..Created model and saved in slm/beta_5/n_10/sigma_sq_1/
## ..Created model and saved in slm/beta_5/n_10/sigma_sq_1/
## ..Created model and saved in slm/beta_5/n_10/sigma_sq_1/
## ..Created model and saved in slm/beta_5/n_10/sigma_sq_1/
## ..Created model and saved in slm/beta_5/n_10/sigma_sq_1/
```

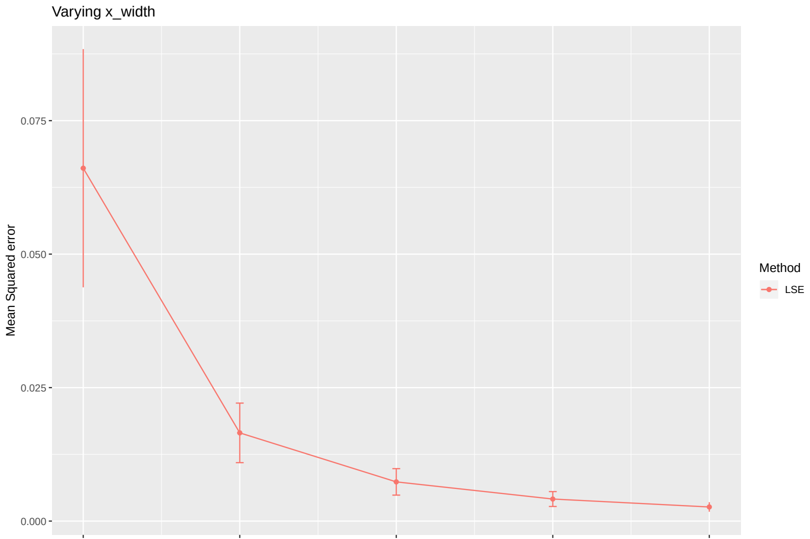
Oh no! Your adviser wants to see how the results change with `x_width`

```
evaluation_plot <- simulated_data %>%
  run_method(list(lse)) %>%
  evaluate(list(squared_error)) %>%
  plot_eval_by("squared_error", varying = "x_width")
```

```
## ..Performed LSE in 0 seconds (on average over 10 sims)
## ..Performed LSE in 0 seconds (on average over 10 sims)
## ..Performed LSE in 0 seconds (on average over 10 sims)
## ..Performed LSE in 0 seconds (on average over 10 sims)
## ..Performed LSE in 0 seconds (on average over 10 sims)
## ..Evaluated LSE in terms of Squared error, Computing ti
## ..Evaluated LSE in terms of Squared error, Computing ti
## ..Evaluated LSE in terms of Squared error, Computing ti
## ..Evaluated LSE in terms of Squared error, Computing ti
## ..Evaluated LSE in terms of Squared error, Computing ti
```

Oh no! Your adviser wants to see how the results change with `x_width`

```
evaluation_plot
```



Oh no! Your adviser wants more simulations!

Easy! Increase `nsim` - Better way: to avoid overwriting existing simulations, run `simulate_from_model` again with new indices

Oh no, it's taking forever

Easy! Distribute it across multiple cores... or a cluster!

```
bigger_sim <- generated_model %>%
  simulate_from_model(nsim = 40,
                      index = 5:8,
                      parallel = list(socket_names = 4)) %>
  run_method(list(lse)) %>%
  evaluate(list(squared_error))
```

Oh no! Someone just published a new method!

Adding another `run_method` does not overwrite your results

```
save_simulation(sim=results)
load_simulation("results") %>%
  run_method(list(the_new_method)) %>%
  evaluate(list(squared_error))
```

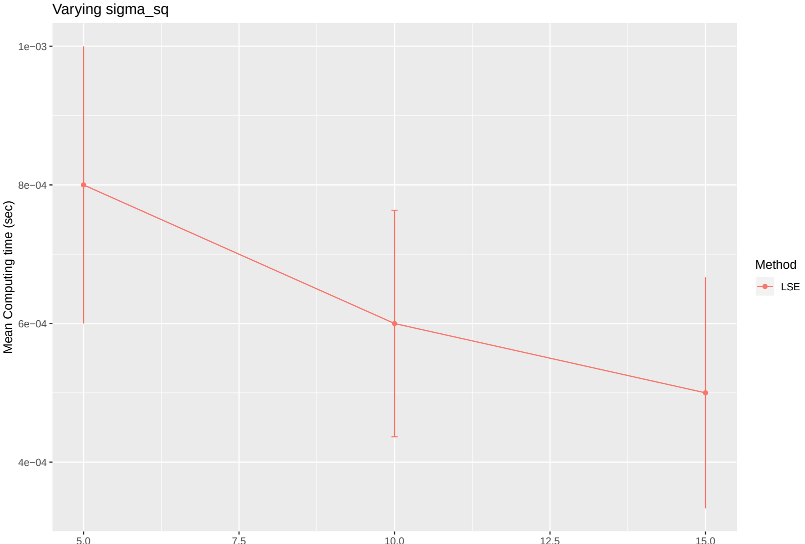
Oh no! Your adviser prefers L6-loss

```r
load_simulation("results") %>%
  evaluate(list(squared_error, wacky_evaluation))
```

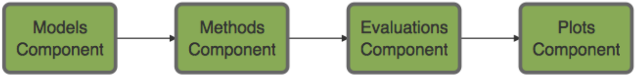
Oh no! A reviewer asks for computation time

Computational time is calculated by default

```
plot_eval_by(first_sim, "time", varying = "sigma_sq")
```



Pipeline of interlocking components



- You "plug in" Models, Methods, and Metrics; simulator does the rest!
- Modularity faciliates code sharing
- When Models, Methods and Metrics are defined, they are also labeled – accessed by plot and table functions downstream

Solicited and unsolicited feedback

- Bryan Martin: “I highly recommend the simulator for reproducibility, organization, and speed”

Solicited and unsolicited feedback

- Alex Paynter: “simulator’s infrastructure encourages a big picture view of how code fits together (models, methods, evaluations), and simulations are highly human-readable. Coding errors are harder to write when the structural details are handled by the package, and looking into the details of a simulation after it has run is relatively easy.”

Solicited and unsolicited feedback

- Kendrick Li: "I love simulator. simulator makes me happy."

Getting started with the simulator

```
create("test_idea")
```

```
## New simulation template created!  Go to test_idea/main.R
```

Then go to the directory `test_idea` to fill out the details.

Results saved to file

- no need to rerun parts of simulation that haven't changed
- results saved at each stage of pipeline – allows one to examine intermediate stages for better understanding of results
- organized file structure (though user never has to explicitly learn the particulars since there are a series of simulator functions that take care of loading files)

Simulation object

- `Simulation` object (S4 class) is passed through pipeline
- gets fed through the components; accumulates "record" of simulation
- **Important:** consists of *references* to objects – not the objects themselves
- makes `Simulation` objects fast-to-load and easy to work with

Parallel processing and streams

- Jacob Bien: "uses L'Ecuyer-CMRG generator to get separate streams"
- Identical results whether run in sequence or in parallel

Unified interface for making plots and tables

```
tabulate_eval(sim, "sqr_err")
```

```
plot_eval(sim, "sqr_err")
```

Also automated report generation via `knitr`

References

https://github.com/jacobbien/simulator

Vignettes:

- Getting started with the simulator
- James-Stein estimator
- Benjamini-Hochberg procedure
- Lasso
- Elastic net

Many thanks to Jacob Bien for teaching materials and writing such a great package!

Coming up

- Homework 4 due next Wednesday at the usual time in the usual way; posted soon