CSSS508, Week 6

Loops

Chuck Lanfear

Nov 4, 2020

Updated: Nov 11, 2020

Bad Repetition

If someone doesn't know better, they might find the means of variables in the `swiss` data by typing in a line of code for each column:

```
mean1 <- mean(swiss$Fertility)
mean2 <- mean(swiss$Agriculture)
mean3 <- mean(swissExamination)
mean4 <- mean(swiss$Fertility)
mean5 <- mean(swiss$Catholic)
mean5 <- mean(swiss$Infant.Mortality)
c(mean1, mean2 mean3, mean4, mean5, man6)
```

Can you spot the problems?

How upset would they be if the `swiss` data had 200 columns instead of 6?

Good Repetition

You will learn general solutions today—and better but more specific ones next week:

```
swiss_means <- setNames(numeric(ncol(swiss)), colnames(swiss))
for(i in seq_along(swiss)) {
    swiss_means[i] <- mean(swiss[[i]])
}
swiss_means
```

```
##        Fertility      Agriculture      Examination        Education
##             70.1             50.7             16.5             11.0
##         Catholic Infant.Mortality
##             41.1             19.9
```

`setNames()` adds `names` (second argument) to its first argument.
`numeric()` creates a numeric vector of length equal to its first argument.

Don't Repeat Yourself (DRY)!

The **DRY** idea: Computers are much better at doing the same thing over and over again than we are.

Writing code to repeat tasks for us reduces the most common human coding mistakes.

It also *substantially* reduces the time and effort involved in processing large volumes of data.

Lastly, compact code is more readable and easier to troubleshoot.

The Agenda: Programming

Today:

- `for()` and `while()` **loop** programming (general methods)
- Vectorization to *avoid* loops

Next week:

- Writing your own **functions**!
- Looping methods based on functions

What is a Loop?

I'll be bad and cite Wikipedia:

> A loop is a sequence of statements which is specified once but which may be carried out several times in succession. The code "inside" the loop is obeyed a specified number of times, or once for each of a collection of items, or until some condition is met, or indefinitely." (Wikipedia)

`for()` Loops

The `for()` Loop

`for()` loops are the most general kind of *loop*, found in pretty much every programming language.

"**For** each of these values—in order—do *this*"

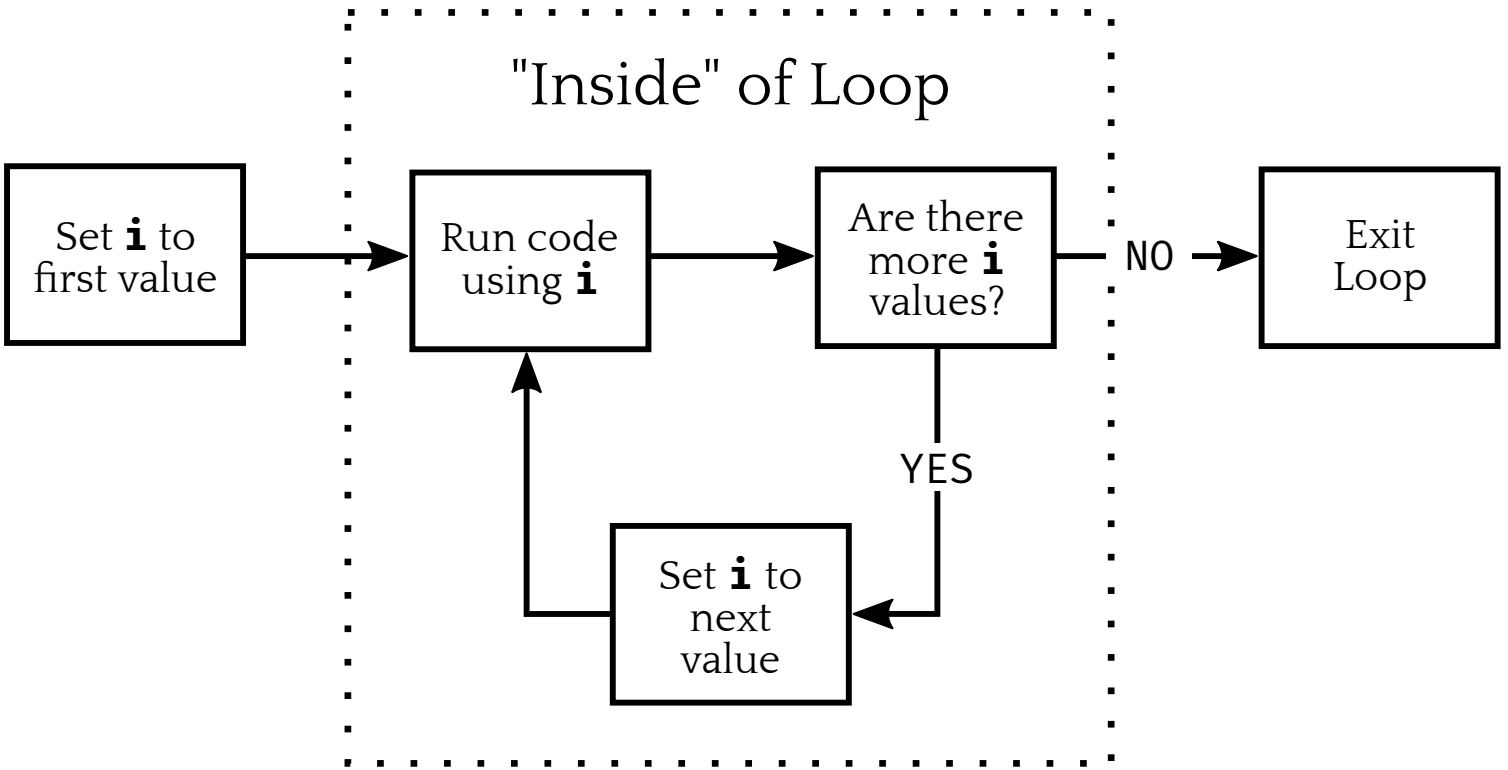
Given a set of values...

1. You set an index variable (often `i`) equal to the first value
2. Do some set of things (usually depending on current value)
3. Is there a next value?
    - *YES*: Update to next value, go back to 2.
    - *NO*: Exit loop

We are *looping* through values and repeating some actions.

`for()` Loop: Diagram

Given a set of values...



`for()` Loop: Example

```
for(i in 1:10) {
    # inside for, output won't show up without print()
    print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

Note this runs *10 separate print commands*, which is why each line starts with `[1]`.

These Do the Same Thing

```
for(i in 1:3) {
    print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
```

```
i <- 1
print(i^2)
i <- 2
print(i^2)
i <- 3
print(i^2)
```

```
## [1] 1
## [1] 4
## [1] 9
```

Iteration Conventions

- We call what happens in the loop for a particular value one **iteration**.
- Iterating over indices `1:n` is *very* common. `n` might be the length of a vector, the number of rows or columns in a matrix or data frame, or the length of a list.
- Common notation: `i` is the object that holds the current value inside the loop.
  - If loops are nested, you will often see `j` and `k` used for the inner loops.
  - This notation is similar to indexing in mathematical symbols (e.g $\sum_{i=1}^{n}$)
- Note `i` (and `j`,`k`, etc) are just normal objects. You can use any other names you want.
  - Ex: When iterating over rows and/or columns, I often use `row` and/or `col`!

Iterate Over Characters

What we iterate over doesn't have to be numbers `1:n` or numbers at all! You can also iterate over a character vector in R:

```
some_letters <- letters[4:6] # Vector of letters d,e,f
for(i in some_letters) {
    print(i)
}
```

```
## [1] "d"
## [1] "e"
## [1] "f"
```

```
i # in R, this will exist outside of the loop!
```

```
## [1] "f"
```

`seq_along()` and Messages

`seq_along(x)` creates an integer vector equal to `1:length(x)`.

When you want to loop over something that isn't numeric but want to use a numeric index of where you are in the loop, `seq_along` is useful:

```
for(a in seq_along(some_letters)) {
    print(paste0("Letter ", a, ": ", some_letters[a]))
}
```

```
## [1] "Letter 1: d"
## [1] "Letter 2: e"
## [1] "Letter 3: f"
```

```
a # The object `a` contains the number of the last iteration
```

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

Pre-Allocation

Usually in a `for()` loop, you aren't just printing output, but want to store results from calculations in each iteration somewhere.

To do that, figure out what you want to store, and **pre-allocate** an object of the right size as a placeholder (typically with missing values as placeholders).

Examples of what to pre-allocate based on what you store:

- Single numeric value per iteration: `numeric(num_of_iters)`
- Single character value per iteration: `character(num_of_iters)`
- Single true/false value per iteration: `logical(num_of_iters)`
- Numeric vector per iteration: `matrix(NA, nrow = num_of_iters, ncol = length_of_vector)`
- Some complicated object per iteration: `vector("list", num_of_iters)`

Pre-Allocation: Numeric

```
iters <- 10 # Set number of interations
output <- numeric(iters) # Pre-allocate numeric vector

for(i in 1:iters) { # Run code below iters times
    output[i] <- (i-1)^2 + (i-2)^2
}
output # Display output
```

```
##  [1]   1   1   5  13  25  41  61  85 113 145
```

Steps:

1. Set a number of iterations
2. Pre-allocated a numeric vector of that length
3. Ran ten iterations where the output is a mathematical function of each iteration number.

`setNames()`

The function `setNames()` can be handy for pre-allocating a named vector:

```
(names_to_use <- paste0("iter ", letters[1:5]))
```

```
## [1] "iter a" "iter b" "iter c" "iter d" "iter e"
```

```
# without setNames:
a_vector <- numeric(5)
names(a_vector) <- names_to_use

# with setNames: first arg = values, second = names
(a_vector <- setNames(numeric(5), names_to_use))
```

```
## iter a iter b iter c iter d iter e
##      0      0      0      0      0
```

Extended Regression Example

Premise

Suppose we have some data that we want to try fitting several regression models to.

We want to store the results of fitting each regression as elements of a list so that we can compare them.

To do this consistently, we'll write a loop. That way no matter if we had 2 models or 200 models, we could use the same code.

After we do this, we'll try something more advanced with loops: **Cross-validating** regressions to get an estimate of their true accuracy in predicting values out-of-sample.

Simulating Data

Let's simulate some fake data for this using the `rnorm()` function to generate random values from a normal distribution.

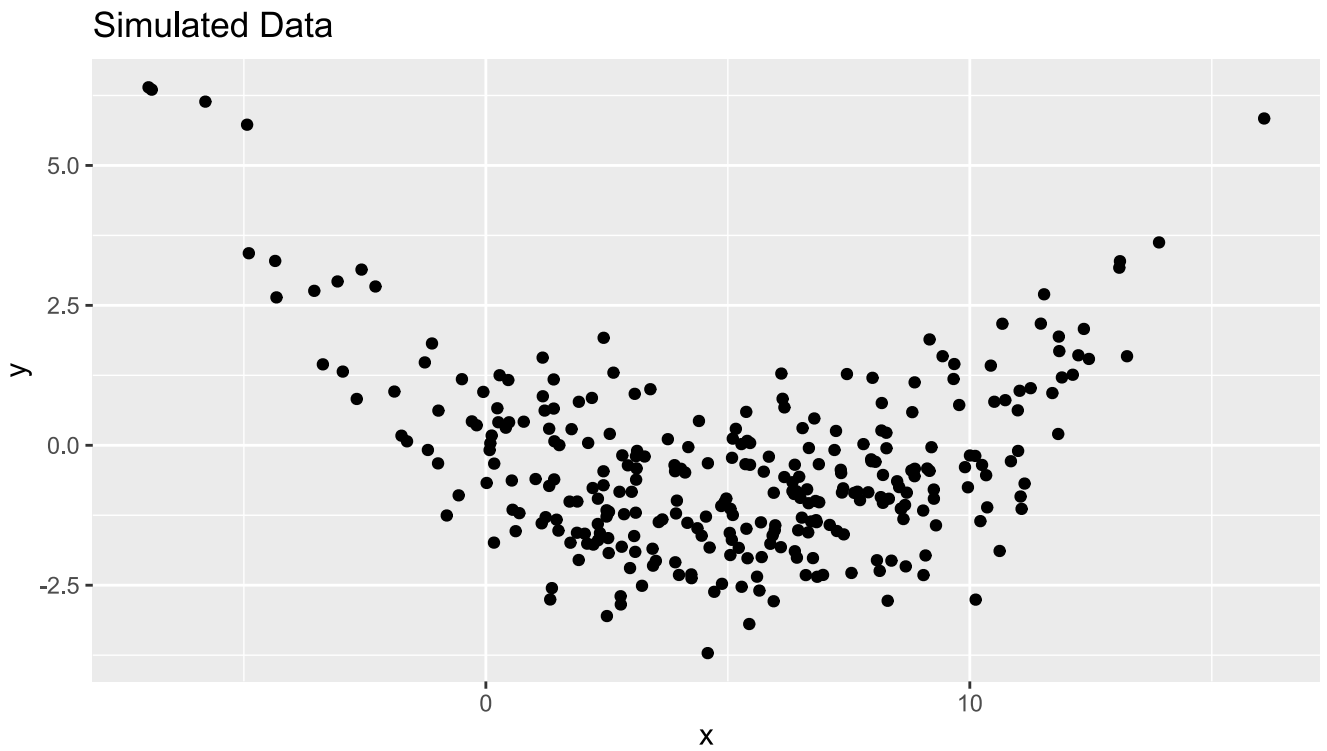
```
set.seed(98105) # Making sure values always the same
n <- 300
x <- rnorm(n, mean = 5, sd = 4)
sim_data <-
  data.frame(x = x,
             y = -0.5 * x + 0.05 * x^2 + rnorm(n, sd = 1))
```

This generates a dataframe of 300 observations where `y` is dependent on `x`, with some uncorrelated, normally-distributed residual (from `rnorm()`).

If you followed the 2014 scandal in political science when a grad student faked data for a publication in *Science*, it is believed he used the `rnorm()` function to add noise to an existing dataset to get his values.

Plot of `sim_data`

```
ggplot(data = sim_data, aes(x = x, y = y)) +
  geom_point() +
  ggtitle("Simulated Data")
```



Candidate Regression Models

Let's say we want to consider several different regression models to draw trendlines through these data:

- **Intercept Only:** draw a horizontal line that best fits the `y` values.

$$E[y_i|x_i] = \beta_0$$

- **Linear Model:** draw a line that best fits the `y` values as a function of `x`.

$$E[y_i|x_i] = \beta_0 + \beta_1 \cdot x_i$$

- **Quadratic Model:** draw a quadratic curve that best summarizes the `y` values as a function of `x`.

$$E[y_i|x_i] = \beta_0 + \beta_1 \cdot x_i + \beta_2 \cdot x_i^2$$

- **Cubic Model:** draw a cubic curve that best summarizes the `y` values as a function of `x`.

$$E[y_i|x_i] = \beta_0 + \beta_1 \cdot x_i + \beta_2 \cdot x_i^2 + \beta_3 \cdot x_i^3$$

Preallocating a List

Let's make a *named character vector* for the formulas we'll use in `lm()`:

```
models <- c("intercept only" = "y ~ 1", # Name on left, formula on right
            "linear"         = "y ~ x",
            "quadratic"      = "y ~ x + I(x^2)",
            "cubic"          = "y ~ x + I(x^2) + I(x^3)")
```

Then pre-allocate a list to *store* the fitted models:

```
fitted_lms        <- vector("list", length(models)) # initialize list
names(fitted_lms) <- names(models) # give entries good names
fitted_lms # display the pre-allocated (empty) list
```

```
## $`intercept only`
## NULL
##
## $linear
## NULL
##
## $quadratic
## NULL
##
## $cubic
## NULL
```

Fitting Models in `for()` Loop

Next, we'll loop over the `models` vector and fit each one, storing it in the appropriate slot.

The `formula()` function converts a character string describing a model to a formula object readable by `lm()`:

```
for(mod in names(models)) {
    fitted_lms[[mod]] <- lm(formula(models[mod]), data = sim_data)
}
```

What this does:

For each model name (which will be referred to as `mod`)...

1. Fit a `lm()` using the formula with that name (`mod`).
2. Assign the output to the element of the list with the matching name.

Getting Predictions

To plot the fitted models, we can first get predicted `y` values from each at the `x` values in our data.

```
# initialize data frame to hold predictions
predicted_data <- sim_data
for(mod in names(models)) {
    # make a new column in predicted_data for each model's predictions
    predicted_data[[mod]] <- predict(fitted_lms[[mod]],
                                     newdata = predicted_data)
}
```

What this does:

For each named `lm()` model output...

1. Get predicted values from the associated model (`predict()`)
2. Save each of those predicted values as a *new column* in `predicted_data`.
    - Values of `mod` will be the new column names!

Predictions

```
head(predicted_data, 10)
```

```
##         x      y intercept only linear quadratic  cubic
## 1  -5.80  6.140          -0.372  0.160     4.985  4.983
## 2   9.26 -0.790          -0.372 -0.579    -0.312 -0.312
## 3   1.40  1.177          -0.372 -0.194    -0.568 -0.567
## 4   5.23 -1.833          -0.372 -0.381    -1.266 -1.266
## 5  12.24  1.608          -0.372 -0.726     1.516  1.517
## 6   8.30 -2.778          -0.372 -0.532    -0.694 -0.694
## 7   9.66  1.185          -0.372 -0.599    -0.118 -0.118
## 8  10.25 -0.351          -0.372 -0.628     0.192  0.192
## 9   9.44  1.592          -0.372 -0.588    -0.228 -0.228
## 10  1.46 -1.329          -0.372 -0.197    -0.591 -0.590
```

Gathering Predictions

We can use `tidyr::pivot_longer()` to tidy the predictions, then set the levels of the `Model` variable.

```
library(tidyr)
tidy_predicted_data <- predicted_data %>%
    pivot_longer(3:6,
                 names_to="Model",
                 values_to="Prediction") %>%
    mutate(Model = factor(Model, levels = names(models)))
head(tidy_predicted_data) # Displaying some rows
```

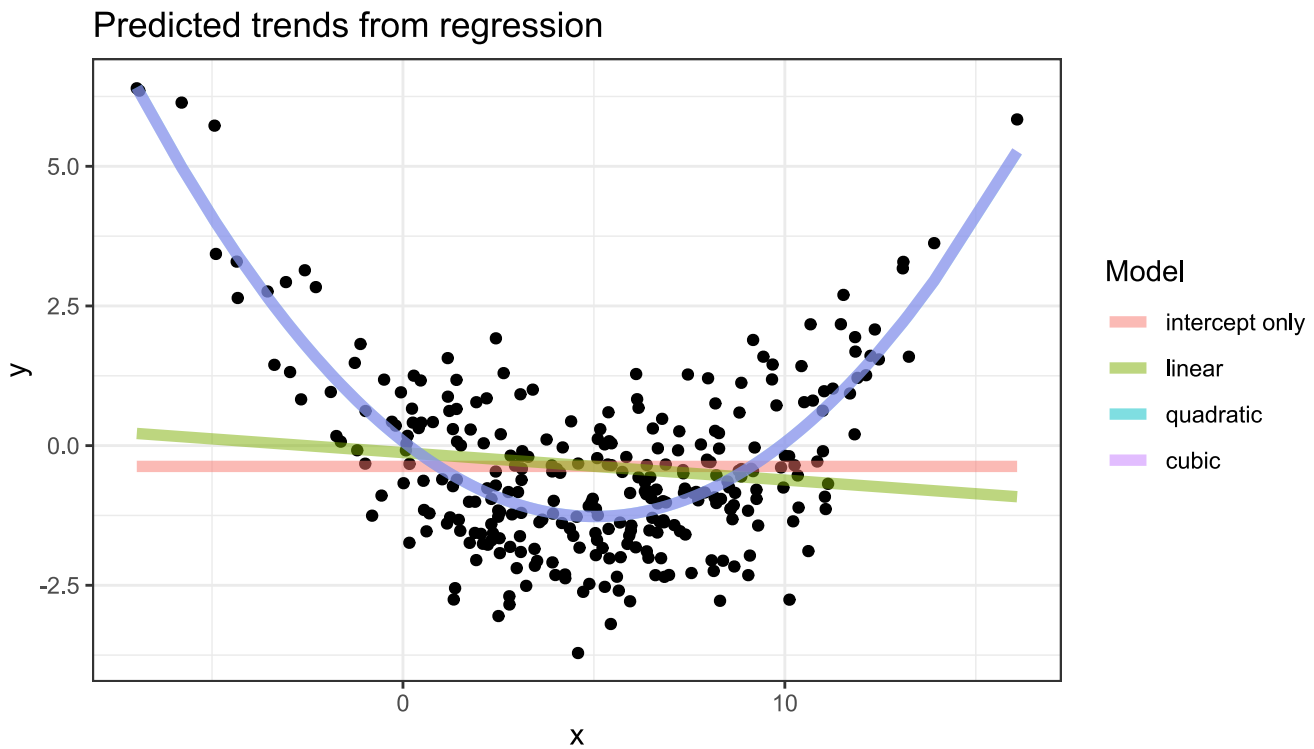
```
## # A tibble: 6 x 4
##       x      y Model          Prediction
##   <dbl>  <dbl> <fct>               <dbl>
## 1 -5.80  6.14  intercept only     -0.372
## 2 -5.80  6.14  linear             0.160
## 3 -5.80  6.14  quadratic          4.98
## 4 -5.80  6.14  cubic              4.98
## 5  9.26 -0.790 intercept only     -0.372
## 6  9.26 -0.790 linear             -0.579
```

Plotting Predictions

We'll use `ggplot2` to plot these tidied up predictions. You'll see us use multiple data sets on the same plot: Look at the `geom_line()` call.

```
ggplot() +
    geom_point(data = sim_data, # Original data as points
               aes(x = x,
                   y = y)) +
    geom_line(data = tidy_predicted_data, # Predicted data!
              aes(x     = x,
                  y     = Prediction,
                  group = Model,
                  color = Model),
              alpha = 0.5,
              size  = 2) +
    ggtitle("Predicted trends from regression") +
    theme_bw()
```

Plotted Predictions



Which looks best to you?

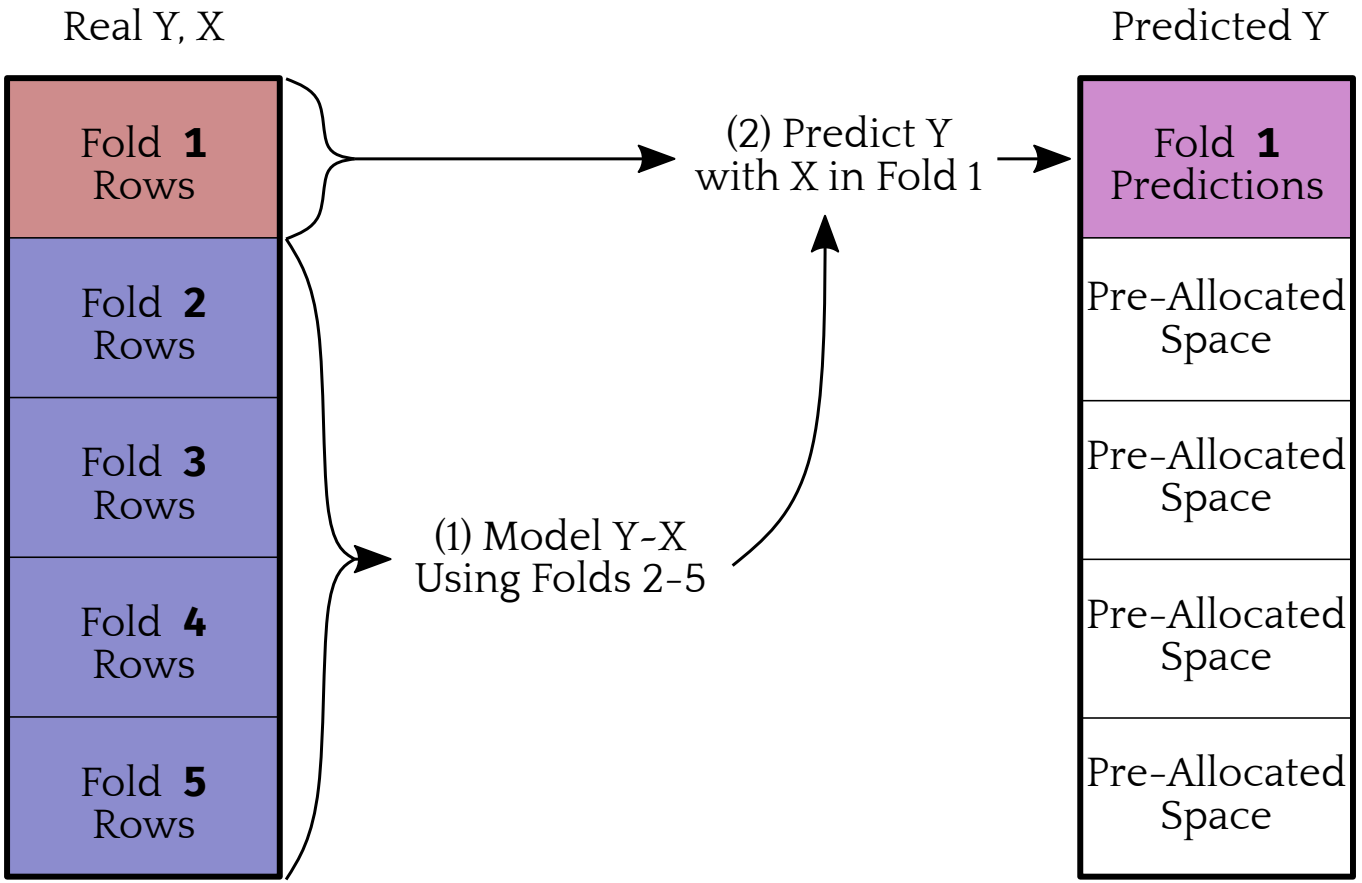
Cross Validation: What is it?

Cross validation is a widely-used method to estimate how well a model makes predictions on unseen data (data not used in fitting the model). The procedure:

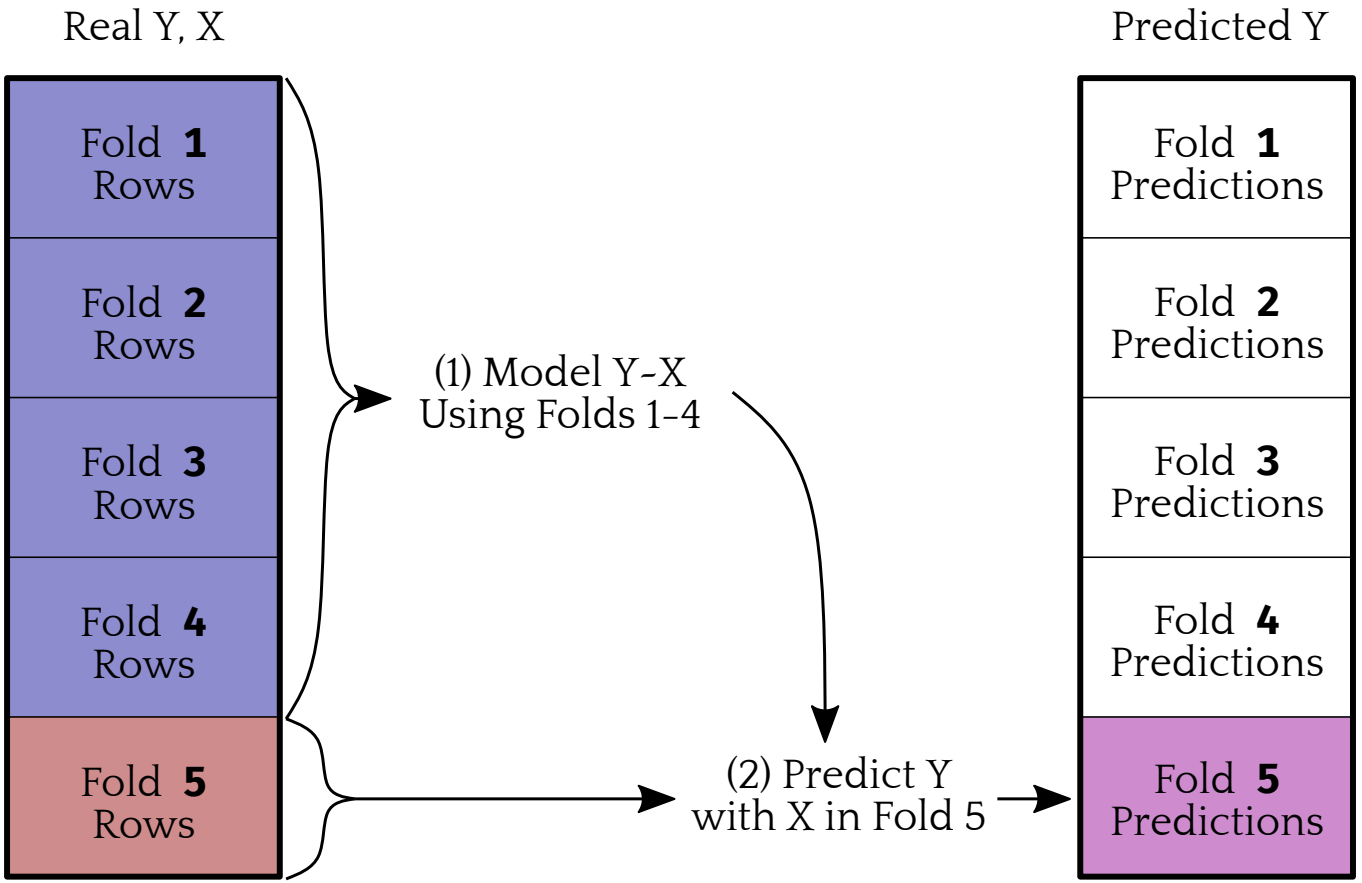
- Split your data into K **folds** (pieces)
- For each fold $i = 1, \ldots, K$:
  - Fit the model to all the data *except* that in fold i
  - Make predictions for the omitted data in fold i
- Calculate the mean squared error (or your favorite measure of accuracy comparing predictions to actuals):
  $\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (\text{actual } y_i - \text{predicted } y_i)^2$

A model that fits well will have *low mean squared error*. Models that are either too simple or too complicated will tend to make bad predictions and thus high mean squared error.

Iteration **1**


Real Y, X
Fold **1** Rows
Fold **2** Rows
Fold **3** Rows
Fold **4** Rows
Fold **5** Rows
(1) Model Y-X Using Folds 2-5
(2) Predict Y with X in Fold 1
Predicted Y
Fold **1** Predictions
Pre-Allocated Space
Pre-Allocated Space
Pre-Allocated Space
Pre-Allocated Space

Iteration **5**


Real Y, X
Fold **1** Rows
Fold **2** Rows
Fold **3** Rows
Fold **4** Rows
Fold **5** Rows
(1) Model Y~X Using Folds 1-4
(2) Predict Y with X in Fold 5
Predicted Y
Fold **1** Predictions
Fold **2** Predictions
Fold **3** Predictions
Fold **4** Predictions
Fold **5** Predictions

Pre-Allocating for CV

Let's split the data into $K = 10$ folds. We'll make a new data frame to hold the data and sampled fold numbers that we will add predictions to later.

We'll get the folds using the `sample()` function without replacement on a vector as long as our data that contains the numbers 1 through K repeated:

```
K <- 10
CV_predictions      <- sim_data
CV_predictions$fold <- sample(rep(1:K, length.out = nrow(CV_predictions)),
                              replace = FALSE)
CV_predictions[ , names(models)] <- NA
head(CV_predictions)
```

```
##       x     y fold intercept only linear quadratic cubic
## 1 -5.80  6.14    7              NA     NA        NA    NA
## 2  9.26 -0.79    4              NA     NA        NA    NA
## 3  1.40  1.18    6              NA     NA        NA    NA
## 4  5.23 -1.83    4              NA     NA        NA    NA
## 5 12.24  1.61    5              NA     NA        NA    NA
## 6  8.30 -2.78    3              NA     NA        NA    NA
```

Double-Looping for CV

Next, let's loop *over* each model (`mod`), and *within* each model loop over each fold (`k`) to fit the model and make predictions.

```
for(mod in names(models)) {
    for(k in 1:K) {
        # TRUE/FALSE vector of rows in the fold
        fold_rows <- (CV_predictions$fold == k)
        # fit model to data not in fold
        temp_mod <- lm(formula(models[mod]),
                       data = CV_predictions[!fold_rows, ])
        # predict on data in fold
        CV_predictions[fold_rows, mod] <-
           predict(temp_mod, newdata = CV_predictions[fold_rows, ])
    }
}
```

Note the models are fit *without* the fold rows, but prediction is done on *only the left-out fold rows*.

Which Did Best?

Let's write another loop to compute the mean squared error of these CV predictions.

The squared error is equal to the difference between the real values and predictions squared. The MSE is the mean of all the squared errors of each prediction.

```
CV_MSE <- setNames(numeric(length(models)), names(models))
for(mod in names(models)) {
    pred_sq_error <- (CV_predictions$y - CV_predictions[[mod]])^2
    CV_MSE[mod]   <- mean(pred_sq_error)
}
CV_MSE
```

```
## intercept only         linear      quadratic          cubic
##           2.56           2.60           1.06           1.06
```

Based on these results, which model would you choose?

Conditional Flow

`if()` then `else`

You've seen `ifelse()` before for logical checks on a whole vector. For checking whether a *single* logical statement holds and then conditionally executing a set of actions, use `if()` and `else`:

```
for(i in 1:10) {
  if(i %% 2 == 0) { # %% gets remainder after division
    print(paste0("The number ", i, " is even."))
  } else if(i %% 3 == 0) {
    print(paste0("The number ", i, " is divisible by 3."))
  } else {
    print(paste0("The number ", i, " is not divisible by 2 or 3."))
  }
}
```

Warning! `else` needs to be on same line as the closing brace `}` of previous `if()`.

`if()`, `else`: Output

The loop from the previous slide produces this output:

```
## [1] "The number 1 is not divisible by 2 or 3."
## [1] "The number 2 is even."
## [1] "The number 3 is divisible by 3."
## [1] "The number 4 is even."
## [1] "The number 5 is not divisible by 2 or 3."
## [1] "The number 6 is even."
## [1] "The number 7 is not divisible by 2 or 3."
## [1] "The number 8 is even."
## [1] "The number 9 is divisible by 3."
## [1] "The number 10 is even."
```

Handling Special Cases

Aside from the previous toy example, `if()` statements are useful when you have to handle special cases.

`if()` statements can be used to make a loop ignore or fix problematic cases.

They are also useful for producing error messages, by generating a message *if* an input value is not what is expected.

Loop Load Example 1

One common use of a loop is for loading many individual data files at once—like an entire directory of Excel files—to combine them into one data set.

(1) Let's say we have a folder filled with .csv files. We can get the file names in a folder using `list.files()`.

```
(file_list  <- list.files("./example_data/"))
```

```
##  [1] "ex_dat_1.csv"  "ex_dat_10.csv" "ex_dat_2.csv"  "ex_dat_3.csv"
##  [5] "ex_dat_4.csv"  "ex_dat_5.csv"  "ex_dat_6.csv"  "ex_dat_7.csv"
##  [9] "ex_dat_8.csv"  "ex_dat_9.csv"
```

Loop Load Example 2

(2) Let's populate an empty list of the same length as our file list:

```
data_list  <- vector("list", length(file_list))
```

(3) For nice object names, we can use `stringr` to remove the `.csv`.

```
(data_names <- stringr::str_remove(file_list, ".csv"))
```

```
##  [1] "ex_dat_1"  "ex_dat_10" "ex_dat_2"  "ex_dat_3"  "ex_dat_4"
##  [6] "ex_dat_5"  "ex_dat_6"  "ex_dat_7"  "ex_dat_8"  "ex_dat_9"
```

(4) Then assign those names to the list.

```
names(data_list) <- data_names
```

Loop Load Example 3

(5) Then, let's run a loop that reads in each data file and writes it to the appropriate element of `data_list`.

```
library(readr) # readr to load the csv files
for (i in seq_along(file_list)){
  data_list[[ data_names[i] ]] <-
    read_csv(paste0("./example_data/", file_list[i]))
}
head(data_list[[1]], 3)
```

```
## # A tibble: 3 x 3
##      id      x     z
##   <dbl>  <dbl> <dbl>
## 1    44  0.516 0.381
## 2    49  2.17  0.346
## 3    50 -0.122 0.711
```

Loop Load Example 4

```
names(data_list[[1]])
```

```
## [1] "id" "x"  "z"
```

```
names(data_list[[2]])
```

```
## [1] "id" "x"  "z"
```

In this case every data frame has the same columns, so we can combine them all into one data frame.

```
complete_data <- bind_rows(data_list)
glimpse(complete_data)
```

```
## Rows: 10,000
## Columns: 3
## $ id <dbl> 44, 49, 50, 60, 62, 76, 81, 91, 113, 114, 116, 156, 15...
## $ x  <dbl> 0.5156, 2.1673, -0.1216, 1.0551, 0.6660, -0.0213, 1.98...
## $ z  <dbl> 0.3808, 0.3460, 0.7115, 0.4344, 0.6379, 0.6739, 0.9808...
```

`while()` Loops

`while()`

A lesser-used looping structure is the `while()` loop.

Rather than iterating over a predefined vector, the loop keeps going until some condition is no longer true.

Let's see how many times we need to flip a coin to get 4 heads:

```
num_heads <- 0
num_flips <- 0
while(num_heads < 4) {
  coin_flip <- rbinom(n = 1, size = 1, prob = 0.5)
  if (coin_flip == 1) { num_heads <- num_heads + 1 }
    num_flips <- num_flips + 1
}
num_flips # follows negative binomial distribution
```

```
## [1] 6
```

Vectorization

Non-Vectorized Example

We have a vector of numbers, and we want to add 1 to each element.

```
my_vector <- rnorm(100000000) # Length 100 million random vector
```

A `for()` loop works but is super slow:

```
for_start <- proc.time() # start the clock
new_vector <- rep(NA, length(my_vector))
for(position in 1:length(my_vector)) {
    new_vector[position] <- my_vector[position] + 1
}
(for_time <- proc.time() - for_start) # time elapsed
```

```
##    user  system elapsed
##   93.78    0.12   94.34
```

Vectorization Wins

Recognize that we can instead use R's vector addition (with recycling):

```
vec_start  <- proc.time()
new_vector <- my_vector + 1
(vec_time  <- proc.time() - vec_start)
```

```
##    user  system elapsed
##    0.13    0.03    0.15
```

```
for_time / vec_time
```

```
##    user  system elapsed
##     721       4     629
```

The vectorized method was 721 times as fast! Vector/matrix arithmetic is implemented using fast, optimized functions that a `for()` loop can't compete with.

Vectorization Examples

- `rowSums()`, `colSums()`, `rowMeans()`, `colMeans()` give sums or averages over rows or columns of matrices/data frames

```
(a_matrix <- matrix(1:12, nrow = 3, ncol = 4))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
rowSums(a_matrix)
```

```
## [1] 22 26 30
```

More Vectorization Examples

- `cumsum()`, `cumprod()`, `cummin()`, `cummax()` give back a vector with cumulative quantities (e.g. running totals)

```
cumsum(1:7)
```

```
## [1]  1  3  6 10 15 21 28
```

- `pmax()` and `pmin()` take a matrix or set of vectors, output the min or max for each **p**osition (after recycling):

```
pmax(c(0, 2, 4), c(1, 1, 1), c(2, 2, 2))
```

```
## [1] 2 2 4
```

Vectorized File Loading

`vroom` is a package with vectorized loading of delimited files like `.csv` files.

```
complete_data_vroom <- vroom::vroom(list.files("./example_data/", full.names=T))
```

```
## Rows: 10,000
## Columns: 3
## Delimiter: ","
## dbl [3]: id, x, z
##
## Use `spec()` to retrieve the guessed column specification
## Pass a specification to the `col_types` argument to quiet this message
```

```
glimpse(complete_data_vroom)
```

```
## Rows: 10,000
## Columns: 3
## $ id <dbl> 44, 49, 50, 60, 62, 76, 81, 91, 113, 114, 116, 156, 15...
## $ x  <dbl> 0.5156, 2.1673, -0.1216, 1.0551, 0.6660, -0.0213, 1.98...
## $ z  <dbl> 0.3808, 0.3460, 0.7115, 0.4344, 0.6379, 0.6739, 0.9808...
```

This replaced our entire data loading loop with one line!

Homework

Reminder: HW 5 Part II assigned last week is due midnight Tuesday.