S3 Objects - A Study in Linear Regression

James Balamuta

Department of Informatics, Statistics
University of Illinois at Urbana-Champaign

July 11, 2017

On the Agenda

1. Administrative
   - HW3
2. S3 Objects
   - What is Object-oriented programming (OOP)?
   - How are S3 Methods implemented in R?
3. The `lm` function
   - Constructing `lm()`
   - Constructing Inference via `summary.lm()`

Administrative Items

- HW3 Posted Tonight (sorry for the delay!)
  - Due on: **Tuesday, July 18th at 2:00 PM CST**.

On the Agenda

Programming Up Until Now...

- In the past lectures, the focus has been on the creation of modular code via **functions**.
- The goals behind creating **functions** were to:
  1. create reusable chunks of code
  2. decrease the probability of error in code
  3. make shareable code
- Now, we're going to take it one step further with **Object-oriented programming (OOP)**.

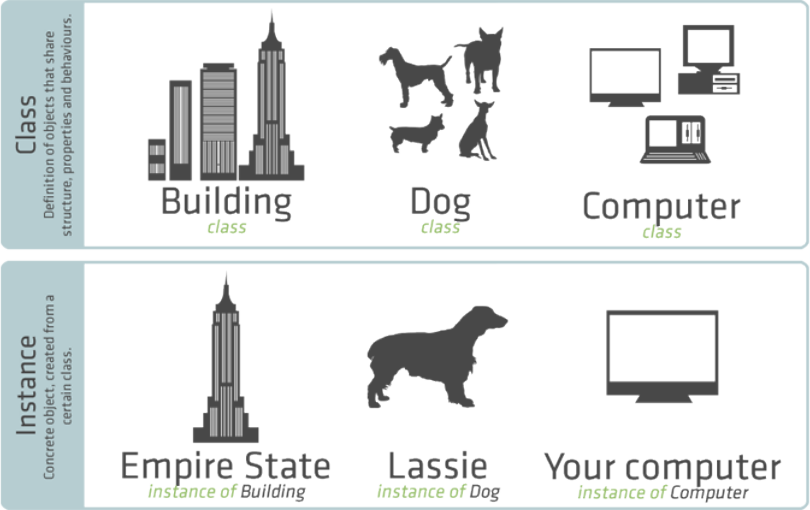
Object-oriented programming (OOP)

Definition: **Object-oriented programming (OOP)** is a programming paradigm where real world ideas can be described as a collection of items that are able to interact together.

Definitions of OOP Concepts

- Definition: **Classes** are definitions of what an **object** *is*.
  - Example: **Student** has properties of **Name**, **NetID**, **Grades**, **Address**, ...
- Definition: **Objects** are instances of a **class**. (*noun*)
  - Example: **Brian** and **Grant** are instances of a **Student**
- Definition: **Methods** are functions that performs specific calculations on objects of a specific class. (*verb*)
  - Example: **in_class()** and **get_grade()**

Think of it as...



Core Tenets of OOP

- **Encapsulation:** Enables the combination of data and functions into classes
- **Polymorphism:** Functions are able to act differently across classes
- **Inheritence:** Extend a parent class by creating a child classes without copying!

OOP Concept Check

- How would an **instructor** class be defined?
- How might we abstract both so that they share a common class?

Why use OOP?

1. **Increased Modularity:**
    - Code for classes can be implemented and maintained separately from other classes.
2. **Hide Subroutines:**
    - Avoid having multiple method functions known.
3. **Code Reuse and Recycling Across Packages:**
    - Easily extend classes defined in other R packages or within the base *R* system. (e.g. Allen wrenches)
4. **Features and Debugging**:
    - Problematic class? Remove or easily revert class to an earlier version without worrying about headache across the entire system.

OOP in R

"To understand computations in R, two slogans are helpful:

- ***Everything that exists is an object.***
- *Everything that happens is a function call."*

—John Chambers

Question: *How* is everything in *R* an object?

OOP in R - Answer

- In order for an **object** to exist, it must have a **class**.
- Every object in *R* returns a class when `class(x)` is called.
  - Even *functions* and *environments* have classes!

```
class(3)             # Number
```

```
## [1] "numeric"
```

```
class(sum)           # Function
```

```
## [1] "function"
```

```
class(.GlobalEnv)    # Global Environment
```

```
## [1] "environment"
```

R's OOP Systems

- There are three OOP systems in *R* that differ in terms of how classes and methods are defined:
  - **S3:** Very casual/informal OO system that is used throughout *R*
  - **S4:** More formal and rigorous with class definitions.
  - **Reference classes (RC):** Very new and shiny OOP system that mimicks traditional Java and C++ message-passing OO.
- Today, we'll focus on just working within the **S3** System.
- Depending on interest, we may later explore **S4** and **RC**.

Detecting Object Type

- Before we begin, it's important to be able to detect the type of OOP systems begin used within the method.
- To reliable detect the OOP system, please use `ftype()` in pryr R Package.

```
pryr::ftype(print)
```

```
## [1] "s3"      "generic"
```

On the Agenda

R's S3 System

- Definition: A **generic function** is used to determine the class of its arguments and select the appropriate method.
  - Examples: `summary()`, `print()`, and `plot()`
  - The `lm` class has: `summary.lm()`, `print.lm()`, and `plot.lm()`
- Generic functions have a method naming convention of: `generic.class()`
- If a class has not been defined for use in a generics, it **will** fail.
  - To avoid the failure define `generic.default()` (e.g. `summary.default`)

Generic Function

- S3 generic detectable by looking at a function's source for `UseMethod()`

```
summary
```

```
## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x7fb9039a0580>
## <environment: namespace:base>
```

Viewing S3 Methods Associated with Generic

```
# All classes with a summary.*() function
methods(summary)
```

```
## [1] "summary.aov"     "summary.aovlist"
```

```
# Methods using a particular class
methods(class='matrix')
```

```
## [1] "anyDuplicated.matrix"      "as.data.frame.matrix"
## [3] "as.raster.matrix"          "boxplot.matrix"
## [5] "coerce,ANY,matrix-method" "determinant.matrix"
```

Note: Output has been suppressed, there are considerably more usages. Try running the commands yourself!

Constructing an S3 Object

Part of S3's ability to be informal is the ease of construction.

There are two different flavors of construction:

- All in one
- The two-step

These constructions are **informal** as there is no forced upfront definition of a *class*.

Constructing an S3 Object - One Step

```r
# ----- One Step S3 Construct

# Create object `shawn` and assign class `student`
shawn = structure(list(), class = "student")

class(shawn)                    # Check class
```

```
## [1] "student"
```

```r
str(shawn)                      # Structure
```

```
##  list()
##  - attr(*, "class")= chr "student"
```

Constructing an S3 Object - Two Step by Class

```r
# ----- Two Step S3 Construct

shawn = list()                # Create object shawn
                              # as a list class

class(shawn) = "student"      # then set class to student

class(shawn)                  # Check obj type
```

```
## [1] "student"
```

```r
str(shawn)                    # Structure
```

```
##  list()
##  - attr(*, "class")= chr "student"
```

Constructing an S3 Object - Two Step by Attribute

```r
# ----- Two Step S3 Construct with Attributes

shawn = list()                        # Create object shawn
                                      # as a list class

attr(shawn,"class") = "student"  # Set class to student

class(shawn)                          # Check obj type
```

```
## [1] "student"
```

```r
str(shawn)                            # Structure
```

```
##  list()
##  - attr(*, "class")= chr "student"
```

Checking for Object Status

To determine whether an object is of a specific class use `inherits(x, "class")`

```
inherits(shawn, "student")
```

```
## [1] TRUE
```

```
inherits(shawn, "list")
```

```
## [1] FALSE
```

Note: The list inheritance check failed as we removed that `class` definition.

Creating Classes

- Prior to defining a `generic` function, we need to establish the hierarchy of classes and their properties.
- Consider three classes: `human`, `instructor`, and `student`.
- The natural hierarchy would be:
  - $instructor \subseteq human \subseteq list$
  - $student \subseteq human \subseteq list$
- **Note:** `instructor` and `student` are different child classes of `human`.

Creating Class Definitions

- Each `generic` function should be able to rely upon the properties being of a specified class.
- To ensure each `generic` has that ability, we opt to define the following properties per class.
  - `human` has `fname` (First Name)
  - `instructor` has `fname` and `course` (Teaching)
  - `student` has `fname`, `course`, and `grade` (In course)

Creating a New Generic

To begin, we aim to create `generic` function for the parent class: `human`

```
# Create a role identifier


# method `role` for class `human`
role.human = function(x){
  cat("Hi there human", x$fname, "!\n")
}
```

Note:

- Here we have only defined a method to operate on the `human` class.
- If we wanted to use a specific information (e.g. `student` and `instructor`), we need to define the method to work with .

Creating a New Generic

```r
# method `role` for class `instructor`
role.instructor = function(x){
  cat("Greetings and Salutations", x$fname, ",\n",
      "You are an instructor for", x$course, "\n")
}

# method `role` for class `student`
role.student = function(x){
    cat("Hey", x$fname, "!\n",
      "You are in", x$course, "\n",
      "Your grade is:", x$grade, "\n")
}
```

Notes:

- `student` and `instructor` are the **classes**
- `role` is the method.
- There are no `objects`! (No instances.)

UseMethod() Properties

To create a generic function, we only need to do:

```
# Create a default case
role = function(x, ...){
  UseMethod("role")
}
```

A few notes:

- The `generic` function will call the first class it finds with an implementation based on searching from left to right
- If no class is found, then the dispatch uses the `generic.default()` function *if it has been defined!*
- The `...` are ellipses and they enable additional parameters to be passed through.

Example Call of Generic - One Class

- An initial class `instructor` in S3 would look like so:

```
# Create object `james` and assign class `instructor`
james = structure(list(fname = 'James',
                       course = "STAT385"),
                  class = "instructor")

role(james)
```

```
## Greetings and Salutations James ,
##  You are an instructor for STAT385
```

Example Call of Generic - Two Classes

- Here the `david` object has two class types.
- Only the first class (from left to right) will be called.

```
# Create object `david` and assign classes
# `instructor` and `student`
david = structure(list(fname = 'David',
                       course = "STAT385",
                       grade = "A"),
                  class = c('student','instructor'))

role(david)
```

```
## Hey David !
##  You are inL STAT385
##  Your grade is: A
```

Example Call of Generic - Unknown Class

- If we do **not**:
  1. define a `generic.*()` for a `class`
  2. define a `generic.default()`,
- then the `generic` dispatch will **error**

```
toad = structure(list(fname = 'McToady',
                      course = "STAT385",
                       grade = "A"),
                 class = 'humbug')

role(toad)

### Error in UseMethod("role") :
###  no applicable method for 'role' applied
### to an object of class "list"
```

Protecting Generics with `generic.default()`

- Always protect your generic with a `generic.default()`!

```
role.default = function(x){     # Default case
  cat("I have no clue what your role is. Who are you?")
}

# Try again
role(toad)
```

```
## I have no clue what your role is. Who are you?
```

Use Inheritance!

- When assigning classes to objects, use the *inheritance* tenet!
- Write classes in decreasing order starting with the most-specific first and then classes with less properties
  - *instructor* $\subseteq$ *human* $\subseteq$ *list*

```
james = structure(list(fname = "James",
                       course = "STAT385"),
                  class = c("instructor", # Specific
                            "human",      # Less specific
                            "list"))      # Default Object

str(james)                                # Structure
```

```
## List of 2
##  $ fname : chr "James"
##  $ course: chr "STAT385"
```

Practical Note

- Avoid calling the methods function directly.
  - Use a `generic` function to dispatch the methods to objects
  - e.g. use `summary()` instead of `summary.yourobj()`.

```
# Bad
role.instructor(james) # Not always an instructor!

# Good
role(james) # Adapts to future change!
```

Note: Objects (instances) may change in terms of classes assigned!

Summary on S3

- Very informal and easy to work with.
- Be on your guard as it relates to class definitions.
- Define a `generic.default()` method for extra protection.

On the Agenda

Understanding the Algorithm

Before we can implement an algorithm, we must understand the following:

- *What* logic is being used?
- *How* does the logic apply in a procedural form?
- *Why* is this logic present?

Thus, let's take a bit of a closer look at Multiple Linear Regression (MLR) *before* we start to implement it.

Multiple Linear Regression (MLR)

Solutions:

$$\begin{aligned}
\hat{\beta}_{p\times 1} &= \left(X^T X\right)^{-1}_{p\times p} X^T_{p\times n} y_{n\times 1} \\
E\left(\hat{\beta}\right) &= \beta_{p\times 1} \\
Cov\left(\hat{\beta}\right) &= \sigma^2 \left(X^T X\right)^{-1}_{p\times p}
\end{aligned}$$

Freebies:

$$\begin{aligned}
df &= n - p \\
\sigma^2 &= \frac{\mathbf{e}^T \mathbf{e}}{df} = \frac{RSS}{n-p}
\end{aligned}$$

Derivations of MLR...

For those interested in the derivations of MLR, please see:

http://thecoatlessprofessor.com/

Writing a my_lm() function - Part 1

To begin, we start with the basic definition for a generic method.

```
my_lm = function(x, ...){
  UseMethod("my_lm")
}
```

Note: Under this approach, we can extend my_lm to work with formula (e.g y ~ x)

Writing a my_lm() function - Part 2

Now, let's implement the my_lm default method.

```
my_lm.default = function(x, y, ...){

  # Obtain the QR Decomposition of X
  # Not a good approach for rank-deficient matrices
  qr_x = qr(x)

  # Compute the Beta_hat = (X^T X)^(-1) X^T y estimator
  beta_hat =  solve.qr(qr_x, y)

  # Compute the Degrees of Freedom
  df = nrow(x) - ncol(x)    # n - p
```

Writing a my_lm() function - Part 3

```
# Compute the Standard Deviation of the Residuals
sigma2 = sum((y - x %*% beta_hat) ^ 2) / df

# Compute the Covariance Matrix
# Cov(Beta_hat) = sigma^2 * (X^T X)^(-1)
cov_mat = sigma2 * chol2inv(qr_x$qr)

# Make name symmetric in covariance matrix
rownames(cov_mat) = colnames(x)
colnames(cov_mat) = colnames(x)

# Return a list
return(structure(list(coefs = beta_hat,
                      cov_mat = cov_mat,
                      sigma = sqrt(sigma2), df = df),
                 class = "my_lm"))
```

Writing a print.my_lm() function - Part 4

- We can hook the `my_lm` class directly into generic `print` function

```
print.my_lm = function(x, ...){
  cat("\nCoefficients:\n")
  print(x$coefs)
}
```

Notes: - Very basic print extension. - Here we end up calling the default matrix print method using `x$coefs`.

Comparing print() Output (print.my_lm() vs. print.lm())

```r
# Our Implementation of lm
my_lm(x = cbind(1, mtcars$disp), y = mtcars$mpg)
```

```
##
## Coefficients:
## [1] 29.59985476 -0.04121512
```

Comparing print() Output (print.my_lm() vs. print.lm())

```r
# Base R implementation
lm(mpg~disp, data = mtcars)
```

```
##
## Call:
## lm(formula = mpg ~ disp, data = mtcars)
##
## Coefficients:
## (Intercept)         disp
##    29.59985     -0.04122
```

Writing a summary.my_lm() function - Part 5

```
# Note that summary(object, ...) instead of summary(x, ...)!
summary.my_lm = function(object, ...){

  estimate = object$coefs                    # Beta Hat
  sterr = sqrt(diag(object$cov_mat)) # STD Error
  t_test = estimate / sterr                  # t-Test value
  pval = 2*pt(-abs(t_test), df=object$df) # p-value

  # Make output matrix
  mat = cbind("Estimate"= estimate, "Std. Err" = sterr,
              "t value" = t_test, "Pr(>|t|)" = pval)

  rownames(mat) = rownames(object$cov_mat) # Naming

  return(structure(list(mat = mat),
                   class = "summary.my_lm"))
```

Writing a print.summary.my_lm() function - Part 5

- We can control how the summary generic function should look like on print via `print.summary.my_lm`.
- Here we make use of the `printCoefmat()` functionality.

```
# Note that print(x,...)!!
print.summary.my_lm = function(x, ...) {
  printCoefmat(x$mat,
               P.value = TRUE,
               has.Pvalue = TRUE)
}
```

Comparing summary() Output: summary.my_lm()

```r
# Our Implementation of lm
summary(my_lm(x = cbind("(Intercept)" = 1,
                        "disp" = mtcars$disp),
              y = mtcars$mpg))
```

```
##                Estimate   Std. Err t value  Pr(>|t|)
## (Intercept) 29.5998548  1.2297195 24.0704 < 2.2e-16 ***
## disp        -0.0412151  0.0047118 -8.7472  9.38e-10 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Comparing summary() Output: summary.my_lm()

```r
# Base R implementation
summary(lm(mpg~disp, data = mtcars))
```

```
##
## Call:
## lm(formula = mpg ~ disp, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.8922 -2.2022 -0.9631  1.6272  7.2305
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 29.599855   1.229720  24.070  < 2e-16 ***
## disp        -0.041215   0.004712  -8.747 9.38e-10 ***
## ---
```