POLI803: Maximum Likelihood Estimation

Led by Dr. Bird

Ph.D. Student, Sanghoon Park (Univ. of South Carolina)

8/20/2019

R 기본 리뷰

데이터의 구조

가장 기본적으로(그리고 중요하게) 알아두어야 할 데이터 구조는 바로 원자 벡터(atomic vectors), 리스트(lists), 그리고 데이터프레임(dataframes)과 티블(tibbles)이다.

```
# 이제부터는 거의 모든 데이터를 관리하는 데 있어 tidyverse 패키지를 사용한다.
library(tidyverse)

# 원자 벡터는 c() 라는 함수로 만들 수 있다.
my_vector <- c(1, 2, 3) # 벡터는 반드시 동일한 유형의 요소들만을 가져야 한다.
typeof(my_vector)
```

```
## [1] "double"
```

```
# 연산자들은 벡터 전체에 걸쳐서 적용된다.
my_vector * 2
```

```
## [1] 2 4 6
```

```
# 벡터는 1차원의 자료만을 가진다. 즉, 벡터 안에 다른 벡터를 중첩하여 만들 수 없다(cannot nest).
new_vector <- c(my_vector, c(4, 5), c(6, 7)) # 자, 어디 중첩이 되나 보자.
# 여기서의 벡터 결과는 중첩되지 않고 단지 1차원의 단순 벡터일 뿐이다.
# (리스트는 이 결과와는 달리 자료 내에 중첩이 가능하다.)
print(new_vector)
```

```
## [1] 1 2 3 4 5 6 7
```

```
# 리스트는 서로 다른 유형의 데이터, R에서 지원하는 모든 유형의 데이터를 담을 수 있다.
my_list <- list(1, 3L, 'hello, world', TRUE)
print(map_chr(my_list, function(x) typeof(x))) # 이건 잠시 후에 좀 더 살펴보자.
```

```
## [1] "double"    "integer"   "character" "logical"
```

```
# 데이터프레임은 데이터 분석에서 앞으로 종종 사용하게 될 리스트에 속한 자료 유형이라고 할 수 있다.
df <- data.frame(
  age = c(sample(20:40, 15, replace = T)),
  collgrad = sample(c('No', 'Yes'), 15, replace = T)
)
print(df)
```

```
##    age collgrad
## 1   38      Yes
## 2   27       No
## 3   38      Yes
## 4   26      Yes
## 5   31      Yes
```

```
## 6   34       No
## 7   31       No
## 8   38      Yes
## 9   37       No
## 10  27       No
## 11  20      Yes
## 12  32       No
## 13  22      Yes
## 14  37       No
## 15  36      Yes
```

```r
# 티블은 데이터프레임이 좀 더 발전한 형태의 자료 유형이다.
# 데이터프레임은 자료를 변환할 때 문자형(strings)을 자동으로 다 분류형(factors)으로 바꿔 버린다.
# 그에 반해 티블은 자료를 원형을 유지하여 변환한다,
# (데이터프레임에서는 위에서 만든 'collgrad' 변수의 값들이 문자열로 입력되었음에도
# 분류형이 된 것을 확인할 수 있다.)
class(df$collgrad)
```

```
## [1] "factor"
```

```r
# tidyverse 패키지의 속한 티블은 데이터프레임 함수와는 다르다.
# 우선 티블의 장점은 더 유저 친화적이라는 것이다. 예를 들어, 티블은 R 콘솔창에서 한 눈에
# 확인할 수 있을 정도로 데이터의 구조를 출력해주고, 행 이름을 생성해주지 않는다.
# 그리고 변수들의 자료유형이 어떤 것인지를 표시한다.
# 데이터프레임을 티블로 강제 변환해야 할 경우가 있다.
# 이때는 as_tibble() 를 사용하면 된다.

df <- tibble(
  age = c(sample(20:40, 15, replace = T)),
  collgrad = sample(c('No', 'Yes'), 15, replace = T)
)
print(df)
```

```
## # A tibble: 15 x 2
##      age collgrad
##    <int> <chr>
##  1    40 Yes
##  2    30 No
##  3    20 No
##  4    26 Yes
##  5    23 No
##  6    27 Yes
##  7    38 No
##  8    22 Yes
##  9    32 Yes
## 10    40 No
## 11    30 Yes
## 12    34 No
## 13    22 Yes
## 14    25 No
## 15    24 Yes
```

함수

함수는 아마도 R 프로그래밍에서 가장 중요하다고 해도 과언이 아닐 것이다.

```r
# 함수를 만드는 방법은 다음과 같다.
greeting <- function(name) { #내가 만들 함수이름 (내가 집어넣을 객체)
  result <- paste0('Hello ', name, '!') # '결과'라는 객체에 우측의 함수를 수행하고
  return(result) # 그 결과를 반환하라는 명령어다.
}

# 자, 이제 위의 'greeting'이라는 함수는 {} 안의 작업을 수행하는 하나의 함수다.
# 이제 'Robert'라는 객체에 그 함수를 적용해보자.
greeting('Robert')
```

```
## [1] "Hello Robert!"
```

루프와 함수형 프로그래밍 (Loops & functionals)

루프는 R에서 일정한 횟수의 과정을 반복하도록 하게하는 프로그래밍을 말한다.

```r
# 몇 개의 이름을 가지고 있는 벡터에 새롭게 만든 인삿말을 표현하도록 만든 함수와 결합해보자.
# 벡터의 각각의 이름들을 해당 함수에 집어넣는 과정을 자동으로 반복되게 할 것이다.
my_names <- c('Robert', 'James', 'Liz', 'Jennifer', 'Oscar')
results1 <- c(NA, NA, NA, NA, NA)
for (i in 1:length(my_names)) {
  results1[i] <- greeting(my_names[i])
}
print(results1)
```

```
## [1] "Hello Robert!"   "Hello James!"    "Hello Liz!"      "Hello Jennifer!"
## [5] "Hello Oscar!"
```

```r
# 위에서 수행한 루프를 좀 더 간략하게 만든 것이다.
# 이것도 작동은 하지만 벡터의 각 요소의 순서(index)를 이용하지 않고,
# 객체를 바로 집어넣어 작동시킨 것이다.
for (i in my_names) {
  greeting(i)
}
```

여기서 한 가지 알아두어야 할 사실은 우리가 만든 함수, 객체가 작동하는 범주(scope)가 중요하다는 것이다. 변수가 실제로 유효하게 존재하는 범주는 어디인가? 예를 들어, 앞의 `greeting` 함수에서 `result` 변수는 오직 그 함수를 정의하는 중괄호({}) 내부에서만 유효한 변수이다. 만약 우리가 `print(result)`를 이용해 `result` 함수를 출력하라고 해도, 우리는 아무것도 얻지 못한다. 왜냐하면 그 함수엣만 소모되는 변수이므로 우리의 작업환경(environment)에는 별도로 저장되어 있지 않기 때문이다.

```r
# 위의 루프문은 함수형을 이용하여 더 간략하고 파워풀하게 만들 수도 있다.
# 함수형은 tidyverse 패키지의 한 부류인 purrr package에서 지원된다.
greeting_list <- map(my_names, greeting) # 리스트로 저장
print(greeting_list)
```

```
## [[1]]
## [1] "Hello Robert!"
##
## [[2]]
## [1] "Hello James!"
##
## [[3]]
## [1] "Hello Liz!"
##
## [[4]]
```

```
## [1] "Hello Jennifer!"
##
## [[5]]
## [1] "Hello Oscar!"
```

```
greeting_vector <- map_chr(my_names, greeting) # 벡터로 저장
print(greeting_vector)
```

```
## [1] "Hello Robert!"   "Hello James!"    "Hello Liz!"      "Hello Jennifer!"
## [5] "Hello Oscar!"
```

```
# 둘 다, 우리가 원하는 루프의 결과와 동일한 값을 가진다.
# 단지 자료 유형(나열의 방식)에서 차이가 있다.
```

함수형은 R 인터페이스에는 존재하지 않는, 사용자들의 편의 상 추가된 개념이라고 생각하면 된다.

```
# 파이프는 앞으로 R 프로그래밍을 하는 데 있어 필수적인 기능이다.
# tidyr 패키지(마찬가지로 tidyverse 패키지의 일부)로 불러올 수 있지만,
# 원래는 magrittr 패키지에 있는 함수이다.
library(magrittr)

# R은 함수를 가장 안쪽의 괄호에서부터 바깥쪽으로 순차적으로 읽어나온다.
random_vector <- sample(1:100, 25, replace = T)
result <- sum(as.integer(is.na(unique(random_vector))))
# 두 번째 result 객체를 구성하는 함수는 한 줄로 나열되어 있어서 읽기 좀 어렵고,
# 왜 이렇게 코드를 짰는지 이해하기 어렵다.
print(result)
```

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

```
# 파이프 (%>%)를 이용하면 코드를 더 보기 좋게 짤 수 있다.
# 파이프에서 이전 데이터는 (.)으로 표현된다.
result <- unique(random_vector) %>% ## random_vector의 중복값 없이 고유값만 남겨라
  is.na(.) %>%  ## 앞의 중복값을 없앤 random_vector 자료에 결측치가 있는지 확인해라
                ## 이 경우 is.na는 논리형 연산자이기 때문에 결측치는 TRUE, 아니면 FALSE
  as.integer(.) %>%  ## 앞의 random_vector를 정수형으로 변환해라.
                     ## R에서 TRUE는 1, FALSE는 0
  sum(.)             ## 변환한 결과의 총합을 구하라. 결측치가 없으면 FALSE들만 있을 것
print(result)        ## 즉, result의 결과값은 데이터의 결측치(NA)의 수를 말하는 것이다.
```

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

```
# If the first argument is the data, then you don't even need the .
result <- unique(random_vector) %>%
  is.na() %>%
  as.integer() %>%
  sum()
print(result)
```

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

tidyverse에 대해 더 자세한 정보를 원하면, 다음의 무료 ebook R for Data Science을 참고해보기를 바란다. 국문의 경우 R을 활용한 데이터과학의 번역된 웹페이지가 존재한다.

마찬가지로 프로그래밍 언어로서 R에 대해 더 자세히 알고자 한다면 다음의 ebook을 무료로 확인할 수 있다.: Advanced R 2nd Edition.

최소자승법에 대한 간단한 리뷰

종속변수가 연속형일 때 우리는 최소자승법을 통해 모형을 추정할 수 있다. 다음과 같은 모형을 가지고 있다고 가정해보자:

$$y_i = \alpha + \beta_1 X_1 + ... + \beta_n X_n + \epsilon_i$$

.

우리는 위의 모형을 다음과 같이 재구성할 수 있다:

$$\epsilon_i^2 = (y_i - \alpha - \beta_1 X_1 - ... - \beta_n X_n)^2$$

요컨대 우리는 제곱된 오차(squared error)를 최소화하기 위하여 위의 재구성된 모형에 도함수를 취하여 0이 되게끔 하는 해를 구할 수 있다.

그러나 만약 종속변수가 연속형이 아닐 경우에는 최소자승법—제곱된 오차항의 값을 최소화하는 해를 구하는 방법을 사용할 수가 없기 때문에 우리는 다른 추정 방법을 사용할 수밖에 없다. 여기에서 최대가능도(Maximum Likelihood Estimation, MLE)의 개념이 등장한다.

MLE에 대한 간략한 개관

OLS를 가지고 우리는 자료의 오차를 최소화하는 모수값(α and β)을 찾고자 한다.

MLE는 주어진 데이터를 가지고 조금 다른 접근법을 취한다. 어떤 모수가 주어진 데이터를 갖게끔 할지, 그 가능도를 최대화하는 모수가 무엇인지를 구하는 방법이 바로 MLE이다. 종속변수가 연속형이 아닐때, 특히 0과 1로 나타날 때 우리를 항상 괴롭히는 베르누이 시행 분포의 대표적인 사례, 동전 던지기로 MLE를 살펴보자.

동전 던지기 예시

동전 던지기를 한다고 할 때, 뒷면이 나오면 0, 앞면이 나오면 1이라고 하자. 그러나 동시에 우리가 그 동전을 던져서 뒷면과 앞면이 나올 확률이 공정할지 아닐지(즉, 앞면:뒷면의 확률이 5:5일지 아닐지) 모른다고 할 때, 우리는 앞면이 나올 기대값(expected values)을 알고자 노력할 것이다. 아무것도 모른다고 할 때, 우리는 일단 그 기대값을 0.5라고 생각할 것이다. 어찌보면 당연한 것이다. 앞면과 뒷면이라고 선택지가 둘 밖에 없는데 어설프게 앞면 확률 0.3이라고 하는 것보다는 상식적으로 반반, 아무것도 모를 때는 0.5라고 생각하는 것이 타당할 것이다.

그러나 만약 우리가 동전을 1,000번 던졌고, 900번 앞면이 나왔다고 한다면, 과연 우리는 앞면이 나올 확률의 기대값을 여전히 0.5라고 할 수 있을까? MLE는 주어진 자료에서 기대값에 대한 최선의 추측—1/2가 아니라 9/10이라는 값을 제시한다. 우리가 주어진 자료를 가지고 있을 때, 무엇이 우리가 추정하고자 하는 모수에 대한 최선의 추정인가? 이 모수를 MLE에서는 θ라고 한다.

조금 더 수학적인 예시

동전을 딱 세 번 던지며, 이때 동전의 앞면이 나올 확률, 모수를 0.5라고 가정하기보다는 모른다고 하자. 그리고 우리는 그 알 수 없는 모수(unknown parameter)를 추정하고자 하며, 그 모수를 θ라고 부른다고 하자.

위의 예제에 따라서 동전을 던져서 앞면, 앞면, 그리고 뒷면이 나왔다고 할 때, 우리는 정확하게 그 결과를 얻을 수 있는 가능도(likelihood)에 대해 알고자할 것이다. 우리가 알고싶은 가능도는 다음과 같이 표현할 수 있다:

$$L = \theta^2(1-\theta)$$

.

만약 우리가 양변에 로그를 취하면 다음과 같이 표현할 수 있다:

$$log(L) = 2log(\theta) + log(1-\theta)$$

일반적으로 $\frac{\partial}{\partial x} log_a(X)$에 도함수를 취한 결과는 $\frac{1}{Xln(a)}$와 같기 때문에 우리는 공식의 도함수를 취해

$$2\frac{1}{\theta ln(e)} - 1\frac{1}{(1-\theta)ln(e)}$$

로 나타낼 수 있다.

이 공식을 축약하여 0과 같게 되는 해를 구하게 된다면,

$$\frac{\partial logL}{\partial \theta} = \frac{2}{\theta} - \frac{1}{(1-\theta)} = 0$$

라고 다시 재구성하여

$$\hat{\theta} = \frac{2}{3}$$

라는 결과를 얻게 된다.