Home Work 2

Isaiah Thompson Ocansey Statistical Programming
University of Texas at El Paso (UTEP)

2022-07-21

```
rm(list=ls())
```

Question (2) Download diamond-sizes.Rmd from https://github.com/hadley/r4ds/tree/master/rmarkdown. Add a section that describes the largest 20 diamonds, including a table that displays their most important attributes.

```
library(ggplot2)
data("diamonds")
head(diamonds);dim(diamonds)
```

```
## # A tibble: 6 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
## 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
## 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
## 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
## 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

```
## [1] 53940    10
```

The data has 10 variables and 53940 observations. we would now add a section that describes the largest 20 diamonds, including a table that displays their most important attributes as shown below

```
library(ggplot2)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
diamonds %>%
  arrange(desc(carat)) %>%
  slice(1:20) %>%
  select(carat, cut, color, clarity) %>%
  knitr::kable(
    caption = "The largest 20 diamonds in the `diamonds` dataset."
  )
```

Table 1: The largest 20 diamonds in the `diamonds` dataset.

carat	cut	color	clarity
5.01	Fair	J	I1
4.50	Fair	J	I1
4.13	Fair	H	I1
4.01	Premium	I	I1
4.01	Premium	J	I1
4.00	Very Good	I	I1
3.67	Premium	I	I1
3.65	Fair	H	I1
3.51	Premium	J	VS2
3.50	Ideal	H	I1
3.40	Fair	D	I1
3.24	Premium	H	I1
3.22	Ideal	I	I1
3.11	Fair	J	I1
3.05	Premium	E	I1
3.04	Very Good	I	SI2
3.04	Premium	I	SI2
3.02	Fair	I	I1
3.01	Premium	I	I1
3.01	Premium	F	I1

Most of the twenty largest data sets are in the lowest clarity category ("I1"), with one being in the second best category ("VVS2").The top twenty diamonds have colors ranging from the worst,"J"*, to best, "D",categories, though most are in the lower categories "J" and "I". The top twenty diamonds are more evenly distributed among the cut categories, from "Fair" to "Ideal", although the worst category (Fair)* is *the most common.*

Question (3) Modify diamonds-sizes.Rmd to use comma() to produce nicely formatted output. Also include the percentage of diamonds that are larger than 2.5 carats.

```
library(ggplot2)
library(dplyr)

comma <- function(x) format(x, digits = 2, big.mark = ",")

diamonds %>%
  summarise(`Percentage of diamonds larger than 2.5 carats` = ((sum(carat > 2.5) / n())*100) %>%
              comma()) %>%
  knitr::kable()
```

Percentage of diamonds larger than 2.5 carats
0.23

We have used the comma() to produce a nicely formatted output as shown above and the percentage of diamonds larger than 2.5 carats is 0.23%

Question 1 (section 25.2.6) With the basics of C++ in hand, it's now a great time to practice by reading and writing some simple C++ functions. For each of the following functions, read the code and figure out what the corresponding base R function is. You might not understand every part of the code yet, but you should be able to figure out the basics of what the function does.

Answer: The code in question 1(section 25.2.6) corresponds to the following base R functions:

f1: mean() f2: cumsum() f3: any() f4: Position() f5: pmin()

Question 2 is bonus according to Dr. Chatla.

Question 1(Section 10.2.6) The definition of force() is simple:

#force function (x) x

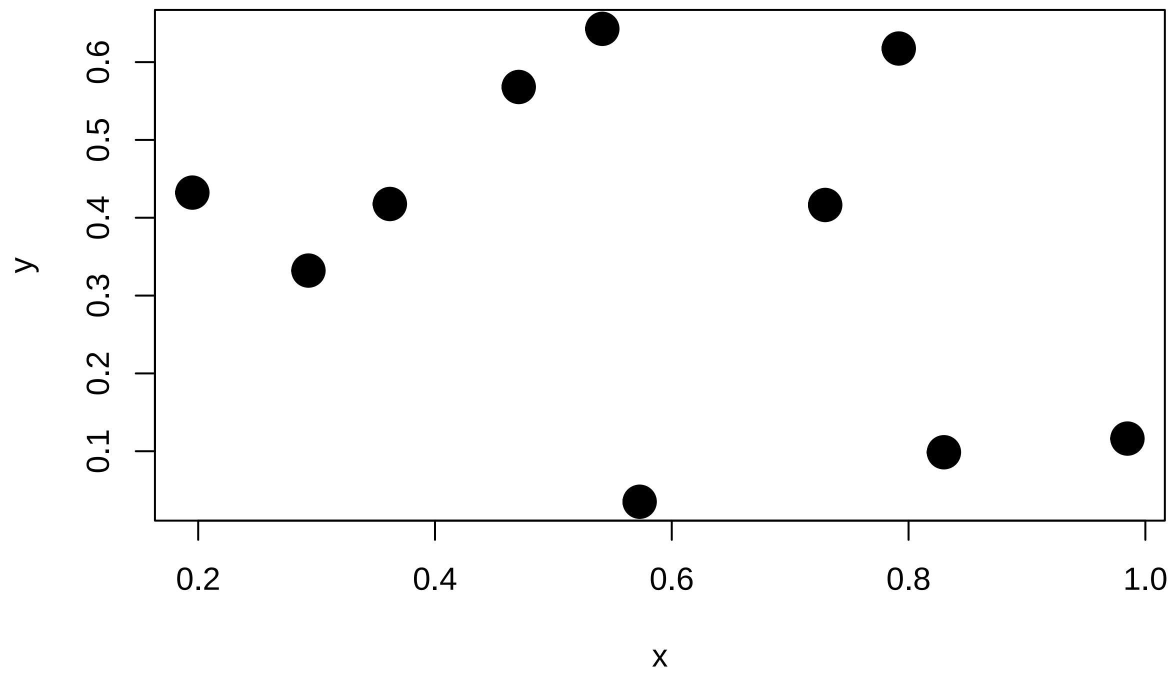
Why is it better to force(x) instead of just x?

Answer: *`force(x)` is similar to* `x`.* *As mentioned in **Advanced R, we prefer this explicit form, because ***using this function clearly indicates that you're forcing evaluation, not that you've accidentally typed x."***

2) Base R contains two function factories, approxfun() and ecdf(). Read their documentation and experiment to figure out what the functions do and what they return.

Answer: *Let's begin with `approxfun()` as it is used within `ecdf()` as well: `approxfun()` takes a combination of data points (x and y values) as input and returns a stepwise linear (or constant) interpolation function. To find out what this means exactly, we first create a few random data points.*

```
x <- runif(10)
y <- runif(10)
plot(x, y, lwd = 10)
```



*Next, we use **`approxfun()`** to construct the linear and constant interpolation functions for our **`x`** and **`y`** values.*

```
y <- runif(10)
f_lin <- approxfun(x,y)
f_con <- approxfun(x, y, method = "constant")
# Both functions exactly reproduce their input y values
identical(f_lin(x), y)
```

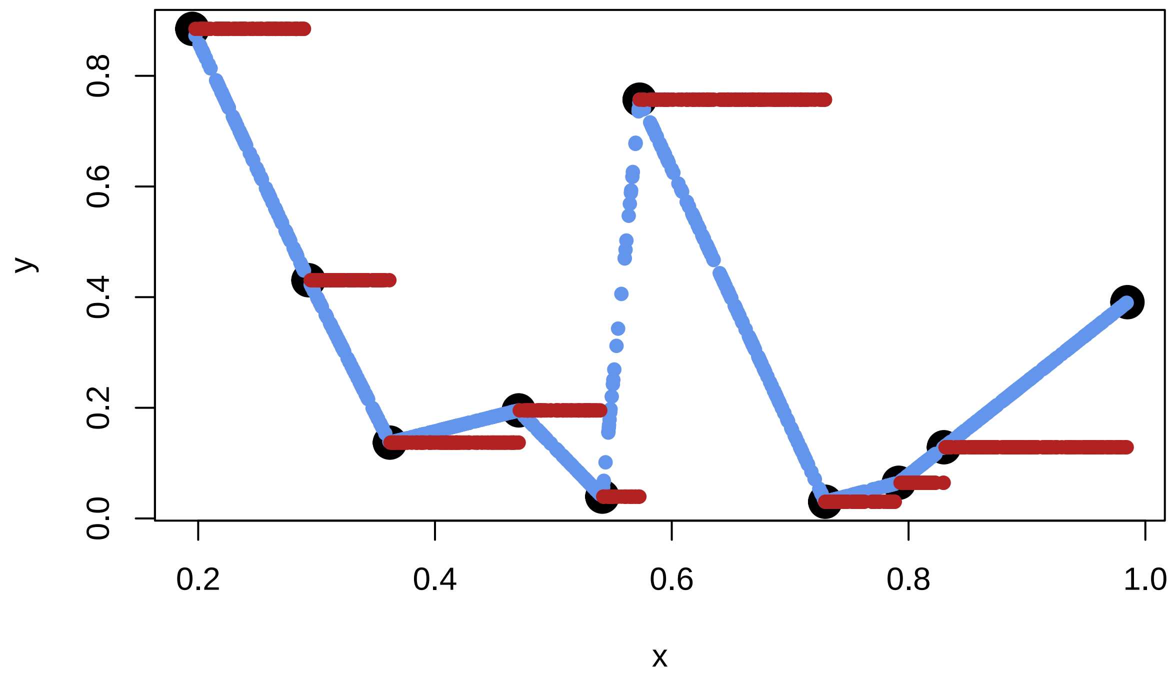
```
## [1] TRUE
```

```
identical(f_con(x), y)
```

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

When we apply these functions to new x values, these are mapped to the lines connecting the initial y values (linear case) or to the same y value as for the next smallest initial x value (constant case).

```
set.seed(123)
x_new <- runif(1000)
plot(x, y, lwd = 10)
points(x_new, f_lin(x_new), col = "cornflowerblue", pch = 16)
points(x_new, f_con(x_new), col = "firebrick", pch = 16)
```



However, both functions are only defined within ***`range(x)`***.

```
f_lin(range(x))
```

```
## [1] 0.8849239 0.3909381
```

```
f_con(range(x))
```

```
## [1] 0.8849239 0.3909381
```

```
(eps <- .Machine$double.neg.eps)
```

```
## [1] 1.110223e-16
```

```
f_lin(c(min(x) - eps, max(x) + eps))
```

```
## [1] NA NA
```

```
f_con(c(min(x) - eps, max(x) + eps))
```

```
## [1] NA NA
```

To change this behaviour, one can set `rule = 2`. This leads to the result that for values outside of `range(x)` the boundary values of the function are returned.

```
f_lin <- approxfun(x, y, rule = 2)
f_con <- approxfun(x, y, method = "constant", rule = 2)
f_lin(c(-Inf, Inf))
```

```
## [1] 0.8849239 0.3909381
```

```
f_con(c(-Inf, Inf))
```

```
## [1] 0.8849239 0.3909381
```

Another option is to customise the return values as individual constants for each side via `yleft` and/or `yright`.

```
f_lin <- approxfun(x, y, yleft = 5)
f_con <- approxfun(x, y, method = "constant", yleft = 5, yright = -5)
f_lin(c(-Inf, Inf))
```

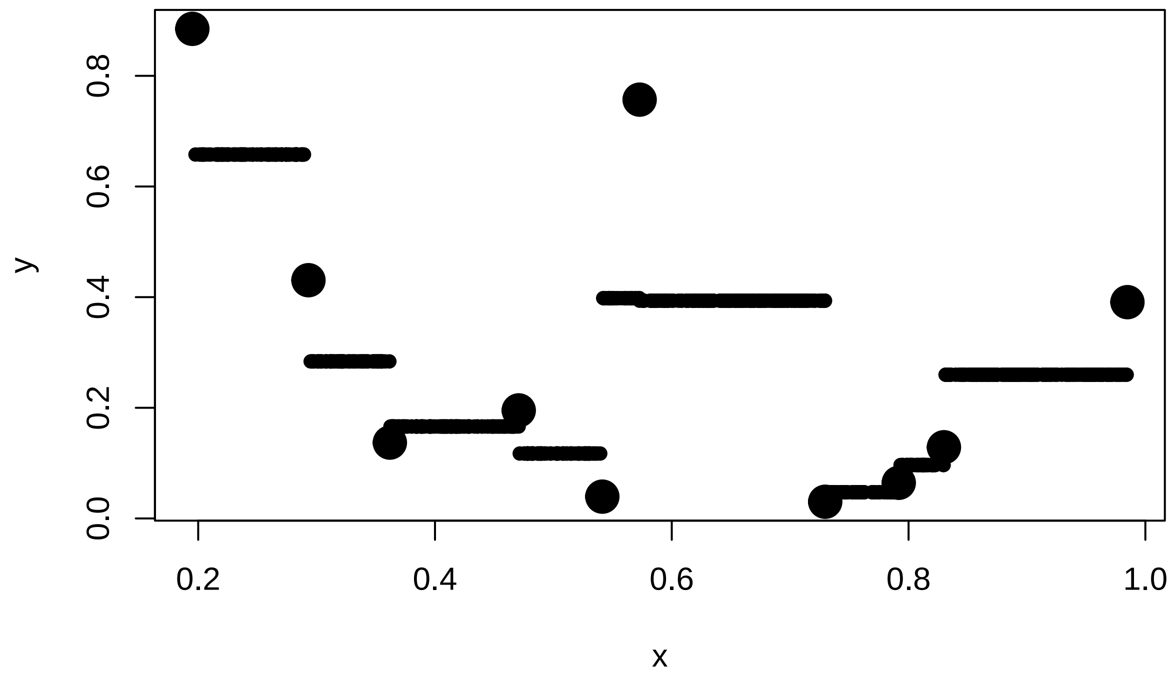
```
## [1]  5 NA
```

```
f_con(c(-Inf, Inf))
```

```
## [1]  5 -5
```

Further, `approxfun()` provides the option to shift the y values for `method = "constant"` between their left and right values. According to the documentation this indicates a compromise between left- and right-continuous steps.

```
f_con <- approxfun(x, y, method = "constant", f = .5)
plot(x, y, lwd = 10)
points(x_new, f_con(x_new), pch = 16)
```

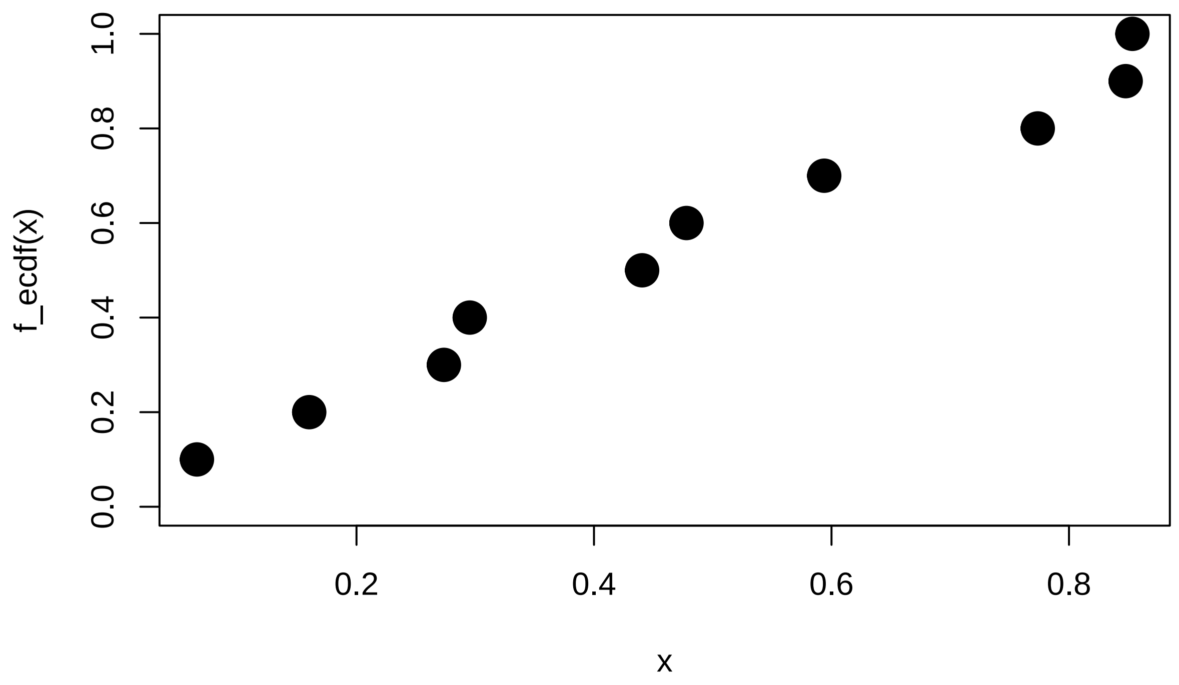


Finally, the `ties` *argument allows to aggregate y values if multiple ones were provided for the same x value. For example, in the following line we use* `mean()` *to aggregate these y values before they are used for the interpolation* `approxfun(x = c(1,1,2), y = 1:3, ties = mean)`*. Next, we focus on* `ecdf()`*. "ecdf" is an acronym for empirical cumulative distribution function. For a numeric vector of density values,* `ecdf()` *initially creates the (x, y) pairs for the nodes of the density function and then passes these pairs to* `approxfun()`*, which gets called with specifically adapted settings (*`approxfun(vals, cumsum(tabulate(match(x, vals)))/n, method = "constant", yleft = 0, yright = 1, f = 0, ties = "ordered")`*).*

```
x <- runif(10)
f_ecdf <- ecdf(x)
class(f_ecdf)
```

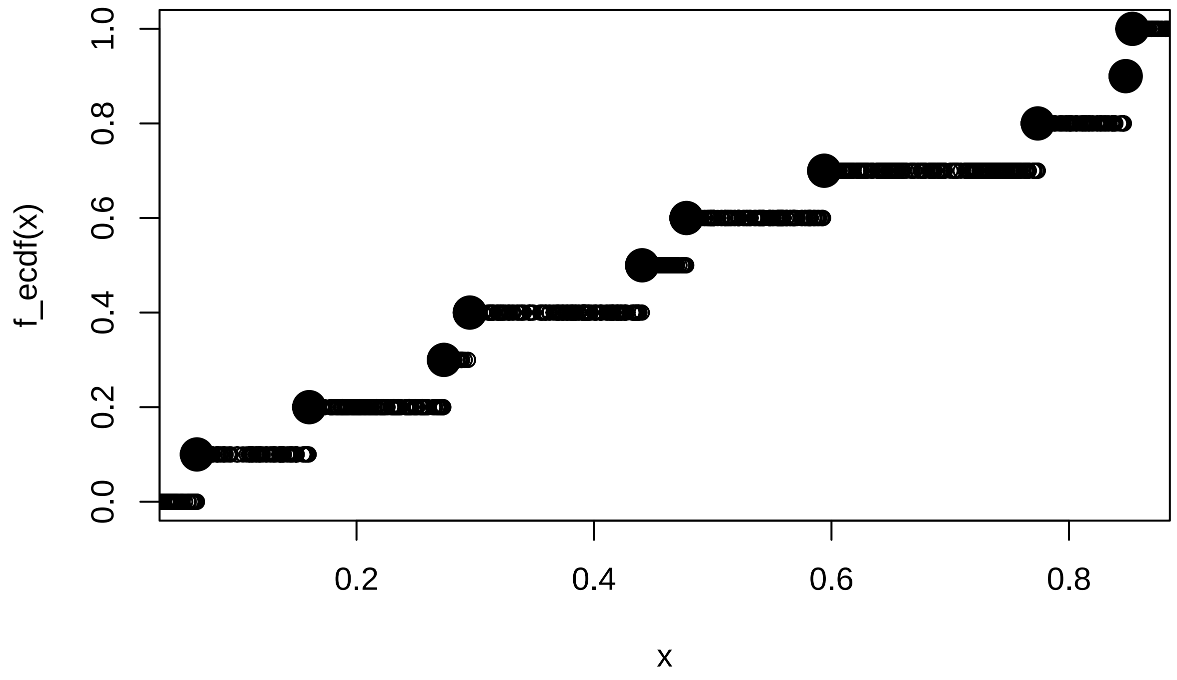
```
## [1] "ecdf"     "stepfun"  "function"
```

```
plot(x, f_ecdf(x), lwd = 10, ylim = 0:1)
```



New values are then mapped on the y value of the next smallest x value from within the initial input.

```
x_new <- runif(1000)
plot(x, f_ecdf(x), lwd = 10, ylim = 0:1)
points(x_new, f_ecdf(x_new), ylim = 0:1)
```



3) Create a function pick() that takes an index, i, as an argument and returns a function with an argument x that subsets x with i.

```
pick(1)(x)
# should be equivalent to
x[[1]]
lapply(mtcars, pick(5))
# should be equivalent to
lapply(mtcars, function(x) x[[5]])
```

Answer: *In this exercise* ***pick(i)*** *acts as a function factory, which returns the required subsetting function.*

```
pick <- function(i) {
  force(i)

  function(x) x[[i]]
}
x <- 1:3
identical(x[[1]], pick(1)(x))
```

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

```
identical(
  lapply(mtcars, function(x) x[[5]]),
  lapply(mtcars, pick(5))
)
```

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

4) Create a function that creates functions that compute the ithcentral moment of a numeric vector. You can test it by running the following code:

m1 <- moment(1) m2 <- moment(2)

x <- runif(100) stopifnot(all.equal(m1(x), 0)) stopifnot(all.equal(m2(x), var(x) * 99 / 100))

Answer:

*The first moment is closely related to the mean and describes the average deviation from the mean, which is 0 . The second moment describes the variance of the input data. If we want to compare it to `var()`, we need to undo [Bessel's correction] by multiplying with $\frac{N-1}{N}$.

```
moment <- function(i) {
  force(i)

  function(x) sum((x - mean(x)) ^ i) / length(x)
}
m1 <- moment(1)
m2 <- moment(2)
x <- runif(100)
all.equal(m1(x), 0)  # removed stopifnot() for clarity
```

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

```
all.equal(m2(x), var(x) * 99 / 100)
```

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

5) What happens if you don't use a closure? Make predictions, then verify with the code below.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}
```

Without the captured and encapsulated environment of a closure the counts will be stored in the global environment. Here they can be overwritten or deleted as well as interfere with other counters.

```
new_counter2()
```

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

```
i
```

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

```
new_counter2()
```

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

```
i
```

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

```
i <- 0
new_counter2()
```

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

```
i
```

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

6)What happens if you use <- instead of «-? Make predictions, then verify with the code below.

```
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

Without the super assignment `<<-`*, the counter will always return 1. The counter always starts in a new execution environment within the same enclosing environment, which contains an unchanged value for* `i` *(in this case it remains 0).*

```
new_counter_3 <- new_counter3()
new_counter_3()
```

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

```
new_counter_3()
```

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