More Tips and Tricks

Abhinav Anand, IIMB

2019/06/18

Scripts in R

Scripts are files (ending in `.R`) that contain sequences of instructions that we can command a machine to follow. Consier a typical empirical project: we need to read data stored in some type of file; then clean and tidy it, then process it and use descriptive statistics and plots to delineate its features etc. All this can be achieved by storing a sequence of instructions in a script file. Data can be read using the the `read_csv()` function, processing can be done using the package `dplyr` etc.

Linear regression in R

Generally a linear model takes the following form:

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_m x_m + u$$

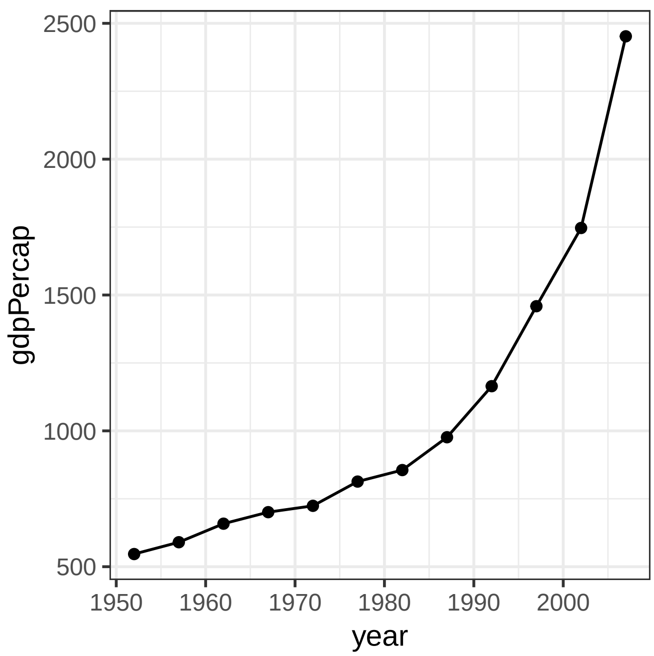
where $u_{n\times 1}$ is the error term. This setup corresponds to an overdetermined linear system of equations leading to a least squares solution:

$$\hat{\beta} = (X^\top X)^{-1} X^\top y$$

where the explanatory matrix $X_{m\times n}$ contains independent variables $x_1, \ldots, x_m$ as column vectors of size $n \times 1$.

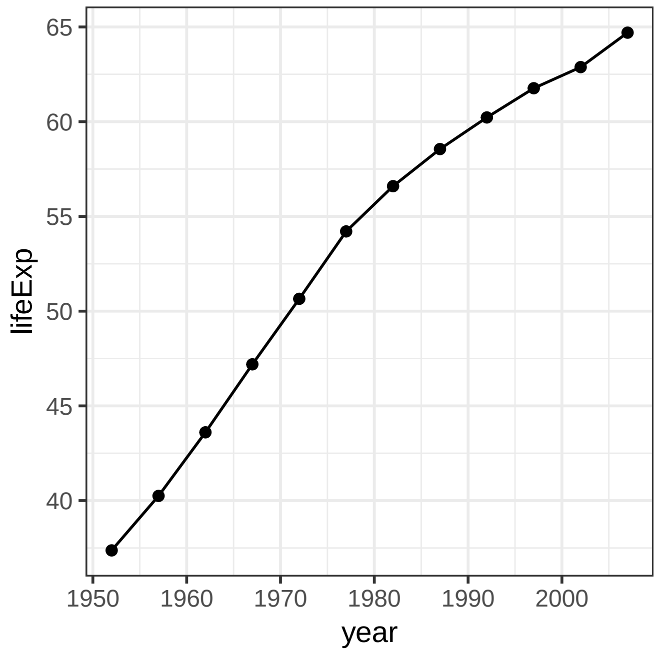
One of the strengths of R is the flexibility and support it offers for linear regression modeling. In order to illustrate it more fully, let us consider data for India in the `gapminder` dataset.

```
data_Ind <- gapminder::gapminder %>%
  dplyr::filter(country == "India")

ggplot(data_Ind, aes(year, gdpPercap)) +
  geom_point() +
  geom_line()
```



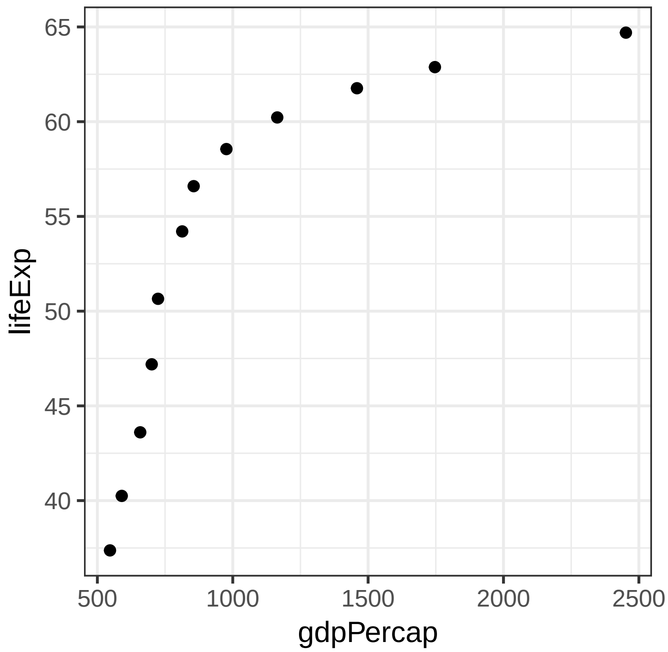
We see that there has been a large increase in GDP per capita in India. A similar trend is observed for life expectancy:

```
ggplot(data_Ind, aes(year, lifeExp)) +
  geom_point() +
  geom_line()
```



What about the relationship between the two? For example, (all else equal) does GDP per capita of India explain the life expectancy trends observed?

```
ggplot(data_Ind, aes(gdpPercap, lifeExp)) +
  geom_point()
```



This suggests that the two variables share a positive relation. We can try to check this by means of a linear regression in the following way:

$$\text{life exp} = \beta_0 + \beta_1(\text{gdp percap}) + u$$

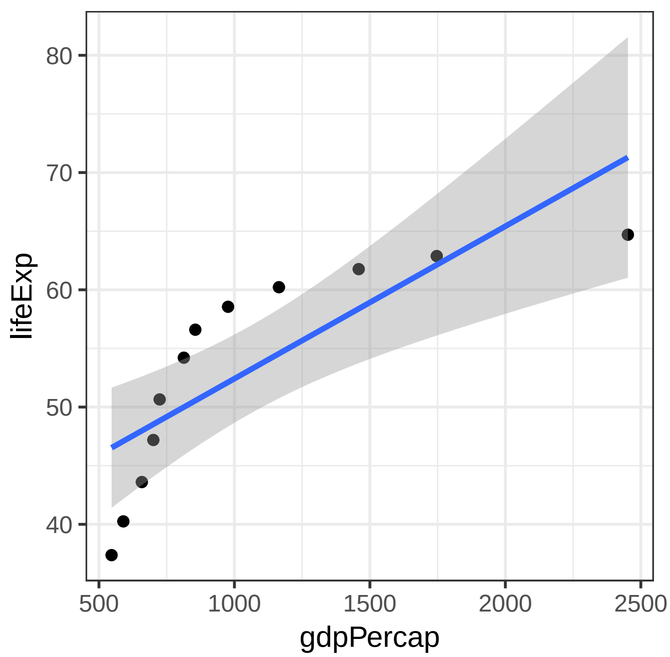
In order to implement this step in R is via the following:

```
lm_formula <- lifeExp ~ gdpPercap

lm_life_gdppc <- lm(data = data_Ind, formula = lm_formula)

summary(lm_life_gdppc)
```

```
##
## Call:
## lm(formula = lm_formula, data = data_Ind)
##
## Residuals:
```

```
##      Min      1Q  Median      3Q     Max
## -9.155 -4.931  1.284  4.576  6.437
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 39.423336   3.659432  10.773    8e-07 ***
## gdpPercap    0.012998   0.003075   4.227  0.00175 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.816 on 10 degrees of freedom
## Multiple R-squared:  0.6411, Adjusted R-squared:  0.6052
## F-statistic: 17.86 on 1 and 10 DF,  p-value: 0.001753
```

```r
ggplot(data_Ind, aes(gdpPercap, lifeExp)) +
  geom_point() +
  geom_smooth(method = "lm")
```



What is this object `lm_life_gdppc`? What is its structure? We can quickly check by accessing its contents:

```
names(lm_life_gdppc)
```

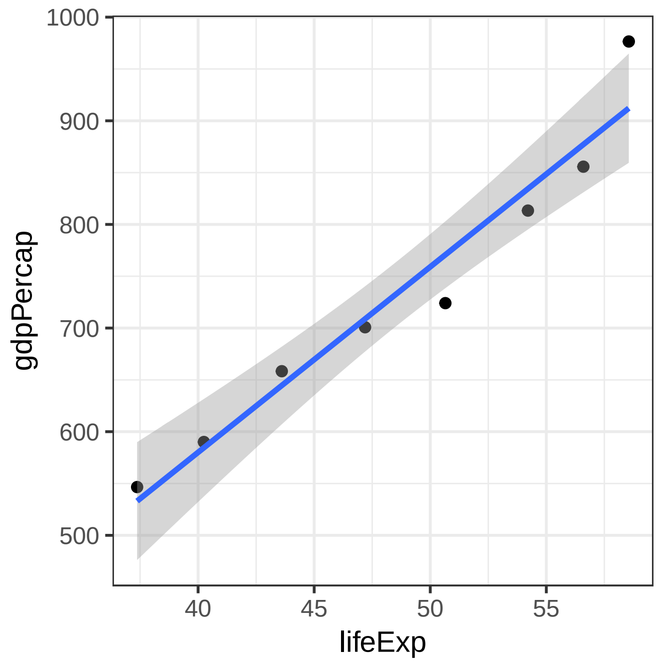
```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

What about some subset of data, say the period before 1990?

```
lm(filter(data_Ind, year <= 1990), formula = lm_formula) %>%
  summary()
```

```
##
## Call:
## lm(formula = lm_formula, data = filter(data_Ind, year <= 1990))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.8626 -1.0747 -0.1943  1.4541  2.5804
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  9.80149    3.83775   2.554   0.0433 *
## gdpPercap    0.05286    0.00515  10.264 4.99e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.945 on 6 degrees of freedom
## Multiple R-squared:  0.9461, Adjusted R-squared:  0.9371
## F-statistic: 105.3 on 1 and 6 DF,  p-value: 4.992e-05
```

```r
ggplot(filter(data_Ind, year <= 1990), aes(lifeExp, gdpPercap)) +
  geom_point() +
  geom_smooth(method = "lm")
```



Nonlinear relationships

The plot between the dependent and independent variable suggest a nonlinear relationship. Can we test this simply? Let's consider the following modification:

$$\text{life exp} = \beta_0 + \beta_1(\text{gdp percap})^2 + u$$

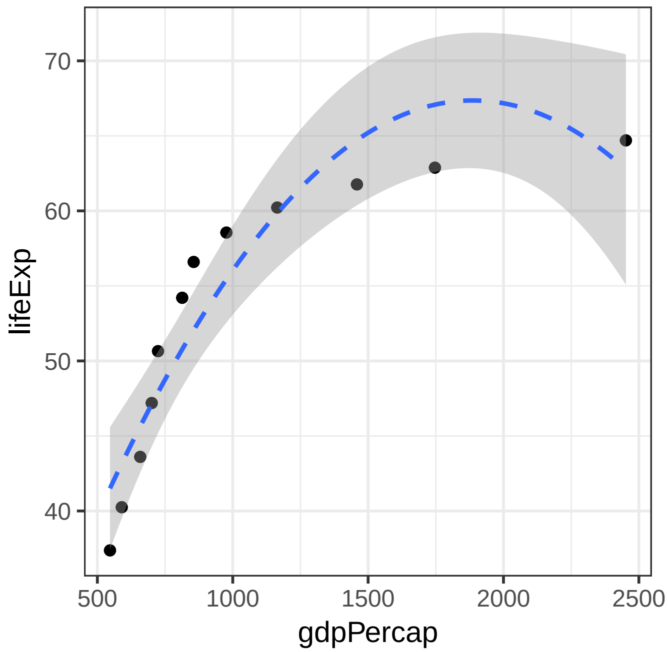
In general, R can accommodate independent variables involving mathematical operators in a regression equation with the function `I()`.

```r
lm_formula_quad <- lifeExp ~ I(gdpPercap)^2

lm_life_gdppc_quad <- lm(data = data_Ind, formula = lm_formula_quad)
```

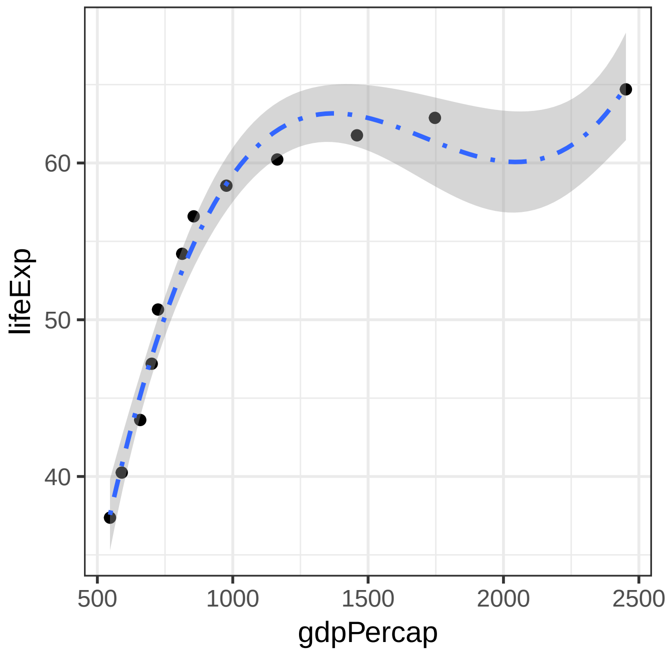
```r
summary(lm_life_gdppc_quad)
```

```
##
## Call:
## lm(formula = lm_formula_quad, data = data_Ind)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -9.155 -4.931  1.284  4.576  6.437
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  39.423336   3.659432  10.773    8e-07 ***
## I(gdpPercap)  0.012998   0.003075   4.227  0.00175 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.816 on 10 degrees of freedom
## Multiple R-squared:  0.6411, Adjusted R-squared:  0.6052
## F-statistic: 17.86 on 1 and 10 DF,  p-value: 0.001753
```

```r
ggplot(data_Ind, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  stat_smooth(method = "lm",
              formula = y ~ poly(x, 2), #polynomial order 2
              size = 0.8,
              linetype = "dashed"
              )
```




```r
# What about higher order polynomials?
ggplot(data_Ind, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  stat_smooth(method = "lm",
              formula = y ~ poly(x, 3), #polynomial order 3
              size = 0.8,
              linetype = "dotdash"
              )
```



Are visually better fits also evidence of better underlying models? This is a hard question to answer in general—all else equal we prefer models that are parsimonious (have fewer explanatory variables).

Functional Programming in R

Another very powerful feature of R is its support for functional programming, which in general, involves applying functions to arrays, dataframes, lists etc.

For example, how should one compute the mean across rows of a matrix?

```
df <- data.frame(C_1 = rnorm(10, 0, 1),
                 C_2 = rnorm(10, 1, 2),
                 C_3 = rnorm(10, 2, 3)
                 )


head(df)
```

```
##           C_1        C_2         C_3
## 1  0.5381310  2.0581087 -0.87550785
## 2  1.6755015  1.3252460  0.76989995
## 3 -1.2965104  5.2609430  3.65960670
## 4  0.4062685  0.2999016  4.13398703
## 5  1.0145668  3.8479344 -3.90830778
## 6 -1.0538232 -0.2574456 -0.07364864
```

```r
# One way to solve the problem
rmean_1 <- rowMeans(df)
print(rmean_1)
```

```
##  [1]  0.5735773  1.2568825  2.5413464  1.6133857  0.3180645 -0.4616391
##  [7]  0.2895387  0.3109647  1.6144940  0.8859763
```

```r
# Another more 'functional' way
func_mean <- function(vec)
{
  return(mean(vec, na.rm = T))
}


# Apply function on rows
rmean_2 <- apply(df, 1, func_mean)
print(rmean_1)
```

```
##  [1]  0.5735773  1.2568825  2.5413464  1.6133857  0.3180645 -0.4616391
##  [7]  0.2895387  0.3109647  1.6144940  0.8859763
```

```r
# Apply function on columns
rmean_3 <- apply(df, 2, func_mean)
print(rmean_3)
```

```
##          C_1          C_2          C_3
## -0.002382187  1.569520037  1.115639449
```

Note how to use the `apply()` function. We `apply()` the function over rows or columns or other dimensions. In general that's the philosophy of the `apply()` family of functions, which includes functions `lapply()` (list-apply) and `sapply()` (simplify-apply) etc. The function `lapply` returns a list and `sapply` a vector (if possible). In both cases the first argument is a list (or dataframe) and the second argument is the name of a function.

What is a list? It's essentially a more general version of a dataframe and can contain not only dissimilar data types but also, say dataframes within them.

```
temp_list <- list(a = runif(10),
                  b = "Happy birthday",
                  c = data.frame(x = rnorm(10, 0, 1)),
                  d = sample(letters, 7, replace = TRUE)
                  )
str(temp_list) #structure of the list
```

```
## List of 4
##  $ a: num [1:10] 0.704 0.597 0.584 0.173 0.474 ...
##  $ b: chr "Happy birthday"
##  $ c:'data.frame':   10 obs. of  1 variable:
##   ..$ x: num [1:10] 0.347 -2.513 -0.394 0.044 -1.187 ...
##  $ d: chr [1:7] "e" "u" "z" "j" ...
```

```
# lapply() is used to apply the same function to each
# "cell" of the list
lapply(temp_list, is.numeric) #check if each cell is numeric
```

```
## $a
## [1] TRUE
##
## $b
## [1] FALSE
```

```
##
## $c
## [1] FALSE
##
## $d
## [1] FALSE
```

```r
# contrast with sapply()
sapply(temp_list, is.numeric)
```

```
##     a     b     c     d
##  TRUE FALSE FALSE FALSE
```

The `map()` family from purrr

The `map` function does the exact same operation as `apply` but is consistent with the output format type. For example `map()` returns a list, `map_dbl()` returns a double type vector, `map_int()` returns an integer type vector etc. As with `read_csv`, the `map` family improves upon the base R code by being faster and more consistent.

```r
map(df, mean)
```

```
## $C_1
## [1] -0.002382187
##
## $C_2
## [1] 1.56952
##
## $C_3
## [1] 1.115639
```

```r
map_dbl(df, median)
```

```
##       C_1       C_2       C_3
## 0.4158354 0.8125738 1.1505824
```

```r
# Also compare this
z <- list(x = 1:3, y = 4:5)
map_int(z, length) #names are preserved
```

```
## x y
## 3 2
```