THE AUSTRALIAN ECONOMIC REVIEW

The Australian Economic Review, vol. 00, no. 00, pp. 1–19 DOI: 10.1111/1467-8462.12362

For the student

Applied R in the Classroom

J. D. Long and Dusty Turner*

Abstract

R, with support from a number of add-on packages, is an excellent teaching tool that can greatly enhance learning of exploratory data analysis (EDA) and linear regression. This article illustrates using R along with the packages `tidyverse`*,* `GGally`*,* `esquisse` *and* `lindia` *to walk through an example of basic EDA and linear regression that might be used in an introductory class along with code examples for every step. Teaching with R can give learners control of their analysis, lower the intimidating coding barrier, and provide a platform on which to learn modeling and analysis.*

* Long: RenaissanceRe Finance Inc, 3128 Highwoods Blvd, #230 Raleigh, NC 27604, USA. Turner: Center for Army Analysis, 6001 Goethals Road, Fort Belvoir, VA, 22060, USA and United States Military Academy Department of Mathematical Sciences, 646 Swift Road, West Point, NY 10996-1905, USA. Corresponding author: Turner, email <dusty.s.turner@gmail.com>. Opinions, conclusions, and recommendations expressed or implied within are solely those of the authors and do not necessarily represent the views of the United States Army, the Department of Defense, or any other US government agency.

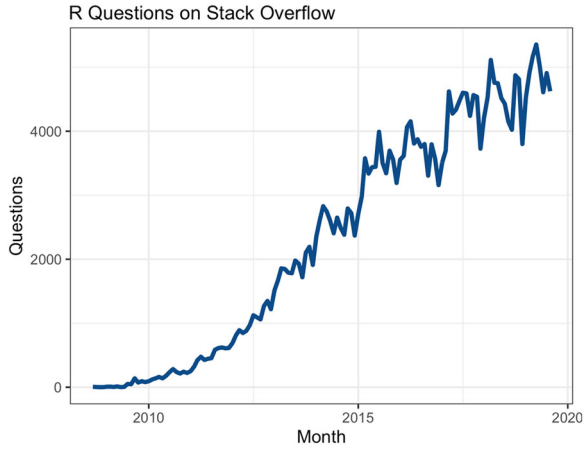
1. Introduction

The father of modern economics, Adam Smith, once frustratingly stated (Smith and Krueger 2003):

> The discipline of colleges and universities is in general contrived, not for the benefit of the students, but for the … ease of the masters.

Adam Smith here makes the claim that methods of instruction used by professors are those that are easiest for the instructor, but not necessarily what is best for the student. This is understandable, as research and other administrative demands force instructors to rely on tried and tested teaching techniques that are familiar to the teacher and possibly not what will best equip the student to be most successful in their career. The authors of this article will make the case that the R open source statistical programming language can bridge this gap between Smith's proverbial teacher's ease and a student's benefit. As R continues to be one of the more popular coding languages for statistical analysis with ever-increasing technical support, the barrier for entry keeps falling. There are many tools available as add-ons to R that can aide the teaching process to get students loading and exploring data quickly with manageable overhead for the instructor. With ample open source support, a wide acceptance in industry, and many additional features to explore and

Figure 1 R Growth on Stack Overflow



present data, teaching with R both satisfies the ease for the instructor and has long-term benefits for the students. Learning the basics of R means students have a tool set that they can take with them to either future academic career paths or to apply in industry.

The growth of the R language can be appreciated if we look at the increase in R questions being asked each month on the popular question-and-answer site, Stack Overflow (Stack 2019). In Figure 1 we can see that R-related new question activity on the site has grown steadily from the site's origin in 2008. Currently there are over 4,000 new R-related questions every month.

Equipping students with a popular tool that also enables them to work quickly while learning is something we are certain Adam Smith would appreciate.

2. The R Ecosystem

The advantages of using R in an academic setting or in industry are myriad. Since R and all the tooling we discuss below are open source, they have no financial costs to adoption. In addition, R has a very rich ecosystem of add-on packages that expand R and add functionality. These packages add features ranging from adding the ability to connect to commercial database systems to implementation of new machine learning algorithms. The nature of open source also allows user-written packages and routines to be easily distributed. In addition, the popularity of R in academia leads to availability of packages containing state-of-the-art methods.

2.1 CRAN

The online home of R is the Comprehensive R Archive Network (CRAN). CRAN is where a new user can download R and access packages that expand R's functionality. There are currently more than 10,000 R packages hosted on CRAN for free download (CRAN 2019). Some new users to R are overwhelmed by the sheer volume of packages. To help make sense of the CRAN ecosystem, CRAN has published CRAN Task Views which organise popular packages into categories of use (e.g., econometrics or spatial statistics). Each task view is written and maintained by a subject matter expert. There are more than 35 such task views that can help new R users make sense of the packages available in their areas of interest and know which packages are recommended by an expert in their domain of interest. This provides instructors and students a curated view into packages that might match their interests or field of study.[1]

2.2 RStudio

When R is downloaded from CRAN, it comes with an engine for executing R code along with a few core packages for doing statistical analysis and graphics. Collectively these tools are referred to as 'Base R'. Base R comes with a basic text editor for editing and executing R scripts. However, most users quickly discover that writing R code is easier with an integrated development environment (IDE). The most popular IDE for R is RStudio Desktop which we highly recommend for teaching.

RStudio.com provides RStudio Desktop for free as open source software. In addition to the desktop IDE, RStudio makes a server-based IDE for using R on remote machines. RStudio Server is also available in a professional version that offers more features such as authentication and collaborative editing. RStudio offers its professional tools to academics for free.[2]

For many instructors, the freely available RStudio.cloud service[3] greatly simplifies instruction by providing a fully functional and configured R and RStudio environment running on cloud-hosted hardware. For screenshots in this article we will exclusively use RStudio.cloud in our examples. Instructors can set up projects in RStudio.cloud and share those projects with students to simplify distributing course material. The basic RStudio.cloud interface is shown in Figure 2.

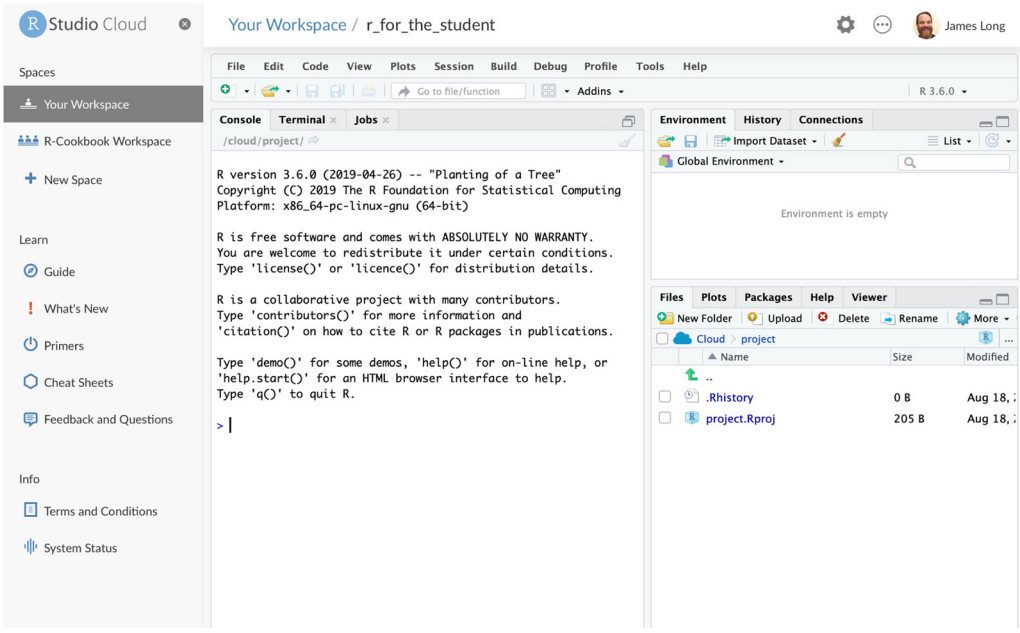
Instructors who do not want to use the cloud solutions, or cannot access them because of connectivity restrictions, can download and install R and RStudio on local hardware. For details on downloading and installing, see section 1.1 in R Cookbook (Teetor and Long 2019).[4]

2.3 Projects

RStudio introduces a powerful organisational tool called an RStudio Project. Projects help users by doing the following:

- storing RStudio project settings;
- restoring window position in RStudio so when a project is closed and repoened, window contents are preserved;
- setting the working directory.

Figure 2 RStudio.cloud



RStudio creates a project file with an *.Rproj* extension in the project directory. RStudio also creates a hidden directory, *Rproj.user*, for temporary files related to your project.

We have found that helping students organise their files using projects from the start helps students build good organisational practices and prevents lost files and file path-related issues that commonly flummox new learners. However, instructors should expect to teach the basics of absolute and relative file paths,[5] as this concept is sometimes new to learners and can slow the learning progress.

2.4 Tidyverse

In addition to the RStudio IDE, the RStudio company supports the development of a number of open source packages designed to work together to make R easier to use and faster to learn. These libraries are collectively known as the 'tidyverse'. The most concise definition of the tidyverse comes directly from its originator and core maintainer, Wickham (2015):

> The tidyverse is a set of packages that work in harmony because they share common data representations and API design. The tidyverse package is designed to make it easy to install and load core packages from the tidyverse in a single command. The best place to learn about all the packages in the tidyverse and how they fit together is *R for Data Science*.

The authors have had very good experiences with introducing learners to the tidyverse from the very beginning of the learning journey because these tools help learners achieve very quick successes which, in turn, keeps them engaged in the learning process. The popular plotting package `ggplot2` and the data manipulation package `dplyr` are both core tidyverse packages.

The tidyverse meta-package, like any CRAN package, can be installed from the R Console:

```
install.packages("tidyverse")
```

2.4.1 Tidyverse Packages

When a user installs the tidyverse, 19 packages are installed (RStudio 2019; Wickham 2019). When the user loads the tidyverse using `library(tidyverse)` a core subset of eight packages are loaded into R. To use any of the packages not loaded with the core tidyverse, a user must explicitly load those packages (e.g., `library(readxl)`) or name the packages using the package name prefix (e.g., `readxl::read_xlsx()` to run the `read_xlsx()` function from the `readxl` package).

The packages listed below are in the 'core tidyverse' and loaded with `library(tidyverse)`.

Core tidyverse

`ggplot2`: data visualisation
`dplyr`: data manipulation
`tidyr`: data reshaping
`readr`: data import
`purrr`: functional programming
`tibble`: tidy dataframes
`stringr`: string manipulation
`forcats`: factor use

Additional tidyverse

There are 11 additional tidyverse packages that install, but do not automatically load. These add functions for more specialised uses (RStudio 2019; Wickham 2019).

Import

`readxl`: reading Excel files
`haven`: reading SPSS, Stata, and SAS data
`jsonlite`: manipulating JSON
`xml2`: reading xml
`httr`: accessing web APIs
`rvest`: web scraping
`feather`: data sharing with Python and beyond

Wrangle

`lubridate`: date manipulation
`hms`: time-of-day manipulation

Modelling

`modelr`: modelling pipelines

`broom`: takes model results and makes them tidy[6]

3. Harnessing the Power of R, Tidyverse and Other Helpful Packages

In the following sections, we will highlight some of the features of R and the `tidyverse` and how they can be useful in a classroom setting while teaching new students. The reader will notice that the authors use many packages to support their analysis and recommend teachers do the same when instructing students. These prebuilt, add-on packages make syntax simple, easy to interpret and less intimidating for new users.

3.1 Loading Data (Gapminder)

To aide our instructions, we will explore the Gapminder data set. The Gapminder data set is created by the Gapminder Foundation which is a non-profit organisation that promotes sustainable global development (Gapminder 2019).

Showing students how to load in example data is a crucial first step. Data can be easily loaded from the local file system:

```
library(tidyverse)
gapminder <- read_csv("01_data/gapminder.csv")
```

Or directly from a URL:

```
gapminder <- read_csv("https://raw.githubusercontent.com/CerebralMastication/r_for_the_student/master/01_data/gapminder.csv")
```

> As a quick aside, a common stumbling block for students is executing lines of code. RStudio makes this simple. `Ctrl+Enter` (for Windows) or `Command+Enter` (for MAC) will execute the line of code where the cursor currently exists. Students can also execute multiple lines of code by highlighting the desired code and pressing `Ctrl+Enter` (Windows) or `Command+Enter` (MAC).

Once the student reads the data into R, the environment tab in the top right of the computer will reflect that the data are loaded. An example environment pane after loading the gapminder data is shown in Figure 3 on the next page.

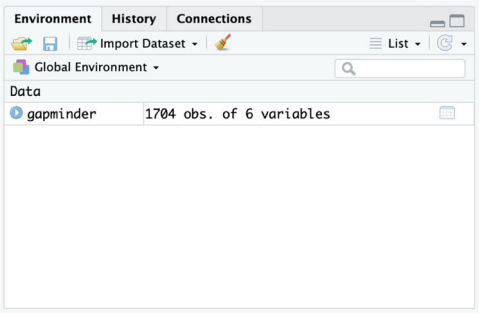
3.2 Initial Data Exploration

Simply printing the resulting data frame can give students a bundle of information about the data just loaded. Let us look at the output below.

```
gapminder

## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Afghanistan Asia       1957    30.3  9240934      821.
##  3 Afghanistan Asia       1962    32.0 10267083      853.
##  4 Afghanistan Asia       1967    34.0 11537966      836.
##  5 Afghanistan Asia       1972    36.1 13079460      740.
##  6 Afghanistan Asia       1977    38.4 14880372      786.
##  7 Afghanistan Asia       1982    39.9 12881816      978.
##  8 Afghanistan Asia       1987    40.8 13867957      852.
##  9 Afghanistan Asia       1992    41.7 16317921      649.
## 10 Afghanistan Asia       1997    41.8 22227415      635.
## # … with 1,694 more rows
```

Figure 3 Environment with Gapminder Data



From the output on the previous page, one can see the `gapminder` data set is a `tibble` consisting of 1,704 observational units with six columns of information. When reading in data, the `read_csv` command identifies the most likely class of data for each column. When we print the data out in R, the display lists the class below the column name. We can see that we have a mix of `character` and `double` class types. A `tibble` is a tidyverse take on R's traditional `data.frame`. We have found that it usually is sufficient to tell students that 'a `tibble` is a table'. Since the goal is to get students using data to *do* things, we often do not spend excessive time on definitions and instead focus on helping students doing something interesting.

Students also notice that printing out the data does not provide everything an analyst would like to know about the data. One can see that the output provides information about countries, but are they all Afghanistan? Likely no, as you can see there are 1,694 more rows and they probably are not all Afghanistan. Let us look at other ways to understand the data (Figure 4).

We recommend using the `skim` command from the `skimr` library (Quinn et al. 2019). Students will need to download the `skimr` package from CRAN.

```
install.packages("skimr")
library(skimr)
skim(gapminder)
```

The `skim` output in Figure 4 gives students a much better overview of the data. `skim` separates the data by class and provides important information for each data type. We find that the `skim` function gives students the quickest and best description of the data. Other options of presenting detailed information about the data include the `summary(gapminder)` and `glimpse(gapminder)`. We typically only teach one of these techniques to quickly motivate the students by looking at the data instead of becoming distracted by worrying about the pros and cons of the different ways of exploring the data.

In the output above, for each class of data (`character/numeric`), `skim` provides how many missing, complete and total (n) observations for each column.

For the character class, `skim` provides additional information about each column. This includes the minimum/maximum length of each character string as well as the number of unique (n_unique) observations.

For the numeric class, `skim` provides mean, standard deviation (sd) and percentile breaks (p0, p25, …, p100). It also provides a small histogram to help visualise the distribution of your numeric data (hist).

This clearly does not give the student a full understanding of a data set, but it is a good start. With just a `skim` the student still does

Figure 4 Skim of Gapminder Data

```
── Variable type:character ──────────────────────────────────────────────────────
  variable missing complete    n min max empty n_unique
 continent       0     1704 1704   4   8     0        5
   country       0     1704 1704   4  24     0      142

── Variable type:numeric ────────────────────────────────────────────────────────
  variable missing complete    n      mean         sd       p0       p25      p50      p75         p100     hist
 gdpPercap       0     1704 1704  7215.33 9857.45      241.17    1202.06  3531.85  9325.46 113523.13
   lifeExp       0     1704 1704    59.47   12.92       23.6       48.2     60.71    70.85     82.6
       pop       0     1704 1704 3e+07       1.1e+08 60011    2793664    7e+06    2e+07        1.3e+09
      year       0     1704 1704  1979.5    17.27     1952       1965.75  1979.5   1993.25   2007
```

not understand which countries are in the data and many other curiosities. We will provide more in-depth methods of understanding the data in Section 3.4 after we look briefly at the 6 `dplyr` verbs that allow students to begin rapidly exploring the data.

```
gapminder_selected = select(gapminder, country, year, pop, gdpPercap)

gapminder_selected
```

```
## # A tibble: 1,704 x 4
##    country      year      pop gdpPercap
##    <chr>       <dbl>    <dbl>     <dbl>
##  1 Afghanistan  1952  8425333      779.
##  2 Afghanistan  1957  9240934      821.
##  3 Afghanistan  1962 10267083      853.
##  4 Afghanistan  1967 11537966      836.
##  5 Afghanistan  1972 13079460      740.
##  6 Afghanistan  1977 14880372      786.
##  7 Afghanistan  1982 12881816      978.
##  8 Afghanistan  1987 13867957      852.
##  9 Afghanistan  1992 16317921      649.
## 10 Afghanistan  1997 22227415      635.
## # … with 1,694 more rows
```

3.3 dplyr Verbs

Almost any time a student works with data, they will need to manipulate that data in some way. Below, we will introduce the main six `dplyr` verbs to help wrangle data to gain additional insight. These verbs—`select`, `filter`, `mutate`, `group_by`, `summarise` `arrange` are explained in detail below.

In order to motivate use of the aforementioned verbs, we will look to answer the following question:
What is the average gross domestic product (GDP) per country since 1980?

3.3.1 Verb 1: Select

To begin, we only need to work with certain columns. The relevant columns for this question are `country`, `year`, `pop` and `gdpPercap`. We can make this selection using the `select` function. The first argument in the `select` function is the data we wish to select from. The subsequent arguments are the names of the columns from the data we will select. In the code below, we save our selected columns into a new `tibble` called `gapminder_selected`. We use a different name for the output data frame so as to not overwrite the original data frame object.

Once we view the data, we see that we still have 1,704 rows but only four columns.

3.3.2 Verb 2: Filter

To further answer our question, we need to filter our data down to the years of interest. We can achieve this goal using the `filter` function. Like the `select` function, the first argument in the `filter` function is the data and the subsequent argument is the logical statement of which you wish to filter. In the code below, we filter our selected data and save our filtered data into a new tibble called `gapminder_filtered`.

```
gapminder_filtered = filter(gapminder_selected, year >= 1980)

gapminder_filtered
```

```
## # A tibble: 852 x 4
##    country      year      pop gdpPercap
##    <chr>       <dbl>    <dbl>     <dbl>
##  1 Afghanistan  1982 12881816      978.
##  2 Afghanistan  1987 13867957      852.
##  3 Afghanistan  1992 16317921      649.
##  4 Afghanistan  1997 22227415      635.
##  5 Afghanistan  2002 25268405      727.
##  6 Afghanistan  2007 31889923      975.
##  7 Albania      1982  2780097     3631.
##  8 Albania      1987  3075321     3739.
##  9 Albania      1992  3326498     2497.
## 10 Albania      1997  3428038     3193.
## # … with 842 more rows
```

Once we view the data, we see that our data now consist of 852 rows. This represents the rows of data since 1980.

3.3.3 Verb 3: Mutate

The next step in answering our question is creating a column that contains the GDP. The `mutate` function creates new columns according to a specific function that we provide. To answer our question, we need to determine the GDP in each year. To find the GDP, we need to multiply the `gdpPercap` by the `pop`. Similar to the previous two verbs, the first argument in the `mutate` function is the data. Subsequent arguments are columns you wish to create with corresponding formulas. In the code below, we mutate our filtered data and save our mutated data into a new tibble called `gapminder_mutated`.

```
gapminder_mutated = mutate(gapminder_filtered, GDP = gdpPercap * pop)

gapminder_mutated
```

```
## # A tibble: 852 x 5
##    country      year      pop gdpPercap          GDP
##    <chr>       <dbl>    <dbl>     <dbl>        <dbl>
##  1 Afghanistan  1982 12881816      978. 12598563401.
##  2 Afghanistan  1987 13867957      852. 11820990309.
##  3 Afghanistan  1992 16317921      649. 10595901589.
##  4 Afghanistan  1997 22227415      635. 14121995875.
##  5 Afghanistan  2002 25268405      727. 18363410424.
##  6 Afghanistan  2007 31889923      975. 31079291949.
##  7 Albania      1982  2780097     3631. 10094200603.
##  8 Albania      1987  3075321     3739. 11498418358.
##  9 Albania      1992  3326498     2497.  8307722183.
## 10 Albania      1997  3428038     3193. 10945912519.
## # … with 842 more rows
```

Once we view the data, we see the new column, `GDP`, has been added to the end of our tibble.

3.3.4 Verb 4: Group_By

The next step will be to group our data by the field of interest. In this instance, since we want to know the GDP by country, we need to group the data by country. A way to conceptualise this step is to think of placing each group of data into a specific room. In subsequent steps we will apply a function to each group (or room) of data. Just like the previous verbs, the first argument in the `group_by` function is the data. The following arguments are the columns you wish to group by. In the code below, we group our mutated data and save our grouped data into a new tibble called `gapminder_grouped`.

Students will notice that there appears to be no change to the data. This is mostly true as we have simply told R that we would like to apply future functions to each group of data instead of to the entire tibble. The only difference in output is a note explaining what the data has been grouped into and the number of groups. In this case, it explains the data is grouped by `country` and that there are 142 groups.

3.3.5 Verb 5: Summarise

Next, in order to determine the average GDP by country, we need to apply a function to each group we have identified. Specifically, we will need to take the average GDP over each country. Since we have already grouped by country, we next need to apply the summarise function. Like the previous verbs, the first argument in the `summarise` function is the data. The following arguments are the function to apply to each group. In the code on the next page, we summarise the grouped data and save the summarised data into a new tibble called `gapminder_summarised`.

```
gapminder_grouped = group_by(gapminder_mutated, country)

gapminder_grouped
```

```
## # A tibble: 852 x 5
## # Groups:   country [142]
##    country      year      pop gdpPercap          GDP
##    <chr>       <dbl>    <dbl>     <dbl>        <dbl>
##  1 Afghanistan  1982 12881816      978. 12598563401.
##  2 Afghanistan  1987 13867957      852. 11820990309.
##  3 Afghanistan  1992 16317921      649. 10595901589.
##  4 Afghanistan  1997 22227415      635. 14121995875.
##  5 Afghanistan  2002 25268405      727. 18363410424.
##  6 Afghanistan  2007 31889923      975. 31079291949.
##  7 Albania      1982  2780097     3631. 10094200603.
##  8 Albania      1987  3075321     3739. 11498418358.
##  9 Albania      1992  3326498     2497.  8307722183.
## 10 Albania      1997  3428038     3193. 10945912519.
## # … with 842 more rows
```

```
gapminder_summarised = summarise(gapminder_grouped, AVG_GDP = mean(GDP))

gapminder_summarised

## # A tibble: 142 x 2
##    country           AVG_GDP
##    <chr>               <dbl>
##  1 Afghanistan  16430025591.
##  2 Albania      13062766192.
##  3 Algeria     148613140752.
##  4 Angola       28940373965.
##  5 Argentina   353071702131.
##  6 Australia   477639321504.
##  7 Austria     225388780040.
##  8 Bahrain      12397050418.
##  9 Bangladesh  119954904364.
## 10 Belgium     271944511091.
## # … with 132 more rows
```

We now have an answer to our question. The tibble above shows the average GDP per country since 1980.

3.3.6 *Verb 6: Arrange*

However, we can refine our result to provide more understanding. Currently, our data are sorted alphabetically by country. This does not provide much insight. We can use the `arrange` function to sort the data by average GDP; either ascending or descending. Like all other verbs, the first argument in the `arrange` function is the data. The following arguments are the one or more columns by which you wish to sort. In the code below, we arrange our summarised data and save

```
gapminder_arragned = arrange(gapminder_summarised, AVG_GDP)

gapminder_arragned

## # A tibble: 142 x 2
##    country                      AVG_GDP
##    <chr>                          <dbl>
##  1 Sao Tome and Principe     213138942.
##  2 Comoros                   585013190.
##  3 Guinea-Bissau             788263198.
##  4 Gambia                    806874307.
##  5 Djibouti                  894915489.
##  6 Liberia                  1274437021.
##  7 Lesotho                  2041066151.
##  8 Equatorial Guinea        2131596160.
##  9 Eritrea                  2545841271.
## 10 Central African Republic 2670573945.
## # … with 132 more rows
```

```
gapminder_arragned_descending = arrange(gapminder_summarised, -AVG_GDP)

gapminder_arragned_descending
```

```
## # A tibble: 142 x 2
##    country        AVG_GDP
##    <chr>            <dbl>
##  1 United States  9.20e12
##  2 Japan          3.28e12
##  3 China          2.96e12
##  4 Germany        2.20e12
##  5 United Kingdom 1.48e12
##  6 France         1.48e12
##  7 India          1.39e12
##  8 Italy          1.33e12
##  9 Brazil         1.27e12
## 10 Mexico         9.26e11
## # … with 132 more rows
```

our arranged data into a new tibble called `gapminder_arranged`.

We see, from the output of two code chunks above, the countries with the smallest average GDP since 1980.

It may be more interesting, however, to sort the average GDP in descending order so we can learn which countries have the highest average GDP. To do this we simply place a 'minus' sign in front of `AVG_GDP` in the code above.

```
gapminder_arragned_descending_chained =
  gapminder %>%
  select(country, year, pop, gdpPercap) %>%
  filter(year >= 1980) %>%
  mutate(GDP = gdpPercap * pop) %>%
  group_by(country) %>%
  summarise(AVG_GDP = mean(GDP)) %>%
  arrange(-AVG_GDP)

gapminder_arragned_descending_chained
```

```
## # A tibble: 142 x 2
##    country        AVG_GDP
##    <chr>            <dbl>
##  1 United States  9.20e12
##  2 Japan          3.28e12
##  3 China          2.96e12
##  4 Germany        2.20e12
##  5 United Kingdom 1.48e12
##  6 France         1.48e12
##  7 India          1.39e12
##  8 Italy          1.33e12
##  9 Brazil         1.27e12
## 10 Mexico         9.26e11
## # … with 132 more rows
```

3.3.7 *Simplifying Code with the Pipe Operator:* `%>%`

After helping learners see how each function works we can introduce the pipe operator (`%>%`). This helpful code chains together commands and passes the results of one function directly into the next function. This results in very logical data manipulation steps that are fairly easy to learn and makes the code easy to understand:

If students struggle to understand the role of the pipe, we often explain the operator as follows. A pipe takes the result of the previous function and 'pipes' it into the first argument of the next function. In that way, we can chain together our verbs to manipulate and gain insight into the data without creating multiple intermediate data frames. We find the pipe helpful in creating an analysis code that is easy to read and follow.

3.4 *Plotting and Exploratory Data Analysis*

From the preceding example, you can see one way for students to gain insight from data by manipulating the `tibble` with `dplyr` verbs.

Figure 5 Addins Menu

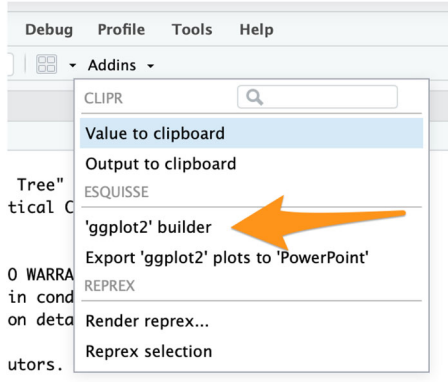


Figure 6 esquisse UI

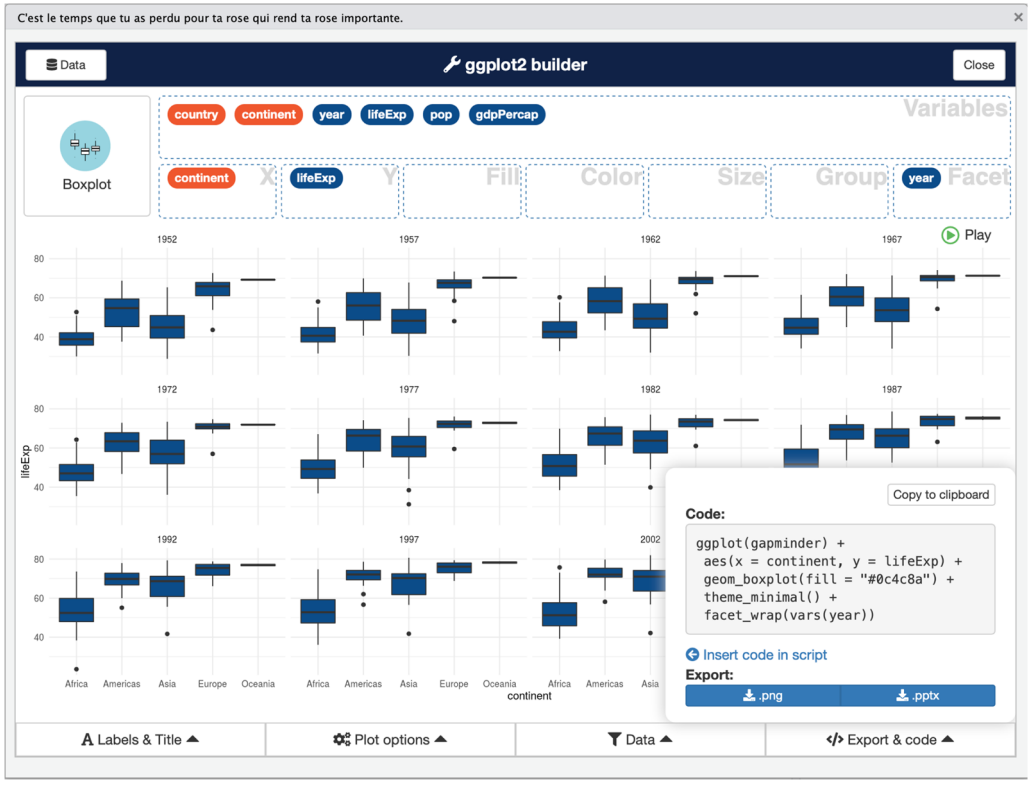
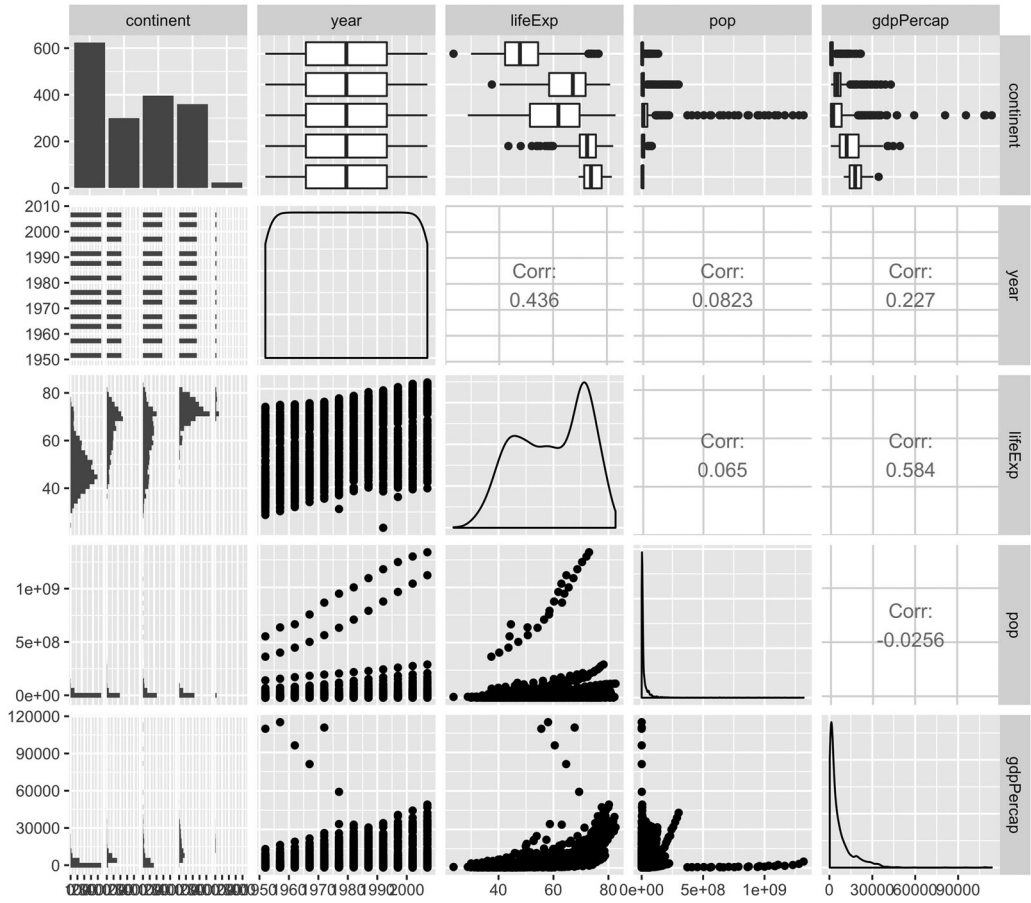


Figure 7 Example Pairs Plot



In addition, it is important for students to gain insight from data by viewing it graphically. Let us look briefly at some plotting basics and how to help students *quickly* create very insightful data visualisations.

3.4.1 `esquisse`

Typically, before building any models or doing other analyses, students benefit from learning basic EDA. One tool for getting students a quick win with learning data visualisation is to use `ggplot2` along with the helper package `esquisse` to give them a graphical user interface for basic `ggplot2` code (Meyer and Perrier 2019). `esquisse` is pronounced εs.kis and is the French word for an initial rough sketch. `esquisse` supports only a subset of the myriad features available in `ggplot2` but because it is a drag and drop GUI, it is very helpful in jump starting students to see how code can take data and turn them into tangible visualisations.

Since `esquisse` is a CRAN package, install it by running `install.packages("esquisse")`. After installation, `esquisse` shows up in the *Addins* menu of RStudio shown in Figure 5.

The `'ggplot2' builder` menu option opens the graphical interface for building `ggplot2` graphics using a helper UI. In Figure 6 we show how the `esquisse` interface appears with the `gapminder` data selected.

The strength of `esquisse` is that it produces the `ggplot2` code that allows the R learner to see how to build the syntax themselves in the future. Most importantly, it reduces the likelihood that students will get stuck when trying to make their first `ggplot2` figures.[7]

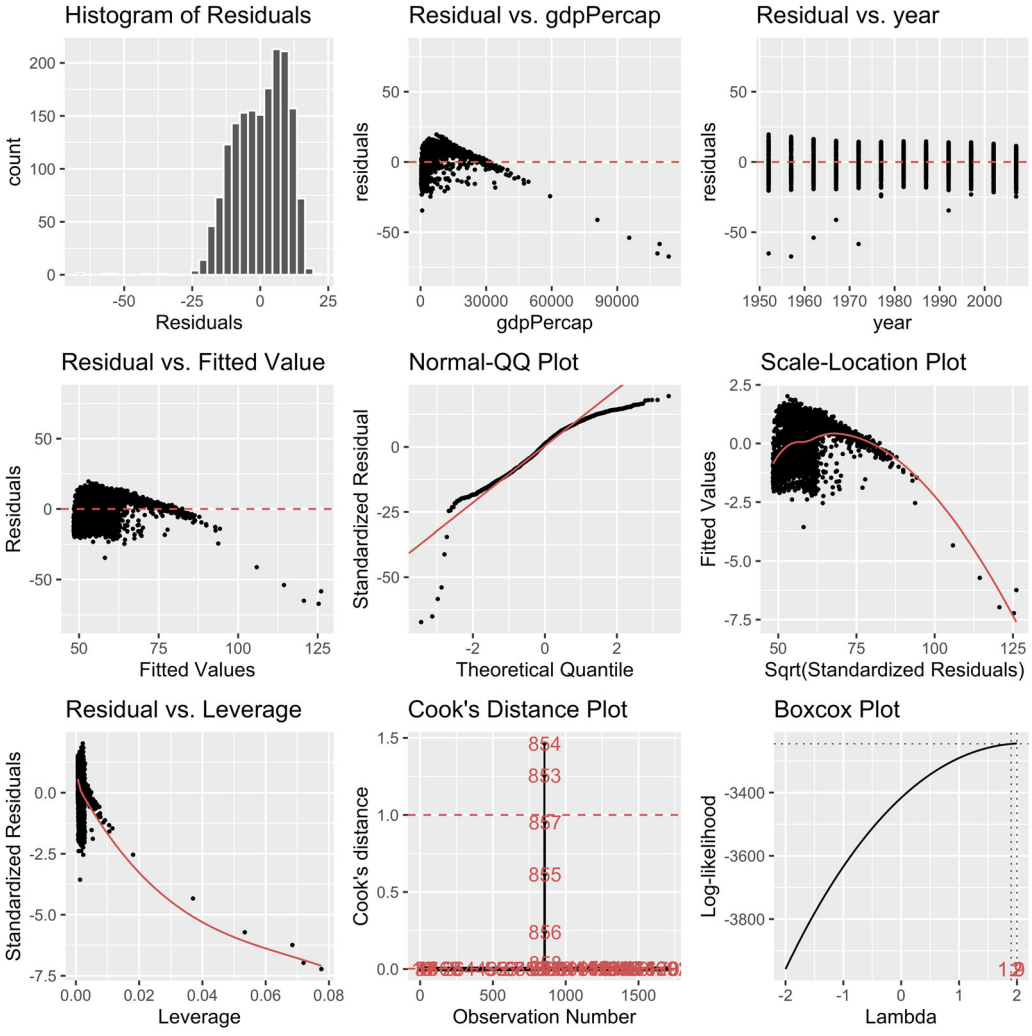
3.4.2 `ggpairs`

A supplemental technique to explore data is to look at a pairs plot. The `GGally` package provides a powerful tool with simple syntax that provides the user with clearly organised information about the data (Schloerke et al. 2018). As with other CRAN packages, the library should be installed with `install.packages("GGally")`.

A pairs plot does a good job of visualising relationships between continuous variables or character variables. In gapminder data, there are 142 different countries. For the pairs plot in Figure 7, we remove the `country` column then call the `ggpairs` function to generate the visualisation.

```
library(GGally)
gapminder %>%
select(-country) %>%
ggpairs()
```

Figure 8 Example Diagnostic Plot



There is much to glean from the pairs plot. We see that this tool provides pairwise plot and correlations between continuous variables and histograms and boxplots between discrete and continuous variables. While the x axis tends to be crowded, we can still see general trends and patterns from the pairs plot.

Briefly, we can see in Figure 7 that there is an increasing life expectancy and population as the year increases as well as several countries that separate themselves from the pack. An easily creatable pairs plot like this can springboard students into further analysis, an example of which we will highlight below.

3.5 Regression

The first statistical modelling method that most students are taught is ordinary least squares linear regression. In the base stats package and with one additional library, R offers easy to execute and understandable tools to implement everything we would expect a student to learn in linear regression.

To highlight these tools, let us explore the linear relationship between several variables and life expectancy (`lifeExp`) in the Gapminder data.

3.5.1 Simple Linear Regression

To implement a linear regression, we need to specify two arguments in the `lm` (linear model) function: the formula and the data. As can be seen in the example below, our formula is in the format `y ~ x`—pronounced y by x. As expected, the `data` are equal to `gapminder`.

```
model1 = lm(formula = lifeExp ~ gdpPercap, data = gapminder)
```

```
summary(model1)
```

```
##
## Call:
## lm(formula = lifeExp ~ gdpPercap, data = gapminder)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -82.75  -7.76   2.18   8.23  18.43
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 5.40e+01   3.15e-01   171.3   <2e-16 ***
## gdpPercap   7.65e-04   2.58e-05    29.7   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10 on 1702 degrees of freedom
## Multiple R-squared:  0.341,  Adjusted R-squared:  0.34
## F-statistic:  880 on 1 and 1702 DF,  p-value: <2e-16
```

R executes this command and saves it as `model1`. To retrieve our simple linear regression model, we place `model1` in the summary command below.

From the output, the student sees pertinent information about the model including the equation, a small summary of the residuals, regression coefficients, p-values, and familiar model evaluation statistics such as R^2 and Adjusted R^2.

Adjusting the linear model is simple. Should we desire to add another factor to

```
model2 = lm(formula = lifeExp ~ gdpPercap + year, data = gapminder)

summary(model2)

##
## Call:
## lm(formula = lifeExp ~ gdpPercap + year, data = gapminder)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -67.26  -6.95   1.22   7.76  19.55
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -4.18e+02   2.76e+01   -15.2   <2e-16 ***
## gdpPercap    6.70e-04   2.45e-05    27.4   <2e-16 ***
## year         2.39e-01   1.40e-02    17.1   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.7 on 1701 degrees of freedom
## Multiple R-squared:  0.437,  Adjusted R-squared:  0.437
## F-statistic:  661 on 2 and 1701 DF,  p-value: <2e-16
```

the model, we can do it simply by adding it to the formula like the example below.

R's linear model function is flexible enough to easily add interaction terms. To add an interaction term between `gdpPercap` and `year`, we add a colon between the independent variables.

```
model3 = lm(formula = lifeExp ~ gdpPercap + year + gdpPercap:year,
            data = gapminder)

summary(model3)

##
## Call:
## lm(formula = lifeExp ~ gdpPercap + year + gdpPercap:year, data = gapminder
)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -54.23  -7.31   1.00   7.95  19.78
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)    -3.53e+02   3.27e+01  -10.81  < 2e-16 ***
## gdpPercap      -8.75e-03   2.55e-03   -3.44  0.00060 ***
## year            2.06e-01   1.65e-02   12.46  < 2e-16 ***
## gdpPercap:year  4.75e-06   1.28e-06    3.70  0.00022 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.7 on 1700 degrees of freedom
## Multiple R-squared:  0.442,  Adjusted R-squared:  0.441
## F-statistic:  449 on 3 and 1700 DF,  p-value: <2e-16
```

3.5.2 *Linear Regression Modelling Assumptions*

One of the staples of teaching linear regression is helping students determine if their model meets the assumptions necessary for a linear model to be valid. For the purpose of

illustration we will look at the following assumptions:

- independent observations;
- normal errors;
- constant variance of errors (homoscedasticity);
- linear relationships.

Let us look briefly at how we help students think about each of these assumptions using R code where helpful.

Independent Observations

To determine independence, we must know how the data were collected. There are likely independence issues as the data for each country in the Gapminder collection are likely influenced by each other. We will acknowledge this and proceed with the other assumptions.

Other Assumptions

To verify the other assumptions, we need to create two plots: the residual plots and a qqplot.

To create the plots, we will rely on the `lindia` (Lee and Ventura 2017) package that makes diagnostic plots easy. If you do not have it installed already, execute `install.packages("lindia")`. Then pass the results of a model into the `gg_diagnose` function as shown in Figure 8.

```
library(lindia)
model2%>%
gg_diagnose(plot.all=TRUE,
boxcox=TRUE)
```

The `gg_diagnose` command is a one-stop shop for all diagnostic plots to evaluate the OLS assumptions.

Normal Errors

From the plots, we see that our normality is slightly skewed (first graph in the second row of Figure 8) in the positive direction and `gdpPercap` appears to be the culprit. The `qqplot` also supports this conclusion as the observed standardised residuals are more extreme than what we would expect if the residuals followed the ideal theoretical z distribution.

Constant Variance

Variance of the residuals (the right two graphs on the top row in Figure 8) appears to remain constant at all levels of each x variable.

Linear Relationships

The linearity assumption, however, seems to be violated at the `gdpPercap` variable and we also see evidence of this in the residuals versus fitted plot in Figure 8. In both plots, instead of seeing a patternless array of errors, we see an arching trend indicating a

```
modeladj = lm(formula = lifeExp ~ log(gdpPercap) + year, data = gapminder)

summary(modeladj)

##
## Call:
## lm(formula = lifeExp ~ log(gdpPercap) + year, data = gapminder)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -27.229  -3.845   0.607   4.774  17.864
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)    -391.05135   19.41829   -20.1   <2e-16 ***
## log(gdpPercap)    7.77032    0.13808    56.3   <2e-16 ***
## year              0.19557    0.00993    19.7   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.9 on 1701 degrees of freedom
## Multiple R-squared:  0.717,  Adjusted R-squared:  0.717
## F-statistic: 2.15e+03 on 2 and 1701 DF,  p-value: <2e-16
```

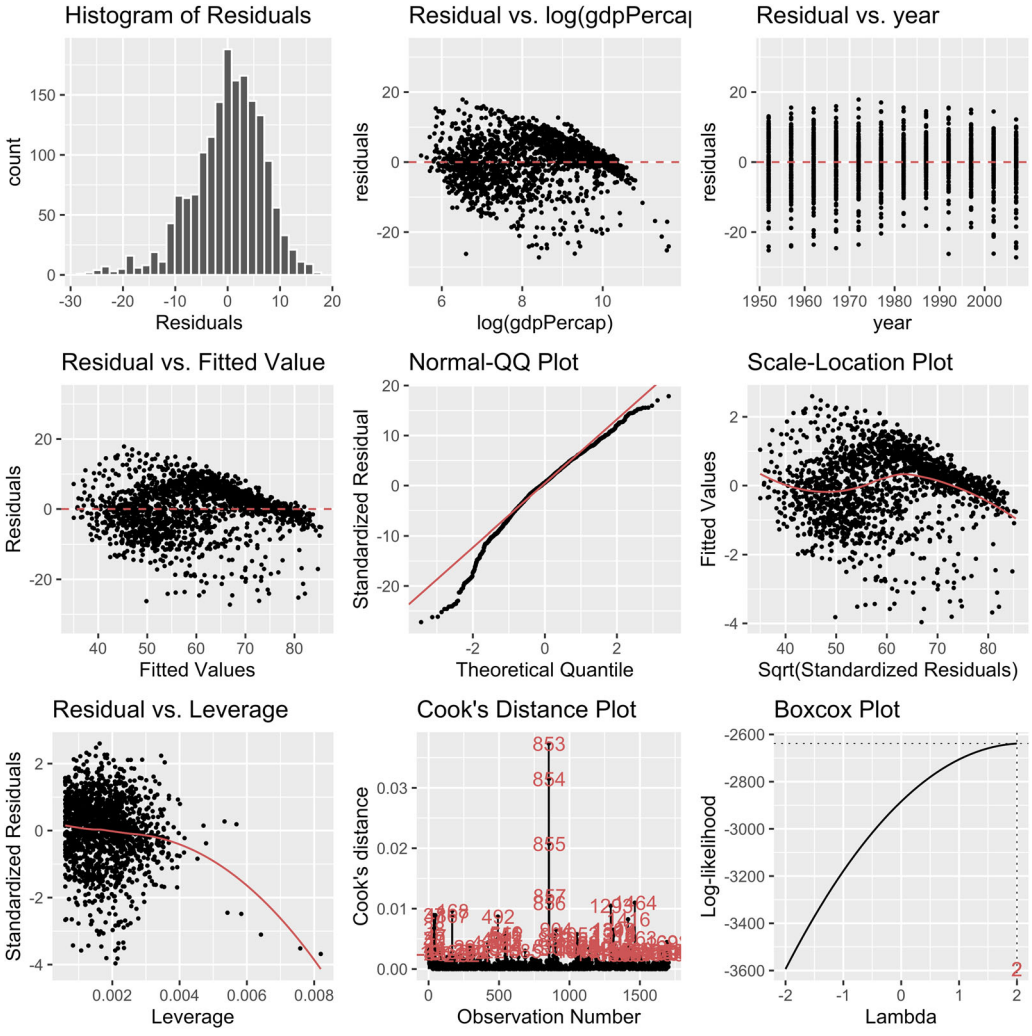
non-linear relationship between at least one of the x variables and `lifeExp`. The residuals start negative, peak in the positive and then taper off negative again as each value on the x axis increases.

3.5.3 *Transformations and Residual Revaluation*

From our previous critique, and after some instruction on how to interpret the results, students should have all the information available to them to make decisions about how to improve their model. To improve our example analysis, we will apply a transformation on `gdpPercap` to improve linearity and normality of errors. A transformation on the *y* variable is not necessary as we have no issues with constant variance and the Box Cox plot shows no value of lambda that maximises the log-likelihood of our model.

Since our coefficients are still significant, we will take a look at the diagnostic plots to see if our assumptions are more plausible. R makes it easy to repeat the same plots on the new model. Figure 9 shows the diagnostics after we perform the transform.

Figure 9 Improved Model Diagnostic Plot



```
  modeladj %>%
  gg_diagnose(plot.all=TRUE,
boxcox=TRUE)
```

While there is more work to be done to validate this model, we can see that our transformation has improved the normality of errors and has improved the linearity of `gdppercap`.

4. Conclusion

R, along with supporting CRAN packages, can be an excellent platform from which to teach analysis and modelling to students. The RStudio.cloud online environment is a solid hosted platform that helps reduce the friction of getting students started with R, and add-on packages from CRAN like `tidyverse`, `lindia`, `GGally` and `esquisse` can be combined together in a teaching environment to help students get tangible results quickly. We have found that students are willing to learn a considerable amount of R if we can make the on-ramp to the basics quick and easy. We hope these tools help you and your students get up and running quickly with using R for data analysis and modelling. We wholeheartedly believe Adam Smith will be pleased to see the ease of the masters and the ease of the students aligned.

Endnotes

1. https://cran.r-project.org/web/views/

2. See https://www.rstudio.com/pricing/academic-pricing/ for more info.

3. http://rstudio.cloud

4. https://rc2e.com/gettingstarted#recipe-id001

5. Absolute file paths are the entire file path starting from the root directory while relative files paths are relative from a starting directory.

6. More information on each can be found at https://tidyverse.tidyverse.org.

7. For more information about `esquisse` see the project CRAN website: https://cran.r-project.org/web/packages/esquisse/readme/README.html

References

Cran 2019, *The Comprehensive R Archive Network*, viewed January 2019, <https://cran.r-project.org/>.

Gapminder 2019, *Data. Gapminder*, viewed January 2019, <https://www.gapminder.org/data/>.

Lee, Y. Y. and Ventura, S. 2017, *Lindia: Automated Linear Regression Diagnostic*, viewed January 2019, <https://CRAN.R-project.org/package=lindia>.

Meyer, F. and Perrier, V. 2019, *Esquisse: Explore and Visualize Your Data Interactively*, viewed January 2019, <https://CRAN.R-project.org/package=esquisse>.

Quinn, M., McNamara, A., Arino de la Rubia, E., Zhu, H. and Ellis, S. 2019, *Skimr: Compact and Flexible Summaries of Data*, viewed January 2019, <https://CRAN.R-project.org/package=skimr>.

RStudio 2019, *Tidyverse Packages*, viewed January 2019, <https://www.tidyverse.org/packages/>.

Schloerke, B., Crowley, J., Cook, D., Briatte, F., Marbach, M., Thoen, E., Elberg, A. and Larmarange, J. 2018, *GGally: Extension to 'ggplot2'*, viewed January 2019, <https://CRAN.R-project.org/package=GGally>.

Smith, A. and Krueger, A. B. 2003, *The Wealth of Nations*, Bantam, New York.

Stack 2019, *Browse Queries. Stack Exchange Data Explorer*, viewed January 2019, <https://data.stackexchange.com/stackoverflow/>.

Teetor, P. and Long, J. 2019, *R Cookbook*, 2nd edn, viewed January 2019, <http://www.rc2e.com/>.

Wickham, H. 2015, *Tidyverse 1.0.0. RStudio Blog*, viewed January 2019, <https://blog.rstudio.com/2016/09/15/tidyverse-1-0-0/>.

Wickham, H. 2019, Easily Install and Load the 'Tidyverse', viewed January 2019, <https://tidyverse.tidyverse.org/>.