

# Instructions for R

## 1. Introduction

R is an open-source programming language that is used extensively for statistical computing and data visualizations. The program was developed by Ross Ihaka and Robert Gentleman in the early 1990s, and its usage has grown rapidly since then. The Comprehensive R Archive Network (CRAN) stores R's executable files, source code, and documentation. The latest version of R is available for free via CRAN at the following website: <https://cran.r-project.org/>.

Most users interact with R using an “integrated development environment” (IDE) called RStudio, which makes programming in R much simpler. The coding examples used below will utilize RStudio, which can be downloaded for free at: <https://posit.co/>. *From this point forward, when we talk about ‘launching’ or ‘running’ R, we will be doing so via RStudio.*

One key feature of R is that, because it is open source and has such a large community of users, there have been literally thousands of “packages” developed, which are freely available, that make carrying out computations in R much easier. Most of these packages are available from the CRAN website. There is some debate as to whether R should be taught to new users using only “base R” programming that is built into the R programming language, or whether those new to R should be exposed to packages as well. My view is that using packages is more efficient and will make it easier for new users to be able to perform tasks in R more quickly. As such, we will make significant usage of something called the “tidyverse” which is a collection of packages

that facilitate various tasks in R. The tidyverse is very popular among R users. Details about the tidyverse package and its components are available at: <https://www.tidyverse.org/>. Also available at this website are ‘chest sheets’ that summarize some the basic functions for the packages contained in the tidyverse package. In addition to the tidyverse, we will make use of some more specific packages that are designed for regression methodologies covered in *Regression Basics*.

It is also worthwhile noting that, unlike SPSS<sup>®</sup> and Stata<sup>®</sup> where drop-down menus can be used to carry out most computations, R requires scripting code to accomplish such tasks. Thus, we will be creating R script files that contain a list of commands that we want R to carry out. Further, unlike SPSS and Stata, R uses object-oriented programming. Thus, everything in R is an object, and anything that does something using objects is a function. This distinction will become clearer as we begin coding in R.

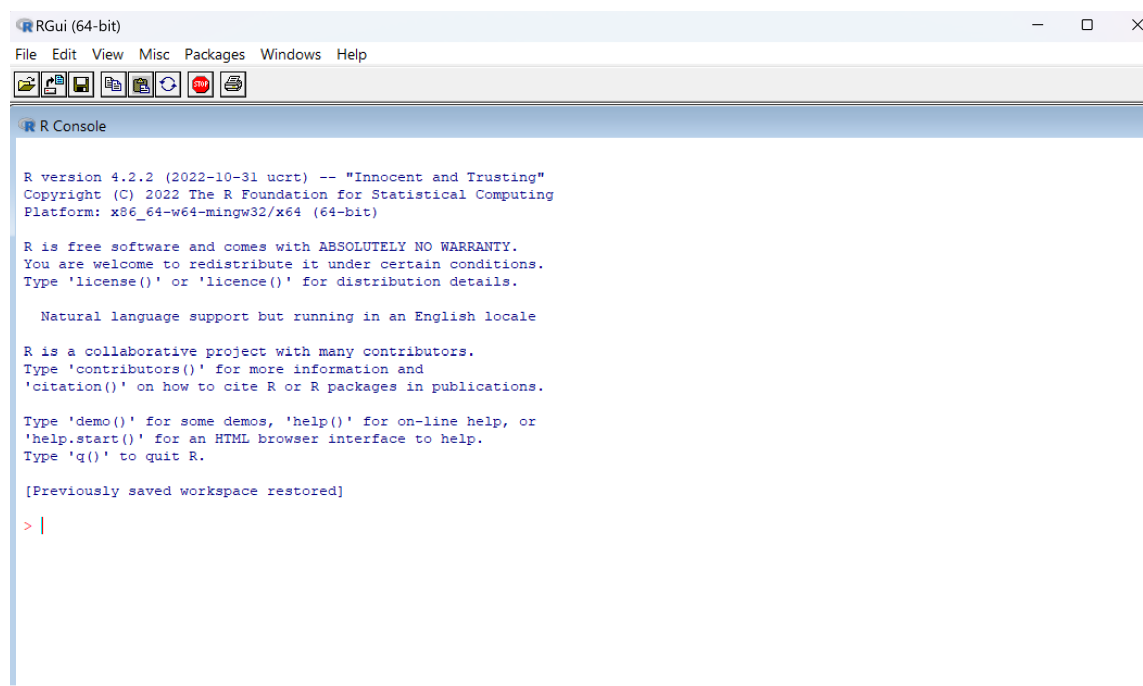
Lastly, as is the case with most statistical programs, there are often multiple ways of coding in R (as well as multiple packages available) to accomplish the same tasks. The coding examples below reflect my preferred approach, and are typically not the only way to get things done in R.

## 2. Appearance


[Note: The screen shots provided below are from a Windows version of R and RStudio.

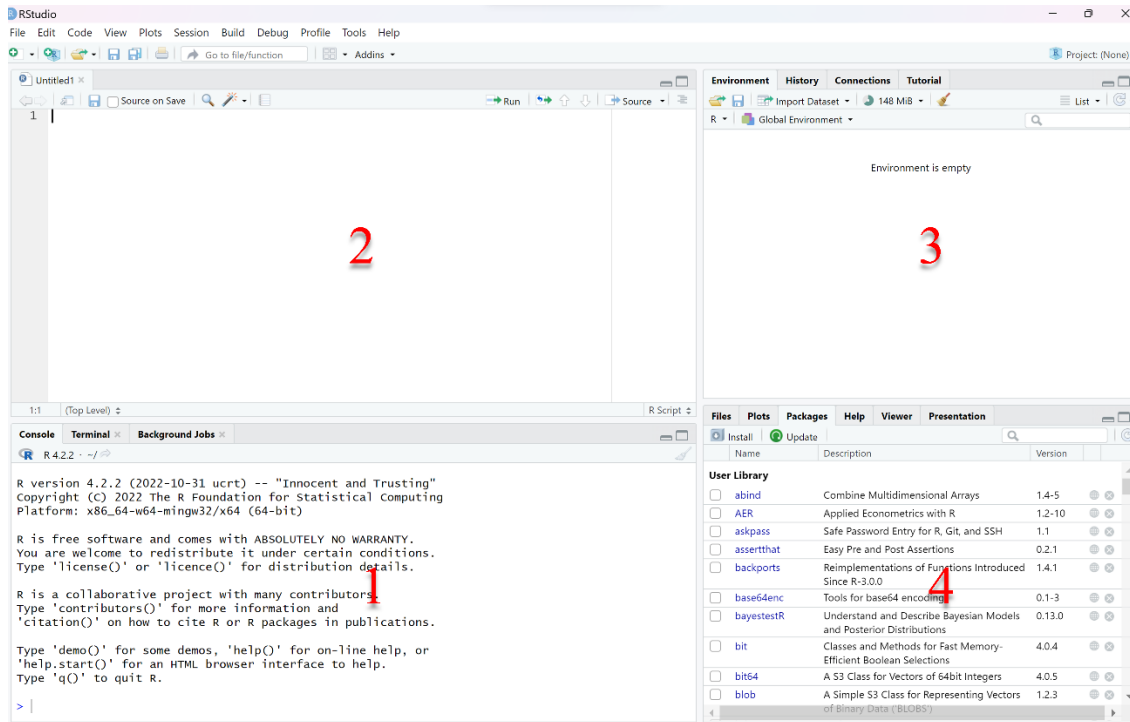
Versions for other operating systems (e.g., for MacOS) may appear slightly different.]

Once R is downloaded and launched, we see an opening screen like that shown below:



Users can begin to code by typing commands directly into R at the red **>** prompt. However, as noted above, most users use the RStudio IDE when coding for R. Launching RStudio gives us the following opening screen,<sup>1</sup>

<sup>1</sup> Note that when you first launch a new session in RStudio, the 'Console' window may be covering up the 'Source' window. Clicking the Console window's resize button, , will resize the Console window and expose the 'Source' window.



As indicated, there are 4 areas shown on the opening screen (we will focus on the primary tabs). (Note that the size of these areas can be adjusted by clicking and dragging their borders within the opening window.)

Area **1**: This is the console. Here you can type commands at the blue **>** prompt. Hitting the ‘Enter’ key will execute a command. The output of a command is also shown in this window. At the startup of R, you will typically see a message here noting the version of R being used.

Area **2**: This is the source window, where users type a list of commands known as a ‘script’. The script can be sent to the console for execution as a whole by highlighting it and then clicking the ‘Run’ button on the ribbon at the top of this window. Alternatively, parts of a script can be run by highlighting the specific portion and hitting the ‘Run’ button.

Area **3**: This is the environment window. Here you will find the objects you have imported or created within R (e.g., data sets, data frames, regression results, etc.). The history tab in this window shows you a list of all the commands you have executed during an R coding session.

Area **4**: The final window has multiple tabs. The ‘Packages’ tab shows you all the packages (described below) that you have downloaded into R, and which can be loaded and used during a coding session. The ‘Plots’ tab will show plots that have been created during an R session. The ‘Help’ tab is useful for finding help with packages or various R functions. The user can use the `help()` function in the console to find help with something. For example, you can type: `help(install.packages)` in the console. After hitting ‘Enter’, the information will be displayed in the ‘Help’ tab. (Alternatively, we could type `?install.packages` into the console to get the same information.)

To see the current working directory that R is using, the user can type the following into the console at the `>` prompt,

```
getwd()
```

Hitting ‘Enter’ we are shown the address of the current working directory in the console. If the user wants to change the working directory, we can type:

```
setwd("C://file/path")
```

where “C://file/path” is the address you wish to set (double quotation marks should be included in the command). Note that, unlike directory commands in Windows, R uses ‘forward slashes’ (/) in the directory path. As an example, typing the command at the console prompt:

```
setwd(“C:/Users/leoka/project1”)
```

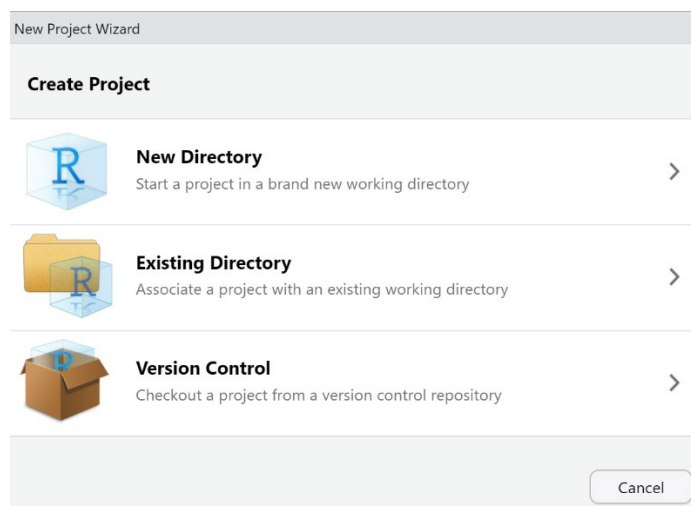
hitting ‘Enter’ would change the working directory to C:/Users/leoka/project1, (assuming this location exists already).

### 3. Creating Projects

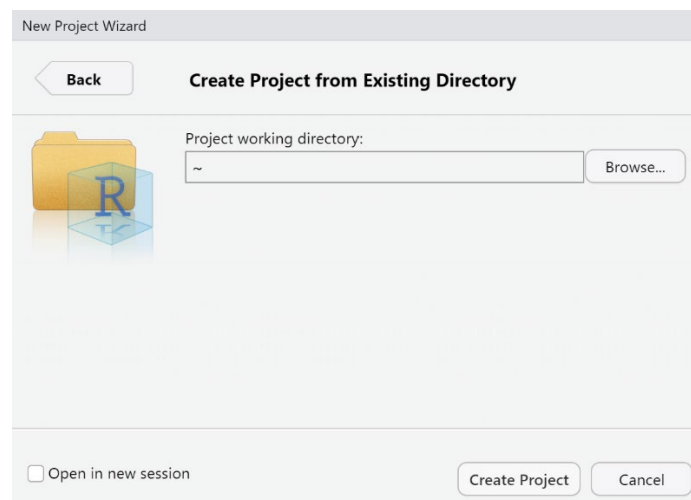
One of the advantages with R is that it can work with multiple data sets at the same time, with potentially each data set being in different locations on your computer. While useful, it can be a bit tricky to keep the directories straight and typing errors when specifying directory locations can lead to coding errors. For this reason, it is advisable to set up a ‘project’ when using R. In this project folder we can store our raw data files, scripts, etc., and we can save any output created during an R session. Once an R project is created and saved, we can simply click on an R script file (designated with a ‘.R’ file extension) or project file (designated with a ‘.Rproj’ file extension) in the project folder. This will launch RStudio and the default directory will be set to the location where the script file is saved. For example, suppose we wish to explore the factors affecting NBA player salaries using the nba2021\_22.csv data set (available at the *Regression Basics* website). We can start by creating a folder called nba\_sal, say, on the Desktop (or any other desired location). Next, we can place the nba2021\_22.csv file in this newly created folder. We can now launch RStudio and choose ‘File’, on the main menu, then ‘New Project’,



This will launch the ‘New Project Wizard’,



Choosing ‘Existing Directory’ we get,



Using the ‘Browse’ button, we can navigate to the location of the `nba_sal` folder, and then click on ‘Create Project’. This will create a project file in the `nba_sal` folder called, `nba_sal.Rproj`.

From this point forward the working directory will be set to this `nba_sal` folder location and any files created with R will be stored in this folder as well (unless changed by the user).

## 4. Installing Packages and Using Libraries

As noted earlier, there are literally thousands of “packages” that have been (and continue to be) created by the community of R users. These packages contain libraries of functions that make carrying out certain R tasks much easier and more efficient than working with base R only.

Packages are installed in R by using the `install.packages("name")` function, where "name" is the name of the package to be installed, (with the quotation marks included). When we are going to use a library in an R session, they need to be loaded using the `library(name)` function, (no quotes in this case). One of the most important packages we will use is the "tidyverse" which contains within it a handful of packages we will use extensively later. Two key packages included in the tidyverse are "dplyr" and "ggplot2". The former is very useful for data manipulation (e.g., filtering data, transforming variables, etc.), whereas the latter is useful for creating data visualizations. To install the tidyverse package we can use the following code,

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

Typing the above function into the console at the blue `>` prompt, and then hitting the ‘Enter’ key will start the installation. Note that once we have installed a package, it does not need to be installed again. To see a list of installed packages, we can launch R, and then click the ‘Packages’ tab in window 4 noted earlier.



To use a library from an installed package we can use the `library(name)` command (without quotes) where `name` is the name of the library we wish to use. For example, to load the `tidyverse` library, we would type,

```
library(tidyverse)
```

Library commands usually appear near the beginning of an R script file. Further, we do not need to load the entire `tidyverse` package. If we only want to work with, say, the `dplyr` package we can use,

```
library(dplyr)
```

In the examples below we will frequently be working with multiple `tidyverse` packages thus, to simplify the coding, we will usually load the entire `tidyverse` library.

## 5. Starting and Saving Script Files

As noted earlier, most R sessions will involve writing script files which contain a list of commands, and then running the script file. When a new R session is launched there will be a blank script file in window 2 of the opening screen (see the earlier picture) with a default name of 'Untitled1'. If we click on 'File' on the main menu, then select 'Save As...', we can choose a name for the script file and save it in the working directory. Once the script file is saved, we can begin to add commands to it, saving periodically along the way. Saving script files is very important for two key reasons. First, it allows us to trace our steps in an empirical project and be able to reproduce our results, (reproducibility has become a very important aspect of doing empirical research). Second, if we want to make some changes to our empirical work, (e.g., using a log-transformation of a variable in a regression model) we only need to edit our script

file and then re-run it. This saves a lot of time by not having to re-type the whole list of commands.<sup>2</sup>

It is *highly* recommended that users include comments in their script file that describe what you are doing, and why, as you add commands. Comments in R are identified by the hashtag symbol, #. Below is an example of creating a small data set (or ‘data frame’) by creating a short script file and then running it in R,

```
#Creating a data frame by-hand
name <- c("Alan", "Bob", "Cathy", "Denise")
age <- c(24, 20, 31, 46)
dframe <- data.frame(name, age)
print(dframe)
```

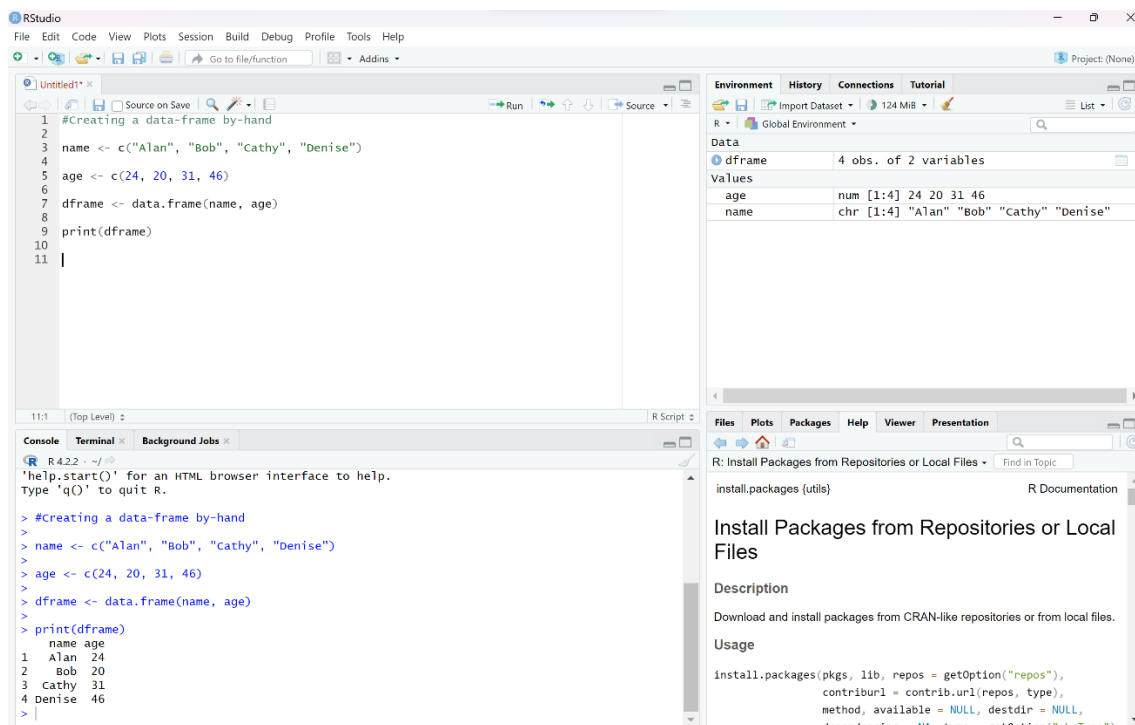
The script above illustrates several coding aspects for R. The ‘#’ in the first line identifies this first line as a comment that will be printed as part of the output generated by R. The next two lines are creating two new objects, one called name, the other called age. In these two lines we see the symbol, <-. This symbol is an assignment operator that tells R to assign to the object name the results of the following function.<sup>3</sup> The part following the assignment operator, c("Alan", "Bob", "Cathy", "Denise"), is a function that creates a vector with the elements being the strings "Alan", "Bob", "Cathy", "Denise". The double quotes (" ") tell R that these elements

---

<sup>2</sup> Note that it is common for users to run short, quick commands by typing directly into the console at the command prompt.

<sup>3</sup> The equal sign symbol, =, can also be used as the assignment operator, though most R programmers tend to use the <- symbol.

are strings (i.e., text). The next line, `age <- c(24, 20, 31, 46)`, is similar and assigns to the object `age` to the results of the function containing the values of 24, 20, 31 and 46. The fourth line assigns to the object `dframe` (the name we've given to our data frame we are creating) the results of the `data.frame()` function that takes the two objects, `name` and `age`, and combines them to create a data frame. The last line, `print(dframe)`, prints the `dframe` object. Pasting the above script into R, highlighting all the script, and then clicking the 'Run' button (try it!) we get,



Notice that RStudio's script editor (the top, left window) shows our script with color coding, (e.g., green for comments, blue for numeric values). We also see in the console (the lower, left window) the results of our script file's execution. Lastly, we can see in the 'Environment' window (top, right) that three objects are now in memory: one data frame (`dframe`) and two objects (`name` and `age`).

Now that we have a script file that carries out our commands, we can save it by clicking on ‘File’ on the main menu, then choosing ‘Save as...’, and then give the file a name and choose a location where it will be saved. We can then later double click on the R script file and R will open and display the saved script which can be re-run to produce the same results.

## 6. Getting Data into R

As demonstrated by the last example, we can build a data set manually by creating vectors of data and then combining them into a data frame using the `data.frame()` function. This approach, however, is seldom used when working with large data sets. In most cases (including all the examples in *Regression Basics*) we will be importing data sets into R. R can import all sorts of data types, (including data from popular statistical programs such as SPSS®, Stata®, and SAS®).<sup>4</sup> In the examples below, we will be importing .csv files which are plain text files with values separated by commas. This can easily be done by using the `read.csv()` function which is part of base R. However, as noted earlier, we will be making use of some packages that will make our coding in R easier and more efficient. Particularly, we will be using the “tidyverse” package discussed earlier. The tidyverse contains a variety of packages within it, including one called “readr” which makes importing .csv files very easy. Assuming the tidyverse package has already been installed, we will load its library at the beginning of a script file.<sup>5</sup> The script file we will be building will use the `nba2021_22.csv` file that was introduced earlier when we created our `nba_sal` project folder. Opening this project in R (you can simply double click the

---

<sup>4</sup> The ‘haven’ package, available from CRAN makes it easy to import these file types (see: <https://cran.r-project.org/web/packages/haven/index.html>).

<sup>5</sup> If not, you can use the `install.packages("tidyverse")` function to install it. Further, once installed, we could load just the library for “readr” to help us import the .csv file. However, since we will be using other packages in the tidyverse package, we will load the whole tidyverse library (which includes readr).

nba\_sal.Rproj file), and then starting a new R Script file from the main ‘File’ menu, we can type (or copy and paste) the following commands into the source window,

```
#Analysis of NBA Salary Determinants

#Loading the tidyverse library
library(tidyverse)

#Reading in the nba2021_22.csv data set
nba_data <- read_csv("nba2021_22.csv")

#Computing summary statistics
summary(nba_data)
```

The above script file first loads the tidyverse library, then assigns to the object nba\_data the contents of the nba2021\_22.csv file by using the read\_csv() function, (with quotes included).<sup>6</sup> Finally, we use the summary() function to compute summary statistics for the variables in the nba\_data object just created. Before running this script, we can save it with an appropriate name, such as nba by using the ‘Save As...’ option from the main ‘File’ menu. This will save the script file in our default nba\_sal project folder (it will appear as an nba.R script file). Highlighting all the script and then clicking the ‘Run’ button, we get the following results,<sup>7</sup>

---

<sup>6</sup> Note that we do not need to include a path to the nba\_2021\_22.csv data set here because it is in our R Project folder, which is the current default location. If the data set is located elsewhere, a path would be needed.

<sup>7</sup> The reader is encouraged to participate by typing (or copying and pasting) this, and the following commands into R and produce results like those that will be presented.

```

1 #Analysis of NBA Salary Determinants
2
3 #Loading the tidyverse library
4 library(tidyverse)
5
6 #Reading in the nba2021_22.csv data set
7 nba_data <- read_csv("nba2021_22.csv")
8
9 #Computing summary statistics
10 summary(nba_data)
11
12

```

```

> #Computing summary statistics
> summary(nba_data)
      player      position      age      salary
Length:311 Length:311      Min.   :20.00      Min.   : 0.9543
Class :character Class :character 1st Qu.:23.00 1st Qu.: 2.5044
Mode  :character Mode  :character Median :26.00 Median : 5.0000
      Mean :26.73 Mean : 9.8487
      3rd Qu.:30.00 3rd Qu.:13.0182
      Max. :41.00 Max. :44.0664

      team      weight      height      games      minspg
Length:311      Min.   :164.0      Min.   :72.00      Min.   : 3.0      Min.   : 3.50
Class :character 1st Qu.:198.0 1st Qu.:76.00 1st Qu.: 96.5 1st Qu.:17.61
Mode  :character Median :214.0 Median :78.00 Median :253.0 Median :23.20
      Mean :214.5 Mean :77.98 Mean :329.9 Mean :22.61
      3rd Qu.:230.0 3rd Qu.:80.00 3rd Qu.:526.5 3rd Qu.:28.70
      Max. :290.0 Max. :87.00 Max. :1310.0 Max. :37.99

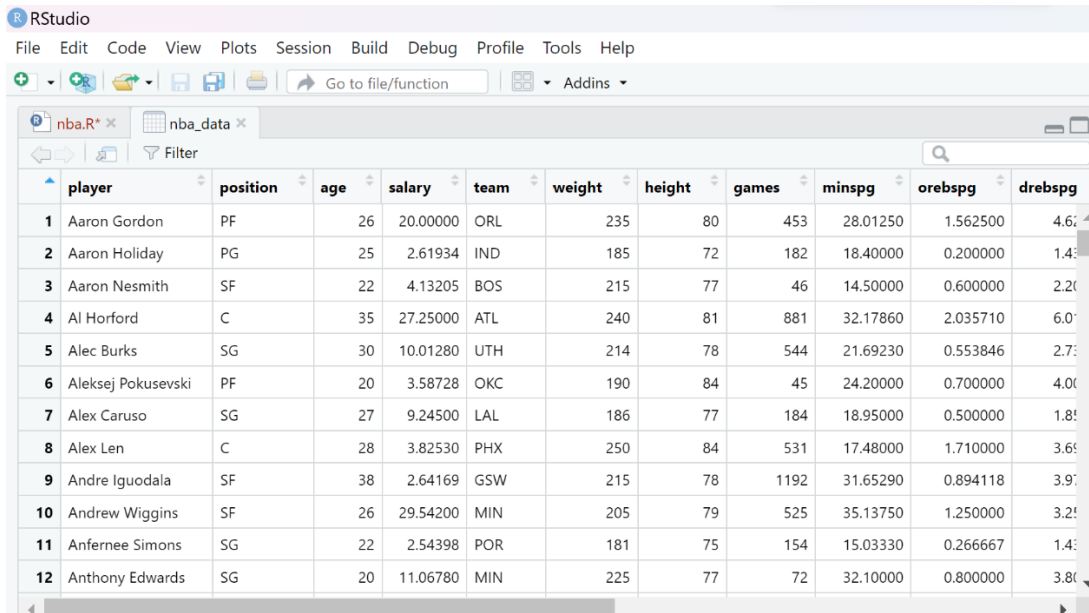
      orebsp      drebsp      trebsp      astsp      stlsp
Min.   :0.0000 Min.   :0.0000 Min.   :0.0000 Min.   :0.0000 Min.   :0.0000

```

Environment: nba\_data 311 obs. of 19 variables

We can see in the console window the summary statistics of the variables in the object called `nba_data`. We see that for the first two columns of data, which are the NBA player names and their positions, the summary statistics tell us that these columns are characters and, as such, no statistics are reported. For the variable `age`, we see the minimum value (20), first quartile value (23), the median value (26), the mean value (26.73), etc., and similar results for each of the other numeric variables in the data set. We also see in the environment window that we have a data frame with 311 observations on 19 variables.

If we want to view the data frame just created, we can click on the data frame name in the environment window, or equivalently type `view(nba_data)` into the console and hit ‘Enter’. This will bring up a ‘spreadsheet’ type tab in the source window,



The screenshot shows the RStudio interface with a data table named 'nba\_data' loaded. The table has 12 columns: player, position, age, salary, team, weight, height, games, minspg, orebsp, and drebsp. The first 12 rows of data are visible, listing players from Aaron Gordon to Anthony Edwards.

	player	position	age	salary	team	weight	height	games	minspg	orebsp	drebsp
1	Aaron Gordon	PF	26	20.00000	ORL	235	80	453	28.01250	1.562500	4.6
2	Aaron Holiday	PG	25	2.61934	IND	185	72	182	18.40000	0.200000	1.4
3	Aaron Nesmith	SF	22	4.13205	BOS	215	77	46	14.50000	0.600000	2.2
4	Al Horford	C	35	27.25000	ATL	240	81	881	32.17860	2.035710	6.0
5	Alec Burks	SG	30	10.01280	UTH	214	78	544	21.69230	0.553846	2.7
6	Aleksej Pokusevski	PF	20	3.58728	OKC	190	84	45	24.20000	0.700000	4.0
7	Alex Caruso	SG	27	9.24500	LAL	186	77	184	18.95000	0.500000	1.8
8	Alex Len	C	28	3.82530	PHX	250	84	531	17.48000	1.710000	3.6
9	Andre Iguodala	SF	38	2.64169	GSW	215	78	1192	31.65290	0.894118	3.9
10	Andrew Wiggins	SF	26	29.54200	MIN	205	79	525	35.13750	1.250000	3.2
11	Anfernee Simons	SG	22	2.54398	POR	181	75	154	15.03330	0.266667	1.4
12	Anthony Edwards	SG	20	11.06780	MIN	225	77	72	32.10000	0.800000	3.8

### *Renaming a Variable*

The variable `salary` is a player's salary in millions of dollars. If we wanted to be more specific with the name, we could rename this variable `sal_m` in order to emphasize that it is in millions of dollars. We can rename the column within R by using the `rename()` function, (as part of the `dplyr` package included in the `tidyverse` package). The basic syntax for the `rename()` function is: `rename("new_column_name" = "old_column_name")`, with quotes included. Inserting some additional code into our current script file will accomplish this task,

```
#Analysis of NBA Salary Determinants

#Loading the tidyverse library
library(tidyverse)

#Reading in the nba2021_22.csv data set
nba_data <- read_csv("nba2021_22.csv")

#Computing summary statistics
summary(nba_data)

#Renaming salary to sal_m
nba_data <- nba_data %>%
  rename("sal_m" = "salary")
```

The added code demonstrates an operator that is included in the tidyverse package named the “pipe” operator, `%>%`. This operator can be read as “and then”. Thus, the last two lines of code could be read as: assign to the object `data_nba` the contents of the existing object, `data_nba`, and then, rename the column `salary` to `sal_m`. Notice that we are essentially replacing the previous data frame with a new data frame that has a column that was renamed.

## 7. Transforming and Dropping Variables

Once we have imported data into R, we may want to transform some variables. For example, we may want to compute the natural log of a variable or compute the squared value of a variable.

This can be done in base R, or we can use the `mutate()` function which is part of the `dplyr` package, (which, in turn, is part of the tidyverse package). Let’s have a look at both approaches.

Using base R, we can use the following code to create the natural log of our salary column,

```
sal_m,
```



```
#Computing the natural log of sal_m and adding it to our data frame
nba_data$lnsal_m <- log(nba_data$sal_m)
```

We could read the above code as, assign to the object `nba_data$lnsal_m` the results of the function `log(nba_data$sal_m)`. The `$` operator in R refers to a particular part of an object. In this case, it refers to a column in the data frame `nba_data`. Thus, the above code is essentially creating a new column in the data frame called `lnsal_m`, which is equal to the natural log (`log`) of the column `sal_m` in the existing data frame called `nba_data`.

Alternatively, we can use the `mutate()` function in the following way,

```
#Computing the natural log of sal_m and adding it to our data frame
nba_data <- mutate(nba_data, lnsal_m = log(sal_m))
```

The basic form for the `mutate()` function is,

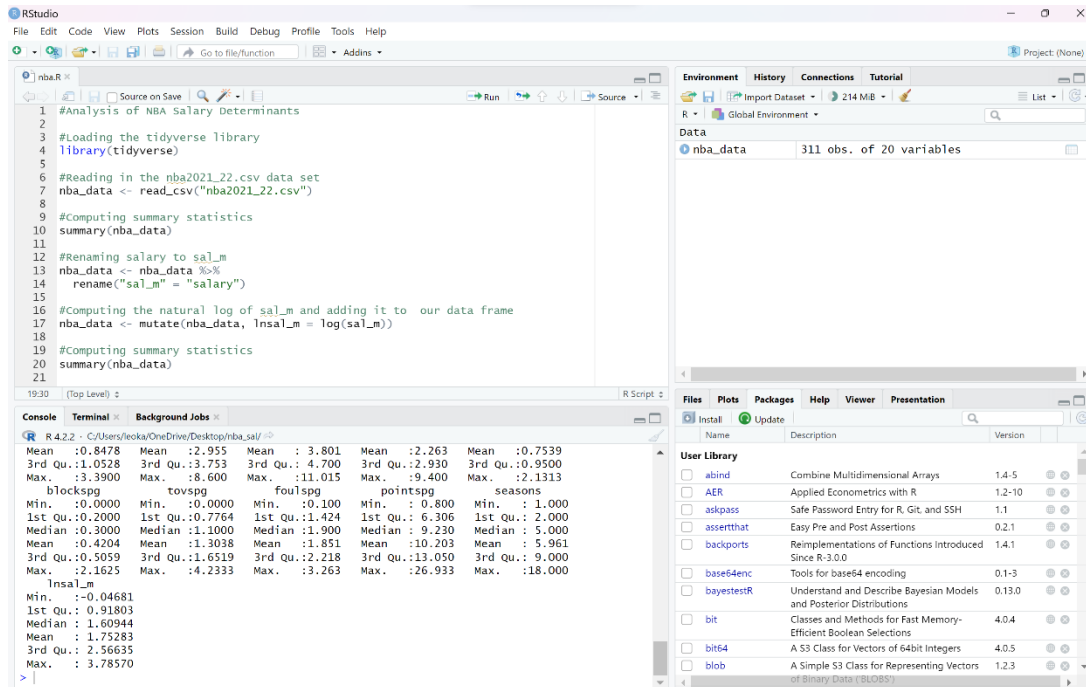
```
mutate(df, new_variable = function_of_existing_variable)
```

Where `df` is the name of the data frame containing the variable we wish to transform,

`new_variable` is the name of the new variable we wish to create, and

`function_of_existing_variable` is where we type the equation for the new variable, (`log()` in this example). We can see that no `$` operator is needed with the `mutate()` function.

Adding the above code to our current R script, as well as a new `summary(nba_data)` command, and running it, we find,



We see in the environment window that our data frame `nba_data` now has an additional column (for a total of 20) containing the variable `lnsal_m`.

Other common transformations include the following:

`mutate(df, xsq = x^2)` ← this will create a new variable called `xsq` equal to  $x^2$

`mutate(df, absx = abs(x))` ← this will create a new variable `absx` equal to the absolute value of the variable `x`

`mutate(df, xy = x*y)` ← this will create a new variable `xy` equal to the product of the variables `x` and `y`

### *Dropping a Variable*

If one wishes to drop one or more variables from the data set, this can be done using the `select()` function, which is part of the `dplyr` package in the `tidyverse` package. The basic form is,

```
select(df, -c(var1, var2,...))
```

Notice that we have `-c(var1, var2,...)`, which will remove the columns for variables `var1, var2,...`. If we used `c(var1, var2,...)`, without the `-` sign, it would only keep `var1, var2,...`.

As an example, if we wanted to remove the newly created `lnsal_m` variable, we could type:

```
#Dropping lnsal_m using select
nba_data <- select(nba_data, -c(lnsal_m))
```

Adding this to our R script file would replace the data frame `nba_data` with a new version that has the same name, but now excludes the `lnsal_m` variable.

An alternative approach to dropping a variable is to use the `mutate` function (again, as part of the `dplyr` package that is part of the `tidyverse` package),

```
#Dropping lnsal_m using mutate
nba_data <- nba_data %>%
  mutate(lnsal_m=NULL)
```

Here, again, we are making use of the pipe (`%>%`) operator. The `lnsal_m=NULL` portion tells R to set the column `lnsal_m` to `NULL`, which removes the column from the `nba_data` data frame.

## 8. Summary Statistics and Correlations

### *Summary Statistics*

One of the first things that a researcher will compute once their data set has been read into R are summary statistics (also called descriptive statistics). This will allow the researcher to get a feel for the values in the data set (e.g., how big are typical values, and how much they vary). It may also reveal unusual observations that will show up as minimum or maximum values. (In some cases, it may uncover errors in data. For example, if the minimum value for players' weight was -160, then this would indicate that there is at least one data point that was entered incorrectly into the data set since weight cannot be negative.)

We have already seen that the `summary()` function will do this for us. In some cases, however, we may want to produce summary statistics for just a subset of the columns of data in a data frame. This can be done by modifying the `summary()` function to select only specific columns. For example, to compute summary statistics for the variables `age`, `sal_m`, and `seasons`, we can add the following to our R script,

```
#Computing summary statistics on a subset of columns  
summary(nba_data[c("age", "sal_m", "seasons")])
```

The results shown in the console would be,

```

> #Computing summary statistics on a subset of columns
> summary(nba_data[c("age", "sal_m", "seasons")])
      age      sal_m      seasons
Min.   :20.00   Min.   : 0.9543   Min.   : 1.000
1st Qu.:23.00   1st Qu.: 2.5044   1st Qu.: 2.000
Median :26.00   Median : 5.0000   Median : 5.000
Mean   :26.73   Mean   : 9.8487   Mean   : 5.961
3rd Qu.:30.00   3rd Qu.:13.0182   3rd Qu.: 9.000
Max.   :41.00   Max.   :44.0664   Max.   :18.000
> |

```

## Correlations

In addition to descriptive statistics, a researcher may wish to study the correlation between various pairs of variables. This may be particularly useful when we suspect that high multicollinearity is a problem in our regression analysis, (see Chapter 8 in *Regression Basics*). Suppose we wish to compute the correlations between the variables age, weight, and height. This can be done in R using the `cor()` function. The code looks very similar to the previous example for summary statistics on a subset of columns,<sup>8</sup>

```

#Computing pairwise correlation for several numeric variables
cor(nba_data[c("age", "weight", "height")])

```

The output from the console is:

```

> #Computing pairwise correlation for several numeric variables
> cor(nba_data[c("age", "weight", "height")])
      age      weight      height
age      1.00000000  0.1081250 -0.04188815
weight  0.10812497  1.0000000  0.72516561
height -0.04188815  0.7251656  1.00000000
>

```

We see the resulting matrix shows a relatively high correlation between weight and height, as one would expect.

---

<sup>8</sup> Note that we cannot compute correlations on string variables, such as `player`. Attempting to do so will produce an error message.

## 9. Ordinary Least Squares (OLS) Regression

### *Estimating an OLS Regression*

Suppose we wish to estimate an OLS regression using our NBA data set with `sal_m` as the dependent variable and `seasons` and `pointspg` (points per game) as our independent variables.

This can be done using R's linear model function, `lm()`. The basic form for this function is,

```
lm(dv ~ iv1 + iv2 +..., data=df)
```

where `dv` is the name of our dependent variable (e.g., `sal_m`), and `iv1`, `iv2`, ..., are the names of our independent variables (e.g., `seasons` and `pointspg`), and `df` is the name of the data frame that contains the data we will use. An example of code is,

```
#Running an OLS regression of sal_m on seasons and pointspg
reg1 <- lm(sal_m ~ seasons + pointspg, data=nba_data)
summary(reg1)
```

We see that the above code assigns to the object `reg1` the results produced by the function,

`lm(sal_m ~ seasons + pointspg, data=nba_data)`, then we use the function `summary(reg1)` to display the results contained in the object `reg1`. Adding this code to our R script file and running it we get the results shown in the console,

```

Console Terminal x Background Jobs x
R 4.2.2 · C:/Users/leoka/OneDrive/Desktop/nba_sal/

> #Running an OLS regression of sal_m on seasons and pointspg
> reg1 <- lm(sal_m ~ seasons + pointspg, data=nba_data)
> summary(reg1)

Call:
lm(formula = sal_m ~ seasons + pointspg, data = nba_data)

Residuals:
    Min       1Q   Median       3Q      Max
-24.5211  -4.5505  -0.1624   3.9268  28.7581

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -5.18771    0.96257  -5.389 1.41e-07 ***
seasons         0.29984    0.10250   2.925  0.0037 **
pointspg       1.29856    0.08437  15.392 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.502 on 308 degrees of freedom
Multiple R-squared:  0.5063,    Adjusted R-squared:  0.5031
F-statistic: 157.9 on 2 and 308 DF,  p-value: < 2.2e-16

```

Note that we can economize on our coding by nesting the `lm()` function inside the `summary()` function,

```
#Running an OLS regression of sal_m on seasons and pointspg (nested version)
summary(reg1 <- lm(sal_m ~ seasons + pointspg, data=nba_data))
```

The above code will produce the exact same results as the previous example (try it!). This demonstrates one of the nice features in R, namely the ability to nest functions within other functions. We can also see that the object `reg1` has been added to our environment window,

```

#Dropping the lnsal_m column of data using select
#nba_data <- select(nba_data, -c(lnsal_m))
#Dropping the lnsal_m column of data using mutate
nba_data <- nba_data %>%
  mutate(lnsal_m=NULL)
#Computing summary statistics
summary(nba_data)
#Computing summary statistics on a subset of columns
summary(nba_data[c("age", "sal_m", "seasons")])
#Computing pairwise correlation for several numeric variables
cor(nba_data[c("age", "weight", "height")])
#Running an OLS regression of sal_m on seasons and pointspg
reg1 <- lm(sal_m ~ seasons + pointspg, data=nba_data)
summary(reg1)
#Running an OLS regression of sal_m on seasons and pointspg (nested version)
summary(reg1 <- lm(sal_m ~ seasons + pointspg, data=nba_data))

```

```

Call:
lm(formula = sal_m ~ seasons + pointspg, data = nba_data)

Residuals:
    Min       1Q   Median       3Q      Max
-24.5211  -4.5505  -0.1624   3.9268  28.7581

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -5.18771    0.96257  -5.389 1.41e-07 ***
seasons      0.29984    0.10250   2.925  0.0037 **
pointspg     1.29856    0.08437  15.392 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.502 on 308 degrees of freedom
Multiple R-squared:  0.5063,    Adjusted R-squared:  0.5031
F-statistic: 157.9 on 2 and 308 DF,  p-value: < 2.2e-16

```

## Note on Significance Codes:

When R produces OLS output, it includes significance codes near the bottom of the output. In the previous output table, we see a key:

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

These codes indicate what level of significance is achieved for the zero-hypothesis test for each estimated coefficient. Thus, for pointspg we see \*\*\* at the end of the line for that coefficient.

According to the codes, this means the estimated coefficient has a p-value between 0 and 0.001 (i.e., 0 '\*\*\*' 0.001). For seasons we have \*\* following its estimated coefficient, meaning the p-value is between 0.001 and 0.01 (i.e., 0.001 '\*\*' 0.01), and so on.



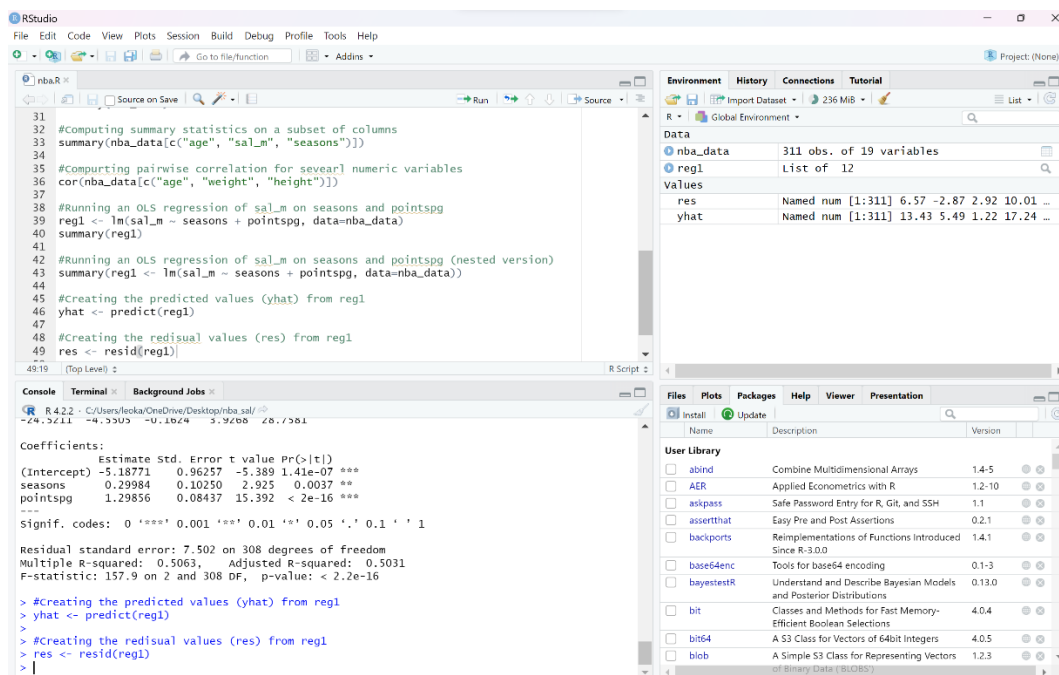
## *Saving the Predicted Values and Residuals*

Once a regression has been estimated, there are a number of post estimation functions that we may wish to employ. For example, we can create the predicted values (the  $\hat{Y}$ 's) and the residuals (the  $e$ 's) by using the following functions:

`yhat <- predict(reg1)`       $\leftarrow$  this will create a new object called `yhat` with the predicted values for the dependent variable (`sal_m`) used in `reg1`.

`res <- resid(reg1)`       $\leftarrow$  this will create a new object called `resid` with the predicted residuals for the regression `reg1`.

Adding these two lines of code to our R script file, and running them, we get,



We see in the environment window that two objects have appeared, `res`, and `yhat`. These objects contain the residuals and predicted values, respectively, from our regression (`reg1`). We can view the residuals by typing `view(res)` at the prompt in the console and then hitting 'Enter'. For the

predicted values, we type `view(yhat)` at the prompt in the console and then hit ‘Enter’. We will later see how we can use these new objects to create a ‘residuals vs. fitted’ graph that is useful for visually checking of heteroskedasticity, (see Chapter 8 in *Regression Basics*).<sup>9</sup>

### *Creating Dummy Variables*

In some cases, we would like to create a dummy variable (also known as an indicator variable) to cover categories represented by a string variable (or a numeric variable representing categories) in our data set. For example, in our NBA data set we have a variable called ‘position’ which is a string variable where the entry ‘C’ stands for the position ‘Center’, ‘PF’ is ‘Power Forward’, and so on. We would like to create a dummy variable that equals 1 if a player is, say, a center, 0 otherwise. Similarly, we can create another dummy variable equal to 1 if a player is a power forward, 0 otherwise. We can do this for the other positions as well. There are several ways we can create these dummy variables in R. First, we can use base R’s `ifelse()` function,

```
#Creating dummy variables for player position with base R

#Centers
nba_data$center <- ifelse(nba_data$position=="C", 1, 0)

#Power forwards
nba_data$pforward <- ifelse(nba_data$position=="PF", 1, 0)

#Small forwards
nba_data$sforward <- ifelse(nba_data$position=="SF", 1, 0)

#Point guards
nba_data$pguard <- ifelse(nba_data$position=="PG", 1, 0)

#Shooting guards
nba_data$sguard <- ifelse(nba_data$position=="SG", 1, 0)
```

---

<sup>9</sup> The predicted values are also known as the fitted values.

Above we can see code that creates five dummy variables, one for each position in our data set.

Focusing on the case for centers (denoted with a “C” in our data frame), we have,

```
#Centers
nba_data$center <- ifelse(nba_data$position=="C", 1, 0)
```

This line of code says to assign to the object `nba_data$center` the results from the function

`ifelse(nba_data$position=="C", 1, 0)`. Note that in both chunks of the code we are using the

`$` operator to specify a column in the data frame `nba_data`. The first part of the code,

`nba_data$center <-`, tells R that we are assigning a new column called `center` to the existing

frame `nba_data`. The second part of the code, `ifelse(nba_data$position=="C", 1, 0)`, is an ‘if-

else’ statement that says, ‘if a row in the data frame `nba_data` has an entry equal to the text value

`C` for the column `position`, assign the object `nba_data$center` a value of 1, otherwise assign it a

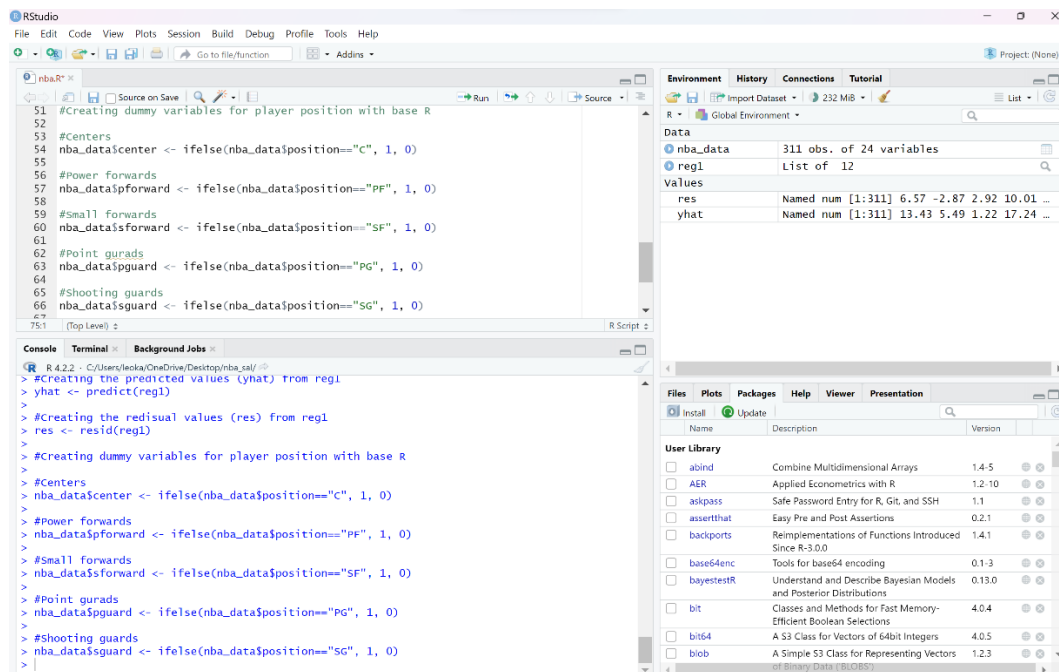
value of 0’. Note that we see the operator `==` here. This is the ‘equals’ operator used by R (and

other programs such, such as Stata), and can be read as ‘is equal to’. This will create a new

column in our data frame `nba_data` called `center` which has a 1 for players who are centers, and

a 0 for players who are not centers. The lines of code for the other positions are similar. Adding

this code to our R script and running it, we get,



We see in our environment window that we now have 24 variables in our data frame `nba_data`, which now includes the five position dummies we have created. Viewing the data frame we can see the added columns,

The screenshot shows the `nba_data` data frame in RStudio. The data frame has 11 columns: `tspsg`, `blockspg`, `tovspg`, `foulspg`, `pointspg`, `seasons`, `center`, `pfoward`, `sforward`, `pguard`, and `sguard`. The data is displayed in a table with 10 rows of player statistics.

tspsg	blockspg	tovspg	foulspg	pointspg	seasons	center	pfoward	sforward	pguard	sguard
0.725000	0.650000	1.512500	1.962500	12.48750	8	0	1	0	0	0
0.633333	0.233333	1.033330	1.533330	7.53333	3	0	0	0	1	0
0.300000	0.200000	0.500000	1.900000	4.70000	1	0	0	1	0	0
0.828571	1.185710	1.550000	2.185710	14.03570	14	1	0	0	0	0
0.623077	0.192308	1.169230	1.676920	10.19230	13	0	0	0	0	1
0.400000	0.900000	2.200000	1.300000	8.20000	1	0	1	0	0	0
0.950000	0.325000	1.225000	1.750000	6.17500	4	0	0	0	0	1
0.350000	0.950000	1.130000	2.370000	6.89000	10	1	0	0	0	0
1.394120	0.541176	1.788240	1.764710	11.01180	17	0	0	1	0	0

Another approach to creating position dummies is to use the R package called `fastDummies`. This package, available for installation at CRAN, has a function called `dummy_cols()` that can transform a factor variable (i.e., a categorical variable like `position`) in a data frame into new columns of data with a dummy variable for each category in the factor variable. The first step to

using this function is to install the `fastDummies` package. We can do this by typing at the console prompt: `install.packages("fastDummies")`, and hitting the 'Enter' key.

Next, we can include the following code in our R script file:

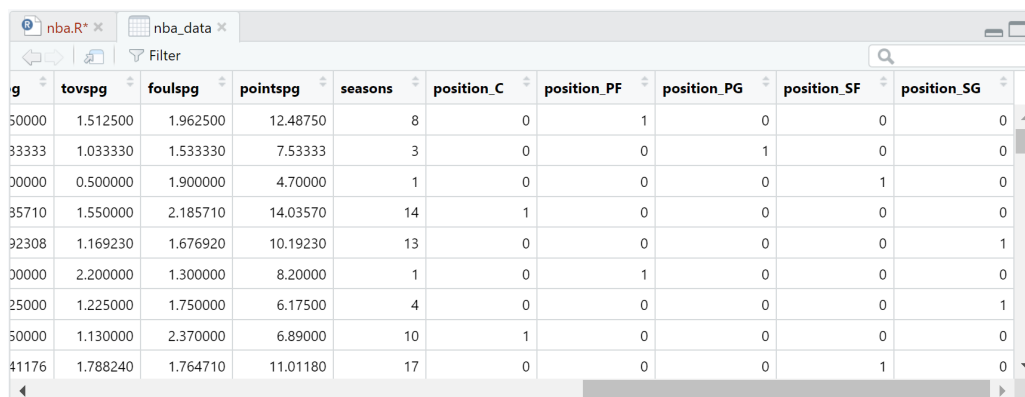
```
#Creating dummy variables for player position with dummy_cols()
library(fastDummies)
nba_data <- dummy_cols(nba_data, select_columns = c("position"))
```

The above code first loads the `fastDummies` library. The next line of code assigns to the object `nba_data`, our existing data frame, the results of the function,

```
dummy_cols(nba_data, select_columns = c("position"))
```

The first argument in this function is the data frame, `nba_data`, that contains the factor variable for which we want to create dummies. The second argument in this function,

`select_columns = c("position")`, identifies the specific column of data for which we want to create dummies.<sup>10</sup> Adding the above code to our R script and running it gives us,



g	tovspg	foulspg	pointspg	seasons	position_C	position_PF	position_PG	position_SF	position_SG
50000	1.512500	1.962500	12.48750	8	0	1	0	0	0
33333	1.033330	1.533330	7.53333	3	0	0	1	0	0
00000	0.500000	1.900000	4.70000	1	0	0	0	1	0
35710	1.550000	2.185710	14.03570	14	1	0	0	0	0
92308	1.169230	1.676920	10.19230	13	0	0	0	0	1
00000	2.200000	1.300000	8.20000	1	0	1	0	0	0
25000	1.225000	1.750000	6.17500	4	0	0	0	0	1
50000	1.130000	2.370000	6.89000	10	1	0	0	0	0
41176	1.788240	1.764710	11.01180	17	0	0	0	1	0

The last five columns are our newly created position dummies. Note that if we also want to create dummies for the factor variable `team`, we would include this column name in the combine

<sup>10</sup> If we leave out the `select_columns` option then the function's default is to create dummy variables for all possible factor variables in the data frame, including the `player` column that has player names. Thus, we would have a dummy for each individual player, which we obviously do not want or need in this case.

function, `c("position", "team")`. In this case, in addition to the five player position dummies, we would have another thirty team dummies, one for each NBA team in our data frame. Using the `fastDummies` package illustrates how user-created packages can often make our coding much more efficient.

We can also include dummy variables for player position in our regression model by using the `as.factor()` function as part of our `lm()` function that estimates our linear regression model. We can add the following code to our R script file,

```
#Including dummy variables for player position using as.factor()
reg2 <- lm(sal_m ~ seasons + pointspg + as.factor(position), data=nba_data)
summary(reg2)
```

In this case, the `as.factor(position)` component will create dummy variables for each position category, excluding one category which will serve as our base case, and include these dummies in the regression estimation. Note that the `as.factor()` will not save the dummy variables in the data frame, they are only created and included in the regression and then dropped. Adding the above code produced the following output,

The screenshot shows the RStudio interface. The script editor on the left contains the following code:

```

64
65 #Shooting guards
66 nba_data$sguard <- ifelse(nba_data$position=="SG", 1, 0)
67
68
69 #Creating dummy variables for player position with dummy_cols()
70 library(fastDummies)
71 nba_data <- dummy_cols(nba_data, select_columns = c("position"))
72
73
74 #Including dummy variables for player position using as.factor()
75 reg2 <- lm(sal_m ~ seasons + pointspg + as.factor(position), data=nba_data)
76 summary(reg2)
77
78

```

The console on the bottom left shows the output of the regression:

```

call:
lm(formula = sal_m ~ seasons + pointspg + as.factor(position),
    data = nba_data)

Residuals:
    Min       1Q   Median       3Q      Max
-24.9316  -4.5537   0.0756   3.7809  29.0597

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -5.72158    1.57614   -3.630 0.000332 ***
seasons         0.29976    0.10366    2.892 0.004107 **
pointspg       1.31869    0.08762   15.050 < 2e-16 ***
as.factor(position)PF  0.49973    1.70352    0.293 0.769456
as.factor(position)PG -0.42805    1.66838   -0.257 0.797686
as.factor(position)SF  1.23972    1.62892    0.761 0.447208
as.factor(position)SG  0.20072    1.59481    0.126 0.899929
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.53 on 304 degrees of freedom
Multiple R-squared:  0.5091, Adjusted R-squared:  0.4994
F-statistic: 52.55 on 6 and 304 DF,  p-value: < 2.2e-16

```

The environment pane on the right shows the following objects:

Object	Type	Length
nba_data	Data frame	311 obs. of 29 variables
reg1	lm	List of 12
reg2	lm	List of 13
res	Named num	[1:311] 6.57 -2.87 2.92 10.01 ...
yhat	Named num	[1:311] 13.43 5.49 1.22 17.24 ...

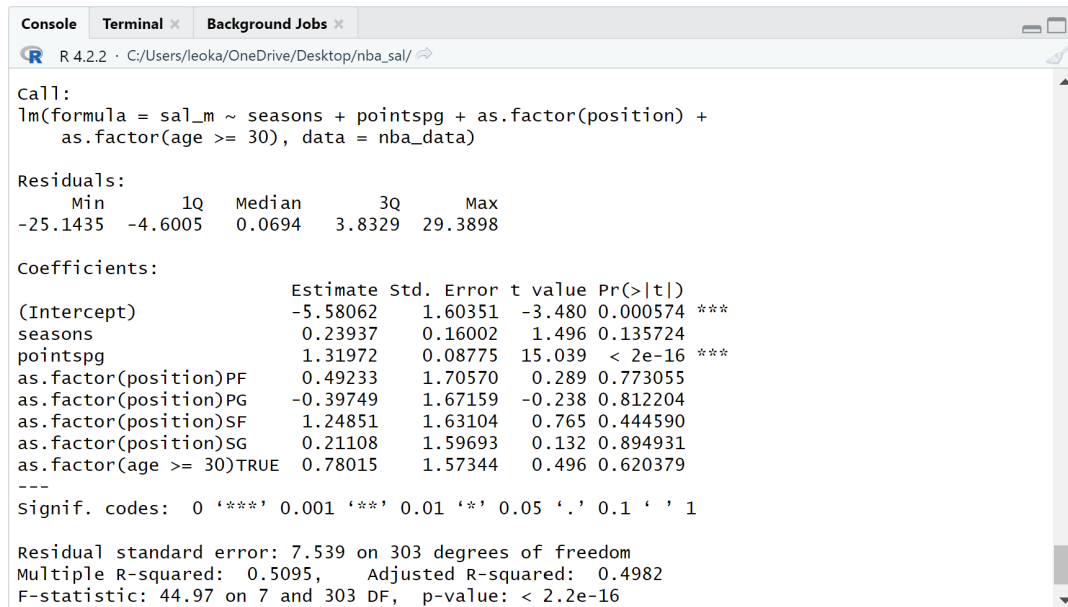
We can see in the console that the new regression results include dummy variables for player position, excluding C (centers) which serves as our base case.<sup>11</sup> The `as.factor()` function may be the desired approach when we have many categories and we don't necessarily want to create and keep dummy variables for each category in our data frame.

Lastly, we can also create dummy variables for numeric measures. For example, suppose we wanted to identify players who were 30 years old or older. We can include a dummy variable for players who are 30 or more years old in our script,

<sup>11</sup> If we want to use a different category as the base case, the `relevel()` function can be used (an internet search will locate examples of how to use this function).

```
#Including dummy variables for player position and age>=30 using as.factor()
reg3 <- lm(sal_m ~ seasons + pointspg + as.factor(position) + as.factor(age>=30),
           data=nba_data)
summary(reg3)
```

Including the above code and re-running our script file yields the following regression output,



```
Call:
lm(formula = sal_m ~ seasons + pointspg + as.factor(position) +
    as.factor(age >= 30), data = nba_data)

Residuals:
    Min       1Q   Median       3Q      Max
-25.1435  -4.6005   0.0694   3.8329  29.3898

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)    -5.58062    1.60351  -3.480 0.000574 ***
seasons          0.23937    0.16002   1.496 0.135724
pointspg        1.31972    0.08775  15.039 < 2e-16 ***
as.factor(position)PF  0.49233    1.70570   0.289 0.773055
as.factor(position)PG -0.39749    1.67159  -0.238 0.812204
as.factor(position)SF  1.24851    1.63104   0.765 0.444590
as.factor(position)SG  0.21108    1.59693   0.132 0.894931
as.factor(age >= 30)TRUE 0.78015    1.57344   0.496 0.620379
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.539 on 303 degrees of freedom
Multiple R-squared:  0.5095,    Adjusted R-squared:  0.4982
F-statistic: 44.97 on 7 and 303 DF,  p-value: < 2.2e-16
```

The above output shows the estimated coefficient for a dummy variable for players with an age greater than or equal to 30 years old of about 0.78 (though it is not statistically significant).



## 10. Creating a Scatterplot with Reference Line

It may be useful to create a scatterplot for a pair of variables and include a reference line in the graph. This is a simple way of visually understanding the relationship between two variables, and it may also be helpful for visually checking for heteroskedasticity (see Chapter 8 in *Regression Basics*). Making graphs in R can be done in a variety of ways. We will start with some created using base R code. We will also see how the `ggplot2` package, which is part of the `tidyverse` package, can be used to make very attractive graphs. The richness of the `ggplot2` package is hard to overstate and is one of the reasons that many data scientists work with R to create their visualizations.

### *Base R Scatterplot with OLS Regression Line*

The `plot()` function is capable of creating simple scatterplots for two variables in a data frame. It has two main arguments, the variable plotted on the x-axis, and the variable plotted on the y-axis. We can also add labels for the two axes and a label for the graph overall. Below is a new R script file (called `nba2.R`) that reads the `nba2021_22.csv` data set into R and then creates a scatterplot of the natural log of salary against `pointspg`,

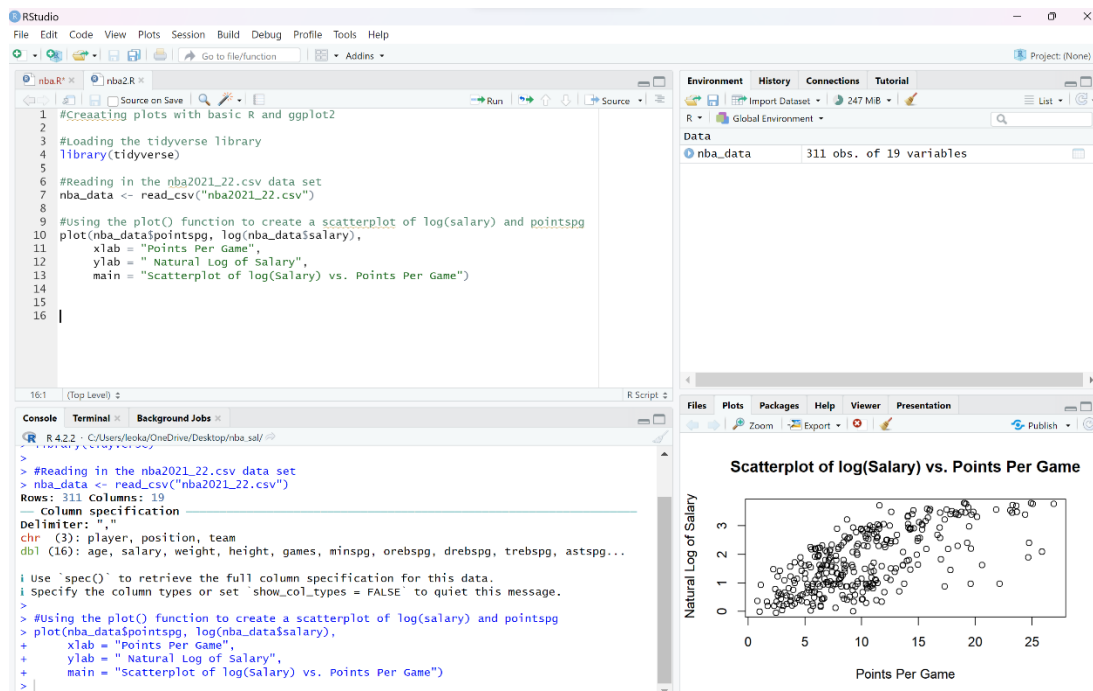
```
#Creaating plots with base R and ggplot2

#Loading the tidyverse library
library(tidyverse)

#Reading in the nba2021_22.csv data set
nba_data <- read_csv("nba2021_22.csv")

#Using the plot() function to create a scatterplot of log(salary) and pointspg
plot(nba_data$pointspg, log(nba_data$salary),
     xlab = "Points Per Game",
     ylab = "Natural Log of Salary",
     main = "Scatterplot of log(Salary) vs. Points Per Game")
```

We see in the line, `plot(nba_data$pointspg, log(nba_data$salary))`, that we are identifying the two columns in the data frame `nba_data` that we want to plot, `pointspg` on the horizontal axis, and the natural log of salary on the vertical axis, using the `log()` function here. (Recall that the `$` operator is used to refer to a particular part of an object, e.g., a column in a data frame.) The next three lines are labeling the x-axis (`xlab = "Points Per Game"`), the y-axis (`ylab = "Natural Log of Salary"`), and the overall graph (`main = "Scatterplot of log(Salary) vs. Points Per Game"`). Pasting the above code into an R script file and running it we get,<sup>12</sup>



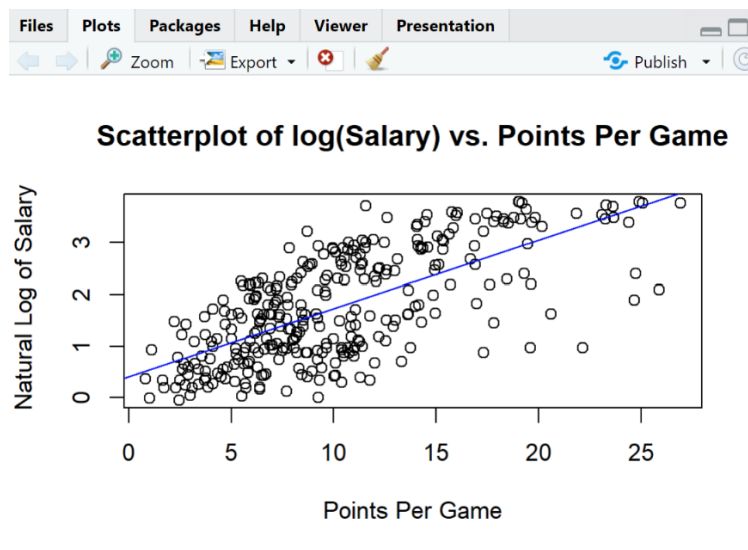
We can see in the lower right window that our scatterplot is displayed. Note that there is an option to 'Export' the plot on the menu in this window. Users can export the plot and then copy it into a document.

<sup>12</sup> This new R script file was saved in our previously created project folder, "nba\_sal", which contains our `nba2021_22.csv` data set, thus there is no need to provide the path in the `read_csv()` function. If you are not working within this project folder, you will need to include the path to the `nba2021_22.csv` file.

We can add an OLS regression line to the previous plot using the `abline()` function. This function adds a line to a plot and does so with two arguments, an intercept term (`a = value1`) and a slope term (`b = value2`). Where `value1` would be the intercept value, and `value2` would be the value for the slope. We could run a regression using the `lm()` function to find the intercept and slope terms to use for `value1` and `value2`.<sup>13</sup> Alternatively, we can use R's ability to nest functions and add the following line of code to our script file,

```
#Adding OLS regression line
abline(lm(log(salary) ~ pointspg, data=nba_data), col="blue")
```

Here the values for the arguments `a = value1`, `b = value2`, are the OLS results of these two values produced by the `lm()` function. We have also added an option, `col="blue"` which tells R to plot a blue line rather than the default, which is a black line. Adding this code to our script file and running it adds the OLS regression line,



<sup>13</sup> Specifically, we could add the line: `summary(lm(log(salary) ~ pointspg, data=nba_data))`. Running this regression, we would find the intercept term of approximately 0.411, and a slope term of about 0.1315. These values could then be inserted into the `abline()`, so we would have: `abline(a = 0.411, b = 0.131)`.

*Residuals vs. Fitted Plot*

As noted earlier, another common graph is one that, following a regression estimation, plots the residuals on the vertical axis against the fitted (a.k.a. ‘predicted’) values on the horizontal axis.

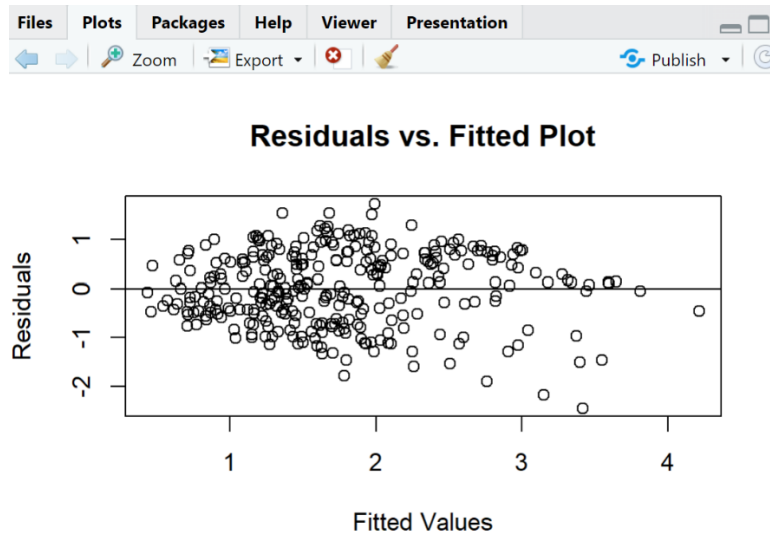
Such a graph is useful for visually checking for heteroskedasticity. This can be done using the `plot()` function described earlier with the following code,

```
#Plotting the residuals vs. fitted values after OLS regression

#First running the OLS regression of log(salary) on seasons and pointspg
reg1 <- lm(log(salary) ~ seasons + pointspg, data = nba_data)

#Using the plot() function
plot(fitted(reg1), residuals(reg1),
     xlab = "Fitted Values",
     ylab = "Residuals",
     main = "Residuals vs. Fitted Plot")
abline(h=0)
```

The first step is to run an OLS regression with `log(salary)` as the dependent variable, and `seasons` and `pointspg` as the independent variables. We assign the results of this regression to the object `reg1`. Next we use the `plot()` function and for the x-axis and y-axis, we use the `fitted()` and `residuals()` functions’ values (once again, illustrating how we can nest functions in R). The next three lines of code are similar to the previous example where we provide labels for the x-axis, the y-axis, and the graph overall. The last line of code, `abline(h=0)`, uses the `abline()` function to add a horizontal line at zero. Adding the above code to our script file and running it give us,



Once again, this plot can be exported by clicking on the ‘Export’ button in the window where the plot is displayed.

### *Using ggplot2 to Create Plots*

While the `plot()` function gets the job done, there is another R package that is much more powerful in creating data visualizations, namely the `ggplot2` package. This is one of the packages included in the `tidyverse` package we have been working with. We will see how we can make plots like those shown earlier, but it should be noted that there are entire books devoted to describing how `ggplot2` can be used to make amazing, highly customizable visualizations.<sup>14</sup>

The basic syntax used in a `ggplot2` visualization looks like the following,

```
ggplot(data = , aes(x = , y = )) +  
  geom_point()
```

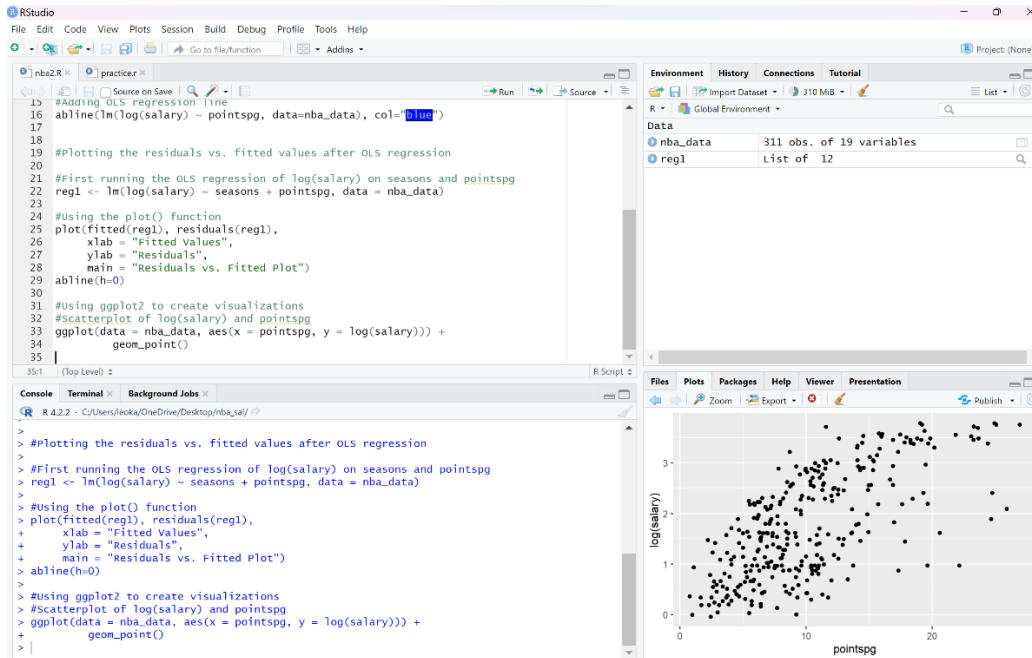
<sup>14</sup> A popular book is, *R Graphics Cookbook*, 2nd edition, by Winston Chang. See: <https://r-graphics.org/>.

We can see that there are four components: the `ggplot()` function, the `data` parameter, the `aes()` function, and the `geom_point()` object. The `ggplot()` function simply initiates the creation of a visualization using the `ggplot2` package that is installed. The `data` parameter is where we specify which data frame we will be using for the visualization. The `geom` component is the geometric object we wish to create, in this case `geom_point()` indicates that we wish to plot points (i.e., create a scatterplot). (Other ‘geoms’ are possible here, such as line plots, histograms, etc.) The default of `geom_point()` is to plot black, round points. If we wanted to customize the color of the points and/or their size and shape, we would add more details between the parentheses of the `geom_point()` component (e.g., adding `color = "red"` will plot round, red points).

The `aes(x = , y = )` describes the ‘aesthetic’ mapping we will use. That is, the columns of data we will use from the data frame that will be ‘mapped’ into the object we are creating, which is a scatterplot in this case. An example of code that would produce a scatterplot with `pointspg` on the x-axis, and `log(salary)` on the y-axis, we have,

```
#Using ggplot2 to create visualizations
#Scatterplot of log(salary) and pointspg
ggplot(data = nba_data, aes(x = pointspg, y = log(salary))) +
  geom_point()
```

Adding this code to our script file and running it we get,



We see in the lower right window the scatter plot has been created. The plot can be exported by hitting the ‘Export’ button.<sup>15</sup>

An alternative to simply calling the `ggplot` function and having the graph created in the viewer, we can assign the plot to an object, and then have R display this object. While this produces the same plot, it has the advantage of allowing us to very easily modify or enhance the plot by simply referring to the object and then adding our additions. For example, consider this alternative code for our scatterplot,

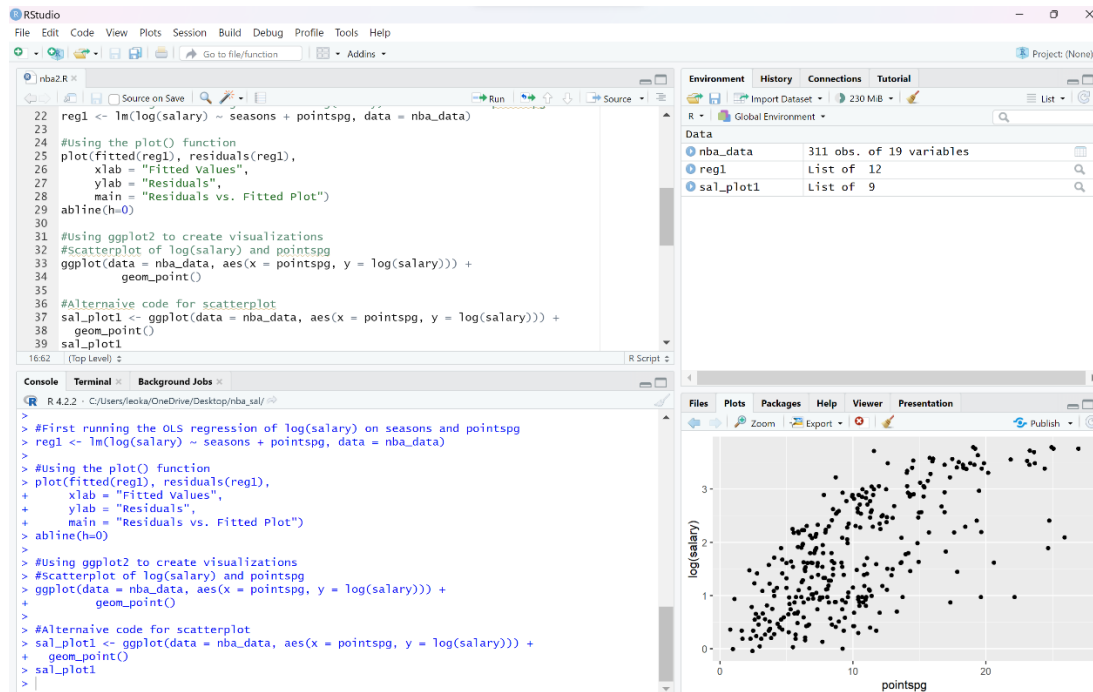
```

#Alternative code for scatterplot
sal_plot1 <- ggplot(data = nba_data, aes(x = pointspg, y = log(salary))) +
  geom_point()
sal_plot1

```

<sup>15</sup> Alternatively, the `ggsave()` function can be used to save the plot with a specific file type, (e.g., .png, .jpeg, .tiff, etc.). See <https://ggplot2.tidyverse.org/reference/ggsave.html> for details on how to use this function.

The above code (after the comment) assigns to the object `sal_plot1` the results of the `ggplot` function (which is the same code used earlier). The next line, `sal_plot1` simply has R display this object. Adding the above code to our script and running it produces,



This is the same result as before, but now we have a newly created object listed in the environment window, `sal_plot1`. How is this useful? Now we can make changes or enhancements to our scatterplot by simply referring to this object and then adding elements. For example, suppose we want to identify each player's position by coloring the plotted points. We can do this by using the categorical variable `position` that is part of our `nba_data` data frame and including this in the aesthetics (`aes()`) component. For example, we can use the following code,

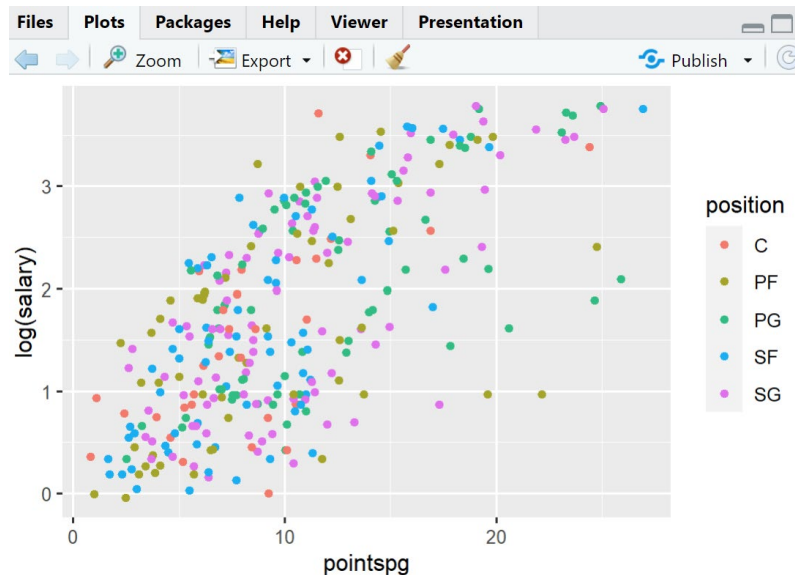
```

#Adding color to plotted points identifying player position
sal_plot2 <- sal_plot1 + aes(color=position)
sal_plot2

```



The above code assigns to the object `sal_plot2` the contents of the previously created object `sal_plot1`, and modifies it by adding the code, `+ aes(color=position)`, running this new code chunk produces the following plot,

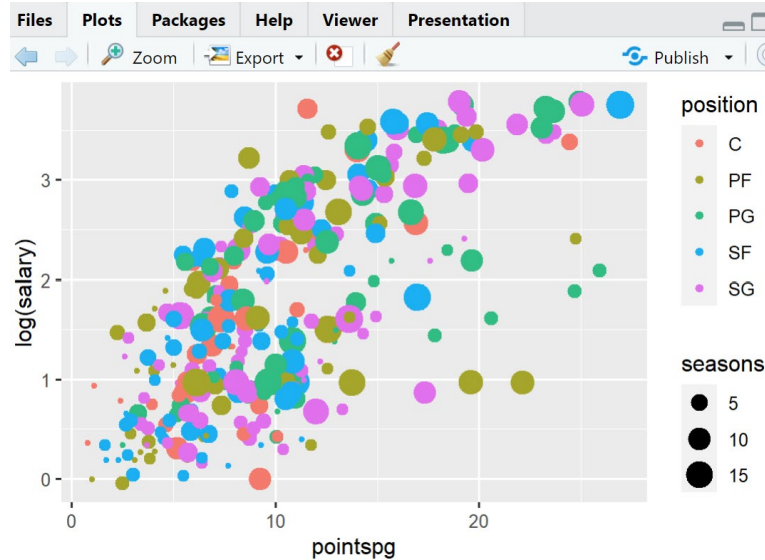


We see that the plotted points have been colored and a key is added to identify positions by color.<sup>16</sup> We can add another feature by adjusting the point size by having it reflect the number of seasons a player has played in the NBA. The additional code is,

```
#Make point size reflect seasons of experience
sal_plot3 <- sal_plot2 + aes(size=seasons)
sal_plot3
```

Adding this code chunk to our script file and running it produces the following plot,

<sup>16</sup> We could have added the code, `+ aes(color=position)`, to our original `ggplot` code and achieved the same results. But the alternative approach of creating the `sal_plot1` object first and then altering our plot allows us to efficiently “build out” our visualization in steps and reduces the likelihood of coding errors.



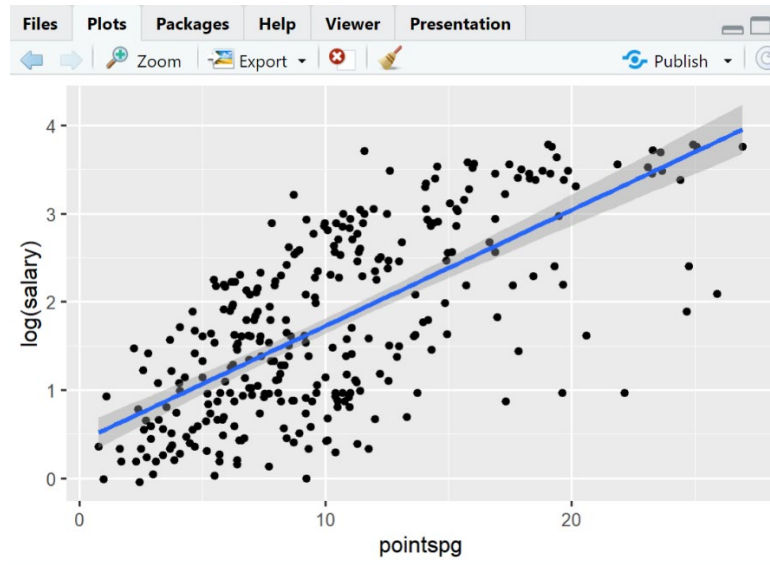
### *Adding an OLS Regression Line*

If we wish to add an OLS regression line to our scatterplot, we can simply add another “geom” to the existing code. We can add this to our first scatterplot, `sal_plot1`, with the following code,<sup>17</sup>

```
#Adding a best fit line to our scatterplot
sal_plot4 <- sal_plot1 + geom_smooth(method="lm")
sal_plot4
```

The added code, `+ geom_smooth(method="lm")`, essentially adds another layer to our plot which is a smooth line determined by the linear model ("lm") method. Adding this code and running it give us,

<sup>17</sup> We could add this to the previous plot, `sal_plot3`, however this would overly clutter the plot.



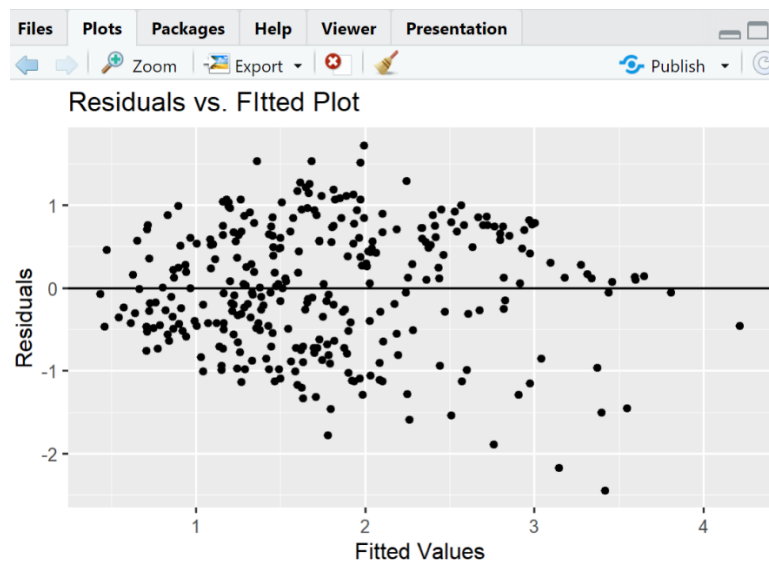
We see that a blue (the default color) OLS regression line is added, as well as a 95% confidence level band. We could customize the colors and confidence level with options added to the `geom_smooth(method="lm")` component. For example, `geom_smooth(method="lm", color="red", level=0.90)` will plot a red OLS regression line with a 90% confidence level band.

### *Residuals vs. Fitted Plot*

We can also create a residuals vs. fitted plot using the `ggplot()` function. For example, we can add the following code to our script file,

```
#Plotting the residuals vs. fitted using ggplot2
fit <- fitted(reg1)
resid <- residuals(reg1)
ggplot(data = nba_data, aes(x = fit, y = resid)) +
  geom_point() +
  geom_hline(yintercept = 0) +
  labs(x = "Fitted Values", y = "Residuals") +
  ggtitle("Residuals vs. Fitted Plot")
```

In the above code we can see some added features. We see that we have added a “geom”, `geom_hline(yintercept = 0)`, which will add a horizontal line to our plot at the value 0. We have also added two other functions, `labs(x = "Fitted Values", y = "Residuals")`, and `ggtitle("Residuals vs. Fitted Plot")`. The first renames the axes labels, and the second provides a title for the plot. Adding the above code to our script file and running it, we get,



Before moving on, it is worthwhile noting that the above examples are just scratching the surface of the possibilities with the `ggplot()` function. Extraordinary visualizations can be made with this package and, in addition to the many books on how to work with `ggplot2`, there are numerous online resources available. A good place to start is: <https://ggplot2.tidyverse.org/>.

## 11. Breusch-Pagan Test for Heteroskedasticity and OLS with Robust Standard Errors

As discussed in Chapter 8 or *Regression Basics*, the presence of heteroskedasticity creates problems as the regular OLS coefficient standard errors used for inference tests are no longer legitimate. We saw in our scatterplots earlier that there seems to be evidence of heteroskedasticity in our NBA salary regression. To be more careful about our conclusion, we can conduct the Breusch-Pagan (“BP”) test for heteroskedasticity, and if present then use robust standard errors to allow for legitimate inference tests. To illustrate the BP test, we will create a new R script file, `nba3.R`, which we will use to focus on this, and various regression diagnostics tests. We will continue to work within the NBA project folder, `nba_sal`, to simplify our code regarding paths to files.

We will begin by first estimating a benchmark OLS regression for our NBA data frame with `log(salary)` as the dependent variable, and `seasons` and `pointspg` (points per game) as our independent variables. We will then run a BP test for heteroskedasticity, and lastly, we will re-estimate our regression model using robust standard errors.

The following code reads our data set into R and estimates our benchmark model,

```
#Linear model diagnostics

#Loading the tidyverse library
library(tidyverse)

#Reading in the nba2021_22.csv data set
nba_data <- read_csv("nba2021_22.csv")

#Estimating a regression of log(salary) on seasons and pointspg
reg1 <- lm(log(salary) ~ seasons + pointspg, data = nba_data)
summary(reg1)
```

Running this code produces our benchmark OLS results,

```
Call:
lm(formula = log(salary) ~ seasons + pointspg, data = nba_data)

Residuals:
    Min       1Q   Median       3Q      Max
-2.4455 -0.5147  0.0337  0.6111  1.7209

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.301828   0.096189   3.138  0.001867 **
seasons       0.036162   0.010243   3.530  0.000478 ***
pointspg     0.121088   0.008431  14.362 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.7497 on 308 degrees of freedom
Multiple R-squared:  0.4849,    Adjusted R-squared:  0.4815
F-statistic:  145 on 2 and 308 DF,  p-value: < 2.2e-16
```

To conduct the BP test we will make use of the `lmtest` package, which is available from CRAN, and which has several linear model diagnostic tests. The package should be installed by typing in the console,

```
install.packages("lmtest")
```

Hitting the ‘Enter’ key will install the package, (this only needs to be done once). Once installed we can load the package with a `library(lmtest)` command. We can add the following code to our R script file,

```
#Loading lmtest library
library(lmtest)

#Running the Breusch-Pagan test for heteroskedasticity
bptest(reg1)
```

As is evident, the code used to conduct the BP test is very simple, `bptest(reg1)`.<sup>18</sup> Running this code chunk gives us the following results in the console,

---

<sup>18</sup> Note that we could have nested the `lm()` function within the `bptest()` function rather than referring to the object `reg1`, i.e., `bptest(lm(log(salary) ~ seasons + pointspg, data = nba_data))`

```
> #Running the Breusch-Pagan test for heteroskedasticity
> bptest(reg1)

studentized Breusch-Pagan test

data:  reg1
BP = 15.504, df = 2, p-value = 0.00043
```

The null hypothesis for the BP test is that the errors are homoskedastic, (see Chapter 8 in *Regression Basics*). The reported p-value from the BP test, 0.00043, is less than the benchmark 0.05 value and, thus, rejects the null hypothesis. Or, in other words, there is evidence of heteroskedasticity and the reported standard errors for the estimated coefficients in our benchmark OLS model are not legitimate.<sup>19</sup>

Now that we have evidence of heteroskedasticity, we can use robust standard errors to make our inference legitimate. This is easily done using the `estimatr` package, available from CRAN.<sup>20</sup> First we will need to install this package by typing the following into the console and hitting the ‘Enter’ button,

```
install.packages("estimatr")
```

After the package installs (which needs to be done just once), we can load the package by including a `library(estimatr)` command in our R script file. The syntax needed for a basic application of robust standard errors for an OLS estimation is the same as that used for the `lm()` function, except we use the `lm_robust()` function. Thus, we can add the following code to our script file,

---

<sup>19</sup> The BP test performed by Stata produces a slightly different result. This is due to some different defaults used by Stata vs. R (namely, the ‘studentized’ default used by R, and the simplified version of the BP test used by Stata, see Chapter 8, footnote 17 in *Regression Basics*).

<sup>20</sup> See: <https://cran.r-project.org/package=estimatr>. Note that this package has a variety of specialized estimators to deal with various problems.

```
#Re-estimating our regression model with robust standard errors
library(estimatr)
reg2 <- lm_robust(log(salary) ~ seasons + pointspg, data = nba_data)
summary(reg2)
```

Running the script file produces,

```
Call:
lm_robust(formula = log(salary) ~ seasons + pointspg, data = nba_data)

Standard error type: HC2

Coefficients:
              Estimate Std. Error t value Pr(>|t|) CI Lower CI Upper DF
(Intercept)  0.30183    0.085207   3.542 4.581e-04  0.13417  0.46949 308
seasons      0.03616    0.010809   3.346 9.226e-04  0.01489  0.05743 308
pointspg     0.12109    0.008594  14.090 3.851e-35  0.10418  0.13800 308

Multiple R-squared:  0.4849 , Adjusted R-squared:  0.4815
F-statistic: 143.3 on 2 and 308 DF, p-value: < 2.2e-16
```

While the above output is formatted a little differently, the estimated coefficients are the same (except for some rounding), but the standard errors,  $t$  statistics, and associated  $p$ -values have changed. This correction to the standard errors allows for legitimate inference tests for the estimated coefficients.<sup>21</sup>

## 12. Testing for Multicollinearity: Variance Inflation Factor (VIF)

As discussed in Chapter 8 of *Regression Basics*, high multicollinearity can be a nuisance in multiple regression models. The variance inflation factor (VIF) is a common measure used to test for high multicollinearity. To illustrate how to produce VIF estimates with R, we will add an additional independent variable to our NBA salary model, namely games, which is the career number of games a player has played in the NBA. This variable would likely be highly

---

<sup>21</sup> As noted in Chapter 8 of *Regression Basics*, (see footnotes 31 and 32), there are several ways that the standard errors can be adjusted. Stata's default is to use the 'HC1' method. R's default is 'HC2', whereas SPSS lets the user choose between 'HC1', 'HC2', and 'HC3'. The results for the above regression are very similar for each method.



correlated with seasons, which is already in our regression model. Adding the following code to our script file,

```
#Checking for high multicollinearity with the VIF
reg3 <- lm(log(salary) ~ seasons + pointspg + games, data = nba_data)
summary(reg3)
```

Running the regression we get,

```
Call:
lm(formula = log(salary) ~ seasons + pointspg + games, data = nba_data)

Residuals:
    Min       1Q   Median       3Q      Max
-2.46756 -0.54238  0.02852  0.61496  1.61865

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.3869228   0.1063067    3.640  0.00032 ***
seasons     -0.0274590   0.0359101   -0.765  0.44506
pointspg     0.1134996   0.0093483   12.141 < 2e-16 ***
games        0.0011264   0.0006096    1.848  0.06559 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.7468 on 307 degrees of freedom
Multiple R-squared:  0.4905,    Adjusted R-squared:  0.4856
F-statistic: 98.54 on 3 and 307 DF,  p-value: < 2.2e-16
```

Notice that pointspg and games are positive as expected, and significant (with a p-value less than 0.05 for the former, and nearly so for the latter). But for seasons we see that it is no longer significant, and it has an unexpected negative sign. These are signs that we may have high multicollinearity present in the model. [Note that this regression also suffers from heteroskedasticity, but we will ignore this issue for now.] In order to create the VIF measures, we first install the car package from CRAN which has, among many others, a `vif()` function. Thus, we can type into the console,

```
install.packages("car")
```

Hitting the ‘Enter’ key will install the car package, then we can add the following code to our script file,

```
library(car)
vif(reg3)
```

Running this code chunk produces the following output in the console,

```
> library(car)
> vif(reg3)
      seasons  pointspg      games
14.123030    1.412721  15.574543
```

We see that both seasons and games have VIF values greater than 10, indicating high multicollinearity.

### 13. Testing for Autocorrelation and Prais-Winsten Estimation

Autocorrelation, (also known as serial correlation), in the error structure is a common problem with time series data. When present the variance of the residuals (the  $e_t$ 's) tends to underestimate the variance of the true population's errors (the  $u_t$ 's). Since the standard errors of the parameter estimates are based on the variance of the residuals, this means the  $t$  statistics and their associated p-values are invalid. To illustrate how to test and correct for autocorrelation we will work with our alcoholic beverages sales example using the “alc\_sales.csv” data set (available on the *Regression Basics* companion website). Our sample regression model was, (see Chapter 7),

$$\ln alc\_sales_t = a + b_1(tindex) + b_2q2_t + b_3q3_t + b_4q4_t + e_t$$

Where  $\ln alc\_sales_t$  is the natural log of alcoholic beverages sales in the U.S.,  $tindex$  is a time index, and  $q2_t$ ,  $q3_t$ , and  $q4_t$ , were quarter dummies to pick up seasonal effects. Below we will detail how to use R to test for autocorrelation visually and statistically, then we will show how we can implement the Prais-Winsten (PW) estimation to correct for this problem. In order to

carry out these steps we will create a new project folder called “alc\_proj” and place the “alc\_sales.csv” data file in this folder. Next, we can launch R, and create a new project by clicking on ‘File’ on the main menu, then choose, ‘New Project...’. We select ‘Existing Directory’ from the next menu, choose ‘Browse...’, and then navigate to where the newly created “alc\_proj” folder is, click on this folder and then, select ‘Open’, then click ‘Create Project’. This will initiate the new project and set the default path for R to this folder.

Now that the project is set up, we can begin by creating an R script file, we can call it `als_sales.R`, and save it to the project folder.

### *Graphical Detection of Autocorrelation*

As discussed in Chapter 8, one method of detecting autocorrelation is to plot the errors from an OLS regression and check for a pattern. This can be done in this case by estimating the above regression, saving the residuals, and then creating a connected line graph of the residuals over the time periods. This can be accomplished by adding the following code to our R script file,

```
#Analysis of Alcohol Beverages Sales in the U.S.

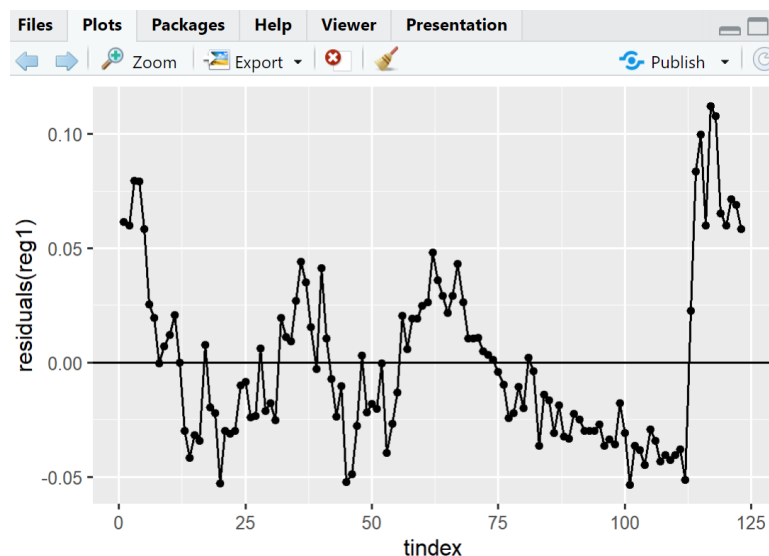
#Loading the tidyverse library
library(tidyverse)

#Reading in the ald_sales.csv data set
alc_dat <- read_csv("alc_sales.csv")

#Regression of log(alc_sales) on tindex and quarter dummies
reg1 <- lm(log(alc_sales) ~ tindex + q2 + q3 +q4, data = alc_dat)
summary(reg1)

#Plotting residuals across tindex
res_plot1 <- ggplot(alc_dat, aes(x = tindex, y = residuals(reg1))) +
  geom_line() +
  geom_point() +
  geom_hline(yintercept=0)
res_plot1
```

The first two code chunks above load the `tidyverse` library, and create the data frame `alc_dat`. Next, we run the regression, and show the regression results stored in `reg1`. The last chunk uses the `ggplot` function to plot the residuals across time (i.e., the variable `tindex`). Note that the part, `y = residuals(reg1)`, is using R's ability to nest functions, where the `residuals()` function is being used to specify what the y-axis values will be. Note that we have three 'geoms'. The first, `geom_line()`, specifies a line graph. The second, `geom_point()`, adds a point for each of the observations. And the third, `geom_hline(yintercept=0)`, plots a horizontal line at the value 0. Running this code produces the following plot,

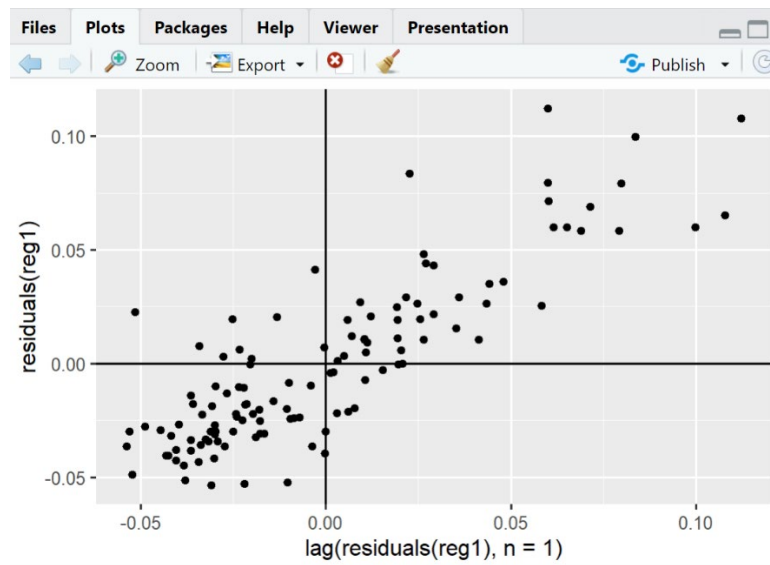


Viewing the graph, (which is the same as Figure 8.6a in *Regression Basics*), we see evidence of positive autocorrelation, with long strings of positive errors, then long strings of negative errors. In fact, from periods 82 to 112 we have thirty-one negative errors in a row – this is clearly *not* a random pattern.

An alternative graph plots the current period residuals against the previous period's residuals (see Figure 8.6b in *Regression Basics*). This can be accomplished by employing R's lag function, `lag()`. The following code shows how to nest this function within the `ggplot` function,

```
#Plotting residuals against lagged residuals
res_plot2 <- ggplot(alc_dat, aes(x = lag(residuals(reg1), n=1), y = residuals(reg1))) +
  geom_point() +
  geom_hline(yintercept=0) +
  geom_vline(xintercept=0)
res_plot2
```

Here we have removed `tindex` from the `aes()` function, and added `lag(residuals(reg1), n=1)`. This component is illustrating a double nesting in R with the `residuals()` function being nested within the `lag()` function, which in turn is nested within the `aes()` function. The `lag()` function takes the lagged value of an object (in this case, the `residuals(reg1)` object), where the component, `n=1`, specifies to take a one-period lag. The other changes are that we have removed the `geom_line()` function, and added `geom_vline(xintercept=0)`, which adds a vertical line at a value of 0. Adding this code to our script file and running it produces,



As is evident, the upward sloping nature of the dots suggests that we have positive autocorrelation.

### *Durbin-Watson Test of Autocorrelation*

As described in Chapter 8 in *Regression Basics*, the Durbin-Watson statistic is often used to detect autocorrelation. We can compute this statistic in R using the `car` package discussed earlier. Assuming this package has already been installed in R, the code is straight forward,

```
#Computing the Durbin-Watson statistic for our linear model
library(car)
durbinWatsonTest(reg1)
```

As seen above, we simply load the `car` package, and then call the `durbinWatsonTest()` function,<sup>22</sup> and supply the object for the test, which is our linear model object, `reg1`. Adding this code to our script file and running it give us the following output in the console,

```
> #Computing the Durbin-Watson statistic for our linear model
> library(car)
> durbinWatsonTest(reg1)
lag Autocorrelation D-W Statistic p-value
1      0.8388043      0.2802158      0
Alternative hypothesis: rho != 0
```

The output provides us with four measures. The `lag` value of 1 tells us that we are considering an AR(1) error structure (see Chapter 8 or *Regression Basics*). The next piece, `Autocorrelation`, is an estimate of  $\hat{\rho}$ , our coefficient of autocorrelation, given as 0.8388043. The third measure, `D-W Statistic`, is the computed Durbin-Watson statistic, reported as 0.2802158, which is a very low value, indicating likely positive autocorrelation. The fourth measure, `p-value`, reported as 0, is the p-value associate with a hypothesis test where the null hypothesis is,  $\rho = 0$ , i.e., that there is no autocorrelation. The alternative hypothesis is shown by the R output,

Alternative hypothesis:  $\rho \neq 0$ , where the symbol  $\neq$  is commonly used in programming to mean ‘not equal to’.<sup>23</sup> Thus, our conclusion from this test is that there is strong evidence of autocorrelation in the error structure.

---

<sup>22</sup> We can also call this function more succinctly with, `dwt()`.

### *Prais-Winsten Estimation*

One possible solution to the problem of autocorrelation is to use the Prais-Winsten (PW) estimator. This estimator uses an iterative approach to estimate the autocorrelation coefficient ( $\hat{\rho}$ ) and then using this estimate transforms the model, hopefully purging it of autocorrelation, and then using OLS on the transformed model. The PW estimation can be estimated in R using a package called `prais`. We begin by installing this package by typing into the console,<sup>24</sup>

```
install.packages("prais ")
```

Hitting the ‘Enter’ key installs the package (this needs to be done only once). Next we load the package with a `library()` function, and then provide the code for the `prais_winsten()` function which needs at least three pieces of information: a formula for the regression, a name of the data frame to use, and the name of an object in the data frame that is keeping track of time for each observation. In our case, we can use the following code,

```
#Performing a Prais-Winsten estimation using the prais() function
library(prais)
reg2 <- prais_winsten(log(alc_sales) ~ tindex + q2 + q3 + q4, data = alc_dat,
                     index = "tindex")
summary(reg2)
```

We see in the above code that our formula is just the log of alcohol sales model estimated earlier, `log(alc_sales) ~ tindex + q2 + q3 + q4, data = alc_dat`, which also identifies the data frame, `data = alc_dat`. We then indicate the object in the data frame that is our time

---

<sup>23</sup> As noted in Chapter 8 or *Regression Basics*, the distributional behavior of the Durbin-Watson statistic is not easy to specify, thus Durbin and Watson (1951) provide tables that can be used for inference tests. An alternative approach is to use a ‘bootstrap’ method to produce a p-value for the Durbin-Watson statistic, which is what is reported here, (see: “Bootstrap Tests for Autocorrelation,” by Jeong and Chung in *Computational Statistics & Data Analysis*, 38 (2001) 49–69.)

<sup>24</sup> Note that the `install.packages("prais")` command may not work on some versions of R. In this case, the `prais` package can be installed directly from GitHub by using the following commands:

```
install.packages("devtools")
devtools::install_github("franzmohr/prais")
```

variable, `index = "tindex"`. Adding the above code to our script file and running it give us the following output displayed in the console,

```
> #Perfroming a Prais-Winsten estimation using the prais() function
> library(prais)
> reg2 <- prais_winsten(log(alc_sales) ~ tindex + q2 + q3 + q4, data = alc_dat,
+                       index = "tindex")
Iteration 0: rho = 0
Iteration 1: rho = 0.856
Iteration 2: rho = 0.8604
Iteration 3: rho = 0.8606
Iteration 4: rho = 0.8606
Iteration 5: rho = 0.8606
> summary(reg2)

Call:
prais_winsten(formula = log(alc_sales) ~ tindex + q2 + q3 + q4,
              data = alc_dat, index = "tindex")

Residuals:
      Min       1Q   Median       3Q      Max
-0.06029 -0.03501 -0.01561  0.01483  0.10748

AR(1) coefficient rho after 5 iterations: 0.8606

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  8.377875    0.022691  369.22  <2e-16 ***
tindex        0.009997    0.000309   32.35  <2e-16 ***
q2           0.117632    0.003272   35.95  <2e-16 ***
q3           0.138353    0.003793   36.48  <2e-16 ***
q4           0.246376    0.003308   74.49  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.01956 on 118 degrees of freedom
Multiple R-squared:  0.9961,    Adjusted R-squared:  0.996
F-statistic: 7583 on 4 and 118 DF,  p-value: < 2.2e-16

Durbin-Watson statistic (original): 0.2802
Durbin-Watson statistic (transformed): 2.033
```

The `prais()` function shows us the iteration history as  $\hat{\rho}$  is being estimated, and the `summary()` function displays the PW regression results, including at the bottom of the table we see the original Durbin-Watson value, and the one after using  $\hat{\rho}$  to transform the model.

## 14. Studentized Residuals and Leverage

In Chapter 8 of *Regression Basics* there is a discussion on influential observations. These are observations that contain the ingredients for having a strong impact on the coefficients of a regression estimation. The two main ingredients are being a strong outlier and having strong



leverage. A measure called “studentized residuals” was used to detect strong outliers. This measure essentially looks for observations that have unusually large residuals ( $e_i$ 's). A studentized residual greater than 3 in absolute terms generally indicates a strong outlier.

The second ingredient for being an influential observation is having strong leverage, ( $lev_i$ ). As noted in Chapter 8, this essentially means that an observation's value for one or more of the independent ( $X_i$ ) variables is unusually large or small, compared to the other observations. In cases where  $lev_i$  is more than 3 times the mean value for leverage for all the observations, these may indicate strong leverage.

Computing studentized residuals and leverage values in R can be easily done. Returning to our nba\_sal project folder, we can use the following code in a new script file, nba4.R to compute these measures,

```
#Analysis of NBA Salary Determinants

#Loading the tidyverse library
library(tidyverse)

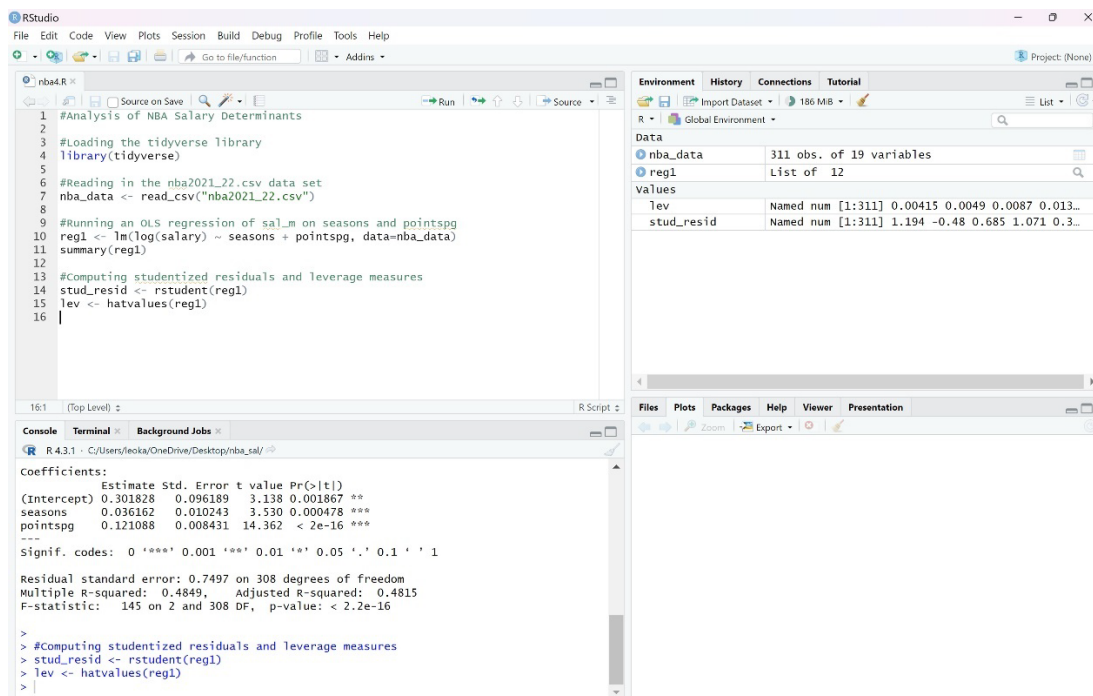
#Reading in the nba2021_22.csv data set
nba_data <- read_csv("nba2021_22.csv")

#Running an OLS regression of log(sal_m) on seasons and pointspg
reg1 <- lm(log(salary) ~ seasons + pointspg, data=nba_data)
summary(reg1)

#Computing studentized residuals and leverage measures
stud_resid <- rstudent(reg1)
lev <- hatvalues(reg1)
```

In the above code, after loading the data into a data frame, nba\_dat, and running our NBA salary regression, `log(salary) ~ seasons + pointspg, data=nba_data`, we assign the regression results to the object, reg1. The last code chunk assigns to the object stud\_resid the output of the

`rstudent()` function which computes the studentized residuals following a regression. We specified the object `reg1` for this function, thus the studentized residuals will be computed for our regression of `log(salary)` regressed on `seasons` and `pointspg`. In a similar way, the next line of code assigns to the object `lev` the output of the `hatvalues()` function which produces leverage measures.<sup>25</sup> Running the above code produces two new objects in the environment window, `stud_resid` and `lev`,



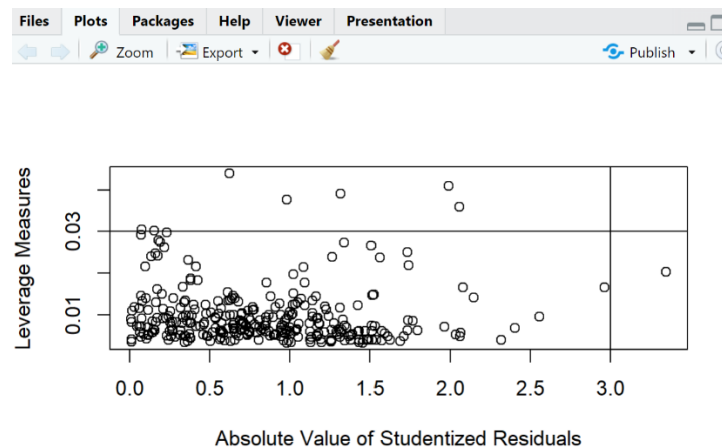
Recall from Chapter 8 or *Regression Basics* that an observation that has a leverage value greater than 3 times the mean for all leverage values, *and* a studentized residual greater than 3 in absolute terms, is potentially ‘influential’. Thus, we can create a plot that has the absolute value of the studentized residuals on the horizontal axis, and the leverage values on the vertical axis.

<sup>25</sup> Leverage measures are sometimes called “hat values” because they are derived from something called the “hat matrix” which is part of a matrix approach to least-squares regression, (a topic beyond the scope of *Regression Basics*).

The mean value for `lev` is about 0.00965, thus three times this value is approximately 0.03.<sup>26</sup> We can add a horizontal line on our plot at 0.03 to indicate observations with high leverage. In addition, we can add a vertical line on our plot at 3 to indicate observations with large absolute values for the studentized residuals. The following base R code will create the desired plot,

```
#Plotting absolute value of stud_resid and lev
abs_sr <- abs(stud_resid)
plot(abs_sr, lev,
      xlab = "Absolute Value of Studentized Residuals",
      ylab = "Leverage Measures")
abline(h=0.03)
abline(v=3)
```

The second line of code, `abs_sr <- abs(stud_resid)`, uses the `abs()` function to compute the absolute value of `stud_resid` and assign it to the object `abs_sr`. The next three lines are similar to our earlier `plot()` function example. The code `abline(h=0.03)` and `abline(v=3)` adds a horizontal line at 0.03, and a vertical line at 3, respectively. Adding the above code to our script file and running produce the following plot,



We can see in the above plot that no observations exceed both thresholds (i.e., no observations lie in the upper right corner of the plot), thus there do not seem to be any influential observations in this model.

<sup>26</sup> The mean value of `lev` can be found by typing `summary(lev)` into the console and hitting enter.

## 15. Producing Publication-Quality Tables in R

By now the reader should be aware of how powerful R is when it comes to carrying out empirical research. Throughout *Regression Basics* there are examples of statistical output produced by this and other programs, and the output shown are simply ‘screen shots’ of what is displayed by these programs on the computer screen. While easy to copy and paste, this kind of output is generally not appropriate for inclusion in empirical research manuscripts. Well-formatted tables with easy-to-read variable names and values should be employed. Fortunately, R can create such tables. There are multiple packages designed for this purpose, but I will focus on one that in my experience is easy to use, is highly customizable, and works very well. Below I will show how to produce a well-formatted table of summary statistics (also known as descriptive statistics), and a well-formatted table of regression results.

### *The modelsummary Package*

This package is available at CRAN, and can be installed in R by typing the following command in the R console and hitting ‘Enter’ (this needs to be done only once),

```
install.packages("modelsummary")
```

Once installed, the package can be loaded with a `library()` command. The package contains multiple components, and details about what can be done with this package are available here:

<https://modelsummary.com/>. We will focus on how to use it to make a table of descriptive statistics using the `datasummary_skim()` function, and a table of regression output using the `modelsummary()` function.

### *Summary Statistics*

The syntax for the `datasummary_skim()` function couldn't be much easier. Once a data frame (`df`) has been loaded into R, we simply load the `modelsummary` library and then type:

`datasummary_skim(df, ...options...)`. If we omit any options, then the summary statistics table is sent to the viewer. A better approach, however, is to use the option, `output =`, and provide a name and file type for the output table. The `datasummary_skim()` function can output tables to a variety of file types, including: HTML, LaTeX, Microsoft Word<sup>®</sup> and Powerpoint<sup>®</sup>, Text/Markdown, PDF, RTF, or Image files.

As an example, we can work with our `nba_sal` project folder and create a new R script file called `nba_tables.R`, with the following code,

```
#Using modelsummary to make summary statistics and regression tables

#Loading the tidyverse library
library(tidyverse)

#Loading the modelsummary library
library(modelsummary)

#Reading in the nba2021_22.csv data set
nba_data <- read_csv("nba2021_22.csv")

#Using datasummary_skim() to make a table of summary statistics
datasummary_skim(nba_data, output = "sum_stats.docx")
```

Running this code produces a MS Word document, saved in the current working directory, called “`sum_stats.docx`”, which looks like the following,

Unique (#)	Missing (%)	Mean	SD	Min	Median	Max
---------------	----------------	------	----	-----	--------	-----

	Unique (#)	Missing (%)	Mean	SD	Min	Median	Max
age	20	0	26.7	4.3	20.0	26.0	41.0
salary	223	0	9.8	10.6	1.0	5.0	44.1
weight	73	0	214.5	23.8	164.0	214.0	290.0
height	14	0	78.0	3.0	72.0	78.0	87.0
games	243	0	329.9	274.6	3.0	253.0	1310.0
minspg	290	0	22.6	7.6	3.5	23.2	38.0
orebsp	177	0	0.8	0.6	0.0	0.7	3.4
drebsp	224	0	3.0	1.4	0.4	2.9	8.6
trebsp	239	0	3.8	1.8	0.5	3.5	11.0
astspg	209	0	2.3	1.8	0.0	1.7	9.4
stlspg	158	0	0.8	0.4	0.0	0.7	2.1
blockspg	150	0	0.4	0.4	0.0	0.3	2.2
tovspg	198	0	1.3	0.8	0.0	1.1	4.2
foulspg	194	0	1.9	0.6	0.1	1.9	3.3
pointspg	281	0	10.2	5.4	0.8	9.2	26.9
seasons	18	0	6.0	4.4	1.0	5.0	18.0

Note that the above table is simply an MS Word table and is editable like any other, including font choice, deletion of rows/columns, and editing row/column names, and row/column sizes. An edited version of this table shown below illustrates some of these changes,

	Mean	SD	Min	Median	Max
age	26.7	4.3	20.0	26.0	41.0
salary (millions of \$'s)	9.8	10.6	1.0	5.0	44.1
weight (in pounds)	214.5	23.8	164.0	214.0	290.0
height (in inches)	78.0	3.0	72.0	78.0	87.0
career games played	329.9	274.6	3.0	253.0	1310.0
career points per game	10.2	5.4	0.8	9.2	26.9
seasons in the NBA	6.0	4.4	1.0	5.0	18.0

As noted earlier, the `datasummary_skim()` function is *highly* customizable, including adding a title to the table, and a footnote with notes about the table, (see:

<https://modelsummary.com/articles/datasummary.html>).

### *Regression Tables*

Tables showing regression output can vary in terms of layout and content. Estimates for models with just a few independent variables sometimes are written in the form of a function. As an example, suppose we estimate a simple regression with our NBA data set where the  $\log(\text{salary})$  is regressed on `seasons` and `pointspg`. We can present the results in the following equation,

$$\log(\widehat{\text{salary}}_i) = 0.302 + 0.036\text{seasons}_i + 0.121\text{pointspg}_i$$

(0.096) (0.010) (0.008)

n = 311      R<sup>2</sup> = 0.485

We see in the above equation the estimated intercept and coefficients to the independent variables are reported, with their standard errors in parentheses below each.

When more complicated models are estimated with many independent variables, the equation format does not work well. A common format in this case is to have columns of output that contain estimated coefficients, standard errors (typically in parentheses) below each coefficient, and some symbol, (often an asterisk or ‘star’) indicating if an estimated coefficient is statistically different from zero. We can add the following code to our script file, where the `modelsummary()` function will be used to create a table of output:

```
#Estimating a regression of log(salary) on seasons and pointspg
reg1 <- lm(log(salary) ~ seasons + pointspg, data = nba_data)

#Using the modelsummary() function to make a table of output
modelsummary(reg1, output = "reg1.docx", stars = TRUE)
```

Here we see the code used `modelsummary()`, we simply provide the object’s name that contains the regression results (`reg1`), an output command, and any other options. In the above example, we added the option, `stars = TRUE`, which tells the function to include stars (i.e., asterisks) to indicate statistical significance, (the default for this option is to not include stars). Adding and running this code, we get an MS Word document titled “reg1.docx” saved in our current working directory. Below is the resulting table,<sup>27</sup>

---

<sup>27</sup> Sometimes the output table needs to be widened so that all the variable labels are on the same line.



	(1)
(Intercept)	0.302** (0.096)
seasons	0.036*** (0.010)
pointspg	0.121*** (0.008)
Num.Obs.	311
R2	0.485
R2 Adj.	0.482
AIC	1798.6
BIC	1813.6
Log.Lik.	-350.185
F	144.960
RMSE	0.75
+ p < 0.1, * p < 0.05, ** p < 0.01, *** p < 0.001	

As with the previous example, the above table is an MS Word table and is editable. Changing the font and removing some of the rows, and making some cosmetic changes we have,<sup>28</sup>

<sup>28</sup> The `modelsummary()` function provides a variety of diagnostic measures, some of which were not covered in *Regression Basics*, such as the AIC, BIC, etc.

	(1)
(Intercept)	0.302** (0.096)
seasons in the NBA	0.036*** (0.010)
career points per game	0.121*** (0.008)
Num.Obs.	311
R <sup>2</sup>	0.485
R <sup>2</sup> Adj.	0.482
+ p < 0.1, * p < 0.05, ** p < 0.01, *** p < 0.001 Standard errors in parentheses.	

We see at the bottom of the table a key for the meaning of the stars, with coefficients to seasons (‘seasons in the NBA’) and pointspg (‘career points per game’) being statistically different from zero at less than 0.001 (well below our benchmark 0.05 level).

Up to this point, we have been considering a very simple regression model with just two independent variables. Often, we will be considering more complex models, and furthermore we may start with a simple version of a model, and then ‘build out’ a more comprehensive model. In this case, we can create a regression table with multiple columns of output, one for each model estimated. This is easily done with `modelsummary()` by incorporating a `list()` function. Consider the following code,

```
#Incorporating the list() function for multiple models
models <- list(
  "reg1" = lm(log(salary) ~ seasons + pointspg, data = nba_data),
  "reg2" = lm(log(salary) ~ seasons + pointspg + height + weight,
              data = nba_data),
  "reg3" = lm(log(salary) ~ seasons + I(seasons^2) + pointspg + height + weight,
              data = nba_data)
)

modelsummary(models, stars = TRUE, output = "models.docx")
```

The above code assigns to the object `models` the content of the `list()` function. In the list we have created three objects: "reg1", "reg2", and "reg3". Each of these objects, in turn, are assigned to the output created by the three `lm()` functions. Note that the linear models start with our basic model ("reg1"), then we add to that two variables, height and weight in the second model ("reg2"), and finally the third model ("reg3"), adds the squared value of seasons by including the code, `I(seasons^2)`. Note that the `I()` function tells R to accept the results inside the parentheses “as-is”. Wrapping `seasons^2` inside `I()` will allow it to be evaluated in the expected way (as seasons squared) and it will be included as a separate variable. Finally, we call the `modelsummary()` function and pass into it the objects from the `list()` function.

Alternatively, we could run the three regressions and then nest the `list()` function in the `modelsummary()` function,

```
#Nesting the list() function in the modelsummary() function for multiple models

reg1 <- lm(log(salary) ~ seasons + pointspg, data = nba_data)
reg2 <- lm(log(salary) ~ seasons + pointspg + height + weight,
           data = nba_data)
reg3 <- lm(log(salary) ~ seasons + I(seasons^2) + pointspg + height + weight,
           data = nba_data)

modelsummary(list(reg1, reg2, reg3), stars = TRUE, output = "models.docx")
```

Adding either of these code chunks to our script file and running it produces a new table, “models.docx”, which is saved in the working directory. The table looks like this,

	reg1	reg2	reg3
(Intercept)	0.302** (0.096)	-0.384 (1.330)	-0.503 (1.250)
seasons	0.036*** (0.010)	0.037*** (0.010)	0.240*** (0.033)
pointspg	0.121*** (0.008)	0.122*** (0.009)	0.116*** (0.008)
height		0.012 (0.021)	0.007 (0.020)
weight		-0.001 (0.003)	-0.001 (0.003)
l(seasons^2)			-0.013*** (0.002)
Num.Obs.	311	311	311
R2	0.485	0.485	0.547
R2 Adj.	0.482	0.479	0.539
AIC	1798.6	1802.3	1764.8
BIC	1813.6	1824.7	1791.0
Log.Lik.	-350.185	-350.022	-330.261
F	144.960	72.165	73.607
RMSE	0.75	0.75	0.70

+ p < 0.1, \* p < 0.05, \*\* p < 0.01, \*\*\* p < 0.001

As before, we can edit this table to match the format of our manuscript. In addition, it may be desirable to replace some of the variable names, which can sometimes appear a bit cryptic to readers, with more descriptive versions,

	reg1	reg2	reg3
(Intercept)	0.302** (0.096)	-0.384 (1.330)	-0.503 (1.250)
seasons in the NBA	0.036*** (0.010)	0.037*** (0.010)	0.240*** (0.033)
career points per game	0.121*** (0.008)	0.122*** (0.009)	0.116*** (0.008)
height (in inches)		0.012 (0.021)	0.007 (0.020)
weight (in pounds)		-0.001 (0.003)	-0.001 (0.003)
seasons squared			-0.013*** (0.002)
Num.Obs.	311	311	311
R <sup>2</sup>	0.485	0.485	0.547
R <sup>2</sup> Adj.	0.482	0.479	0.539

+ p < 0.1, \* p < 0.05, \*\* p < 0.01, \*\*\* p < 0.001

Standard errors in parentheses.

The above table is an example of one that can be found in professional publications. As noted earlier, `modelsummary` is *highly* customizable and the reader is encouraged to visit the following website for details: <https://modelsummary.com/>.