Linear Regression - R

James ‘Mac’ Stewart

2023-03-02

Necessary Packages

```
library(tidyverse)
library(tidymodels)
library(GGally)
library(reticulate)
```

Importing Data

The following code chunk was taken from tidy Tuesday (https://github.com/rfordatascience/tidytuesday/blob/master/data/2022/2022-01-25/readme.md) where we pull in data. I chose to use their data cleaning code as this is not the main goal of the project.

```
ratings <- readr::read_csv('https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2022/2022-01-25/ratings.csv')
details <- readr::read_csv('https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2022/2022-01-25/details.csv')
```

Begin Language Comparison

This section will go through the process of creating linear regression models. Because this is the first model that we are going through in the project, I will go through the data cleaning in both R and Python to show how easy it is for both. Each subsection will be put into two parts. First, you will see it done in R and then you will see it done in python.

As noted, the python parts will utilize the reticulate package.

Cleaning Data - R

After pulling in the two included data sets, I then want to combine them into a single flat file. I am pulling in only the average variable from the ratings data set. I will also be dropping a few of the variables included in the details tab as they will not be used for the scope of this project.

```
ratings_small <- ratings %>% select(id, name, average)
details_small <- details %>% select(-c(num, description, wishing, wanting, trading, owned, boardgameexpansion, boardgameimplementation))
```

This function allows us to pull the first part of each of the listed variables like boardgamecategory. While the full list would be more beneficial, the first listed item in these categories are enough for the scope of this project.

```
get_first_part <- function(word){
  list = str_split(word, "'")
  first_word = list[[1]][2]
  return(first_word)
}
```

We will then apply this function to the variables boardgamecategory, boardgamemechanic, boardgamefamily, boardgamedesigner, boardgameartist, and boardgamepublisher. It is important that, if run multiple times, that you apply the function on the original data set columns. Otherwise, it will not find error out on the next run.

```
details_small$boardgamecategory <- details %>%
  select(boardgamecategory) %>%
  apply(1, get_first_part)

details_small$boardgamemechanic <- details %>%
  select(boardgamemechanic) %>%
  apply(1, get_first_part)

details_small$boardgamefamily <- details %>%
  select(boardgamefamily) %>%
  apply(1, get_first_part)

details_small$boardgamedesigner <- details %>%
  select(boardgamedesigner) %>%
  apply(1, get_first_part)

details_small$boardgameartist <- details %>%
  select(boardgameartist) %>%
  apply(1, get_first_part)

details_small$boardgamepublisher <- details %>%
  select(boardgamepublisher) %>%
  apply(1, get_first_part)
```

Finally, we need to join the two tables together to create the final data set.

```
final_data <- merge(details_small, ratings_small, "id") %>%
  select(-primary, -id)
```

Cleaning Data - Python

```
# importing padas package
import pandas  as pd

# getting the datasets into pandas dataframes and selecting the specific variables
ratings = pd.DataFrame(r.ratings) # you use 'r.' to access objects held in the R working library
ratings_small = ratings[['id', 'name', 'average']]

details = pd.DataFrame(r.details)
details_small = details.drop(['num', 'description', 'wishing', 'wanting', 'trading', 'owned', 'b
oardgameexpansion', 'boardgameimplementation'], axis = 1)
```

For my first time using reticulate and access different objects, it seems to be pretty intuitive. I used an article (https://rstudio.github.io/reticulate/) on the rstudio github page to start with the basics.

In terms of comparing, I personally like working in pandas and the way they access/modify data frames in terms like this. However, I think the tidy data frame pipelines are going to be more efficient and intuitive which we will see later.

```
def get_first_part(word):
  words = word.replace("[","").split(",")
  return words[0]
```

In terms of making basic functions, python seems to be more intuitive. The actual text altering seemed relatively the same, although python views strings as a list which could make it slightly easier for some more complex modifications.

```
details_small['boardgamecategory'] = [get_first_part(x) for x in details_small['boardgamecategor
y']]
details_small['boardgamemechanic'] = [get_first_part(x) for x in details_small['boardgamemechani
c']]
details_small['boardgamefamily'] = [get_first_part(x) for x in details_small['boardgamefamily']]
details_small['boardgamedesigner'] = [get_first_part(x) for x in details_small['boardgamedesigne
r']]
details_small['boardgameartist'] = [get_first_part(x) for x in details_small['boardgameartist']]
details_small['boardgamepublisher'] = [get_first_part(x) for x in details_small['boardgamepublis
her']]
```

I think that python's list comprehension is really neat and has a lot more use than the functions we have here. However, with how simplistic this type of function is, I think R dplyr's apply() function is probably better.

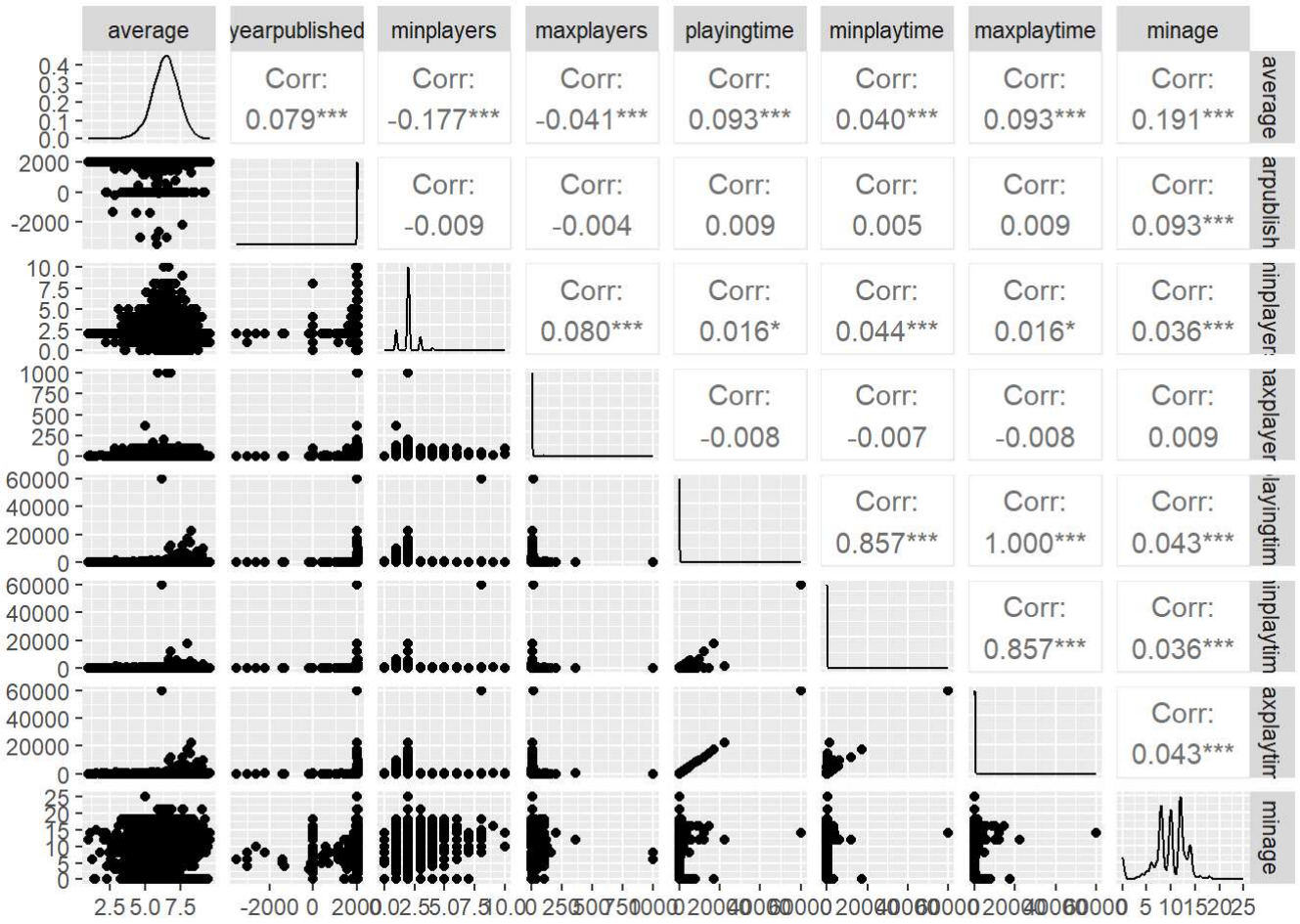
```
final_data = details_small.merge(ratings_small, how = 'left', left_on = 'id', right_on = 'id').d
rop(['id', 'primary'], axis = 1)
```

In this simplistic example, I don't think there is much of a difference between python and r when it comes to merging dataframes. You have to supply more information with pandas merge, but not enough to where it makes a difference.

Graphing the Variables - R

We will then graph each of the independent variables against the dependent variable. This will allow us to see which ones might have a correlation which will aide in choosing which factors to include in our analysis. Due to the large number of possibilities for the character variables, I will only be graphing the relationships with the numeric variables.

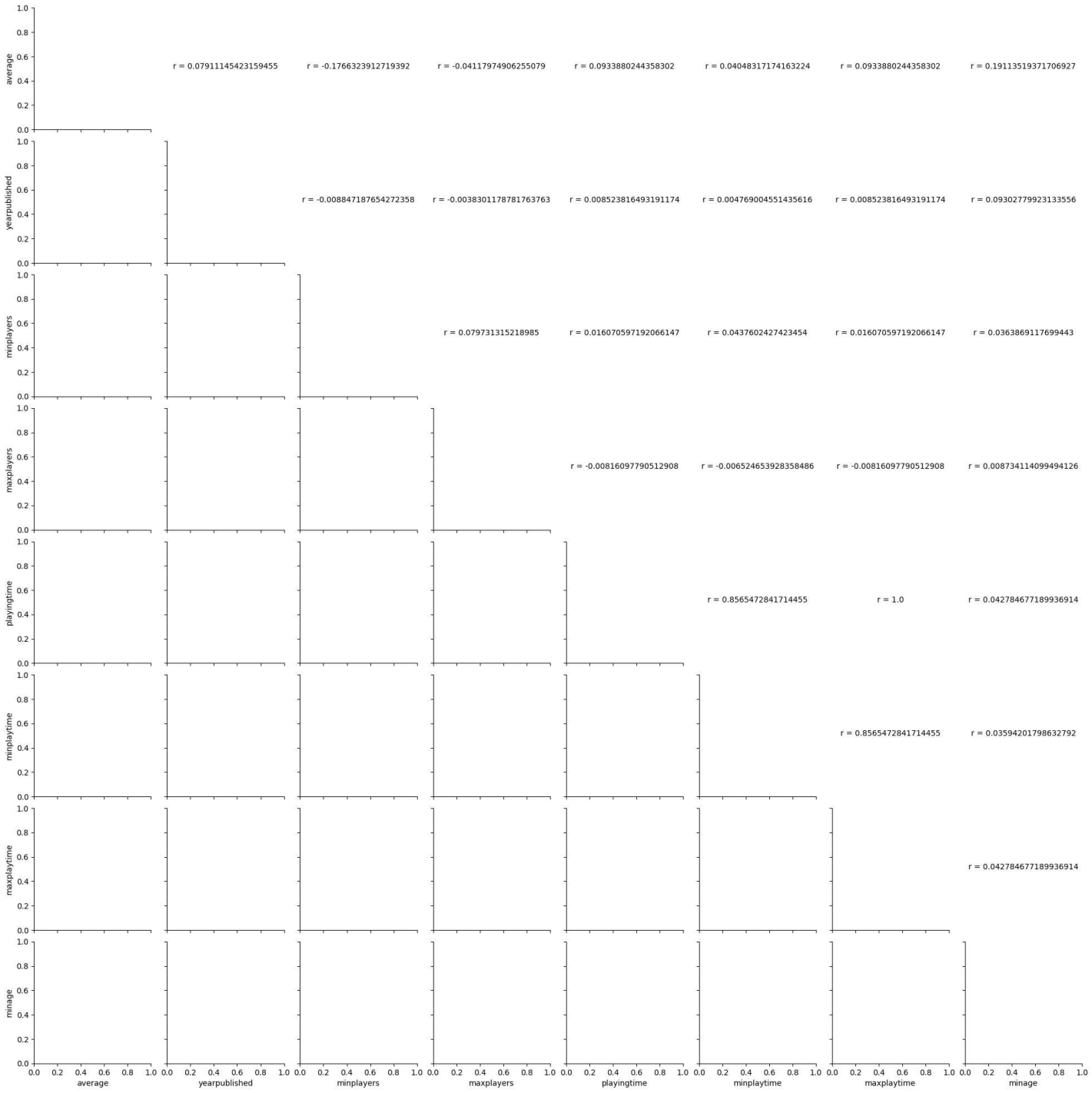
```
final_data %>%
  select(average, yearpublished, minplayers, maxplayers, playingtime, minplaytime, maxplaytime,
minage) %>%
  ggpairs()
```



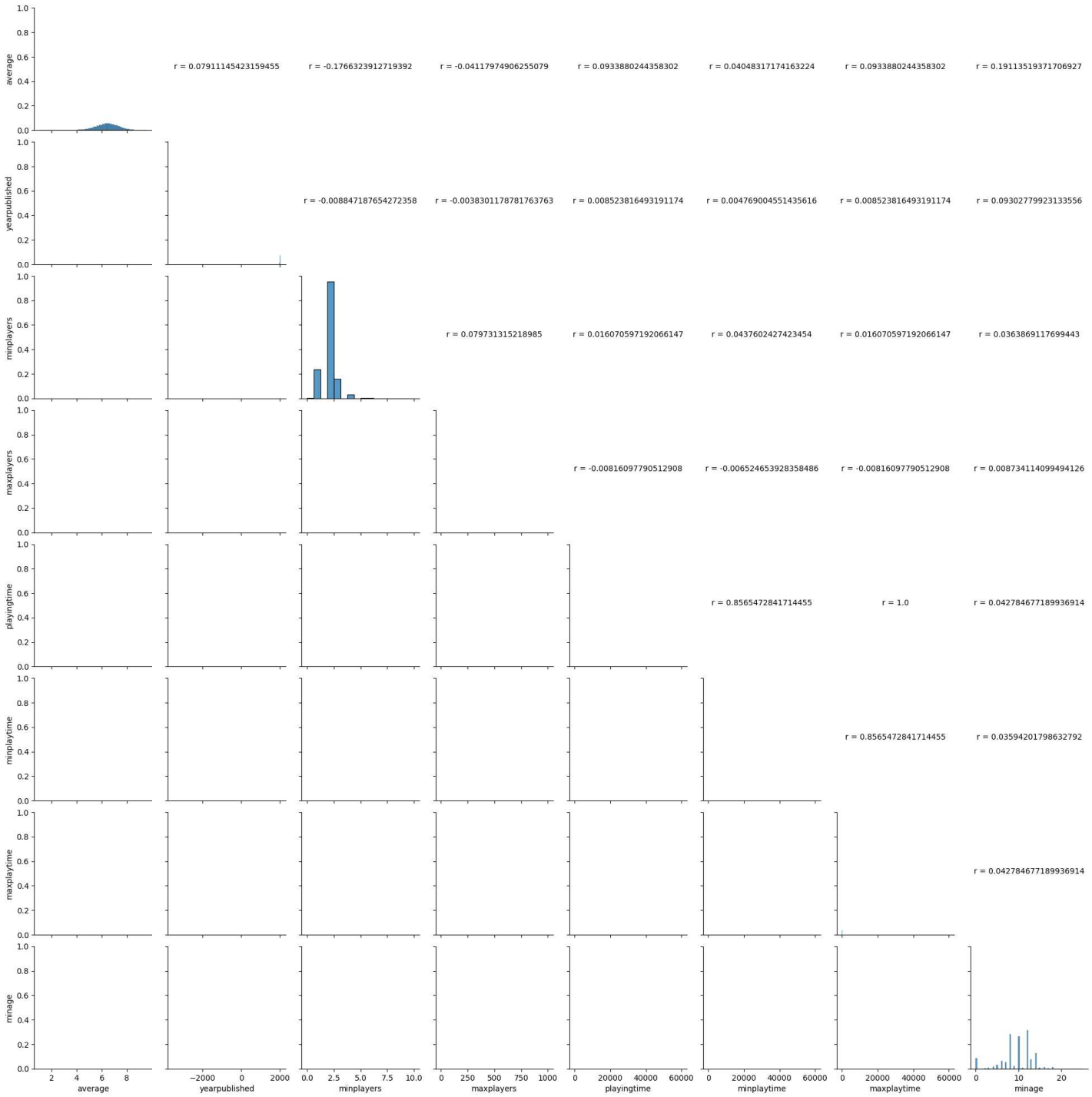
Graphing the Variables - Python

I was a little worried about this step. However a stack overflow post (https://stackoverflow.com/questions/68967458/is-there-is-an-equivalent-of-rs-ggallyggpairs-function-in-python) recommended using Seaborn's pairgrid package.

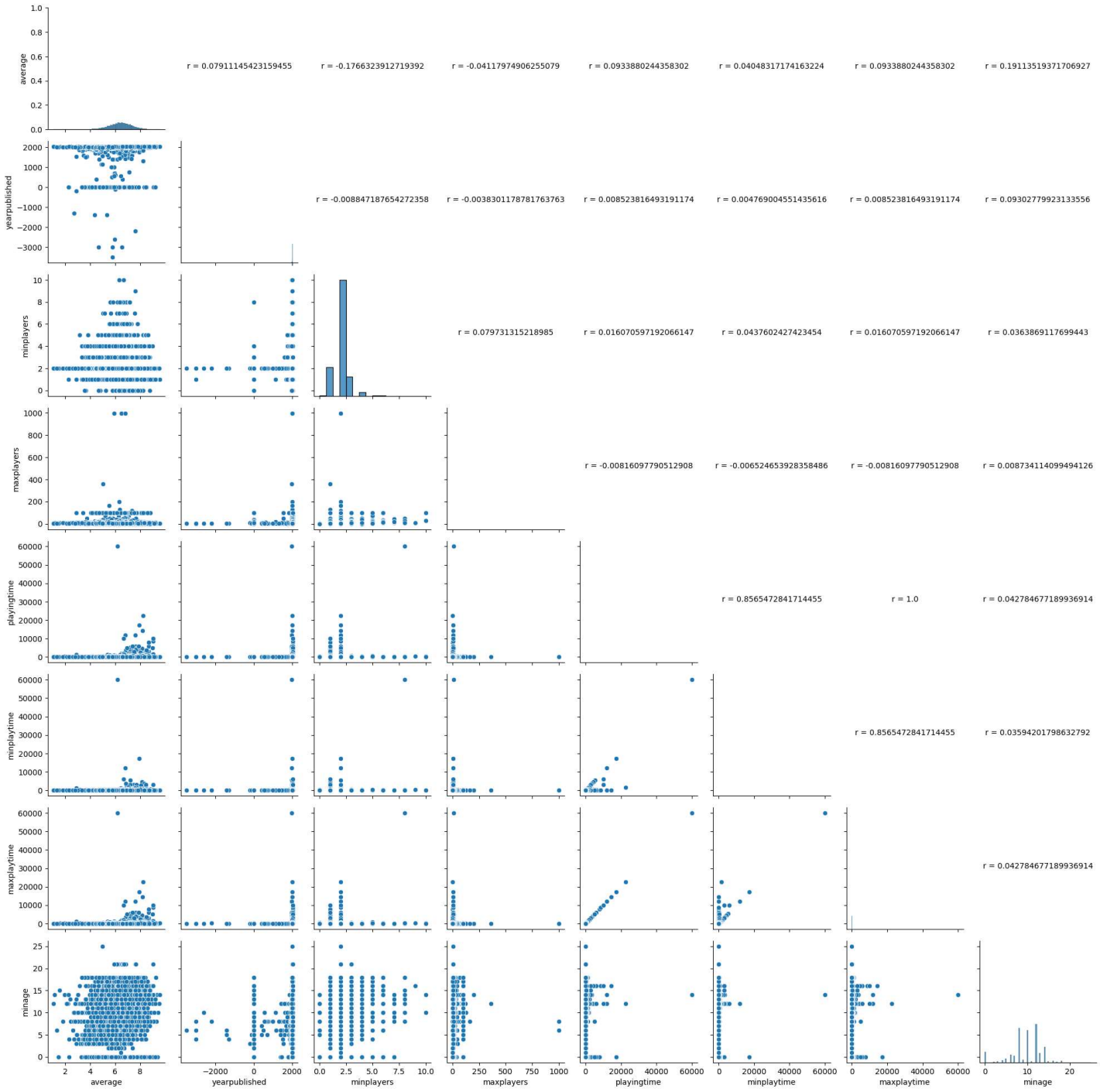
```
# reaidng packages
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import pearsonr

# creating a function to get correlation
def correlation(x,y, **kwargs):
    ax = plt.gca()
    r, p = pearsonr(x, y)
    ax.annotate('r = {}'.format(r),
                xy=(0.5, 0.5), #setting the dimension of the output
                xycoords='axes fraction', #tells the value to print based on the coordinates of
the box
                ha='center') #specifying horizontal alignment
    ax.set_axis_off()

# creating the pairgrid for all of the following variables
g = sns.PairGrid(final_data[['average', 'yearpublished', 'minplayers', 'maxplayers', 'playingtim
e', 'minplaytime', 'maxplaytime', 'minage']])
g.map_upper(correlation) #setting the top boxes to print out the correlation created from our fu
nction
```




```
g.map_diag(sns.histplot) # setting the boxes on the diagonal to be histograms
```




```
g.map_lower(sns.scatterplot) #setting the bottom boxes to be scatterplots
```




```
fig = g.fig #saving the output as a figure so it can be saved

fig.savefig("./pair_grid.png") #saving the figure. This is assuming the current working director
y is the linear-regression folder
```

There are a couple of notes. First off, python's version of the ggally:ggpair() with seaborn's parigrid is a lot more complicated. However, I can see that there are a lot of options to modify different aspects of the results. It required consulting different posts here (https://stackoverflow.com/questions/66893103/pairgrid-use-upper-triangle-for-correlations-access-full-data) and here (https://stackoverflow.com/questions/32244753/how-to-save-a-seaborn-plot-into-a-file) to reproduce what is almost immediate for the r package. However, the output in python is drastically better in quality. So if a more in-depth look at the visuals is required it might be worth using python's version. But, for any quick look, r's version will definitely be preferred.

Simple Linear Regression - R

It appears that minage has the greatest (negative) correlation with the average rating score. Because of this, we will use this variable for our simple linear regression.

```
slrm1 <- lm(average ~ minage, data = final_data)

tidy(slrm1)
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic   p.value
##   <chr>          <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)   5.95     0.0175      340.  0
## 2 minage        0.0488   0.00170      28.6 4.55e-177
```

```
glance(slrm1)
```

```
## # A tibble: 1 x 12
##   r.squared adj.r.~1 sigma stati~2   p.value    df  logLik    AIC    BIC devia~3
##       <dbl>    <dbl> <dbl>   <dbl>     <dbl> <dbl>   <dbl>  <dbl>  <dbl>   <dbl>
## 1    0.0365   0.0365 0.912    820. 4.55e-177     1 -28705. 57416. 57440.  17999.
## # ... with 2 more variables: df.residual <int>, nobs <int>, and abbreviated
## #   variable names 1: adj.r.squared, 2: statistic, 3: deviance
```

With an r^2 value of ~0.037, a simple linear regression does not seem to be appropriate to model the relationship between the independent and dependent variables. Because of this, a multiple linear regression will be created.

Simple Linear Regression - Python