**R vs Python for Data Analysis — An Objective Comparison**

*Published: October 21, 2020*

**R vs Python: Importing a CSV**

**R**

library(readr)

ba <- read\_csv("nba\_2013.csv")

**Python**

import pandas

ba = pandas.read\_csv("nba\_2013.csv")

[Dataframes](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html) are available in both R and Python — they are two-dimensional arrays (matrices) where each column can be of a different datatype. You can think of them as being like the programming version of a data table or a spreadsheet. At the end of this step, the CSV file has been loaded by both languages into a dataframe.

**Finding the number of rows**

**R**

dim(nba)

[1] 481 31

/code>

**Python**

nba.shape

(481, 31)

/code>

The output above tells us that this data set has 481 rows and 31 columns.

**Inspecting the first row of the data**

**R**

head(nba, 1)

player pos age bref\_team\_id

Quincy Acy SF 23 TOT[output truncated]

/code>

**Python**

nba.head(1)

player pos age bref\_team\_id

Quincy Acy SF 23 TOT[output truncated]

/code>

It’s worth noting that Python is more object-oriented here — head is a method on the dataframe object, whereas R has a separate head function.

This is a common theme we’ll see as we start to do analysis with these languages. Python is more object-oriented, and R is more functional.

**R vs Python: Finding Averages for Each Statistic**

The columns, as we can see, have names like fg (field goals made), and ast (assists). These are the season-long statistics and our data set tracks them for each row (each row represents an individual player).

If you’d like a fuller explanation of all the stats, look [here](https://stats.nba.com/help/glossary). Let’s take a look at how R and Python handle summary statistics by finding the average values for each stat in the data:

**R**

library(purrr)

ibrary(dplyr)

ba %>%

select\_if(is.numeric) %>%

map\_dbl(mean, na.rm = TRUE)

player NA

os NA

ge 26.5093555093555

ref\_team\_id NA

output truncated]

**Python**

nba.mean()

age 26.509356

53.253638

s 25.571726

output truncated]

/code>

Now we can see some major differences in the approaches taken by R vs Python.

In both, we’re applying a function across the dataframe columns. In Python, using the mean method on a dataframe will find the mean of each column by default.

In R, it’s a little more complicated. We can use functions from two popular packages to select the columns we want to average and apply the mean function to them. The %>% operator, referred to as “the pipe”, passes output of one function as input to the next.  
Taking the mean of string values (in other words, text data that cannot be averaged) will just result in NA — not available. We can take the mean of only the numeric columns by using select\_if.

However, we do need to ignore NA values when we take the mean (requiring us to pass na.rm=TRUE into the mean function). If we don’t, we end up with NA for the mean of columns like x3p.. This column is three point percentage. Some players didn’t take three point shots, so their percentage is missing. If we try the mean function in R, we get NA as a response, unless we specify na.rm=TRUE, which ignores NA values when taking the mean.

In contrast, the .mean() method in Python already ignores these values by default.

**Making Pairwise Scatterplots**

One common way to explore a data set is to see how different columns correlate to others. Let’s compare the ast, fg, and trb columns.

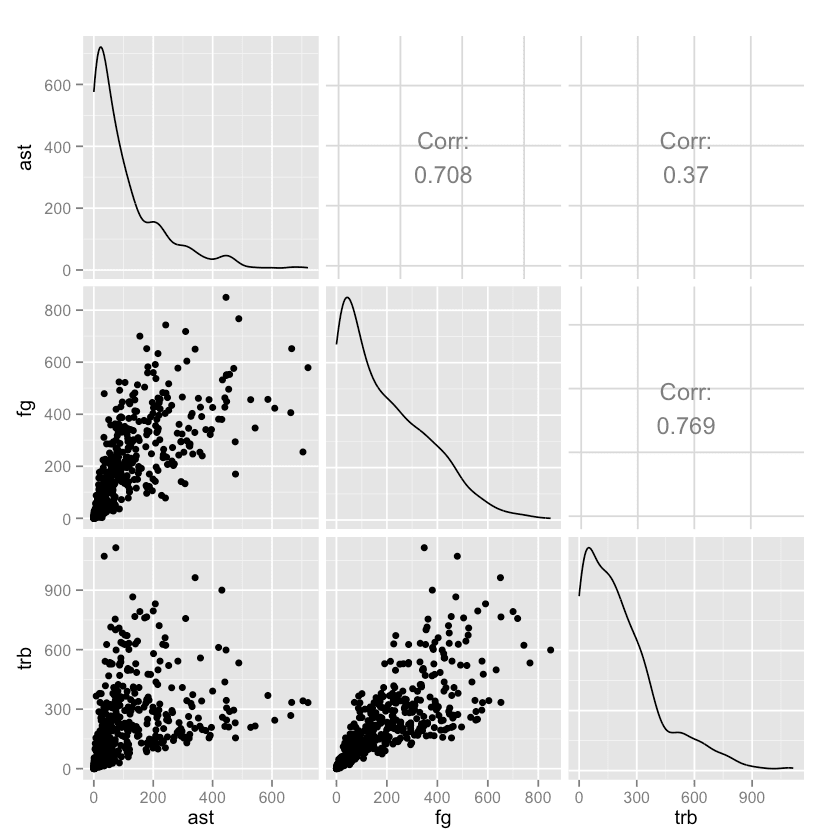
**R**

library(GGally)

ba %>%

elect(ast, fg, trb) %>%

gpairs()



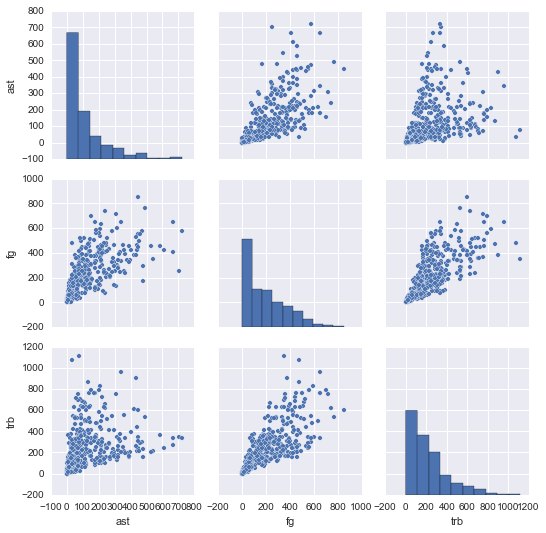
**Python**

import seaborn as sns

span class="token keyword">import matplotlib.pyplot as plt

ns.pairplot(nba[["ast", "fg", "trb"]])

lt.show()



In the end, both languages produce very similar plots. But in the code, we can see how the R data science ecosystem has many smaller packages ([GGally](https://cran.r-project.org/web/packages/GGally/index.html) is a helper package for [ggplot2](https://ggplot2.tidyverse.org/), the most-used R plotting package), and more visualization packages in general.

In Python, [matplotlib](https://matplotlib.org/) is the primary plotting package, and [seaborn](https://seaborn.pydata.org/) is a widely used layer over matplotlib.

With [visualization in Python](https://www.dataquest.io/path/data-analysis-and-visualization-with-python/), there is generally one main way to do something, whereas in R, there are many packages supporting different methods of doing things (there are at least a half-dozen packages to make pair plots, for instance).

Again, neither approach is “better”, but R may offer more flexibility just in terms of being able to pick and choose the package that works best for you.

**Making Clusters of the Players**

Another good way to explore this kind of data is to generate cluster plots. These will show which players are most similar.

(For now, we’re just going to make the clusters; we’ll plot them visually in the next step.)

**R**

library(cluster)

et.seed(1)

sGoodCol <- function(col){

sum(is.na(col)) == 0 && is.numeric(col)

span class="token punctuation">}

oodCols <- sapply(nba, isGoodCol)

lusters <- kmeans(nba[,goodCols], centers=5)

abels <- clusters$cluster

**Python**

from sklearn.cluster import KMeans

means\_model = KMeans(n\_clusters=5, random\_state=1)

ood\_columns = nba.\_get\_numeric\_data().dropna(axis=1)

means\_model.fit(good\_columns)

abels = kmeans\_model.labels\_

In order to cluster properly, we need to remove any non-numeric columns and columns with missing values (NA, Nan, etc).

In R, we do this by applying a function across each column, and removing the column if it has any missing values or isn’t numeric. We then use the [cluster](https://cran.r-project.org/web/packages/cluster/index.html) package to perform [k-means](https://en.wikipedia.org/wiki/K-means_clustering) and find 5 clusters in our data. We set a random seed using set.seed to be able to reproduce our results.

In Python, we use the main Python machine learning package, [scikit-learn](https://scikit-learn.org/), to fit a k-means clustering model and get our cluster labels. We perform very similar methods to prepare the data that we used in R, except we use the get\_numeric\_data and dropna methods to remove non-numeric columns and columns with missing values.

**Plotting Players by Cluster**

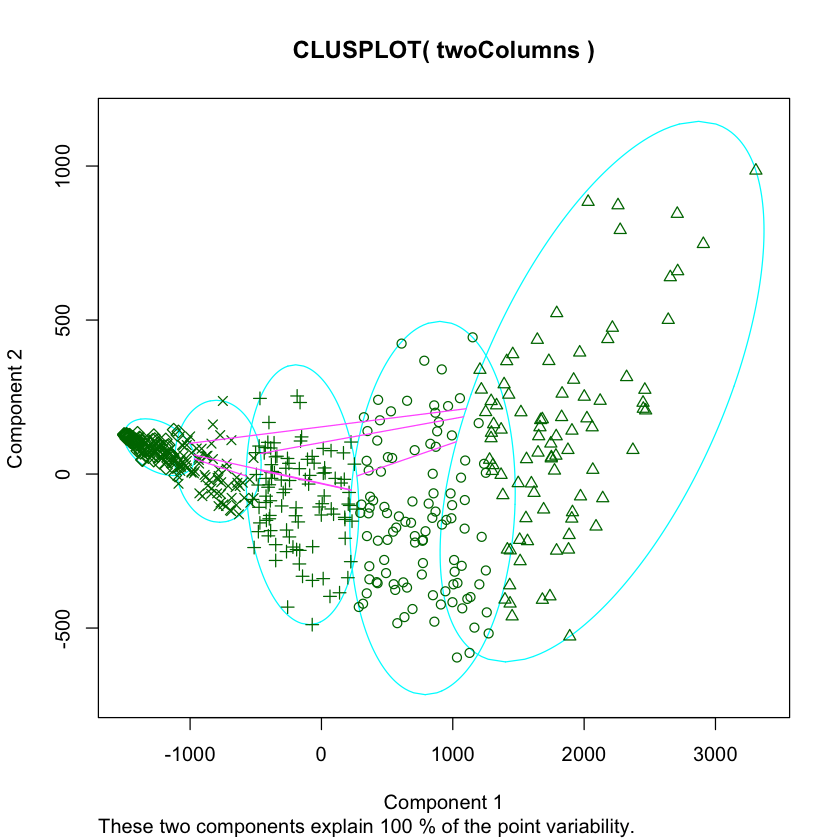
We can now plot out the players by cluster to discover patterns. One way to do this is to first use [PCA](https://en.wikipedia.org/wiki/Principal_component_analysis) to make our data  two-dimensional, then plot it, and shade each point according to cluster association.

**R**

nba2d <- prcomp(nba[,goodCols], center=TRUE)

woColumns <- nba2d$x[,1:2]

lusplot(twoColumns, labels)



**Python**

from sklearn.decomposition import PCA

ca\_2 = PCA(2)

lot\_columns = pca\_2.fit\_transform(good\_columns)

lt.scatter(x=plot\_columns[:,0], y=plot\_columns[:,1], c=labels)

lt.show()

Chart, scatter chart

Description automatically generated

Above, we made a scatter plot of our data, and shaded or changed the icon of each data point according to its cluster.

In R, we used the clusplot function, which is part of the cluster library. We performed PCA via the pccomp function that is built into R.

With Python, we used the PCA class in the scikit-learn library. We used matplotlib to create the plot.

Once again, we can see that while both languages take slightly different approaches, the final result and the amount of code required to get it is pretty similar.

**Splitting Data into Training and Testing Sets**

If we want to use R or Python for supervised machine learning, it’s a good idea to split the data into training and testing sets so we don’t [overfit](https://en.wikipedia.org/wiki/Overfitting).

Let’s compare how each language handles this common machine learning task:

**R**

trainRowCount <- floor(0.8 \* nrow(nba))

et.seed(1)

rainIndex <- sample(1:nrow(nba), trainRowCount)

rain <- nba[trainIndex,]

est <- nba[-trainIndex,]

**Python**

train = nba.sample(frac=0.8, random\_state=1)

est = nba.loc[~nba.index.isin(train.index)]

Comparing Python vs R, we can see that R has more data analysis capability built-in, like floor, sample, and set.seed, whereas these in Python these are called via packages (math.floor, random.sample, random.seed).

In Python, a recent version of pandas came with a sample method that returns a certain proportion of rows randomly sampled from a source dataframe — this makes the code much more concise.

In R, there are packages to make sampling simpler, but they aren’t much more concise than using the built-in sample function. In both cases, we set a random seed to make the results reproducible.

**R vs Python: Univariate Linear Regression**

Continuing with common machine learning tasks, let’s say we want to predict number of assists per player from field goals made per player:

**R**

fit <- lm(ast ~ fg, data=train)

redictions <- predict(fit, test)

**Python**

from sklearn.linear\_model import LinearRegression

r = LinearRegression()

r.fit(train[["fg"]], train["ast"])

redictions = lr.predict(test[["fg"]])

Python was a bit more concise in our previous step, but now R is more concise here!

Python’s Scikit-learn package has a linear regression model that we can fit and generate predictions from.

R relies on the built-in lm and predict functions. predict will behave differently depending on the kind of fitted model that is passed into it — it can be used with a variety of fitted models.

**Calculating Summary Statistics for the Model**

Another common machine learning task:

**R**

summary(fit)

Call:

m(formula = ast ~ fg, data = train)

esiduals:Min 1Q Median 3Q Max

228.26 -35.38 -11.45 11.99 559.61

output truncated]

**Python**

import statsmodels.formula.api as sm

odel = sm.ols(formula='ast ~ fga', data=train)

itted = model.fit()

itted.summary()

Dep. Variable: ast

-squared: 0.568

odel: OLS

dj. R-squared: 0.567

output truncated]

As we can see above, we’ll need to do a bit more in Python than in R if we want to get summary statistics about the fit, like [r-squared value](https://en.wikipedia.org/wiki/Coefficient_of_determination).

With R, we can use the built-in summary function to get information on the model immediately. With Python, we need to use the [statsmodels](https://statsmodels.sourceforge.net/) package, which enables many statistical methods to be used in Python.

We get similar results, although generally it’s a bit harder to do statistical analysis in Python, and some statistical methods that exist in R don’t exist in Python.

**Fit a random forest model**

Our linear regression worked well in the single variable case, but let’s say we suspect there may be [nonlinearities](https://en.wikipedia.org/wiki/Nonlinear_system) in the data. Thus, we want to fit a [random forest](https://en.wikipedia.org/wiki/Random_forest) model.

Here’s how we might do that in each language:

**R**

library(randomForest)

redictorColumns <- c("age", "mp", "fg", "trb", "stl", "blk")

f <- randomForest(train[predictorColumns], train$ast, ntree=100)

redictions <- predict(rf, test[predictorColumns])

**Python**

from sklearn.ensemble import RandomForestRegressor

redictor\_columns = ["age", "mp", "fg", "trb", "stl", "blk"]

f = RandomForestRegressor(n\_estimators=100, min\_samples\_leaf=3)

f.fit(train[predictor\_columns], train["ast"])

redictions = rf.predict(test[predictor\_columns])

The main difference here is that we needed to use the randomForest library in R to use the algorithm, whereas this is already built in to scikit-learn in Python.

Scikit-learn has a unified interface for working with many different machine learning algorithms in Python. There’s usually only one main implementation of each algorithm.

With R, there are many smaller packages containing individual algorithms, often with inconsistent ways to access them. This results in a greater diversity of algorithms (many have several implementations, and some are fresh out of research labs), but with a bit of a usability hit.

In other words, Python may be easier to use here, but R may be more flexible.

**Calculating Error**

Now that we’ve fit two models, let’s calculate error in R and Python. We’ll use [MSE](https://en.wikipedia.org/wiki/Mean_squared_error).

**R**

mean((test["ast"] - predictions)^2)

4573.86778567462

**Python**

from sklearn.metrics import mean\_squared\_error

an\_squared\_error(test["ast"], predictions)

4166.9202475632374

In Python, the scikit-learn library has a variety of error metrics that we can use. In R, there are likely some smaller libraries that calculate MSE, but doing it manually is pretty easy in either language.

You may notice there’s a small difference in the results here — that’s almost certainly due to parameter tuning, and isn’t a big deal.

(If you run this code on your own, you may also get slightly different numbers, depending on the versions of each package and language you’re using).

**R vs Python: Web Scraping, Part 1**

We have data on NBA players from 2013-2014, but let’s web-scrape some additional data to supplement it.

We’ll just look at one box score from the NBA Finals [here](https://www.basketball-reference.com/boxscores/201506140GSW.html) to save time.

**R**

library(RCurl)

rl <- "https://www.basketball-reference.com/boxscores/201506140GSW.html"

ata <- readLines(url)

**Python**

import requests

rl = "https://www.basketball-reference.com/boxscores/201506140GSW.html"

ata = requests.get(url).content

In Python, the [requests](https://2.python-requests.org/en/latest/) package makes downloading web pages straightforward, with a consistent API for all request types.

In R, [RCurl](https://cran.r-project.org/web/packages/RCurl/index.html) provides a similarly simple way to make requests.

Both download the webpage to a character datatype.

Note: this step is unnecessary for the next step in R, but is shown for comparison’s sake.

**Web Scraping, Part 2**

Now that we have the web page dowloaded with both Python and R, we’ll need to parse it to extract scores for players.

**R**

library(rvest)

age <- read\_html(url)

able <- html\_nodes(page, ".stats\_table")[3]

ows <- html\_nodes(table, "tr")

ells <- html\_nodes(rows, "td a")

eams <- html\_text(cells)

xtractRow <- function(rows, i){

if(i == 1){

return

}

row <- rows[i]

tag <- "td"

if(i == 2){

tag <- "th"

}

items <- html\_nodes(row, tag)

html\_text(items)

span class="token punctuation">}

crapeData <- function(team){

teamData <- html\_nodes(page, paste("#",team,"\_basic", sep=""))

rows <- html\_nodes(teamData, "tr")

lapply(seq\_along(rows), extractRow, rows=rows)

span class="token punctuation">}

ata <- lapply(teams, scrapeData)

**Python**

from bs4 import BeautifulSoup

span class="token keyword">import re

oup = BeautifulSoup(data, 'html.parser')

ox\_scores = []

span class="token keyword">for tag in soup.find\_all(id=re.compile("[A-Z]{3,}\_basic")):

rows = []

for i, row in enumerate(tag.find\_all("tr")):

if i == 0:

continue

elif i == 1:

tag = "th"

else:

tag = "td"

row\_data = [item.get\_text() for item in row.find\_all(tag)]

rows.append(row\_data)

box\_scores.append(rows)

In both languages, this code will create a list containing two lists

1. The box score for CLE
2. The box score for GSW

Both lists contain the headers, along with each player and their in-game stats. We won’t turn this into more training data now, but it could easily be transformed into a format that could be added to our nba dataframe.

The R code is more complex than the Python code, because there isn’t a convenient way to use regular expressions to select items, so we have to do additional parsing to get the team names from the HTML.

R also discourages using for loops in favor of applying functions along vectors. We use lapply to do this, but since we need to treat each row differently depending on whether it’s a header or not, we pass the index of the item we want, and the entire rows list into the function.

In R, we use rvest, a widely-used R web scraping package to extract the data we need. Note that we can pass a url directly into rvest, so the previous step wasn’t actually needed in R.

In Python, we use [BeautifulSoup](https://www.crummy.com/software/BeautifulSoup/bs4/doc/), the most commonly used web scraping package. It enables us to loop through the tags and construct a list of lists in a straightforward way.

**R vs Python: Which is Better? It Depends!**

We’ve now taken a look at how to analyze a data set with both R and Python. And as we can see, although they do things a little differently, both languages tend to require about the same amount of code to achieve the same output.

Of course, there are many tasks we didn’t dive into, such as persisting the results of our analysis, sharing the results with others, testing and making things production-ready, and making more visualizations.

There is a lot more to discuss on this topic, but just based on what we’ve done above, we can draw some meaningful conclusions about how the two differ.

(As far as which is actually better, that’s a matter of personal preference.)

**R is more functional, Python is more object-oriented.**

As we saw from functions like lm, predict, and others, R lets functions do most of the work. Contrast this to the LinearRegression class in Python, and the sample method on Dataframes.

In terms of data analysis and data science, either approach works.  
**R has more data analysis functionality built-in, Python relies on packages.**

When we looked at summary statistics, we could use the summary built-in function in R, but had to import the statsmodels package in Python. The Dataframe is a built-in construct in R, but must be imported via the pandas package in Python.

**Python has “main” packages for data analysis tasks, R has a larger ecosystem of small packages.**

With Python, we can do linear regression, random forests, and more with the scikit-learn package. It offers a consistent API, and is well-maintained.

In R, we have a greater diversity of packages, but also greater fragmentation and less consistency (linear regression is a built-in, lm, randomForest is a separate package, etc).

**R has more statistical support in general.**

R was built as a statistical language, and it shows. statsmodels in Python and other packages provide decent coverage for statistical methods, but the R ecosystem is far larger.

**It’s usually more straightforward to do non-statistical tasks in Python.**

With well-maintained libraries like BeautifulSoup and requests, web scraping in Python is more straightforward than in R.

This also applies to other tasks that we didn’t look into closely, like saving to databases, deploying web servers, or running complex workflows.

Since Python is used across a variety of industries and programming disciplines, it may be the better choice if you’re combining your data work with other kinds of programming tasks.

On the other hand, if you’re focused on data and statistics, R offers some advantages due to its having been developed with a focus on statistics.  
**There are many parallels between the data analysis workflow in both.**

There are clear points of similarity between both R and Python (pandas Dataframes were inspired by R dataframes, the *rvest* package was inspired by *BeautifulSoup*), and both ecosystems continue to grow stronger.