Comparing Runtimes of Python, R, and Matlab

Alexander Berliner, Mohamed Ibrahim,
Woody March-Steinman, Kamaljeet Singh

May 2023

Contents

1 Introduction

As organizations and scientists collect ever-increasing troves of data regarding their experimental subjects, it has become necessary to develop analytical pipelines that are fast, accurate, and energy-efficient. For graduate students in statistics, mathematics, neuroscience, and computer science, it is important to know which programming language provides the fastest output to complete their tasks. In the realm of data processing and analysis, the efficiency of these tasks can be substantially influenced by the run-time of the programming language or tool being utilized.

The objective of this research is to compare the run-times of Python, R, and MATLAB for typical analytical tasks, with the goal of finding the overall fastest language for completing the tasks. While the comparison of these three languages has been studied qualitatively [1], and individual languages or packages have been benchmarked for speed[3] or energy efficiency [2], it is hard to find a concise study directly comparing average speed across tasks commonly used in a typical data science workflow. We aim to determine which of the three programming languages studied is the most efficient overall in terms of run-time performance exclusive of task and dataset size. We begin with the assumption that these factors do not interact, but also perform additional analysis that investigates interactions between these three factors.

2 Experimental Design

Our initial design aimed to evaluate the effect of programming language choice on task execution speed via a replicated Latin square design.

The primary effect we were concerned with was the effect of programming language (Python, MATLAB, or R) on computation speed. Other factors under consideration were task type (simple loops, matrix inversion, and logistic regression) and task size (small, medium, and large, with dimensions dependent on task).

In general, we developed the following design with replicates indicated.

Replicate 1			
	Small	Medium	Large
Simple	Python	R	MATLAB
Matrix	R	MATLAB	Python
Logistic	MATLAB	Python	R

Replicate 2			
	Small	Medium	Large
Logistic	Python	R	MATLAB
Simple	R	MATLAB	Python
Matrix	MATLAB	Python	R

Replicate 3			
	Small	Medium	Large
Matrix	Python	R	MATLAB
Simple	R	MATLAB	Python
Logistic	MATLAB	Python	R

2.1 Programming languages

We decided to choose Python, MATLAB, and R for our comparisons due to their ubiquity in the field of data science. Each has different approaches to handling data types and memory management, but all have libraries that are commonly used for numerical computation tasks. We chose libraries that were commonly used for the tasks chosen, including numpy (python), sklearn (python), and matlib (R).

2.2 Task Selection

We chose three tasks to evaluate. Our simple task involved an array search and comparison using a loop construct. In this case, the task was to count the number of even numbers in a randomly generated natural number array. The goal was to simulate commonly-performed simple operations on data sets of varying size.

The matrix inversion task was chosen for its nonlinear time complexity. While most numerical computation avoids calculating matrix inversion at all costs, this task seemed to allow a strong comparison between various approaches to numerical linear algebra, variable typing, and memory management between our three programming languages. Note that the matrices used contain only positive integer values to maintain $\mathcal{O}(n^3)$ runtime for inversion.

Finally, logistic regression was chosen as a common complex task performed by data scientists across wide scales of data. We made use of statsmodel glm() for python, the fitglm() function in MATLAB, and glmfit() in R.

2.3 Task size selection

Our tasks demanded different data shapes, so size selection was task-dependent. The matrix shapes used are demonstrated below:

	Small	Medium	Large
Simple	1000x1	100000x1	1000000x1
Matrix	100x100	500x500	1150x1150
Regression	2000x10	20000x10	50000x30

Table 1: Data shape per task.

Data matrix sizes were chosen to offset the difference in time complexity of the different tasks and to avoid significant memory constraints as tasks became more complex.

3 Methods

All experiments were performed on an M1 13-inch MacBook Pro (2020). Experiments were performed while not connected to external power and no applications open aside from the application used for running code.

Python experiments were performed using Jupyter notebook, R experiments were performed in RStudio, and MATLAB experiments were performed in MATLAB.

All experiments were run using single-core functionality for consistency. Timing was performed in all cases by subtraction of end time of the procedure in question from the start time, excluding all setup and output.

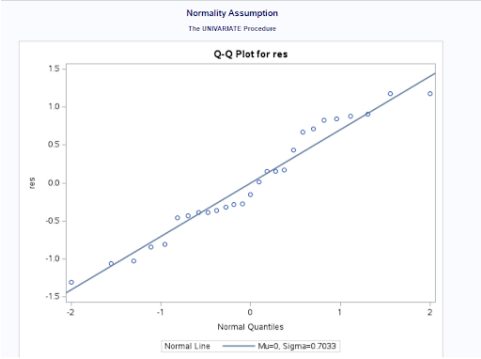
Because of memory constraints, solve() was used in R as opposed to the numpy.linalg.inverse() function in python and the inv() function in MATLAB. This decision was made to accomodate fixed memory limits and with the intent of design choices between the three languages to be more apparent.

Each task was replicated three times with the same data to account for the effects of different background processes on computation time.

4 Results

4.1 Data and Statistical Analysis

Given a model



Tests for Normality				
Test	Statistic		p Value	
Shapiro-Wilk	W	0.953049	Pr < W	0.2540
Kolmogorov-Smirnov	D	0.129926	Pr > D	>0.1500
Cramer-von Mises	W-Sq	0.07504	Pr > W-Sq	0.2357
Anderson-Darling	A-Sq	0.459913	Pr > A-Sq	0.2455

Figure 1: (a) QQ plot and (b) normality tests

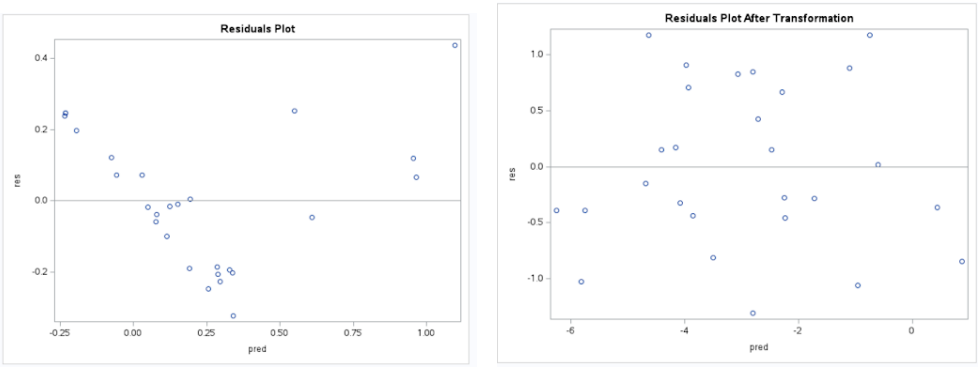


Figure 2: (a) Residuals plot, (b) Residuals plot with log transformation on time response

Source	DF	Type III SS	Mean Square	F Value	Pr > F
Task	2	0.07448766	0.03724383	0.35	0.7077
Language	2	0.70765566	0.35382783	3.30	0.0423
q	1	2.06724739	2.06724739	19.28	<.0001

Figure 3: Additivity analysis (q = pred*pred)

Source	DF	Type III SS	Mean Square	F Value	Pr > F
Language	2	17.89173377	8.94586688	9.38	0.0026
Task(LSReplicate)	6	11.36085114	1.89347519	1.99	0.1363
Size	2	49.88689366	24.94344683	26.15	<.0001
LSReplicate	2	1.76938515	0.88469257	0.93	0.4185

Figure 4: ANOVA analysis of the model

4.2 Discussion

Both a QQ-plot and normality tests showed no reason to reject a null hypothesis of normally-distributed data given $\alpha = 0.05$. Initial analysis showed non-constant variance of our data. We noted that, due to experimental design, there was the potential for an exponential relationship in our response variable based on task size, so the response variable was log-transformed after box-cox analysis. This led to plots demonstrating constant variance, allowing us to proceed with ANOVA analysis. We note that we can reject the additivity of our model ($p < 0.0001$), which leads to issues with the design here, though a 3^3 factorial approach yields similar mean language effects.

Ultimately, our determination demonstrated a significant effect of language ($p = 0.0026$) and task size ($p < 0.0001$), which is not unexpected. Ultimately, task impact

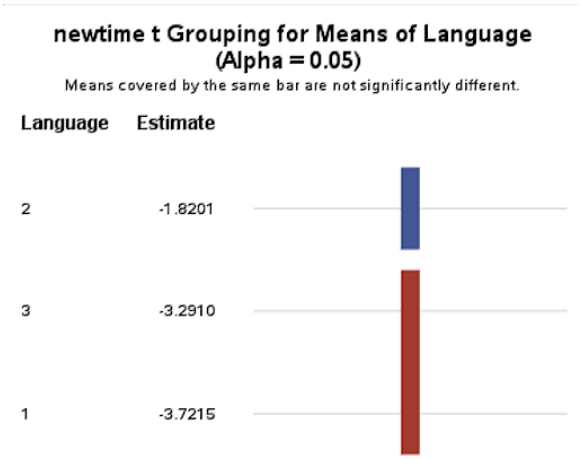


Figure 5: Post-ANOVA analysis via Least Significant Difference

was mitigated by the actual size factor levels chosen for each task.

Post-ANOVA analysis via Least Significant Difference (Figure 5) demonstrates a significant difference in mean log-transformed runtime between language factor level 2 (R) and the other two levels (1, Python and 3, MATLAB). MATLAB and Python show no significant difference in mean. Based on this analysis, we posit that average performance for R is slower (-1.8201 in mean log-transformed runtime) compared to Python and R (-3.7216 and -3.2910 mean log-transformed runtime, respectively).

5 Conclusion

We found that both MATLAB and Python are not statistically distinguishable with our given tests, but R has is significantly slower given our testing conditions. Ultimately, the choice of programming language for any data scientist should take into account many factors, including library convenience and availability, memory footprint, speed, and overall usability.

Further exploration in this topic would aim to make determinations about particular strengths and weaknesses of each programming language, evaluate memory allocation and footprint, focus on performance for only one task. We could also take into consideration the effect of data loading and pre-processing, which was ignored here but appeared to have noticeable differences across languages. Expanding investigation to more math-specific languages like Julia is also an option, or to differentiate between scripting languages and compiled languages (while these results are typically well-studied, the goal would be to look within the context of a standard data science workflow). Additionally, it would be interesting to further evaluate MATLAB and R based on energy efficiency, as there has been a systematic study of other languages in this area (Python included)[2].

References

[1] Ceyhun Ozgur et al. "MatLab vs. Python vs. R". In: *Journal of Data Science* 15.3 (2022), pp. 355–372. ISSN: 1680-743X. DOI: 10.6339/JDS.201707_15(3).0001.

[2] Rui Pereira et al. "Ranking programming languages by energy efficiency". In: *Science of Computer Programming* 205 (2021), p. 102609. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102609. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[3] Tamanna Siddiqui, Mohammad AlKadri, and Najeeb Ahmad Khan. "Review of Programming Languages and Tools for Big Data Analytics". In: *International Journal of Advanced Research in Computer Science* 8 (2017), pp. 1112–1118.

A Program Implementation

A.1 Python implementation

```
#Code for Simple Task:
import time
import csv
with open('numbers_mil.csv', 'r', encoding='utf-8-sig') as f:
    reader = csv.reader(f)
    numbers = [int(row[0]) for row in reader]

start_time = time.time()
count_even = sum([1 for n in numbers if n % 2 == 0])
end_time = time.time()
print(f"There are {count_even} even numbers in the array.")

print("Computation time taken: {:.5f} seconds".format(end_time - start_time))

#Code for Matrix task:
import time
import numpy as np

# Replace "matrix.csv" with our csv data files for the project
matrix = np.loadtxt('matrix_1150.csv', delimiter=',')
start_time = time.time()
inverse = np.linalg.inv(matrix)
end_time = time.time()

# Print the original matrix and its inverse
#print("Original matrix:\n", matrix)
#print("Inverse matrix:\n", inverse)

print("Computation time taken: {:.6f} seconds".format(end_time - start_time))

#Code for Logistic Task

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import statsmodels.formula.api as smf
import statsmodels.api as sm
```

```
# Replace "Logistic_Regression_1" with other csv data files for the project (Logist
data = pd.read_csv("Logistic_Regression_1.csv")

# Split the data into training and testing datasets
train_data, test_data, train_labels, test_labels = train_test_split(data.iloc[:,:-1],
train_data = data.iloc[:, :-1]
train_labels = data.iloc[:,-1]

start_time = time.time()
model = sm.GLM(train_labels, train_data, family=sm.families.Binomial())
end_time = time.time()
print("Computation time taken: {:.6f} seconds".format(end_time - start_time))
```

A.2 R implementation

```
# Array Search
x<- read.csv(file = "numbers_mil.csv", header = FALSE)
x <- x$V1
time_start <- Sys.time()
count <- 0
  for (i in x){
    if (i %% 2 == 0){
      count <- count + 1
    }
  }
#print(sum(x%%2 == 0))
time_end <- Sys.time()
print(time_end - time_start)

#Matrix Inversion
library(matlib)
#Numerical Computation
#Matrix inverse
#n<-1150 #number of rows and columns (Square Matrix required for Inverse)
#data<-sample(1:n, n^2, replace = TRUE)

#mat<-matrix(data = data,nrow=n,ncol=n)
#write.csv(mat, file = "matrix_1150.csv", col.names = FALSE, row.names = FALSE)

mat <- read.csv(file = "matrix_1150.csv", header = FALSE)
time_start <- Sys.time()
```

```r
x<-solve(as.matrix(mat));
time_end <- Sys.time()
print(time_end - time_start)

#Logistic
library(MASS)

n<-25000  #number of observations
p<-30    #number of variables (dimensions)
mu1<-sample(c(0,1),p,replace = TRUE,prob = c(0.3,0.7)) #mean vector of first class
mu2<-sample(c(0,1),p,replace = TRUE,prob = c(0.7,0.1)) #mean vector of second class
Sigma<-diag(p)  #Sigma matrix, using Identity matrix for both the classes

class1<- mvrnorm(n,mu1,Sigma)
class0 <- mvrnorm(n,mu2,Sigma)
x <- rbind(class1,class0)
Y <- c(rep(1,n),rep(0,n))
x<-cbind(x,Y)
write.csv(x, file = "./Logistic_Regression_3.csv", row.names = FALSE)

#small <- read.csv("./Logistic_Regression_1.csv")
med <- read.csv("./Logistic_Regression_2.csv")
#large <- read.csv("./Logistic_Regression_3.csv")
start <- Sys.time()
logistic<-glm(Y~.,data=data.frame(med),family = binomial(link="logit"))
end <- Sys.time()
print(end-start)
```

A.3 MATLAB implementation

```
#ArraySearch
function number = arraysearch %Array search (even/odd)
x = readmatrix('numbers_mil.csv'); %input the data here
tic
count = 0;
for i = 1:length(x)
    if mod(x(i),2) == 0
        count = count + 1;
    end
end
```

```
toc
number = count;

#Matrix Inversion
function y = inverse %Computation of matrix inverse.

k = readmatrix('matrix_1150.csv');
tic
inv(k);
toc

#Logistic Regression
function y = logistic(dataset) %logistic regression from dataset
tic
model = fitglm(X, y, 'link', 'logit', 'Distribution', 'binomial');
toc
```

A.4 SAS code for analysis

```
/* Replicated Latin Square Design*/
PROC IMPORT DATAFILE="/home/u63064808/Stat571B/LSD Project Data New.xlsx"
        DBMS=xlsx
        OUT=data
        REPLACE;
        GETNAMES=YES;
RUN;


/*Original Analysis*/
title 'Latin Square Design';
proc glm data=data;
class language task size LSReplicate;
model time=language task(LSReplicate) size LSReplicate;
output out=output r=res p=pred;
run;

/* check constant variance */
title 'Constant variance checking';
proc sgplot data=output;
scatter x=pred y=res;
refline 0;
run;
```

```
/* check normality */
title 'Normality checking';
proc univariate data=output normal;
var res;
qqplot res/normal (mu=0 sigma=est);
run;

/* check constant variance */
title 'Constant variance checking';
proc sgplot data=output;
scatter x=pred y=res;
refline 0;
run;

/* check normality */
title 'Normality checking';
proc univariate data=output normal;
var res;
qqplot res/normal (mu=0 sigma=est);
run;


/*Do Transformation*/
title 'Transformation';
proc transreg data=output;
 model boxcox(time/convenient lambda=-2.0 to 2.0 by 0.1)=class(language task size LSR
run;

/* do transformation on the response */
title 'Log Transform';
data newdata;
 set data;
 newtime=log(time);
run;


/*Do new analysis*/
title 'Latin Square Design - NEW';
proc glm data=newdata;
class language task size LSReplicate;
model newtime=language task(LSReplicate) size LSReplicate;
```

```
output out=newoutput r=res p=pred;
run;


/* check constant variance */
title 'Constant variance checking - NEW';
proc sgplot data=newoutput;
scatter x=pred y=res;
refline 0;
run;

/* check normality */
title 'Normality checking - NEW';
proc univariate data=newoutput normal;
var res;
qqplot res/normal (mu=0 sigma=est);
run;

/* post-ANOVA comparison */
title 'Post-ANOVA Comparison';
proc glm data=newoutput;
class language task size LSReplicate;
model newtime=language task(LSReplicate) size LSReplicate;
means language/ lines lsd;
output out=new r=res p=pred;
run;
```