Stat 502 Project Report

Arman Bilge, Cheng Wang, and Zexuan Zhou

December 2, 2016

1. Introduction

Python is one of the most popular programming languages, with many applications in scientific computing. Thus, maximizing the performance of Python code is of particular interest. Because Python is an interpreted language, its performance depends on the particular runtime used. Interpreted languages often rely on a just-in-time (JIT) compiler to dynamically optimize the code at runtime, which the official CPython interpreter lacks. Fortunately, there are many options for Python runtimes besides CPython, including PyPy, and IronPython. PyPy uses its own JIT compiler, while IronPython is built on top of .NET framework which encompasses its own JIT compiler. In this study we hoped to determine which Python interpreter has the best performance and how the interpreters' performance varies across operating systems and hardware. In particular, we wanted to test whether the use of a runtime with a JIT compiler offers a performance benefit over the official CPython implementations.

2. Experimental Design

We considered the four Python interpreters aforementioned: CPython 2 and 3, PyPy, and IronPython. Our pilot study also considered a fifth Python interpreter, Jython. However, we experienced several critical bugs when running programs in Jython and deemed it unfit for general use and thus excluded it from the experiment. In addition, we considered three different operating systems, Ubuntu Linux, Macintosh, and Windows running on different hardware.

Each experimental unit was a small program that is a solution to one of the Project Euler problems (Project Euler is a collection of programming puzzles). In this study we considered 45 such programs. Because we expected each program to have its own mean running time, we blocked our results on the program. Our design was a complete block design, as each program was "treated" or run on all combinations of OS and interpreter.

It is not straightforward to accurately measure the performance of a runtime that uses a JIT compiler. A method may need to be called hundreds times before the JIT compiler determines that it is worth optimizing. Because we wanted to assess the asymptotic performance of the interpreters, Before taking any measurements, each program was run 100 times as a "warm-up" to provide ample opportunity for the JIT compiler to make its optimizations.

To control for the state of the computer that we were taking measurements on, we then ran and measured the runtime of each program 10 times. The program runtime was measured using the `time` module in the Python standard library from just before to just after code execution. Note that this purposefully excludes the time that the interpreters took to start up or shut down.

3. Data and Analysis

After investigating an additive model and finding that we have an unequal variance for the residual, we used the Box-Cox procedure to perform a transformation. Unfortunately, the timer in Windows appeared to have less accuracy than in Mac and Linux resulting in several runtimes of zero reported for fast running times. Thus we shifted our timing results by a small amount before performing Box-Cox. The result was that we did a log transformation for our running time and the residual tends to stabilize but still there are some divergence in the lower range. However, The number of these data points is relatively small compared to the total number of all data points so we assumed that we have sufficiently achieved the goal to stabilize the variance among residuals.

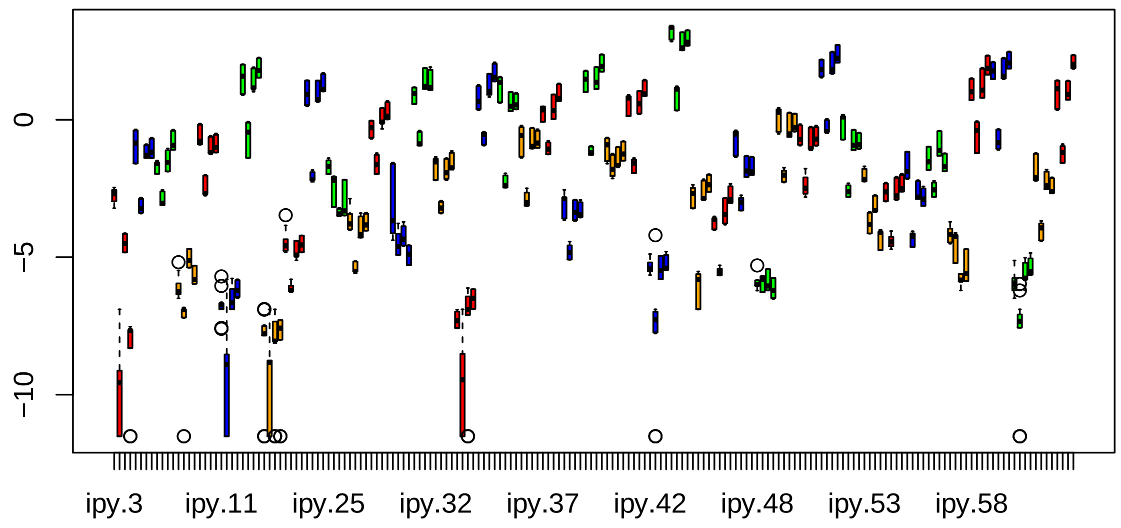
Interpreter	Median	Mean	SD
iPy	-1.41	-1.73	2.77
PyPy	-3	-3.69	2.82
Python	-1.55	-1.92	2.81
Python3	-1.39	-1.83	3.03

Comparing different interpreters, we found that on average the interpreter PyPy is faster than the rest and interpreters with JIT (iPy and Pypy) are faster than the interpreters without JIT (Python and Python3).

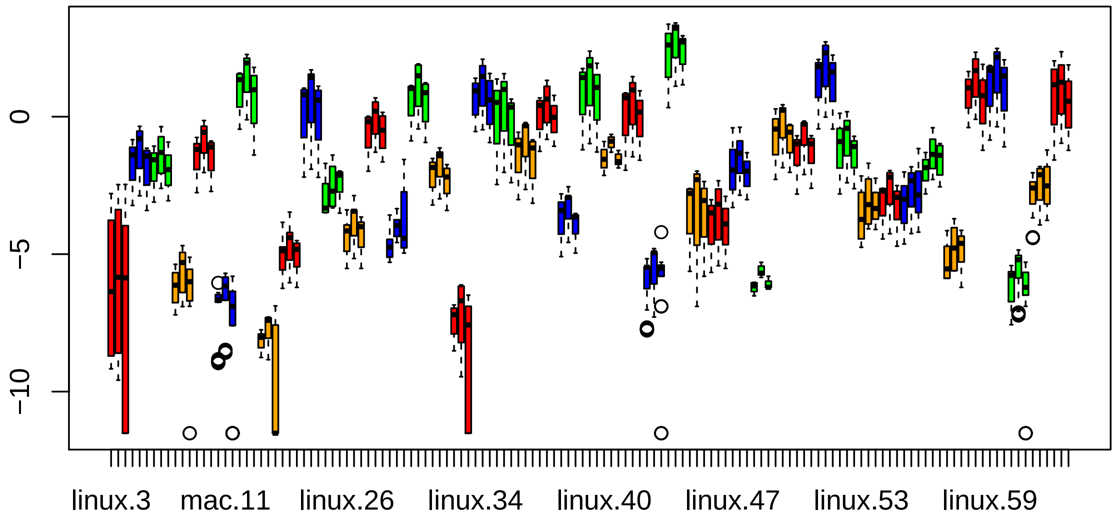
Interpreter	Median	Mean	SD
Linux	-2.16	-2.41	2.87
Mac	-1.78	-2.94	2.93
Windows	-2.02	-2.52	3.09

Comparing different OS we found that on average Linux is faster, however the means of all three interpreters are close to each other so we need to conduct further analysis.

The first boxplot shows the performance of the interpreters. Consecutive boxes of the same color are the same program run on the four interpreters. It is immediately obvious that the second interpreter, PyPy, is often significantly faster than the other tree interpreters.



The second boxplot shows the performance of the OSes. Consecutive boxes of the same color are the same program run on the four OSes. Here, the most consistent trend is that the second OS, Mac, appears to have the slowest running time.



We test these observations for their significance in the following sections.

Models, ANOVA, and violation of assumptions

Recall that we are interested in the following questions:
1. Which interpreters perform best on average?
2. Does OS have influence on the running time?
3. Is there any interaction effect on running time between the OS and the interpreters?
Based on these questions we proposed two models, an additive model and an interaction model. We use the 45 programs as our blocking factor.

$$\text{additive: } y_{ijkl} = \mu_i + \alpha_j + \beta_k + \epsilon_{ijkl},\ \epsilon_{ijkl} \sim N(0, \sigma)$$

$$\text{interaction: } y_{ijkl} = \mu_i + \alpha_j + \beta_k + (\alpha\beta)_{jk} + \epsilon_{ijkl},\ \epsilon_{ijkl} \sim N(0, \sigma)$$

$$\mu\text{: blocks/programs, } i = 1, \ldots, 45; \epsilon\text{: error, } l = 1, \ldots, 5400$$

$$\alpha\text{: OS, } j = 1, 2, 3; \beta\text{: interpreter, } k = 1, 2, 3, 4$$

$$\alpha\beta\text{: interaction effect of OS and interpreter}$$

```
## Analysis of Variance Table
##
## Response: time
##                     Df Sum Sq Mean Sq F value Pr(>F)
## factor(program)     44  41024     932    1796 <2e-16 ***
## os                   2    336     168     323 <2e-16 ***
## interpreter          3   3573    1191    2295 <2e-16 ***
## Residuals         5350   2777       1
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The ANOVA table for the additive model suggests that the OS does have an influence on the running time of programs. However, the portion of variance that the OS factor accounts for is pretty small because we observe a relatively small value for MST of the OS factor.

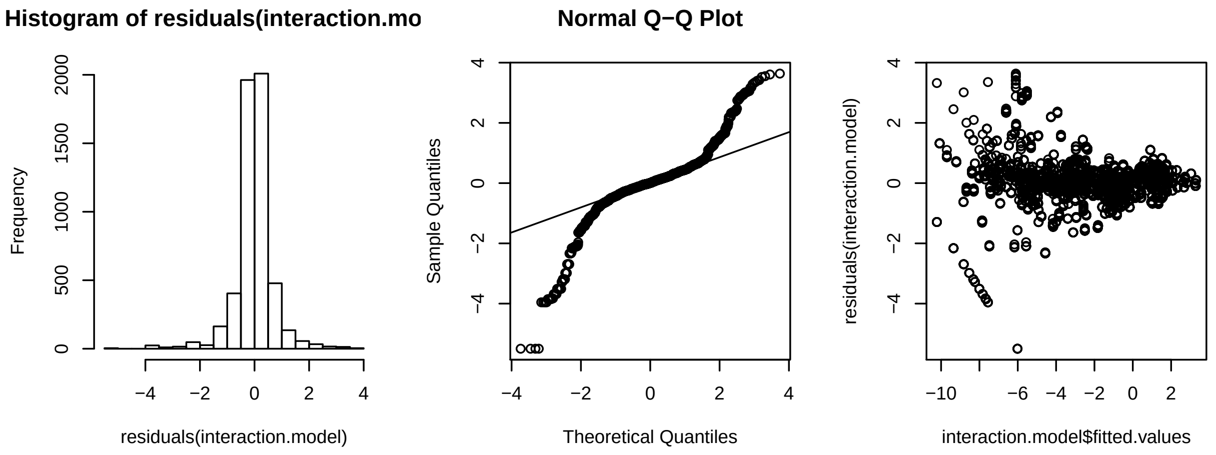
```
## Analysis of Variance Table
##
## Response: time
##                    Df Sum Sq Mean Sq F value  Pr(>F)
## factor(program)   44  41024     932  1820.2 < 2e-16 ***
## os                 2    336     168   327.8 < 2e-16 ***
## interpreter        3   3573    1191  2325.1 < 2e-16 ***
## os:interpreter     6     39       7    12.8 2.1e-14 ***
## Residuals       5344   2737       1
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The ANOVA table for the interaction model suggests that there exists an interaction effect between OS and interpreter. However, the portion of variance that the interaction term accounts for is pretty small because we observe a relatively small value for MST of the OS factor. As for which combination of OS and interpreter perform better we will do a contrast test latter.

We conducted a model selection test to determine whether we should use the full model (interaction model) or the reduced model (additive model).

```
## Analysis of Variance Table
##
## Model 1: time ~ factor(program) + os + interpreter
## Model 2: time ~ factor(program) + os * interpreter
##   Res.Df  RSS Df Sum of Sq    F  Pr(>F)
## 1   5350 2777
## 2   5344 2737  6      39.4 12.8 2.1e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The ANOVA table tells us that there is a significant difference between two models. Based on our questions of interest, we decided to choose the interaction model, which also will help us conduct contrast test to determine what combination of OS and interpreter performs best.



The normality assumption is violated and the equal variance assumption also seems violated. However, because ANOVA is generally robust to these violations, we can still have some confidence in our analysis. There are some skewed points in two sides. We think that these violations might be because we run 100 times for warmup but only run each program 10 times for measurements.

Contrasts

We proposed following contrasts:
$C_1 : \beta_1 + \beta_2 - \beta_3 - \beta_4 = 0$ (JIT vs Non-JIT)
$C_2 : \beta_1 - \beta_2 = 0$ (which one is the best within JIT, iPy or PyPy)
$C_3 : \alpha_1 - \alpha_2 = 0$ (Mac vs Win)
$C_4 : \alpha_2 - \alpha_3 = 0$ (Mac vs Linux)
$C_5 : (\alpha\beta_{23} - \alpha\beta_{33}) - (\alpha\beta_{24} - \alpha\beta_{34})$ (OS*Py vs OS*Py3)
The reason we only do one contrast for the interaction effect is because that we found that the other interaction terms are not significant in our interaction model, which means that we can ignore those terms.

Contrast	Confidence	Interval
C1	-1.37	-1.26
C2	-2.11	-2.03
C3	0.534	0.614
C4	0.22	0.299
C5	0.256	0.369

We see that all of the 95% confidence intervals for our contrasts do not contain zero so we conclude the following:
C_1: JIT is faster than Non-JIT.
C_2: Within JIT, PyPy is faster. This also implies that iPy performs best among all four interpreters.
C_3: Windows OS is faster than Mac OS.
C_4: Windows OS is faster than Linux OS. Along with C_3 we conclude that Windows OS performs best.
C_5: Python3 has a better performance than Python when the OS changes from Mac to Windows.

4. Conclusion

Based on the analysis, implementations in interpreters with JIT compliers are fastest than non-JIT and Windows is faster than Linux and Mac. Sometimes there can be some interactive effects; for example, Python3 is faster than Python when the operating system is changed from Mac to Windows. This benchmark provides a comparison of four interpreters under three different operating systems. The overall comparison shows that a person should choose an appropriate interpreter with operating system, taking into account the time expected. In general, PyPy and Windows appeared to run the fastest and we recommend the use of these for any intensive computation in Python