Performance Assessment of Compiled, Interpreted and Hybrid Programming Languages for Machine Learning Algorithms:Python,Scala and Rust

Engin Tekin

June 2020

1 Introduction

Machine learning gained significance in last decade and it continues increasing its popularity. It's applications mostly require programs to run on computers. These programs are often computationally expensive and consume a lot of memory. Therefore, it is essential to take programming languages performances into consideration while developing and implementing such applications. In addition to performance, there are other factors that affect development process of machine learning applications such as maintainability, portability, learning curve(language), lines of code and etc.

Machine learning programs can be described as set of instructions that also has variables and tells computers what to do with them. These instructions are mostly written and preferred to be written in high-level languages. But these high-level languages can not be run on hardware, hence, these languages should be translated in to machine code[2]. This translation process and language rules create big differences in languages in terms of performance, maintainability and portability. There are mainly three ways of translating high-level languages called source into machine code. Therefore, there are compiled, interpreted and hybrid languages.

This project aims to compare highly widely used three languages(Python/Scala/Rust) in terms of translation process, maintainability, portability, learning process and performance with an experimental analysis.

2 Methodology

Compiled, Interpreted and Hybrid languages were compared in terms translation process, maintainability, portability and performance.

2.1 Translation Process

Translating high-level language (source code) in to machine code that can run on central processing unit (CPU) can vary for different translators. There are three methods that achieves this translation:Compiler, Interpreter and Hybrid methods[Figure 1].

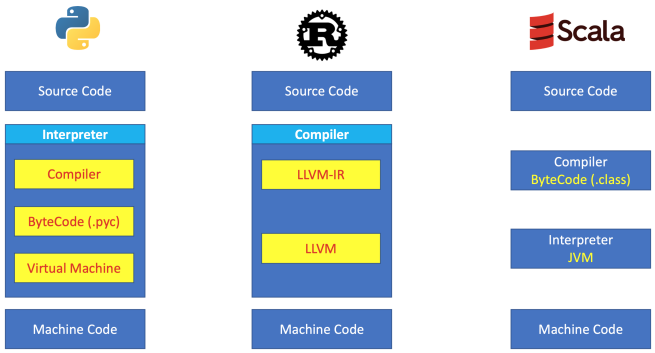

Figure 1: Translation Process

2.1.1 Compiled Languages:Rust

Compilers are translator that translate source code into machine code for a specific hardware configuration. Compiling is achieved in two-main steps:compiling and linking. Compiling process is mainly converting source code into an intermediate representation called object code and linking process links this object code with external libraries by using a linker in order to produce a final executable code in specific hardware[3]. Compiling process can be divided into several steps.

- *LexicalAnalysis* : In this step, initial reading and word level analysis is made by compiler. Program is divided into tokens.
- *SyntaxAnalysis*(*Parsing*) : Obtained tokens are arranged in an abstract syntax tree (AST) which represents structure of program.
- *SemanticAnalysis* : This step verifies and enforce the constraints of types and other requirements.
- *IntermediateCodeGeneration* : Source code is translated into an intermediate language. In Rust this intermediate language is called Low-Level Virtual Machine Intermediate Representation (LLVM-IR). In this step program is still machine independent.
- *MachineCodeGeneration* In this step LLVM runs optimization on code and translates it into assembly or LLVM Bitcode which is machine specific.
- *Linking* In this last step assembly is translated into binary representation and and linker invoked in order to combine object files into a single executable.

Rust heavily relay on LLVM in terms of optimization and machine code generation[5]. LLVM is a C++ library that consists nearly two millions lines of code and have been developed by hundreds of people. LLVM enables Rust to provide more abstraction while being able to generate a source code that can be run as fast as C on CPU.

2.1.2 Interpreted Languages:Python

The translation process in interpreted languages are not made in advance. In this project we will default Cpython interpreter for Python. Interpretation steps can be described in four steps[1].

- *LexicelAnalysis* : In this step, interpreter performs an word level analysis and convert characters into tokens.
- *SyntaxAnalysis* : An AST is created
- *IntermediateCodeGeneration*(*ByteCode.pyc*) AST is converted into Control Flow Graph(CFP)[1]
- *MachineCodeGeneration* : ByteCode (.pyc) is sent to Python Virtual Machine to be executed.

External libraries in Python are translated into ByteCode only once unless there is change in code. Main script that is called in order to execute program is translated into ByteCode from code every time and stored in memory during execution.

There are also other interpreters for Python which uses Just-In-Time (JIT) methods in order to speed up execution time performance. Just-In-Time method can store execution frequency of code blocks and compiles them into machine code. Therefore this block of code is not interpreted in each time.

2.1.3 Hybrid Languages:Scala

Hybrid languages utilizes both compilers and interpreters. In hybrid languages source code is translated into ByteCode that can run on Virtual Machines rather than CPU. In Scala source code is translated into Java Class with .class extension which contains ByteCode that can run on Java Virtual Machine (JVM). JVM uses object file(.class) as input and gives the output. JVM utilizes JIT method in contrast to Python interpreter.

2.2 Maintainability

Maintainability can be defined as easiness of modification of a program to correct faults or improve performance.

2.2.1 Compiled Languages:Rust

Rust is a static typed language. Therefore most of the errors can be detected at compile time and it is less likely to face a run-time error in comparison to dynamic typed languages. Even though Rust requires developer to define variable types in low level (signed, unsigned, # of bytes etc.), type inference feature mitigates difficulty of dealing with this. Besides, Rust provides good documentation in the form of type signatures for intent of programmer. For machine learning purposes, Rust has almost no community as library support. Total amount of code written for this program is 115.

2.2.2 Interpreted Languages:Python

Python is a dynamic type language. Python debugging process is mostly done by running code and monitoring execution. Therefore errors in interpreted languages are detected in run time. This can be interpreted as advantage or disadvantage. This feature sometimes makes it easier to find bugs but, mostly makes program crush unexpectedly in run-time due to type checking etc. Python has a

[1] A control flow graph (often referenced by its acronym, CFG) is a directed graph that models the flow of a program using basic blocks that contain the intermediate representation (abbreviated "IR", and in this case is Python bytecode) within the blocks[1].

quite big community and library support for machine learning applications. Total amount of code written for this program is 85

2.2.3 Hybrid Languages:Scala

Scala is also static typed language with type inference support. Type checking is done compile time and it is well documented. Scala has recently started gaining popularity in machine learning community especially for big data applications. But it has fewer support with respect to Python. Total amount of code written for this program is 95.

2.3 Portability

Portability can be defined as easiness to move a program from a specific platform to an other one.

2.3.1 Compiled Languages:Rust

Rust code is translated into machine code that can be run on a specific target machine. Developer needs to service several versions of program for different target machines. Therefore compiled languages have low portability with respect to interpreted and hybrid languages

2.3.2 Interpreted Languages:Python

Python source code is translated into Python ByteCode and interpreted on the target machine. Since source code is machine independent Python has high portability with respect to compiled languages. One thing that has to be considered in Python is dependencies that has to be maintained on target machine.

2.3.3 Hybrid Languages:Scala

Scala code is translated into Java Class files (.class). Java Class files are machine independent. Therefore developer can produce only one output as program that can be executed by JVM on target machine.

2.4 Performance

In order to measure performance (running time, memory usage) of three languages a well known classification method, logistic regression, is implemented from scratch. No external libraries are used in implementation in order to make performance comparison solely based on language structures other than algorithm differences. There are mainly two process handled reading data from .csv file and running LR algorithm. Algorithms are run on single core except fore Scala-4 (4-Core) in order to utilize Scala multi-threading compiling

Software and Hardware Specifications:

- rustc 1.43.1 (8d69840ab 2020-05-04)
- Scala code runner version 2.11.12 – Copyright 2002-2017, LAMP/EPFL OpenJDK Runtime Environment (build 11.0.7+10-post-Ubuntu-2ubuntu218.04)
- Python 3.6.9 - Cpython

- Linux 5.3.0-53-generic $x86_64$
- CPU: Intel(R) Core(TM) i7-4850HQ CPU @ 2.30GHz CPU max MHz: 3500.0000 CPU min MHz: 800.0000
- RAM: Type: DDR3 Type Detail: Synchronous Speed: 1600 MT/s
- Hard Drive: APPLE SSD SM0512F

2.4.1 Logistic Regression with Stochastic Gradient Descent

Logistic Regression(LR) is a well known linear model in order to perform classification tasks in machine learning. There is several methods to perform this task. We will use Stochastic Gradient Descent(SGD) in order to find classification parameters.

1. Data: Moon Data set Features has dimensions of (N,2) where $\mathbf{N} \in \{1000/10000/100000/1000000\}$
2. Parameters: **# of Epochs** $\in \{1, 5, 10\}$

Pseudo code of algorithm implemented can be seen in Algorithm 1[4].

Data: L(),f(),x,y,n_epoch,l_rate
Result: θ

```
// L: Loss function (Cross-Entropy)
// f:  Function parameterized by θ
// x:  set of training points (N,2)
// y:  set of training outputs (0,1)
θ← 0;
while n_epoch is not reached do
    for each training instance (x^i, y^i) in random order do
        Compute yhat^i ;                                   // Our estimation
        Compute the Loss L(yhat^i, y^i) ;                  // How far are we from real label
        g ← ∇_θ L(yhat^i, y^i) ;                           // How should we move θ to maximize L
        θ ← θ − l_rate ∗ g ;                               // Move in opposite direction in order to decrease L
    end
end
```

Algorithm 1: Stochastic Gradient Descent Algorithm

2.4.2 Results

Performance metrics are selected as below:

- Program - Elapsed CPU Time: Time elapsed by CPU for complete running time including compile/interpretation
- Read - Elapsed CPU Time: Time elapsed by CPU to read .csv file into List/Vec/ArrayBuffer
- SGD - Elapsed CPU Time: Time elapsed by CPU to run SGD algorithm for 5 epochs.
- Max Memory Usage: Max Memory Used while running program
- Note: CPU Time is Elapsed by *perf stat* and max memory usage is elapsed by */usr/bin/time*

Memory performance of languages can be seen in figure 2. Scala has worst memory utilization among other languages. Rust is better than Python for low and medium data sizes while they seem to be equal for big data applications.

Total elapsed time obtained by languages are shown in figure 3. Rust is winner in this metric by far. Considering this language is compiled it seems like compiled languages have great advantage since they are translated to machine code ahead of time. Unexpectedly Python performs better than Scala in this metric for single core. Scala can utilize multiple core even if code written in single thread. Therefore Scala-4 (4-Core) can run in less time than Python.

Time elapsed for reading operation is shown in figure 4. We can see Rust performs better than others in this metric as well. While Scala-1 is slower than Python unexpectedly, Scala-4 performs better than Python as expected.

Time elapsed by SGD algorithm is shown in figure 5. We can see that Rust is fastest language as expected. Except for data size 1000, Scala-1 and Scala-4 performs better than Python which gives us impression that Scala is faster than Python in matrix multiplication tasks.

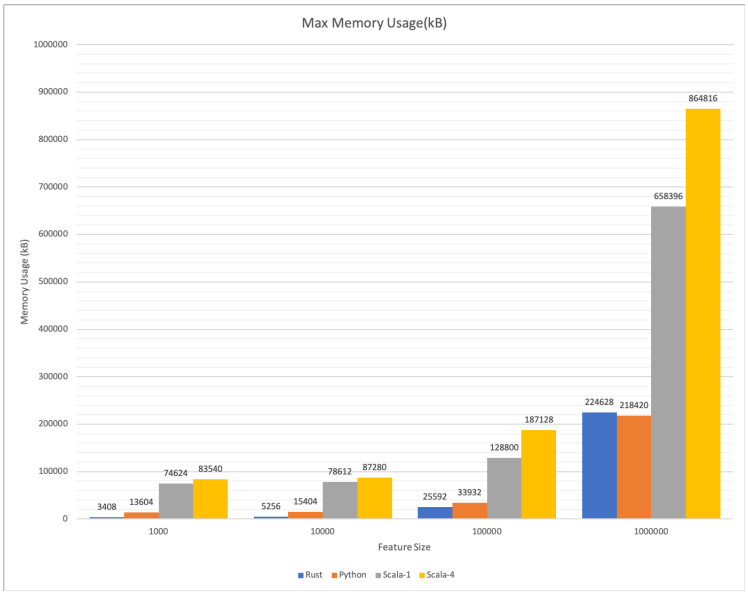

Figure 2: Max Memory Usage

3 Conclusion

This project have shown that compiled(Rust), interpreted(Python) and hybrid(Scala) languages have their advantages and disadvantages. Rust is fast and memory efficient. However it can be painful for beginners to learn since every time it needs to be compiled and run to see outputs. In addition, Rust has be to be compiled for each different target machine. On the other hand Python

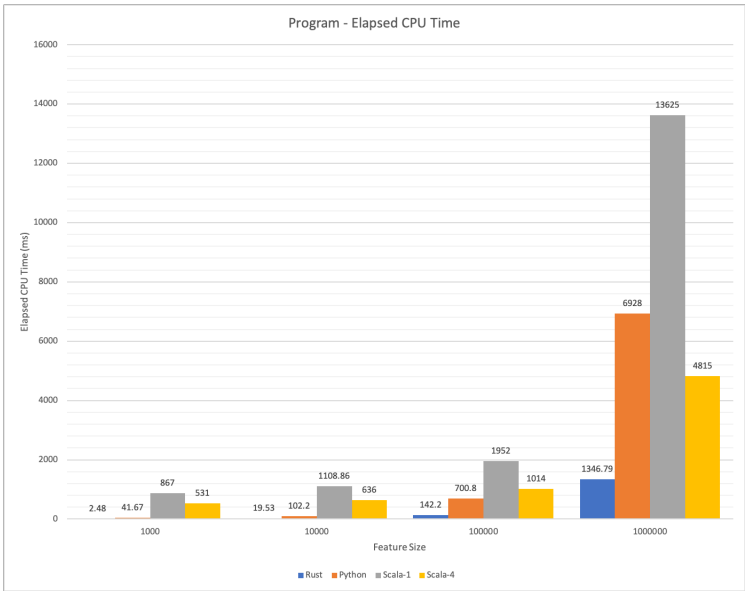

Figure 3: Program Running Time

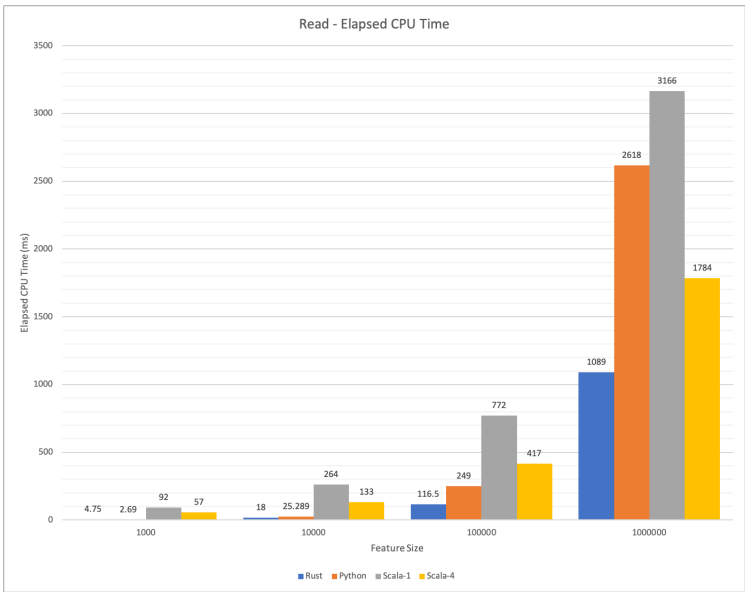

Figure 4: Read Time

is distributed as source code and can be run any platform. But it is slower and memory inefficient with respect to Rust. Python is easy for beginners since it is easy to edit a piece of code and run in interactive mode. Not being have to define types explicitly is also a plus for beginners. Most importantly Python has advantage of having extremely big community and library support for machine learning applications. Scala is a hybrid language that eliminates most of the disadvantages

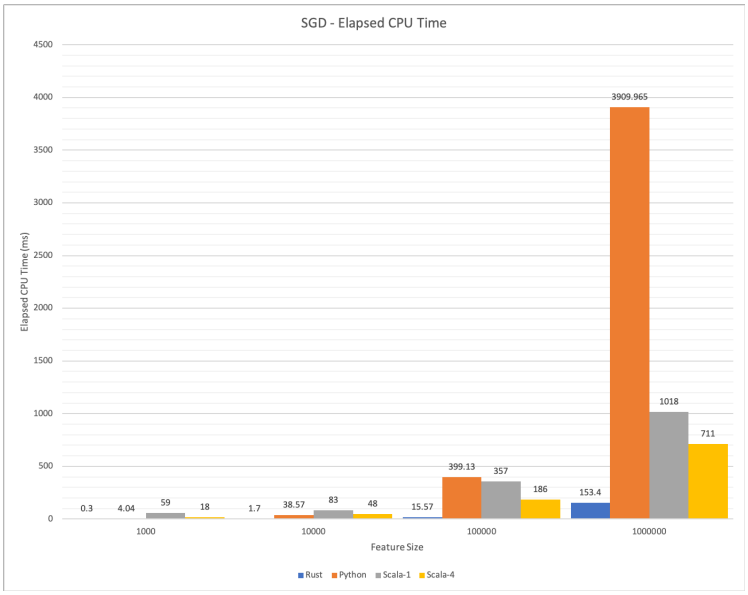

Figure 5: SGD Time

while trying to keep advantages of compiled and interpreted languages. Obtained results shows that Scala is also user-friendly and fast for big data applications. But it lacks memory utilization and community support. Python is convenient for most machine learning applications. Rust and Scala can be used for specific tasks when needed.

4 Future Work

For future work number of the languages implemented can be extended different popular languages R, C++ and etc. Different machine learning algorithms can be implemented including neural networks etc.

5 Source Code

`https://github.com/tekinengin/python-scala-rust-LogisticRegression-with-SGD`

References

[1] *Design of CPython's Compiler*, 2015. `https://devguide.python.org/compiler/`.

[2] Ernest Ampomah, Ezekiel Mensah, and Abilimi Gilbert. Qualitative assessment of compiled, interpreted and hybrid programming languages. *Communications on Applied Electronics*, 7:8–13, 10 2017.

[3] Richard Blum. *Professional Assembly Language*. Wiley Publishing, Inc, 2005.

[4] Daniel Jurafsky James H. Martin. Speech and language processing. Stanford University, 2019.

[5] Niko Matsakis. *Introducing MIR: Rust Blog*, 2016. https://blog.rust-lang.org/2016/04/19/MIR.html.