Comparing Python and Mojo performance in terms of Machine learning and other Computational Tasks

Inhouse Summer Training Internship report submitted in fulfillment of the requirements for the Degree of Bachelor's of Technology In COMPUTER SCIENCE

By

Arbash Hussain
9920103050

Department of Computer Science and Engineering and Information Technology JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY

AUGUST, 2023

TABLE OF CONTENTS

DECLARATION

I hereby declare that this submission is my own work and that, to the best of my knowledge and beliefs, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma from a university or other institute of higher learning, except where due acknowledgment has been made in the text.

Date: 3 August, 2023
Name: Arbash Hussain
Enrollment: 9920103050

ACKNOWLEDGEMENT

I would like to place on record my deep sense of gratitude to Dr. Varsha Garg, Jaypee Institute of Information Technology, India for her generous guidance, help and useful suggestions. I also took some help from the internet in order to get insights on the project that I have worked on to improve the quality of the project work.

Signature:

Arbash

Arbash Hussain (9920103050)

CERTIFICATE

This is to certify that the work titled “Comparing Python and Mojo performance in terms of Machine learning and other Computational Tasks” submitted by Arbash Hussain (9920103050) B.Tech CSE of Jaypee Institute of Information Technology, Noida has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of any other degree or diploma.

ABSTRACT

The research internship project aimed to evaluate and compare the efficiency and performance of Python and the new programming language Mojo for machine learning (ML) model development and general computational tasks. The project involved studying various supervised ML algorithms, exploring Mojo's feasibility for ML model implementation, learning advanced statistics for machine learning, and conducting performance comparisons between Python and Mojo. The report provides insights into the strengths and limitations of both languages.

INTRODUCTION

In today's data-driven world, efficient machine learning algorithms and high-performance computing are vital for diverse applications. This Inhouse Summer training research internship project sought to investigate the potential of Mojo, a new programming language, for ML model development and general computational tasks. The report compares key aspects of Mojo and Python, including speed, memory management, interoperability, learning curve, community support, and ecosystem maturity.

PROBLEM STATEMENT

The Summer training inhouse research internship project aims to compare Python and the new programming language Mojo in terms of machine learning (ML) model efficiency and performance in other Computational tasks. The goal is to evaluate the feasibility of using Mojo for development and to understand its strengths and limitations compared to Python.

The specific objectives of the project are as follows:

<u>Study and Implement ML Algorithms:</u> Learn and implement various supervised machine learning algorithms, including linear regression, elastic net regression, logistic regression, decision trees, support vector machines (SVM), naive Bayes, and k-nearest neighbors (kNN) in Python.

<u>Advanced Statistics for Machine Learning:</u> Gain a solid foundation in advanced statistical concepts essential for ML model evaluation, performance analysis, and comparison between Python and Mojo.

<u>Explore Mojo's Feasibility:</u> Investigate the capabilities and limitations of Mojo for ML model development. Assess the compatibility of Python libraries and the potential for using Mojo as a superset of Python for seamless integration of existing tools and libraries.

<u>Performance Comparison:</u> Conduct performance comparisons between Python and Mojo for ML models and general computational tasks. Evaluate execution times, memory utilization, and overall efficiency to understand the relative performance of the two languages.

TECHNOLOGIES, MODELS AND LANGUAGE USED

In this Project, as the title itself is explanatory, we will be comparing Mojo and Python based on Machine learning algorithms and other computing tasks. We will go through the various algorithms like:

Logistic Regression
Linear Regression
Elastic Net Regression
Decision Trees
Support Vector Machine
K Nearest Neighbors
Naive Bayes
Hyperparameter Tuning
K Fold Cross validation
Fibonacci Calculation
Sudoku Solver

For carrying out these models, we will make use of Iris dataset and some other custom datasets that we can create using the Scikit learn library.

ABOUT Python

Python is a general-purpose, high-level programming language. Its design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.

ABOUT Mojo

Mojo🔥

Mojo is a new programming language that is designed to be a superset of Python. It combines the ease of use and flexibility of Python with the performance and control of systems languages, like C++ and Rust.

Chris Lattner created the Mojo programming language. Lattner is a well-known figure in the programming world, having previously created the Swift programming language and the LLVM Compiler Infrastructure. He is currently the CEO of Modular Inc., the company that developed Mojo.

Although Mojo is still under development, it has the potential to be a powerful language for AI and machine learning applications. It is designed to be efficient and memory-safe, and it can be used to access low-level hardware primitives.

WHY COMPARE PYTHON AND MOJO?

- Both are general-purpose programming languages. This means that they can be used for a wide variety of tasks, including web development, software development, data analysis, and machine learning.
- Both are relatively easy to learn. This makes them good choices for beginners who are looking to learn a new programming language.
- Both have their own strengths and weaknesses. Python is known for its ease of use and flexibility, while Mojo is known for its performance and memory safety.
- As a result of these similarities, it is natural to compare Python and Mojo to see which language is a better fit for a particular task. For example, if you are looking for a language that is easy to learn and use for general-purpose tasks, then Python may be a good choice. However, if you are looking for a language that is designed for performance and memory safety, then Mojo may be a better choice.

Methodology

The research methodology encompassed the following key elements:

2.1. Study and Implementation of ML Algorithms:

In the initial weeks, various supervised ML algorithms, such as linear regression, elastic net regression, logistic regression, decision trees, SVM, naive Bayes, and kNN, were studied and implemented in Python. These algorithms served as a reference for comparison with Mojo's capabilities.

2.2. Advanced Statistics for Machine Learning:

To enhance the project's statistical foundation, time was devoted to learning advanced statistical concepts essential for ML model evaluation and comparison. Concepts like hypothesis testing, confidence intervals, and model evaluation metrics were explored in depth.

2.3. Evaluation of Mojo's Feasibility:

Due to limitations in Mojo's current state, such as missing support for essential features and modules, direct ML model implementation from scratch was not feasible. As an alternative, Python codes for ML models were executed within the Mojo kernel to assess performance differences.

2.4. Performance Comparison:

Performance comparisons were conducted between Python and Mojo for both ML models and general computational tasks. Execution times, memory utilization, and overall performance were measured to evaluate the languages' relative efficiency.

Findings

3.1. Strengths of Mojo:

Speed: Mojo's compiled nature enables faster execution compared to Python's interpreted approach, making it well-suited for computationally intensive tasks.

Memory Management: Mojo's greater control over memory management reduces the likelihood of memory-related issues compared to Python.

Interoperability: Mojo's ability to work with Python libraries enhances its usability and facilitates the integration of existing Python tools and libraries.

3.2. Limitations of Mojo:

Learning Curve: Mojo's newer language features and steeper learning curve may pose challenges for newcomers, whereas Python's familiarity and extensive resources ease the learning process.

Community Support: Python enjoys a larger and more established community, leading to a wealth of resources and library support, while Mojo's community is still growing.

Ecosystem Maturity: Python's mature ecosystem offers a plethora of applications and company support, which is currently less abundant for Mojo, although it is the goal of Mojo to be the superset of Python.

Theoretical Comparison between Mojo and Python:

4.1. Features of Mojo not in Python:

Static Typing: Mojo's static typing requires explicit declaration of variable types, ensuring better type safety than Python's dynamic typing.

Pointers: Mojo's support for pointers enables direct memory access, providing advantages for low-level programming.

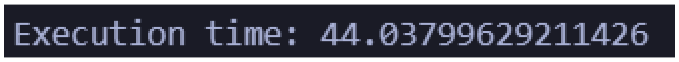
Modules: Mojo modules are more powerful, allowing the inclusion of code, data, and types, unlike Python modules that can only contain code.

Practical Comparison:

5.1. Machine Learning Model Comparison of Python in Mojo kernel:

Hyperparameter tuning on SVR in Python Notebook:

```
from sklearn.model_selection import GridSearchCV
st = time.time()
params = {
    'kernel':['linear'],
    'C':[0.1,1,10,100,1000],
    'gamma':[1,0.1,0.01,0.001,0.0001],
    'epsilon':[0.1,0.2,0.3]
}
gridcv =
GridSearchCV(SVR(),param_grid=params,cv=5,scoring='neg_mean_squared_error',verbose=3
,refit=True)
gridcv.fit(X_train,y_train)
et = time.time()
print('Execution time:',et-st)
```



```
Execution time: 44.03799629211426
```

After running the same code in Mojo Notebook a significant increase in speed was seen

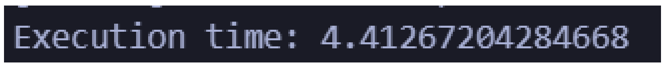
Python code in Python kernel took **44 secs** to train

Python code in Mojo kernel took just **4.412 secs** to train

Python code in Mojo Notebook is **10 Times Faster!**

Hyperparameter tuning on SVR in Mojo Notebook:

```
%%python
st = time.time()
params = {
    'kernel':['linear'],
    'C':[0.1,1,10,100,1000],
    'gamma':[1,0.1,0.01,0.001,0.0001],
    'epsilon':[0.1,0.2,0.3]
}
gridcv =
GridSearchCV(SVR(),param_grid=params,cv=5,scoring='neg_mean_squared_error',verbose=3
,refit=True)
gridcv.fit(X_train,y_train)
et = time.time()
print('Execution time:',et-st)
```



```
Execution time: 4.41267204284668
```

5.2. Fibonacci calculation of 40:

Python implementation

```
%%python
from timeit import timeit
import time
def fib(n):
    if n <= 1:
       return n
    else:
       return fib(n-1) + fib(n-2)

st = time.time()
print(fib(40))
et = time.time()
python_secs = et-st
print("Execution Time in Seconds:",python_secs)
```

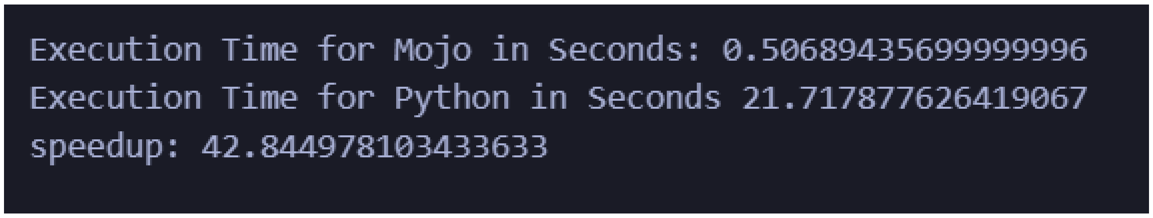
Mojo Implementation

```
from Benchmark import Benchmark
fn fib(n: Int) -> Int:
    if n <= 1:
       return n
    else:
       return fib(n-1) + fib(n-2)

fn bench():
    fn closure():
        for i in range(40):
            _ = fib(i)

    let nanoseconds = Benchmark().run[closure]()
    let mojo_secs = Float64(nanoseconds) / 1e9
    print("Execution Time for Mojo in Seconds:", mojo_secs)
    print("Execution Time for Python in Seconds",21.717877626419067)
    print("speedup:", 21.717877626419067 / mojo_secs)

bench()
```



```
Execution Time for Mojo in Seconds: 0.50689435699999996
Execution Time for Python in Seconds 21.717877626419067
speedup: 42.844978103433633
```

Speedup of 42 through Mojo code!

5.3. Sudoku Solver Speed Comparison:

Python Implementation

```
%%python
def is_valid(board, row, col, num):
    for x in range(9):
        if board[row][x] == num:
            return False
    for x in range(9):
        if board[x][col] == num:
            return False

    start_row = row - row % 3
    start_col = col - col % 3
    for i in range(3):
        for j in range(3):
            if board[i+start_row][j+start_col] == num:
                return False
    return True

def solve_sudoku(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                for num in range(1, 10):
                    if is_valid(board, i, j, num):
                        board[i][j] = num
                        if solve_sudoku(board):
                            return True
                        board[i][j] = 0
                return False
    return True

board = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]

print("Solved:", solve_sudoku(board))
for i in board:
    print(i)
```

Mojo Implementation

```
from Pointer import DTypePointer
from DType import DType
from Buffer import NDBuffer
from List import DimList
from Random import randint
from List import VariadicList
from Math import sqrt
from Numerics import FPUtils


struct Board[grid_size: Int]:
    var data: DTypePointer[DType.uint8]
    var sub_size: Int
    alias elements = grid_size**2

    fn __init__(inout self, *values: Int) raises:
        let args_list = VariadicList(values)
        if len(args_list) != self.elements:
            raise Error("The amount of elements must be equal to the grid_size
parameter squared")

        let sub_size = sqrt(Float64(grid_size))
        if sub_size - sub_size.cast[DType.int64]().cast[DType.float64]() > 0:
            raise Error("The square root of the grid grid_size must be a whole
number 9 = 3, 16 = 4")
        self.sub_size = sub_size.cast[DType.int64]().to_int()


        self.data = DTypePointer[DType.uint8].alloc(grid_size**2)
        for i in range(len(args_list)):
            self.data.simd_store[1](i, args_list[i])

    fn __getitem__(self, row: Int, col: Int) -> UInt8:
        return self.data.simd_load[1](row * grid_size + col)

    fn __setitem__(self, row: Int, col: Int, data: UInt8):
        self.data.simd_store[1](row * grid_size + col, data)

    fn print_board(inout self):
        for i in range(grid_size):
            print(self.data.simd_load[grid_size](i * grid_size))

    fn is_valid(self, row: Int, col: Int, num: Int) -> Bool:
        # Check the given number in the row
        for x in range(grid_size):
            if self[row, x] == num:
                return False

        # Check the given number in the col
        for x in range(grid_size):
            if self[x, col] == num:
                return False

        # Check the given number in the box
        let start_row = row - row % self.sub_size
        let start_col = col - col % self.sub_size
        for i in range(self.sub_size):
            for j in range(self.sub_size):
                if self[i+start_row, j+start_col] == num:
                    return False
        return True

    fn solve(self) -> Bool:
        for i in range(grid_size):
            for j in range(grid_size):
                if self[i, j] == 0:
                    for num in range(1, 10):
                        if self.is_valid(i, j, num):
                            self[i, j] = num
                            if self.solve():
                                return True
                            # If this number leads to no solution, then undo it
                            self[i, j] = 0
                    return False
        return True


let board = Board[9](
    5, 3, 0, 0, 7, 0, 0, 0, 0,
    6, 0, 0, 1, 9, 5, 0, 0, 0,
    0, 9, 8, 0, 0, 0, 0, 6, 0,
    8, 0, 0, 0, 6, 0, 0, 0, 3,
    4, 0, 0, 8, 0, 3, 0, 0, 1,
    7, 0, 0, 0, 2, 0, 0, 0, 6,
    0, 6, 0, 0, 0, 0, 2, 8, 0,
    0, 0, 0, 4, 1, 9, 0, 0, 5,
    0, 0, 0, 0, 8, 0, 0, 7, 9
)

print("Solved:", board.solve())
board.print_board()
```

Comparing

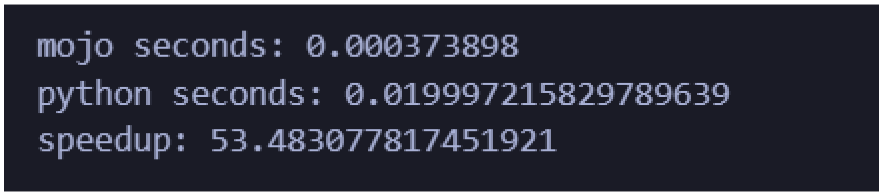
```
from Benchmark import Benchmark
alias board_size = 9
fn bench(python_secs: Float64):
    @parameter
    fn init_board() raises -> Board[board_size]:
        return Board[board_size](
            5, 3, 0, 0, 7, 0, 0, 0, 0,
            6, 0, 0, 1, 9, 5, 0, 0, 0,
            0, 9, 8, 0, 0, 0, 0, 6, 0,
            8, 0, 0, 0, 6, 0, 0, 0, 3,
            4, 0, 0, 8, 0, 3, 0, 0, 1,
            7, 0, 0, 0, 2, 0, 0, 0, 6,
            0, 6, 0, 0, 0, 0, 2, 8, 0,
            0, 0, 0, 4, 1, 9, 0, 0, 5,
            0, 0, 0, 0, 8, 0, 0, 7, 9
        )

    fn solve():
        try:
            let board = init_board()
            _ = board.solve()
        except:
            pass

    let mojo_secs = Benchmark().run[solve]() / 1e9
    print("mojo seconds:", mojo_secs)
    print("python seconds:",python_secs)
    print("speedup:", python_secs / mojo_secs)

bench(python_secs.to_float64())
```

Output:



```
mojo seconds: 0.000373898
python seconds: 0.019997215829789639
speedup: 53.483077817451921
```

Speedup of 53 through Mojo code!

Conclusion

Mojo exhibits strengths in terms of speed, memory management, and interoperability with Python libraries, making it a compelling option for certain computational tasks. However, its limitations, including the learning curve, smaller community, and less mature ecosystem, restrict its practical usability for complex ML model development when compared to Python.

GitHub link:
https://github.com/CC-KEH/Comparing-Python-and-Mojo

REFERENCES

https://docs.modular.com/mojo/

https://docs.python.org/3/

https://scikit-learn.org/stable/

https://www.javatpoint.com/machine-learning-algorithms

Thankyou

Arbash Hussain
9920103050