Improving the processing speed of medium sized CSV files using Python

Sander van Wickeren
sander.vanwickeren@student.uva.nl
University of Amsterdam

ABSTRACT

The amount of data available is increasing worldwide, and imposes the need for sophisticated methods to process it all. However, not every company has petabytes of data and could process it on a single computer. Velocity, one of the important concepts for big data also applies to smaller amounts. Because Python is one of the popular languages, this research tries to find the optimal way to process medium sized CSV files on a local computer using Python. Through timed experimentation with the analysis of Amazon reviews using Pandas, Numpy, Numba and Cython a conclusion is drawn that overall Cython achieves the highest speeds on strings, while Numpy excels in numerical calculations.

1 INTRODUCTION

The volume of data is increasing worldwide. As of the beginning of 2019, 56.1% of the world's population can surf the internet [3]. Estimated is that for every person on earth, 1.7MB of data will be created every second in 2020 [2]. This growth imposes the need for sophisticated methods to process all the data. Programmers could trust on Moore's law until around the beginning of the twentieth century, to improve the performance of their programs [18]. Since the stagnation of the law, more attention should be paid to architectures, algorithms and the distribution of work. One such improvement is MapReduce, developed at Google by Dean & Ghemawat [7]. The work is distributed among different workers each parsing the key / value pairs of the input and feeding them to the user-defined map function. After a map worker is finished the result gets forwarded to the reduce workers, which group occurrences of the same keys together and save it to the final output file. Another improvement is Apache Spark, which is a solution for the broad range of different processing types needed to process big data, by providing an environment that simplifies combining all the processing steps [23]. Tools like MapReduce and Spark are used by companies with terabytes, exabytes or even petabytes of data, processed on clusters of computers. While large companies deal with such quantities, smaller companies have much less data at their disposal and can process on a smaller scale without cluster, cloud or grid computing. Although many solutions and/or architectures are explained for big data [22], there are few scientific articles on accelerating data analysis without the use of big data architectures. Looking at the most important V's within big data (volume, velocity and variety), regardless of the amount, the velocity of processing the data is important. It is a waste of time to wait until the data is processed, while faster solutions are within reach. For example, when a calculation has to be displayed on a web page on request, one does not want to wait long before loading is completed. One way to improve the processing times of data is to optimise the code.

Optimising code prematurely is often called a bad-practise, or in the words of Sir Tony Hoare (popularized by Donald Knuth) "premature optimisation is the root of all evil". However, Hyde describes Hoare's interpretation as a fallacy [11]. Optimisation meant in the time of origin something different than today. The quote referred to micro-optimising the amount of CPU cycles or instructions while coding in Assembly, which is being abstracted by compilers nowadays. Currently, it should instruct developers to focus on good algorithm design and good implementations, before worrying about micro-optimising. Wrong interpretation of the quote causes optimisation to be postponed in the development cycle, so that it is usually neglected for economic / time-to-market reasons [11]. So, optimising code from the start is not a bad idea at all, micro-optimising prematurely is.

Despite there being faster programming languages like C or Rust, Python is one of the most popular ones, mainly because of its ease of use [4, 15]. The vast amount of available packages, makes the number of solutions for a problem endless. Due to the popularity of Python and the importance of speed, even for smaller data sets, this research explores a selection of optimisation approaches to process medium-sized datasets, that could improve the speed on a local computer using Python.

2 LITERATURE REVIEW

Optimising code

Code optimisation can be approached from multiple angles. Firstly, there are various ways to solve code problems. Function-wise, iterating a list of numbers and adding one to each entry can be written in multiple ways. From fast to slow one could use a list comprehension, the map function or a for loop. This order is not fixed, functions perform some tasks better than others. Algorithmically, programmers should try to keep the run-time as low as possible. In Computer Science the big O notation (O) is used to classify the speed or complexity of the algorithm. Ideally, the complexity should be $O(1)$, which means that no matter how much data or operations are used running the algorithm, the execution time should stay the same.

Secondly, scientific packages like Numpy and Pandas can be used. Originally, Python was not built with fast numerical calculations in mind. These packages are built as a solution and offer multiple ways to solve a programming problem. The fastest way of using these packages is often described in blog posts, concluding that Numpy achieves the quickest execution times [8, 10]. Numpy is fast, because its functions are implemented in C, it tries to parallelize as much as possible and introduces Numpy arrays which are densely packed in memory and support different data-types better than the original list implementation of Python [21].

Thirdly, multiprocessing can be used. Its brother, multi-threading, is not always an option, because Python's Global Interpreter Lock (GIL) prevents multiple threads running in parallel within the same Python process [19], therefore providing no performance boost. Or in other words, because of the GIL, a thread has to wait for a previous thread to be finished before being executed, making the the execution actually single threaded. Multiprocessing does not suffer from this problem, because it invokes different instances each with their own GIL. Running multiple processes side by side increases the speed of the execution.

Finally, using a different Python compiler. Largely due the dynamic nature of Python, low-level computational code tends to be slow [6]. To mitigate this, Python offers a Python/C API (CPython) to allow writing parts of the program in C. However, it has a sharp learning curve, making it hard to use for less experienced developers [6]. Therefore, there are some solutions that compile Python directly to C, existent in two variants. Firstly, a just-in-time (JIT) compiler, which runs after a program has been started and compiles the code into bytecode when a method is invoked. Secondly, an ahead-of-time compiler (AOT), which compiles all the code before the program is started for the first time [16]. The latter is standard for Python, while the former is used with other implementations. There are a couple of different Python compilers in the field.

First, Cython, an AOT based compiler which can be installed using Python's package manager Pip. It allows compilation from Python to C if variables are defined as static instead of dynamic types (displayed in the second declaration of the function).

```
def foo(a):
    return a + a
```

```
cpdef int foo(int a):
    return a + a
```

Secondly, Numba, which is a function-at-a-time JIT compiler for CPython [12]. Through the usage of a Python decorator (@jit), Numba compiles the code to machine code in *nopython mode* or uses the CPython API as a fallback if it cannot infer the type of the code. Its initial focus is to prevent users having to write code in low-level languages for better performance [12].

```
from numba import jit

@jit
def foo(a):
    return a + a
```

Thirdly, PyCOMPS, the required library to use COMPS in Python [20]. COMPS is a framework developed to let programmers write programs sequential, avoiding the need to think of parallelisation and or distribution. This package is not used during the experimentation because it relies on another framework, which is not written in Python and because the documentation is poor [1].

In addition there are alternative implementations of Python such as PyPy, which uses its own interpreter and therefore cannot be installed with Pip. The goal of PyPy is to keep the language as pythonic as possible, while improving the speed [5]. Some disadvantages are that it does not support all packages that Python supports and that it always runs a version behind Python. Another alternative implementation is Nuitka [9], which translates the Python code into C. Experiments showed, that Nuitka performed the least compared to PyPy, CPython and Cython [14]. Both Nuitka and PyPy won't be used during the experimentation, because it has to be installed as a different Python version instead of as a package via Pip.

3 METHODOLOGY

In order to test the different approaches, an Amazon food review dataset has been used. The dataset, retrieved from Kaggle and created by Mcauly et al., has ten columns containing product and user ids, helpfulness score of a review, score, time summary and the text [13, 17]. The dataset has been used to test and compare the speed of the following environments:

- Raw python, where only standard packages were allowed.
- Pandas, using built-in functions like *iterrows()*, *itertuples()* and *apply()*.
- Numpy, using vectorization or built-in functions like *.std()*.

Table 1: Comparing read speed of the Amazon review data set using Python and Pandas.

Approach	Load time	Std. dev.
csv.reader()	5.27 s	72 ms
pandas.read_csv()	3.76 s	78 ms

- Numba, applying the Numba compiler on Python and the other packages (except Cython).
- Cython, adding types and applying the compiler on Python and the other packages (except Numba).

Experiments

Performance can be measured in multiple ways, such as memory usage, CPU usage and execution time. Because the goal of the project was to see which of the packages are the quickest, the experimentation focused solely on speed. As mentioned in the introduction, some functions are faster in one situation than in others. Therefore, multiple different experiments have been ran to prevent skewed results and to highlight strength and weaknesses in different scenarios. The experiments are described as follows:

(1) **Chocolate:** Goal of this experiment was to get the review scores for all reviews containing the word *Chocolate*, testing how well string operations perform in loops.
(2) **Deviation:** Goal of this experiment was to calculate the sample standard deviation of the score column, testing how well numerical operations perform.
(3) **Sorting strings:** Sorting the titles of the reviews, testing how fast text can be sorted.
(4) **Sorting integers:** Sorting the scores of the reviews, testing how fast numerical values can be sorted.

The experiments have been ran seven times per implementation in a Jupyter Notebook. Through the built-in magic command '*%%timeit*' an average has been calculated which is used to compare with the other implementations.

The testing system consisted of a Intel Core i7-2600K quad core CPU @ 3.40GHZ, 8GB GDDR4 RAM, NVIDIA GeForce GTX 550 Ti graphics card (1 GB), Crucial MX500 500gb SSD and ran Windows 10. Python 3.7.4 has been used as the core version. The code together with the versions of the packages are available on Github[1]. To make sure the experiments were not influenced by other activities from the computer, only the Notebook was running. After each function the kernel was cleared and restarted to carry out the experiments with the most comparable conditions as possible.

In order to make a fair comparison, it had to be taken into account that Pandas loads the the data quicker than Python's standard *csv.reader* function (both read times compared in table 1). Because the goal of this project was to measure the speed of different function implementations and not of loading the data into memory, the data was saved into a variable before the functions were executed. Loading the data into memory is only possible for small files, which is a limitation if the results are applied to much bigger data sets, where loading all the data into memory is often not feasible.

[1]https://github.com/SandervWickeren/SpeedComparison

Table 2: Experiment 1

Approach	Average	std. dev.
Python; for loop	559 ms	2.67 ms
Python; list comprehension + map	607 ms	8.81 ms
Python; list comprehension	487 ms	2.51 ms
Pandas; contains	854 ms	3.17 ms
Pandas; iterrows	56700 ms	506 ms
Pandas; itertuples	1420 ms	16.8 ms
Pandas; apply lambda	600 ms	8.14 ms
Pandas; map lambda	604 ms	3.76 ms
Numpy; vectorize	517 ms	2.38 ms
Numpy; vectorize cast values	516 ms	2.29 ms
Numba + Numpy	5190 ms	151 ms
Cython; for loop	464 ms	6.22 ms
Cython; list comprehension	466 ms	3.37 ms
Cython + Numpy	480 ms	11 ms

4 RESULTS

The results of each experiment can be found in table 2-5. The quickest executions are marked with a gray background. For the first experiment (filtering data based on a string), a combination of Cython and a standard *for loop* performed best with 464 milliseconds, while looping through the data using Pandas *iterrows()* is by far the slowest with 56.7 seconds.

For the second experiment (calculating the sample standard deviation of the score), Numpy's built-in *std()* was the fastest with 3.4 ms and Numba the slowest with 5.1 seconds. Python's statistics packages performed worse than using a list comprehension with respectively 1970 ms and 487 ms.

The third experiment (sorting strings), showed that Numpy in combination with Python's standard *sorted()* function (353 ms) was faster than using Numpy's built-in function (732 ms), while the fourth experiment (sorting integers), resulted in Numpy's built-in *sort()* being the fastest (11.1 ms). Numpy was faster in processing integers and Python was faster in processing strings.

Based on the results the following observations can be prompted:

- When using Python, list comprehensions are the quickest according to each experiment.
- Looping through a Pandas DataFrame using either *iterrows()* or *itertuples()* was slow.

Table 3: Experiment 2

Approach	Average	std. dev.
Python; for loop	356 ms	1.94 ms
Python; list comprehension + map	396 ms	14.2 ms
Python; list comprehension	313 ms	3.09 ms
Python; statistics package	1970 ms	25.5 ms
Pandas + sum(list)	54.5 ms	408 µs
Pandas + list.sum()	9.49 ms	162 µs
Pandas; built-in std() function	7.33 ms	157 µs
Numpy; built-in std() function	3.4 ms	49.5 µs
Numpy + math	5.36 ms	35.7 µs
Numba + list comprehension	505 ms	3.56 ms
Numba + Numpy array	2850 ms	58.6 ms
Cython; for loop	221 ms	2.31 ms
Cython; list comprehension	9.41 ms	267 µs
Cython + Numpy + math	5.42 ms	104 µs
Cython + Numpy	3.54 ms	188 µs

Table 4: Experiment 3

Approach	Average	std. dev.
Python; for loop	456 ms	5.35 ms
Python; list comprehension	425 ms	1.47 ms
Pandas; built-in .sort_values()	945 ms	6.93 ms
Pandas + sorted(list())	396 ms	8.21 ms
Pandas + sorted()	386 ms	8.58 ms
Numpy; built-in sort()	732 ms	3.24 ms
Numpy + sorted()	353 ms	10.7 ms
Numba + quicksort	1270 ms	7.3 ms
Cython + list comprehension	390 ms	2.01 ms

Table 5: Experiment 4

Approach	Average	std. dev.
Python; for loop	128 ms	3.77 ms
Python; list comprehension	105 ms	1.67 ms
Pandas; built-in .sort_values()	37.6 ms	489 µs
Pandas + sorted(list())	80 ms	407 µs
Pandas + sorted()	77.6 ms	1.5 ms
Numpy; built-in sort()	11.1 ms	65 µs
Numpy + sorted()	391 ms	6.63 ms
Numba + quicksort	1540 ms	22.2 ms
Cython + list comprehension	78.9 ms	557 µs

- For numerical operations, Numpy gave the best result.
- Built-in functions of Pandas and Numpy were faster, than custom implementations.
- Built-in function of Pandas and Numpy can be faster, than Python's standard implementation (for example using Pandas' *list.sum()* was faster than using Python's *sum(list)*).
- Numba was according to the experiments among the slowest.
- Cython improved the speed of standard Python code.
- The standard Python was better in sorting strings.

5 CONCLUSION

This research explored different solutions for a selection of experiments. Python in combination with Pandas, Numpy, Numba and Cython have been tested on speed, answering how the processing speed of medium-size CSV files can be improved. Based on the literature and experiments with string operations, numerical operations and sorting, a number of conclusions can be formulated. Firstly, multiprocessing instead of multi-threading should be used. Because of Python's GIL, multi-threading has no speed increase, while multiprocessing does. Secondly, for numerical operations always use Numpy, because it is by far the quickest. Thirdly, check if built-in functions from packages are faster than standard implementation in Python. Pandas' *list.sum()* is faster than using Python's *sum(list)*, but Python's *sorted()* is faster than Pandas' *.sort_values()* (with string operations). Finally, for non-numerical operations use Cython or regular Python, because the other packages perform worse.

6 DISCUSSION AND FUTURE WORK

During the research a couple of things stood out.

Research and environment

This research has been limited to speed, while memory usage, CPU usage and the loading speed of data could be an interesting addition. Besides, the experimentation was executed through Jupyter Notebooks, while raw python may perform better with for example Numba. For future research one could also take a look into the usage of more test resources. *Cprofiling* or *line_profiling* are more precise because it gives the speed line for line. These resources weren't used in this research, because the goal was not to micro-optimise code, but give the reader insights in how different implementations can perform better or worse.

Using Numba

After researching the literature, the Numba package seemed easy to work with, but in practice it is quite difficult to use. A lot of standard Python functions are not implemented, the tutorials only showed some basic numerical functions and its errors are rather unclear. The use of Numba felt like doing a difficult puzzle, rather than the ease described in the literature. Therfore, future research should look at non-numerical implementations of Numba.

Cython

How difficult Numba was, so easy was the use of Cython in combination with standard Python. Adding the types manual wasn't much work, and added the benefit of quickly recognizing the input and output types. During the experimentation, combining Cython and Numpy didn't work that well because of the custom Numpy types. Future research should look at the possibility of custom types in Cython and test if this improves speed even further.

Finally, this paper focused on a small part of optimisation, in reality there are many ways to improve speed. Through faster algorithms, better computers, using graphics cards and many more. Besides, only a couple of the available Python packages have been used. There are more packages available which have not been discussed. Future research could dive into one of these different aspects.

REFERENCES

[1] [n.d.]. PyCOMPS Examples. https://institutefordiseasemodeling.github.io/Documentation/comps/pycomps_examples.html

[2] 2018. Domo Resource - Data Never Sleeps 6.0. https://www.domo.com/learn/data-never-sleeps-6

[3] 2019. Domo Resource - Data Never Sleeps 7.0. https://www.domo.com/learn/data-never-sleeps-7

[4] 2019. Stack Overflow Developer Survey 2019. https://insights.stackoverflow.com/survey/2019

[5] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D Matsakis. 2007. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 symposium on Dynamic languages*. 53–64.

[6] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2011), 31–39.

[7] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. (2004).

[8] Benedikt Droste. 2019. How To Make Your Pandas Loop 71.803 Times Faster. https://towardsdatascience.com/how-to-make-your-pandas-loop-71-803-times-faster-805030df4f06

[9] Kay Hayen. 2010. What is Nuitka. https://nuitka.net/pages/overview.html

[10] Heisler. 2017. A Beginner's Guide to Optimizing Pandas Code for Speed. https://engineering.upside.com/a-beginners-guide-to-optimizing-pandas-code-for-speed-c09ef2c6a4d6

[11] Randall Hyde. 2009. The fallacy of premature optimization.

[12] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.

[13] Julian John McAuley and Jure Leskovec. 2013. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In *Proceedings of the 22nd international conference on World Wide Web*. 897–908.

[14] Riccardo Murri. 2014. Performance of Python runtimes on a non-numeric scientific code. *arXiv preprint arXiv:1404.6388* (2014).

[15] Abhinav Nagpal and Goldie Gabrani. 2019. Python for data analytics, scientific and technical applications. In *2019 Amity International Conference on Artificial Intelligence (AICAI)*. IEEE, 140–145.

[16] Michael P Plezbert and Ron K Cytron. 1997. Does "just in time"="better late than never"?. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 120–131.

[17] Stanford Network Analysis Project. 2017. Amazon Fine Food Reviews. https://www.kaggle.com/snap/amazon-fine-food-reviews/data

[18] Robert R Schaller. 1997. Moore's law: past, present and future. *IEEE spectrum* 34, 6 (1997), 52–59.

[19] Jacob Schreiber. 2017. Pomegranate: fast and flexible probabilistic modeling in python. *The Journal of Machine Learning Research* 18, 1 (2017), 5992–5997.

[20] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. 2017. PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications* 31, 1 (2017), 66–82.

[21] Shiva Verma. 2020. How Fast Numpy Really is and Why? https://towardsdatascience.com/how-fast-numpy-really-is-e9111df44347

[22] Hai Wang, Zeshui Xu, Hamido Fujita, and Shousheng Liu. 2016. Towards felicitous decision making: An overview on challenges and trends of Big Data. *Information Sciences* 367 (2016), 747–765.

[23] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.