Optimizing Pandas' *Apply* Operations

OLIVIA SAMPLES

College of Computing & Technology, Lipscomb University, Nashville, TN, osamples@mail.lipscomb.edu

1 INTRODUCTION

Pandas *apply* function is a well-known and used data processing tool within Python, allowing users to apply a function to every single value of the Pandas series. However, this operation is essentially applied over a for loop, causing extreme computing time for large datasets. Pandas users have sought to expedite this process using tools such as multiprocessing, parallelization, and vectorization. We will compare these processes in our analysis below.

2 METHOD

2.1 Dataset Description

The dataset used for analysis is "Bar Crawl: Detecting Heavy Drinking Data Set" provided by UCI's Machine Learning Repository. There exist 5 different features describing 14,057,567 records: a timestamp, a participant ID, and three samples from each access of an accelerometer (used to measure alcohol consumption). We focus on the "timestamp" feature by applying the *to_datetime()* function on this column, as the timestamp feature is in Unix Epoch time. This process can be useful for engineering year, month, or day features in a dataset.

2.2 Standard Pandas Data Exploration

The standard way of applying the *to_datetime()* function to the "timestamp" feature is limited to Pandas' slow *apply* function. The runtime of the process without additional optimization tools was *5772.0235* seconds. This standard time will be used to compare three distinct Pandas optimization tools: Multiprocessing, Multiprocessing with Pool, and Swifter.

Note: The macOS Catalina computer used to process this data is equipped with 8GB of RAM and a 2.7 GHz Dual-Core Intel Core i5 processing system. Computers with different specifications will have different runtime results.

3 RESULTS

3.1 Multiprocessing with Process

Multiprocessing allows the function to be processed across all available CPU cores in parallel by working around Python's Global Interpreter Lock (GIL). This should theoretically cut down time because the work is being divided simultaneously. Multiprocessing with Process tends to be better for smaller tasks because it allocates all tasks in memory. The runtime for Multiprocessing with Process was *2170.4605* seconds, about 2.66 times faster than when using Pandas *apply*.

3.2 Multiprocessing with Pool

Multiprocessing with Pool is better for larger tasks because it only allocates executed tasks in memory. Pool allows you to do multiple jobs per process by assigning them to the different CPU cores. Pool was processed with *10* partitions and *4* cores in this analysis. Here, Multiprocessing with Pool performed *2.22* times better than Process with a runtime of *977.0077* seconds.

3.3 Vectorization

Vectorization allows you to apply a function on an entire vector of values at one time, rather than looping over each instance separately. Here, *to_datetime()* is a vectorized function, meaning it should run extremely fast using vectorization. This is correct as the runtime was *0.1446* seconds.

3.4 Swifter

Swifter is the newest optimization tool for pandas that must be run in Python3. While there are many options for expediting the linear operation of *apply* such as vectorization and multiprocessing, Swifter eliminates the need to choose between them. Swifter automatically decides the fastest processing option, starting with vectorization, dask multiprocessing, then pandas *apply* function. The runtime for Swifter was *0.2650* seconds. This is slightly longer than vectorization, most likely due to the time used to decide the fastest processing option.

4 CONCLUSION

It is clear that the time-consuming Pandas *apply* function can be improved using optimization tools. In all cases, the improvements performed better than the standard. Multiprocessing with Pool performed twice as well as Multiprocessing with Process due to the size of the dataset. Swifter performed exponentially better than Multiprocessing, only coming second to Vectorization. Vectorization took *0.1446* seconds compared to the standard's *5772.0235* seconds. It performed so well because the function being used (*to_datetime()*) is a vectorized function which allows the fast transformation. Further tests can be performed to determine the fastest tool to use for non-vectorized functions, but the general best-use case is Swifter because it tests all possible optimizations.

REFERENCES

[1] Carpenter, Jason. April 17, 2018. Swifter — automatically efficient pandas apply operations. Retrieved June 11, 2020 from https://medium.com/@jmcarpenter2/swiftapply-automatically-efficient-pandas-apply-operations-50e1058909f9.

[2] Mane, Priyanka. October 4, 2017. Python Multiprocessing: Pool vs Process – Comparative Analysis. Retrieved June 11, 2020 from https://www.ellicium.com/python-multiprocessing-pool-process/.

[3] Sagav, Abhinav. November 3, 2019. A Hands-on Guide to Multiprocessing in Python. Retrieved June 11, 2020 from https://towardsdatascience.com/a-hands-on-guide-to-multiprocessing-in-python-48b59bfcc89e.

[4] Killian, J.A., Passino, K.M., Nandi, A., Madden, D.R. and Clapp, J., Learning to Detect Heavy Drinking Episodes Using Smartphone Accelerometer Data. In Proceedings of the 4th International Workshop on Knowledge Discovery in Healthcare Data co-located with the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019) (pp. 35-42). Retrieved June 11, 2020 from https://archive.ics.uci.edu/ml/datasets/Bar+Crawl%3A+Detecting+Heavy+Drinking.