Are They What They Claim: A Comprehensive Study of Ordinary Linear Regression Among the Top Machine Learning Libraries in Python

Sam Johnson[1], Josh Elms[1], Madhavan K R[1], Keerthana Sugasi[1], Parichit Sharma[1], Hasan Kurban[3,1,2], Mehmet M. Dalkilic[1,2]

[1]Computer Science Department, Indiana University, Bloomington, Indiana, IN, USA
[2]Data Science Program, Indiana University, Bloomington, Indiana, IN, USA
[3]Electrical and Computer Engineering, Texas A&M University at Qatar, Doha, Qatar
Email: {sj110, jmelms, madhkr, ksugasi, parishar, hakurban, dalkilic}@iu.edu

***Abstract*—The Least Squares method is the oldest ML algorithm but remains the most popular and ubiquitous across domains. Now, with powerful, cheap technology and the popular open-source language Python, users of every sort have access to various libraries, all serving ordinary least squares. In this work, we ask the question of whether users can count on the different implementations *ceteris paribus* producing the same results. We conduct a comprehensive survey of current Python implementations of the Least Squares method, providing a comparative analysis of their features and performance (time and space). Additionally, we test models over cumbersome real-world datasets to examine behavior in this decade of very big data. Our investigation covers the most popular and well-established libraries: TensorFlow, PyTorch, scikit-learn, and MXNet. Our results unexpectedly show that sufficient significant differences exist such that users must scrutinize their choices and not confine themselves to a single library.**

1. Introduction

Significant improvements in technology–cheaper, faster, with more memory–have led to a boom in open-source machine learning (ML) algorithms. Interestingly, amidst these changes, the oldest [1], the Least Squares (LS) method [2], remains the most popular and widely used [3] algorithm. As a versatile and efficient optimization technique, LS has undergone improvements, giving rise to a number of variants where many are associated with particular domains [4]. With the increasing interest in data science that leverages big data, software engineering, statistics, and machine learning, many versions of this original algorithm have been created. A natural question arose: given these different packages, did their ordinary LS (OLS) produce uniform results? Outside of experts, people would, understandably, assume there exists no difference. Our aim was to find the truth.

In this paper, we present a comprehensive survey of the current Python implementations of OLS. We highlight their similarities and differences and benchmark their performance. By evaluating these implementations, we aim to provide a valuable resource to the data science community, enabling practitioners and researchers to make informed decisions when selecting the most suitable implementation for their specific needs. Furthermore, this study contributes to the understanding of the OLS method's role in contemporary ML and helps pave the way for future developments and improvements. We chose Python due to its popularity for open-source ML algorithms. Python has become the go-to choice for most practitioners and researchers due to its readability, flexibility, and vast ecosystem of freely available libraries and tools [5].

The remainder of this paper is organized as follows: Section 2 provides an overview of popular Python ML libraries for the OLS method. Section 3 presents a detailed review of the current Python implementations, covering well-established libraries such as TensorFlow [6], PyTorch [7], scikit-learn [8] and MXNet [9]. Section 4 offers a comparative analysis of the surveyed implementations, including run-time and memory comparisons and performance on degenerate data. Section 5 concludes with a summary of the key findings and our future research directions.

2. Background

We do not fully develop the topic here because of space constraints, but instead point to [10] and give the following: for data $D = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ the aim is to minimize the residual sum of squares (RSS)

$$RSS(\beta_0, \beta_1, \ldots, \beta_N) = \sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2$$

where $|x| = p$ and β are coefficients to be determined. While there exist analytic solutions (shown using calculus or linear algebra), implementations, on the other hand, face real-world problems as we will see with a few different approaches.

2.1. Popular Machine Learning Libraries

TensorFlow is an end-to-end ML platform for producing artificial neural networks (ANN). Thanks to TensorFlow's flexibility and robust computation model, it has quickly become the most popular ML library in the Python ecosystem,

Algorithm	Library	Time Complexity (flops)
Cholesky Decomposition (NE-CD)	TensorFlow	$O(mn^2 + n^3)$
Complete Orthogonal Decomposition (COD)	TensorFlow	$O(2mnr - r^2(m+n) + 2r^3/3 + r(n-r))$
QR with Column Pivoting (QRCP)	PyTorch	$O(4mnr - 2r^2(m+n) + 4r^3/3)$
QR Factorization (QR)	PyTorch	$O(2mn^2 - 2n^3/3)$
Singular Value Decomposition (SVD)	PyTorch	$O(4mn^2 + 8n^3)$
SVD Divide-and-Conquer (SVDDC)	scikit-learn, MXNet, PyTorch	$O(mn^2)$

Table 2: The computational complexity of least squares algorithms implemented by popular Python ML libraries [19] [20] [21] [22]. In keeping with ML formalism, we use m to indicate the number of rows in the input matrix, n as the number of columns and r as the rank.

widely used in both academia and industry [11]. TensorFlow implements its own linear algebra library in which regression algorithms can be created. PyTorch is a high-level ML framework with two primary features: a NumPy-like tensor computation suite and a module to handle the creation of ANNs. Beyond ANNs, PyTorch also offers a linear algebra library that can be used to construct regression algorithms. Scikit-learn deploys 80+ ML algorithms, including 40+ for regression, and is one of the most widely-used Python libraries in commercial and academic applications [12]. As with most Python packages that require fast computation, scikit-learn largely relies on NumPy and SciPy (array math and scientific computing, respectively). Scikit-learn builds on top of these libraries with their own implementations of optimized regression algorithms. MXNet is an open-source ML library intended to support the generation and deployment of ANNs by blending the advantages of preceding ML software [9]. MXNet is built for scalability, fast computation, and usage across a breadth of platforms [13]. It relies on NumPy and SciPy for its linear regression routines as well.

2.2. Selection Process

It is estimated that 85% of data scientists use Python [14], so our paper is focused on widely-used ML packages in the Python ecosystem. To identify the most popular machine learning libraries in Python with sufficient rigor, we conducted an extensive exploration using the Libraries.io dataset, a project from Tidelift [15]. The dataset contains information on 4,612,919 software packages that are publicly available on the internet, last updated on January 12, 2020.

Library	Creator	★	Rank
TensorFlow	Google Brain	173,880	11
PyTorch	Meta AI	66,165	73
scikit-learn	Cournapeau	54,076	109
MXNet	Chen *et al.*	20,397	373

Table 1: Most popular Python ML libraries [6] [7] [8] [9]. Popularity is determined via Github stars. When a user wants to show appreciation for a repository or mark it for later use, they will 'star' ($\star$) the repository [16]. The star ranking is relative to all repositories on Github [17].

We trim the set of packages to those relevant to ML by only including packages with ML-related keywords in their description (*e.g.* "artificial intelligence", "big data"). This filter narrows our search to 187,886 packages. From these, we filter again to yield only those with 10,000 or more *stars* on Github at the date of the dataset's publication. Although impossible to completely validate, stars reflect popularity and, thus, usage. We then comb through the remaining 48 packages to find only well-maintained software with modeling capabilities and a Python API. We credit Li and Bezemer for inspiration for this search process [18]. Our paper will focus on the Python ML packages that fit our criteria with the greatest amount of stars on Github as of May 7, 2023.

3. Least Squares Implementations

TensorFlow implements its own linear algebra package and the least squares function from this module computes solutions to least squares problems using one of two algorithms. One routine solves the normal equations using Cholesky decomposition (NE-CD) and the other performs an optimized complete orthogonal decomposition (COD) algorithm [23]. PyTorch provides several options for the least squares solver in its linear algebra package. The default setting is a LAPACK (standard linear algebra routines in Fortran) driver 'gelsy' that performs QR factorization with column pivoting (QRCP). If the feature matrix A is full rank (*i.e.*, $\text{rank}(A) = min(\text{rows}, \text{columns})$), then PyTorch recommends LAPACK driver 'gels', a QR factorization (QR) to solve the least squares problem. This is the default setting when PyTorch has access to a GPU. PyTorch also provides two other options for solvers, 'gelsd' and 'gelss'. The 'gelsd' LAPACK routine implements Householder Bidiagonalization and singular value decomposition with a divide-and-conquer method (SVDDC), and the 'gelss' driver performs complete singular value decomposition (SVD). Scikit-learn's linear regression model is a wrapper for SciPy's least squares function which, in its default setting, also makes use of LAPACK's 'gelsd' solver. MXNet does not implement its own linear algebra package, but instead directly interfaces with NumPy's linear algebra package. The NumPy linear algebra library is a lightweight version of SciPy's linear algebra library [24] and the least squares function uses the same LAPACK driver, 'gelsd'. MXNet was omitted from most experiments in this paper due to dependency conflicts. The theoretical computational complexity of these various least squares algorithms is recorded in Table 2.

	Tensorflow		PyTorch				scikit-learn
Rows	NE-CD	COD	QRCP	QR	SVD	SVDDC	SVDDC
10^1	1763.713	958.242	2608.284	274.621	40.926	836.778	2520.289
10^2	44.961	29.567	18.142	16.044	6.975	35.279	94.538
10^3	5.238	3.86	1.633	1.63	0.888	3.78	8.87
10^4	0.833	1.592	0.313	0.289	0.148	0.608	1.7
10^5	0.583	1.358	0.167	0.154	0.074	0.308	0.982
10^6	0.649	3.036	0.279	0.262	0.131	0.525	1.726
10^7	0.661	3.466	0.363	0.326	0.161	0.645	1.867
10^8	0.664	3.778	0.391	0.368	0.195	0.783	2.004
10^9	0.766	5.982	0.436	0.396	0.2	0.798	2.179

Table 3: Runtime ratio $\frac{\text{Actual}}{\text{Theoretical}}$ on Quartz with M = 10^9 rows. Values greater than one have been highlighted to indicate unfaithfulness to a theoretical runtime bound, calling attention to poor performers scikit-learn (SVDDC) and TensorFlow (COD).

4. Methodology and Experimental Results

4.1. Runtime Comparison

The complexity analysis of LS is well-known, but engineering requirements can affect these bounds. In this part of our work, we investigate whether runtimes deviate from what is known about complexity. We settled on the usage of synthetic data for profiling these solvers, since it gives us much more control in terms of data properties. The only specifications for the data are M rows and N columns, and all points are drawn from a standard normal distribution. In this experiment, $N = 10$ to focus on the effect that M has on the implementation's runtime. This decision is supported by the prevalence of datasets in ML where $M \gg N$. Assuming that the upper bound for M is 10^L, the row counts m for the experiment are $[10^1, 10^{1.5}, 10^2, \ldots, 10^L]$.

An algorithm's actual runtime is measured with the process_time_ns function from the Python standard library [25]. We decided prior to experimentation to record process time rather than total time. This is an attempt to prevent external loads on the system from influencing our measurements. While the use of Python's process_time_ns prevents us from considering non-CPU operations such as input/output and data transfer, we've deemed this to be an acceptable concession to draw accurate conclusions across platforms. While actual runtimes are measured in seconds, time complexities for regression algorithms are measured in the number of mathematical operations that are required for the algorithm (floating point operations a.k.a. flops). In order to determine the theoretical runtime in seconds, it is necessary to establish both the time complexity of the algorithm and the speed at which the computer can perform those operations, measured in flops per second. This information is used in the simple equation $T = O \cdot P$, where T is the estimated runtime in seconds (s), O is the time complexity of the algorithm in flops, and P is the speed of the processor in flops per second (flop/s). O is shown for each algorithm in Table 2 and P is shown for each relevant processor in Table 7. We selected the Linpack Benchmark [26] for profiling because it is the standard method of determining the speed of a processor in flop/s. The test consists of recording the time to solve large, dense matrices, a process for which the time complexity is well-established. In this paper, the C version of the test is compiled with the default options on each system to establish the processor's average speed in flop/s. The Linpack test produces output in million flop/s (MFLOPS). In this paper, we present results for experiments conducted on Indiana University's Quartz supercomputer [27], with most of our experiments being performed on one core of one node. This core was clocked at 870 MFLOPS in testing and detailed information about each processor tested can be found in the Supplementary Materials.

Figures 1, 2, 3, and 4 demonstrate how the actual runtimes of different OLS implementations relate to their theoretical bounds. This relationship is also captured in Table 3. Our experiment was able to check runtime claims made by the respective libraries. TensorFlow claims that their NE-CD implementation is six to seven times faster than their COD implementation [23]. Our results are consistent with that claim. SciPy mentions that the LAPACK solver 'gelsy', the solver for PyTorch (QRCP) is "faster on many problems" than OLS with LAPACK solver 'gelsd', the solver for PyTorch (SVDDC) and scikit-learn (SVDDC). PyTorch (SVDDC) performed better than PyTorch (QRCP) in our experiment, but scikit-learn's implementation of SVDDC performed much worse. Scipy also states that LAPACK solver 'gelss' is generally slow [28]. PyTorch (SVD) makes use of this LAPACK solver, and we observe that it performs among the quickest of the solvers. The actual runtimes for all solvers on the Quartz system are visualized in Table 9 in Supplementary Materials.

4.2. Memory Comparison

As big data becomes actually big, ML algorithms face system challenges that, a decade ago, seemed distant. Conducting experiments to test how size affects regression is, therefore, necessary.

Recording the memory usage of a program requires the use of specialized tools that track the memory allocated by the program during its execution. Memray is a Python-

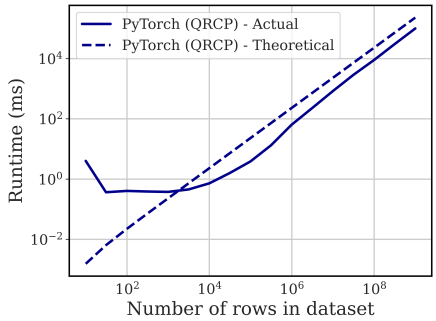


Figure 1: Actual vs. Theoretical runtime on a log scale makes the initial non-least squares operations apparent. See failure to adhere to expected runtime when $M < 10^3$.

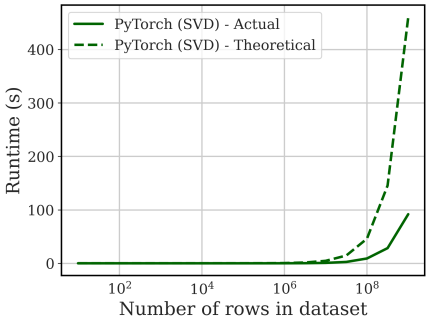


Figure 2: Actual vs. Theoretical runtime for a solver that outperforms its stated time complexity by some constant. This occurs in both PyTorch (QR) and PyTorch (QRCP) as well, though to a lesser extent.

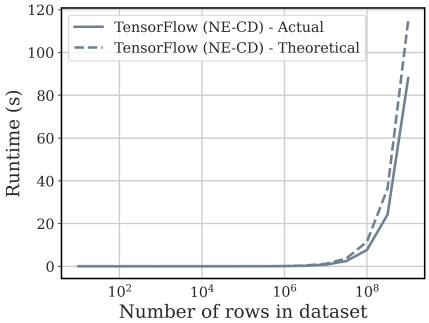


Figure 3: Actual vs. Theoretical runtime for a solver that approximately matches its stated time complexity. This occurs in PyTorch (SVDDC) as well.

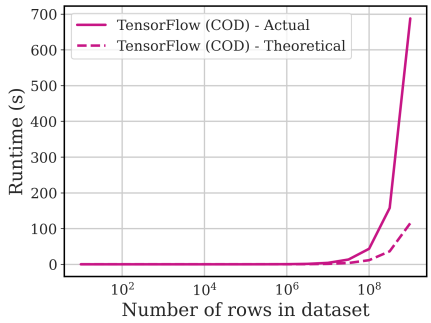


Figure 4: Actual vs. Theoretical runtime for a solver that fails to meet its expected runtime. This occurs in scikit-learn (SVDDC) as well, though the difference is not as drastic.

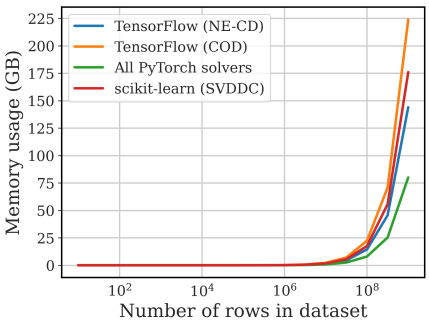


Figure 5: Maximal memory usage (GB) of each implementation is given for a number of rows. Each of the PyTorch algorithms performs identically in this regard and with significantly lower memory usage than the other solvers.

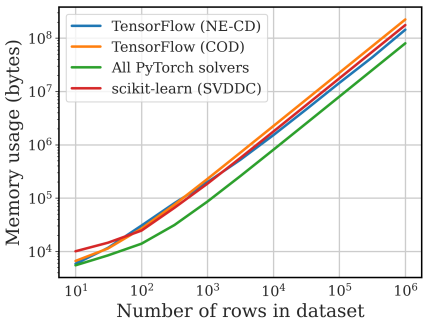


Figure 6: Maximal memory usage (bytes) of each solver is given for a number of rows on a log scale. Similar to the time complexity experiment, a small amount of memory overhead is visible for non-least squares allocations. This is apparent for regression when $M < 10^2$.

specific memory profiling tool capable of recording a program's maximal memory usage by tracing the allocations made not only from within Python but also from within the external functions called by the Python program [29]. It allows for the individual allocations made by C and Fortran code to be tracked. We apply the same experimental design as before. Unfortunately, with the lack of documentation available, no explicit comparison can be made between actual and expected memory usage. We report, therefore, only actual memory usage. The upper limit is determined by physical memory availability: the largest system available to us is Indiana University's Quartz, with 512 gigabytes of system memory per node limiting datasets to 10^9 rows. Figure 6 shows that, for a small dataset, there are varying amounts of startup memory associated with each OLS implementation. As the dataset size increases, the algorithms approach a consistent ratio, with TensorFlow (COD) consuming the most memory, followed by scikit-learn (SVDDC), TensorFlow (NE-CD), and lastly every OLS algorithm offered in PyTorch. It is interesting to observe that PyTorch (SVDDC) and PyTorch (SVD) consume the same amount of memory in this experiment despite PyTorch's recommendation to use SVD if memory usage issues arise while running SVDDC [30]. Memory usage is also visualized in Figure 5 and recorded explicitly for all tested processors in Supplementary Materials.

4.3. Performance on Subsections of Circular Data

We designed this experiment to test the hypothesis that the OLS algorithms we gathered would respond differently to modifications in degenerate input data. The underlying assumption in linear regression modeling is linearity in the data; to stress the solvers, we generate data that represents evenly-spaced points along the perimeter of a circle with a diameter of 10 units in 2D Euclidean space. The circle is then partitioned into n arcs of equal length, forming the set S. Subsequently, to capture every possible combination of arcs, the powerset of the set S is generated (excluding the empty set). The powerset is then pruned so that the elements representing less than half of a circle are removed. The remaining elements e are

$$\{e \mid e \in 2^S \ \wedge \ |e| \geq \frac{1}{2}n\}$$

For each element of the remaining set, a dataset is constructed to represent the corresponding partial circle. This

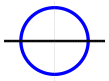
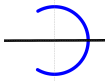
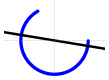
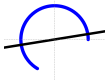
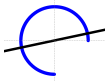
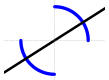
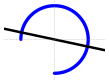
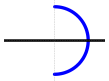
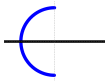
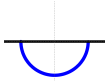
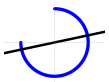
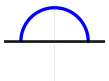
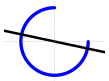
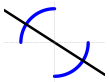
Data	Rotation					
Full Circle	0°	5°	15°	30°	60°	90°
	50.00	50.00	50.00	50.00	50.00	50.00
3 Arcs	0°	5°	15°	30°	60°	90°
	60.43	60.23	58.47	53.53	43.39	39.57
	43.49	42.35	40.66	39.68	43.29	53.22
	43.32	44.63	47.77	53.27	60.32	53.49
4 Arcs	0°	5°	15°	30°	60°	90°
	47.69	49.51	53.35	58.45	58.55	47.81
	29.62	33.28	43.40	65.98	66.57	29.86
	47.81	46.11	43.21	40.36	40.31	47.69
	50.00	50.00	50.00	50.00	50.00	50.00
	50.00	50.00	50.00	50.00	50.00	50.00
	50.00	50.00	50.00	50.00	50.00	50.00
	47.69	49.51	53.35	58.45	58.55	47.81
	50.00	50.00	50.00	50.00	50.00	50.00
	47.81	46.11	43.21	40.36	40.31	47.69
	29.86	26.87	22.62	19.19	19.14	29.62

Table 4: OLS performance over circular data composed of non-overlapping arcs (# of arcs - 0, 3, 4). The partial circle data with overlaid regression line is depicted in the leftmost column, and MSE of the regression line is recorded as function of rotation in the other columns.

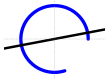
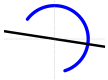
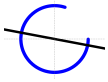
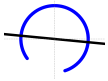
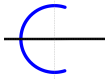
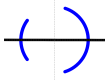
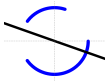
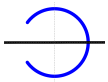
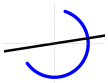
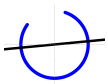
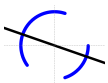
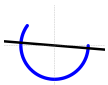
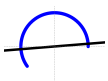
Data	Rotation					
5 Arcs	0°	5°	15°	30°	60°	90°
	51.14	52.85	56.05	59.14	55.25	45.60
	36.92	39.33	45.41	57.18	69.43	47.82
	51.30	49.92	47.30	44.21	42.32	46.50
	51.27	49.57	46.44	42.83	40.67	45.50
	41.84	41.22	40.58	41.09	47.22	56.88
	57.80	57.65	56.43	52.96	45.29	42.20
	29.59	29.71	30.76	34.54	52.21	70.41
	62.35	58.31	50.11	40.06	30.37	31.29
	62.07	65.70	70.11	66.56	43.64	31.36
	59.46	59.27	57.70	53.31	44.09	40.54
	51.20	52.59	55.14	57.55	54.51	46.58
	41.79	42.58	44.7	49.12	58.25	56.97
	37.08	35.04	32.01	29.79	32.68	47.55
	43.34	42.81	42.24	42.69	47.96	55.79
	43.30	43.99	45.84	49.54	56.86	55.86

Table 5: OLS performance over circular data composed of non-overlapping arcs (# of arcs - 5). The best performance is observed when the gaps in the circular data are positioned at the top or bottom of the circle.

	TensorFlow		PyTorch				scikit-learn	MXNet
Data	NE-CD	COD	QRCP	QR	SVD	SVDDC	SVDDC	SVDDC
Blog Feedback [31]	Failed	926.9	3439.907	Failed	937.748	937.747	926.9	926.921
Communities and Crime [32]	0.019	0.019	0.104	899.167	0.076	0.076	0.019	0.019
Facebook Comment Volume [33]	Failed	10900.276	27776.551	Failed	13029.275	13029.275	10900.276	10900.276
Geographical Origin of Music [34]	Failed	2376.676	2615.101	Failed	2376.675	2376.673	2376.676	2376.676
Hailstone [35]	Failed	0.254	0.403	7.894×10^6	0.262	0.262	0.254	0.254
Hourly Energy Demand [36]	Failed	87.04	149.759	Failed	141.445	141.445	87.04	87.04
KEGG Metabolic Pathway [37]	Failed	5.737	2.888×10^5	5.737	628.818	628.818	5.737	5.737
Online News Popularity [38]	Failed	1.341×10^8	2.826×10^8	3.618×10^9	1.348×10^8	1.348×10^8	8.075×10^{17}	1.341×10^8
Residential Building [39]	Failed	1653.68	4.568×10^5	8.856×10^{11}	4078.649	4086.893	1653.68	1653.68
Superconductivity [40]	310.574	310.574	832.526	310.574	503.458	503.458	310.574	310.574

Table 6: Performance of OLS algorithms over high-dimensional datasets, measured in MSE. If an OLS implementation failed to fit a model on any CV fold, the algorithm is said to have 'Failed' for that dataset. A cell is shaded according to that OLS algorithm's performance relative to the performance of the other OLS implementations on that particular dataset; lighter shading indicates better performance.

process is carried out for each value of $n \in \{3, 4, 5\}$. Finally, each of the generated datasets is multiplied by a rotation matrix for every rotation $r \in \{0°, 5°, 15°, 30°, 60°, 90°\}$. The OLS algorithms are then applied to each of the 168 resulting datasets. The training error are recorded in Tables 4 and 5 for each dataset, along with a plot of each partial circle and its regression line. We note that for approximately 83% of the datasets, a rotation introduced a change in the error, which is to be expected. There was no notable difference in error observed between the OLS algorithms we tested for any combination of rotations or subsections, which leads us to reject the prenominate hypothesis.

4.4. Performance on High-Dimensional Data

As data grows, so do features. In this experiment, we explore how feature size affects OLS. Ten publicly available datasets were selected for both regression suitability and number of features (> 25). Non-numerical features were removed, and missing values were imputed with zeros. We thought this simple protocol would least perturb the findings. The eight OLS algorithms were then trained on these datasets. The validation Mean Squared Error (MSE) was computed for each of the ten models produced in CV training and then averaged to yield one error metric for the entire dataset. This process was completed for each dataset, and the results are displayed in Table 6. On several occasions, while attempting to train a model on a dataset, a library would raise a non-descriptive error and cease training. These failures to generate a model over a particular dataset are also recorded in 6. TensorFlow (NE-CD) failed to generate a model for eight out of the ten datasets. PyTorch (QR) failed to generate a model for four datasets and generated highly inaccurate models for three other datasets. On each dataset, several of the models achieve the same, low error. TensorFlow (COD) and MXNet (SVDDC) achieved this low error in each of our experiments, performing well relative to other algorithms, and scikit-learn (SVDDC) performed well on all data except the Online News Popularity dataset. This is surprising, given that MXNet and scikit-learn rely on the same LAPACK solver.

5. Summary and Conclusions

In this paper, we asked what appeared to be a straightforward question: given the simplicity of OLS, do the most popular libraries' various implementations behave the same? The results make apparent the value of this question. We observe that TensorFlow (COD) and scikit-learn (SVDDC) do not adhere to their respective theoretical bounds for runtime. We approximate the constant value asymptote of the ratio between actual runtime and theoretical runtime for each solver in the bottom row of Table 3. We observe that memory usage scales similarly for each OLS implementation, with PyTorch algorithms performing best.

Most puzzling is the disagreement among the implementations for high-dimensional data. There are a number of interesting possibilities: the original algorithm needs attention, the implementations have differing innate limitations, the OS is affecting the answers, etc. TensorFlow (COD) and MXNet (SVDDC) consistently perform well, and PyTorch (QRCP) and TensorFlow (NE-CD) may not be well-equipped to handle high-dimensional data.

Rather than a single library with a single implementation, a number of different OLS implementations should be used to get a consensus of results. Perhaps validating open-source datasets will help with OLS verification. Our rather mundane question has yielded a number of others–perhaps some general survey of the top ten most popular ML algorithms and their respective implementations should be conducted. Future work includes applying our profiling framework to algorithms relevant in deep learning, such as backpropagation and stochastic gradient descent. The performance of OLS implementations of lesser-known ML libraries could also be measured. Additionally, we want to develop a more rigorous memory profiling experiment at the level of numerical computation libraries. We conclude with what is likely the most controversial: perhaps a new set of time and memory complexities that reflect contemporary technology as it works in the wild is needed rather than relying solely on traditional ones–then users could gauge more effectively the increasing choices for ML.

References

[1] A.-M. Legendre, *Nouvelles méthodes pour la détermination des orbites des comètes [New Methods for the Determination of the Orbits of Comets] (in French)*, 1805. [Online]. Available: https://babel.hathitrust.org/cgi/pt?id=nyp.33433069112559&view=1up&seq=9

[2] S. M. Stigler, "Gauss and the invention of least squares," *The Annals of Statistics*, pp. 465–474, 1981.

[3] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[4] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[5] J. VanderPlas, *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media, Inc., 2016.

[6] M. Abadi and et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011.

[9] T. Chen, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems." in *Proceedings of Neural Information Processing Systems, Workshop on Machine Learning Systems, 12 December Montreal, Canada* [Online]. Available: arXiv, [Accessed: Jan 5, 2023], 2015. [Online]. Available: https://arxiv.org/

[10] T. Hastie, R. Tibshirani, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*, 2nd ed. New York, Springer, 2009.

[11] K. Katsiapis, "Towards ml engineering: A brief history of tensorflow extended (tfx)," *The TensorFlow Blog*. [Online]. Available: https://blog.tensorflow.org/2020/09/brief-history-of-tensorflow-extended-tfx.html

[12] "About us," 2023, [Accessed: November 20, 2022]. [Online]. Available: https://scikit-learn.org/stable/about.html

[13] "Apache mxnet," 2023, [Accessed: January 12, 2023]. [Online]. Available: https://www.nvidia.com/en-us/glossary/data-science/mxnet/

[14] "2021 state of data science," Anaconda Inc., 1108 Lavaca St Suite 110-645, Austin, TX 78701, Tech. Rep., July 2021.

[15] J. Katz, "Libraries.io open source repository and dependency metadata (1.6.0)," 2020, zenodo [Dataset] [Accessed: November 19, 2022]. [Online]. Available: https://doi.org/10.5281/zenodo.3626071

[16] "Saving repositories with stars," 2023, [Accessed: November 20, 2022]. [Online]. Available: https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars

[17] "Compare oss projects," [Accessed: May 7, 2023]. [Online]. Available: https://ossrank.com/compare?project_id=342&projects=341%2C44%2C354%2C&commit=Compare

[18] H. Li and C. Bezemer, "Studying popular open source machine learning libraries and their cross-ecosystem bindings." [Online]. Available: https://arxiv.org/

[19] G. Golub and C. Van Loan, *Matrix Computations*, ser. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 1996, pp. 206–274. [Online]. Available: https://books.google.com/books?id=mlOa7wPX6OYC

[20] N. J. Higham and R. S. Schreiber, "Fast polar decomposition of an arbitrary matrix," *SIAM Journal on Scientific and Statistical Computing*, vol. 11, no. 4, pp. 648–655, 1990.

[21] "Developer reference for intel® oneapi math kernel library for c," 2023, [Accessed: March 30, 2023]. [Online]. Available: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2023-0/gelsd.html#GUID-EC167C45-1A4E-4D3C-8652-9B48C788CDF0

[22] "1.1 linear models – scikit-learn 1.2.2 documentation," 2023, [Accessed: April 3, 2023]. [Online]. Available: https://scikit-learn.org/stable/modules/linear_model.html

[23] "tf.linalg.lstsq — tensorflow v2.12.0," March 2023, [Accessed: March 30, 2023]. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/linalg/lstsq#args

[24] "Lite version of scipy linalg." [Online]. Available: https://github.com/numpy/numpy/blob/v1.24.0/numpy/linalg/linalg.py#L2150-L2307

[25] "time.process_tim_ns — python 3.10," March 2023, [Accessed: April 29, 2023]. [Online]. Available: https://docs.python.org/3/library/time.html#time.process_time_ns

[26] J. Dongarra, "Performance of various computers using standard linear equations software," *Computer Science Technical Report*, vol. Number CS - 89 – 85, June 2014. [Online]. Available: http://www.netlib.org/benchmark/performance.ps

[27] https://kb.iu.edu/d/qrtz, this research was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute [accessed 5-May-2023].

[28] "scipy.linalg.lstsq," [Accessed: May 7, 2023]. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lstsq.html

[29] "Overview - memray," [Accessed: April 29, 2023]. [Online]. Available: https://bloomberg.github.io/memray/overview.html

[30] "torch.linalg.lstsq – pytorch 2.0 documentation," 2023, [Accessed: March 30, 2023]. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.linalg.lstsq.html#torch.linalg.lstsq

[31] K. Buza, *Feedback Prediction for Blogs*, 10 2014, pp. 145–152.

[32] M. Redmond and A. Baveja, "A data-driven software tool for enabling cooperative information sharing among police departments," *European Journal of Operational Research*, vol. 141, pp. 660–678, 09 2002.

[33] K. Singh, R. K. Sandhu, and D. Kumar, "Comment volume prediction using neural networks and decision trees," in *IEEE UKSim-AMSS 17th International Conference on Computer Modelling and Simulation, UKSim2015 (UKSim2015)*, Cambridge, United Kingdom, mar 2015.

[34] F. Zhou, Q. Claire, and R. D. King, "Predicting the geographical origin of music," in *2014 IEEE International Conference on Data Mining*, 2014, pp. 1115–1120.

[35] C. Kirkpatrick, "Hail proximity soundings from the north american mesoscale model."

[36] N. Jhana, "Hourly energy demand generation and weather," https://www.kaggle.com/datasets/nicholasjhana/energy-consumption-generation-prices-and-weather.

[37] S. A. Muhammad Naeem, "Kegg metabolic reaction network (undirected) data set," https://archive.ics.uci.edu/ml/datasets/KEGG+Metabolic+Reaction+Network+%28Undirected%29.

[38] K. Fernandes, P. Vinagre, and P. Cortez, "A proactive intelligent decision support system for predicting the popularity of online news," in *Progress in Artificial Intelligence*, F. Pereira, P. Machado, E. Costa, and A. Cardoso, Eds. Cham: Springer International Publishing, 2015, pp. 535–546.

[39] M. H. Rafiei and H. Adeli, "A novel machine learning model for estimation of sale prices of real estate units," *Journal of Construction Engineering and Management*, vol. 142, p. 04015066, 08 2015.

[40] K. Hamidieh, "A data-driven statistical model for predicting the critical temperature of a superconductor," *Computational Materials Science*, vol. 154, pp. 346–354, 2018.

System	Processor	Mem. (GB)	MFLOPS
Quartz*	AMD EPYC 7742 64-Core Processor	512	870
Carbonate*	Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz	256	750
2020 MacBook Pro†	Intel(R) Core(TM) i7-1068NG7 CPU @ 2.30GHz	32	1250

Table 7: System details for the processors used in the experiment.* for more details, see iu.edu † for more details, see apple.com

	TensorFlow		PyTorch				scikit-learn
Rows	NE-CD	COD	QRCP	QR	SVD	SVDDC	SVDDC
10^1	0.006	0.007	0.005	0.005	0.005	0.011	0.01
10^2	0.03	0.027	0.014	0.013	0.013	0.018	0.025
10^3	0.198	0.228	0.086	0.085	0.085	0.09	0.186
10^4	1.516	2.244	0.806	0.805	0.805	0.81	1.77
10^5	14.476	22.404	8.006	8.005	8.005	8.01	17.61
10^6	144.076	224.004	80.006	80.005	80.005	80.01	176.01
10^7	1440.076	2240.004	800.006	800.005	800.005	800.01	1760.01
10^8	14400.076	22400.004	8000.006	8000.005	8000.005	8000.01	17600.01
10^9	144000.076	224000.004	80000.006	80000.005	80000.005	80000.01	176000.01

Table 8: Maximal Memory Usage (in MB) for each implementation on Quartz with M = 10^9 rows.

	TensorFlow		PyTorch				scikit-learn
Rows	NE-CD	COD	QRCP	QR	SVD	SVDDC	SVDDC
10^1	4.055	0.734	3.996	0.421	0.564	0.962	2.897
10^2	0.568	0.328	0.403	0.357	0.385	0.406	1.087
10^3	0.608	0.442	0.374	0.373	0.417	0.434	1.02
10^4	0.958	1.829	0.72	0.664	0.681	0.699	1.954
10^5	6.702	15.607	3.85	3.551	3.399	3.536	11.283
10^6	74.646	348.966	64.039	60.296	60.363	60.361	198.403
10^7	759.359	3983.838	834.784	750.175	738.807	741.29	2145.786
10^8	7631.434	43425.057	8991.294	8466.698	8974.244	9001.976	23028.82
10^9	88077.922	687569.091	100164.22	91116.327	91747.932	91761.323	250507.101

Table 9: Raw runtime (in ms) of OLS implementations on Quartz with M = 10^9 rows.

Package	Version
scikit-learn	1.2.2
PyTorch	1.12.1
TensorFlow	2.10.0
MXNet	1.9.1

Table 10: Package versions used. These may have conflicting dependencies. If so, the experiments can be completed using separate environments for the packages.

6. Supplementary Results

Table 7 contains detailed information on various types for each system that was used in the Runtime and Memory experiments. The results of these experiments can be found at https://github.com/sej2020/are-they-what-they-claim-supplementary-materials. Table 8 depicts Maximal Memory Usage on the Quartz supercomputer, and Table 9 displays the actual runtime of OLS implementations on the Quartz supercomputer.