DS-GA 1019 - Advanced Python for Data Science - Course Project

Jennifer Rodriguez-Trujillo
NYU CDS (2023)
jr5951@nyu.edu

Joseph Schuman
NYU CDS (2023)
js12580@nyu.edu

Khevna Parikh
NYU CDS (2023)
kp2936@nyu.edu

Kristin Mullaney
NYU CDS (2023)
kmm9492@nyu.edu

Rodrigo Kreis de Paula
NYU CDS (2023)
rk4197@nyu.edu

Sarvesh Patki
NYU CDS (2023)
ssp6603@nyu.edu

1 Introduction

Our project goal is to enhance the efficiency of the code that supports an existing logistic regression model. To achieve this, we utilize the techniques taught in class to optimize the code's performance without changing the underlying model. The improved code will enable faster predictions, which will enhance the practical value of the logistic regression model in real-world applications. The model is used to predict whether a customer will default on their credit card payments within 120 days, which is essential for managing risk in a consumer lending business. By optimizing our code, we can effectively analyze the extensive and complex data to quickly identify future defaults, ultimately leading to cost savings for businesses and consumers. We utilize techniques such as line profiling, minimizing repetitive for-loops, and vectorization through NumPy, Python Jax, and Numba to improve the performance of our code. Our focus is not on the actual model itself, but rather how well we can optimize our code to process the data set efficiently.

Our project was based on the code of our CDS colleague, Adeet Patel, who used his work for his final project in the same course last year. We used his code as our baseline and expanded upon it using the techniques we learned throughout the course. Adeet's code provided a solid foundation for our project, and we were able to enhance it significantly using our newly acquired knowledge. His contributions were instrumental in helping us achieve our project's objectives, and we are grateful for the opportunity to learn from his work.

All codes we developed and that were used for this Project are publicly available on the GitHub Repository (link here).

Keywords: *Parallel computing - Python Jax - NumPy - Cython - Numba*

2 Data set

The data set used in this project is a large and complex one, consisting of 5,531,451 records that track the credit card statement histories of different customers. The data set contains 190 variables, which are anonymized and normalized for privacy reasons. These variables are grouped into different categories, such as delinquency, spending, payment, balance, and risk. Out of these variables, only three are categorical - the customer_ID column, S_2 (which is the date-time), and D_63 and D_64 (which are delinquency variables). The values for these variables correspond to the observation of a particular customer's profile in an 18-month window performance period.

To create the target variable, the performance of each customer is observed for an 18-month window after their latest statement, and if they fail to pay their due amount within 120 days, it is considered a default event. The target variable in this data set is binary, representing the "Default Status" of a customer. A value of 1 indicates that the customer did not pay their credit card statement 120 days after it was issued, while a value of 0 indicates that they did pay.

One important thing to note about the data set is that the negative class has been sub-sampled at 5%. This means that the negative class is weighted 20 times higher than the positive class in the scoring metric. This was done to address the class imbalance issue in the data set, where there are significantly more non-default events than default events.

All the data files used in this project can be found on our Google Shared Folder for the project on this link.

3 Baseline Model

The baseline comprises a logistic regression model with Stochastic Gradient Descent (SGD) optimiza-

tion and contains several key functions. Firstly, the sigmoid function is responsible for computing the sigmoid activation for a given input. Secondly, the loss function is in charge of calculating the logistic loss based on the provided weights, input data, and target labels. The gradients function computes the gradients of the logistic loss with respect to the weights and bias. The normalize function is responsible for performing column-wise normalization of the input data. The train function trains the logistic regression model using SGD by dividing the input data into batches with a specified batch size (bs) and training it for a specified number of epochs (epochs) with a given learning rate (lr). The predict function then makes predictions of the class labels for the input data using the trained weights and bias. Finally, the accuracy function computes the accuracy of the predictions, and the compare function trains the logistic regression model, predicts the class labels for the input data, and computes the accuracy of the predictions.

The train function in our logistic regression model takes input data (X), target labels (y), and the following hyperparameters:

- batch size (bs) = 100,
- number of epochs (epochs) = 1000, and
- learning rate (lr) = 0.001

The same hyperparameters were used for both the baseline and optimization implementations to ensure a fair comparison. To reduce computation time, we took a sub-sample of the training data set to run the baseline on. We achieved a 87% accuracy on the training data set. Even with this reduced data set, it still took 2.5 hours to train the model on 100,000 records, indicating the need for optimization techniques.

4 Implementations

In order to optimize the performance of our Python code, we implemented **line profiling** to identify the time and usage of resources in each line. This allowed us to identify bottlenecks and optimize the code for faster execution. However, our training data set presented a unique challenge due to its large size - at 16.3 GB in CSV format, reading in the file took several minutes, significantly slowing down the code execution. To mitigate this issue, we compressed the data set into **Parquet** files, which resulted in a smaller size of 3 GB. This allowed us to read the data much more quickly (approximately 6 seconds), and significantly improved the overall performance of our code.

We used various optimization techniques to improve the efficiency of our logistic regression model. First, we minimized repetitive for-loops and used vectorization through **NumPy** to take advantage of its efficient array operations. We also explored the use of **Python Jax**, a library for high-performance numerical computing. Additionally, we used **Numba**, a just-in-time (JIT) compiler as well as **Cython**, a superset of Python used to improve performance. Lastly, we reduced function calls and overhead to further speed up the training process. By implementing these techniques, we were able to significantly reduce the training time of our model and improve its efficiency.

4.1 Strategy 1: Python Jax

We optimized our logistic regression model by leveraging Jax, a machine learning library developed by Google that is designed to perform efficient computation on hardware accelerators such as **CPUs**, **GPUs**, and **TPUs**. One of the main advantages of Jax is its ability to perform automatic differentiation, which enables us to compute derivatives of our loss function with respect to the weights and biases of our model.

Another key advantage of Jax is its NumPy-like interface that supports automatic differentiation. To take advantage of these features, we used the jnp module provided by Jax instead of NumPy for matrix multiplication and other computations. For instance, we used the jnp.sum and jnp.mean functions instead of their NumPy equivalents. In addition to these optimizations, we took advantage of Jax's ability to automatically broadcast over the input array, eliminating the need to loop over individual elements. We also used Jax's composable functions 'jit' and 'vmap' to vectorize functions where applicable. These functions allow us to compile code for specific hardware, making our computations more efficient.

Our optimized code using Jax demonstrated remarkable performance gains over the baseline implementation. While the baseline took approximately 2.5 hours to run on 100,000 records, our optimized Jax code was able to process the same data set in just 328.63 seconds, which is approximately 5.47 minutes. This represents a significant improvement in performance, with Jax running about 30

times faster than our baseline model.

Jax allowed us to optimize our logistic regression model by leveraging its advanced capabilities for efficient computation on hardware accelerators. By using Jax's composable functions, automatic differentiation, and broadcasting abilities, we were able to significantly improve our model's performance. These approaches were particularly advantageous when working with large data sets, as even small optimizations had a significant impact on performance.

4.2 Strategy 2: NumPy

To further optimize our linear regression model, we re-implemented the baseline with NumPy, a Python library designed in the low-level programming language C. NumPy was designed to be fast for particularly numerical operations on large arrays. Furthermore, NumPy is also built using optimized algorithms to perform mathematical operations.

Given the nature of our data set and the structure of the baseline model, it presented key traits that would benefit from NumPy and vectorization. For instance, np.dot function performs matrix multiplication between two arrays. This function provides the following benefits:

- Simplicity: a simple way to compute matrix multiplication without for-loops
- Speed: the function is already implemented using highly optimized techniques that run much faster than running operations in loops
- Flexibility: NumPy's np.dot function is compatible without existing libraries and can be flexibly used with Pandas, SciPy, and other Python libraries.

Our model contained room for improvement beginning with the way arrays were created to how mathematical operations were created. Through the use of methods such as np.where, np.mean, np.std, and others, we were able to run 100,000 records in 2 minutes and 11 seconds per loop. In other words, NumPy's abilities are efficient. Furthermore, we noted that NumPy allowed us to perform mathematical computations at a faster rate than traditional computational techniques. After observing its computational capacity, we opted to utilize NumPy's model as a baseline when attempting to optimize the model even further.

4.3 Strategy 3: Cython

Once we transformed the baseline model into a vectorized format using NumPy, we explored methods to enhance its performance by leveraging low-level machine code. We experimented with two distinct low-level machine implementations. The first one was Cython, which integrates C implementations within Python for optimized performance. We implemented Cython by specifying datatypes for functions and using 'cdef' and specified datatypes to declare variables. However, we observed that there was no appreciable difference between the NumPy implementation and the Cython performance, with almost identical runtimes. One possible reason for this could be that the code was already well-optimized using NumPy and there were no bottlenecks in the code that Cython could optimize further. Additionally, adding explicit datatype declarations in Cython code can be time-consuming and error-prone, which can negate the benefits of using Cython.

4.4 Strategy 4: Numba

To further optimize the code, which now heavily relies on NumPy, we decided to leverage the Numba library, which is well-known for its ability to accelerate numerical computations. As a result of utilizing Numba, the code experienced a 30% increase in performance.

To achieve the best performance with Numba, several techniques were employed within the implementation. Firstly, the @njit decorator was used, which is Numba's "Just in Time" with nopython=True. This configuration ensures that the code is fully compiled and not executed using the Python interpreter, thus providing a faster runtime.

In addition, function signatures were specified for both arguments and return types of the functions. This practice allows Numba to optimize the code more effectively by having a clear understanding of the input and output types. We also set "Parallel=True", which enables Numba to automatically **parallelize** the code, taking advantage of multi-core processors and further improving execution times.

To boost performance even more, "nogil=True" was employed, which releases the Global Interpreter Lock (GIL) during the execution of the function. This allows for better **concurrency** and can significantly improve the overall performance of the code when running on multi-core systems.

Lastly, "fastmath=True" was utilized to enable less accurate but faster mathematical operations, trading off some numerical precision for increased speed.

4.5 Strategy 5: Reducing Function Calls

After achieving significant performance improvement using low-level machine code, just-in-time compilation, and parallelization employed by Numba's njit decorator, we focused on minimizing the number of function calls and unnecessary computations. To reduce function overhead, we consolidated the sigmoid, normalize, loss, predict, and gradients functions into the train and compare functions. Ultimately, we had a train function that still utilized Numba's parallelization capabilities to optimize the weights and biases of the model, and a compare function called the train function, made predictions with the weights and biases that the train returned and evaluated the model's prediction accuracy. The reduction in function overhead resulted in approximately a 7% improvement in CPU runtime.

In addition, we discovered that the original baseline model calculated and stored losses throughout the code but did not use them for anything. Profiling the code with a lineprofiler revealed that these loss calculations accounted for a significant amount of computation time. Therefore, it was not surprising that by removing all code related to these unused computations, we achieved an additional runtime performance improvement of 40%.

5 Results

Table 1: Results

Optimization technique	Runtime (seconds)	Performance Improvement
Baseline	12,614	N/A
Jax	328	38x
NumPy	131	96x
Cython	282	45x
Numba	95	133x
Function Calls	27.2	463x

Table 1 contains the results achieved through each of our optimization techniques. This includes runtime for each iteration of the code and the corresponding performance improvement compared to the baseline speed. The final version of our code ran 463 times faster than our baseline.

6 Discussion

The baseline code for our project was computationally inefficient. Employing the aforementioned techniques we were able to improve runtimes by several orders of magnitude. Our initial focus was on implementing Jax and NumPy, both of which resulted in massive improvements. However, we found that NumPy was faster than Jax and therefore decided to continue iterating off of our NumPy implementation.

As we sought to enhance the code's performance even further, we explored the use of packages that leverage the performance of C and low-level machine code. Specifically, we turned to Cython and Numba for this purpose. We noticed appreciable improvements with Numba, which is optimized to improve NumPy code, but not with Cython.

Finally, we were able to achieve an additional 70% reduction in runtime by reducing function calls and eliminating unnecessary functions that did not contribute to the model output.

Overall, our project demonstrated the significant impact that optimizing code can have on computational efficiency. By implementing various techniques and packages, we were able to achieve substantial improvements in our code's performance, allowing us to produce results in a fraction of the time it took with the baseline code.

Group Contribution

All members of our group contributed equally to the research and development of this project's presentations, coding, and results.

Acknowledgements

We want to thank Professor Yifei Sun, our section leader, Krishna Kartik Darsipudi, and the "DS-GA 1019 - Advanced Python for Data Science" graders for their expertise, dedication, and efforts in making the course an invaluable learning experience. Their insightful lectures, challenging assignments, and constructive feedback helped us better understand advanced Python concepts in Data Science.

References

- **Numba**

- Numba documentation: https://numba.pydata.org/
- "Numba: A High Performance Python Compiler" by Siu Kwan Lam, Avik Sengupta, and Stanley Seibert (2015)
- 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, pages 1–6.

- **Cython**
  - Cython documentation: https://cython.readthedocs.io/
  - "Cython: The Best of Both Worlds" by Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith (2011)

- **JAX**
  - JAX documentation: https://jax.readthedocs.io/
  - "JAX: composable transformations of Python+NumPy programs" by Matthew Johnson, Dougal Maclaurin, and Chris Leary (2020)
  - Jax Quickstart — JAX documentation. https://jax.readthedocs.io/en/latest/notebooks/quickstart.html. [Online; accessed 2023-05-07]

- **NumPy**
  - NumPy documentation: https://numpy.org/doc/
  - "NumPy: A Guide to NumPy" by Travis E. Oliphant (2006)