Automating Numba Optimization with Large Language Models: A Case Study on Mutual Information

Lučka Kozamernik
Teads
lucka.kozamernik@teads.com

Blaž Škrlj
Teads
blaz.skrlj@teads.com

Martin Jakomin
Teads
martin.jakomin@teads.com

Jasna Urbančič
Teads
jasna.urbancic@teads.com

Abstract

Contemporary large language models (LLMs) enable fast research cycles when developing or optimizing new algorithms. In this work, we investigate whether existing LLMs are sufficient to automatically, under constraints of unit tests, produce implementations of computational extensive algorithms such as the mutual information algorithm that would out-perform existing human-made baselines. We establish an evaluation pipeline where new proposed AI implementations are rigorously tested, evaluated, and benchmarked against existing baselines. We used synthetic numeric datasets of different sizes and results show 10-times speed-up using LLM optimized implementations compared to the naive Numba-based optimization while producing consistently correct mutual information scores.

Keywords

optimization, mutual information, LLM, Numba

1 Introduction

Mutual Information (MI) stands as a fundamental measure in information theory, quantifying the statistical dependency between two random variables. Its application is widespread and critical across numerous domains, including feature selection in machine learning, neuroscience for analyzing neural spike trains, and bioinformatics for understanding gene regulatory networks. The versatility of MI lies in its ability to capture arbitrary non-linear relationships, a significant advantage over linear correlation measures like Pearson's coefficient.

However, the computational cost of calculating mutual information, especially for large datasets with continuous variables, presents a substantial bottleneck. The standard approach involves discretizing the data into bins in order to estimate probability distributions, a process whose accuracy and performance are highly sensitive to the chosen binning strategy and the efficiency of the underlying implementation. For practitioners working within the Python ecosystem, libraries like NumPy and SciPy are standard tools, but their performance on MI calculations can be suboptimal for high-throughput screening or large-scale data exploration tasks.

To address this performance gap, Just-In-Time (JIT) compilers like Numba [4] have become indispensable. By translating Python and NumPy code into optimized machine code at runtime, Numba offers C-like performance without sacrificing the flexibility and ease of use of the Python language. A well-written, Numba-accelerated MI function can be orders of magnitude faster than its pure Python equivalent. Despite these gains, achieving optimal performance with Numba is not always straightforward. The efficiency of *Numba-jitted* code is highly dependent on the specific implementation patterns, data access methods, and loop structures used—subtleties that often require significant programmer expertise to navigate.

This paper introduces a novel approach to bridge this gap: the use of Large Language Models (LLMs) to automatically optimize Numba-based mutual information algorithms. We hypothesize that modern LLMs, trained on vast repositories of code, possess the capability to analyze suboptimal Numba implementations and refactor them into more efficient versions. Our work explores whether an LLM can identify and correct common performance anti-patterns in Numba code, such as improper loop organization or inefficient data type usage, to generate an MI implementation that surpasses a naively written Numba function. We present a framework for systematically prompting an LLM with a baseline algorithm and evaluating the performance of its generated optimizations, demonstrating the potential for AI-driven code acceleration in scientific computing.

2 Related work

This research builds upon three principal areas of study: the computation of mutual information, performance optimization with JIT compilers, and the application of Large Language Models to code intelligence tasks.

Mutual Information estimation is the long-standing challenge of accurately and efficiently estimating mutual information from given data. Defined as

$$I(X;Y) = \mathbb{E}_{p(X,Y)}\left[\log\left(\frac{p(X,Y)}{p(X)p(Y)}\right)\right],$$

it measures the pairwise relationships between random variables (continuous or discrete). The most common methods, as reviewed by Fraser and Swinney (1986) [2] and explored in detail by Kraskov, Stögbauer, and Grassberger (2004) [3], are based on data discretization (binning) or k-nearest neighbors (k-NN) estimators. While k-NN methods avoid the issue of bin selection, they typically incur higher computational complexity. Binned methods, though conceptually simpler, depend heavily on the binning strategy for accuracy and performance, a topic extensively studied by Steuer et al. (2002) [7]. Our work focuses on the binned approach, as it is highly amenable to loop-based array computations where Numba excels.

Information Society 2025, Ljubljana, Slovenia

The performance limitations of Python for numerical computation led to the development of various acceleration tools, specifically JIT compilers for Scientific Python. Numba, introduced by Lam, Pitrou, and Seibert in 2015 [4], has emerged as a leading solution by providing a decorator-based JIT compiler that integrates seamlessly with NumPy. It allows developers to accelerate functions containing Python and NumPy syntax, often achieving performance comparable to compiled languages. Research and community best practices have established a set of optimization techniques for Numba, such as managing memory layout, ensuring type stability, and structuring loops for parallelization and vectorization. This body of knowledge forms the basis against which we evaluate the LLM's optimization capabilities. Our work differs from traditional performance tuning by attempting to automate the discovery and application of these techniques solely through an AI model.

The emergence of robust Large Language Models (LLMs), such as OpenAI's Codex (the technology powering GitHub Copilot), has revolutionized software development. These models have demonstrated remarkable proficiency in code generation, translation, and explanation [1]. More recently, research has shifted towards their application in more nuanced tasks like code refactoring and optimization. For instance, studies have explored using LLMs to suggest improvements for energy efficiency or to refactor code for better readability. However, the specific domain of optimizing numerical algorithms within a JIT compilation framework like Numba remains relatively unexplored. While LLMs are known to generate functional code, their ability to produce code that is performant by adhering to the specific constraints and best practices of a framework like Numba is an open and compelling research question that this paper directly addresses.

3 Using LLMs to optimize existing code

To facilitate a systematic experimentation with LLM-optimized code, we set up a novel framework. The workflow consists of the following basic steps:

(1) Prompt the LLM with the task and context.
(2) Test the proposed optimizations against the unit tests.
(3) Benchmark the proposed implementation.

The framework is LLM-agnostic, meaning that any LLM can be used with it. We opt for the latest and most advanced versions of two popular LLMs, namely ChatGPT 5 and Gemini 2.5-Pro. Both are freely available and excel in complex tasks such as reasoning and coding.

To ensure a fair comparison between the models, both evaluated LLMs received the same prompt and the same context. The prompt was *"Can you make this code computationally more efficient, this meaning it computes faster?"*, while the context included the code that needed to be optimized. The initial code used in the input already contained some Numba instructions, however those were basic and naive. The tested code is a part of OutRank, an open-source tool for computing cardinality-aware feature ranking [6] and encompasses an implementation of the mutual information estimation.

The LLM output was first tested on unit tests to ensure that the optimizations still produced valid code and did not change any functionalities. By testing the proposed solution before using it for benchmarking, we are guaranteed that the code and its output are correct, consistent, and stable. Although not part of the framework at this stage, the output of the unit tests could serve as additional prompts to the LLM in order to improve itself and the code on the areas where the tests are failing.

```
@njit(
    'Tuple((int32[:], int32[:]))(int32[:])',
    cache=True,
    fastmath=True,
    error_model='numpy',
    boundscheck=True,
)
def numba_unique(a):
    """Identify unique elements in an array, fast"""

    container = np.zeros(np.max(a) + 1, dtype=np.int32)
    for val in a:
        container[val] += 1

    unique_values = np.nonzero(container)[0]
    unique_counts = container[unique_values]
    return unique_values.astype(np.int32),
          unique_counts.astype(np.int32)
```

```
@njit('Tuple((int32[:], int32[:]))(int32[:])', cache=True,
      fastmath=True)
def numba_unique(a):
    # assumes a >= 0
    maxv = 0
    for i in range(a.size):
        if a[i] > maxv:
            maxv = a[i]
    container = np.zeros(maxv + 1, dtype=np.int32)
    for i in range(a.size):
        container[a[i]] += 1
    unique_values = np.nonzero(container)[0].astype(np.int32)
    unique_counts = container[unique_values].astype(np.int32)
    return unique_values, unique_counts
```

```
@njit('Tuple((int32[:], int32[:]))(int32[:])', cache=True,
      fastmath=True)
def numba_unique(a):
    """
    Identify unique elements and their counts in a non-negative
         integer array.
    This version finds the max value in one pass to size the
         container.
    """
    # Assumes a >= 0
    maxv = 0
    if a.size > 0:
        for i in range(a.size):
            if a[i] > maxv:
                maxv = a[i]
    container = np.zeros(maxv + 1, dtype=np.int32)
    for i in range(a.size):
        container[a[i]] += 1
    unique_values = np.nonzero(container)[0].astype(np.int32)
    unique_counts = container[unique_values].astype(np.int32)
    return unique_values, unique_counts
```

Figure 1: Examples of proposed code changes. On the top is the initial function, followed by ChatGPTs solution and on the bottom is the code from Gemini 2.5-Pro.

Finally, in the last step of the framework, the resulted implementations were extensively benchmarked. The metric we were most interested in was the time needed to compute the mutual information for a given dataset; however, other metrics, such as memory utilization or GPU utilization, could also be used for a different use case. We further discuss our experimental setup in the results section.

3.1 Reviewing the LLM optimized code

The implementations of mutual information, produced by the selected LLMs, are remarkably similar — both in syntax and in the naming convention. However, there are subtle differences that

Implementation	Row count	Relative row count change
Baseline	182	0%
ChatGPT5	213	+17%
Gemini 2.5-Pro	262	+43%

Table 1: Row count for each of the implementations. Whitespace and comments are included in the row count.

set them apart, which we will address later. AI-aided implementations have in common that they completely omit error-handling model inherited from NumPy opting for the native Python instead. Moreover, they disregard bound checks for matrix operations before hand leaving the code to crash if it goes out of bounds. The latter is, according to the official documentation, advised for debug purposes only and should be turned off for production, as it slows down the code significantly. In line with the change in error handling, both implementations prefer elementary operations over the native NumPy functions. For example, to find the maximal value in an array, the LLM optimized code goes through all elements in the array by the index and compares to the current maximum instead of calling the built-in NumPy function. There is more evidence for this preference in the code. Such changes make the code appear much more C-like than native Python. Whenever there is the need for typecasting, the optimized code performs it at definition, instead of on return, which is commonly used in the naive implementation. The two types of proposed changes are illustrated with the code samples in Figure 1. Lastly, both LLMs introduced additional function that performs the prebuilt grouping to avoid unnecessary allocations and relocations in the loop. While the core techniques used for optimization are the same for both LLMs, Gemini 2.5-Pro used Numba's prange in one of the main computational loops, which adds parallelization, and makes the implementation faster on multicore machines. It also took the use of elementary operations much further than ChatGPT 5 — it replaced nearly all NumPy operations with native operations, increasing the row count twice as much as ChatGPT 5 did. The numbers are reported in Table 1 In addition, Gemini 2.5-Pro implemented its own in-code bounds checks based on elementary operations, while ChatCPT 5 did not. Contributing to the increase in the row count is also the amount of comments. The code review also revealed that Gemini 2.5-Pro was more consistent in code commenting and the comments were much more useful and informative for the developer.

4 Results

The setup for our benchmark was the following. We evaluated four different implementations of mutual information. For the two baselines, we used the standard and generic Sci-Kit learn mutual information and OutRank's basic MI-numba (that already contains some Numba instructions to optimize the performance). And as discussed before, two LLM optimized implementations were tested— MI-numba-chatgpt5 and MI-numba-gemini, which also support subsampling with a factor in range $(0, 1]$. For the evaluation, the subsampling factor ranges from 0.1 to 1, which means that no subsampling was applied.

To gauge how the performance scales with different parameters of the dataset, namely the number of examples (rows) and number of features (columns), we synthetically generated several datasets, containing raw numerical features with non-negative values (and varied the numbers of examples and features). The number of features ranged from 40 up to 200 in increments of 20,

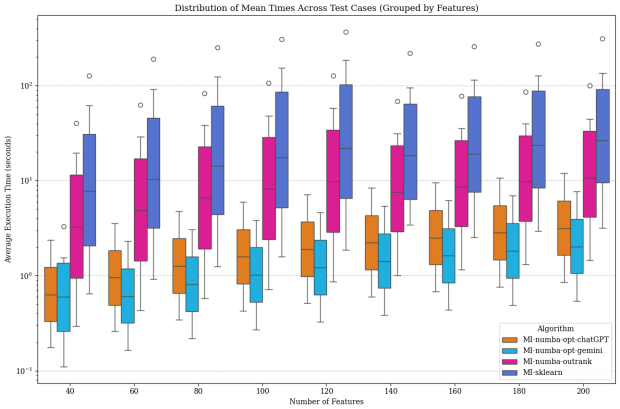

Figure 2: Distribution of Mean Times Across Test Cases (grouped by the number of features) showing the most efficient implementation is the one optimized with Gemini 2.5-Pro.

while the number of examples ranged from 200.000 to 20.000.000 in eight logarithmic steps. For each combination, represented by a tuple (algorithm, subsampling factor (where applicable), number of examples, number of features), we made five runs of the code. For each run, we recorded the time to compute mutual information using Python's time function.

The results are shown in Figure 2. The boxes represent 25th percentile in the bottom and 75th percentile on the top. For all test case, the LLM optimized implementations were significantly faster than the baselines (the naive Numba implementation of mutual information from OutRank and the generic Sci-Kit learn mutual information), with Gemini's implementation being the most efficient regardless of the number of features, number of samples or approximation factor. The LLMs sped up the computation of mutual information for approximately 10 times, while the difference between ChatGPT's and Gemini's version was much smaller. This implies that the biggest contribution to the speedup comes from the code changes that the two LLM optimized solutions have in common. Those are primarily the prebuilt grouping, which aims to reduce in-loop allocations, and the heavy use of elementary operations. Although parallelization in the Gemini 2.5-Pro's implementation still plays a role, its effect is less significant.

To verify that the computed mutual information is consistent with the generic implementations, namely the Sci-Kit learn implementation, we plotted the mutual information for each number of features. We show the results in Figure 3, where we can observe that the computed mutual information is almost identical for all implementations, regardless of the number of features and different optimizations applied. We conclude that the code optimized by LLMs is valid and correct.

5 Discussion

In our experiment, we used the latest and most advanced versions of two popular LLMs, namely ChatGPT 5 and Gemini 2.5-Pro, with Gemini 2.5-Pro being specifically targeted for coding. While we did put two different LLMs to the test, the goal was not so much to compare them, but to develop a framework that would serve well for evaluating LLM-based optimizations in scientific computing. As the new versions of LLMs and new LLMs are periodically appearing in the market, the framework can serve

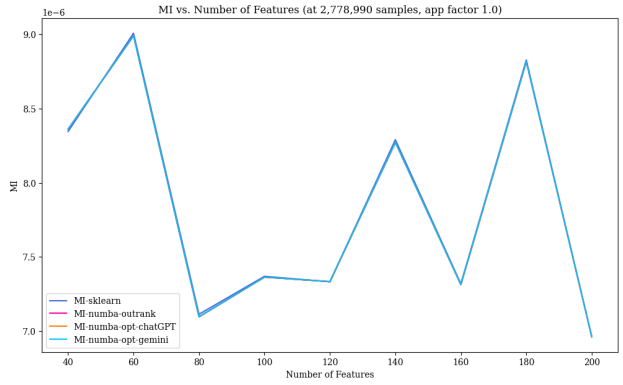

Figure 3: Computed Mutual Information for all tested implementations and for various numbers of feature.

to keep improving the existing code or, on the other hand, can be used to quantify the improvements (specifically for the coding subdomain) in the LLMs themselves as the new versions are released. Additionally, using the framework in development phase for scientific experiments can reduce the computational time and computational resources needed, leading to a lower cost for the experiments.

Focusing on the LLM aspect of the framework, the question remains what the result of the LLM-based optimization would be, had the context represented by the initial code not used Numba optimizations already. Few additional experiments could be done to explore that:

(1) Use Python code without Numba instructions and explicitly mention Numba in the prompt
(2) Use Python code without Numba instructions and do not mention Numba in the prompt
(3) Task the LLM to prepare the most computationally efficient implementation of mutual information in Python

6 Conclusions

In this work, we presented an initial framework for automatic code optimization via LLM achieving a very impressive 10-fold speedup compared to the naive baseline in the benchmarking experiments while maintaining correctness of the code. We were very impressed by the remarkable similarity of the code produced by two different and independent LLMs. The proposed solutions from both models focused on the same key areas: adding an auxiliary function that creates the pre-built groupings to reduce the in-loop allocations, and shifting the paradigm from native NumPy to C-like Python code relying on elementary operations.

While the optimization process is not yet fully automatic, our contribution outlines a possible direction for efficient use of LLMs in scientific computing. To reach the fully automatic stage when referring to Numba optimization, we propose the following steps are incorporated in the framework:

(1) Use unit test output in case of failure as the next prompt for the LLM to give it a chance to correct the code.
(2) Use the result of the benchmarking experiments as feedback to the LLM and iterate on the proposed optimization.

Both of these suggestions create feedback loops back to the LLMs, thus enabling an iterative process like the one proposed in Novikov et al. [5]. By comparing the outputs with the existing solutions, we have shown that the LLMs maintained the correctness when introducing optimizations.

References

[1] Mark Chen et al. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374. https://arxiv.org/abs/2107.03374 arXiv: 2107.03374.
[2] Andrew M Fraser and Harry L Swinney. 1986. Independent coordinates for strange attractors from mutual information. *Physical review A*, 33, 2, 1134.
[3] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. 2004. Estimating mutual information. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 69, 6, 066138.
[4] 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.
[5] Alexander Novikov et al. 2025. Alphaevolve: a coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*.
[6] Blaz Skrlj and Blaž Mramor. 2023. Outrank: speeding up automl-based model search for large sparse data sets with cardinality-aware feature ranking. In *Proceedings of the 17th ACM Conference on Recommender Systems*, 1078–1083.
[7] Ralf Steuer, Jürgen Kurths, Carsten O Daub, Janko Weise, and Joachim Selbig. 2002. The mutual information: detecting and evaluating dependencies between variables. *Bioinformatics*, 18, suppl_2, S231–S240.