Refactoring Code Through Library Design

Žiga Kovačič*
Cornell University

Justin T Chiu
Recursive

Celine Lee
Cornell University

Wenting Zhao
Cornell University

Kevin Ellis*
Cornell University

Abstract

Maintainable and general software allows developers to build robust applications efficiently, yet achieving these qualities often requires refactoring specialized solutions into reusable components. This challenge becomes particularly relevant as code agents are increasingly used to solve isolated one-off programming problems. We investigate code agents' capacity to refactor code in ways that support consolidation and reusability. We first investigate what makes a good refactoring, finding via simulation results and a human study that proxies of program size, such as Minimum Description Length, better predict preferable refactorings than extant software engineering metrics, such as Maintainability Index. We then present LIBRARIAN, a method that combines divide-and-conquer with sample-and-rerank to generate reusable libraries. We compare LIBRARIAN to state-of-the-art library generation methods, and study it on real-world Python programs.

1 Introduction

Writing code is mainly a matter of *re*writing code: debugging, refactoring, optimizing, and other activities within the software engineering lifecycle. But poor rewrites incur technical debt, with such debt costing up to $2 *trillion* annually [1]. This problem will likely worsen as language models become increasingly responsible for generating code, because they excel at solving isolated programming problems, but their context length demands a myopic view of the codebase. Code refactoring is a design problem whose concerns center on re-usability and maintainability. A classic example illustrates this design challenge: Human programmers often create overly-specialized, redundant solutions to similar problems that would benefit from consolidation into a shared library.

Here we focus on refactoring multiple files into a reusable software library, which raises two questions: (1) How should we quantify the quality of a library refactoring, and (2) How can we use LLMs to refactor specialized code into reusable libraries? To answer those questions, we develop a new method, and test it on both existing and new refactoring benchmarks. This goes beyond past work in *library learning* [2, 3, 4, 5, 6, 7, 8] which synthesizes subroutines across small programs in i.e. λ-calculus, because instead we address the more naturalistic problem of redesigning code written in contemporary high-level languages, such as Python, producing classes, methods, and helper functions in the style of a human-written library. Our method, LIBRARIAN (Figure 1), samples possible code rewrites and then reranks those samples based on criteria designed to capture a good refactoring. To generate potential rewrites, we cluster together pieces of code sharing common structure so that a succinct prompt can rewrite them jointly into their refactored form. To find strong criteria for ranking potential rewrites, we study a variety of objective functions across machine learning and software engineering, both on programming benchmarks and via a human study.

*Corresponding authors: ziga@stanford.edu, kellis@cornell.edu.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

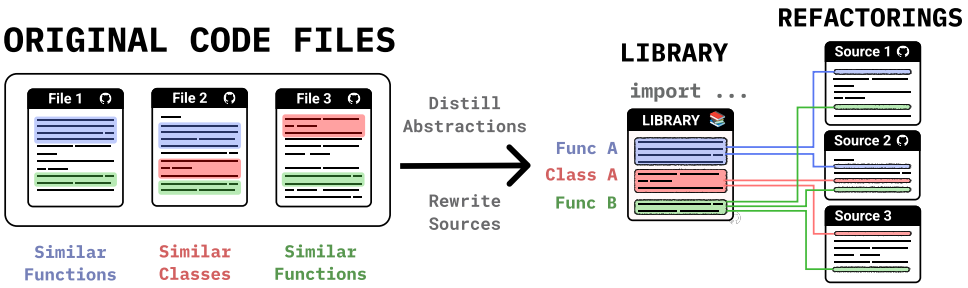

Figure 1: A refactoring task comprises a set of files. We refactor the files by designing a new library. Candidate refactorings are evaluated based on a refactoring objective, and are expected to maintain correctness of the original code sources (pass rate). We explore several refactoring objectives.

To evaluate our method and systematically assess the capability of current agents to generate libraries, we assemble a collection of refactoring benchmarks. This includes the programming puzzles often tackled by classic library learners, but most importantly also incorporates real-world languages and real-world programs, specifically source files from widely used HuggingFace models.

We contribute the following:

1. We run a user study with program synthesis benchmarks to evaluate different objective functions for library creation. We find that, surprisingly, neural language models offer a stronger signal of refactoring quality than classic objectives from the software engineering community.
2. A collection of refactoring benchmark problems spanning classic library learning challenges, competition programming problems, and real-world machine learning programs from the HuggingFace Transformers and Diffusers libraries.
3. A new algorithm, LIBRARIAN, which outperforms prior library learners on existing benchmarks and scales to real-world human-written Python code.
4. Refactorings of real-world Python sources widely used in the machine learning community. Our method successfully refactors files from the Huggingface Transformers and Diffusers libraries to be shorter and more reusable. To the best of our knowledge this is the first time library learning has been successfully applied to real-world software projects.

2 Related Work

Library Learning. Systems which perform library learning research discover shared abstractions across a large number of small programs, which they use to automatically define new subroutines. Systems such as DreamCoder [4], Trove [9], LiLo [7], and REGAL [3] automatically construct such libraries with the goal of making future program synthesis tasks easier to solve, once the learned library is in hand. Our work is closest to REGAL [3], which clusters related code and refactors using language models. However, existing library learning approaches have primarily been demonstrated in small-scale, constrained domains, limiting their applicability to typical software engineering tasks, such as consolidating multiple real-world source files into cohesive libraries. By framing library learning within the context of realistic software development, we expand the relevance of library learning to the everyday practice of software engineering.

Repo-level coding benchmarks. Our work considers multi-file refactoring, *not* repo-wide refactors, but recent work has considered repo-level software engineering tasks. Existing benchmarks include SWE-bench [10], which evaluates models on their ability to resolve real-life GitHub issues, and Commit-0 [11], which requires agents to fill in function definitions. Such benchmarks primarily evaluate functional correctness via unit tests, without assessing the quality or maintainability of the resulting codebase. RefactorBench [12] takes a step in this direction by benchmarking the ability to follow specific refactoring instructions. Our work differs by requiring models to perform a more open-ended task: Redesigning code to be more modular and compact by discovering reused abstractions, while retaining verifiability by re-using downstream unit tests. Additionally, libraries

must be created without any scaffolding limitations such as preexisting function definitions affording more design freedom than Commit-0.

Program optimization. While our goal is to optimize the quality of libraries, other works focus on improving execution speed through correctness-preserving transformations [13, 14, 15]. Both forms of program optimization, compression and speed, are more open-ended than optimizing only for correctness, as there does not exist a ground-truth answer. Prior work on program optimization benchmarks study code at the file level. We propose a benchmark that transforms programs at a larger scale, across multiple code files.

3 Problem Statement

Given N related files $\{\rho_n\}_{n=1}^N$, the goal is to create a library $\mathcal{L}$ that captures shared abstractions. The original files $\{\rho_n\}_{n=1}^N$, which define the refactoring problem, we call a **task**. The new library must support all original use cases in the task by extracting latent shared abstractions. This is accomplished by searching for refactorings that are both correct and 'natural'. Correctness is defined via unit tests, but naturalness is more challenging to quantify.

Shorter code is potentially simpler and less redundant. Compression is also well-known to foster generalization, suggesting compressive refactoring may also expose more opportunities for reuse. Prior works have defined their objective function by counting the number of tokens, lines-of-code, or syntax tree nodes in the proposed library and refactored code [6, 16, 5, 17]. But minimizing program size has obvious failure modes: code should also be understandable and extensible, which can be in tension with merely finding the shortest program.[2] Other work in program synthesis [18, 19, 4] instead optimizes *Minimum Description Length* (MDL), or negative log probability under a reference distribution. In the software engineering community, other metrics such as Cyclomatic Complexity (CC) and Maintainability Index (MI) have been defined for similar purposes: These are more complex objective functions that examine the syntax tree, call graph, and other statically-analyzable structures [20]. What objective should we use? We revisit this question in Section 6, where we compare candidate objectives under a human study and through simulation results.

For now assume a placeholder objective J measuring refactoring quality; we seek to optimize J while preserving correctness. Given a task comprising files $\{\rho_n\}_{n=1}^N$, we output both a new library $\mathcal{L}$, as well as rewritten refactorings of the original files, $\{\rho'_n\}_{n=1}^N$. We define tests passed $\tau(\rho_n)$ as the set of unit tests ρ_n passes, and consider both refactoring several files ($N > 1$) and also refactoring a single large file ($N = 1$). We optimize the following, which prefers passing at least the same tests as the original programs *and* optimizing the chosen objective J:

$$\ell_J\left(\mathcal{L}, \{\rho'_n\}\right) = \begin{cases} J(\mathcal{L}, \{\rho'_n\}) & \forall \rho_n, \tau(\rho_n) \subseteq \tau(\rho'_n) \\ \infty & \text{otherwise} \end{cases} \tag{1}$$

4 Benchmarks for testing library design

Past work in library learning has typically benchmarked against relatively small programs in i.e. λ-calculus, because the methods did not scale to large programs in rich modern languages such as Python. In our work, we seek to address this limitation in the literature by compiling a collection of benchmark problems that includes realistic refactoring of real-world code, but also includes classic library learning problems (to facilitate fair comparison) and small-scale programs in modern languages (to facilitate fast evaluation). Each benchmark task comprises a set of files, and systems refactor the files into a unified library alongside refactorings of the original files. There are two key desiderata for benchmark tasks: They should have related files sharing latent abstractions, and should also be verifiable, to measure how well refactored files preserve functional correctness. We source a variety of problems (Table 1).

[2] Perl Golf is a game where participants attempt to write the shortest Perl program accomplishing a given task. The resulting code is famously incomprehensible, even by the standards of Perl.

Table 1: Testing Statistics

Domain	Files	Tasks	Avg LoC	Avg Tests / file
Code Contests [21]	300	10	87	10
Transformers [22]	15	3	690	181
Diffusers [23]	17	5	745	75
Logo [2]	300	1	10	1
Date [24]	246	1	14	1

CodeContests. Competition problems are crafted with specific variations of algorithmic approaches in mind, resulting in both shared latent concepts and the required test cases. As a result, competition coding is both verifiable, and ready to refactor. We therefore take solutions, prompts, and tests from CODECONTESTS [25], a competition programming dataset.

Huggingface Transformers Library. We test refactoring across implementations of large language and vision–language models from the Huggingface `transformers` repository (`modelling_<name>.py` files, e.g., Qwen2, LLaMA, DeepSeek-V3). Unlike competition coding, these sources are production-scale and Huggingface requires that all changes pass an extensive suite of integration tests before merging into the main branch. A refactoring is only deemed correct if it passes the unmodified Transformers test suite, making this a high-stakes setting that requires correctness and compatibility. Our evaluation setup comprises 15 modeling files across 3 clusters, averaging 690 lines of code and 181 integration tests per file.

Huggingface Diffusers Library. We test refactoring across implementations of diffusion models from the Huggingface `diffusers` repository (`unet_<name>.py` and `scheduler_<name>.py` files, e.g., Stable Diffusion UNet, DDPMScheduler), yielding two distinct tasks. Like Transformers, Diffusers requires that all changes pass a comprehensive suite of integration tests. Our evaluation setup comprises 17 files across 5 clusters (three scheduler clusters, two UNet clusters), averaging 745 lines of code and 75 integration tests per file.

Logo & Date. The library learning literature already has existing benchmarks: Typically they seek to learn a single library from a task comprising many sources, and then test that library on holdout program synthesis tasks. Logo and Date were used in the recent related work REGAL [3], which we incorporate wholesale to understand how our new method compares to state-of-the-art library learning. The associated programming problems were created by humans, but their solutions were generated by gpt-3.5-turbo.

5 LIBRARIAN: Refactoring Code to Create Libraries

LIBRARIAN generates a new library from a set of files, while migrating the files to use that new library (Figure 1), following a sample-and-rerank framework: Prompting a backend LLM or agent to sample K candidate refactoring, and picking the one minimizing the loss ℓ. Naively,

$$\mathcal{L}, \{\rho'_n\} = \underset{\mathcal{L}, \{\rho'_n\} \in \textsc{Sample}_K(\{\rho_n\})}{\arg\min} \ell_J\left(\mathcal{L}, \{\rho'_n\}\right) \tag{2}$$

for objective J and sampling budget K. But this cannot work for large tasks with many programs, which would not fit into the context of many LLMs. Even long context models cannot process the entirety of e.g the Linux kernel. To address this, we wrap sample-and-rerank with a clustering algorithm that decomposes the task into manageable chunks, described next.

Clustering. Meaningful abstractions arise when programs share underlying functionality or structure. To surface these, we cluster the task's files into small groups that are likely to share reusable structure, and refactor each cluster separately from the rest. This decomposition shrinks the prompt size, and gives independent searches for the best per-cluster refactoring, which may be more tractable.

LIBRARIAN's clustering extends REGAL [3], which clusters programs by assuming each program is paired with a natural language description of the problem it solves, and clustering embeddings

of those descriptions. Since similar problems need not imply similar solution code, we instead prompt a model to summarize each file and cluster by these summaries. Specifically, we define $\textsc{Cluster}_S(\{\rho_n\})$ as performing agglomerative clustering [26] on the task's files $\{\rho_n\}$ to produce a set of set of files, each of which is a cluster of size S. We use text-embedding-ada-002 to embed descriptions of code sources for clustering.

Combining clustering with sample-and-rerank. The simplest approach is to refactor each cluster independently and take the union (concatenation) of each cluster's library:

$$\mathcal{L}^\star = \bigcup_{c \in \textsc{Cluster}_S(\{\rho_n\})} \mathcal{L}_c \qquad \{\rho'_n\} = \bigcup_{c \in \textsc{Cluster}_S(\{\rho_n\})} \left\{\rho'_{c,i}\right\}_{i=1}^{|c|}$$

$$\mathcal{L}_c, \left\{\rho'_{c,i}\right\}_{i=1}^{|c|} = \underset{\mathcal{L}, \left\{\rho'_{c,i}\right\}_{i=1}^{|c|} \in \textsc{Sample}_K(c)}{\arg\min} \ell_J\left(\mathcal{L}, \left\{\rho'_{c,i}\right\}_{i=1}^{|c|}\right), \quad \text{for each } c \in \textsc{Cluster}_S(\{\rho_n\})$$

The approach above ignores the fact that library abstractions discovered in one cluster might be useful in another cluster. A more sophisticated approach accumulates a library across clusters, and when refactoring a cluster, adds the accumulated library to the prompt. Appendix A describes this.

6 What Makes a Good Refactoring?

We compare different objectives J measuring the quality of a refactoring:

Tokens counts the total tokens in the refactored files *and* in the library. This minimizes program size, but not at the expense of a bloated library: Replacing every program with its own monolithic library function would not decrease tokens, because it measures library size as well. Concretely, $J_{\text{tokens}}(\mathcal{L}, \{\rho'_n\}) = \textsc{Tokens}(\mathcal{L}) + \sum_n \textsc{Tokens}(\rho'_n)$. Tokens most resembles existing library learning objectives, which minimize abstract syntax tree size (possibly with learned weights) [6, 18, 27, 4, 17].

Minimum Description Length (MDL) evaluates the negative log probability under a reference distribution, taking into account both the library and refactored sources. Unlike previous library learners, we use a language model as our reference distribution, defining $J_{\text{MDL}}(\mathcal{L}, \{\rho'_n\}) = -\log p_{\text{LM}}(\mathcal{L}) + \sum_n -\log p_{\text{LM}}(\rho'_n \mid \mathcal{L})$, where $p_{\text{LM}}(\rho'_n \mid \mathcal{L})$ is concatenating the library and the program into one prompt, but only counting the perplexity of the later program tokens. This has a Bayesian justification: The MDL library is the maximum aposteriori estimate of $\mathcal{L}$ given conditionally-independent programs. MDL encourages compression but differs from tokens: For example, a token-minimizing program deletes all comments, while MDL can encourage comments that increase the likelihood of subsequent code. We use Qwen-2.5-3B as our reference language model, as it is modern and open. To check sensitivity to the underlying LLM, we compute MDL values for 15 Code Contests clusters, each with 50 valid refactorings, using Qwen-2.5-3B, Llama-3.2-3B, DeepSeek-Coder, and Mistral, finding 84.2% agreement in the MDL-minimizing refactoring.

Cyclomatic Complexity (CC) is a longstanding software engineering metric measuring the number of linearly independent paths through the control flow graph [20]. Smaller programs often have lower cyclomatic complexity. Concretely, $J_{\text{CC}}(\mathcal{L}, \{\rho'_n\}) = \textsc{cc}(\mathcal{L}) + \sum_n \textsc{cc}(\rho'_n)$ and $\textsc{cc}(\rho) = E(\rho) - N(\rho) + 2P(\rho)$, where E, N, and P count control flow edges, nodes, and connected components.

Maintainability Index (MI) is a more modern software engineering metric combining lines-of-code, cyclomatic complexity, and Halstead volume. We define $J_{\text{MI}}(\mathcal{L}, \{\rho'_n\}) = -\text{MI}(\mathcal{L}) + \sum_n -\text{MI}(\rho'_n)$.

6.1 Comparing library reusability when optimizing different objectives

Are these different objectives equally effective at encouraging reusable libraries? To answer this question we run LIBRARIAN with different objectives (MDL/tokens/MI/CC), and then measure proxies of library reuse, specifically (1) total library usage within the refactored sources, and (2) average use of each new library function/class. Here we use tasks from HuggingFace Transformers and Diffusers, and also CodeContests, while varying the sample budget K.[3] We use ClaudeCode to sample refactoring of HuggingFace sources and o4-mini to refactor CodeContests sources.[4]

[3] We use Best@k estimator for the expected value of metrics. Appendix G describes the estimator and proves its correctness.

[4] ClaudeCode proved necessary for scaling to the larger sources present in the HuggingFace tasks. Surprisingly, on CodeContest, o4-mini outperformed coding agents such as Codex and ClaudeCode (Appendix K)

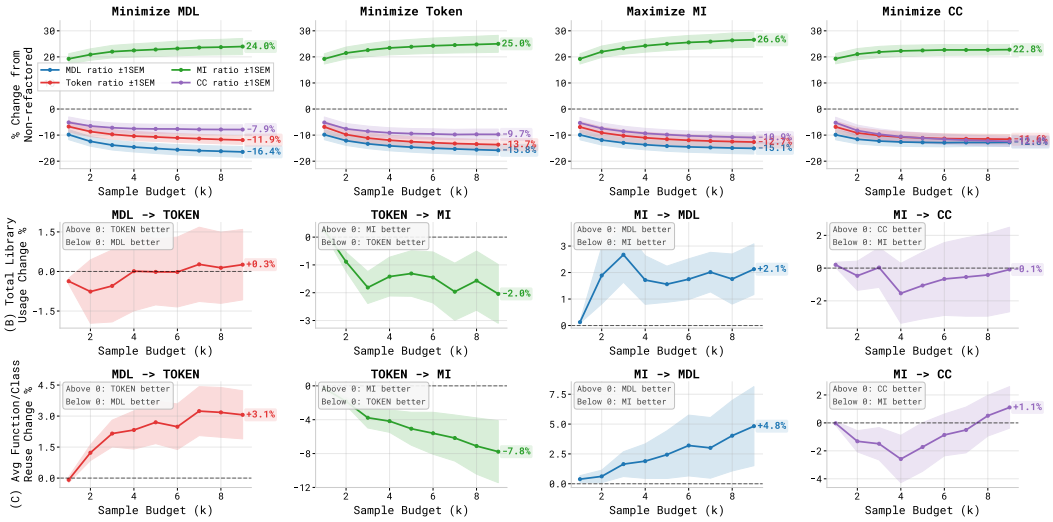

Figure 2: Metrics of library reuse for HuggingFace Transformers/Diffusers, measured on original refactored sources. Figure 3 repeats the same analysis for unseen holdout source files.

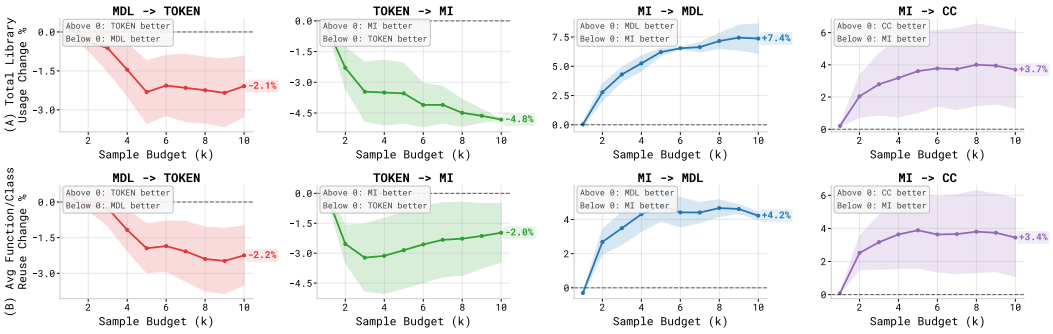

Figure 3: Metrics of library reuse for HuggingFace Transformers/Diffusers on unseen holdout sources.

Optimizing MDL and tokens yields libraries that are more heavily reused on the original source files, compared to optimizing MI and CC (Figures 2 and 5). For real-world sources, such as HuggingFace files, optimizing compression (MDL/tokens) also improves software engineering metrics (MI/CC), and vice versa. Surprisingly, for smaller-scale programs (CodeContests), the objectives dissociate: Optimizing MDL *does not* improve MI, and vice-versa, suggesting that compression remains effective across program scales, while classic software engineering metrics lose signal for small programs.

However, the results so far, which do not evaluate the library on holdout source files, show little difference between MDL and tokens. Absent holdout data, optimizing MDL (as opposed to tokens) improves reuse on small programs (CodeContests) but degrades reuse on large programs (HuggingFace), but all with small effect size. To tease apart MDL and tokens, we next examine how well optimizing these objectives generalize to unseen source files.

Do the libraries also generalize to unseen source files? We use each learned library to refactor HuggingFace transformers and diffusers sources that were *not* used to learn the library. This yields a stronger separation between MDL and tokens: MDL-optimized libraries generalize better to unseen sources compared to token-optimized libraries (which generalize better than MI/CC).

These quantitative results indicate that compression encourages reusable libraries better than software engineering metrics, and that MDL is somewhat more effective than tokens, particularly for libraries that prove useful on new unseen source files. Qualitatively, how is this reuse accomplished? Figure 4

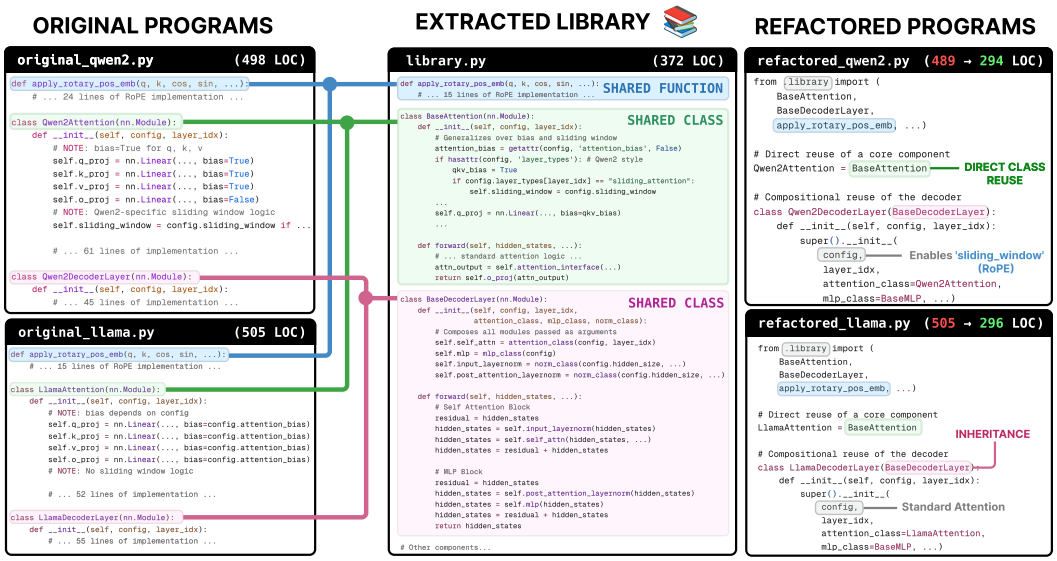

Figure 4: Representative result for refactoring HuggingFace Transformers using LIBRARIAN

illustrates a representative refactoring of transformers. LIBRARIAN distilled repeated abstractions such as MLPs, Attention, Decoder classes, RoPE helper functions, etc., lowering MDL to 67.2% of its original value while still passing all integration tests. The top-3 refactorings based on MDL have an average of 18 ± 4.4 functions/classes in the library, each called 4.59 ± 0.39 times in the refactored models. For Diffusers, scheduler clusters yielded top-3 MDL refactorings with an average of 12.3 ± 1.6 functions and 3.0 ± 0.4 calls per function, while UNet refactorings produced richer abstractions with an average of 17.0 ± 5.6 functions/classes and 3.43 ± 0.67 calls each.

MDL comparison to human HuggingFace refactoring. To the best of our knowledge, this is the first time any library learning algorithm has been successfully applied to real-world software projects. HuggingFace Transformers library is used by nearly 400k GitHub projects, and its developers are concurrently refactoring the Transformers library. On the same source files, their refactoring lowers MDL by 33.5%, suggesting even if human developers are not explicitly optimizing MDL, their refactoring improves it. We, the authors, refactored the sources ourselves, lowering MDL by 38%. LIBRARIAN lowers MDL by 32.8%, roughly in line with human coders.

Do we need objective functions in the era of coding agents? We compare against simply asking Claude Code to "reduce code duplication by extracting shared code into a library" in a single pass across Transformer source files. This baseline extracts 7 library definitions used an average of 5.86 times each (41 total uses). In contrast, running LIBRARIAN—with clustering, sampling, and MDL reranking—more than doubles the number of library functions from 7 to 15. Our bigger library is also more reusable: averaging 6.67 uses per function, total around 100 uses in the refactored files. There is significant margin between LIBRARIAN and even the latest coding agents.

Objectives & inference-time scaling. Studying these objectives while varying sampling budget K helps understand their inference-time scaling behavior. While the underlying metric itself improves (with diminishing returns), this was not to be taken for granted: The backend language model must produce sufficient diversity to steadily improve these metrics. Prior state-of-the-art, such as [3], takes a single sample, but our results suggest benefit from further test-time search. Significantly, we observe that real-world code benefits from increased samples steady improvement with increased samples. But our proxies of library utility plateau much earlier, around $K = 5$ to $K = 10$ samples, suggesting large K is unnecessary in practice: Library building benefits from test-time compute, but does not demand exorbitant compute.

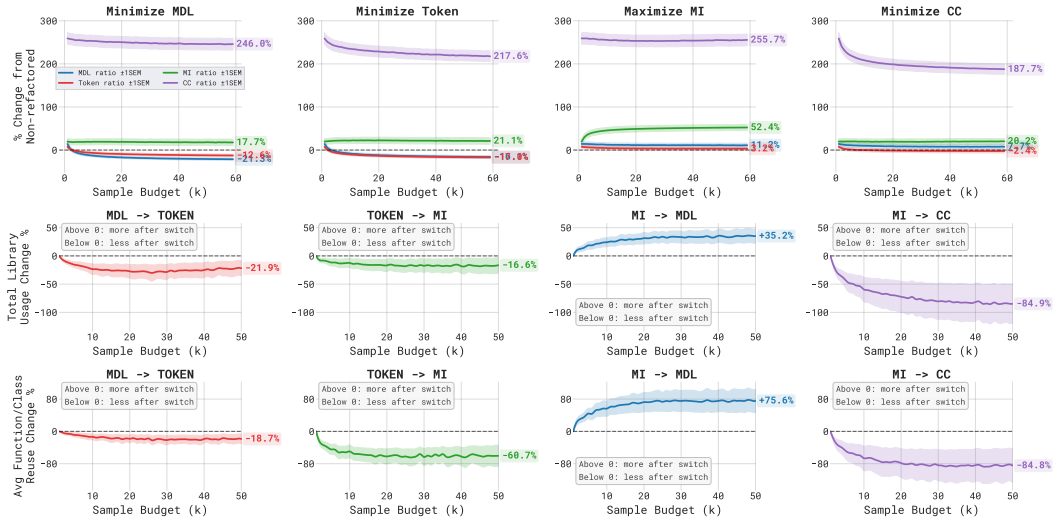

Figure 5: Metrics of library reuse for CodeContests clusters.

6.2 Human evaluation of refactoring metrics

We perform a human study to corroborate these findings using the same CodeContests[5] data (Appendix F). The study compares tokens, MDL, and MI by presenting participants with the original sources and their refactorings under pairs of metrics, and eliciting pairwise preferences.

We elicited 129 judgments from 14 human participants. Humans prefer MDL-minimizing libraries (Figure 6): Specifically, the data indicate MDL>MI, MDL≥Tokens, and Tokens≥MI. Although more participants would sharpen some of these inequalities (narrow our error bars), already we see a general preference for compression-based metrics (MDL and Tokens) with only MDL crossing the threshold of statistical significance. Indeed, minimizing MDL is not quite the same as minimizing tokens, as Appendix Figure 8 illustrates: We believe basically every human coder would prefer the MDL-minimizing program in this example.

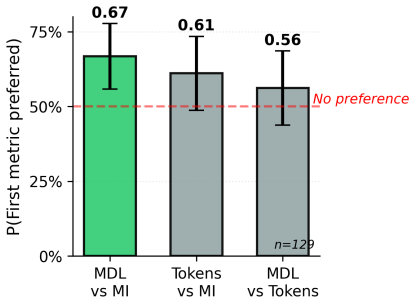

Figure 6: Human evaluation of different refactoring objectives. Judges compare pairs of refactorings that both pass all test cases. MDL aligns best with human preferences.

We therefore adopt J_{MDL} as the primary objective in the remainder of this paper: In addition to support from this human study, Bayesian arguments support MDL; corner cases in the style of 'Perl golf' provide existence proofs of the liability of merely minimizing tokens; and reasonable proxies for library reuse favor MDL both for small-scale CodeContest programs and for real-world HuggingFace sources (Section 6.1).

7 Using libraries for downstream program synthesis

The classic library learning setting is de novo program synthesis: First learning a library, then synthesizing new programs from scratch (but given the library), with the assumption that a good library helps solve unseen programs synthesis problems. The analyses in Section 6 measure whether the library helps refactor new programs, not whether it helps discover them in the first place. We now revisit representative program synthesis problems that have recently animated the library learning community, and evaluate LIBRARIAN on those tasks.

[5]Doing a human study on HuggingFace data would mean human participants would have to examine very long sources, and participants may already be aware of those sources, making it an unsuitable target for the human study.

We compare against REGAL [3], a state-of-the-art library learning system. REGAL targets program synthesis, rather than the basic refactoring problems explored earlier in this paper. Representative REGAL datasets include Logo and Date, which involve graphics program synthesis and string manipulation, respectively. Logo is particularly challenging: It is a harder version of the graphics synthesis problems introduced by the library learner DreamCoder [4], made harder to serve as a durable challenge to future systems in that family.

Table 3 shows LIBRARIAN significantly improves upon REGAL, especially on the harder Logo dataset, where LIBRARIAN solves about 40% more program synthesis problems, using a library constructed using just $K = 5$ samples. This demonstrates that there was significant headroom in these datasets which could be addressed solely through improved library learning.

8 Consolidating one-off solutions into reusable code

We study competition programming because it is a clear example of a setting where human coders produce quick one-off solutions, but where many of those solutions share common structure, making them amenable to library learning. This differs from the previous program synthesis evaluations because the library is not asked to write new programs from scratch, but instead to consolidate existing solutions into a new library. We test on CodeContests with a cluster size of $S = 3$ and a sample budget of $K = 8$ draws from o4-mini.[6] Table 2 shows that the resulting refactors and libraries approximately halve the MDL, which incidentally reduces tokens as well. Pass rate modestly improves as an incidental consequence of sampling and filtering with test cases. Libraries average 10 functions, each heavily reused: Averaging 5 uses per function within tasks comprising only 10 programs. But almost 40% of library functions are only used once. Why is that?

A signature of the MDL objective is a preference for whatever a language model assigns high apriori probability to. Although a single-use function does not reduce line count or tokens—the function could simply be inlined—it improves MDL if it yields a more natural decomposition of the target programs. Indeed, human-written libraries sometimes include functions that are seldom used, provided they serve as a conceptually modular abstraction. We therefore should see single-use functions as a feature, not a bug, provided such refactorings prove valuable on unseen files—and indeed, Section 6.1 showed that MDL (but not tokens) generalized better to new unseen files. See Appendix L for an example refactoring candidate on CodeContests.

Table 2: LIBRARIAN on Code Contests

Metric	Value
Pass Rate Improvement	6.33% ±1.41
MDL Ratio	0.53 ±0.03
Token Ratio	0.66 ±0.04
Library Functions	10.30 ±1.41
Avg Calls per Function	5.17 ±1.08
% Single Use Functions	38.03% ±4.88

Table 3: Solving holdout test program synthesis tasks using learned libraries.

Dataset	Model	Pass Rate
Logo	REGAL (gpt-3.5-turbo)	49.3% ±1.1
	LIBRARIAN (3.5-turbo)	69.9% ±0.9
Date	REGAL (gpt-3.5-turbo)	90.2% ±0.5
	LIBRARIAN (3.5-turbo)	94.7% ±0.7

9 Conclusion

We introduce LIBRARIAN for refactoring files into reusable libraries, and evaluate it on program synthesis problems from the library learning literature, competition programming solutions, and real-world software widely used within the machine learning community. We view our most important contributions as being twofold: First, an analysis of different objective functions for library learning, finding that MDL under a language model encourages more reusable libraries compared to token counts (analogous to past library learners' AST-size objectives), and compared to existing software engineering metrics (CC and MI). Second, we provide a working demonstration of real-world library learning on widely used software. These contributions interact: We could only scale to real-world software by using modern coding agents, but we find that the objective function we identify (plus clustering) still significantly improves upon simply asking ClaudeCode to make a library. Together we hope that these results move library learning into the real world, which we show is valuable

[6] Code agents such as Codex, Claude Code, and others underperformed o4-mini (Appendix K)

for solving new programming problems, consolidating isolated one-off solutions, and rewriting widely-used source code into more abstracted and reusable forms.

Limitations. We do not refactor whole repositories, leaving that to future work. Refactoring at even this scale proved expensive: Each HuggingFace refactoring took approximately 30 minutes to generate and test. But this is a one-off cost, and in our view, the refactored Transformers and Diffusers sources are much cleaner, and the new library is transparently reusable (Figure 4). Our method also relies on existing test cases to validate refactorings, but in general, refactoring code might require adding new test cases, which we defer to future work. Our clustering algorithm groups related files, but cannot split files into pieces to make more targeted clusters, which could prove important for long files that mix-and-match different concerns. Finally, compression might not always predict reusability to new unseen programs. Compression predicted reusability on Transformers and Diffusers, but this could be studied across many codebases.

Acknowledgements. K.E. was supported by NSF Grant #2310350.

References

[1] Shane Tews. Inside tech's $2 trillion technical debt | american enterprise institute - aei.

[2] Catherine Wong, Kevin Ellis, Joshua B. Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, 2021.

[3] Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. Regal: refactoring programs to discover generalizable abstractions. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[4] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *PLDI*, 2021.

[5] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL), January 2023.

[6] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, 2013.

[7] Gabriel Grand, Li Siang Wong, Matthew Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code. *ArXiv*, abs/2310.19791, 2023.

[8] Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, volume 10, pages 639–646, 2010.

[9] Zhiruo Wang, Daniel Fried, and Graham Neubig. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks, 2024.

[10] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[11] Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations*, 2025.

[12] Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. Refactorbench: Evaluating stateful reasoning in language agents through code. In *The Thirteenth International Conference on Learning Representations*, 2025.

[13] Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? *arXiv preprint arXiv:2407.14044*, 2024.

[14] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025.

[15] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 305–316, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.

[17] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning better abstractions with e-graphs and anti-unification. *POPL*, 2023.

[18] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, 2010.

[19] Ray J Solomonoff. A formal theory of inductive inference. *Information and control*, 7(1):1–22, 1964.

[20] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[21] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[22] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[23] Patrick von Platen, Suraj Patil, Anton Lozhkov, Pedro Cuenca, Nathan Lambert, Kashif Rasul, Mishig Davaadorj, Dhruv Nair, Sayak Paul, William Berman, Yiyi Xu, Steven Liu, and Thomas Wolf. Diffusers: State-of-the-art diffusion models. `https://github.com/huggingface/diffusers`, 2022.

[24] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adri Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on machine learning research*, 2023.

[25] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[26] Joe H Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[27] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *POPL*, 2023.

[28] OpenAI. Introducing o3 and o4 mini. `https://openai.com/index/introducing-o3-and-o4-mini/`, 2024. Accessed: 2025-05-13.

[29] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[30] Anthropic. Claude code: An agentic coding tool that lives in your terminal. `https://github.com/anthropics/claude-code`, 2025. An agentic coding tool that lives in your terminal, understands your codebase, and helps you code faster by executing routine tasks, explaining complex code, and handling git workflows - all through natural language commands.

A Incremental Version of LIBRARIAN

We support sharing library components across clusters through the below incremental version of our algorithm. It processes clusters sequentially, rather than in parallel, and its output can depend on the ordering imposed upon the clusters.

$$\mathcal{L}_0 = \varnothing$$

$$\{\rho'_n\} = \bigcup_{c \in \text{CLUSTER}(\{\rho_n\})} \{\rho'_{c,i}\}_{i=1}^{|c|}$$

$$(\Delta\mathcal{L}_c, \{\rho'_{c,i}\}_{i=1}^{|c|}) = \arg\min_{(\Delta\mathcal{L}, \{\rho'_i\}) \in \text{SAMPLE}_k(c;\, \mathcal{L}_{t-1})} \ell(\mathcal{L}_{t-1} \cup \Delta\mathcal{L}, \{\rho'_i\}_{i=1}^{|c|}) \quad \forall c \in \text{CLUSTER}(\{\rho_n\})$$

$$\mathcal{L}_t = \mathcal{L}_{t-1} \cup \Delta\mathcal{L}_t, \qquad \mathcal{L}^\star = \mathcal{L}_{|\text{CLUSTER}(\{\rho_n\})|}$$

B Algorithm

Algorithm 1 Refactoring Specialized Programs into a Joint Library

Require: Set of independent, specialized programs $P_{initial} = \{\rho_1, \rho_2, \ldots, \rho_n\}$
Require: Sample Budget K
Ensure: Joint library $\mathcal{L}_{final}$ and set of refactored programs P_{final}

$\mathcal{C} \leftarrow$ Cluster$(P_{initial})$
$\mathcal{L}_{final} \leftarrow \emptyset$, $P_{final} \leftarrow \emptyset$
for all cluster $c \in \mathcal{C}$ **do** ▷ Each cluster independently
  $T_C \leftarrow$ GroupIntoTuples(c) ▷ Get tuples for each cluster
  for all tuples $\tau \in T_C$ **do**
    $\{f_{retrieved}\} \leftarrow$ RetrieveRelevantFromLibrary$(\mathcal{L}, \tau)$
    $S \leftarrow \emptyset$
    for $i = 1$ to K **do** ▷ Sample k times
      $(\{f_{new,i}\}, \{\rho'_i\}) \leftarrow \textsc{Sample}\,(f_{retrieved}, \tau)$
      $S \leftarrow S \cup \{(\{f_{new,i}\}, \{\rho'_i\})\}$
    end for
    $(f_{best}, \{\rho'_{best}\}) \leftarrow$ RerankAndSelectBest$(S, \ell_J(\cdot))$ ▷ Rerank using objective
    $\mathcal{L}_{final} \cup \{f_{best}\}$
    $P_{final} \cup \{\rho'_{best}\}$
  end for
end for
return $\mathcal{L}_{final}, P_{final}$

C Experimental Setup

Grouping Programs into Collections To facilitate parallel application of LIBRARIAN and manage the dataset scale, we assume that semantically distant files will have minimal overlap in their optimal

library functions. Therefore, our overall approach partitions the dataset into disjoint collections through clustering.

For **CodeContests**, these collections are constructed from an initial corpus of ∼9k problems with Python solutions: We first filter these files, removing those whose selected canonical solution is under 10 lines (minimal refactoring potential). For the remaining 4596 solutions we use a language model to generate textual descriptions of canonical solutions—emphasizing reusable components—which are embedded using OpenAI's `text-embedding-ada-002`.

Agglomerative Clustering [26] is subsequently applied to these embeddings to partition the files into a predefined number of initial clusters, in our case 120. To create uniformly sized experimental units, we subsample each such cluster to form collections of 30 files. This collection size was empirically chosen because it balanced between the runtime of LIBRARIAN without limiting compression. We select 10 collections that we then use to evaluate our methods.

For Transformers, since the number of models is on the lower end, we manually chose a set of popular LLM / VLM models and passed them to the agent in collections of 5 code sources.

REGAL Baselines. To evaluate the ability of our libraries to support reuse on new problems, we turn to the program synthesis tasks used in REGAL, where learned libraries are added to help the program synthesizer. We evaluate on the two domains published by the authors, Logo and Date. Because our clustering is inspired by REGAL but adds additional complexity, for fair comparison, we keep their setup the same and only augment the training using sample + MDL rerank procedure described in Section 5.

Code Contests. To evaluate LIBRARIAN on refactoring Code Contests we select 6 collections of 30 files (problems). In each collection we group the problems into tuples of size 3. We set the sample budget to be $K = 8$, since our ablations show that with larger K we discover better libraries **??**. We use the MDL objective for rankings. The model used for sampling is OpenAI's o4-mini [28]. To obtain MDL scores we use Qwen 2.5 7B Instruct [29] as a balance between quality, speed, and cost.

Code Agents on Transformers and Diffusers Repositories. To fairly evaluate performance on the task by state-of-the-art systems, we use coding agents that advertise long-context ability to reason about, write, and refactor code repositories. Specifically, we use Claude Code (Cl) [30] which uses the Opus 4.1.

We test whether code agents can refactor collections of code sources autonomously, without human intervention. Refactoring repositories with code agents involves planning and iterative (re-)implementation and testing. Code agents are prompted to perform each of these steps, with feedback from the unit tests. Agents must run and repair unit tests autonomously. We run coding agents multiple times per task, logging their progress in checklists stored in text files.

D Human study

Participants were compensated $40 for their participation which took about 45min for each participant. Our IRB reviewed our protocol and approved it. The instruction provided to human evaluators is as follows:

```
## 1. Materials Provided

You will be given a set of files for each example case:

* **‘original_programs.py‘**: This file contains a set of 3 distinct Python programs, each presented with its corresponding problem description/query. This represents the "before" state.
* **‘v1.py‘**: This file presents the first refactoring approach. It includes:
    * The 3 refactored versions of the original programs.
    * A "library" section (e.g., ‘codebank.py‘ or inline) containing helper functions. These helper functions might be retrieved from an existing common library or newly created during this refactoring.
    * Either the retrieved or the new helper function sections may be _empty_, in case no programs existed in the codebank at the time or if no need helper functions were created by the LLM.
* **‘v2.py‘**: This file presents the second, alternative refactoring approach. Similar to ‘refactoring_v1.py‘, it includes:
    * The 3 refactored versions of the original programs (using a different strategy than v1).
    * A "library" section with its own set of helper functions.


**NOTE**:  both refactorings had accuracy at least as good as the original programs.

## 2. Your Task

Your primary task is to:

1.  **Review** the ‘original_programs.py‘ to understand the initial code and the problems being solved.
2.  **Analyze** both ‘refactoring_v1.py‘ and ‘refactoring_v2.py‘. Pay close attention to how the original programs have been restructured and what functionalities have been extracted into their respective libraries.
3.  **Decide which refactoring (Version 1 or Version 2) you believe is "better,"** based on the evaluation criteria provided below (or your own criteria!).

## 3. Evaluation Criteria: What to Consider for Your Choice

When comparing ‘refactoring_v1.py‘ and ‘refactoring_v2.py‘, please *consider* the following aspects to inform your choice. The "better" refactoring should ideally excel in these areas:

> Most importantly, make sure that the extracted functions are **actually reusable and not too specific.** If the main programs are short, the refactoring is not immediately "better"! Try to think whether the extracted functions could actually be used in a different program down the line.

* **Reusability of Helper Functions :**
    * **Generality:** Are the new helper functions general-purpose and potentially useful for *other, different* programs and problems beyond the three presented?
    * **Reuse:** How much were existing helper functions reused?
    * **Specificity:** Are the functions too specialized to the current set of problems, limiting their broader applicability? _Avoid functions that are essentially just the original program broken out into a "helper."_
    * Composability
* **Maintainability:**
    * Readability & Understandability
    * Ease of Modification
    * Separation of Concerns

## 4. What NOT to Focus On:

* **Comments:** Please disregard the presence or absence of comments in the code for this evaluation. These are superficially generated by LLMs in some occasions and could be added manually after with a single pass.
* **Minor Stylistic Differences:** Do not focus on trivial differences in variable naming or formatting, unless they significantly impact readability or understanding.

## 5. How to Provide Your Feedback

For each example case, please provide:

1.  **Your Preferred Version:** (e.g., "Version 1" or "Version 2")
```

Listing 1: Human Evaluation Instruction

E Further Results on Refactoring Metrics

Here we include the full graph of asymptotic behavior of all four scoring metrics (MDL, tokens, MI and CC), reporting the raw library metrics and the metric ratios. We can see that refactored programs have higher CC than the baseline and that optimizing CC as an objective does not decrese the other

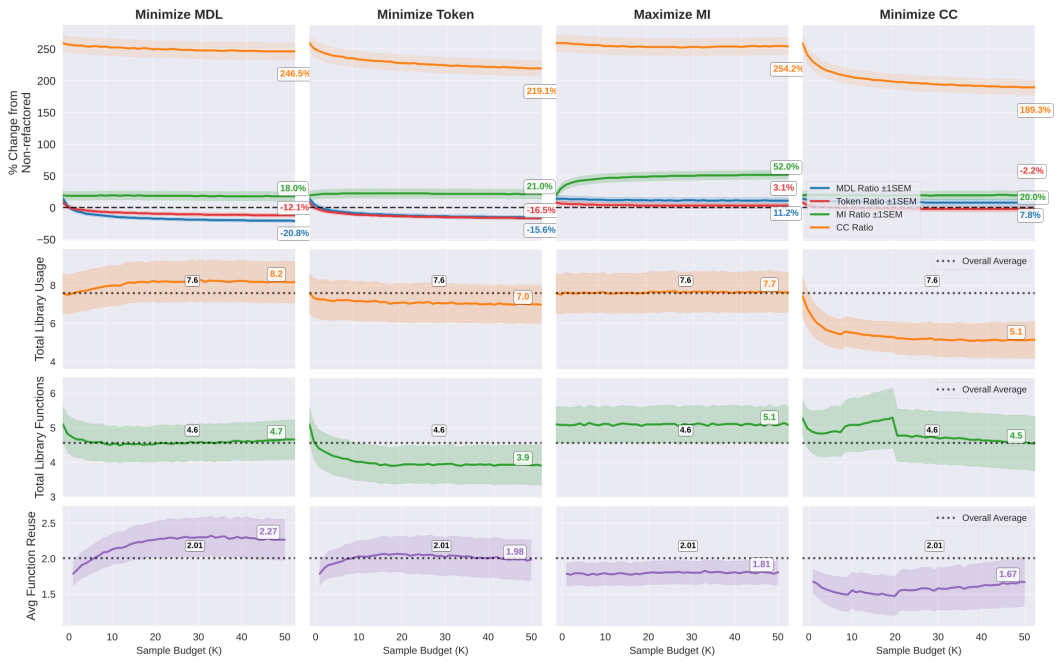

Figure 7: Asymptotic behavior of metrics for scoring libraries and refactorings (columns) varying refactoring budget (horizontal axes).

metrics. MDL performs best on total raw library usage as well as on average times a library function is used in the refactorings. MI ends up optimizing the number of library functions the best, but their reusability is below the average for the refactorings. Optimizing for tokens produces smaller libraries with less usage per function compared to optimizing MDL.

F Human Study Details

We ran a user study with 14 participants where each participant had to judge 10 refactoring pairs. Each pair was comparing two out of three metrics, MDL, tokens, and MI. We showed the participants the original programs, as well as both of the refactoring versions (including the programs and the learned libraries). The participants were able to choose either version or say that the two refactorings are almost the same.

To quantify pairwise preferences between refactoring metrics, we employed the Bradley-Terry model, a standard framework for analyzing paired comparison data. We fit the model using maximum likelihood estimation with maintainability index (MI) as the reference category ($\pi_M I = 1$). To address potential noise in human judgments, we applied consensus-based filtering with a 75% threshold, retaining all responses for comparisons with low consensus (indicating genuine ambiguity) while excluding minority responses on high-consensus comparisons where the majority preference likely indicates the correct judgment. This conservative approach preserved 92.4% of responses while strengthening the statistical evidence for metric preferences, with MDL significantly outperforming MI ($p = 0.67$, 95% Confidence Interval: $[0.56, 0.78]$). Even without the filtering MDL preference over MI was statistically significant.

G Best@k Compression is a U-Statistic

We wish to estimate the expected compression ratio achieved by our *sample + rerank* method, which samples k candidate refactorings, discards any that do not pass the tests, and selects the one with the lowest score (total log-prob).

Background on U-Statistics. Let $Z_1, \ldots, Z_n \overset{\text{i.i.d.}}{\sim} F$. For a symmetric function $h : \mathcal{Z}^k \to \mathbb{R}$, the *U-statistic of order* k is defined as

$$U_n \;=\; \binom{n}{k}^{-1} \sum_{1 \le i_1 < \cdots < i_k \le n} h(Z_{i_1}, \ldots, Z_{i_k}). \tag{3}$$

By construction,

$$\mathbb{E}[U_n] \;=\; \mathbb{E}[h(Z_1, \ldots, Z_k)], \tag{4}$$

so U_n is an unbiased estimator of the population quantity $\theta = \mathbb{E}[h(Z_1, \ldots, Z_k)]$.

Application to Best@k Compression. Let each valid refactoring be a pair $Z = (S, C)$, where S is the score and C is the compression ratio. Define the symmetric function

$$h_k(z_1, \ldots, z_k) \;=\; C_{j^*}, \qquad j^* = \arg\min_{1 \le j \le k} S_j, \tag{5}$$

the compression ratio of the lowest-score refactoring among k draws. The population target is then

$$\theta_k = \mathbb{E}[h_k(Z_1, \ldots, Z_k)]. \tag{6}$$

Given n valid samples, our estimator is

$$\widehat{\theta}_k \;=\; \binom{n}{k}^{-1} \sum_{1 \le i_1 < \cdots < i_k \le n} h_k(Z_{i_1}, \ldots, Z_{i_k}). \tag{7}$$

Proposition. $\widehat{\theta}_k$ is a U-statistic of order k with function h_k, and hence an unbiased estimator of θ_k.

Proof. (1) *Symmetry of the function* h_k. h_k selects the compression associated with the lowest score among its k arguments. Permuting the inputs does not affect this outcome (ties can be resolved with a fixed, permutation-invariant rule). Thus h_k is symmetric.

(2) *U-statistic form.* By definition, a U-statistic of order k with kernel h_k is

$$U_n = \binom{n}{k}^{-1} \sum_{1 \le i_1 < \cdots < i_k \le n} h_k(Z_{i_1}, \ldots, Z_{i_k}),$$

which matches $\widehat{\theta}_k$ exactly.

Therefore, $\widehat{\theta}_k$ is a U-statistic of order k. By the unbiasedness property of U-statistics,

$$\mathbb{E}[\widehat{\theta}_k] = \theta_k.$$

□

Thus, our reported *best@k compression curves* provide unbiased estimates of the expected performance of the *sample + rerank* method.

H Obfuscation Example

(a) Original library definitions

```
from ..shared_library import (
    rotate_half,
    apply_rotary_pos_emb,
    repeat_kv,
    eager_attention_forward,
    RMSNorm,
    BaseMLP,
    BaseRotaryEmbedding,
    BaseAttention,
    BaseDecoderLayer,
)

class LlamaRMSNorm(RMSNorm):
    ...

class LlamaRotaryEmbedding(BaseRotaryEmbedding):
    ...

class LlamaMLP(BaseMLP):
    def __init__(self, config):
        super().__init__(
            config,
            mlp_bias=config.mlp_bias
        )
...
```

(b) Obfuscated version

```
from ..shared_library import rotate_half,apply_rotary_pos_emb,repeat_kv,eager_attention_forward,RMSNorm as
      R,BaseMLP as M,BaseRotaryEmbedding as E,BaseAttention as A,BaseDecoderLayer as Y
class Z(R):...
class I(E):...
class J(M):
 def __init__(s,g):
  super().__init__(g,mlp_bias=g.mlp_bias)
...
```

Figure 8: Example where tokens and MDL diverge: obfuscating the original library definitions reduces tokens but increases MDL.

I Clustering Analysis: CodeContests

We analyze the coherence of the clusters underlying collections in MINICODE-**CodeContests**. In particular, we compare clustering based on o4-mini generated file descriptions against task descriptions. Since task descriptions in competition coding problems are designed to hide the algorithmic approach needed to solve problem, we expect that clusters based on file descriptions are more coherent. We use Normalized Tag Instance Entropy and Herfindahl-Hirschman Index to evaluate clusterings. Figure 9 shows our clustering approach yields more thematically coherent clusters, evidenced by achieving lower entropy and higher HHI values across the entire tested range of N. We provide definitions of our measures below.

I.1 Collection Coherence Measures

We use two measures to evaluate the thematic coherence of collections: Good collections should group files with a (1) concentrated and (2) identifiable set of shared *conceptual tags*, which for CodeContests are provided as ground truth (`trees`, `graphs`, etc.).

We provide the full definitions of the collection coherence measures below.

Normalized Tag Instance Entropy: This measures the concentration of tag *instances* within a collection C. Let p_i be the proportion of the i-th unique tag type among all tag instances in C, and

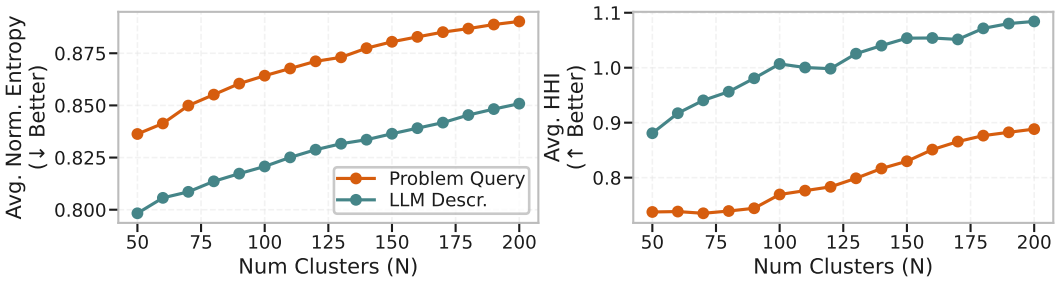

Figure 9: Clustering analysis of 4,596 Code Contest problems, comparing the thematic coherence of clusters formed using our proposed method versus REGAL-style clustering.

D_C be the number of distinct tag types in C. If $D_C > 1$, the normalized entropy H_N is defined as:

$$H_N = -\frac{\sum_{i=1}^{D_C} p_i \log_2 p_i}{\log_2 D_C} \tag{8}$$

If $D_C \leq 1$, then $H_N = 0$. Lower H_N (closer to 0) indicates higher thematic purity, meaning fewer tag types dominate the bulk of tag mentions.

Herfindahl-Hirschman Index (HHI) for Problem Presence: This measures tag concentration across distinct *problems* in a cluster C. Let s_t be the proportion of problems in C that include tag t (a problem contributes to s_t if t is one of its unique tags). A higher HHI signifies that the problems are collectively characterized by a smaller, more focused set of tags.

$$\text{HHI} = \sum_{t \in \text{Tags}(C)} s_t^2 \tag{9}$$

where $\text{Tags}(C)$ represents the set of unique tags present in cluster C.

J Benchmark Comparison

We compare our benchmark, MINICODE, to similar benchmarks in Table 4. We define creativity and design as the need to explore diverse solutions in order to find the best solution possible. For example, optimizing for program correctness alone does not require exploring a large solutions space, whereas optimizing a program for speed would. In the case of compressing large files, we must explore the large space of shared abstractions afforded by libraries in order to maximize compression.

Table 4: Comparison of Code Benchmarks

Benchmark	Creativity/Design	Scale
SWE-bench [10]	Low	Repository
Commit-0 [11]	Medium	Repository
RefactorBench [12]	Low	File
ECCO [13]	High	Function
KernelBench [14]	High	Function
MINICODE(Ours)	High	Repository

K Full MINICODE Codecontests Results

We present the full agent scores for the CodeContests split in Table 5. The results are given both for each cluster of code sources, as well as averaged across clusters.

Cluster	Agent	Tokens	CC	Pass %	MDL	MDL %
0	original	9088	95	80.3	11745.85	100.0
	sonnet 3.7	18114	176	87.0	15005.18	127.7
	sonnet 4	11121	138	80.3	9901.53	84.3
	codex-mini	9321	95	80.3	9990.74	85.1
1	original	12531	255	89.7	13431.86	100.0
	sonnet 3.7	10470	239	96.7	8933.65	66.5
	sonnet 4	11325	298	96.7	8214.42	61.2
	codex-mini	12762	255	89.7	11798.73	87.8
2	original	14087	376	89.0	15012.77	100.0
	sonnet 3.7	17345	429	91.3	13145.02	87.6
	sonnet 4	14270	356	93.0	10522.66	70.1
	codex-mini	14318	376	89.0	13273.81	88.4
3	original	14261	246	90.3	13348.82	100.0
	sonnet 3.7	20749	241	97.7	15859.02	118.8
	sonnet 4	13433	197	80.7	11937.04	89.4
	codex-mini	14495	246	90.3	11616.41	87.0
4	original	17693	336	80.7	14665.16	100.0
	sonnet 3.7	29860	358	100.0	20666.52	141.0
	sonnet 4	18684	352	82.0	12801.21	87.3
	codex-mini	17923	336	80.7	12902.09	88.0
5	original	12588	286	92.0	12790.11	100.0
	sonnet 3.7	10580	128	99.3	8435.12	65.9
	sonnet 4	10416	155	99.3	9167.85	71.7
	codex-mini	12819	286	92.0	11086.19	86.7
6	original	11020	131	54.3	13540.41	100.0
	sonnet 3.7	21747	502	88.0	19446.07	143.6
	sonnet 4	11177	143	57.3	10492.00	77.5
	codex-mini	11251	131	54.3	11651.65	86.1
7	original	12301	180	80.0	12393.73	100.0
	sonnet 3.7	16390	166	91.0	13371.59	107.9
	sonnet 4	11625	150	85.7	9304.25	75.1
	codex-mini	12534	180	80.0	10549.04	85.1
Avg	original	12946	238	82.0	13366.09	100.0
	sonnet 3.7	18157	280	93.9	14357.77	107.4
	sonnet 4	12756	224	84.4	10292.62	77.1
	codex-mini	13178	238	82.0	11608.58	86.8

Table 5: Comparison of the pass rate and compression metrics of the original files, Claude Sonnet 4 and codex-mini refactorings across CodeContests clusters.

L Refactoring examples of LIBRARIAN on Code Contests

L.1 Example 1

In code snippets 3, 2, 5, 4 one example of 2 refactoring versions. Specifically, the versions are both passing at least as many test cases as the original and they have the biggest difference in MDL among all the sample refactorings for that tuple. Sample + rerank filtering selected refactoring V2. You can observe that refactoring V1 introduces some problem specific functions like `build_max_beauty_perm()`, while refactoring V2 sticks to more generally useful functions.

```python
# ==== NEW HELPER FUNCTIONS ====
def compute_full_mask(i):
    """Return mask of all 1s of the bit-length of i."""
    return (1 << i.bit_length()) - 1

def build_max_beauty_perm(n):
    """Build permutation of 0..n maximizing sum of i^p[i]."""
    ans = [0] * (n + 1)
    used = set()
    for i in range(n, -1, -1):
        if i in used:
            continue
        mask = compute_full_mask(i)
        j = i ^ mask
        ans[i], ans[j] = j, i
        used.add(i)
        used.add(j)
    beauty = sum(i ^ ans[i] for i in range(n + 1))
    return ans, beauty

def solve_xor_sum(u, v):
    """
    Find shortest array whose xor is u and sum is v.
    Return list or None if impossible.
    """
    if u > v or (v - u) % 2:
        return None
    if u == v:
        return [] if u == 0 else [u]
    x = (v - u) // 2
    # try two elements
    if ((u + x) ^ x) == u:
        return [u + x, x]
    # fallback to three elements
    return [u, x, x]

def build_trie(keys):
    """
    Build a binary trie with counts for 30-bit numbers.
    Each node: [left_index, right_index, count].
    """
    tree = [[0, 0, 0]]
    for x in keys:
        now = 0
        tree[now][2] += 1
        for i in range(29, -1, -1):
            b = (x >> i) & 1
            if tree[now][b] == 0:
                tree[now][b] = len(tree)
                tree.append([0, 0, 0])
            now = tree[now][b]
            tree[now][2] += 1
    return tree

def trie_pop_min_xor(tree, x):
    """
    Pop one key from trie to minimize x^key and return that minimal xor.
    Decrements counts along the path.
    """
    now = 0
    res = 0
    for i in range(29, -1, -1):
        b = (x >> i) & 1
        nxt = tree[now][b]
        if nxt and tree[nxt][2] > 0:
            now = nxt
        else:
            now = tree[now][b ^ 1]
            res |= (1 << i)
        tree[now][2] -= 1
    return res
```

Listing 2: Version 1, New Helpers

```

# ########## PROGRAM: node_16:cc_python_16 ##########

from codebank import *

def main():
    import sys
    data = sys.stdin.readline()
    if not data:
        return
    n = int(data)
    perm, beauty = build_max_beauty_perm(n)
    print(beauty)
    print(*perm)

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_19:cc_python_19 ##########

from codebank import *

def main():
    import sys
    data = sys.stdin.readline
    n = int(data())
    A = list(map(int, data().split()))
    P = list(map(int, data().split()))
    trie = build_trie(P)
    O = [trie_pop_min_xor(trie, a) for a in A]
    print(*O)

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_25:cc_python_25 ##########

from codebank import *

def main():
    import sys
    u, v = map(int, sys.stdin.readline().split())
    res = solve_xor_sum(u, v)
    if res is None:
        print(-1)
    else:
        print(len(res))
        if res:
            print(*res)

if __name__ == "__main__":
    main()
```

Listing 3: Version 1, Refactored Programs

```
# ==== NEW HELPER FUNCTIONS ====
def compute_complement(i):
    return i ^ ((1 << i.bit_length()) - 1)

def trie_add(trie, x, max_bit):
    trie[0][2] += 1
    now = 0
    for i in range(max_bit, -1, -1):
        bit = (x >> i) & 1
        if trie[now][bit] == 0:
            trie[now][bit] = len(trie)
            trie.append([0, 0, 0])
        now = trie[now][bit]
        trie[now][2] += 1

def trie_find_min_xor(trie, x, max_bit):
    now = 0
    ans = 0
    for i in range(max_bit, -1, -1):
        bit = (x >> i) & 1
        if trie[now][bit] and trie[trie[now][bit]][2] > 0:
            now = trie[now][bit]
        else:
            now = trie[now][bit ^ 1]
            ans |= (1 << i)
        trie[now][2] -= 1
    return ans
```

Listing 4: Version 2, New Helpers

```python
# ########## PROGRAM: node_16:cc_python_16 ##########

from codebank import *

def main():
    import sys
    input = sys.stdin.readline
    n = int(input())
    ans = [-1] * (n + 1)
    for i in range(n, -1, -1):
        if ans[i] == -1:
            z = compute_complement(i)
            ans[i] = z
            ans[z] = i
    m = sum(i ^ ans[i] for i in range(n + 1))
    print(m)
    print(*ans)

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_19:cc_python_19 ##########

from codebank import *

def main():
    import sys
    input = sys.stdin.readline
    n = int(input())
    A = list(map(int, input().split()))
    P = list(map(int, input().split()))
    max_bit = max(max(A, default=0), max(P, default=0)).bit_length() - 1
    trie = [[0, 0, 0]]
    for x in P:
        trie_add(trie, x, max_bit)
    res = [trie_find_min_xor(trie, x, max_bit) for x in A]
    print(*res)

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_25:cc_python_25 ##########

from codebank import *

def main():
    u, v = map(int, input().split())
    if u > v or ((v - u) & 1):
        print(-1)
    elif u == 0 and v == 0:
        print(0)
    elif u == v:
        print(1)
        print(u)
    else:
        w = (v - u) // 2
        if (w & u) == 0:
            d = u + w
            print(2)
            print(d, w)
        else:
            print(3)
            print(u, w, w)

if __name__ == "__main__":
    main()
```

Listing 5: Version 2, Refactored Programs

L.2 Example 2

In code snippets 7, 6, 9, 8 is another example of 2 refactorings where V1 was better according to LIBRARIAN. We can observe that V2 creates helper functions that are overly specific to the problem. You can see that refactoring V2 introduces overly specialized functions like `dijkstra_special()` or `compute_min_moves_opposite_parity()`. In comparison, refactoring V1 generates only general versions of these functions (e.g. `dijkstra()`).

```

# ==== NEW HELPER FUNCTIONS ====
def read_ints():
    return list(map(int, input().split()))

def build_adj_undirected(n, edges):
    adj = [[] for _ in range(n)]
    for u, v, w in edges:
        adj[u].append((v, w))
        adj[v].append((u, w))
    return adj

def dijkstra(adj, src):
    from heapq import heappush, heappop
    INF = 10**18
    n = len(adj)
    dist = [INF]*n
    parent = [-1]*n
    dist[src] = 0
    heap = [(0, src)]
    while heap:
        d, u = heappop(heap)
        if d > dist[u]:
            continue
        for v, w in adj[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                parent[v] = u
                heappush(heap, (nd, v))
    return dist, parent

def reconstruct_path(parent, dest):
    path = []
    u = dest
    while u != -1:
        path.append(u+1)
        u = parent[u]
    return path[::-1]

def multi_source_bfs(neighbors, sources):
    from collections import deque
    n = len(neighbors)
    dist = [-1]*n
    dq = deque()
    for u in sources:
        if dist[u] == -1:
            dist[u] = 0
            dq.append(u)
    while dq:
        u = dq.popleft()
        for v in neighbors[u]:
            if dist[v] == -1:
                dist[v] = dist[u] + 1
                dq.append(v)
    return dist
```

Listing 6: Version 1, New Helpers

```python
# ########## PROGRAM: node_16:cc_python_16 ##########

from codebank import *

def main():
    import heapq
    n, m = read_ints()
    edges = [(u-1, v-1, w) for u, v, w in (read_ints() for _ in range(m))]
    adj = build_adj_undirected(n, edges)
    INF = 10**20
    dist = [INF]*n
    dist[0] = 0
    last_w = [0]*n
    heap = [(0, 0)]
    while heap:
        d, u = heapq.heappop(heap)
        if d > dist[u]:
            continue
        # record last edges
        for v, w in adj[u]:
            last_w[v] = w
        # expand two-edge moves
        for v, w1 in adj[u]:
            tw = last_w[v]
            for x, w2 in adj[v]:
                nd = d + (tw + w2)**2
                if nd < dist[x]:
                    dist[x] = nd
                    heapq.heappush(heap, (nd, x))
    out = []
    for x in dist:
        out.append(str(x if x < INF else -1))
    print(" ".join(out))

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_17:cc_python_17 ##########

from codebank import *

def main():
    n, m = read_ints()
    edges = [(u-1, v-1, w) for u, v, w in (read_ints() for _ in range(m))]
    adj = build_adj_undirected(n, edges)
    dist, parent = dijkstra(adj, 0)
    if dist[n-1] >= 10**18:
        print(-1)
    else:
        path = reconstruct_path(parent, n-1)
        print(*path)

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_19:cc_python_19 ##########

from codebank import *

def main():
    n = int(input())
    a = read_ints()
    # build reversed graph: for each move i->j, add edge j->i
    neighbors = [[] for _ in range(n)]
    for i, val in enumerate(a):
        for j in (i - val, i + val):
            if 0 <= j < n:
                neighbors[j].append(i)
    # BFS from all even and all odd positions separately
    even_sources = [i for i, val in enumerate(a) if val % 2 == 0]
    odd_sources  = [i for i, val in enumerate(a) if val % 2 == 1]
    dist_even = multi_source_bfs(neighbors, even_sources)
    dist_odd  = multi_source_bfs(neighbors, odd_sources)
    # for odd a[i], answer is dist to nearest even => dist_even; else dist_odd
    ans = [dist_even[i] if a[i] % 2 == 1 else dist_odd[i] for i in range(n)]
    print(*ans)

if __name__ == "__main__":
    main()
```

Listing 7: Version 1, Refactored Programs

```
#
# ==== NEW HELPER FUNCTIONS ====
def read_ints():
    return list(map(int, input().split()))

def build_undirected_weighted_graph(n, m):
    from collections import defaultdict
    adj = defaultdict(list)
    for _ in range(m):
        u, v, w = read_ints()
        u -= 1; v -= 1
        adj[u].append((v, w))
        adj[v].append((u, w))
    return adj

def dijkstra(adj, src, n):
    import heapq
    INF = 10**18
    dist = [INF]*n
    parent = [-1]*n
    visited = [False]*n
    dist[src] = 0
    heap = [(0, src)]
    while heap:
        d, u = heapq.heappop(heap)
        if visited[u]:
            continue
        visited[u] = True
        for v, w in adj.get(u, ()):
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                parent[v] = u
                heapq.heappush(heap, (nd, v))
    return dist, parent

def reconstruct_path(parent, dest):
    path = []
    while dest != -1:
        path.append(dest+1)
        dest = parent[dest]
    return path[::-1]

def dijkstra_special(e, n, src):
    import heapq
    INF = 10**18
    d = [INF]*n
    d[src] = 0
    heap = [(0, src)]
    while heap:
        cd, v = heapq.heappop(heap)
        if cd > d[v]:
            continue
        td = {}
        for u, w in e.get(v, ()):
            td[u] = w
        for u, w1 in td.items():
            for x, w2 in e.get(u, ()):
                cost = cd + (w1 + w2)**2
                if cost < d[x]:
                    d[x] = cost
                    heapq.heappush(heap, (cost, x))
    return d

def compute_min_moves_opposite_parity(a):
    from collections import deque
    n = len(a)
    go = [[] for _ in range(n)]
    ans = [-1]*n
    q = deque()
    for i, val in enumerate(a):
        for j in (i - val, i + val):
            if 0 <= j < n:
                if (a[j] % 2) != (val % 2):
                    ans[i] = 1
                    q.append(i)
                    break
                else:
                    go[j].append(i)
    while q:
        u = q.popleft()
        for v in go[u]:
            if ans[v] == -1:
                ans[v] = ans[u] + 1
                q.append(v)
    return ans
```

Listing 8: Version 2, New Helpers

```
#
# ########## PROGRAM: node_16:cc_python_16 ##########

from codebank import *

def main():
    n, m = read_ints()
    e = {}
    for _ in range(m):
        u, v, w = read_ints()
        u -= 1; v -= 1
        e.setdefault(u, []).append((v, w))
        e.setdefault(v, []).append((u, w))
    d = dijkstra_special(e, n, 0)
    print(" ".join(str(-1 if x >= 10**18 else int(x)) for x in d))

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_17:cc_python_17 ##########

from codebank import *

def main():
    n, m = read_ints()
    adj = build_undirected_weighted_graph(n, m)
    dist, parent = dijkstra(adj, 0, n)
    if dist[n-1] >= 10**18:
        print(-1)
    else:
        path = reconstruct_path(parent, n-1)
        print(" ".join(map(str, path)))

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_19:cc_python_19 ##########

from codebank import *

def main():
    n = int(input())
    a = read_ints()
    ans = compute_min_moves_opposite_parity(a)
    print(" ".join(map(str, ans)))

if __name__ == "__main__":
    main()
```

Listing 9: Version 2, Refactored Programs

M Obfuscation example

We provide the full source code, both refactored and obfuscated, for the MDL and token comparison here.

Listing 10: Refactored code from modeling_llama.py from the refactored Transformers repository.

```
from typing import Optional, Union

import torch
from torch import nn

from ...cache_utils import Cache, DynamicCache
from ...generation import GenerationMixin
from ...masking_utils import create_causal_mask
from ...modeling_layers import (
    GenericForQuestionAnswering,
    GenericForSequenceClassification,
    GenericForTokenClassification,
)
from ...modeling_outputs import (
    BaseModelOutputWithPast,
    CausalLMOutputWithPast,
)
from ...modeling_utils import PreTrainedModel
from ...processing_utils import Unpack
from ...utils import TransformersKwargs, auto_docstring, can_return_tuple, logging
from ...utils.generic import check_model_inputs
from .configuration_llama import LlamaConfig

from ..shared_library import (
```

```
    rotate_half,
    apply_rotary_pos_emb,
    repeat_kv,
    eager_attention_forward,
    RMSNorm,
    BaseMLP,
    BaseRotaryEmbedding,
    BaseAttention,
    BaseDecoderLayer,
)


logger = logging.get_logger(__name__)


class LlamaRMSNorm(RMSNorm):
    pass


class LlamaRotaryEmbedding(BaseRotaryEmbedding):
    pass


class LlamaMLP(BaseMLP):
    def __init__(self, config):
        super().__init__(config, mlp_bias=config.mlp_bias)


class LlamaAttention(BaseAttention):
    def __init__(self, config: LlamaConfig, layer_idx: int):
        super().__init__(
            config=config,
            layer_idx=layer_idx,
            attention_bias=config.attention_bias,
            sliding_window=None
        )


class LlamaDecoderLayer(BaseDecoderLayer):
    def __init__(self, config: LlamaConfig, layer_idx: int):
        super().__init__(
            config=config,
            layer_idx=layer_idx,
            norm_class=LlamaRMSNorm,
            mlp_class=LlamaMLP,
            attention_class=LlamaAttention
        )


@auto_docstring
class LlamaPreTrainedModel(PreTrainedModel):
    config: LlamaConfig
    base_model_prefix = "model"
    supports_gradient_checkpointing = True
    _no_split_modules = ["LlamaDecoderLayer"]
    _skip_keys_device_placement = ["past_key_values"]
    _supports_flash_attn = True
    _supports_sdpa = True
    _supports_flex_attn = True

    _can_compile_fullgraph = True
    _supports_attention_backend = True
    _can_record_outputs = {
        "hidden_states": LlamaDecoderLayer,
        "attentions": LlamaAttention,
    }


@auto_docstring
class LlamaModel(LlamaPreTrainedModel):
    def __init__(self, config: LlamaConfig):
        super().__init__(config)
        self.padding_idx = config.pad_token_id
        self.vocab_size = config.vocab_size

        self.embed_tokens = nn.Embedding(config.vocab_size, config.hidden_size, self.padding_idx)
        self.layers = nn.ModuleList(
            [LlamaDecoderLayer(config, layer_idx) for layer_idx in range(config.num_hidden_layers)]
        )
        self.norm = LlamaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)
        self.rotary_emb = LlamaRotaryEmbedding(config=config)
```

```
        self.gradient_checkpointing = False

        self.post_init()

    @check_model_inputs
    @auto_docstring
    def forward(
        self,
        input_ids: Optional[torch.LongTensor] = None,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        past_key_values: Optional[Cache] = None,
        inputs_embeds: Optional[torch.FloatTensor] = None,
        cache_position: Optional[torch.LongTensor] = None,
        use_cache: Optional[bool] = None,
        **kwargs: Unpack[TransformersKwargs],
    ) -> BaseModelOutputWithPast:
        if (input_ids is None) ^ (inputs_embeds is not None):
            raise ValueError("You must specify exactly one of input_ids or inputs_embeds")

        if inputs_embeds is None:
            inputs_embeds: torch.Tensor = self.embed_tokens(input_ids)

        if use_cache and past_key_values is None:
            past_key_values = DynamicCache(config=self.config)

        if cache_position is None:
            past_seen_tokens = past_key_values.get_seq_length() if past_key_values is not None else 0
            cache_position: torch.Tensor = torch.arange(
                past_seen_tokens, past_seen_tokens + inputs_embeds.shape[1], device=inputs_embeds.device
            )

        if position_ids is None:
            position_ids = cache_position.unsqueeze(0)

        causal_mask = create_causal_mask(
            config=self.config,
            input_embeds=inputs_embeds,
            attention_mask=attention_mask,
            cache_position=cache_position,
            past_key_values=past_key_values,
            position_ids=position_ids,
        )

        hidden_states = inputs_embeds
        position_embeddings = self.rotary_emb(hidden_states, position_ids)

        for decoder_layer in self.layers[: self.config.num_hidden_layers]:
            hidden_states = decoder_layer(
                hidden_states,
                attention_mask=causal_mask,
                position_ids=position_ids,
                past_key_values=past_key_values,
                cache_position=cache_position,
                position_embeddings=position_embeddings,
                **kwargs,
            )

        hidden_states = self.norm(hidden_states)
        return BaseModelOutputWithPast(
            last_hidden_state=hidden_states,
            past_key_values=past_key_values,
        )


@auto_docstring
class LlamaForCausalLM(LlamaPreTrainedModel, GenerationMixin):
    _tied_weights_keys = ["lm_head.weight"]
    _tp_plan = {"lm_head": "colwise_rep"}
    _pp_plan = {"lm_head": (["hidden_states"], ["logits"])}

    def __init__(self, config):
        super().__init__(config)
        self.model = LlamaModel(config)
        self.vocab_size = config.vocab_size
        self.lm_head = nn.Linear(config.hidden_size, config.vocab_size, bias=False)

        self.post_init()

    def set_decoder(self, decoder):
        self.model = decoder
```

```
    def get_decoder(self):
        return self.model

    @can_return_tuple
    @auto_docstring
    def forward(
        self,
        input_ids: Optional[torch.LongTensor] = None,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        past_key_values: Optional[Cache] = None,
        inputs_embeds: Optional[torch.FloatTensor] = None,
        labels: Optional[torch.LongTensor] = None,
        use_cache: Optional[bool] = None,
        cache_position: Optional[torch.LongTensor] = None,
        logits_to_keep: Union[int, torch.Tensor] = 0,
        **kwargs: Unpack[TransformersKwargs],
    ) -> CausalLMOutputWithPast:
        r"""
        Example:

        ```python
        >>> from transformers import AutoTokenizer, LlamaForCausalLM

        >>> model = LlamaForCausalLM.from_pretrained("meta-llama/Llama-2-7b-hf")
        >>> tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-2-7b-hf")

        >>> prompt = "Hey, are you conscious? Can you talk to me?"
        >>> inputs = tokenizer(prompt, return_tensors="pt")

        >>> # Generate
        >>> generate_ids = model.generate(inputs.input_ids, max_length=30)
        >>> tokenizer.batch_decode(generate_ids, skip_special_tokens=True, clean_up_tokenization_spaces=False)[0]
        "Hey, are you conscious? Can you talk to me?\nI'm not conscious, but I can talk to you."
        ```"""
        outputs: BaseModelOutputWithPast = self.model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            position_ids=position_ids,
            past_key_values=past_key_values,
            inputs_embeds=inputs_embeds,
            use_cache=use_cache,
            cache_position=cache_position,
            **kwargs,
)

        hidden_states = outputs.last_hidden_state
        slice_indices = slice(-logits_to_keep, None) if isinstance(logits_to_keep, int) else logits_to_keep
        logits = self.lm_head(hidden_states[:, slice_indices, :])

        loss = None
        if labels is not None:
            loss = self.loss_function(logits=logits, labels=labels, vocab_size=self.config.vocab_size, **kwargs)

        return CausalLMOutputWithPast(
            loss=loss,
            logits=logits,
            past_key_values=outputs.past_key_values,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
)

class LlamaForSequenceClassification(GenericForSequenceClassification, LlamaPreTrainedModel): ...

class LlamaForQuestionAnswering(GenericForQuestionAnswering, LlamaPreTrainedModel):
    base_model_prefix = "transformer"

class LlamaForTokenClassification(GenericForTokenClassification, LlamaPreTrainedModel): ...

__all__ = [
    "LlamaForCausalLM",
    "LlamaModel",
    "LlamaPreTrainedModel",
    "LlamaForSequenceClassification",
    "LlamaForQuestionAnswering",
    "LlamaForTokenClassification",
```

```
]
```

Listing 11: Obfuscated code from modeling_llama.py from the refactored Transformers repository.

```
from typing import Optional as O,Union as U
import torch as T
from torch import nn as N
from ...cache_utils import Cache as C,DynamicCache as D
from ...generation import GenerationMixin as G
from ...masking_utils import create_causal_mask as m
from ...modeling_layers import GenericForQuestionAnswering as Q,GenericForSequenceClassification as S,
    GenericForTokenClassification as K
from ...modeling_outputs import BaseModelOutputWithPast as B,CausalLMOutputWithPast as L
from ...modeling_utils import PreTrainedModel as P
from ...processing_utils import Unpack as X
from ...utils import TransformersKwargs as W,auto_docstring as a,can_return_tuple as r,logging
from ...utils.generic import check_model_inputs as c
from .configuration_llama import LlamaConfig as F
from ..shared_library import rotate_half,apply_rotary_pos_emb,repeat_kv,eager_attention_forward,RMSNorm as R,
    BaseMLP as M,BaseRotaryEmbedding as E,BaseAttention as A,BaseDecoderLayer as Y
l=logging.get_logger(__name__)
class Z(R):pass
class I(E):pass
class J(M):
 def __init__(s,g):super().__init__(g,mlp_bias=g.mlp_bias)
class H(A):
 def __init__(s,g:F,i:int):super().__init__(config=g,layer_idx=i,attention_bias=g.attention_bias,sliding_window=
     None)
class V(Y):
 def __init__(s,g:F,i:int):super().__init__(config=g,layer_idx=i,norm_class=Z,mlp_class=J,attention_class=H)
@a
class o(P):
 config:F
 base_model_prefix="model"
 supports_gradient_checkpointing=True
 _no_split_modules=["LlamaDecoderLayer"]
 _skip_keys_device_placement=["past_key_values"]
 _supports_flash_attn=True
 _supports_sdpa=True
 _supports_flex_attn=True
 _can_compile_fullgraph=True
 _supports_attention_backend=True
 _can_record_outputs={"hidden_states":V,"attentions":H}
@a
class u(o):
 def __init__(s,g:F):
  super().__init__(g)
  s.padding_idx=g.pad_token_id
  s.vocab_size=g.vocab_size
  s.embed_tokens=N.Embedding(g.vocab_size,g.hidden_size,s.padding_idx)
  s.layers=N.ModuleList([V(g,i)for i in range(g.num_hidden_layers)])
  s.norm=Z(g.hidden_size,eps=g.rms_norm_eps)
  s.rotary_emb=I(config=g)
  s.gradient_checkpointing=False
  s.post_init()
 @c
 @a
 def forward(s,input_ids:O[T.LongTensor]=None,attention_mask:O[T.Tensor]=None,position_ids:O[T.LongTensor]=None,
     past_key_values:O[C]=None,inputs_embeds:O[T.FloatTensor]=None,cache_position:O[T.LongTensor]=None,use_cache:
     O[bool]=None,**k:X[W])->B:
  if(input_ids is None)^(inputs_embeds is not None):raise ValueError("You must specify exactly one of input_ids
     or inputs_embeds")
  if inputs_embeds is None:inputs_embeds:T.Tensor=s.embed_tokens(input_ids)
  if use_cache and past_key_values is None:past_key_values=D(config=s.config)
  if cache_position is None:
   p=past_key_values.get_seq_length()if past_key_values is not None else 0
   cache_position:T.Tensor=T.arange(p,p+inputs_embeds.shape[1],device=inputs_embeds.device)
  if position_ids is None:position_ids=cache_position.unsqueeze(0)
  f=m(config=s.config,input_embeds=inputs_embeds,attention_mask=attention_mask,cache_position=cache_position,
     past_key_values=past_key_values,position_ids=position_ids)
  h=inputs_embeds
  e=s.rotary_emb(h,position_ids)
  for d in s.layers[:s.config.num_hidden_layers]:h=d(h,attention_mask=f,position_ids=position_ids,past_key_values
     =past_key_values,cache_position=cache_position,position_embeddings=e,**k)
  h=s.norm(h)
  return B(last_hidden_state=h,past_key_values=past_key_values)
@a
class t(o,G):
 _tied_weights_keys=["lm_head.weight"]
 _tp_plan={"lm_head":"colwise_rep"}
```

```
 _pp_plan={"lm_head":(["hidden_states"],["logits"])}
 def __init__(s,g):
  super().__init__(g)
  s.model=u(g)
  s.vocab_size=g.vocab_size
  s.lm_head=N.Linear(g.hidden_size,g.vocab_size,bias=False)
  s.post_init()
 def set_decoder(s,d):s.model=d
 def get_decoder(s):return s.model
 @r
 @a
 def forward(s,input_ids:O[T.LongTensor]=None,attention_mask:O[T.Tensor]=None,position_ids:O[T.LongTensor]=None,
     past_key_values:O[C]=None,inputs_embeds:O[T.FloatTensor]=None,labels:O[T.LongTensor]=None,use_cache:O[bool
     ]=None,cache_position:O[T.LongTensor]=None,logits_to_keep:U[int,T.Tensor]=0,**k:X[W])->L:
  o:B=s.model(input_ids=input_ids,attention_mask=attention_mask,position_ids=position_ids,past_key_values=
     past_key_values,inputs_embeds=inputs_embeds,use_cache=use_cache,cache_position=cache_position,**k)
  h=o.last_hidden_state
  i=slice(-logits_to_keep,None)if isinstance(logits_to_keep,int)else logits_to_keep
  g=s.lm_head(h[:,i,:])
  n=None
  if labels is not None:n=s.loss_function(logits=g,labels=labels,vocab_size=s.config.vocab_size,**k)
  return L(loss=n,logits=g,past_key_values=o.past_key_values,hidden_states=o.hidden_states,attentions=o.
     attentions)
class b(S,o):...
class x(Q,o):base_model_prefix="transformer"
class y(K,o):...
__all__=["t","u","o","b","x","y"]
```

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes], [No], or [N/A].
- [N/A] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for [N/A]).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will also be asked to include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While [Yes] is generally preferable to [No], it is perfectly acceptable to answer [No] provided a proper justification is given (e.g., error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering [No] or [N/A] is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We have carefully avoided any claims that we do repository level refactoring.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We have an explicit limitations section in the discussion.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: **[TODO]**

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We are releasing our source code and also the paper contains pseudocode that is relatively easy to reimplement.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: We already have our code released but have not had the chance to anonymize it for submission.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).

- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Most of this information is in section 4, with some remaining details C appendix

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Pervasively! We also derive an error estimator for our key results in appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We describe the time cost when using LLM APIs in the limitations section. Other aspects of our system are cehap to run

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.

- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We did a human study and it was approved by our IRB board.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

    Justification: We do not feel that there are any specific negative societal impacts of our work that would not hold more broadly for machine learning and coding agent research.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: **[TODO]**

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We do use existing datasets which are all under open licenses and which we cite.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: We do not release new assets

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [Yes]

Justification: In appendix

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [Yes]

Justification: We went through our IRB

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: Obviously the system uses LLMs! But in standard ways.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.