PromptDistill: Query-based Selective Token Retention in Intermediate Layers for Efficient Large Language Model Inference

Weisheng Jin[1], Maojia Song[1], Tej Deep Pala[2], Yew Ken Chia[1], Dorien Herremans[1], Soujanya Poria[2]
[1]Singapore University of Technology and Design, Singapore
[2]Nanyang Technological University, Singapore
Correspondence: weisheng_jin@mymail.sutd.edu.sg

Abstract

As large language models (LLMs) tackle increasingly complex tasks and longer documents, their inference cost becomes a major bottleneck. To address this, we propose PromptDistill, a novel, training-free method that improves inference efficiency while preserving generation quality. PromptDistill identifies and retains the most informative tokens by leveraging early-layer attention, preserving their hidden states while reducing computation in later layers. Unlike previous methods such as H2O, SnapKV, and GemFilter, which perform compression only after full input processing, or ignore contextual dependencies, PromptDistill dynamically allocates computation to relevant tokens while maintaining global context. Experiments on LLaMA 3.1 8B, Phi 3.5 Mini, Qwen2 7B/14B, Phi 3 128k across LongBench, InfBench, and Needle in a Haystack demonstrate that PromptDistill achieves substantial efficiency gains with minimal quality loss. With a single-stage selection strategy, it outperforms GemFilter, H2O, and SnapKV, achieving an overall 1% to 5% performance improvement with faster inference. Additionally, we explore multi-stage selection, which further improves efficiency while maintaining strong generation performance. These results demonstrate the adaptability of LLM hidden representations and provide valuable insights into optimizing inference for long-context scenarios. We have made our implementations publicly available [1].

1 Introduction

Large language models (LLMs), such as GPT, LLama, Gemini, Mistral, and Claude (Brown et al., 2020; Grattafiori et al., 2024; Team et al., 2024; Jiang et al., 2023; Anthropic, 2024) have achieved strong performance across NLP tasks. However, their widespread deployment is often hindered by the high computational and memory costs, particularly during inference with long context. As context lengths continue to grow in modern LLM applications, efficiently handling extended inputs while preserving generation quality has become a critical challenge.

Some approaches, such as MCSD and Mamba (Yang et al., 2024b; Gu and Dao, 2024), compress internal states into a single vector, reducing complexity from quadratic to linear but often degrading performance and scalability. Other methods modify model architectures to improve efficiency, but require retraining, limiting applicability to existing models (Wang et al., 2024; Gao et al., 2024). Recent works explore training-based compression (e.g., CCA-Attention (Chen et al., 2025b), QwenLong-CPRS (Shen et al., 2025)), achieving strong results but relying on additional training or architectural changes.

Training-free methods such as H2O and Snap (Zhang et al., 2023; Li et al., 2024b) can be directly applied to existing models, but still process all tokens through all layers once, incurring high cost. GemFilter (Shi et al., 2024) reduces this overhead by selecting important tokens after partial-layer processing and re-running them through the full model. However, it discards hidden representations of unselected tokens, which may lead to information loss and degraded performance.

We propose PromptDistill, a training-free method that compresses prompts while preserving global contextual information. We first process all tokens through several initial layers, then select important tokens based on query–key similarity with the last prompt token. PromptDistill differs fundamentally from GemFilter by retaining the contextualized hidden states at the selection layer and continuing to process them, rather than re-encoding from scratch. This preserves richer context without modifying the model architecture. As these hidden states have already incorporated full-context information, this preserves richer context without modifying the model architecture.

[1]https://github.com/declare-lab/PromptDistill

Proceedings of the 19th International Natural Language Generation Conference: Long Papers, pages 412–435
October 17-21, 2026 ©2026 Association for Computational Linguistics

A potential efficiency challenge is that, during generation, each new token still attends to all prompt tokens in the early layers, incurring additional time and memory costs. To address this, we introduce cache truncation, which removes the keys and values of unselected tokens from the cache for early layers. This significantly reduces computation and memory usage while maintaining performance, as shown in Section 2.3 and Section 3.3. Our method thus achieves slightly better computational efficiency than GemFilter since we avoid recomputing the selected tokens in the first few layers.

Building on its information-preserving design, PromptDistill supports multi-stage selection for progressive compression. Unlike GemFilter, which can select tokens only once at a fixed layer, PromptDistill enables retaining more tokens in early layers and fewer in later ones, improving efficiency while preserving performance and balancing computational savings with contextual fidelity.

In our experiments, we apply PromptDistill to LLMs such as LLaMA-3.1, Qwen2(7B), Qwen2.5(14B), Phi3(mini), Phi3(medium), and Phi3.5(Grattafiori et al., 2024; Yang et al., 2024a; Abdin et al., 2024) and evaluate on reliable benchmarks like Longbench, Infbench, and Needle in a Haystack (Bai et al., 2024; Zhang et al., 2024a; Nelson et al., 2024). These extensive experiments clearly show that our method outperforms baseline methods, including full attention, GemFilter, SnapKV, and H2O (Shi et al., 2024; Li et al., 2024b; Zhang et al., 2023). In particular, compared to GemFilter, our method improves time efficiency and achieves better performance.

Additionally, we provide a comprehensive analysis of other aspects of our method, demonstrating how multi-stage selection improves efficiency while preserving performance.

Our contributions are summarized as:

- We propose PromptDistill, a training-free, generic modification for LLMs that significantly boosts efficiency while maintaining comparable performance to full attention. It selects key tokens and retains their hidden states in intermediate layers to create an informative compression of the entire prompt.
- We conduct extensive zero-shot experiments with single-stage selection to evaluate PromptDistill's ability to balance effectiveness and efficiency compared to several well-designed baselines.
- With multi-stage selection, PromptDistill further reduces computational costs without sacrificing performance. The experiments also provide insights into token representativeness across different layers.

2 Method

2.1 Background

To provide a clear background for our approach, PromptDistill, discussed in Section 2.2, we first define the relevant preliminaries and introduce the token selection method used in our approach. For a typical LLM, let n represent the input token sequence length, s the attended token sequence length, d the hidden feature dimension, and i the index of a layer. For layer i, the query matrix is $Q^{(i)} \in \mathbb{R}^{n\times d}$, the key matrix is $K^{(i)} \in \mathbb{R}^{s\times d}$, and the value matrix is $V^{(i)} \in \mathbb{R}^{s\times d}$. The standard self-attention operation for each token is:

$$\text{Self-Attn}(Q^{(i)}, K^{(i)}, V^{(i)})$$
$$= \text{Softmax}(\frac{Q^{(i)}K^{(i)\top}}{\sqrt{d}}) \cdot V^{(i)},$$

where $Q^{(i)}K^{(i)\top}$ computes pairwise interactions among tokens, indicating the relative importance of each token within layer i. These pairwise scores are key for identifying salient segments in the input sequence.

Given the input prompt $\mathbf{X} = (x_1, x_2, \ldots, x_n)$ (embedded in $\mathbb{R}^d$ per token) for the LLM, our objective is to efficiently process it and generate high-quality responses. The GemFilter method illustrates that the LLMs can identify important tokens in early layers by attention matrices (Shi et al., 2024). Based on this knowledge, the token selection method plays an important role in enabling our approach, PromptDistill, to achieve its objective.

Specifically, the token selection method identifies the top k important tokens for a given selection layer r using the queries and keys defined above:

$$\text{Select}(Q_n^{(r)}, K^{(r)}, k) = \text{Top}_k\text{Argmax}_j Q_n^{(r)} K_j^{(r)T}$$

2.2 PromptDistill

In Section 2.2.1, we introduce the basic design of PromptDistill with single-stage selection, excluding cache truncation. This design has efficiency limitations, which are addressed by the cache truncation process described in Section 2.2.2. The combination of both forms our proposed method. We

Algorithm 1 LLM-forward-PromptDistill (with cache truncation)

Input: LLM $M_{0:R}$, embedded input prompt X, selection layer r, number of selected tokens k
Cache $\leftarrow$ [ ]
for $i \leftarrow 0...R$ **do**
  $X, Q_n^{(i)}, K^{(i)}, V^{(i)} \leftarrow M_i(X)$
  Cache$_i \leftarrow K^{(i)}, V^{(i)}$
  if i equals r **then**
    indices $\leftarrow$ Select$(Q_n^{(i)}, K^{(i)}, k)$
    $X \leftarrow X_{\text{indices}}$
    for $j \leftarrow 0...i$ **do**
      $K^{(j)}, V^{(j)} \leftarrow$Cache$_j$
      $K^{(j)} \leftarrow K^{(j)}_{\text{indices}}$
      $V^{(j)} \leftarrow V^{(j)}_{\text{indices}}$
      Cache$_j \leftarrow K^{(j)}, V^{(j)}$
    end for
  end if
end for
return generate(X), Cache

present comprehensive experiments comparing it to multiple baselines, with results in Section 3.2. In Section 2.2.3, we extend our method to multi-stage selections, with corresponding results in Section 3.3.

2.2.1 Basic design

We outline the basic design of PromptDistill in Algorithm 2 and Algorithm 3 (both included in Appendix Section D). While Algorithm 2 shows the overall process from receiving the prompt to generating the output, Algorithm 3 focuses on the core prompt processing and compression.

In Algorithm 2, after receiving the embedded prompt, Algorithm 3 processes it. We run several layers and collect the key-value cache as usual. After the rth layer, the token selection method in Section 2.1 selects the top k tokens. Unlike GemFilter, we retain the hidden states of selected tokens after the rth layer and process the remaining layers for these tokens. This way, even unselected tokens' information is roughly captured in the selected tokens' hidden states due to prior self-attention.

After Algorithm 3, the first generated token and previous cache are fed into the normal LLM forward process to generate the remaining tokens. Our approach differs from typical LLMs in that the key-value cache is smaller, as only selected tokens are processed in layers after the rth layer.

2.2.2 Cache truncation

Although our basic design is significantly more efficient than typical LLMs, which process all tokens across all layers and retain the full key-value cache for generation, it still has efficiency limitations. Specifically, each newly generated token continues to attend to all prompt tokens in the initial layers, leading to unnecessary computational overhead (further discussed in Section 2.3). To address this, we introduce a cache truncation mechanism to further reduce computation costs.

The forward function of PromptDistill with cache truncation is detailed in Algorithm 1. Compared to the basic design, after selecting important tokens and discarding the hidden states of unselected ones, we further prune the key-value cache by keeping only entries corresponding to the selected tokens across all processed layers. This significantly reduces both computation time and GPU memory usage, making our method even more time-efficient than GemFilter.

Crucially, our experiments show that cache truncation does not degrade performance (see Section 3.3). Therefore, we adopt PromptDistill with cache truncation as our proposed method.

2.2.3 Multi-stage selection

Unlike GemFilter, which can only select tokens at a single layer, our approach allows multiple selection stages. The key insight behind multi-stage selection is that LLMs have limited ability to identify and compress important tokens in very early layers. A single selection in a mid-layer with a small number of tokens requires processing all tokens in many layers before selection, leading to high computational and memory costs. Instead, selecting a large number of tokens in early layers and refining the selection in later layers provides a better balance between efficiency and effectiveness.

Algorithm 4 (included in Appendix Section D) integrates multi-stage selection with cache truncation. Here the selection layer r becomes a list of layers, and k corresponds to the number of selected tokens at each stage. Unlike single-stage selection, where the cache can only be truncated once after token selection is performed at a specific layer, and is thus truncated to match the final selection size, multi-stage selection allows truncation after each selection stage and introduces a user-defined parameter tt, representing how many times truncation is performed. This ensures that the final cache size matches $k[tt-1]$.

The model processes layers normally until reaching a selection layer, where it retains only the selected tokens' hidden states. If this is the p-th selection step (p starting from 0) and $p < tt$, truncation

is applied to remove unselected tokens from all completed layers. The process repeats until all layers are completed, allowing for flexible cache structuring for further token generation.

To maintain consistency in baseline comparisons, we primarily evaluate the single-stage version of PromptDistill. However, we showcase the effectiveness of multi-stage selection in our analysis.

2.3 Theoretical Analysis

In Table 1, we provide a theoretical comparison of computation time and GPU memory usage between our method and various baselines. Variable definitions are provided in the table caption. To ensure consistency, we analyze PromptDistill with single-stage selection and cache truncation. "AllKV" refers to the original LLM using full attention. Baseline values are taken from GemFilter's paper (Shi et al., 2024), with our method's values added for comparison.

Inference efficiency is analyzed in two parts: prompt computation and token generation. For GemFilter, all n tokens are processed for r layers to select k important tokens, which are then used for rerunning the full m layers to generate tokens. They put the part of processing all tokens in r layers in the prompt computation stage and the remaining, including processing the selected prompt, in the generation stage. Following this structure, we also place our computation for the first r layers in the prompt computation stage, while the subsequent processing of selected tokens and token generation falls under the generation stage.

For the time cost, in the first stage, both PromptDistill and GemFilter only need to run r layers for all tokens, while other methods run all layers for all tokens. This gives PromptDistill and GemFilter the same time cost in this stage, both better than other methods. In the second stage, while GemFilter runs selected prompt tokens for $\Theta(mhk^2d)$, PromptDistill only processes the remaining $m - r$ layers for the selected tokens' hidden states, requiring $\Theta((m-r)hk^2d)$. Both methods then generate t tokens with a key-value cache of size k for the prompt, taking $\Theta(mh(kt+t^2)d)$. Thus, the total time cost in the second stage is the sum of the selected tokens' cost and the generation cost: for GemFilter, $\Theta(mhk^2d+mh(kt+t^2)d) = \Theta(mh(k^2+t^2)d)$, while for PromptDistill it is $\Theta((m-r)hk^2d + mh(kt+t^2)d) = \Theta(mh(k^2+t^2)d - rhk^2d)$. Since GemFilter is already more efficient than AllKV, SnapKV, and H2O(Shi et al., 2024) as shown in its paper, and PromptDistill further improves generation time efficiency over GemFilter, PromptDistill achieves the best overall time efficiency.

For GPU memory usage, in the first stage, we require only r layers of model weights, which is rw, the same as GemFilter. Since all keys and values must be stored before selecting tokens, memory usage is $2rhnd$, proportional to the number of layers before selection. In the second stage, we retain the full model weights and store up to $k + t$ tokens' keys and values for all layers, the same as GemFilter, SnapKV, and H2O.

PromptDistill reduces re-computation but incurs higher peak memory than GemFilter, since it stores all token hidden states up to the selection layer. When memory permits, PromptDistill offers better performance and latency; under stricter memory constraints, GemFilter provides a more memory-efficient alternative.

Further analysis about theoretical analysis for different settings of PromptDistill (without Cache Truncation, with multi-stage selection) is presented in Appendix Section E.

3 Experiments

3.1 Experimental Setup

For our main experiments, we use PromptDistill with single-stage selection and cache truncation. Since compression methods primarily improve efficiency for long inputs, we evaluate on three authoritative long-context benchmarks: LongBench, InfBench, and Needle in a Haystack (Bai et al., 2024; Zhang et al., 2024a; Nelson et al., 2024). All models in our experiments use greedy decoding (temperature = 0), so repeated runs yield identical scores (standard deviation = 0). All implementations are built on top of FlashAttention, which is fully compatible with PromptDistill as well as baselines and further improves runtime efficiency without affecting performance.

For base models, we select Llama 3.1 8B Instruct (supporting 128K input length), Phi 3.5 Mini Instruct (supporting 128K input length), Qwen2 7B Instruct (supporting 32K input length), Qwen2.5 14B Instruct (supporting 32K input length), Phi 3 medium 128k instruct (supporting 128K input length), and Phi 3 mini 128k instruct (supporting 128K input length) (Grattafiori et al., 2024; Abdin et al., 2024; Yang et al., 2024a). Baseline methods include full attention (original model), SnapKV,

	Stage	AllKV	SnapKV and H2O	GemFilter	PromptDistill (proposed)	PromptDistill (no truncation)
Time	Prompt	$\Theta(mhn^2d)$	$\Theta(mhn^2d)$	$\Theta(rhn^2d)$	$\Theta(rhn^2d)$	$\Theta(rhn^2d)$
	Generation	$\Theta(mh(nt+t^2)d)$	$\Theta(mh(kt+t^2)d)$	$\Theta(mh(k^2+t^2)d)$	$\Theta(mh(k^2+t^2)d - rhk^2d)$	$\Theta(mh(k^2+t^2)d + rh(nt-kt-k^2)d)$
GPU	Prompt	$mw + 2mhnd$	$mw + 2hnd + 2mhkd$	$rw + 2hnd$	$rw + 2rhnd$	$rw + 2rhnd$
	Generation	$mw + 2mh(n+t)d$	$mw + 2mh(k+t)d$	$mw + 2mh(k+t)d$	$mw + 2mh(k+t)d$	$mw + 2mh(k+t)d + 2rh(n-k)d$

Table 1: Complexity Analysis. Here m is the total layer number of LLM, h is the number of attention heads, n is prompt token length, d is the hidden dimension of each head, r is the index of layer for token selection in GemFilter and PromptDistill, k is the number of selected tokens (for GemFilter and PromptDistill) and the cache size for generation (for SnapKV, H2O, GemFilter and PromptDistill), t is the number of generated tokens, w is the GPU memory used for model parameters in each layer, $n \geq max\{d, k, t\}$

H2O, and GemFilter (Li et al., 2024b; Zhang et al., 2023; Shi et al., 2024). For both PromptDistill and GemFilter (Shi et al., 2024), a selection layer must be specified. We use layer 13 (out of 32) for Llama 3.1, layer 19 (out of 32) for Phi 3.5, layer 16 (out of 28) for Qwen2, layer 24 (out of 48) for Qwen2.5(14B), layer 23 (out of 40) for Phi 3 medium, and layer 19 (out of 32) for Phi 3 mini. The choices for Llama 3.1 and Phi 3.5 follow GemFilter's settings for consistency, while Qwen2's and Phi3's selection layers are chosen based on GemFilter's performance across layers, which will be further discussed in Section 3.3. In Section 3.3, we conduct ablation studies and further exploration, testing various PromptDistill settings, including multi-stage selection and cache truncation removal.

3.2 Main Results

3.2.1 Long Bench

We first apply our method, along with all baseline approaches, to three base models on LongBench, a bilingual long-context benchmark covering tasks such as multi-hop QA and long-document reading comprehension(Bai et al., 2024). These tasks require locating and integrating information over long inputs, which makes them particularly suitable for evaluating compression methods. To ensure fair comparisons, we follow the experimental settings of (Shi et al., 2024). Under these settings, we evaluate All KV (the full-attention original model), H2O with a KV cache size of 4096, SnapKV, GemFilter, and PromptDistill, using selected tokens or KV cache sizes of 1024, 2048, and 4096. We conduct preliminary tests on the baseline methods and find that our results closely match those reported in the GemFilter paper. Therefore, we directly use their reported results for baselines and integrate our findings into the table.

In Table 2, we present all results, highlighting our differences compared to GemFilter. The results show that while PromptDistill is substantially more efficient than full attention, it achieves performance close to full attention in general and even surpasses it when using the Qwen2 model with 4096 selected tokens. Compared to H2O, which is less efficient than our method, PromptDistill demonstrates a clear performance advantage. Similarly, while SnapKV is also less efficient than our method, our results remain comparable overall. Finally, compared to GemFilter, which requires slightly more computation time but is more memory-efficient on GPUs, PromptDistill consistently achieves better results across all cases.

3.2.2 InfBench

We also consider InfBench, a benchmark that stress-tests long-context models with extremely long inputs (average length over 100K tokens), including synthetic tasks such as math finding and retrieval as well as real-world tasks like long-document QA and code understanding(Zhang et al., 2024a). These tasks require extracting information over very long sequences, which is particularly challenging for compression methods. Under these settings, we adopt the same method settings as in LongBench, although baseline results for InfBench from GemFilter's report are not available. The results are presented in Table 3. Compared to H2O and GemFilter, PromptDistill once again demonstrates a clear performance advantage, as observed in LongBench evaluations. More importantly, this time PromptDistill achieves an obvious performance advantage over SnapKV. The extremely poor performance of the Phi-3.5 model with GemFilter at 4096 selected tokens is explained in Appendix Section B.

3.2.3 Needle in a Haystack

We also evaluate the methods on Needle in a Haystack, a diagnostic benchmark that tests whether models can retrieve a single relevant fact inserted at varying positions within a very long document(Nelson et al., 2024). This setup directly

Method	NrQA	qasp	MF-en	hotpqa	2wkqa	musq	gov_rep	qmsum	m_news	trec	triqa	sam	PCount	PRe	Mean
LLama 3.1 8B Instruct															
All KV	32.02	13.04	27.34	16.23	16.05	11.22	34.52	23.41	26.89	73.0	91.64	43.8	7.16	97.73	36.72
H2O-4096	22.94	12.61	26.48	16.63	15.81	10.14	33.51	23.47	26.81	69.0	91.15	43.97	6.66	71.67	33.63
Snap-1024	31.98	11.17	25.33	14.81	15.73	10.69	26.95	22.89	25.86	67.5	91.89	42.85	7.67	98.16	35.25
Gem-1024	20.71	11.0	29.28	19.12	17.01	13.01	30.37	21.75	25.17	63.0	90.7	42.5	7.15	92.22	34.50
PrDist-1024	24.24	11.36	28.01	16.59	14.92	11.7	29.68	22.29	25.77	73.5	91.65	43.61	5.87	95.8	**35.36(+0.86)**
Snap-2048	31.45	11.94	26.24	15.73	16.03	11.66	29.64	23.24	26.44	69.5	91.48	42.68	7.21	98.03	35.80
Gem-2048	24.36	12.63	25.39	19.58	17.03	14.11	33.15	22.31	26.49	69.5	91.59	42.64	4.61	98.75	35.87
PrDist-2048	28.24	12.86	26.75	17.47	15.88	12.46	32.06	23	26.86	73.5	91.54	43.22	5.62	97.79	**36.23(+0.36)**
Snap-4096	32.13	13.12	27.38	16.11	16.08	11.6	32.39	23.47	26.76	71.5	91.64	43.46	7.33	97.24	**36.44**
Gem-4096	25.66	12.95	27.38	17.76	15.6	12.02	34.17	23.25	26.87	70.0	92.36	43.34	5.96	98.0	36.09
PrDist-4096	30.2	13.24	26.91	16.41	16.12	11.6	34.2	23.08	26.86	72.5	91.47	44.17	5.24	97.22	36.37(+0.28)
Phi 3.5 Mini 3.8B Instruct															
All KV	27.51	17.23	35.63	21.7	25.7	11.68	34.14	23.17	24.95	71.5	87.37	13.08	7.17	83.85	34.62
H2O-4096	19.74	16.23	34.17	21.02	23.05	10.49	33.42	21.95	24.95	67.5	86.13	16.71	1.55	47.46	30.31
Snap-1024	24.31	16.03	34.93	20.72	26.02	13.74	28.27	22.03	24.02	67.5	87.71	14.57	6.08	85.6	33.68
Gem-1024	16.57	18.29	35.91	24.22	26.1	9.7	30.29	18.96	23.64	64.5	85.85	23.02	0.2	81.12	32.74
PrDist-1024	23.42	21.79	33.69	23.46	24.07	12.87	30.39	22.07	24.14	68.5	87.51	13.56	3.28	83.58	**33.74(+1)**
Snap-2048	26.41	16.59	36.99	21.8	26.07	12.57	30.88	22.37	24.51	69.5	87.54	13.13	6.57	83.92	**34.20**
Gem-2048	19.63	14.84	35.99	21.38	19.72	10.13	32.39	21.24	24.71	65.0	86.49	20.47	2.17	69.5	31.69
PrDist-2048	23.85	17.81	35.98	21.57	24.78	11.13	32.79	22.22	24.7	69.5	87.42	13.5	4.43	82.75	33.75(+2.06)
Snap-4096	27.25	17.42	36.9	21.37	25.42	12.55	32.9	22.6	24.87	70.5	87.45	13.28	6.81	84.04	**34.53**
Gem-4096	20.95	19.98	35.22	28.82	28.21	13.98	34.2	22.45	25.08	64.5	85.86	18.68	3.43	65.56	33.35
PrDist-4096	24.67	16.5	35.88	20.54	25.58	12.47	34	22.51	24.97	71	87.37	13.57	5.8	82.97	34.13(+0.78)
Qwen2 7B Instruct															
All KV	24.59	44.4	45.25	12.99	12.68	8.9	37.64	24.68	27.32	78.5	88.77	46.57	6	74.5	38.06
H2O-4096	8.98	15.89	16.64	21.38	22.42	8.81	0.02	0.02	0	19	39.8	0	8	56.62	15.54
Snap-1024	24.48	42.66	45.41	12.76	12.33	8.46	28.97	24.23	26.43	71	88.47	45.89	6.67	73.17	**36.50**
Gem-1024	22.8	38.78	43.34	12.8	14.49	9.81	27.85	21.03	26.04	74.5	88.55	43.21	6.88	46.17	34.02
PrDist-1024	24.35	38.54	43.36	13.33	13.3	9.06	29.25	21.54	26.35	74.5	89.44	44.36	11	71.62	36.43(+2.41)
Snap-2048	24.44	44.69	45.05	13.06	12.16	8.72	32.45	24.35	27.1	76	88.77	46.06	6.67	74.83	37.45
Gem-2048	24.01	43.68	44.27	11.77	13.44	10.03	31.49	22.3	27.14	78.5	88.43	43.58	5.88	55.7	35.73
PrDist-2048	25.65	41.99	44.73	13.07	13.53	8.01	33.11	22.47	27.28	77.5	88.99	46.55	8.67	73.67	**37.52(+1.79)**
Snap-4096	24.87	44.36	44.14	13.32	12.39	8.67	35.06	24.83	27.3	77.5	88.77	46.45	6.75	75.17	37.83
Gem-4096	25.49	46.62	43.38	12.09	13.09	9.03	33.85	23.28	27.51	79	89.49	44.85	4.92	64.75	36.95
PrDist-4096	25.76	44.99	44.98	12.95	12.92	8.66	35.65	23.56	27.43	77.5	89.07	46.24	7.33	76.04	**38.08(+1.13)**
Qwen2.5 14B Instruct															
All KV	4.11	10.85	25.58	12.18	13.46	8.8	33.93	21.34	25.27	77.5	90.1	47.54	5.19	89.04	33.21
H2O-4096	4.01	11.3	24.82	10.92	12.85	6.84	32.9	20.1	25.26	76.0	89.25	46.85	5.1	68.08	31.02
Snap-1024	3.97	10.6	23.68	12.82	13.77	9.14	27.81	21.82	24.38	72.5	89.07	46.43	5.78	88.58	**32.17**
Gem-1024	3.5	7.31	18.91	10.67	12.92	7.05	28.73	16.43	23.16	71.5	85.67	41.72	2.42	70.25	28.59
PrDist-1024	3.39	8.92	21.13	10.68	13.28	7.51	28.3	18.07	23.95	78.0	90.2	47.11	2.82	88.83	31.59(+3)
Snap-2048	3.98	10.84	25.4	12.65	13.95	8.81	30.63	21.55	25.23	74.5	89.94	46.69	5.86	88.21	**32.73**
Gem-2048	3.92	9.76	21.23	12.01	13.23	7.31	31.2	17.51	24.9	75.5	89.37	42.38	2.06	82.92	30.95
PrDist-2048	3.63	9.85	22.35	11.89	13.57	7.63	31.03	19.07	25.01	79.0	90.38	47.79	3.62	90.04	32.49(+1.54)
Snap-4096	4.08	10.99	25.29	12.29	13.68	9.07	32.24	21.15	25.42	77.0	89.99	47.12	5.65	88.29	33.02
Gem-4096	3.69	10.72	23.03	11.74	13.59	7.65	33.32	18.46	25.36	77.0	90.35	44.37	1.75	90.04	32.22
PrDist-4096	3.65	10.71	24.75	11.92	13.56	8.4	33.07	19.78	25.56	78.0	90.27	47.96	4.42	90.92	**33.07(+0.85)**
Phi 3 medium 128k instruct															
All KV	19.65	22.83	36.77	26.09	21.87	18.59	35.28	21.01	27.03	74.0	90.28	44.74	7.02	89.33	38.18
H2O-4096	14.36	23.45	36.52	24.03	20.57	13.21	34.83	20.21	26.85	70.5	89.6	43.25	6.37	54.88	34.19
Snap-1024	19.54	18.82	35.08	27.38	22.3	20.04	27.21	22.9	25.66	72.0	90.72	44.98	8.64	88.25	37.39
Gem-1024	15.9	20.08	33.22	20.03	21.73	15.8	31.04	20.45	27.02	62.0	81.94	32.48	2.03	78.83	33.04
PrDist-1024	17.7	22.69	35.79	26.0	22.97	17.91	31.07	21.94	26.24	72.0	90.95	43.75	9.61	88.08	**37.62(+4.58)**
Snap-2048	20.69	21.33	35.65	28.0	22.23	18.8	30.93	22.06	26.66	73.0	90.45	44.46	9.43	88.58	38.02
Gem-2048	18.61	22.73	34.03	22.76	18.8	17.09	32.81	20.94	26.92	66.5	87.18	40.03	4.2	89.42	35.86
PrDist-2048	19.65	22.28	36.28	27.14	23.34	19.68	33.29	21.64	26.98	73.5	90.62	44.6	8.89	87.17	**38.22(+2.36)**
Snap-4096	20.61	22.62	36.58	28.03	21.63	18.85	33.47	22.01	27.0	74.0	90.28	45.12	9.43	88.5	38.44
Gem-4096	9.68	14.87	17.05	11.93	8.99	5.31	2.51	6.61	22.19	41.5	47.19	11.04	1.0	0.0	14.28
PrDist-4096	20.93	22.23	35.47	28.15	22.51	18.84	34.24	21.41	27.01	74.0	90.45	45.19	10.4	88.08	**38.49**
Phi 3 mini 128k instruct															
All KV	11.07	13.07	31.75	17.83	14.67	12.6	34.73	22.73	26.15	67.5	86.71	25.85	4.5	73.79	31.64
H2O-4096	10.15	12.53	30.54	15.77	15.81	9.51	34.07	20.17	26.14	64.0	83.99	19.55	5.27	45.6	28.08
Snap-1024	10.19	11.0	28.14	17.9	13.28	12.58	27.86	21.38	24.62	66.5	86.47	30.93	4.17	71.16	30.44
Gem-1024	9.25	11.05	31.36	15.23	15.33	9.37	31.24	17.21	24.95	62.5	82.85	22.65	2.92	50.25	27.58
PrDist-1024	9.9	14.01	30.49	17.21	15.55	12.41	30.9	21.14	24.9	67.0	86.74	23.67	4.0	68.58	**30.46(+2.88)**
Snap-2048	11.3	12.55	27.77	17.9	15.1	12.89	30.47	22.14	25.75	67.5	86.76	32.37	4.04	73.16	**31.41**
Gem-2048	10.75	12.52	30.56	15.45	15.2	11.03	33.19	20.12	25.73	67.0	87.15	21.23	2.75	39.08	27.98
PrDist-2048	10.05	13.43	29.42	17.76	14.86	12.15	33.22	21.44	25.91	69.0	86.84	23.0	4.0	69.33	30.74(+2.76)
Snap-4096	11.06	13.57	29.9	17.82	15.63	13.81	33.05	22.43	25.9	67.5	86.51	30.88	4.62	74.2	**31.92**
Gem-4096	5.77	6.66	11.58	4.84	3.23	2.59	6.35	4.5	21.75	27.5	13.98	4.03	1.02	1.08	8.21
PrDist-4096	11.25	13.64	29.04	17.81	15.24	12.71	34.7	21.74	26.03	68.0	86.74	23.51	4.5	73.28	31.3

Table 2: Evaluation Results for Longbench (PromptDistill's difference from main baseline GemFilter is labeled). All KV denotes the uncompressed model, offering the highest performance but lowest efficiency. Compression methods are designed to approach its performance while exceeding others in efficiency.

Method	passkey	N_str	kv_retr	Lb_sum_en	Lb_ch_en	Lb_qa_en	Lb_qa_chn	Ld_qa_en	mathf	coderun	codedebug	Mean
LLama 3.1 8B Instruct												
All KV	81.36	81.36	66.6	18.86	67.25	14.85	12.35	15.0	33.14	1.0	1.27	35.73
H2O-4096	6.78	6.1	2.0	18.86	47.16	8.79	10.32	7.0	34.0	0.0	18.53	14.5
Snap-1024	81.36	71.36	0.4	15.45	67.69	11.72	11.89	12.0	34.0	1.0	1.27	28.01
Gem-1024	81.36	81.36	1.8	15.66	36.68	12.08	13.11	14.0	34.29	0.0	19.29	28.15
PrDist-1024	81.36	78.98	2.0	16.02	65.5	12.04	12.39	9.5	32.57	1.25	1.27	**28.44(+0.29)**
Snap-2048	81.36	77.12	1.2	16.78	67.69	13.41	11.86	10.5	34.0	1.0	1.02	28.72
Gem-2048	81.36	81.36	4.4	17.02	43.23	11.77	12.91	17.5	31.71	0.0	19.04	29.12
PrDist-2048	81.36	79.49	4.0	16.94	66.38	14.28	12.2	13.0	32.57	1.25	1.02	**29.32(+0.2)**
Snap-4096	81.36	78.98	3.6	16.44	67.69	14.12	12.21	15.5	34.0	0.75	1.02	29.61
Gem-4096	81.36	81.36	13.0	18.05	44.54	11.81	13.93	11.5	35.14	0.0	14.97	29.61
PrDist-4096	81.36	80.34	9.2	18.26	66.81	14.71	12.48	14.5	32.86	0.25	0.76	**30.14(+0.53)**
Phi 3.5 Mini 3.8B Instruct												
All KV	40.68	40.68	26.8	17.68	48.03	9.1	12.88	10.0	24.57	1.0	19.29	22.79
H2O-4096	6.78	6.78	0.4	16.71	35.37	6.84	9.78	6.5	29.14	0.75	5.58	11.33
Snap-1024	62.54	15.25	0.2	13.87	56.33	8.62	12.71	11.5	14.29	0.0	17.51	19.35
Gem-1024	40.68	40.68	0.6	13.53	45.41	7.01	11.48	7.0	15.14	1.5	10.41	17.59
PrDist-1024	40.68	40.68	1.0	15.18	46.72	7.92	12.88	11.5	22.29	1.0	17.01	**19.71(+2.12)**
Snap-2048	40.51	15.93	0.2	15.18	44.54	8.19	12.95	12.5	26.86	1.25	20.05	18.01
Gem-2048	40.68	40.51	2.0	14.42	45.85	7.33	11.69	7.5	19.43	2.0	16.75	18.92
PrDist-2048	40.68	40.68	2.0	16.0	45.85	8.13	12.53	9.0	23.71	2.0	18.02	**19.87(+0.95)**
Snap-4096	40.68	17.29	1.4	15.83	44.54	8.33	13.2	12.0	26.29	1.75	20.05	18.31
Gem-4096	0.0	0.0	0.0	1.53	27.51	5.17	1.38	0.5	1.71	0.0	0.0	3.44
PrDist-4096	40.68	40.68	6.0	16.17	46.72	8.25	12.7	10.0	24.57	1.75	18.02	**20.5**
Qwen2 7B Instruct												
All KV	27.12	27.12	13.8	16.42	54.59	6.23	10.0	7.0	28.57	0.0	26.14	19.73
H2O-4096	0.85	3.05	0.0	0.0	47.6	1.41	2.11	1.0	24.29	0.0	0.0	7.3
Snap-1024	24.75	5.25	0.4	13.92	55.02	5.91	9.93	8.0	29.14	0.0	26.9	16.29
Gem-1024	27.12	27.12	0.2	13.43	43.67	8.17	18.22	12.0	10.0	0.0	28.43	17.12
PrDist-1024	27.12	27.12	0.2	13.72	54.59	5.66	9.67	11.5	27.14	0.0	28.93	**18.7(+1.58)**
Snap-2048	25.25	6.78	0.6	14.51	55.02	6.11	9.93	8.5	28.29	0.25	26.65	16.54
Gem-2048	27.12	27.12	0.2	14.45	44.54	9.17	18.01	14.0	10.86	0.0	28.17	17.6
PrDist-2048	27.12	27.12	0.2	15.01	55.9	5.97	9.61	12.0	28.29	0.0	28.43	**19.06(+1.46)**
Snap-4096	26.44	9.66	1.2	15.18	55.02	6.21	9.91	9.0	29.14	0.25	26.9	17.17
Gem-4096	27.12	27.12	0.8	15.4	44.54	7.56	12.81	16.0	12.0	0.0	28.68	17.46
PrDist-4096	27.12	27.12	0.6	15.74	56.33	6.09	9.59	11.0	28.29	0.0	28.68	**19.14(+1.68)**
Qwen2.5 14B Instruct												
All KV	27.12	27.12	18.2	17.65	60.7	7.01	8.5	18.5	34.57	0.0	38.07	23.4
H2O-4096	6.78	6.78	0.8	16.78	53.28	4.86	7.29	12.0	31.43	0.0	32.23	15.66
Snap-1024	27.12	21.36	0.8	16.24	60.7	6.79	9.02	13.5	35.43	0.0	38.83	**20.89**
Gem-1024	27.12	27.12	0.0	15.32	40.17	4.1	7.51	16.0	13.71	0.0	25.38	16.04
PrDist-1024	27.12	27.12	0.0	15.93	58.95	5.38	7.86	13.0	33.71	0.0	35.28	20.4(+4.36)
Snap-2048	27.12	22.71	4.2	16.39	60.26	6.9	8.38	14.5	36.0	0.0	38.83	**21.39**
Gem-2048	27.12	27.12	0.0	16.13	44.1	4.59	8.08	17.0	11.71	0.0	25.89	16.52
PrDist-2048	27.12	27.12	0.2	16.76	58.95	5.88	8.45	16.0	34.86	0.0	35.79	21.01(+4.49)
Snap-4096	27.12	24.75	9.6	16.85	59.83	6.96	8.68	17.5	36.57	0.0	38.83	**22.43**
Gem-4096	27.12	27.12	0.2	16.57	47.16	5.15	8.61	21.0	10.57	0.25	26.65	17.31
PrDist-4096	27.12	27.12	0.2	17.49	60.26	6.15	8.26	18.0	35.71	0.0	36.04	21.49(+4.18)
Phi 3 medium 128k instruct												
All KV	40.68	40.51	8.6	15.78	62.01	8.25	15.93	6.5	4.86	1.0	29.7	21.26
H2O-4096	6.78	6.61	0.4	15.9	46.29	6.69	12.63	6.5	31.14	1.0	27.66	14.69
Snap-1024	40.68	20.0	0.2	12.55	55.46	7.83	14.36	5.5	5.14	1.0	29.44	17.47
Gem-1024	40.68	40.68	1.0	15.6	48.03	6.88	14.12	11.0	11.14	0.0	28.17	19.75
PrDist-1024	40.68	40.68	1.0	13.91	62.45	8.22	15.52	8.5	0.0	1.5	29.95	**20.22(+0.47)**
Snap-2048	40.68	24.07	0.2	13.81	58.95	8.16	14.8	8.0	5.14	1.0	29.7	18.59
Gem-2048	40.68	40.68	2.8	15.43	48.03	6.7	13.93	7.5	15.71	0.0	29.44	20.08
PrDist-2048	40.68	40.68	1.8	14.76	63.76	8.26	15.56	8.5	0.29	1.5	29.95	**20.52(+0.44)**
Snap-4096	40.68	29.32	0.4	14.1	59.83	8.4	14.93	10.5	5.43	0.5	30.46	19.5
Gem-4096	0.34	0.0	0.0	0.28	6.11	2.86	2.94	0.0	7.43	0.0	0.25	1.84
PrDist-4096	40.68	40.68	3.0	14.17	63.32	8.67	16.0	9.5	1.43	1.0	29.7	**20.74**
Phi 3 mini 128k instruct												
All KV	40.68	40.51	26.4	15.67	55.02	8.71	11.78	7.0	26.86	1.25	21.32	23.2
H2O-4096	6.78	6.44	1.0	15.23	44.98	6.63	9.25	10.0	30.0	0.25	12.18	12.98
Snap-1024	40.51	13.9	0.2	12.24	55.02	8.11	10.93	7.0	28.57	1.25	21.07	18.07
Gem-1024	40.68	40.68	2.0	13.49	42.79	6.79	12.88	6.0	16.86	1.75	25.38	19.03
PrDist-1024	40.68	40.51	2.2	13.98	55.46	8.76	11.62	8.5	24.86	1.0	17.01	**20.42(+1.39)**
Snap-2048	40.51	14.75	0.6	13.4	55.02	9.15	11.01	7.0	26.29	1.5	21.07	18.21
Gem-2048	40.68	40.51	4.8	13.97	43.23	7.25	12.29	5.0	21.43	1.75	24.62	19.59
PrDist-2048	40.68	40.51	5.2	14.66	54.15	8.01	12.03	11.0	26.57	1.5	20.3	**21.33(+1.74)**
Snap-4096	40.51	15.93	1.6	14.05	55.02	9.12	11.77	8.5	27.43	1.25	22.08	18.84
Gem-4096	0.0	0.0	0.0	2.97	33.19	3.53	1.46	0.0	1.71	0.0	0.0	3.9
PrDist-4096	40.68	40.51	12.2	14.42	54.15	8.19	12.7	9.5	25.14	1.25	21.57	**21.85**

Table 3: Evaluation Results for Infbench (PromptDistill's difference from main baseline GemFilter is labeled). All KV denotes the uncompressed model, offering the highest performance but lowest efficiency. Compression methods are designed to approach its performance while exceeding others in efficiency.

Method	LLa3	Ph3.5	Q2 7B	Q2.5 14B	Ph3 Med	Ph3 Mi
All KV	85.2	91.7	91.2	90.4	91.7	91.7
Snap-1024	64.0	91.7	**90.5**	**90.6**	90.2	91.7
Gem-1024	81.8	91.7	69.7	68.7	91.7	91.7
PrDist-1024	**82.6**	91.7	71.6	73.3	91.7	91.7
Snap-2048	71.7	91.7	**90.8**	**90.3**	91.7	91.7
Gem-2048	**84.7**	91.7	81.8	79.7	91.7	91.7
PrDist-2048	84.3	91.7	82.2	83.8	91.7	91.7
Snap-4096	77.1	91.7	**91.0**	**90.4**	91.7	91.7
Gem-4096	83.3	5.1	89.3	86.9	7.9	7.4
PrDist-4096	**84.6**	91.7	90.2	88.5	91.7	91.7

Table 4: Evaluation Results for Needle in a Haystack. Instruct versions of the models are used.

k	r	Mean Score	Δ_1	Δ_2
1024	13	35.36	0	0.86
8192 1024	8 13	35.39	0.03	0.89
16384 8192 1024	7 8 13	35.42	0.06	0.92
16384 8192 1024	5 8 13	35.53	0.17	1.03
16384 8192 1024	3 8 13	35.42	0.06	0.92
16384 8192 1024	1 8 13	35.32	-0.04	0.82

Table 5: Top-5 multi-stage selection performance on LongBench (LLama 3.1 8B Instruct), with k and r denoting the token numbers and stages for selection, respectively (space-separated for multiple stages). Here, Δ_1 is the difference in mean score from PromptDistill-13-1024, and Δ_2 from GemFilter-13-1024.

probes whether context compression or token pruning discards the critical evidence needed to answer the query. While GemFilter (Shi et al., 2024) only evaluates selected tokens or KV cache sizes of 1024, we conduct a more comprehensive evaluation using settings similar to those in LongBench and InfBench.

Since our results differ from those in GemFilter's report—likely due to differences in hardware, Python, or CUDA versions—we do not use their results as baselines for fair comparison.

For maximum input length, we set 120K for LLaMA 3.1 8B Instruct (following GemFilter's work (Shi et al., 2024)), 86K for Phi-3.5 Mini 3.8B Instruct (due to GPU memory limitations for the full attention method), and 32K for Qwen2 7B Instruct (due to its input length limit). The results are presented in Section 3.2.2, with additional detailed figures in Appendix Section I. For Phi-3.5, all method settings yield the same results except GemFilter-4096, which encounters the same issue discussed in Section 3.2.2 and in Appendix Section B. Overall, PromptDistill outperforms GemFilter in most cases and surpasses SnapKV in half of the evaluations.

3.3 Analysis

We empirically determine the selection layer for Qwen2 7B Instruct, Qwen2.5 14B Instruct, Phi 3 medium 128k instruct, and Phi 3 mini 128k instruct in our main evaluations by assessing GemFilter across different layers on LongBench. We select the layer where GemFilter performs well while ensuring it is as early as possible for efficiency. This approach ensures a fair comparison, as choosing a layer where GemFilter performs well highlights its selection ability. We show further details of this selection process for Qwen2 7B Instruct as an example in Appendix Section H. We present detailed analysis and empirical results in Appendix Section C, showing that Cache Truncation substantially increases efficiency while maintaining performance. In addition to the performance results, we include latency measurements in Appendix Section F to complement the theoretical efficiency analysis in Section 2.3 and illustrate the empirical time efficiency of PromptDistill relative to the baselines.

3.3.1 Multi-stage Selection Analysis

For multi-stage PromptDistill, different configurations can be explored. Using LongBench for evaluation, we start from a baseline that selects 1024 tokens at layer 13, with the KV cache truncated to 1024 tokens in all trials. Well-performing settings are summarized in Appendix Section G (Section G), and the best ones are shown in Table 5.

We first add a selection layer before layer 13 that keeps 2048 tokens. Since early-layer selection of 2048 tokens degrades performance, we focus on layers selecting 4096 tokens. Layer 10 performs best, leading us to test 8192 tokens, first on top of 1024@L13, then 4096@L10+1024@L13. This yields four strong configurations (rows 3–6 in Section G). Further adding 16384-token layers gives the remaining results.

Both tables highlight clear contrasts with GemFilter and single-stage PromptDistill. Some multi-stage settings match single-stage performance, while others slightly reduce accuracy but gain higher efficiency, still surpassing GemFilter. Combined with the theoretical analysis in Section 2.3, these results confirm that multi-stage selection can substantially close the memory–efficiency gap between PromptDistill and GemFilter, offering much faster inference while preserving performance advantages.

Although the design space of multi-stage selec-

tion is large, our experiments reveal a simple heuristic: starting from a strong single-stage configuration, progressively add earlier selection stages that retain more tokens than later ones to further reduce computation. A small validation set suffices to calibrate layers and token budgets, consistently yielding efficient multi-stage configurations with minimal performance loss.

4 Conclusion

PromptDistill is a novel, training-free method that improves large language model inference efficiency by retaining only the most informative hidden states from intermediate layers while preserving essential context. Unlike prior methods such as GemFilter, SnapKV, and H2O, it achieves a better balance between speed, memory, and output quality. Moreover, multi-stage selection further enhances efficiency with minimal performance loss. Extensive experiments across diverse LLMs and benchmarks demonstrate that PromptDistill is both effective and generalizable for long-context tasks.

Limitations

While PromptDistill demonstrates clear improvements in inference efficiency and maintains strong generation quality, several limitations remain. First, the method relies on attention scores from early or intermediate layers to select important tokens. This assumes that early attention signals are sufficient for identifying informative tokens, which may not always hold, especially for tasks requiring nuanced understanding at later layers. Although multi-stage selection partially addresses this, optimal configurations remain empirically determined and model-specific, requiring additional tuning.

Second, PromptDistill incurs additional memory overhead due to its design choice to preserve contextual richness. Specifically, in order to compute informative hidden states for the selected tokens, it must first process and store hidden states of all tokens up to the selection layer. This ensures that selected tokens embed sufficient context information, but also leads to increased memory usage compared to methods like GemFilter, which discard the intermediate states of unselected tokens early on. While PromptDistill reduces recomputation, this storage strategy may limit scalability, particularly for very long contexts or resource-constrained deployments. Future methods may better balance memory usage and context preservation by selectively compressing or approximating the influence of discarded tokens.

Third, our analysis provides theoretical guarantees only for efficiency gains, not for generation quality after token pruning. The observed performance improvements are empirical and may vary across prompts and tasks. In particular, we do not yet have formal bounds ensuring that pruned prompts retain all task-critical information, which implies potential prompt sensitivity and unavoidable information loss from discarded tokens. Establishing theoretical conditions for safe pruning and designing mechanisms to detect or mitigate such failures remain important directions for future work.

Lastly, our current evaluation is conducted on long-context benchmarks such as LongBench, InfBench, and Needle-in-a-Haystack, which are primarily focused on extractive or reading comprehension tasks. While these benchmarks are valuable, they only capture a narrow slice of the challenges faced by LLMs in real-world usage. In practice, models often need to solve more diverse and open-ended problems—such as multi-step reasoning, decision-making, or synthesis of unstructured information—where the ability to efficiently locate, integrate, and reason over relevant content is crucial. Developing broader, task-oriented benchmarks that reflect these realistic and complex settings would allow for a more comprehensive evaluation of long-context methods like PromptDistill.

Acknowledgments

This work has received support from MOE under grant number MOE-T2EP20124-0014.

References

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *Preprint*, arXiv:2404.14219.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. Longbench: A bilingual, multitask benchmark for long context understanding. *Preprint*, arXiv:2308.14508.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, et al. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Guanzheng Chen, Qilong Feng, Jinjie Ni, Xin Li, and Michael Qizhe Shieh. 2025a. Long-context inference with retrieval-augmented speculative decoding. *Preprint*, arXiv:2502.20330.

Yaofo Chen, Zeng You, Shuhai Zhang, Haokun Li, Yirui Li, Yaowei Wang, and Mingkui Tan. 2025b. Core context aware transformers for long context language modeling. *arXiv preprint arXiv:2412.12465v3*.

Tianyu Gao, Alexander Wettig, Howard Yen, and Danqi Chen. 2024. How to train long-context language models (effectively). *Preprint*, arXiv:2410.02660.

Suyu Ge, Xihui Lin, Yunan Zhang, Jiawei Han, and Hao Peng. 2024. A little goes a long way: Efficient long context training and inference with partial contexts. *Preprint*, arXiv:2410.01485.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, et al. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Albert Gu and Tri Dao. 2024. Mamba: Linear-time sequence modeling with selective state spaces. *Preprint*, arXiv:2312.00752.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *Preprint*, arXiv:2310.06825.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Preprint*, arXiv:2407.02490.

Jingyao Li, Han Shi, Xin Jiang, Zhenguo Li, Hong Xu, and Jiaya Jia. 2024a. Quickllama: Query-aware inference acceleration for large language models. *Preprint*, arXiv:2406.07528.

Yucheng Li, Bo Dong, Chenghua Lin, and Frank Guerin. 2023. Compressing context to enhance inference efficiency of large language models. *Preprint*, arXiv:2310.06201.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024b. Snapkv: Llm knows what you are looking for before generation. *Preprint*, arXiv:2404.14469.

Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu. 2024a. Retrievalattention: Accelerating long-context llm inference via vector retrieval. *Preprint*, arXiv:2409.10516.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023a. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Preprint*, arXiv:2305.17118.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. 2023b. Deja vu: Contextual sparsity for efficient LLMs at inference time. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 22137–22176. PMLR.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024b. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*.

Elliot Nelson, Georgios Kollias, Payel Das, Subhajit Chaudhury, and Soham Dan. 2024. Needle in the haystack for memory based large language models. *Preprint*, arXiv:2407.01437.

Weizhou Shen, Chenliang Li, Fanqi Wan, Shengyi Liao, Shaopeng Lai, Bo Zhang, Yingcheng Shi, Yuning Wu, Gang Fu, Zhansheng Li, Bin Yang, Ji Zhang, Fei Huang, Jingren Zhou, and Ming Yan. 2025. Qwenlong-cprs: Towards inf-llms with dynamic context optimization. *arXiv preprint arXiv:2505.18092v2*.

Zhenmei Shi, Yifei Ming, Xuan-Phi Nguyen, Yingyu Liang, and Shafiq Joty. 2024. Discovering the gems in early layers: Accelerating long-context llms with 1000x input token reduction. *Preprint*, arXiv:2409.17422.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, Soroosh Mariooryad, Yifan Ding, Xinyang Geng, Fred Alcober, Roy Frostig, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *Preprint*, arXiv:2403.05530.

Xindi Wang, Mahsa Salmani, Parsa Omidi, Xiangyu Ren, Mehdi Rezagholizadeh, and Armaghan Eshaghi. 2024. Beyond the limits: A survey of techniques

to extend the context length in large language models. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pages 8299–8307. International Joint Conferences on Artificial Intelligence Organization. Survey Track.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*.

Fangyuan Xu, Tanya Goyal, and Eunsol Choi. 2025a. Refreshkv: Updating small kv cache during long-form generation. *Preprint*, arXiv:2411.05787.

Fangyuan Xu, Weijia Shi, and Eunsol Choi. 2023. Recomp: Improving retrieval-augmented lms with compression and selective augmentation. *Preprint*, arXiv:2310.04408.

Yuhui Xu, Zhanming Jie, Hanze Dong, Lei Wang, Xudong Lu, Aojun Zhou, Amrita Saha, Caiming Xiong, and Doyen Sahoo. 2025b. Think: Thinner key cache by query-driven pruning. *Preprint*, arXiv:2407.21018.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, et al. 2024a. Qwen2 technical report. *Preprint*, arXiv:2407.10671.

Hua Yang, Duohai Li, and Shiman Li. 2024b. Mcsd: An efficient language model with diverse fusion. arXiv preprint.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Khai Hao, Xu Han, Zhen Leng Thai, Shuo Wang, Zhiyuan Liu, and Maosong Sun. 2024a. ∞bench: Extending long context evaluation beyond 100k tokens. *Preprint*, arXiv:2402.13718.

Xuechen Zhang, Xiangyu Chang, Mingchen Li, Amit Roy-Chowdhury, Jiasi Chen, and Samet Oymak. 2024b. Selective attention: Enhancing transformer through principled context control. In *Advances in Neural Information Processing Systems*, volume 37, pages 11061–11086. Curran Associates, Inc.

Yuwei Zhang, Jayanth Srinivasa, Gaowen Liu, and Jingbo Shang. 2025. Attention reveals more than tokens: Training-free long-context reasoning with attention-guided retrieval.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H_2o: Heavy-hitter oracle for efficient generative inference of large language models. *Preprint*, arXiv:2306.14048.

A Related Work

A.1 Acceleration of Long-Context LLM Inference

As large language models (LLMs) are increasingly applied to long-context tasks, the computational and memory overhead during inference has become a critical bottleneck. FastGen (Ge et al., 2023) adaptively selects one of five KV cache compression methods based on attention head profiling, reducing generation overhead. StreamingLLM (Xiao et al., 2023) mitigates the "attention sink" issue by introducing placeholder tokens and a local attention mechanism. MInference (Jiang et al., 2024) leverages sparse computation patterns like A-shape, Vertical-Slash, and Block-Sparse to reduce latency in the prefilling stage. ThinK (Xu et al., 2025b) prunes redundant attention channels by observing imbalances in the key cache's channel dimension. SnapKV (Li et al., 2024b) clusters KV positions to focus on key prompt features. Mistral (Jiang et al., 2023) employs a sliding window strategy for efficient long-text inference, while KIVI (Liu et al., 2024b) applies 2-bit KV cache quantization to reduce memory and computation costs. LongGen (Ge et al., 2024) combines sparse attention patterns with lightweight training strategies, while RefreshKV (Xu et al., 2025a) dynamically switches between full-context and partial token attention to improve efficiency. However, our work builds upon these strategies by focusing on identifying informative prompt tokens in intermediate layers. This approach enables efficient token selection without additional training, achieving minimal performance degradation. Furthermore, by leveraging this structure during tuning, we demonstrate improved adaptability of the model under extreme compression scenarios.

A.2 Prompt Selection for Computation Reduction

Minimizing the number of prompt tokens processed during computation is a straightforward yet powerful method for enhancing efficiency. QLLM (Li et al., 2024a) partitions text into blocks and selects relevant ones using query tokens. H2O (Zhang et al., 2023) retains only influential tokens, while Scissorhands (Liu et al., 2023a) discards less relevant tokens based on the "importance persistence hypothesis." Retrieval-augmented speculative Decoding (Chen et al., 2025a) incorporates a retrieval-augmented generation (RAG) strategy,

where a draft LLM generates content based on a simplified context. RetrievalAttention (Liu et al., 2024a) indexes and searches KV cache vectors to selectively retain only information relevant to the current generation. RECOMP (Xu et al., 2023) integrates extraction and summarization techniques to preprocess the original context before feeding it as a prompt, effectively condensing the information. Deja Vu (Liu et al., 2023b) trains a lightweight algorithm to predict context sparsity at each layer, providing guidance for token selection in subsequent steps. Selective Context (Li et al., 2023) leverages self-information calculations to prune redundant lexical units directly. Meanwhile, Selective Attention (Zhang et al., 2024b) introduces targeted modulation within the self-attention mechanism, allowing the model to adjust its attention distribution dynamically based on context importance. ATTRIEVAL (Zhang et al., 2025) employs attention mechanisms to better capture implicit facts, ensuring minimal information loss while reducing token counts. Unlike these approaches, which mainly reduce input context tokens, our method targets effective token selection across intermediate layers, achieving greater resource efficiency while maintaining strong performance in constrained environments.

B Case of extreme performance for GemFilter

We observe that in InfBench and Needle in a Haystack, the Phi-3.5 model with GemFilter performs exceptionally poorly when the selected token count is 4096. After further investigation, we find that on our machines, Phi-3.5 fails to generate meaningful responses when the input length is exactly 4096, whereas inputs of 4000 or 4100 tokens still yield high-quality outputs. This issue may stem from factors such as the specific hardware used, CUDA version differences, or the fine-tuning process of the Phi-3.5 model. Since GemFilter selects important tokens and reruns them, the 4096-token prompt length disrupts its performance.

C Cache Truncation Analysis

Here, we present Figure 1, a bar chart comparing PromptDistill with and without cache truncation on LongBench using Llama 3.1 8B Instruct (selection layer 13, selected token counts 1024, 2048, 4096). We also provide detailed scores for all tasks in Table 6. The results clearly show that performance does not decrease and even improves in these cases. Since PromptDistill with cache truncation is substantially more efficient than the basic design without truncation (discussed in Section 2.3) while maintaining performance, we confidently adopt cache truncation for both higher efficiency and stable performance.

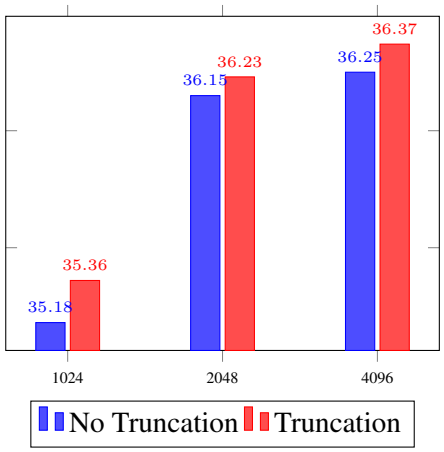


Figure 1: The effect of cache truncation.

Algorithm 2 PromptDistill

Input: LLM M, embedded input prompt X, selection layer(s) r, number(s) of selected tokens k, max generated token length T, (optional) truncation times tt

\# for single-stage selection, r, k will be integers

\# for multi-stage selection, r, k will be lists of integers with the same length

Generation $\leftarrow$ [ ]

new_token, Cache $\leftarrow$ LLM-forward-PromptDistill(M, X, r, k, (optional)tt)

Generation$_0 \leftarrow$ new_token

for $t \leftarrow 1 ... T-1$ **do**

  new_token, Cache $\leftarrow$ LLM-forward(M, new_token, Cache)

  Generation$_t \leftarrow$ new_token

end for

return Generation

D More Algorithms

Three of the algorithms (Algorithm 2, Algorithm 3, Algorithm 4) described in Section 2.2 are presented here.

E More Theoretical Analysis

The efficiency of the basic design without cache truncation is also shown in Table 1 for comparison.

Method	NrQA	qasp	MF-en	hotpqa	2wkqa	musq	gov_rep	qmsum	m_news	trec	triqa	sam	PCount	PRe	Mean
PrDist-1024 (no-trun)	23.1	10.87	26.61	16.12	15.47	11.67	29.88	22.86	25.93	73.5	91.57	43.62	5.41	95.97	35.18
PrDist-1024	24.24	11.36	28.01	16.59	14.92	11.7	29.68	22.29	25.77	73.5	91.65	43.61	5.87	95.8	35.36
PrDist-2048 (no-trun)	27.62	12.61	26.87	16.51	16.39	12.57	31.99	23.39	26.57	73.5	91.25	43.92	5.26	97.67	36.15
PrDist-2048	28.24	12.86	26.75	17.47	15.88	12.46	32.06	23.0	26.86	73.5	91.54	43.22	5.62	97.79	36.23
PrDist-4096 (no-trun)	28.63	13.2	26.98	16.23	16.35	11.45	33.67	23.41	26.82	72.5	91.31	44.11	5.02	97.82	36.25
PrDist-4096	30.2	13.24	26.91	16.41	16.12	11.6	34.2	23.08	26.86	72.5	91.47	44.17	5.24	97.22	36.37

Table 6: Detailed Results for effect of cache truncation with Longbench and Llama 3.1 8B

Algorithm 3 LLM-forward-PromptDistill (single-stage selection, without cache truncation)

Input: LLM $M_{0:R}$, embedded input prompt X, selection layer r, number of selected tokens k
$\#$ M has $R+1$ layers in total
$\#$ X has token length = n
Cache $\leftarrow$ [ ]
for $i \leftarrow 0...R$ **do**
  $X, Q_n^{(i)}, K^{(i)}, V^{(i)} \leftarrow M_i(X)$
  Cache$_i \leftarrow K^{(i)}, V^{(i)}$
  if i equals r **then**
    indices $\leftarrow$ Select$(Q_n^{(i)}, K^{(i)}, k)$
    $X \leftarrow X_{\text{indices}}$
  end if
end for
return generate(X), Cache

Algorithm 4 LLM-forward-PromptDistill (multi-stage selection, with cache truncation)

Input: LLM $M_{0:R}$, embedded input prompt X, selection layers r, numbers of selected tokens k, truncation times tt
$\#$ r, k are lists with indices $0:m-1$
$\#$ n always refer to the end index of X
Cache $\leftarrow$ [ ]
for $i \leftarrow 0...R$ **do**
  $X, Q_n^{(i)}, K^{(i)}, V^{(i)} \leftarrow M_i(X)$
  Cache$_i \leftarrow K^{(i)}, V^{(i)}$
  if i equals r_p for any p **then**
    indices $\leftarrow$ Select$(Q_n^{(i)}, K^{(i)}, k_p)$
    $X \leftarrow X_{\text{indices}}$
    if $p < tt$ **then**
      for $j \leftarrow 0...i$ **do**
        $K^{(j)}, V^{(j)} \leftarrow$Cache$_j$
        $K^{(j)} \leftarrow K^{(j)}_{\text{indices}}$
        $V^{(j)} \leftarrow V^{(j)}_{\text{indices}}$
        Cache$_j \leftarrow K^{(j)}, V^{(j)}$
      end for
    end if
  end if
end for
return generate(X), Cache

In stage one, the time cost is the same as our proposed method since cache truncation occurs after selection. In stage two, while the basic design also requires $\Theta((m-r)hk^2d)$ for processing selected tokens, it differs in key-value cache handling. Since all prompt tokens' keys and values are kept in the first r layers, the generation step takes $\Theta(rh((n+t)t)d + (m-r)h((k+t)t)d) = \Theta(rh(nt+t^2)d + (m-r)h(kt+t^2)d) = \Theta(rh((n-k)t)d + mh(kt+t^2)d)$. The total time cost then becomes $\Theta((m-r)hk^2d + rh((n-k)t)d + mh(kt+t^2)d) = \Theta(mh(k^2+t^2)d + rh((n-k)t)d - rhk^2d) = \Theta(mh(k^2+t^2)d + rh(nt-kt-k^2)d)$. For GPU memory in stage two, since the cache in the first r layers is not truncated, memory usage is $mw + 2rh(n+t)d + 2(m-r)h(k+t)d = mw + 2mh(k+t)d + 2rh(n-k)d$. This confirms that cache truncation is essential for efficiency.

For multi-stage selection, different configurations are possible. As an example, consider selecting k' tokens at layer r' and then k tokens at layer r with $r > r', k' > k$, finally truncating the cache to k tokens. The time cost in stage one is $\Theta(r'hn^2d + (r-r')hk'^2d)$. Since a first truncation occurs after the initial selection, GPU memory usage in stage one is $max\{r'w + 2r'hnd, rw + 2rhk'd\}$. After stage one, the cache size is the same as in single-stage selection, so stage two has identical time and memory costs. This shows that multi-stage selection can substantially improve efficiency with well-chosen selection layers and token counts.

F Empirical Latency Data

In addition to the theoretical efficiency analysis presented in Section 2.3 and Appendix Section E, we report empirical latency measurements in Table 7. Each entry records the total evaluation time of a method on the LongBench benchmark using the Qwen2-7B model.

This work prioritizes method design rather than

Method	Latency (seconds)
All KV	18985.98
H2O-4096	18064.70
Snap-1024	11023.39
Gem-1024	8544.96
PrDist-1024	7958.63
Snap-2048	19243.87
Gem-2048	12152.26
PrDist-2048	10469.19
Snap-4096	20093.65
Gem-4096	18106.55
PrDist-4096	16109.35

Table 7: Total Evaluation Latency with LongBench and Qwen2 7B

system-level implementation optimization. Our PromptDistill implementation produces the intended outputs, but is not explicitly optimized for peak runtime performance; similarly, we do not assume that all baseline implementations are fully optimized. Moreover, GPU-related factors such as machine switching and varying system load can introduce some variance in wall-clock latency measurements.

Despite these factors, the empirical latency results exhibit consistent trends that align well with our theoretical analysis. In particular, PromptDistill demonstrates clear time efficiency advantages over comparable baselines across different configurations. Together with the empirical performance results reported in the main paper, these measurements provide a complementary validation of the time efficiency benefits of the proposed method.

G Multi-stage selection performance

Here in Section G, we present the complete table of Multi-stage Selection performance.

H Layer choice experiments

For the Qwen2-7B Instruct model, which consists of 28 layers, we need to select a layer where GemFilter performs well - demonstrating a strong ability to identify important tokens — while also ensuring efficiency by avoiding layers that are too late in the sequence. As shown in Figure 2, GemFilter achieves a sudden performance boost at layer 16, reaching a score of 34.02, substantially higher than neighboring layers. Although higher scores are observed at layer 21 (35.15) and layer 23 (34.87), these layers are too late in the model, leading to reduced efficiency. Since layer 16 achieves a performance close to the best-performing layers while maintaining better efficiency, we select it as the final selection layer.

I Figures for Needle in a Haystack

Here, we present figures (Figure 3, Figure 4, Figure 5, Figure 6, Figure 7) illustrating the detailed Needle in a Haystack performance, corresponding to the scores shown in Section 3.2.2. Each graph corresponds to a base model with a specific compression method setting. The grids in the graphs display scores for different context sizes, depth percentages (needle locations within the context).

Selection token nums	Selection stages	Mean Score	Delta from PromptDistill-13-1024	Delta from GemFilter-13-1024
1024	13	35.36	0	0.86
4096 1024	10 13	35.00	-0.36	0.5
8192 1024	8 13	35.39	0.03	0.89
8192 1024	5 13	34.78	-0.58	0.28
8192 1024	4 13	34.67	-0.69	0.17
8192 4096 1024	8 10 13	35.03	-0.33	0.53
16384 8192 1024	7 8 13	35.42	0.06	0.92
16384 8192 1024	6 8 13	35.38	0.02	0.88
16384 8192 1024	5 8 13	35.53	0.17	1.03
16384 8192 1024	4 8 13	35.32	-0.04	0.82
16384 8192 1024	3 8 13	35.42	0.06	0.92
16384 8192 1024	2 8 13	35.29	-0.07	0.79
16384 8192 1024	1 8 13	35.32	0.04	0.82
16384 8192 1024	0 8 13	35.30	-0.06	0.8
16384 8192 1024	1 5 13	34.94	-0.42	0.44
16384 8192 1024	0 5 13	34.81	-0.55	0.31
16384 8192 1024	3 4 13	34.81	-0.55	0.31
16384 8192 1024	2 4 13	34.67	-0.69	0.17
16384 8192 1024	1 4 13	34.64	-0.72	0.14
16384 8192 1024	0 4 13	34.57	-0.79	0.07
16384 8192 4096 1024	5 8 10 13	35.00	-0.36	0.5
16384 8192 4096 1024	4 8 10 13	34.95	-0.41	0.45
16384 8192 4096 1024	3 8 10 13	34.92	-0.44	0.42
16384 8192 4096 1024	2 8 10 13	34.85	-0.51	0.35
16384 8192 4096 1024	1 8 10 13	34.90	-0.46	0.4
16384 8192 4096 1024	0 8 10 13	34.80	-0.56	0.3

Table 8: Multi-stage selection performance on LongBench (LLama 3.1 8B Instruct)

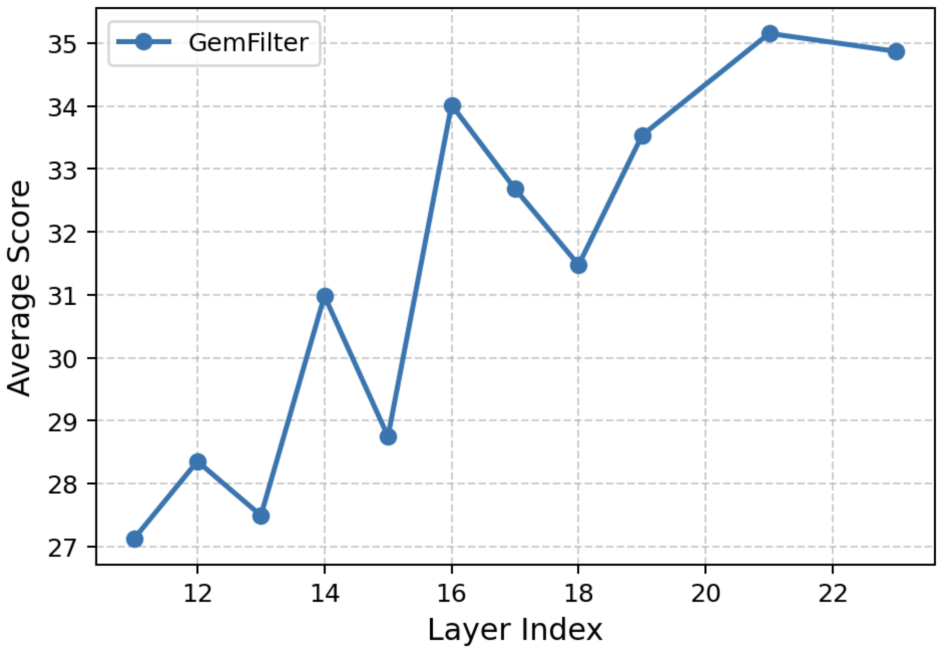
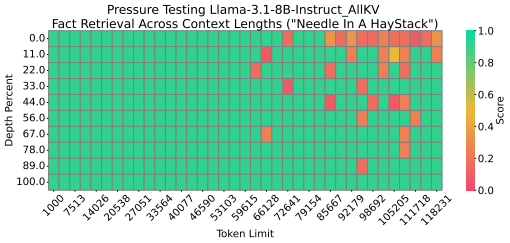
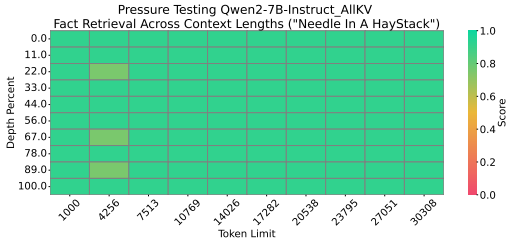


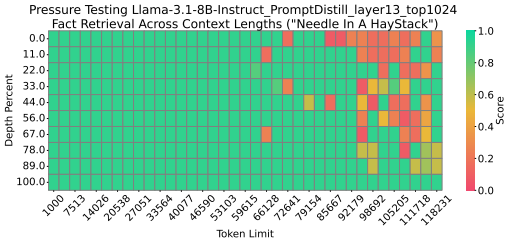
Figure 2: Performance of GemFilter + Qwen2 7B Instruct on LongBench with different selection layers.



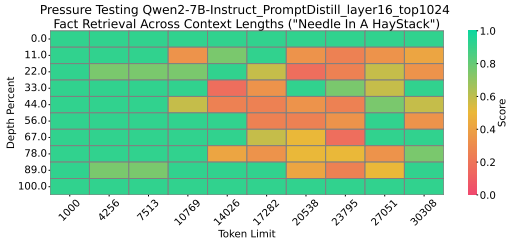
(a) LLama 3.1 8B Instruct + All KV: Score = 85.2



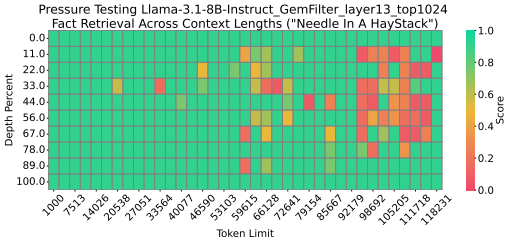
(b) Qwen2 7B Instruct + All KV: Score = 91.2



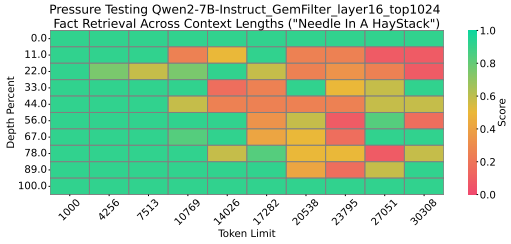
(c) LLama 3.1 8B Instruct + PromptDistill-1024: Score = 82.6



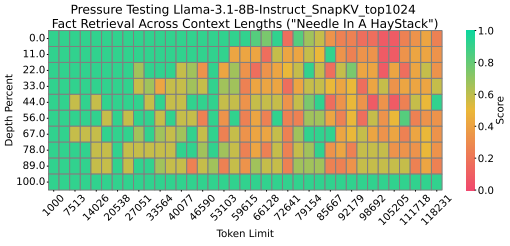
(d) Qwen2 7B Instruct + PromptDistill-1024: Score = 71.6



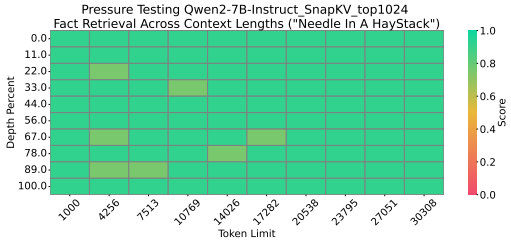
(e) LLama 3.1 8B Instruct + GemFilter-1024: Score = 81.8



(f) Qwen2 7B Instruct + GemFilter-1024: Score = 69.7

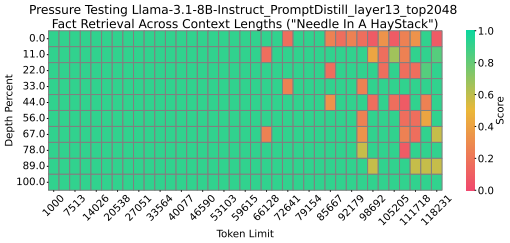


(g) LLama 3.1 8B Instruct + SnapKV-1024: Score = 64.0

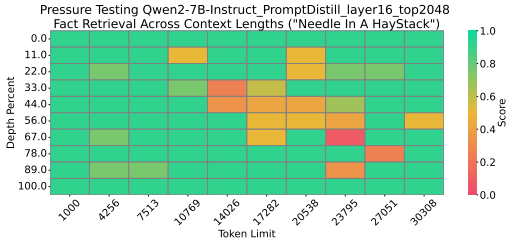


(h) Qwen2 7B Instruct + SnapKV-1024: Score = 90.5

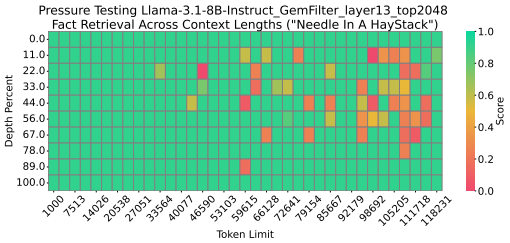
Figure 3: Evaluation Results for Needle in a Haystack: LLaMA 3.1 8B Instruct's and Qwen2 7B Instruct's scores in all test cases with model settings AllKV, PromptDistill-1024, GemFilter-1024, SnapKV-1024. (The x-axis represents the length of the input tokens, while the y-axis shows the position depth percentage of the 'needle' information)



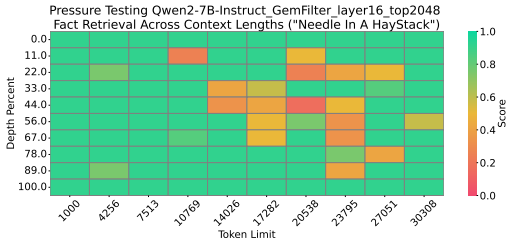
(a) LLama 3.1 8B Instruct + PromptDistill-2048: Score = 84.3



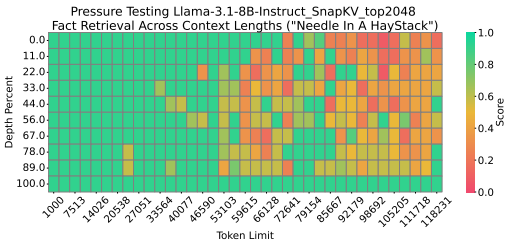
(b) Qwen2 7B Instruct + PromptDistill-2048: Score = 82.2



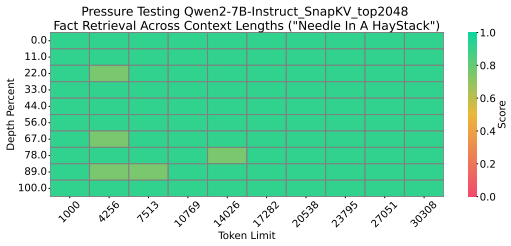
(c) LLama 3.1 8B Instruct + GemFilter-2048: Score = 84.7



(d) Qwen2 7B Instruct + GemFilter-2048: Score = 81.8

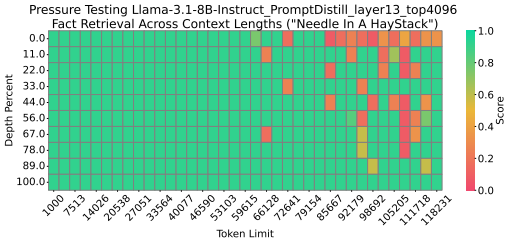


(e) LLama 3.1 8B Instruct + SnapKV-2048: Score = 71.7

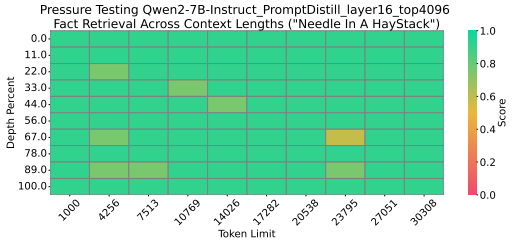


(f) Qwen2 7B Instruct + SnapKV-2048: Score = 90.8

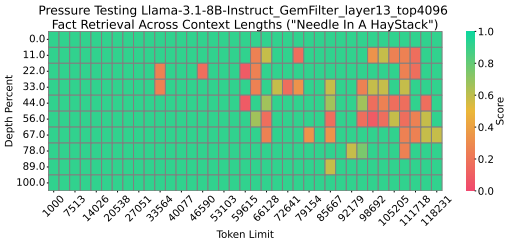
Figure 4: Evaluation Results for Needle in a Haystack: LLaMA 3.1 8B Instruct's and Qwen2 7B Instruct's scores in all test cases with model settings PromptDistill-2048, GemFilter-2048, SnapKV-2048. (The x-axis represents the length of the input tokens, while the y-axis shows the position depth percentage of the 'needle' information)



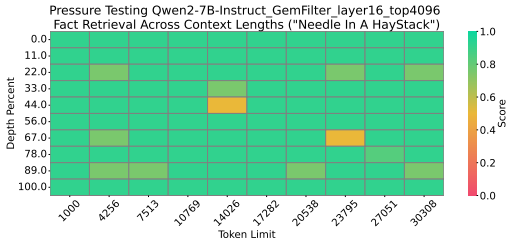
(a) LLama 3.1 8B Instruct + PromptDistill-4096: Score = 84.6



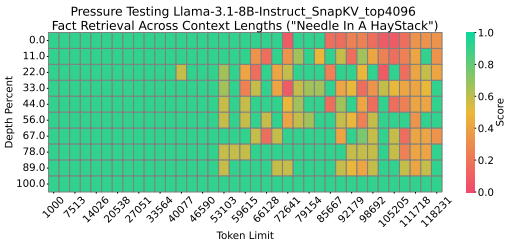
(b) Qwen2 7B Instruct + PromptDistill-4096: Score = 90.2



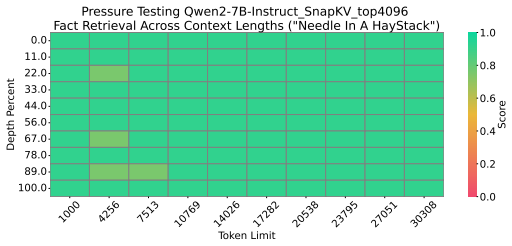
(c) LLama 3.1 8B Instruct + GemFilter-4096: Score = 83.3



(d) Qwen2 7B Instruct + GemFilter-4096: Score = 89.3

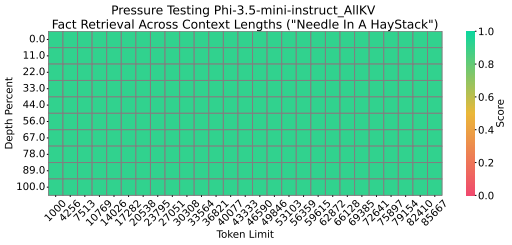


(e) LLama 3.1 8B Instruct + SnapKV-4096: Score = 77.1

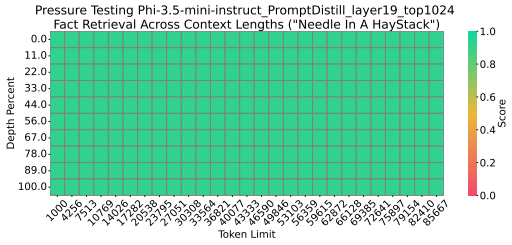


(f) Qwen2 7B Instruct + SnapKV-4096: Score = 91.0

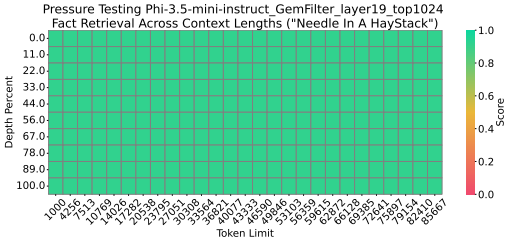
Figure 5: Evaluation Results for Needle in a Haystack: LLaMA 3.1 8B Instruct's and Qwen2 7B Instruct's scores in all test cases with model settings PromptDistill-4096, GemFilter-4096, SnapKV-4096. (The x-axis represents the length of the input tokens, while the y-axis shows the position depth percentage of the 'needle' information)



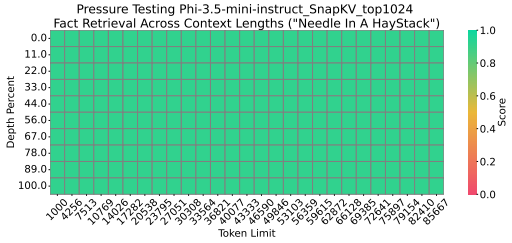
(a) Phi 3.5 Mini 3.8B Instruct + All KV: Score = 91.7



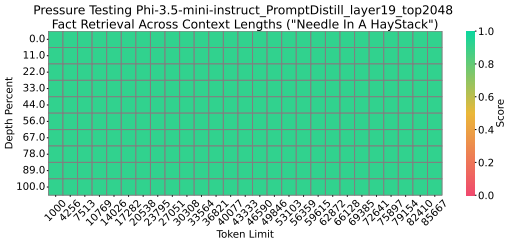
(b) Phi 3.5 Mini 3.8B Instruct + PromptDistill-1024: Score = 91.7



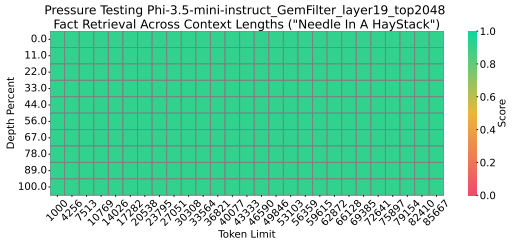
(c) Phi 3.5 Mini 3.8B Instruct + GemFilter-1024: Score = 91.7



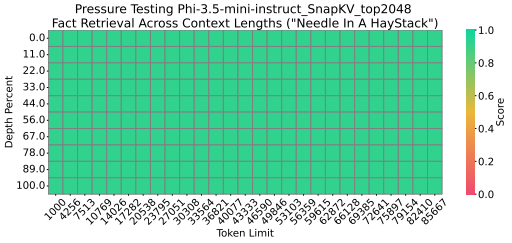
(d) Phi 3.5 Mini 3.8B Instruct + SnapKV-1024: Score = 91.7



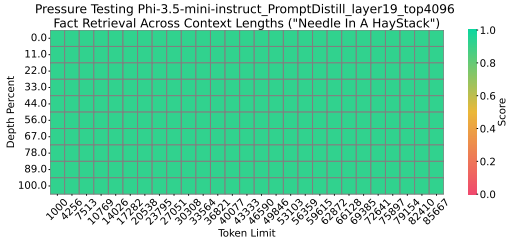
(e) Phi 3.5 Mini 3.8B Instruct + PromptDistill-2048: Score = 91.7



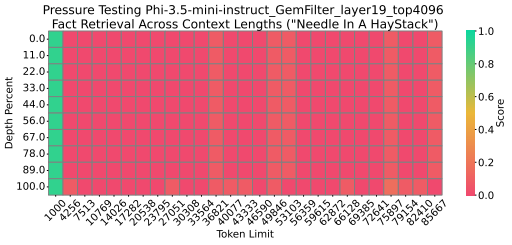
(f) Phi 3.5 Mini 3.8B Instruct + GemFilter-2048: Score = 91.7



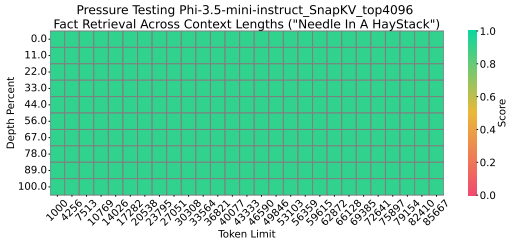
(g) Phi 3.5 Mini 3.8B Instruct + SnapKV-2048: Score = 91.7



(h) Phi 3.5 Mini 3.8B Instruct + PromptDistill-4096: Score = 91.7

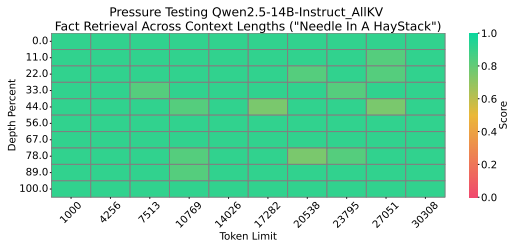


(i) Phi 3.5 Mini 3.8B Instruct + GemFilter-4096: Score = 5.1

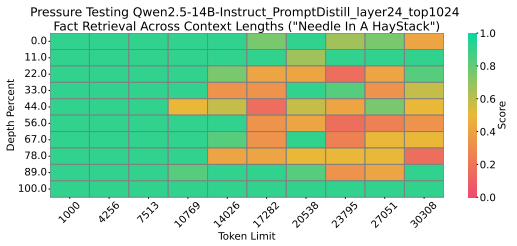


(j) Phi 3.5 Mini 3.8B Instruct + SnapKV-4096: Score = 91.7

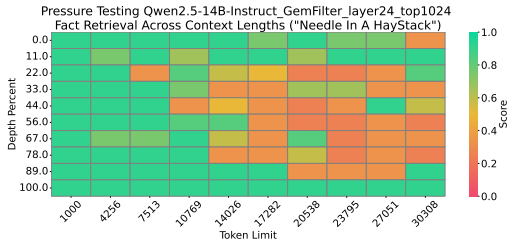
Figure 6: Evaluation Results for Needle in a Haystack: Phi 3.5 Mini 3.8B Instruct's scores in all test cases with all model settings. (The x-axis represents the length of the input tokens, while the y-axis shows the position depth percentage of the 'needle' information)



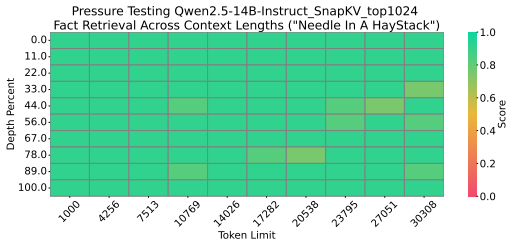
(a) Qwen2.5 14B Instruct + All KV: Score = 90.4



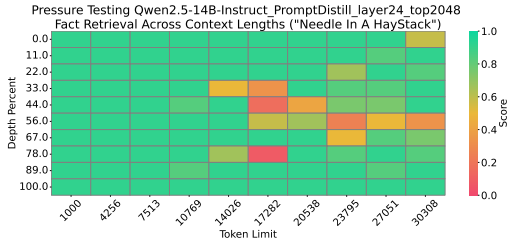
(b) Qwen2.5 14B Instruct + PromptDistill-1024: Score = 73.3



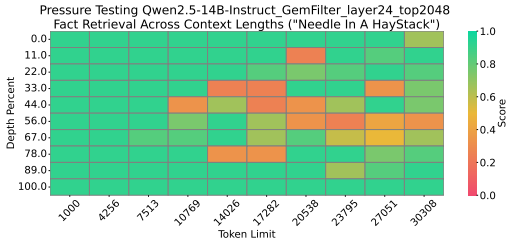
(c) Qwen2.5 14B Instruct + GemFilter-1024: Score = 68.7



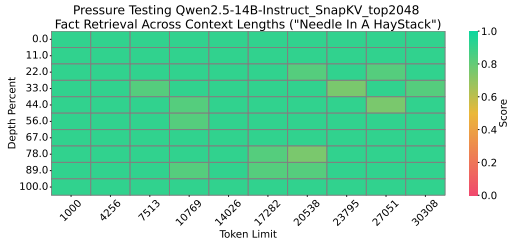
(d) Qwen2.5 14B Instruct + SnapKV-1024: Score = 90.6



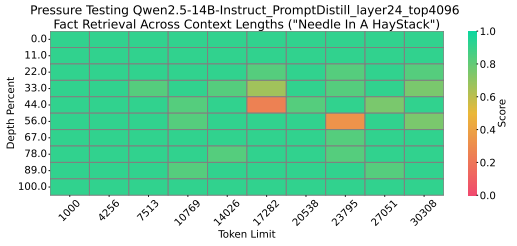
(e) Qwen2.5 14B Instruct + PromptDistill-2048: Score = 83.8



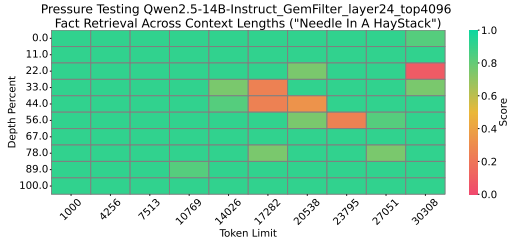
(f) Qwen2.5 14B Instruct + GemFilter-2048: Score = 79.7



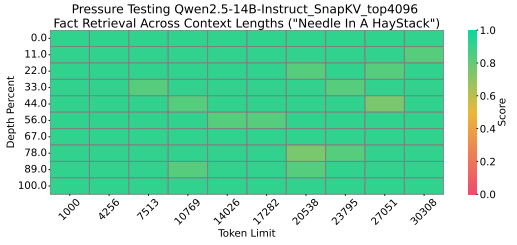
(g) Qwen2.5 14B Instruct + SnapKV-2048: Score = 90.3



(h) Qwen2.5 14B Instruct + PromptDistill-4096: Score = 88.5



(i) Qwen2.5 14B Instruct + GemFilter-4096: Score = 86.9



(j) Qwen2.5 14B Instruct + SnapKV-4096: Score = 90.4

Figure 7: Evaluation Results for Needle in a Haystack: Qwen2.5 14B Instruct's scores in all test cases with all model settings. (The x-axis represents the length of the input tokens, while the y-axis shows the position depth percentage of the 'needle' information)

Responsible NLP Research Checklist

A For every submission

A1 Did you discuss the *limitations* of your work?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

Yes; After section 4(Conclusion)

A2 Did you discuss any potential *risks* of your work?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

No; Our work doesn't have potential risk.

B Did you use or create *scientific artifacts*?

If you answer **Yes**, provide the section number; if you answer **No**, you can skip the rest of this section.

Yes

If yes:

B1 Did you cite the creators of artifacts you used?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

Yes; Section 3 Experiments

B2 Did you discuss the *license or terms* for use and/or distribution of any artifacts?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

No; The models and datasets we use are all open source and we will discuss the license if necessary after paper acceptance.

B3 Did you discuss if your use of existing artifact(s) was consistent with their *intended use*, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

Yes; Section 3 Experiments

B4 Did you discuss the steps taken to check whether the data that was collected/used contains any *information that names or uniquely identifies individual people* or *offensive content*, and the steps taken to protect / anonymize it?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

No; we don't have this kind of data.

B5 Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

No; We have provided comprehensive information about method and experiments and there's no need for specialized documentation.

B6 Did you report relevant statistics like the number of examples, details of train/test/dev splits, etc. for the data that you used/created?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

Yes; Section 3 Experiments

C Did you run *computational experiments*?

If you answer **Yes**, provide the section number; if you answer **No**, you can skip the rest of this section.

Yes

If yes:

C1 Did you report the *number of parameters* in the models used, the *total computational budget* (e.g., GPU hours), and *computing infrastructure* used?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

Yes; Section 3 Experiments

C2 Did you discuss the experimental setup, including *hyperparameter search* and *best-found hyperparameter* values?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

Yes; Section 3 Experiments

C3 Did you report *descriptive statistics* about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

Yes; Section 3 Experiments

C4 If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

Yes; Section 3 Experiments

D Did you use *human annotators* (e.g., crowdworkers) or *research with human subjects*?

If you answer **Yes**, provide the section number; if you answer **No**, you can skip the rest of this section.

No

If yes:

D1 Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

N/A

D2 Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such *payment is adequate* given the participants' demographic (e.g., country of residence)?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

N/A

D3 Did you discuss whether and how *consent* was obtained from people whose data you're using/curating (e.g., did your instructions explain how the data would be used)?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

N/A

D4 Was the data collection protocol *approved (or determined exempt)* by an ethics review board?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

N/A

D5 Did you report the basic demographic and geographic characteristics of the *annotator* population that is the source of the data?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

N/A

E Did you use AI assistants (e.g., ChatGPT, Copilot) in your research, coding, or writing?

If you answer **Yes**, provide the section number; if you answer **No**, you can skip the rest of this section.

Yes

If yes:

E1 If you used any AI assistants, did you include information about your use?

If you answer **Yes**, provide the section number; if you answer **No**, provide a justification.

No; AI is used for debugging but not coding