One Task Vector is not Enough: A Large-Scale Study for In-Context Learning

Anonymous ACL submission

Abstract

In-context learning (ICL) enables Large Language Models (LLMs) to adapt to new tasks using few examples, with task vectors—specific hidden state activations—hypothesized to encode task information. Existing studies are limited by small-scale benchmarks, restricting comprehensive analysis. We introduce QUITEAFEW, a novel dataset of 3,096 diverse few-shot tasks, each with 30 input-output pairs derived from the Alpaca dataset. Experiments with Llama-3-8B on QUITEAFEW reveal: (1) task vector performance peaks at an intermediate layer (e.g., 15th), (2) effectiveness varies significantly by task type, and (3) complex tasks rely on multiple, subtask-specific vectors rather than a single vector, suggesting distributed task knowledge representation.

1 Introduction

Transformer-based Large Language Models (LLMs) (Vaswani et al., 2017) excel at in-context learning (ICL), adapting to new tasks via a few prompt-based examples without weight updates (Brown et al., 2020) and have shown impressive empirical results (Liu et al., 2023; Dong et al., 2022). This capability enables rapid task adaptation; however, how LLMs internally represent and apply task information remains unclear. Recent work points to "task vectors" (Hendel et al., 2023) or "function vectors" (Todd et al., 2024) – specific hidden state activations – as the mechanism for encoding task rules.

Prior studies, such as Hendel et al. (2023), suggest that ICL compresses demonstration sets into task vectors that guide query processing. Todd et al. (2024) used causal analysis to locate these vectors, showing they capture semantic task aspects. While techniques like sparse autoencoders (SAEs) have begun to shed light on the interpretable features within a given task vector (Kharlapenko et al., 2024), the fundamental question of whether a single such vector suffices for complex, multi-faceted tasks remains largely unexplored. Luo et al. (2024) extended this to vision-language models, demonstrating that task vectors are cross-modal, clustering by task rather than input modality (e.g., text or image) and emerging at intermediate layers to summarize tasks before generation. On some tasks, task vectors achieve near-excellent performance, often over 90% accuracy. However, current stud-

Work	Task Categories & Examples
Hendel et al. (2023) (11 tasks)	*Algorithmic*: Next letter, List first, List last, To uppercase *Translation*: Fr → En, Es → En *Linguistic*: Present → Gerund, Singular → Plural *Knowledge*: Country → Capital, Person → Language
Kharlapenko et al. (2024) (9 tasks)	*Linguistic*: Antonyms, Present Tense → Past Tense *Translation*: En → Es, En → Fr, Es → En *Knowledge*: Country → Capital, Person → Profession, Location → Language, Location → Religion
Luo et al. (2024) (6 tasks)	*Knowledge*: Country → Capital, Country → Currency, Animal → Latin, Animal → Young, Food → Color, Food → Flavor
Todd et al. (2024) (Over 40 tasks)	*Linguistic (e.g.)*: Antonyms, Present → Past, Singular → Plural *Knowledge (e.g.)*: Country → Capital *Translation (e.g.)*: English → French *Text Manipulation (e.g.)*: Capitalize
QUITEAFEW (Ours) (3,096 tasks)	Split into categories by the first word of the task: *Given*: 294 tasks *Generate*: 193 tasks *Rewrite*: 178 tasks *Create*: 159 tasks *Classify*: 125 tasks *Identify*: 110 tasks *Write*: 107 tasks *Find*: 99 tasks *Other*: 1,657 tasks

Table 1: Task Dataset Comparison on Task Vectors investigation.

Instruction	Example Input	Category	Explanation
Answer this question with a yes or no.	Will I be able to go to the park tomorrow?	INVALID	Requires future knowledge or personal context that an AI cannot predict
Find a good restaurant near the given address	660 Lexington Avenue, New York, NY 10022	INVALID	Needs real-world data; "good restaurant" is subjective
What is the largest city on this continent?	Africa	LIMITED	Limited size of a category, insufficient for 30+ diverse examples

Table 2: Examples of Alpaca entries filtered out due to being unsuitable for few-shot generation.

ies mainly utilize toy, manually crafted datasets (see Tab. 1), which limits our understanding of task vector dynamics in diverse, large-scale settings.

To address this gap, we introduce QUITEAFEW, a novel dataset comprising 3,096 diverse few-shot learning tasks, each with 30 unique input-output pairs derived from the Alpaca dataset (Taori et al., 2023). This dataset spans a broad spectrum of tasks, from algorithmic operations to open-ended generative challenges, enabling a comprehensive exploration of in-context learning (ICL). Through experiments with (Grattafiori et al., 2024) on QUITEAFEW, we uncover key insights into task vector dynamics. Task vector performance consistently peaks at a specific (such as the 15th on Llama-3-8B) intermediate layer, across diverse task categories like algorithmic processing and text rewriting. However, the effectiveness of single task vectors varies significantly depending on the task type, with some categories demonstrating robust results while others experience notable declines. Our analysis reveals that instead of relying on a single task vector, models utilize multiple subtask-specific vectors, indicating a more distributed task representation within the model.

2 Methodology

2.1 Introduction to Task Vectors

Formally, a task vector is the hidden state at a designated layer for a specific token in the few-shot prompt, often the separator token (e.g., `->`) marking the transition from input to output. For a prompt with k input-output pairs (e.g., `big -> small`), the task vector v_l is extracted as the hidden state at layer l after processing the final token `->`.

To apply a task vector, we employ a causal intervention during zero-shot inference. For a new input (e.g., `hot ->`), the model processes the input up to the token `->`, at which point the hidden state at layer l is replaced with v_l. The model then generates the output autoregressively, using this modified hidden state as part of its standard computation.

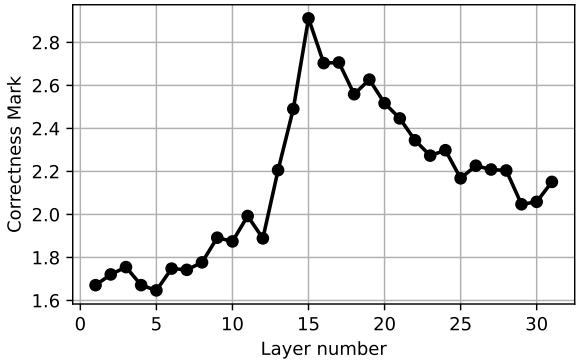

Figure 1: Average task vectors performance on QUITEAFEW dataset.

2.2 Dataset Collection

We built QUITEAFEW by expanding the Alpaca dataset (Taori et al., 2023), which contains instruction-following entries from OpenAI's text-davinci_003. Many Alpaca entries include an `instruction` paired with an example `input` and `output`, providing a structure ideal for generating diverse few-shot learning tasks. The `instruction` (e.g., "Rewrite the given sentence to incorporate a hyperbole") specifies the task, while the example `input` (e.g., "The house was very old.") and `output` (e.g., "The house was older than the hills.") demonstrate the expected transformation, enabling the creation of varied examples (e.g., "The water was very cold." → "The water was colder than the depths of Antarctica."). The instructions cover a wide range of tasks, and could be categorized by their initial verb (e.g., "generate", "rewrite", "classify"), as shown in Tab. 1.

However, not all Alpaca entries were suitable for few-shot task generation. Some contained errors (e.g., incorrect calculations or factual inaccuracies), while others were too restrictive (e.g., tasks with limited input diversity). Examples of such problematic entries are listed in Tab. 2. To ensure quality, we applied a filtering process to select instructions

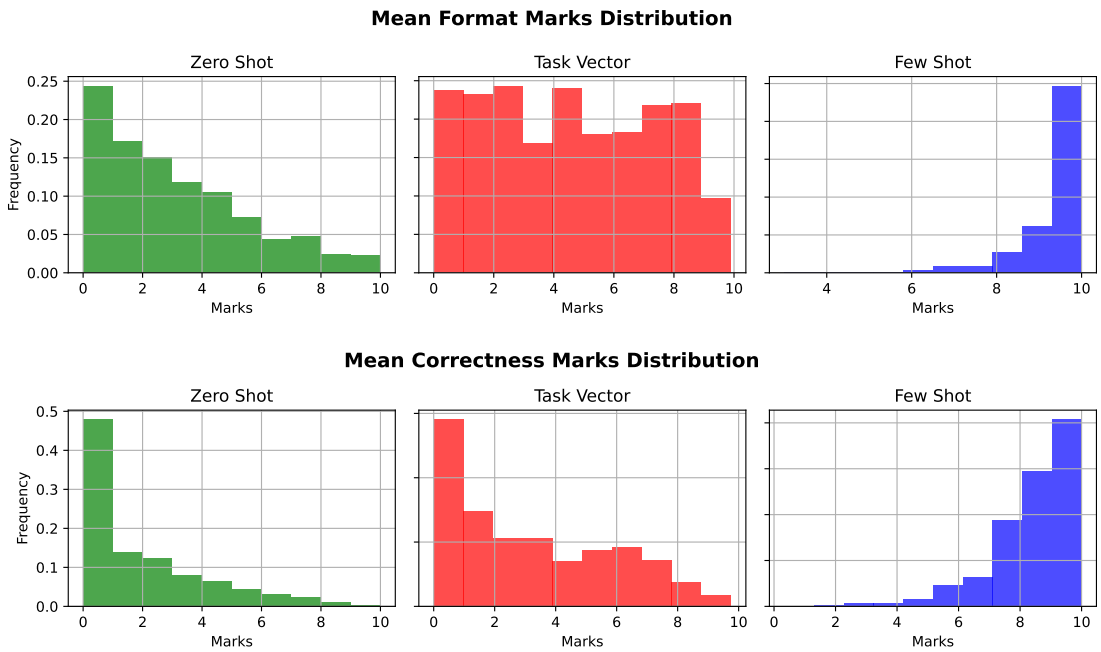

Figure 2: Marks distribution on QUITEAFEW dataset.

appropriate for creating diverse, high-quality few-shot examples.

We used Qwen-2.5-72B with a tailored classification prompt (see Appendix A) to evaluate each Alpaca entry's suitability. The prompt assessed whether an `instruction` and its example `input` could support generating at least 30 distinct input-output pairs. The evaluation criteria were:

- The instruction must allow for ≥ 30 meaningfully different inputs.
- The output's correctness for a given input must be clearly verifiable.

Instructions were classified as:

- **GOOD**: Capable of yielding 30+ diverse input-output pairs.
- **LIMITED**: Unsuitable due to insufficient input variety (< 30).
- **INVALID**: Unsuitable due to reliance on external knowledge, impossibility, or single-output constraints.

This process identified 3,096 **GOOD** instructions for inclusion in QUITEAFEW.

For each **GOOD** instruction, we generated 30-50 new input-output pairs using Qwen-2.5-72B and Qwen-3-235B-A22B (Yang et al., 2025) with a dedicated prompt (see Appendix B). Specifically, Qwen-3-235B-A22B generated 2,072 tasks, and Qwen-2.5-72B generated 1,024 tasks. The prompt instructed the model to analyze the original `instruction`, `example_input`, and `example_output` and produce 30 diverse inputs while maintaining the output format and style. The original example served as a template to ensure consistency.

The resulting QUITEAFEW dataset comprises 3,096 tasks, each with an original instruction and 30 unique input-output pairs. This structure supports robust few-shot ICL prompts and enables comprehensive analysis of task vector dynamics across diverse task types.

2.3 Task Performance Analysis

Prior work evaluating task vectors used tasks with clear, verifiable answers (e.g., antonym generation), enabling simple accuracy metrics. In contrast, QUITEAFEW includes diverse tasks, many lacking a single correct output, e.g., rewriting text in a specific style. To uniformly evaluate all tasks, we used an LLM-based judging approach (see Appendix C), scoring responses on *format* (0–10, adherence to expected output type) and *correctness* (0–10, accuracy or appropriateness).

For each task, we randomly select 8 examples. Of these, 7 are used to construct the few-shot prompt, while the remaining one serves as the test example for zero-shot evaluation with the task vector. This process is repeated 10 times and the results are then averaged.

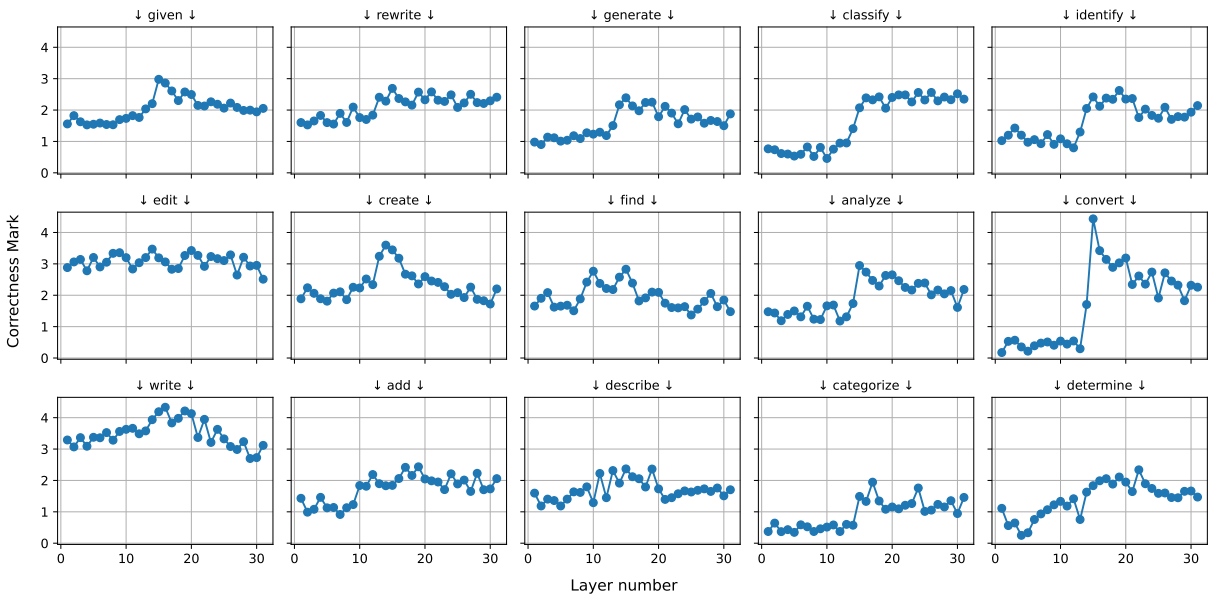

Figure 3: Layer-wise Task Vector Performance across different task categories.

We now turn to evaluating how effectively task vectors encode and apply task-specific information across diverse task types. The following experiments investigate task vector performance, layer-wise dynamics, and their limitations in handling complex, multi-faceted tasks.

3 Experiments and Evaluation

3,096t learning, we conduct a series of experiments aiming to: (1) assess the layer-wise performance of task vectors across diverse task categories (Sec. 3.1), (2) evaluate their effectiveness compared to zero-shot and full few-shot baselines (Sec. 3.2).

Unless otherwise specified, all experiments utilize the Llama-3-8B model. First, we analyzed task vector performance across all layers of Llama-3-8B using a randomly selected subset of 1,000 tasks from QUITEAFEW. The experimental procedure for processing one such scenario was as follows:

3.1 Layer-wise Performance

1. For each layer l in the model, extract the final token's hidden state after processing the few-shot prompt, yielding a layer-specific task vector v_l.

2. Inject v_l into the model's hidden state at layer l at the end of the input for a new zero-shot example.

3. Generate an output for this new example with an injected task vector.

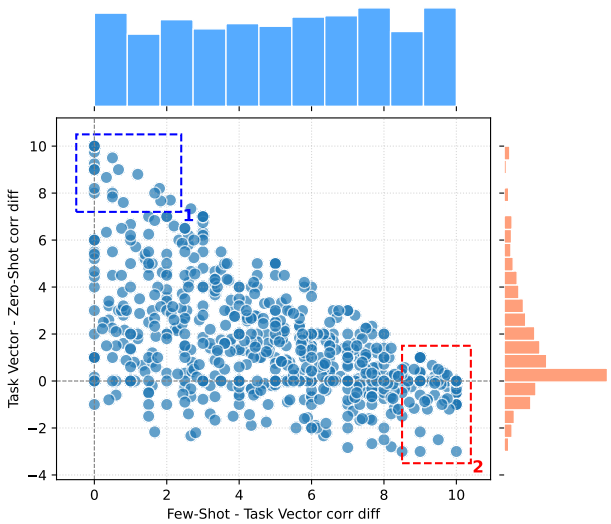

Figure 4: Dual-axis analysis of task vector effectiveness on QUITEAFEW, showing performance boost over zero-shot inference versus deficit relative to full few-shot learning.

4. Evaluate the output using the LLM-based judging prompts (Appendix C), obtaining format and correctness scores.

5. Average the scores across all evaluated tasks for each layer l and compute per-category averages for key task categories (e.g., Classify, Rewrite, Generate).

Our analysis, visualized in Fig. 3, reveals a consistent performance peak around the 15th layer across different task categories, that correlates with existing results, that intermediate layers are critical

Method	Output
Task: Translate the given document into French.	
Input: The novel explores the complexities of human relationships in a dystopian future.	
Zero-shot	The novel examines the fragility of human connections in a world where technology has replaced personal interactions.
Task Vector	Ce livre explore les complexités des relations humaines dans un futur dystopique.
Few-shot	Le roman explore les complexités des relations humaines dans un futur dystopique.
Task: Classify the following topic as either technology, business, travel, or sports.	
Input: World Cup football results	
Zero-shot	A list of the
Task Vector	Sports
Few-shot	Sports
Task: Re-write the given sentence in the imperative form.	
Input: It's crucial to save water.	
Zero-shot	Saving water is crucial.
Task Vector	Save water.
Few-shot	Save water.

Table 3: Examples of tasks from Region 1 (see Fig. 4), where task vectors perform well, showing significant improvement over zero-shot and approaching few-shot performance.

Method	Output
Task: Categorize the given items by type.	
Input: Banana, Pen, Shark	
Zero-shot	Banana, Shark, Pen
Task Vector	Banana, Pen, Shark Pen, Banana, Shark Pen, Shark, Banana
Few-shot	Banana: Fruit, Pen: Writing Instrument, Shark: Animal
Task: Analyze the given sentence and classify it into the parts of speech.	
Input: We love to travel in the summer.	
Zero-shot	We love to travel in the summer.
Task Vector	We love to travel in the summer.
Few-shot	We (pronoun) love (verb) to (adverb) travel (verb) in (preposition) the (article) summer (noun).

Table 4: Examples of tasks from Region 2 (see Fig. 4), where task vectors perform poorly, often no better than zero-shot, despite few-shot success.

for encoding task-specific information. Specifically, task categories such as *convert*, *classify*, *analyze*, etc. exhibit a significant correctness increase at layer 15. However, categories like *edit* and *describe* exhibit more or less the same behavior across all layers.

Since on the 1,000 tasks the most effective was the 15th layer, for all other experiments we stick to only the 15th layer.

The experiments highlight variability in how effectively the model performs in-context learning across the diverse tasks (Fig. 2). As a sanity check, we compared the performance of task vectors against full few-shot performance and a baseline where no task vector was provided in zero-shot settings.

The Format Score for task vectors consistently exceeded the Correctness Score. This might suggest that the model understood it needed to classify into specific classes (e.g., A, B, C, D) but couldn't recall what each option represented. Nevertheless, this indicates that the task vector contains some useful signal necessary for task execution, surpassing the baseline, though not as strong as full few-shot performance.

The other thing to notice, that task vectors do not always successfully handle tasks in terms of Correctness Score, which indicates that there is only a small subset of tasks where task vectors perform effectively out of the box. In the following sections we will investigate it more.

3.2 What tasks are best for task vectors?

Fig. 4 illustrates the effectiveness of task vectors by simultaneously measuring their performance boost compared to zero-shot inference and their deficit relative to full few-shot learning. This dual-axis analysis is crucial because strong raw performance

from a task vector does not solely indicate its efficacy; it might be an easy task where even zero-shot performs well.

Tab. 3 presents examples illustrative of tasks found in Region 1. In these instances, the application of a task vector leads to outputs that are significantly improved compared to zero-shot and closely match the quality of full few-shot prompting.

Conversely, Tab. 4 showcases examples representative of tasks from Region 2. For these tasks, the task vector provides little to no improvement over zero-shot inference, and in some cases, may even lead to a degradation in performance, despite full few-shot prompting demonstrating the task's solvability.

This motivates a deeper investigation into why task vectors might fail for certain types of tasks.

4 Analysis on Complex Tasks

While task vectors have demonstrated utility, our experiments reveal that their effectiveness is limited. But why do task vectors sometimes fail? There are at least two possible explanations: (1) all necessary information for task completion is present within the task vector but is obfuscated by noise, or (2) not all the critical task information is captured by the single task vector, instead this information residing elsewhere in the model's representations.

We hypothesize that many real-world tasks are inherently compound, comprising multiple subtasks. In such cases, a single task vector may not naturally emerge to represent the entire task. Instead, the model develops multiple task vectors—each corresponding to a specific subtask.

4.1 A Motivating Observation

To test this hypothesis, we constructed a synthetic dataset that emulates a realistic complex task: converting unstructured textual descriptions into structured schema representations. Specifically, we synthesized automobile descriptions and required the model to transform these descriptions into a predetermined JSON format specified through few-shot examples.

For instance, given an unstructured input description of a car, such as:

```
Performance enthusiast's dream: red Rolls-Royce Cullinan EWB (2006). Unleash 564 HP from the 6.75L Twin-Turbo V12, reaching 210 km/h. RWD, Cognac Nappa leather, panoramic roof, rear entertainment, premium audio. Located in Sydney, 3,580 km, VIN: WXZFZXBGE96XAUD55. Priced at $210,000.
```

The model was expected to produce a JSON object:

```
{"color":"red","city":"Sydney",
"model":"Cullinan"}
```

In this setup, the few-shot prompts consist of seven examples, each pairing a unique automobile description with its corresponding JSON representation. These JSON object are always consist of three attributes (each of which is a single token)— `color`, `city`, and `model` —always in that order. Here we distinct from the conventional task vector method where a single hidden state intervention occurs after the input description, typically at the `->` token. Instead, at each step of generating the zero-shot JSON output, we test the influence of injecting **each** of the hidden states from the output of the last few-shot example. Specifically, for the current token being generated, we make a series of experiments, each time substituting its layer 15 hidden state with **each** hidden state from **every token position** within the JSON output of the last few-shot example, and recording the resulting next-token prediction. This is also compared against natural generation without intervention.

To illustrate, Fig. 5 visualizes this process. Rows represent tokens in the output sequence being generated in the zero-shot setting, while columns represent tokens from the output of the last few-shot example. Each cell contains a binary value: **1** indicates that substituting the hidden state from the column token (from the few-shot example) at layer 15 during generation of the row token (in zero-shot) correctly predicts the next token; **0** indicates incorrect prediction. This visualization allows us to identify which specific hidden states from the few-shot example contribute to correct predictions for each token in the zero-shot generation.

The token `red` in zero-shot was restored when substituting the hidden state of token `":"` with the state of the corresponding token `":"` in few-shot, where it is followed by token `green`. The token `Sy` — the beginning of the word "Sydney" in zero-shot — was restored from token `":"`, followed by

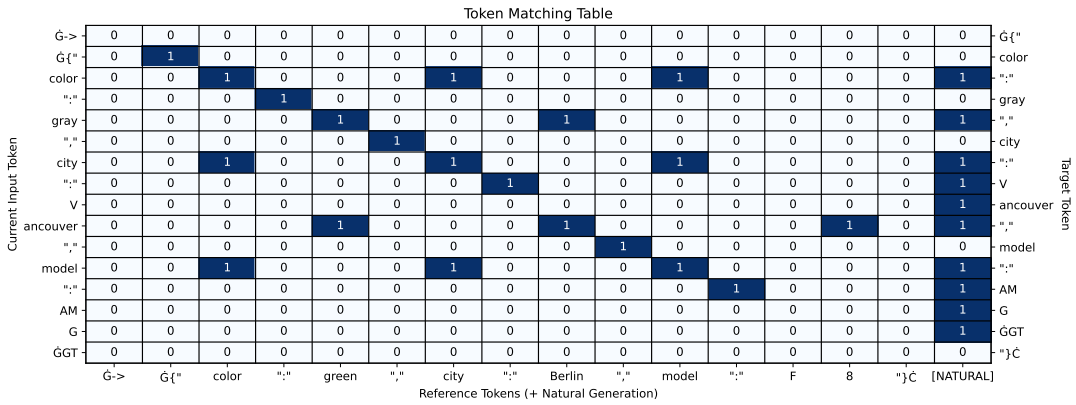

Figure 5: Token-level influence of few-shot hidden states on zero-shot JSON generation. Rows represent tokens being generated in the zero-shot output. Columns represent tokens from the last few-shot example's output, plus a [NATURAL] states for the natural continuation without intervention.

token `Berlin` in few-shot. And the token `C` — the beginning of the word "Culinan" in zero-shot — was restored from token `":"`, followed by token `FB` in few-shot.

The similar behavior holds and for attribute names: the first attribute `color` is correctly predicted by substituting the hidden state of the token `G{"`, followed by token `color` in few-shot. Same for the second attribute `city`, which was restored from the hidden state of the token `","`, preceding the token `city` from the few-shot example; and for the third: `model`, was restored using token `","`, preceding the token `model` from the few-shot example.

It is interesting that the first *"real"* task vector was not in the `G->`, but in the `{"`.

It is interesting that the first *"real"* task vector was not at the beginning of the output sequence (e.g., associated with the `G->` token), but rather emerged at the token `{"`, which directly precedes the first attribute name. This might suggest that the model activates task-specific representations closer to the point of use for each sub-component of the complex output.

4.2 Investigating Task Vector Specialization

We further investigate this phenomenon and quantify the potential insufficiency of a single, global task vector for such compositional tasks, we conducted a scaled experiment using a set of 100 (*few-shot, test sample*) pairs of such automobile descriptions. Our evaluation focused on the model's ability to predict specific tokens within the target JSON output. We aimed to measure how effectively different task vector injection strategies could guide the model in generating both **fixed structural elements** of the JSON, such as tokens for attribute keys (e.g., `color`, `city`), and **dynamic, context-dependent content**, namely the attribute values extracted from the input description (e.g., the tokens for color `green` or city `Berlin`). We assessed performance under three distinct conditions:

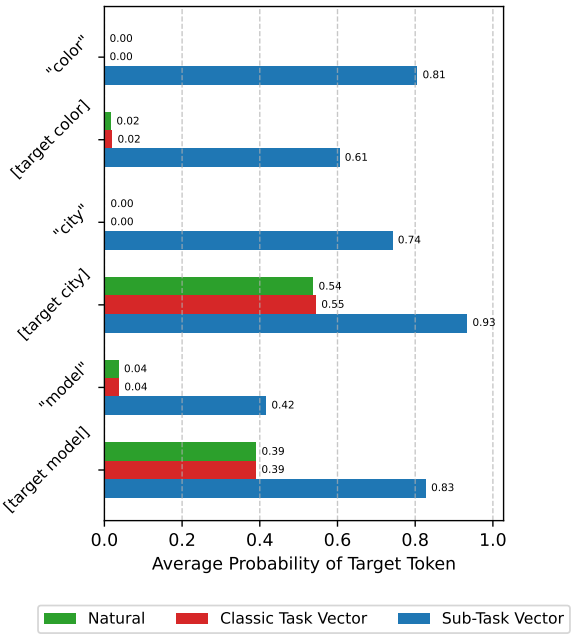

Figure 6: Comparison of token prediction probabilities for JSON generation across 100 automobile descriptions, evaluating Natural Generation, Classic Task Vector, and Sub-Task Vectors strategies.

1. **Natural Generation**: The model generates the output token by token without any task vector intervention, serving as a baseline.

2. **Classic Task Vector**: A single task vector, extracted from the hidden state at the position of the final `G->` token in the few-shot prompt's output, is injected at the position of the `G->` token in the zero-shot input. This evaluates the conventional single task vector approach.

3. **Sub-Task Vectors**: Here we just use the hidden states from the *corresponding* token from the few-shot as a sub-task vectors (it's i-th token `":"` for attribute values, and tokens `G{"` or `","` for attribute names).

The results, summarized in Fig. 6. In contrast, and consistent with our motivating observation, the use of sub-task vectors provides a **substantial** increase in the average probability for the correct target token. This improvement is evident both for predicting fixed attribute *keys* (e.g., token `color`, which are identical across examples) and for predicting the dynamic attribute *values* (e.g., the specific color term, which varies between the few-shot demonstration and the zero-shot query). For instance, predicting the token `color` after token `{"` sees its probability rise significantly with a sub-task vector, as does the prediction of the actual color value after tokens `color` `":"`.

It is noteworthy that for later attributes in the sequence, such as the values for `city` and `model`, the probabilities under Natural Generation are already considerably above random chance. This can be attributed to the model having already processed preceding parts of the JSON structure, thereby gaining contextual cues about the expected format and the current attribute being populated. However, even in these instances where the baseline is stronger, the application of an appropriate sub-task vector still markedly outperforms both Natural Generation and the Classic Task Vector approach. This further reinforces the idea that task execution for complex outputs relies on a sequence of more specialized, context-dependent activations rather than a single, overarching task representation.

*Thus, in this case we need to talk not about **one** task vector for the entire task, but about **many** task vectors for the task.*

5 Conclusion

To our knowledge, this is a first study to systematically evaluate task vectors on diverse set of NLP tasks.

We found out that optimal task vector performance consistently emerges around a specific intermediate model layer (e.g., the 15th layer of Llama-3-8B) across a wide variety of task types. Second, the overall effectiveness of these vectors varies substantially depending on the intrinsic nature of the task, with some task categories yielding strong performance while others show considerable degradation.

Further, our case analysis of composite tasks reveals a fundamental limitation: a single task vector often fails to capture the full scope of a task. Instead, multiple subtask-specific vectors, distributed across the output sequence, are required to effectively represent and execute complex tasks. This finding challenges the notion that task vectors are inherently noisy approximations of task knowledge, demonstrating that critical task information may be absent from a single vector.

Future research should therefore explore these distributed and compositional mechanisms of task representation and execution in LLMs to develop a more nuanced understanding of in-context learning.

Limitations

First, all experiments were conducted on a single model, Llama-3-8B. Findings regarding optimal layer performance and task-type variability may not generalize to models with different architectures, parameter counts, or from different families. Second, we employed LLM-based evaluation due to the diversity of tasks in QUITEAFEW, precluding the use of a single standard metric like F1-score. This approach, while versatile, can introduce evaluator biases and complicates direct comparisons with studies using task-specific metrics. Third, our generation process used a fixed temperature for all evaluations. Varying decoding parameters might yield different insights into task vector efficacy, an aspect not explored here.

Ethics

This study examines the working mechanisms of large language models and, therefore, does not introduce risks beyond those typically associated with natural language processing or computational linguistics research.

We utilize the Alpaca dataset (Taori et al., 2023), which is licensed under the CC BY-NC 4.0 license, a license suitable for research purposes.

Use of AI Assistants We utilize Grammarly to enhance and proofread the text of this paper, correcting grammatical, spelling, and stylistic errors, as well as rephrasing sentences. Consequently, certain sections of our publication may be identified as AI-generated, AI-edited, or a combination of human and AI contributions.

References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey for in-context learning. *arXiv preprint arXiv:2301.00234*.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Etai Hendel, Ori Lieber, Ido Dalmedigos, and Yoav Shoham. 2023. In-context learning creates task vectors. *arXiv preprint arXiv:2310.15916*.

Dmitrii Kharlapenko, neverix, Neel Nanda, and Arthur Conmy. 2024. Extracting sae task features for in-context learning. https://www.alignmentforum.org/posts/5FGXmJ3wqgGRcbyH7/extracting-sae-task-features-for-in-context-learning. Accessed: 2025-05-18.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35.

Ziyi Luo, Yujie Wang, Yifan Zhang, Jiacheng Chen, and Saining Liu. 2024. Task vectors are cross-modal. *arXiv preprint arXiv:2410.22330*.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca.

Nelson Todd, Ori Lieber, Ido Dalmedigos, Etai Hendel, and Yoav Shoham. 2024. Function vectors in large language models. *arXiv preprint arXiv:2310.15213*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

A Instruction Dataset Classification Prompt

```
Your task is to classify each instruction based on how suitable it is for creating
    few-shot examples. An instruction is good for few-shots if you can generate many
     different input-output pairs (at least 30) where:
- The same instruction works for all pairs
- Each input is meaningfully different from others
- The output's correctness can be clearly evaluated

Output pure CSV starting with this header:
instruction|example_input|category|explanation

Categories:
GOOD = Good for few-shots: you can create many (30+) valid input examples
LIMITED = Bad for few-shots: cannot generate enough different examples (requires
    explanation)
INVALID = Invalid: impossible to complete with given input (requires explanation)

Technical rules:
- Start immediately with header
- Process **all** instructions exactly as written, in order
- No quotes in output
- Explain both LIMITED and INVALID cases
```

Listing 1: Prompt for filtering Alpaca instructions

B Few-Shot Creation Prompt

```
You are a specialized AI assistant tasked with generating diverse and meaningful
    examples based on given instructions. Your task is to generate {num_examples}
    different, high-quality input examples for a given instruction, along with
    corresponding outputs. Each example should be unique and demonstrate different
    aspects or applications of the instruction.

Here is the instruction and an example input-output pair for reference:

[INSTRUCTION]
{instruction}

Example format:
Input: {example_input}
Output: {example_output}

Your task is to:
1. Analyze the instruction and understand its scope
2. Generate {num_examples} different, realistic, and diverse inputs that could be
    used with this instruction
3. For each input, provide an appropriate output following the pattern shown in the
    example
4. Ensure each input-output pair is unique and demonstrates different aspects of the
     instruction
5. Format your response exactly as a CSV table with three columns with a header:
    counter|input|output

Requirements:
- Generate exactly {num_examples} examples
- Ensure all examples are distinct and non-repetitive
- Maintain consistent quality across all examples
- Follow the same style and format as the provided example
- Ensure inputs are realistic and contextually appropriate
- Make outputs match the format and style of the example output

Format your response exactly like this:
‘‘‘
counter|input|output
1|[first input]|[corresponding output]
2|[second input]|[corresponding output]
```

```
...
{num_examples}|[{num_examples}th input]|[corresponding output]
```

Important notes:
- Do not include explanations or additional text
- Start directly with the CSV format
- Use | as separator
- Escape any special characters within the text using double quotes
- Maintain consistent formatting throughout
- Ensure each row follows the exact same pattern
- Do not skip numbers or leave gaps in the counter

Begin your response now by outputting exactly {num_examples} examples in the specified CSV format with | as a separator.
```

Listing 2: Prompt for generating few-shot examples

## C Answer Quality Evaluation Prompts

```
You are a judge evaluating responses to tasks. You must provide exactly two scores:

1. Format score (0-10): How well the response matches the semantic type required by
    the task
- Score 10: Response provides exactly the type of answer requested (e.g.,
    classification label for classification tasks, Yes/No for yes/no questions)
- Score 7-9: Provides the right type of answer with minor formatting issues
- Score 4-6: Partially attempts to provide the required type (e.g., some
    classifications missing in classification task)
- Score 1-3: Attempts to answer but mostly missing required type markers
- Score 0: No attempt to provide the required type of answer

2. Correctness score (0-10): How accurate/correct the actual answer is within its
    task domain
- Score 10: Completely correct task completion
- Score 7-9: Mostly correct task completion with minor issues
- Score 4-6: Partial task completion with significant issues
- Score 1-3: Minimal correct task completion
- Score 0: No actual task completion (e.g., just repeating input, missing
    classifications)

Important: Simply repeating input or providing incomplete answers does not count as
    task completion. The response must actually perform the requested operation (
    classify, summarize, etc.) to receive any correctness points.

For classification tasks, any classification label gets a high format score even if
    wrong (e.g., answering "Opinion" for a fact still gets a high format score). For
     Yes/No tasks, any Yes/No answer gets a high format score regardless of
    correctness.

You must output your scores in exactly this format (without quotation marks):
FORMAT_SCORE,CORRECTNESS_SCORE

For example, the output might be:
8,3

Do not provide any explanations or additional text. Only output two numbers
    separated by a comma.
```

Listing 3: System prompt part of evaluating answer quality prompt

```
You are a judge evaluating responses to tasks. You must provide exactly two scores:
1. Instruction: """{instruction}"""
2. Sample Input: """{example_input}"""
3. Expected Output: """{example_output}"""
4. Submission to evaluate: """{output_to_evaluate}"""
```

Listing 4: User prompt part of evaluating answer quality prompt
```