Exploressing the Curious Case of Code Prompts

Li Zhang, Liam Dugan, Hainiu Xu, Chris Callison-Burch
University of Pennsylvania
{zharry, ldugan, seacow, ccb}@seas.upenn.edu

Abstract

Recent work has shown that prompting language models with code-like representations of natural language leads to performance improvements on structured reasoning tasks. However, such tasks comprise only a small subset of all natural language tasks. In our work, we seek to answer whether or not code-prompting is the preferred way of interacting with language models in general. We compare code and text prompts across three popular GPT models (davinci, code-davinci-002, and text-davinci-002) on a broader selection of tasks (e.g., QA, sentiment, summarization) and find that with few exceptions, code prompts do not consistently outperform text prompts. Furthermore, we show that the style of code prompt has a large effect on performance for some but not all tasks and that fine-tuning on text instructions leads to better relative performance of code prompts.

1 Introduction

Recent work has shown that pre-training language models (LMs) on a mixture of text and program code (e.g., Python or Javascript) makes them more capable of reasoning over natural language (Suzgun et al., 2022). Such program-trained language models (PLMs) significantly outperform text-only LMs on tasks such as math problems and tracking shuffled objects despite such tasks lacking any explicit code formulae (Liang et al., 2022).

Furthermore, prompting such PLMs with code-like structures (e.g., Python, JSON, PDDL) instead of text has been shown to lead to performance improvements on structured common sense reasoning (Madaan et al., 2022), event argument extraction (Wang et al., 2022), knowledge graph construction (Bi et al., 2023), story understanding (Dong et al., 2022), and causal reasoning (Zhang et al., 2023).

Such results naturally lead us to ask whether code-prompting is the preferred way of interacting with PLMs in general. While previous work is limited to reasoning tasks, in this work we analyze a broad selection of tasks (e.g., QA, sentiment, summarization) and systematically compare the performance of prompting PLMs with code vs. prompting with text. We find that:

- With the exception of some reasoning tasks, code prompts do not outperform text prompts.
- The style of code prompt has a large effect on performance for some but not all tasks.
- Fine-tuning on text instructions leads to relative improvements when using code prompts.

2 Experimental Design

Model Selection For our text-based LM we use the original 175 billion parameter davinci model introduced by Brown et al. (2020). For our PLM we use the newer code-davinci-002 model which was explicitly trained on text and code. Neither model underwent any supervised instruction fine-tuning. In addition, we analyze performance on text-davinci-002, which is a vari-

The code, prompts, and outputs for our experiments are public at github.com/zharry29/curious_code_prompts
ant of code-davinci-002 trained explicitly on human demonstrations using supervised fine-tuning\footnote{https://platform.openai.com/docs/model-index-for-researchers}.

We include this model to help us determine whether or not fine-tuning PLMs on text instructions affects their ability to interpret code prompts. All three models were queried through the OpenAI API\footnote{https://openai.com/blog/openai-api} and our experiments cost approximately $2700 in total (see Appendix F for the full cost breakdown).

### Task Selection
Following the methodology of Sanh et al. (2022) we select tasks in a top-down fashion by first choosing the categories of interest (e.g. Question Answering, Sentiment Analysis, Summarization) and then selecting datasets from within those categories. We pay special attention to common sense and causal reasoning tasks as PLMs prompted with code have been shown to perform well on such tasks. The resulting 12 tasks are listed in Table 1 and include Commonsense Reasoning, Natural Language Inference, Sentiment Analysis, Question Answering, and Summarization. More details on each task can be found in Appendix A.

### Prompt Formulation
We collect text prompts for each task using the PromptSource dataset (Bach et al., 2022), a publicly available collection of crowd-sourced prompt templates. For tasks with many prompts, we randomly select one from those provided in the dataset. For a few tasks absent on PromptSource, we write the prompts ourselves.

For our code prompts, we manually write four custom code prompts per task. The code prompt types are as follows, from least to most Pythonic.

(i). **Vanilla (Vanilla)**: instructions and inputs are given as variables with generic names;

(ii). **Var Identifier (VI)**: instructions and inputs are given as variables with meaningful names;

(iii). **Var Identifier + Comments (VIC)**: instructions and inputs are given as variables with meaningful names along with comments explaining their purpose;

(iv). **Class + Var Identifier + Comments (CVIC)**: instructions and inputs are given as a task-specific class. Functionality is “implemented” as member functions.

Figure 2 shows an example of the different styles of code prompts for the wikiHow temporal ordering task. Note that we attempt to write our code prompts such that we match the wording of the text-based PromptSource prompt as closely as possible.

At inference time, for each test example, we randomly sample in-context examples from the training set and add them to the context window until the maximum context length is reached. This process circumvents the bias caused by static in-context examples. We conduct an ablation study where we vary the random seed and show that this process produces consistent results (see Appendix D).

### Results

What is the best type of code prompt? We compare performance across the four code prompt types from Section 2 on all 12 tasks using code-davinci-002 and report our results in Figure 3. We find that no single type of code prompt performs significantly better than the others across all tasks and that the relative difference in performance between code prompts also varies significantly across tasks. For example, on IMDb and SQuAD all code prompts have roughly even performance while for tasks such as wikiHow-Temporal and WinoGrande we see a near 14% accuracy dif-

### Table 1: The 12 evaluation tasks. Macro F1 is based on Rajpurkar et al. (2016). For each task, we randomly sample a fixed set of 1000 examples from its validation or test set for evaluation. For OpenPI we are limited to 111 examples.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Task Category</th>
<th>Num. Eval Examples</th>
<th>Metric</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>HellaSwag</td>
<td>Commonsense Reasoning</td>
<td>1000 / 10042</td>
<td>Accuracy</td>
<td>Zellers et al. (2019)</td>
</tr>
<tr>
<td>wikiHow Goal-Step</td>
<td>Commonsense Reasoning</td>
<td>1000 / 1073</td>
<td>Accuracy</td>
<td>Zhang et al. (2020)</td>
</tr>
<tr>
<td>wikiHow Temporal</td>
<td>Commonsense Reasoning</td>
<td>1000 / 3100</td>
<td>Accuracy</td>
<td>Zhang et al. (2020)</td>
</tr>
<tr>
<td>WinoGrande</td>
<td>Commonsense Reasoning</td>
<td>1000 / 1767</td>
<td>Accuracy</td>
<td>Sakaguchi et al. (2021)</td>
</tr>
<tr>
<td>OpenPI</td>
<td>Commonsense Reasoning</td>
<td>111 / 111</td>
<td>ROUGE-F1</td>
<td>Tandon et al. (2020)</td>
</tr>
<tr>
<td>ANLI</td>
<td>Natural Language Inference</td>
<td>1000 / 3000</td>
<td>Accuracy</td>
<td>Nie et al. (2020)</td>
</tr>
<tr>
<td>Yelp</td>
<td>Sentiment Analysis</td>
<td>1000 / 1000</td>
<td>Pearson’s r</td>
<td>Zhang et al. (2015)</td>
</tr>
<tr>
<td>IMDB</td>
<td>Sentiment Analysis</td>
<td>1000 / 2500</td>
<td>Accuracy</td>
<td>Maas et al. (2011)</td>
</tr>
<tr>
<td>HotpotQA</td>
<td>Question Answering</td>
<td>1000 / 7405</td>
<td>Macro-F1</td>
<td>Yang et al. (2018)</td>
</tr>
<tr>
<td>SQuAD</td>
<td>Question Answering</td>
<td>1000 / 11873</td>
<td>Macro-F1</td>
<td>Rajpurkar et al. (2018)</td>
</tr>
<tr>
<td>CNN/Daily Mail</td>
<td>Summarization</td>
<td>1000 / 13368</td>
<td>ROUGE-2</td>
<td>Nallapati et al. (2016)</td>
</tr>
<tr>
<td>XSUM</td>
<td>Summarization</td>
<td>1000 / 11332</td>
<td>ROUGE-2</td>
<td>Narayan et al. (2018)</td>
</tr>
</tbody>
</table>
You are trying to \(\{\text{goal}\} \). You need to do two things:
(a) \(\{\text{step0}\}\)
(b) \(\{\text{step1}\}\)
The first thing to do is \(\{\text{first}\}\) and \(\{\text{second}\}\) refer to the true ordering of the steps.

```python
# The list of correct order of those two steps
order_of_exec = \([\{\text{first}\}, \{\text{second}\}]\)
```

Figure 2: An example of the four styles of manually written code prompts used in our analysis (Vanilla, VI, VIC, and CVIC) for the wikiHow temporal ordering task. At test time, variables in braces are replaced with information from the dataset item (as shown in Figure 1). For this task, \(\{\text{goal}\}\), \(\{\text{step0}\}\), \(\{\text{step1}\}\) refer to the article title and the steps to order while \(\{\text{first}\}\) and \(\{\text{second}\}\) refer to the true ordering of the steps.

![Figure 2](image_url)

Figure 3: Comparison of code-davinci-002 across the four types of code prompts. Figures are split to allow for different y-axis scales. We see that different prompts do better on different tasks and while some tasks have high variance over prompt types, others do not.

![Figure 3](image_url)

Figure 4: Performance score (y-axis) vs number of in-context examples (x-axis, in log scale) using code prompts (VIC) with code-davinci-002. We see that increasing number of examples does not always increase performance and in some cases makes it worse.

![Figure 4](image_url)

In Appendix C, we calculate the average rank of each code prompt type relative to each other and find that the “Var Identifier + Comments” (VIC) prompt is the best across all tasks on average (2.25 avg. rank). We thus use this prompt type for our comparison in all future sections.

**How many in-context examples should we include in our code prompt?** We would like to also investigate how the number of in-context examples in the prompt affects models’ ability to perform the task. We therefore conducted an experiment where we filled the context window of code-davinci-002 with in-context examples up to 2000 tokens, 4000 tokens, and 8000 tokens and plotted the validation accuracy of the model with respect to the number of examples in Figure 4.

Contrary to expectations, we find that the number of in-context examples has little effect on model performance for most tasks and actually has a negative effect on some tasks. This is especially interesting given that previous work on in-context learning with text prompts finds roughly monotonic improvement from adding more in-context examples (Liu et al., 2021). While further research is necessary, it seems that code prompts may have different scaling behavior than text prompts when used in in-context learning.
Which is better: code or text prompts? In our main experiment we compare the performance of the three GPT models on code prompts (VIC style) and text prompts across the 12 datasets. Given the results from Figure 4, we fill the context window of all models with in-context examples up to 4000 tokens to serve as a middle ground for comparing code and text prompts. We report the results of our main experiment in Table 2 and see several surprising trends.

First, we find that prompting PLMs with code leads to substantial increases in performance for certain few reasoning tasks but that this trend does not hold across all tasks—or even all reasoning tasks. For example, when using code prompts with code-davinci-002, we see a 10.5% accuracy increase on wikiHow temporal ordering but a 2.6% accuracy decrease on wikiHow goal-step inference despite both being commonsense reasoning tasks and having identical source material.

Second, we find that supervised instruction fine-tuning on natural language demonstrations does not hurt model performance on code. Rather, we observe that code prompts outperform text prompts on more tasks when using text-davinci-002 than when using code-davinci-002 despite the fact that text-davinci-002 received no additional fine-tuning on code instructions.

Finally, we find that LMs not explicitly trained on code can also benefit from code prompting on certain reasoning tasks. In particular, code prompts outperform text prompts on davinci for 3 out of our 12 tasks—the same proportion as davinci prompts outperform text prompts on certain reasoning tasks. In particular, code and text prompts. We report the results of the three LMs when using code prompts (+Code) vs. using text prompts (+Text). Blank cells indicate tasks for which single test examples could not fit in the context window. Color indicates whether or not code prompts are better, slightly better, slightly worse, or worse than text prompts: We see that while code prompts outperform text prompts for certain tasks (such as wikiHow temporal and Winograd) text prompts are better on average. We also find that instruction fine-tuning (text-002) allows for better code prompt utilization.

4 Conclusion

In this work we investigate whether or not there exists a systematic performance difference between prompting PLMs with code or with text. We confirm that there are indeed tasks for which code prompting is significantly more effective than text prompting and that this finding holds across different types of models. However, for most tasks, we find that text prompting is still the best method for eliciting few-shot generalization from PLMs.

Given this result it seems reasonable to attempt to predict which tasks will benefit from code prompts and which tasks will not. However, we show that making such predictions based on simple heuristics such as domain and task category is difficult and that the larger trends remain unclear. Future work should seek to investigate the core mechanism behind what makes code prompting effective for certain tasks.

Finally, concurrent to our work, a new line of research has emerged wherein models generate code and execute that code to produce valid output (Chen et al., 2022; Mishra et al., 2022; Gao et al., 2022; Lyu et al., 2023). Future work should consider whether or not the tasks that benefit from executable code prompts and non-executable code prompts have any overlap.

Table 2: Performance of the three LMs when using code prompts (+Code) vs. using text prompts (+Text). Blank cells indicate tasks for which single test examples could not fit in the context window. Color indicates whether or not code prompts are better, slightly better, slightly worse, or worse than text prompts. We see that while code prompts outperform text prompts for certain tasks (such as wikiHow temporal and Winograd) text prompts are better on average. We also find that instruction fine-tuning (text-002) allows for better code prompt utilization.
Limitations

One significant limitation to our study is that, as of March 23rd 2023, OpenAI has deprecated access to code-davinci-002\(^4\), thus rendering our results non-replicable for any team not granted special access to these models by OpenAI. We did not anticipate this deprecation while conducting this work and we believe this raises serious questions about the usage of API-based language models in scholarly work.

Another limitation is that the 12 tasks we selected may not be representative of the broader population of natural language tasks. Had we conducted our experiments on a larger selection of tasks there may have been larger-scale trends that we would have been able to uncover.

The largest and most pressing limitation with our work is that the models we are testing on have closed-source pre-training datasets. Thus, we are unable to verify the extent to which our task datasets have been included in the training or instruction fine-tuning data. Given that the training data for most of the models tested in this work cuts off in late 2021, this is a very strong possibility. Our results should be viewed with this limitation strongly in mind.

Finally, while we experimented with different code prompts, the search space of possible prompts is very large. Thus, it is very likely that there exists some prompt that outperforms our chosen prompts for each task. Drawing conclusions based on a limited sampling of prompts is tenuous and while methods exist for searching the space of all prompts, such techniques lack interpretability and erase any distinction between code and text prompt (Li and Liang, 2021).

Acknowledgements

The paper is dedicated to the late Prof. Dragomir Radev, the first mentor in NLP of the author Li Zhang, for igniting his passion for research and inspiring him to be a better human being.

We thank Shuyan Zhou, Aman Madaan, and Niket Tandon for valuable discussions about this work and we thank Alyssa Hwang for her contributions to the structure, presentation, and narrative of the final paper.

This research is based upon work supported in part by the DARPA KAIROS Program (contract FA8750-19-2-1004), the DARPA LwLL Program (contract FA8750-19-2-0201), the Office of the Director of National Intelligence (ODNI) via the IARPA HIATUS Program (contract 2022-22072200005), the NSF (Award 1928631), and gifts from Roblox and Salesforce. Approved for Public Release, Distribution Unlimited. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, ODNI, IARPA, NSF, the U.S. Government, or of Roblox or Salesforce. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References


Dong, Yijiang River, Lara J. Martin, and Chris Callison-Burch. 2022. Corrpus: Detecting story inconsistenc-
ties via codex-bootstrapped neurosymbolic reasoning.


Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weiizhu Chen. 2021. What makes good in-context examples for gpt-3?


the amount of knowledge encapsulated in language models during pre-training.

For our study we pick two open-book QA datasets, SQuADv2 (Rajpurkar et al., 2018) and HotpotQA (Yang et al., 2018), which allow us to focus our evaluation on how structured prompts affect models’ ability to comprehend long text input.

For both SQuADv2 and HotpotQA, we evaluate model performance based on the macro-averaged F1 score as proposed in Rajpurkar et al. (2016). This metric measures the average overlap between the prediction and ground truth answer. It is calculated by treating the prediction and ground truth as bags of tokens, and first computing their F1. Then, the maximum F1 score is taken over all of the ground truth answers for a given question, and that score is averaged over all of the questions to get the final result.

Commonsense Reasoning is a machine reasoning task that demands the use of commonsense knowledge which is oftentimes implicitly present in the text (Sap et al., 2020). The customary formulation of commonsense reasoning tasks are Classification, where the input is a context, optionally with candidate answers as choices, and the output is a label from a pre-defined label space, and Question Answering (QA), where the input is a context followed by a reasoning question and the output is in free-form language.

In this study, we selected four Classification style commonsense reasoning tasks: wikiHow Temporal and wikiHow Goal-Step (Zhang et al., 2020), ANLI (Nie et al., 2020), and HellaSwag (Zellers et al., 2019). We also included one Question Answering style task with OpenPI (Tandon et al., 2020). In addition, we evaluate our models on WinoGrande a comprehensive reasoning benchmark dataset (Sakaguchi et al., 2021).

For wikiHow Goal-Step, wikiHow Temporal, HellaSwag, WinoGrande, and ANLI, we use classification accuracy as the evaluation metric. To evaluate OpenPI, we use F1 score based on the ROUGE metric as described in the original paper (Tandon et al., 2020).

Sentiment Analysis is a task that is concerned with judging emotion and its degree in text. Given a passage, a language model is tasked with classifying the sentiment (positive, negative, neutral) and/or its degree (strongly, weakly, moderately).

The selected datasets, namely IMDb (Maas et al.,
Table 3: Relative performance rank of the four code prompt types from Section 2 across the 12 tasks. Ranks are calculated based on the results reported in Figure 3. We see that the “Variable Identifier + Comments” (VIC) style prompt performs the best out of all code prompt types on average.

2011) and Yelp (Zhang et al., 2015), are both constructed using customer reviews. The IMDb dataset proposes a binary classification problem where the input is a movie review and the label space is \{negative, positive\}. Yelp proposes a five-way classification problem where the input is a restaurant review and the label space is the number of stars (out of 5) the customers assigned to the restaurant.

For IMDb, we use accuracy as the evaluation metric and for Yelp, we use Pearson Correlation between the predicted rating and the ground truth rating as the evaluation metric.

B Hyperparameters

For all our experiments regarding GPT-based models, we use a max token of the maximum of possible output tokens in the ground-truth development set. We use a top p of 1, and no frequency and presence penalty. We use a temperature of 0 for classification and multiple-choice tasks and a temperature of 0.7 for generation tasks.

C Ranking of Code Prompt Styles

In Table 3 we report the rank-based statistics of the four code prompt types from Section 2 on our 12 tasks. Ranks are calculated based on the results reported in Figure 3 of the main paper. The numbers in a row reflect the relative standing of each code prompt on the corresponding task. While we note that all code prompts perform within ±0.5 ranks of each other on average, we see that on average the VIC prompt performs the best across all tasks and the Vanilla prompt performs the worst. Looking to the standard deviation section, we see that the VI prompt performs the most consistently across all tasks and that once again the Vanilla prompt performs the least consistently.

D Ablation Study

To see whether the findings in our Results section could be attributed to variance in the random sampling of in-context training examples per test example, we conduct five repeated runs using code-davinci-002 with different random seeds each time and calculated the standard deviation across the five runs. We report our results in Table 4 and find that the choice of in-context examples accounts for very little of the observed variance across prompt type and context length. This finding is surprising as previous work has shown that the selection and ordering of in-context examples has a very large effect on the performance of models (Liu et al., 2021). However, it seems that our approach of random sampling in-context examples per test item helps to lessen this inherent variance.

E Evaluation on text-davinci-003

While conducting our research into the differences between code and text prompts, OpenAI released the text-davinci-003 model. This model differs from text-davinci-002 in that it is trained using Reinforcement Learning with Human Feedback (RLHF) instead of supervised instruction fine-tuning (Ouyang et al., 2022). The
Table 5: Performance of the three GPT-3.5 models across our 12 datasets with text prompts. (+IFT) indicates the addition of supervised instruction fine-tuning and (+RLHF) indicates the addition of training using Reinforcement Learning from Human Feedback (Ouyang et al., 2022). We see that RLHF does not always improve performance and that for some tasks (HotpotQA and wikiHow Goal-Step) it causes large degradations in performance.

Table 6: The total estimated cost of running davinci, text-davinci-002 and text-davinci-003 for 1000 data samples from each dataset (except for OpenPI).

F Evaluation Cost

In this section we report the approximate cost of conducting our experiments. In our study we use four OpenAI models, namely davinci, code-davinci-002, text-davinci-002 and text-davinci-003. While code-davinci-002 is free to use at the time of this study, we report the approximate cost of running the experiments on the other three models\(^5\) in Table 6. To estimate the cost of an experiment, we calculate the approximate number of tokens necessary for computing one dataset example and then multiplied that by the number of examples in the dataset. For classification tasks, since we fill up the context window to roughly 4000 tokens for every test example, we estimate the number of tokens to be 4000 (3999 tokens for the prompt and 1 token for the label). To estimate cost for generative tasks (OpenPI, HotpotQA, SQuAD, CNN/Daily Mail, and XSUM), we compute the average generation length from our generated samples and assume the in-context examples take up 3500 tokens. While this calculation results in a fairly loose upper bound, we believe this to be a good estimate of the total cost incurred by the project as such overestimates help offset the cost of other miscellaneous API queries done over the course of the project.

\(^5\)The cost of querying davinci, text-davinci-002 and text-davinci-003 is $0.02/1,000 tokens at the time of study. See https://openai.com/pricing for more details.