Coder Reviewer Reranking for Code Generation

Tianyi Zhang [*1] **Tao Yu** [2] **Tatsunori B. Hashimoto** [1] **Mike Lewis** [3] **Wen-tau Yih** [3] **Daniel Fried** [4] **Sida I. Wang** [3]

Abstract

Sampling diverse programs from a code language model and reranking with model likelihood is a popular method for code generation but it is prone to preferring degenerate solutions. Inspired by collaborative programming, we propose Coder-Reviewer reranking. We augment *Coder* language models from past work, which generate programs given language instructions, with *Reviewer* models, which evaluate the likelihood of the instruction given the generated programs. We perform an extensive study across six datasets with eight models from three model families. Experimental results show that Coder-Reviewer reranking leads to consistent and significant improvement (up to 17% absolute accuracy gain) over reranking with the Coder model only. When combined with executability filtering, Coder-Reviewer reranking can often outperform the minimum Bayes risk method. Coder-Reviewer reranking is easy to implement by prompting, can generalize to different programming languages, and works well with off-the-shelf hyperparameters.

1. Introduction

Recent pretrained language models (PLMs) have demonstrated an impressive ability to generate code given natural language instructions (Chen et al., 2021; Fried et al., 2022; Chowdhery et al., 2022; Nijkamp et al., 2022). One popular technique is to use a generative language model trained on code, which we call the Coder model, to sample multiple code solutions for a single instruction and rerank the solutions based on the likelihood the Coder model assigns to each (Chen et al., 2021). Despite its wide-spread use, reranking with the Coder model often mistakenly prefers degenerate solutions, *e.g.*, extremely short code or repetitive solutions. As a result, reranking performance often decreases when the number of candidate program increases (Figure 2c). These biases are known to arise in language models when using mode-seeking inference methods such as greedy decoding (Holtzman et al., 2020) or beam search (Li et al., 2016; Stahlberg & Byrne, 2019).

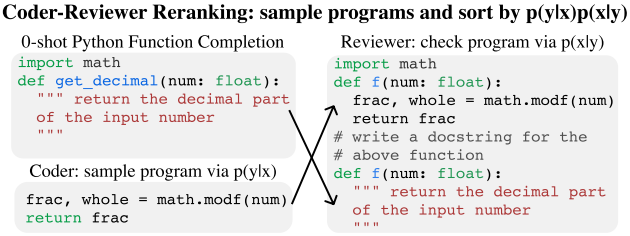


Figure 1. Given a language instruction x, a Coder model samples programs y, and a Reviewer model checks the generated programs against the instruction by measuring $p(x|y)$. Coder-Reviewer reranking solicits a consensus between Coder and Reviewer by ranking with their product $p(x|y)p(y|x)$.

In this work, we take inspiration from collaborative software development. For example, in the standard practice of code review, programmers submit implementations given specifications and have the submitted code cross validated by other code reviewers. We instantiate this idea by using prompting to obtain a Reviewer model, which checks the generated programs against the language instruction. Concretely, we first sample programs y given the instruction x via the Coder model $p(y|x)$ and cross-check via the Reviewer model $p(x|y)$. The Reviewer model reinforces the language instruction by evaluating the likelihood of every word in the instruction.

To obtain a consensus between the Coder and the Reviewer, we propose Coder-Reviewer reranking which selects the solutions by the product of the reviewer model and the coder model, $p(x|y)p(y|x)$. We show that Coder-Reviewer is a specific instantiation of the Maximum Mutual Information (MMI) objective (Li et al., 2016), which favors solutions that have high mutual information with the instruction and down weights generic solutions (where $p(y)$ is high). MMI has also been shown to be effective against degenerate solutions in many other natural language processing tasks (Yin & Neubig, 2019; Lewis & Fan, 2019; Fried et al., 2018).

[*]Equal contribution [1]Stanford University [2]The University of Hong Kong [3]Meta AI - FAIR [4]Carnegie Mellon University. Correspondence to: Tianyi Zhang <tz58@stanford.edu>, Sida I. Wang <sida@meta.com>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

To implement the Reviewer model $p(x|y)$, we propose a simple prompting method. After a program y is generated by the Coder model $p(y|x)$, we invert the order in which the instruction x and the solution y appear in the prompt, and query the pretrained language model again to estimate $p(x|y)$. Our prompting approach avoids any additional training and is easy to generalize to different programming languages. Figure 1 shows an example prompt: the Coder model generates programs given the function header and the docstring; we then extract the generated program and place it before the docstring when prompting the Reviewer model.

We carry out an extensive empirical study on six datasets with three different programming languages and experiment with seven models from three model families. Compared to past works' approach of ranking with the Coder model $p(y|x)$ alone, Coder-Reviewer reranking demonstrates consistent and effective performance gains (up to 17% absolute accuracy gain). When combined with executability filtering, Coder-Reviewer reranking can often outperform the minimum Bayes risk decoding method (Shi et al., 2022), which involves more complex aggregation of the executed outputs. The code will be made available after anonymous period.

2. Related Work

Code Generation. Many prior works explored code generation with neural networks (Allamanis et al., 2015; Ling et al., 2016; Iyer et al., 2018; Yin & Neubig, 2017; Yasunaga & Liang, 2020) and many benchmarks have been proposed to evaluate code model performance (Hendrycks et al., 2021; Yu et al., 2018; Lin et al., 2018). Recently, large language models pretrained on code have shown unprecedented zero-/few-shot ability, and can even perform well in code competitions that are challenging to programmers (Chowdhery et al., 2022; Chen et al., 2021; Austin et al., 2021; Li et al., 2022). Our work builds on the impressive ability of pretrained code models and achieves additional gains by leveraging a Reviewer model that evaluates the probability of the language instruction given generated programs.

Maximum Mutual Information (MMI) and its variants have been shown to be effective in many natural language processing tasks, including text classification (Min et al., 2021), speech processing (Bahl et al., 1986), dialogue (Li et al., 2016), instruction following (Fried et al., 2018), question answering (Lewis & Fan, 2019), passage retrieval (Sachan et al., 2022), and semantic parsing (Yin & Neubig, 2019). In contrast to Maximum Likelihood which optimizes $\log p(x|y)$, MMI optimizes the pointwise mutual information $\log \frac{p(x,y)}{p(x)p(y)}$. In practice, it is popular to optimize a weighted version of the MMI objective (Li et al., 2016). We show in Section 4 that Coder-Reviewer reranking is a specific instantiation of the weighted MMI objective. However, Coder-Reviewer reranking differs from this work by leveraging prompting to obtain the Reviewer model $p(x|y)$, rather than training a separate model, and by showing that the objective produces substantial benefits for the task of code generation. Concurrently, Ye et al. (2022) explore a MMI-like prompting approach for reasoning tasks.

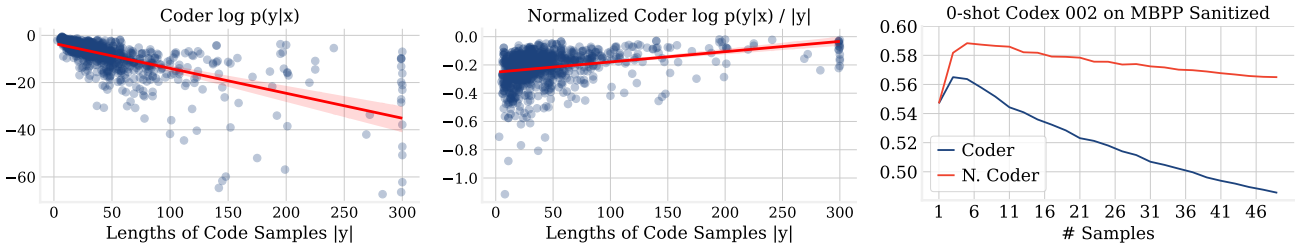
Reranking Methods for Code Generation. Chen et al. (2021) point out that the diverse samples from large language models often contain correct programs and they propose to rank program samples by the Coder model $p(y|x)$. Since then, many methods have been proposed to leverage sample consistency (Shi et al., 2022) or training supervised rerankers (Inala et al., 2022) In particular, Shi et al. (2022) and Li et al. (2022) propose to cluster program surface forms using the executed outputs of the generated programs. Chen et al. (2022) propose to generate unit tests for Python function completion problems and design selection programs that validate generated programs and unit tests against each other. On the one hand, Coder-Reviewer reranking does not require execution, which enables it to be applied in more diverse scenarios, and Coder-Reviewer reranking is not specific to any programming languages or packages. On the other hand, Coder-Reviewer reranking is orthogonal and complementary to methods that incorporate execution semantics: in Section 7.1, we show empirical results on the benefits of combining Coder-Reviewer reranking with execution-based ranking methods.

3. Background

Zero-/Few-shot Code Generation. We are interested in using pretrained code language models to generate code y conditioned on natural language instructions x. In addition, we assume access to a context c that provides useful code context such as package imports or data dependencies. In the example shown in Figure 1, x is the instruction "return decimal part of the input number", y is the generated function body, and c includes importing the math package. For few-shot code generation, we also have n demonstration examples $(\hat{x}_1, \hat{y}_1, \hat{c}_1)$ to $(\hat{x}_n, \hat{y}_n, \hat{c}_n)$.

To solve this problem, our key leverage is a Coder model — a conditional language model $p_\theta(y|c, x)$, which can generate programs given the instruction and context. In this case of few-shot generation, the Coder model also conditions on the demonstration examples, i.e., we consider $p_\theta(y|\hat{c}_1, \hat{x}_1, \hat{y}_1, \ldots, \hat{c}_n, \hat{x}_n, \hat{y}_n, c, x)$. Throughout this work, we implement these conditional models using prompting: placing the relevant objects such as c and x into the input context of the language model.

Collecting Program Samples. To obtain programs from the Coder model p_θ, we sample autoregressively using a temperature-scaled p_θ. If the temperature is low, temperature scaling increases the probability of sampling well-



(a) (b) (c)

Figure 2. 2a and 2b show that the Coder model $p(y|x)$ has a strong dependence on the length of generated code and length normalization can introduce additional biases. 2c shows that in practice, Coder-only reranking (Coder) and normalized Coder-only reranking (N. Coder) have worse performance when the number of samples becomes large, which can be caused by selecting degenerate solutions.

formed programs.

Coder-Only Reranking. Once we have collected 25-100 samples, we rerank and select a single program as our output program. A popular technique in prior work (Chen et al., 2021) is to pick the program y that has the highest Coder model likelihood $\log p(y|x, c)$. However, the Coder model likelihood $\log p(y|x, c)$ is biased toward shorter programs (Stahlberg & Byrne, 2019). To counter this shortcoming, Chen et al. (2021) propose to apply length normalization and instead rerank using the average token-level log likelihood, $\frac{1}{|y|} \log p(y|c, x)$.

Failures of Coder-Only Reranking. Figure 2a and Figure 2b plot the Coder $\log p(y|x)$ and Normalized Coder $\frac{1}{|y|} \log p(y|c, x)$ values against the lengths of generated programs $|y|$. From these figures we can observe that Coder-only reranking prefers short solutions; applying length normalization overcorrects and prefers long solutions. This is a known issue of reranking/searching using the likelihood of neural language models (Holtzman et al., 2020; Stahlberg & Byrne, 2019). In practice, many of these short programs are trivial (*e.g.* containing only a `return` or `pass` statement) and many of these long programs contain repetitive code. Figure 2c plots the accuracy of ranking with Coder model or the length normalized Coder model (N. Coder) versus the number of program samples. We observe that accuracy degrades as the number of program samples increases, since degenerate but high-scoring solutions have an increasing chance of appearing as more programs are sampled. This demonstrates that both types of Coder likelihood are insufficient as reranking objectives for code generation. We further analyze this behavior in Section 5.

4. Coder-Reviewer Reranking

Reviewer Model $p(y|x)$. In real-world professional software development, it is common to have programmers review each other's work. Motivated by this, and the previously-demonstrated shortcomings of Coder-only reranking, we introduce a Reviewer model. Note that we use the same underlying model p_θ for both the Coder and the Reviewer model but prompt differently. In the case of few-shot code generation, we will have $p_\phi(x|c, y, \hat{c}_1, \hat{y}_1, \hat{x}_1, \ldots, \hat{c}_n, \hat{y}_n, \hat{x}_n)$. Compared to the Coder model, the Reviewer model is tasked with evaluating the likelihood of the instruction x. Degenerate programs cannot account for the instruction well and therefore will often have a low Reviewer model likelihood.

3 -shot task-agnostic prompting

Coder Prompt

```
<text>Print info of "bash"</text>
<code>echo $(ls -l /bin/bash)</code>
... 2 more demonstration examples
<text>Change the owner of "dir"
to "nginx"</text>
<code>
```

Reviewer Prompt

```
<code>echo $(ls -l /bin/bash)</code>
<text>Print info of "bash"</text>
... 2 more demonstration examples
<code>chown nginx:nginx dir</code>
<text>Change the owner of "dir"
to "nginx"</text>
```

Figure 3. Example task-agnostic prompt on the NL2Bash dataset. We invert the order in which language instruction and the generated program appear to estimate the Reviewer model $p(x|y)$.

Prompting for the Reviewer Model. We adopt a prompting-based approach to implement the Reviewer model. In short, once the program samples y are generated, we invert the order in which instruction x and program y appear in the input context and query the language model p_θ again to obtain their likelihood. Recall that in Figure 1, we demonstrate a task-specific prompt we designed for Python function completion datasets. In this prompt, we first duplicate the function header and place the generated program y before the instruction docstring x. Additionally, we insert a natural language task specification "write the docstring for the above function" into the prompt to further specify the task to the pretrained language model. In Figure 3, we give another example of a 3-shot task-agnostic prompt on the NL2Bash dataset. Shi et al. (2022) propose this task-agnostic Coder model prompt that marks the location of instruction and generated programs with html-like tags. For prompting the Reviewer model, we swap the language instruction and the generated program, along with their tags. Importantly, we apply this inversion to both the demonstra-

tion examples and the last test example.

Combining Coder and Reviewer. To leverage the information provided by both the Coder and the Reviewer model, we propose Coder-Reviewer reranking, which reranks programs using the product of the Coder and the Reviewer,

$$\log p(x|y)p(y|x) = \log p(x|y) + \log p(y|x)$$
$$\text{(Coder-Reviewer Reranking)}.$$

Similarly, we can apply Coder-Reviewer reranking when using the length normalized Coder score,

$$\frac{\log p(x|y)}{|x|} + \frac{\log p(y|x)}{|y|}$$
$$\text{(Normalized Coder-Reviewer Reranking)}.$$

Here, we normalize the reviewer model score $p(x|y)$ to match the scale of the normalized Coder model score.

Because Coder-Reviewer Reranking combines two models by taking their product, Coder-Reviewer Reranking is sensitive to low probability under either model. As a result, Coder-Reviewer reranking seeks out program samples that obtain a consensus from both the Coder and the Reviewer. In the next section, we illustrate that by seeking a consensus, Coder-Reviewer can successfully alleviate the biases from each individual component.

Understanding the Relation between Coder-Reviewer Reranking and Maximum Mutual Information. We now show that Coder-Reviewer reranking is a special instantiation of the Maximum Mutual Information objective (Li et al., 2016). To avoid notation clutter, we abstract away the problem structure specific to code generation and denote task input x and output y. The usual Maximum Likelihood objective optimizes $p(y|x)$, which is the same as reranking with the Coder model. However, Maximum Likelihood has the risk of preferring generic or degenerate answers (Holtzman et al., 2020) and Li et al. (2016) propose to optimize Mutual Information $\frac{p(x,y)}{p(x)p(y)}$. Moreover, Li et al. (2016) propose to optimize for a weighted version of the Mutual Information objective, using a weighting parameter α:

$$\text{argmax}_y \ \log \frac{p(y,x)}{p(x)p(y)^\alpha} \tag{1}$$
$$= \text{argmax}_y \ (1-\alpha)\log p(y|x) + \alpha \log p(x|y) \tag{2}$$
$$= \text{argmax}_y \ (1-\alpha)\log p(y|x) + \alpha \log \frac{p(y,x)}{p(x)p(y)} \tag{3}$$

From (2), we see that Coder-Reviewer reranking is a special case of this objective where α is set to 0.5 (we include a derivation in Appendix A). From (3), we see that both can be viewed as interpolating likelihood with a mutual information criterion, with strength α. In other words, Coder-Reviwer reranking favors program samples that have high mutual information with the language instruction and therefore can filter out low quality solutions that cannot explain the instruction well. This perspective also motivates exploring using hyperparameter α to control the mixing ratio between the Coder and the Reviewer. In section Appendix C.3, we show that the off-the-shelf hyperparameter $\alpha = 0.5$ usually works well already, and tuning α can give a small additional gain. Next, we conduct an quantitative analysis to show that Coder-Reviewer model can filter low quality programs.

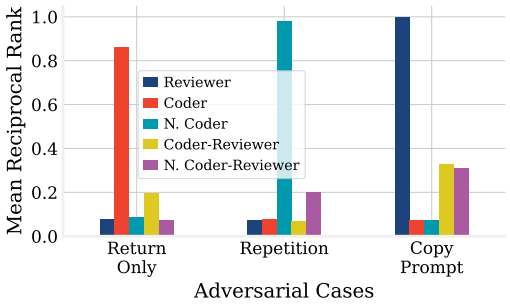


Figure 4. Mean Reciprocal Rank of ranking three typical degenerate cases. Coder Reviewer variants alleviate the biases in its individual components.

5. Analysis of Degenerate Cases

In this section, we experiment with ranking artificially created degenerate programs to analyze the preference of different ranking methods. We experiment with the Codex002 model (Chen et al., 2021) on the 0-shot sanitized MBPP dataset (MBPP-S Austin et al., 2021), which is a Python function completion dataset with example prompt structure shown in Figure 1. We defer experimental details and dataset details to Section 6.

For each MBPP problem, we sampled 25 programs via the Coder model $p(y|x)$. Then, we construct three cases that reflect common degeneracies and inject them among the 125 samples. We construct these cases: A. `ReturnOnly` where the function body only contains a return statement B. `Repetitive` where the function body contains print statements printing from 1 to 50, with each line containing exactly one print statement. C. `CopyPrompt` where the function body only contains a comment that repeats the language instruction.

With these programs, we now compare the reranking behavior of Coder $p(y|x)$, normalized Coder (N.Coder) $\frac{1}{|y|}\log p(y|x)$, and Reviewer $p(x|y)$. We also included Coder-Reviewer reranking and the normalized version N.Coder-Reviewer into the comparison. Figure 4 plots the mean reciprocal rank (MRR) of the three degenerate cases ranked by different methods, where high MRR suggests a bias toward the degenerate case. Figure 4 shows that

both Coder and Reviewer methods fail on different degenerate programs: Coder favors programs that are effectively empty, even though those clearly do not follow the language instructions; Normalized Coder favors programs full of repetitions; and Reviewer can be biased toward spurious surface form overlap between generated programs and the language instruction. In contrast, Coder-Reviewer reranking can alleviate the biases in its individual components.

Degenerate Solutions Rejection. Based on this analysis, we prescribe three easy-to-implement procedures to reject degenerate solutions before reranking takes place. By strengthening the Coder and Reviewer baselines, we know that any performance gain brought by Coder-Reviewer cannot be easily attributed to simple fixes.

- We filter empty programs. For Python function completion datasets, we also filter trivial solutions that only contain `return` or `pass`.
- We filter repetitive programs whose `zlib` compressed representation is more than four times shorter than the original program.[1]
- For all Python code generation problems, we use an off-the-shelf canonicalization software `pyminifier` to remove comments, docstrings, and replace all print and assertion messages to empty strings. This procedure helps reduce the spurious surface form overlap with the language instruction. For function completion problems, we also standardize the function names.

We analyze the effects of the above procedure in Section 7.3. Overall, we find that these procedures help improve both the baseline Coder methods and our proposed Coder-Reviewer methods. Next, we present our extensive empirical study to demonstrate that Coder-Reviewer is an effective and consistent method.

6. Experimental Setup

We present an extensive empirical study that spans six datasets and three model families. Our evaluation data consists of both zero-shot and few-shot settings and investigates both task-specific and task-agnostic prompt design, and we study eight models whose parameter counts range across two orders of magnitude.

6.1. Datasets

We provide a brief description of our experimental datasets and present a detailed description in Appendix B.1. We first consider three zero-shot datasets, which all involve generating Python code. **HumanEval** and **MBPP-Sanitized** are two popular Python function completion datasets. On these datasets, we generally follow the prompt design in Chen et al. (2022) and use the Reviewer prompt presented in Figure 1. **Plotting** is a subset of DS-1000 (Lai et al., 2022) and contains 155 realistic questions adapted from StackOverflow about `matplotlib`. The rest of DS-1000 cannot be easily handled by left-to-right autoregressive because it often introduces additional contexts that are better addressed by infilling models. We leave applying Coder-Reviewer reranking to infilling models for future work.

Next, we consider three three-shot datasets, **MBPP**, **Spider**, and **NL2Bash**, which include programming languages other than Python. Our task-agnostic prompts and other experimental design on these three datasets are all taken from Shi et al. (2022) and are similar to the one presented in Figure 3.

6.2. Models

Codex (Chen et al., 2021) models are descendants of GPT-3 (Brown et al., 2020). We consider three variants of the codex model: Codex002 (`code-davinci-002`), Codex001 (`code-davinci-001`), and Codex-Cushman (`code-cushman`). While the exact modeling details are not made public, we assume based on their codenames that Codex002 and Codex001 are 175B models and Codex-Cushman is a 12B model.

InCoder (Fried et al., 2022) models are trained with the causal mask language modeling objective and are capable of both infilling and left-to-right generation. In this work, we only leverage their autoregressive ability to keep the comparison consistent with other model families and leave the investigation of its infilling ability for future work. We experiment with 6B and 1B models from this family.

CodeGen (Nijkamp et al., 2022) is a family of autoregressive language models pretrained on code data. We experiment with the 16B, 6B, and 2B CodeGen models.

6.3. Implementation Details

Due to the lack of validation split on the benchmarks we experimented with, we restrain ourselves from hyperparameter search and rely on a single set of decoding hyperparameters. On all datasets, we sample with temperature 0.4 and set the max tokens to be 300. For our main results in Table 1, we sample 125 different programs for each problem and then bootstrap 50 times to report the mean accuracy of reranking 25 samples. Given the small size of these datasets, we find bootstrapping to be helpful in reducing variance. In Section 7.2, we analyze the impact of decoding hyperparameters on the validation accuracy. We apply the proposed degenerate solution rejection to all baseline and proposed methods and study its effect in Section 7.3. Additionally, we apply executability filtering to all baseline and

[1] This threshold corresponds to `Repetitive`, which prints from 1 to 50.

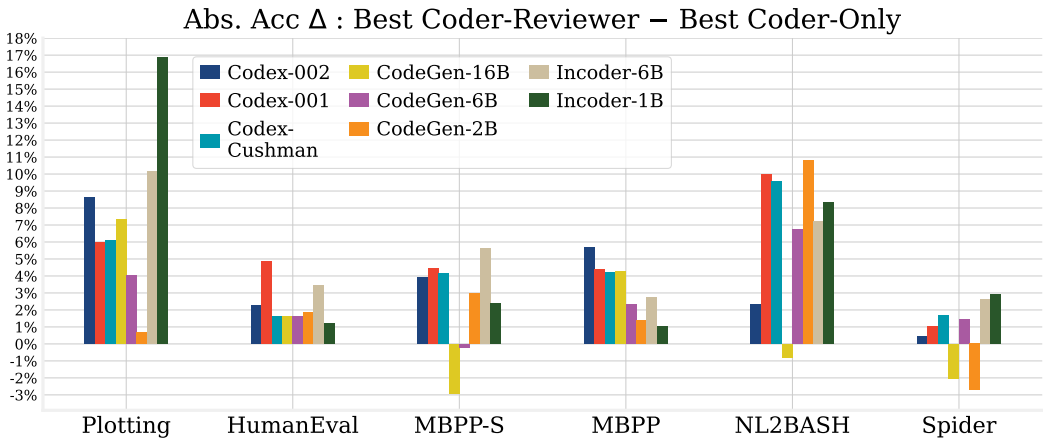


Figure 5. Absolute accuracy difference between the best Coder-Reviewer variants (with or without length normalization) and the best Coder variants (with or without length normalization). We observe performance gain from applying Coder-Reviewer on 43 out of 48 dataset cross model pairs.

proposed methods to better compare with state-of-the-art methods that rely on execution more heavily. Executability filtering (Shi et al., 2022) removes programs that produce runtime errors before applying any other ranking methods. We present the ranking results without executability filtering in Appendix C.1.

6.4. Reranking Methods

Baselines. We first compare to baseline selection methods that are popular in existing work. Random reports the percentage of correct programs after degenerate solution rejection. Coder selects the program that has the highest $p(y|x)$ and N. Coder applies length normalization to the Coder likelihood.

Proposed Methods. We then compare the methods proposed in this work. Reviewer selects programs that has the highest $p(x|y)$. Coder-Reviewer ranks via the product of the Coder and the Reviewer model scores and N. Coder-Reviewer applies length normalization to the component model scores before taking the product.

State-of-the-art Methods. We compare to the competitive Minimum Bayes Risk method (MBR-EXEC; Shi et al., 2022), which compares the executed outputs of the generated programs on test inputs and selects programs whose outputs are most typical. While for Python function completion problems it is straightforward to compare execution outputs, this is less obvious when execution results involve more complex objects. On the plotting dataset, due to the complexity and the multimodal nature of `matplotlib`[2], we compare the output figures directly pixel-by-pixel. On HumanEval, we extract the first test case appearing in the docstring and use the executed output for MBR-EXEC. For other datasets, we follow the practices recommended in Shi et al. (2022). We compare to MBR-EXEC using the same setting with baseline and proposed methods in Table 1.

We also compare to CodeT (Chen et al., 2022), which generates assertions for Python function completion problems and proposes a novel graph-based aggregation process for selecting programs based on program-test agreement. While CodeT is a competitive method, it relies on the ability to generate good assertions and cannot be easily used on other programming languages/tasks. For example, models typically cannot generate unit tests used in the Plotting dataset because good tests involve complicated assertions on the figure object and do not naturally appear in public data. Another example is the spider dataset, where generating additional unit tests involves creating different input database (Zhong et al., 2022), a challenging problem on its own. Therefore, we separate CodeT from the rest of the comparison and compare to its reported numbers in Table 2.

7. Experimental Results

In Section 7.1, we show that Coder-Reviewer reranking is either the best performing method or competitive with the best performing method across all settings. In Section 7.2 and Section 7.3, we perform ablation studies for the different decisions made in developing Coder-Reviewer reranking and show that Coder Reviewer can work well with default hyperparameters and is more stable than Coder reranking.

7.1. Primary Results

First, we observe that Coder-Reviewer variants consistently improves over Coder variants. Figure 5 plots the absolute accuracy difference between the best Coder-Reviewer variant (with and without length normalization) and the best Coder variant. Overall, we observe that Coder Reviewer

[2]plots with different styles such as colors, fonts, etc. might all be correct but it is difficult to aggregate them.

	Plotting			HumanEval			MBPP-S		
	Codex002	InCoder6B	CodeGen16B	Codex002	InCoder6B	CodeGen16B	Codex002	InCoder6B	CodeGen16B
Random	57.6	21.9	36.7	49.2	16.0	32.2	58.8	25.4	45.7
Reviewer	65.2	**35.5**	**46.5**	61.2	17.6	37.3	59.5	28.8	43.6
Coder	57.7	15.5	38.6	45.1	20.1	33.5	59.8	28.1	50.3
Coder-Reviewer	58.3	16.4	41.7	56.7	**23.5**	**40.0**	**64.4**	**33.7**	47.4
N.Coder	59.4	22.6	37.4	60.2	19.2	38.5	60.5	27.1	48.6
N.Coder-Reviewer	**68.0**	32.8	45.9	**62.5**	22.0	39.6	61.6	31.1	45.8
MBR-EXEC	60.9	21.0	37.0	50.5	20.7	35.8	63.9	30.9	**53.5**
	MBPP			NL2BASH			Spider		
	Codex002	InCoder6B	CodeGen16B	Codex002	InCoder6B	CodeGen16B	Codex002	InCoder6B	CodeGen16B
Random	58.1	19.6	39.3	60.0	49.8	35.7	65.2	29.4	25.6
Reviewer	**66.9**	24.4	44.1	63.3	55.3	28.1	67.5	38.4	28.8
Coder	60.2	23.4	41.5	57.4	48.8	31.7	74.1	38.9	**33.7**
Coder-Reviewer	66.4	26.1	46.2	61.9	55.0	37.0	74.5	**41.5**	31.7
N.Coder	60.7	20.2	41.9	61.3	41.5	**37.8**	69.9	38.2	31.1
N.Coder-Reviewer	66.2	24.1	45.4	**63.7**	**55.9**	29.5	71.0	40.3	29.9
MBR-EXEC	63.0	**26.7**	**47.3**	57.4	48.8	32.4	**75.2**	38.2	30.6

Table 1. Bootstrapped reranking results with 25 samples. Bolded numbers indicate the best results on each column and Underlined numbers indicate the second best results. In each subsection, we compare including or not including Reviewer reranking (Random vs. Reviewer, Coder vs. Coder-Reviewer, etc.). Coder-Reviewer variants mostly outperform Coder variants, and often outperforms the competitive MBR-EXEC method. Across all columns except one, a Coder-Reviewer variant is either the best or the second best method.

leads to a consistent and significant improvement with very few exceptions. In certain settings, *e.g.* for InCoder models on the plotting dataset, Coder-Reviewer leads to more than 10% improvement. The only 5 exceptions out of 48 model cross dataset pair where Coder-Reviewer is worse than Coder all involve CodeGen models. We suspect that difference in pretraining can cause a difference in the capability to estimate the Reviewer model $p(x|y)$ using the prompt we employ in this work.

Second, in Table 1, we present the ranking results on the largest models from each model family along with comparison to other methods. From Table 1 we find a Coder-Reviewer variant to be the best method most of the time and even when it is not, it is often the second best method. Analyzing across all data, we find that Coder-Reviewer variants are the best method 62.5% of the time, Reviewer 12.5% of the time, and MBR-EXEC 20.83% of the time. However, Reviewer and MBR-EXEC both have scenarios where they trail behind Coder-Reviewer significantly. For example when applied to the Codex002 model, MBR-EXEC is more than 10% worse than normalized Coder-Reviewer on HumanEval and 8% worse on Plotting. In practice, we find that MBR-EXEC works better when the quality of the test inputs that it executes on is high and when it is easy to aggregate the execution outputs. Compare HumanEval and MBPP-S, which have very simlar format, MBR-Exec performs much better on MBPP-S than on HumanEval. We ascribe this failure to the lower quality of the test inputs in HumanEval.

Third, Table 2 shows the performance of CodeT and Coder-Reviewer in the same setting of reranking 100 samples on HumanEval and MBPP-S. Here the results of CodeT are not strictly comparable to those of Coder-Reviewer. Recall that CodeT generates additional test cases for these Python function completion problems so CodeT does not execute any test cases provided in the input docstring. In contrast, we follow the setup in Shi et al. (2022) and execute the first test case to perform executbaility filtering. Still, Table 2 shows that Coder-Reviewer can achieve very competitive performance while being a more general method that can be applied to datasets where generating test cases is difficult.

In summary, our experimental results show that Coder-Reviewer reranking consistently improves performance over Coder-only reranking from past work across a diverse set of model families and datasets. In the following sections, we carry out ablation studies to understand the impact of design decisions and sensitivity to hyperparameters.

7.2. Understanding Hyperparameters

We explore the effect of different hyperparameters, including the number of samples and the mixing ratio α between Coder and Reviewer. We start by plotting the ranking accuracy of Coder, Reviewer, and Coder-Reviewer on the 0-shot MBPP-S dataset with Codex002 in Figure 6 (Figure 7 in

	HumanEval			MBPP-S		
	Codex002	InCoder6B	CodeGen16B	Codex002	InCoder6B	CodeGen16B
CodeT	65.8	20.6	36.7	67.7	34.4	49.5
Coder-Reviewer	57.9	24.3	42.6	64.7	35.8	50.3
N. Coder-Reviewer	66.9	22.9	40.5	61.0	30.2	46.1

Table 2. Bootstrapped reranking results with 100 samples. CodeT numbers are cited from Chen et al. (2022) and not strictly comparable: Coder-Reviewer variants use the first unit test in the docstring for executability filtering whereas CodeT generated its own unit tests. That being said, this comparison shows that Coder-Reviewer can achieve strong performance on the Python function completion datasets while being easier to generalize to other language/packages.

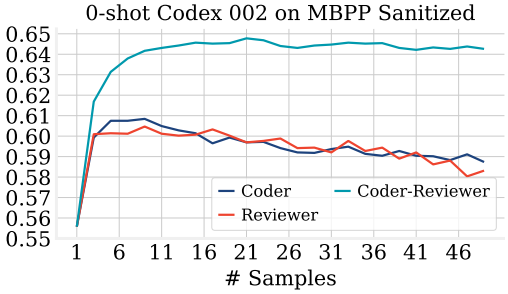


Figure 6. Accuracy versus number of ranking samples. Coder-Reviewer is more stable and robust to degenerate solutions than its individual components.

	HumanEval	MBPP-S
Random	$48.0_{-1.2}$	$58.1_{-0.7}$
Coder	$38.1_{-7.1}$	$55.3_{-4.5}$
N.Coder	$59.7_{-0.5}$	$60.0_{-0.5}$
Reviewer	$57.7_{-3.5}$	$55.8_{-3.7}$
Coder-Reviewer	$53.2_{-3.5}$	$60.5_{-3.9}$
Norm. Coder-Reviewer	$\mathbf{61.5}_{-1.0}$	$\mathbf{60.8}_{-0.7}$

Table 3. Ranking results of Codex002 without applying degenerate solution rejection. Numbers in subscripts showing the performance degradation compared to applying programmatic rejection. We see that degeneration solution rejection is helpful for all methods and unnormalized Coder benefits from the rejection most significantly.

Appendix C.2 includes plots on more methods and datasets). We observe that in contrast to its components Coder and Reviewer, Coder-Reviewer is more stable in ranking more sample programs, suggesting that it is more robust against potential degenerate solutions when compared to ranking with Coder or Reviewer model only.

In addition, recall that we can introduce a hyperparameter to control the mixing ratio α between Coder and Reviewer and have the objective $(1-\alpha)\log p(x|y) + \alpha \log p(y|x)$. In Coder-Reviewer reranking, we fix α at 0.5 for simplicity although the weighted objective is also popular in prior work (Li et al., 2016). From Figure 8 in Appendix C.3, we observe that the improvement from tuning α is typically less than 1%, with a few exceptions with the CodeGen models where tuning can lead to more significant improvements. Finally, we explore a alternate formulation of the Maximum Mutual Information objective and show that it performs generally worse than Coder-reviewer reranking in Appendix C.4.

7.3. Understanding Degenerate Solution Rejection

We start our analysis by removing the proposed degenerate solution rejection on Codex002's generated programs on HumanEval and MBPP-S. Table 3 shows the ranking results along with the performance degradation compared to applying the proposed rejection. We observe that degenerate solution rejection improves all of the baselines and Coder-Reviewer variants, with the most significant effect on Coder. Importantly, we see that the improvement from rejection is larger on Coder, Reviewer, and Coder-Reviewer than on Random, suggesting that these ranking methods have biases toward the degenerate solutions that we reject. On these two datasets, the effect of rejection is less obvious on normalized Coder and normalized Coder Reviewer but we find the rejection to be important for these methods on NL2Bash where it is more likely to have egregious repetitions in the program samples. Finally, these ablation results suggest that Coder-Reviewer still provides a benefit on top of degenerate solution rejection. Overall, we recommend the usage of our rejection methods because they are easy to implement and can effectively remove the most egregious degeneration programs targeting each ranking method.

8. Conclusion

We propose Coder-Reviewer reranking for code generation, which leads to a consistent and significant improvement over the Coder only reranking proposed in prior work. When combined with executability filtering, Coder-Reviewer reranking can often outperform the MBR-EXEC method. Coder-Reviewer is easy to implement with prompting, can generalize to different programming languages, and works well with off-the-shelf hyperparameter.

References

Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. Suggesting accurate method and class names. In *Proceedings of*

the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Bahl, L. R., Brown, P. F., de Souza, P. V., and Mercer, R. L. Maximum mutual information estimation of hidden markov model parameters for speech recognition. *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1986.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., and Chen, W. Codet: Code generation with generated tests. *ArXiv*, abs/2207.10397, 2022.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D. W., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Babuschkin, I., Balaji, S. A., Jain, S., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M. M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A. B., Barnes, P., Tay, Y., Shazeer, N. M., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B. C., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., García, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Díaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K. S., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311, 2022.

Fried, D., Hu, R., Cirik, V., Rohrbach, A., Andreas, J., Morency, L.-P., Berg-Kirkpatrick, T., Saenko, K., Klein, D., and Darrell, T. Speaker-follower models for vision-and-language navigation. In *NeurIPS*, 2018.

Fried, D., Aghajanyan, A., Lin, J., Wang, S. I., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *ArXiv*, abs/2204.05999, 2022.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with APPS. In *NeurIPS Datasets and Benchmarks Track*, 2021.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. The curious case of neural text degeneration. In *ICLR*, 2020.

Inala, J. P., Wang, C., Yang, M., Codas, A., Encarnaci'on, M., Lahiri, S. K., Musuvathi, M., and Gao, J. Fault-aware neural code rankers. *ArXiv*, abs/2206.03865, 2022.

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. Mapping language to code in programmatic context. In *EMNLP*, 2018.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., tau Yih, S. W., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022.

Lewis, M. and Fan, A. Generative question answering: Learning to answer the whole question. In *ICLR*, 2019.

Li, J., Galley, M., Brockett, C., Gao, J., and Dolan, B. A diversity-promoting objective function for neural conversation models. In *NAACL*, 2016.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Lin, X. V., Wang, C., Zettlemoyer, L., and Ernst, M. D. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *LREC*, 2018.

Ling, W., Blunsom, P., Grefenstette, E., Hermann, K. M., Kočiský, T., Wang, F., and Senior, A. Latent predictor networks for code generation. In *ACL*, 2016.

Min, S., Lewis, M., Hajishirzi, H., and Zettlemoyer, L. Noisy channel language model prompting for few-shot text classification. *ArXiv*, abs/2108.04106, 2021.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. 2022.

Sachan, D. S., Lewis, M., Joshi, M., Aghajanyan, A., Yih, W.-t., Pineau, J., and Zettlemoyer, L. Improving passage retrieval with zero-shot question generation. 2022. URL https://arxiv.org/abs/2204.07496.

Shi, F., Fried, D., Ghazvininejad, M., Zettlemoyer, L., and Wang, S. I. Natural language to code translation with execution. In *EMNLP*, 2022.

Stahlberg, F. and Byrne, B. On nmt search errors and model errors: Cat got your tongue? *arXiv preprint arXiv:1908.10090*, 2019.

Yasunaga, M. and Liang, P. Graph-based, self-supervised program repair from diagnostic feedback. In *ICML*, 2020.

Ye, S., Kim, D., Jang, J., Shin, J., and Seo, M. Guess the instruction! flipped learning makes language models stronger zero-shot learners. *ArXiv*, abs/2210.02969, 2022.

Yin, P. and Neubig, G. A syntactic neural model for general-purpose code generation. In *ACL*, 2017.

Yin, P. and Neubig, G. Reranking for neural semantic parsing. In *ACL*, 2019.

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., and Radev, D. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *EMNLP*, 2018.

Zhong, R., Snell, C., Klein, D., and Eisner, J. Active programming by example with a natural language prior. *ArXiv*, 2022.

A. Understanding the Relation between Coder-Reviewer Reranking and Maximum Mutual Information.

We include the derivation from Li et al. (2016) to show that Coder-Reviewer reranking is a special instantiation of the Maximum Mutual Information objective. We can show

$$\begin{aligned}
&\text{argmax}_y \ \log \frac{p(y,x)}{p(x)p(y)^\alpha} \\
&= \text{argmax}_y \ (1-\alpha)\log p(y|x) - \alpha \log p(x) + \alpha \log p(x|y) \\
&= \text{argmax}_y \ (1-\alpha)\log p(y|x) + \alpha \log p(x|y) \\
&= \text{argmax}_y \ (1-\alpha)\log p(y|x) + \alpha \log \frac{p(y,x)}{p(x)p(y)},
\end{aligned}$$

by using the fact that $p(x)$ is a constant and removing/adding it does not change the optimziation objective.

Alternatively, Li et al. (2016) propose another way to instantiate this objective,

$$\begin{aligned}
&\text{argmax}_y \ \log \frac{p(y,x)}{p(x)p(y)^\alpha} \\
&= \text{argmax}_y \ \log p(y|x) - \alpha \log p(y).
\end{aligned}$$

We show in Appendix C.4 that this alternate formulation leads to worse performance, which is a similar conclusion to the original finding in Li et al. (2016).

B. Additional Experimental details

B.1. Detailed Dataset Descriptions

HumanEval (Chen et al., 2021) contains 164 hand-written Python programming questions Chen et al. (2021). We use the prompts released by Chen et al. (2022), which removes the input-output cases present in the original prompts. This dataset evaluates the generated function by multiple assertions on the input-output relations.

MBPP-Sanitized (Austin et al., 2021) contains 427 crowd-sourced Python programming questions. Chen et al. (2022) adapts each problem into having a function header and relocate the natural language instructions into function docstrings, similar to the format of HumanEval. We use this prompt format for our experiment.

MBPP (Austin et al., 2021) original version consists of 974 python questions, with 500 of them used for testing and the rest for few-shot prompting. The prompt taken from Shi et al. (2022) uses one input-output assertion as additional context to help make the language instruction more specific. Unlike the 0-shot sanitized version of MBPP, this 3-shot setting requires the models to define the desired function name on its own.

Spider (Yu et al., 2018) is a benchmark of natural language to SQL query generation. There are 7000 examples for training/demonstration and 1034 questions for testing. The prompt taken from Shi et al. (2022) prepends the database schema as the program context. The generated SQL commands are evaluated with execution accuracy, comparing the database return values to the ones queried by ground-truth SQL commands.

NL2Bash (Lin et al., 2018) is a benchmark of translating natural language to bash commands. Because it is difficult to obtain executable enviroments for bash commands, this dataset evaluates character-level BLEU-4 score.

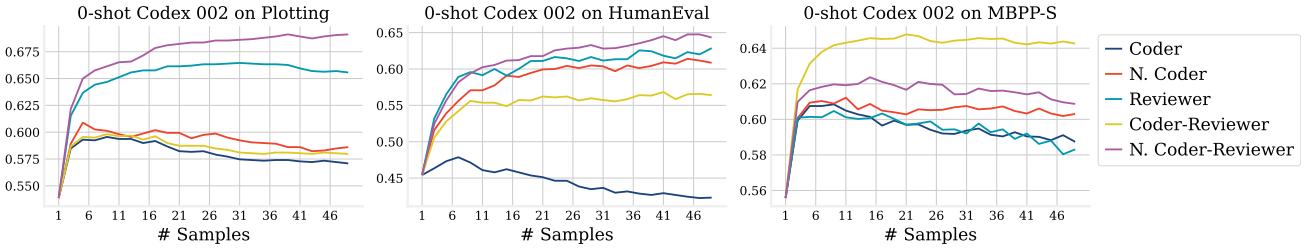


Figure 7. Accuracy versus number of ranking samples. Coder-Reviewer is more stable and robust to degenerate solutions than its individual components.

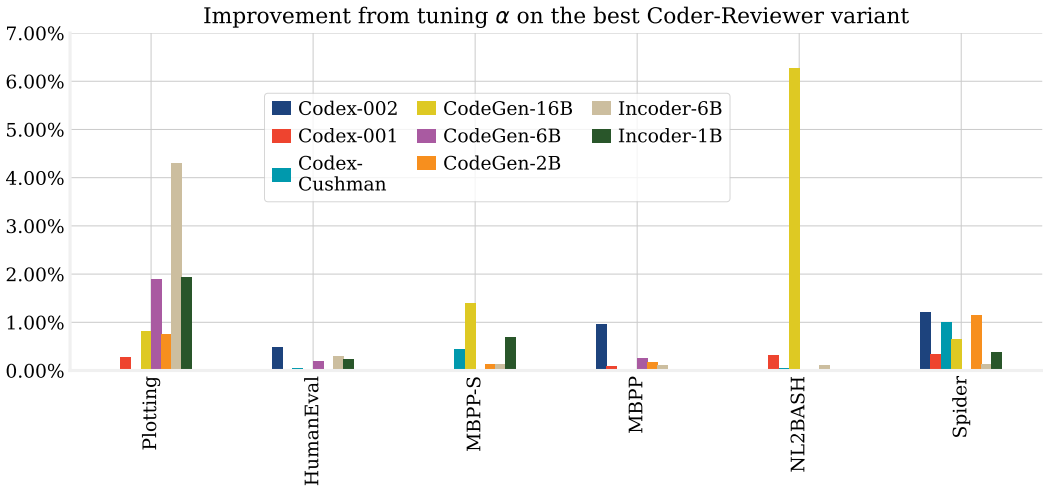


Figure 8. Accuracy improvement from grid searching ensemble mixing ratio α. Setting $\alpha = 0.5$ (i.e., Coder-Reviewer) usually performs well already and further grid searching mostly only lead to a 1-2% improvement. In few settings (CodeGen-16B on NL2Bash and Incoder-6B on Plotting), tuning α leads to a significant improvement.

C. Additional Results

C.1. Results from All Models

Table 5 to Table 10 plot ranking results on all eight models we experimented with and it also shows the results of not applying executability filtering. Analyzing across all data, we find that a Coder-Reviewer variant is the best method 62.5% of the time, Reviewer 12.5% of the time, and MBR-EXEC 20.83% of the time. Notably, a Coder-Reviewer variant is almost always the best method for Codex models with only two exceptions (Codex002 on Spider and MBPP). Even when none of Coder-Reviewer variants is the best method, a Coder-Reviewer variant is mostly the second best method. Overall, executability filtering improves most methods and does not change the comparison between different ranking methods.

C.2. Analyzing the Effect of Number of Program Samples

Figure 7 plots the performance of Coder-Reviewer variants, Coder variants, and Reviewer on all three 0-shot datasets we experimented with. Similar to Figure 6, Figure 7 shows that compared to Coder variants, Coder-Reviewer variants perform more consistently across the number of program samples being reranked. Here we plot the performance after applying degeneration solutions rejection. Still, on all three datasets, the performance of Coder starts to degrade after five or ten samples. Normalized Coder is more stable on HumanEval and MBPP-S but also does worse with more program samples on Plotting.

C.3. Analyzing the Effect of Tuning Mixing Ratio α

In addition, recall that we can introduce a hyperparameter to control the mixing ratio, α between Coder and Reviewer and have the objective $(1-\alpha)\log p(x|y) + \alpha \log p(y|x)$. Coder-Reviewer reranking is equivalent to fixing α at 0.5 although the weighted objective is also popular in prior work (Li et al., 2016). To explore the benefits of tuning this additional hyperparameter of α, we grid search between $\{0.1, 0.2, \ldots, 0.9\}$ across all experimental settings. From Figure 8 in Appendix C.3, we observe that the improvement from tuning α is typically less than 1%, with a few exceptions (Incoder-6B on Plotting and CodeGen-16B on NL2Bash) where tuning can lead to more significant

	HumanEval	MBPP-S
Coder-Reviewer	56.7	**64.4**
N. Coder-Reviewer	**62.5**	61.6
Alternate	55.30	63.0
N. Alternate	50.73	62.4

Table 4. Ranking results of the alternate formulation of Maximum Mutual Information (Appendix C.4), which we observe underperforms Coder-Reviewer.

improvements. Given the lack of validation split in most benchmarks we experimented with, we do not include the result of tuning α as our main results. However, for practitioners who have access to validation data, we recommend that they can grid search over α in case tuning leads to additional gain.

C.4. Alternate Formulation

In Section 4, we show that Coder-Reviewer is related to pointwise mutual information regularization, when the regularization strength α is set to 0.5. There is another formulation of this objective that is also common in literature. With derivation shown in Appendix A, we can observe that

$$\begin{aligned}&\text{argmax}_y\ (1-\alpha)\log p(y|x) + \alpha \log p(x|y)\\ =\ &\text{argmax}_y\ \log p(y|x) - \alpha \log p(y).\end{aligned}$$

Here we compare to this alternate formulation (Alternate) on the 0-shot Python function completion datasets. We use prompting to estimate $p(y)$ by removing the docstring beneath the function header and record the probability of the function body. To apply this formulation with length normalization, we modify the regularization term as $\frac{1}{|y|}\log p(y)$. From Table 4 we observe that the alternate formulation generally underperforms Coder-Reviewer.

Exec. Filter	Method	Codex002	Codex001	Codex-Cushman	InCoder6B	InCoder1B	CodeGen16B	CodeGen6B	CodeGen2B
No	Random	53.7	34.7	38.8	19.9	11.6	28.1	20.4	20.8
	Reviewer	60.7	53.0	47.5	**35.5**	22.3	40.2	31.0	25.7
	Coder	55.4	47.8	43.2	14.4	8.8	31.0	21.9	23.2
	Coder-Reviewer	63.0	54.2	50.3	33.7	21.6	40.3	25.9	25.9
	N.Coder	57.1	44.8	39.4	20.8	9.6	28.9	21.8	23.6
	N.Coder-Reviewer	64.9	53.4	49.6	32.8	21.5	41.0	27.7	24.5
Yes	Random	57.6	46.8	41.0	21.9	13.4	36.7	25.8	28.9
	Reviewer	65.2	57.1	50.9	**35.5**	**30.3**	**46.5**	**32.6**	**32.9**
	Coder	57.7	51.1	45.8	15.5	8.9	38.6	25.6	30.1
	Coder-Reviewer	58.3	51.9	46.0	16.4	10.7	41.7	25.4	30.1
	N.Coder	59.4	47.8	41.8	22.6	11.0	37.4	26.3	28.8
	N.Coder-Reviewer	**68.0**	**57.1**	**52.0**	32.8	27.9	45.9	30.4	30.7
	MBR-EXEC	60.9	46.6	46.4	21.0	8.1	37.0	25.3	30.0

Table 5. Ranking results on the Plotting dataset. We observe that N.Coder-Reviewer and Reviewer alternate to be the best performing method. Executability filtering improves most methods and does not change the comparison between methods.

Exec. Filter	Method	Codex002	Codex001	Codex-Cushman	InCoder6B	InCoder1B	CodeGen16B	CodeGen6B	CodeGen2B
No	Random	44.0	34.6	32.5	14.1	8.2	30.2	25.1	22.6
	Reviewer	53.4	42.9	36.0	14.7	8.4	35.6	29.9	23.3
	Coder	38.7	29.0	30.4	18.5	10.5	31.6	27.5	25.1
	Coder-Reviewer	50.6	41.6	33.9	21.1	10.6	37.3	31.0	27.2
	N.Coder	56.5	44.2	39.7	17.6	9.2	36.1	29.8	26.0
	N.Coder-Reviewer	57.8	49.5	39.7	20.1	10.5	37.8	31.3	26.1
Yes	Random	49.2	38.1	35.1	16.0	9.5	32.2	26.4	24.2
	Reviewer	61.2	47.0	41.3	17.6	9.5	37.3	31.3	25.2
	Coder	45.1	32.7	33.0	20.1	10.8	33.5	28.3	26.3
	Coder-Reviewer	56.7	46.2	39.1	**23.5**	**12.0**	**40.0**	**32.5**	**28.5**
	N.Coder	60.2	47.2	42.1	19.2	10.2	38.5	30.9	26.6
	N.Coder-Reviewer	**62.5**	**52.1**	**43.8**	22.0	11.5	39.6	32.2	27.4
	MBR-EXEC	50.5	36.5	35.6	20.7	10.9	35.8	30.6	27.2

Table 6. Ranking results on the HumanEval dataset. We observe that N.Coder-Reviewer works the best on Codex model families and Coder-Reviewer works the best on CodeGen and Incoder models. Executability filtering improves most methods and does not change the comparison between methods.

Exec. Filter	Method	Codex002	Codex001	Codex-Cushman	InCoder6B	InCoder1B	CodeGen16B	CodeGen6B	CodeGen2B
No	Random	55.6	50.0	44.0	22.6	16.7	43.5	40.1	34.1
	Reviewer	56.3	51.0	45.4	25.6	17.7	42.0	41.3	38.3
	Coder	55.9	52.3	44.7	26.0	20.8	49.2	45.1	36.9
	Coder-Reviewer	61.5	56.9	51.0	31.7	23.4	46.1	44.9	41.5
	N.Coder	58.3	53.6	48.7	24.8	21.0	47.5	44.0	38.2
	N.Coder-Reviewer	59.6	55.6	51.6	28.8	20.0	44.4	43.2	40.1
Yes	Random	58.8	53.3	46.7	25.4	19.3	45.7	42.7	36.5
	Reviewer	59.5	54.7	48.2	28.8	21.0	43.6	43.5	40.3
	Coder	59.8	55.9	47.5	28.1	23.4	50.3	47.3	39.3
	Coder-Reviewer	**64.4**	**60.3**	**53.9**	**33.7**	25.8	47.4	47.1	43.3
	N.Coder	60.5	55.5	49.8	27.1	22.6	48.6	45.3	40.3
	N.Coder-Reviewer	61.6	57.7	53.1	31.1	22.6	45.8	44.7	41.9
	MBR-EXEC	63.9	59.4	50.7	30.9	**26.2**	**53.5**	**48.3**	**43.7**

Table 7. Ranking results on the MBPP-S dataset. We observe that Coder-Reviewer works the best on Codex model families and MBR-EXEC is usually the best on CodeGen and Incoder models. Executability filtering improves most methods and usually does not change the comparison between methods.

Exec. Filter	Method	Codex002	Codex001	Codex-Cushman	InCoder6B	InCoder1B	CodeGen16B	CodeGen6B	CodeGen2B
No	Random	53.6	46.9	35.1	14.8	9.1	33.5	28.5	24.1
	Reviewer	63.3	53.1	41.5	20.8	13.2	40.5	32.7	28.4
	Coder	55.4	49.8	38.8	19.7	12.3	36.1	32.7	28.9
	Coder-Reviewer	62.6	55.2	45.0	23.3	14.8	42.2	34.6	30.5
	N.Coder	55.5	50.6	35.1	15.9	9.3	36.9	32.4	27.9
	N.Coder-Reviewer	62.5	54.5	42.2	20.9	13.2	41.4	33.4	29.3
Yes	Random	58.1	51.6	40.8	19.6	13.1	39.3	34.2	28.2
	Reviewer	**66.9**	57.4	46.7	24.4	15.7	44.1	39.4	31.7
	Coder	60.2	55.1	44.7	23.4	16.0	41.5	37.9	31.9
	Coder-Reviewer	66.4	**59.4**	**48.9**	26.1	17.1	46.2	40.3	33.2
	N.Coder	60.7	54.8	40.8	20.2	13.3	41.9	37.5	31.0
	N.Coder-Reviewer	66.2	58.4	46.3	24.1	15.8	45.4	39.8	32.3
	MBR-EXEC	63.0	58.6	48.3	**26.7**	**18.3**	**47.3**	**41.1**	**35.5**

Table 8. Ranking results on the MBPP dataset. We observe that Coder-Reviewer usually works the best on Codex model families; MBR-EXEC is usually the best on CodeGen and Incoder models and Coder-Reviewer is usually the second best performing method. Executability filtering improves most methods and usually does not change the comparison between methods.

Exec. Filter	Method	Codex002	Codex001	Codex-Cushman	InCoder6B	InCoder1B	CodeGen16B	CodeGen6B	CodeGen2B
No	Random	60.0	55.2	55.2	49.7	41.0	35.6	33.2	25.4
	Reviewer	63.3	58.4	59.9	55.4	43.3	28.3	29.4	19.5
	Coder	57.4	48.0	50.8	48.7	42.6	31.3	25.2	25.6
	Coder-Reviewer	61.9	57.7	58.4	55.0	50.8	37.3	**40.3**	**37.0**
	N.Coder	61.0	49.9	48.2	40.8	37.5	36.8	32.9	25.4
	N.Coder-Reviewer	**63.7**	59.9	60.0	**56.0**	42.8	29.1	31.0	20.8
Yes	Random	60.0	55.2	55.3	49.8	41.1	35.7	33.2	25.7
	Reviewer	63.3	58.5	60.2	55.3	43.3	28.1	29.2	19.9
	Coder	57.4	48.1	50.9	48.8	42.6	31.7	25.4	25.9
	Coder-Reviewer	61.9	57.8	58.5	55.0	**50.9**	37.0	39.8	36.7
	N.Coder	61.3	49.9	48.8	41.5	38.0	**37.8**	33.1	25.6
	N.Coder-Reviewer	63.7	**59.9**	**60.4**	55.9	43.1	29.5	30.8	21.1
	MBR-EXEC	57.4	48.3	51.0	48.8	42.6	32.4	25.8	26.3

Table 9. Ranking results on the NL2Bash dataset. We observe that Coder-Reviewer works the best on Codex model families and MBR-EXEC is usually the best on CodeGen and Incoder models. Executability filtering is implemented with simulated execution via parsing the generated bash code and does not lead to a consistent improvement.

Exec. Filter	Method	Codex002	Codex001	Codex-Cushman	InCoder6B	InCoder1B	CodeGen16B	CodeGen6B	CodeGen2B
No	Random	62.8	46.0	40.2	19.0	11.7	13.6	21.1	11.9
	Reviewer	62.9	50.0	44.2	27.3	16.8	16.0	22.4	13.1
	Coder	71.5	56.8	51.8	32.4	21.4	27.6	30.4	21.5
	Coder-Reviewer	71.6	56.5	52.8	32.8	22.1	23.8	28.6	17.4
	N.Coder	67.4	56.5	44.0	29.5	17.2	22.4	26.9	17.0
	N.Coder-Reviewer	68.0	54.7	48.3	30.7	20.6	18.9	24.8	14.9
Yes	Random	65.2	57.1	50.8	29.4	18.8	25.6	32.7	22.4
	Reviewer	67.5	60.0	54.7	38.4	26.8	28.8	36.8	23.1
	Coder	74.1	64.3	58.8	38.9	26.9	**33.7**	37.3	**27.5**
	Coder-Reviewer	74.5	**65.3**	**60.4**	**41.5**	**29.8**	31.7	**38.8**	24.8
	N.Coder	69.9	63.2	54.1	38.2	26.8	31.1	36.8	25.7
	N.Coder-Reviewer	71.0	62.6	57.6	40.3	29.0	29.9	37.3	23.6
	MBR-EXEC	**75.2**	63.2	59.0	38.2	27.3	30.6	37.0	26.1

Table 10. Ranking results on the Spider dataset. We observe that Coder-Reviewer usually works the best. Executability filtering improves most methods and usually does not change the comparison between methods.