Self-consistency for open-ended generations

Siddhartha Jain*
AWS AI Labs
siddjin@amazon.com

Xiaofei Ma
AWS AI Labs
xiaofeim@amazon.com

Anoop Deoras
AWS AI Labs
adeoras@amazon.com

Bing Xiang
AWS AI Labs
bxiang@amazon.com

Abstract

In this paper, we present a novel approach for improving the quality and consistency of generated outputs from large-scale pre-trained language models (LLMs). Self-consistency has emerged as an effective approach for prompts with fixed answers, selecting the answer with the highest number of votes. In this paper, we introduce a generalized framework for self-consistency that extends its applicability beyond problems that have fixed-answer answers. Through extensive simulations, we demonstrate that our approach consistently recovers the optimal or near-optimal generation from a set of candidates. We also propose lightweight parameter-free similarity functions that show significant and consistent improvements across code generation, autoformalization, and summarization tasks, even without access to token log probabilities. Our method incurs minimal computational overhead, requiring no auxiliary reranker models or modifications to the existing model.

1 Introduction

The rapid advancement and remarkable achievements of large-scale pre-trained language models (LLMs) have brought about a revolutionary transformation in the field of natural language processing (NLP). These models have demonstrated significant enhancements in various NLP applications, such as machine translation, summarization, and code generation. However, it is important to note that the quality of generated outputs can exhibit considerable variability. Although individual generations sampled from the models often yield high-quality results, multiple samplings can produce certain generations of substantially higher quality than the average output of the model.

Several approaches can be employed to address this issue. One strategy involves improving the underlying models themselves. . This can be achieved by enriching the training dataset with higher quality generations, as suggested by recent studies [Hsieh et al., 2023, Ouyang et al., 2022, Ho et al., 2022, Polu et al., 2022, Liu and Liu, 2021]. Another approach is to maintain the integrity of the underlying model while employing a process known as *reranking* to prioritize and select highly ranked generations based on specific criteria [Ravaut et al., 2022, Jiang et al., 2022b, Zhang et al., 2022, Chen et al., 2021, Shi et al., 2022, Li et al., 2022b, Mizumoto and Matsumoto, 2016]. However, it is worth noting that most reranking techniques involve computationally intensive or cumbersome methods to calculate the selection criterion, which can include training an auxiliary model as a reranker or evaluating the probability of the query given the generated answer – the latter of which doubles the inference cost. In case of code generation models, it can also involve executing the generated code on unit tests which can get quite complex and may not be feasible for a lot of use cases, especially as you move beyond the contest coding setting.

*Corresponding author

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

Recently for the special case of problems that have fixed answer, a simple approach, called self-consistency was suggested for selecting the best answer from multiple genrations Wang et al. [2022]. In that paper, the authors sample multiple generations from the LLM, extract the predicted answer from each generation and select the answer with the most number of votes. In their paper, the authors propose a method wherein they sample multiple generations from the LLM and extract the predicted answer from each generation. Subsequently, they select the answer with the highest number of votes, thus achieving substantial improvements over existing baselines, including the widely used approach of ranking generations based on the log probability. However, it is important to note that the self-consistency approach is not applicable to prompts that are open-ended and do not have fixed answers. This limitation becomes particularly relevant in scenarios such as code generation, where multiple implementations of the same function may be valid, or in open-ended text generation, where multiple phrasings can convey the same meaning. The self-consistency method relies on the presence of a clear majority or consensus among the generated answers, which may not exist in these open-ended situations. Consequently, alternative strategies need to be explored to address the challenges posed by such prompt types.

An alternative perspective on self-consistency can be achieved by considering a similarity function that compares different generations and reranks them based on their average similarity to other generations. In the case of self-consistency, the similarity function takes a binary form, indicating whether the answers of two generations are identical. However, this binary similarity function is limited to prompts that have fixed answers. In this work, we develop a framework that formally defines the concept of an optimal generation. We demonstrate that the aforementioned viewpoint on self-consistency allows us to identify the optimal generation within a given set. In simulated scenarios, we provide evidence that our framework is capable of recovering the best or near-best generation in many cases. Additionally, we develop lightweight similarity functions that are suitable for open-ended generation tasks, thereby expanding the applicability of self-consistency to such domains. Furthermore, we demonstrate that the reranking methods utilized in previous works Shi et al. [2022], Li et al. [2022b] can also be understood within the same conceptual framework.

Concretely, our contributions are as follows

- We propose a generalized framework for self-consistency that extends its applicability beyond prompts with fixed answers. By formally defining the concept of an optimal generation for open-ended generations, we are able to show that our framework is capable of recovering the optimal generation if it exists within the set of generations. Through extensive simulations, we show that our approach consistently identifies the best or near-best generation from a set of candidate generations.
- We introduce multiple lightweight similarity functions that require no additional parameters. These functions are evaluated across various tasks, including code generation, autoformalization, and summarization, using six different models. Our evaluations reveal consistent and significant improvements over baseline methods. Notably, one of our similarity functions only relies on the raw generations from the model, making it particularly relevant in situations where token log probabilities are not accessible, as is the case with certain proprietary models like OpenAI's chat models' API.
- Leveraging the pairwise similarity nature of our reranking scheme, we enhance model generation for code generation tasks, particularly when the evaluation metric is $pass@k$ for $k > 1$. This enables us to improve the overall quality of generated code.
- To gain insights into the effectiveness of our similarity function, we conduct various ablation experiments. These experiments allow us to analyze and understand the underlying mechanisms that contribute to the success of our approach.

The rest of the paper is organized as follows. In Section 2 we present our motivation. In Section 3 we present our method and the similarity function. In Section 4, we present and discuss our experimental results. In Section 5, we describe the related work and we finally conclude in Section 6.

2 Motivation

What constitutes a good generation? In the context of code generation, for instance, one metric might be the number of passed unit tests, which provides an approximation of correctness. Additional criteria could be the readability or the computational optimality of the code, as judged by human evaluators. For language tasks, we could similarly establish a set of predicates to evaluate output quality, such as fluency, avoidance of content hallucination, or correctness of responses to questions with fixed answers.

To formalize this, imagine a vector $\mathbf{v}$ of length k, where each element represents a categorical variable. We also have n people who each hold a personal estimate $\mathbf{u}_i$ of $\mathbf{v}$. The only accessible information we have is the pairwise fractional agreement, denoted as $a(\mathbf{u}_i, \mathbf{u_j}) = \frac{1}{k}\sum_{t=1}^{k} \mathbb{I}(\mathbf{u}_i^t = \mathbf{u}_j^t) \forall i, j \in [1, n]$ where i indexes the generations and t the predicates. Our aim is to identify a person i such that $a(\mathbf{u}_i, \mathbf{v})$ is maximized. In this formulation, the elements of $\mathbf{v}$ correspond to the different predicates we wish the generation to satisfy, while the people correspond to different generations.

We only assume access to the fractional agreement rather than to the underlying estimates of $a(\mathbf{u}, \mathbf{v})$ as that we may not have the resources or knowledge to evaluate any or all of the predicates on the different generations at inference time. For example in context of code, we may not have the ability to execute unit tests – either due to latency or security constraints, or lack of an appropriate build system, or unavailability of unit tests. For predicates like content hallucination, might be required, which is not generally feasible at inference time. Consequently, we must resort to approximating the agreement using proxy similarity functions.

We introduce a *self-consistency* assumption stating that, for each individual predicate, the most frequent response is assumed to be correct. Formally if $\mathbf{v}^l$ can take on m_l values $1, \ldots, m_l$ and without loss of generality, $\mathbf{v}^l = 1$, then $1 = \arg\max_j \sum_{i=1}^{n} \mathbb{I}(u_i^l = j)$.

Given this problem formulation and selection criterion, we can establish the following:

Theorem 2.1. *For $k = 1$, we always recover the best $\mathbf{u}$. However for $k > 1$, it is not guaranteed.*

Moreover:

Theorem 2.2. *If there exists $\mathbf{u}_b = v$, then $b = \arg\max_i \frac{1}{n-1}\sum_{i \neq j} a(\mathbf{u}_i, \mathbf{u_j})$.*

Informally this says that if a generation g exists such that its predicate vector perfectly aligns with the target vector, selecting the generation with the highest average fractional agreement with other generations will pick g.

Now if we assume that $\mathbf{u}_i^j$ are iid from $Bernoulli(p_j)$, then we can show that

Theorem 2.3. $\sum_{j=1}^{k} p_i - \sqrt{\frac{k \log k}{2}} \leq \mathbb{E}[\sum_{j}^{k} \mathbf{u}_b^j] \leq \sum_{j=1}^{k} p_i + \sqrt{\frac{k \log k}{2}}$

where $\mathbf{u}_b$ denotes the sequence selected by our method.

All proofs for these theorems are presented in the Supplement. To further substantiate our selection criterion—picking the generation with the highest average fractional agreement with all other generations—we conducted a simulation. Here, we randomly generated $\mathbf{v}$ and $\mathbf{u}$ and evaluated the optimality of our criterion. Our findings suggest that for various k values, our method successfully recovers the best generation the majority of the time, significantly outperforming random selection. Moreover, on average, the generation we recover demonstrates nearly 100% agreement with best generation, even in cases where we do not select the best generation. The full details are in the Supplement.

3 Method

As previously mentioned, we may not have the capability to compute predicates at inference time, thereby rendering the computation of the exact fractional agreement with $\mathbf{v}$ i.e. $a(\mathbf{u}, \mathbf{v})$, unattainable. As a result, we need to resort to a surrogate similarity function. To this end, we define a generalized self-consistency score $GSC_{Sim}(i)$ for each generation i, given by $\frac{1}{M-1}\sum_{j=1, j \neq i}^{M} Sim(i, j)$. Here, Sim denotes the similarity function, and M represents the number of generations.

For generations with fixed answers, fi we have

$$Sim(i, j) = \mathbb{I}(\texttt{Answer in generation } i \texttt{ is an exact match with Answer in generation } j)$$

this is equivalent to the self-consistency criterion. Two other reranking methods - MBR-Exec [Shi et al., 2022] and AlphaCode [Li et al., 2022b] - can be viewed in terms of the same formulation with the difference being that of the similarity function. MBR-Exec *executes* model generated code. It then defines gives a similarity score of 1 if a pair of programs agree on all unit tests and 0 otherwise[2]. For each program, they sum the similarity vs all other programs and pick the program with the highest similarity. Similarly AlphaCode clusters its generated programs by executing them on test cases and selecting a program from the largest cluster – with two programs Centroid together if they agree on on all test cases. This is conceptually equivalent to what MBR-Exec does. We give further evidence that this is a useful way to frame self-consistency by evaluating another OpenAI Ada embedding based similarity function (Section F in the Supplement). While its performance is promising, as the similarity function is a lot more heavyweight requiring a separate embedding model, we chose not to explore it further.

One straightforward way to encode a generation is by using a binary vector that denotes the presence or absence of an n-gram. Surprisingly, we find this simple encoding to be sufficient for defining a robust similarity function. For open-ended generation, we define our similarity function as follows. For each generation we define a vector $\mathbf{v}$ of size $|V|$ where V is set of all possible n-grams for $n = 1$ to $n = K$ where K is a hyperparameter. For the experiments in this paper, we simply use $K = 1$. We show in Section 4.6, increasing K can be helpful though only up to a point. Each element i of $\mathbf{v}$ is simply whether token i is present in the generation or not. We then take the inner product between two such vectors as similarity. We call this the Ngram consistency score (NCS) and refer to the $K = 1$ version as the Unigram consistency score (UCS). Formally

$$UCS(i, j) = \frac{1}{|V|}\mathbf{v}_i \cdot \mathbf{v}_j$$

where

$$\mathbf{v}_i^j = \mathbb{I}(t_j \in g_i)$$

where t_j is the jth token and g_i the ith generation. This definition only requires model generations and incurs minimal computational overhead – we only need to compute the unigram overlap instead of training an auxiliary model, running generated programs, or performing additional inferences using the same model (which will increase compute cost as well as latency). Notably, we don't normalize the inner product by the norm of the vectors. This is a deliberate design choice that encourages more diverse sequences, in response to known issues of neural generation models producing degenerate and repetitive sequences Zhang et al. [2022], Welleck et al. [2019]. We delve into this topic in Section G in the Supplement.

When token probabilities are available, we can leverage them to improve our approach. Intuitively, if a generation has a low token probability for the generated token, then finding a match for that that token should count for less. In accordance with this intuition, we introduce two further variants. First we modify the definition of $\mathbf{v}$ as follows

$$\mathbf{v}_i^j = \begin{cases} \frac{1}{c_j^i} \sum_k^{c_j^i} p(t_j^{i,k}) & \text{if } t_j \in g_i, \\ 0 & \text{otherwise} \end{cases}$$

where c_i^j is the number of times token t_j appears in generation i and $p(t_j^{i,k})$ is the token probability of the jth token's kth appearance in generation i. We call this the weighted n-gram consistency score (WUCS).

The mean log probability of a sequence is an oft-used ranking method. We can combine it with WUCS by further weighting each generation by the per token probability as follows – for a generation i, Consensus-WUCS $= WUCS \cdot e^{(1/|g_i|) \cdot p(g_i)}$ where g_i is the length of generation i.

Finally, to rank the generations, we employ $\arg\max_i GSC_{Sim}(i)$ where Sim can take the form of UCS, WUCS, or Consensus-UCS.

[2] They define it in terms of a loss function but taking 1-the loss function is equivalent to the similarity function we describe above

3.1 Extending to ranked $pass@k$

A common evaluation metric for code generation problems is ranked $pass@k$ wherein we assess whether *any* program among the top k selected programs (selected from a larger set) can pass all the given unit tests for that problem. Typically, the top k generations are selected based on a predetermined ranking. However, with our similarity-based metric, we can apply a more nuanced approach.

For a particular problem, if the highest-ranked generation for a specific prompt is correct, we have already succeeded. We would only need to utilize the remaining generations in our k-budget if the top-ranked generation does not pass some unit test case. In this event, we could consider the top-ranked generation as a hard negative and select the next generation that exhibits lower similarity to the top-ranked generation.

More specifically, if we have selected programs $S_{k'}$ so far ($|S_{k'}| = k' < k$, then we modify the GCS function to select the $k' + 1$th item in the list. In particular, we compute

$$GCS_{Sim}^{ranked} = \frac{1}{M-1}(\sum_{j \notin S_{k'}} Sim(i,j) - \sum_{j \in S_{k'}} Sim(i,j))$$

Note that for $k = 1$, GCS and GCS^{ranked} are equivalent. We demonstrate in Section 4.8 that GCS_{Sim}^{ranked} performs significantly better in ranking for $pass@k$ where $k > 1$ than raw GCS. This approach leads to a more efficient utilization of the ranked generations, improving the overall effectiveness of the code generation task.

4 Results

We conduct experiments utilizing the Codex family of models, specifically Codex-davinci-001, Codex-davinci-002, and Codex-Cushman as well as Llama family of models. In addition we also evaluate GPT-J for Xsum and MiniF2F. We evaluate these models on a range of datasets for code generation tasks – in particular on the HumanEval [Chen et al., 2021], MBPP, MBPP-sanitized [Austin et al., 2021] datasets for code generation. For the autoformalization of MiniF2F to Isabelle, we use the dataset provided by [Jiang et al., 2022a]. For text summarization, we utilize the Xsum dataset [Narayan et al., 2018].

Our primary evaluation metric for code generation is ranked $pass@1$ where we rerank a sample set of generations and assess whether the top-ranked generation successfully passes all unit tests. We also evaluate with ranked $pass@k$ for $k > 1$. For the MiniF2F autoformalization task, we measure the quality using the BLEU score, following Wu et al. [2022]. For Xsum we use the Rouge-2 and Rouge-L scores for evaluation. For all code generation datasets, we sample 125 generations from the models which serves as our dataset for the different experiments For MiniF2F and Xsum, we sample 50 generations from the model. Unless otherwise specified, for all experiments, we use the Codex-davinci-002 model. Following Shi et al. [2022], Zhang et al. [2022], we perform bootstrap sampling 50 times with a sample size of 25 to generate the results.

Our baselines are Random selection, Ranking by mean log probability, Ranking using Centroid in our confidence weighted unigram space, and for code generation - ranking using the Coder Reviewer Ranker method [Zhang et al., 2022]. A full description of the datasets, experiments, and the baselines is in the Supplement.

4.1 UCS scores are higher for correct answers

As a sanity check, we first evaluate whether the UCS scores are indeed higher for the correct generations [3] The results are in Table 4 in the Supplement. The ratios are consistently > 1 for all models except for the UL2-20B model for which they still remain very close to 1.

[3] We used the generations in Li et al. [2022b] provided by them as part of their Supplementary Material.

	Random	Centroid	Mean-logp	UCS	WUCS	Consensus-WUCS
	HumanEval					
Codex002	0.435	0.437	0.539	0.539	**0.558**	**0.568**
Codex001	0.345	0.354	0.408	0.402	**0.426**	**0.445**
Code-Cushman	0.311	0.335	0.355	0.353	**0.373**	**0.381**
Llama-13B	0.142	0.17	0.17	0.177	**0.187**	**0.192**
Llama-30B	0.207	0.225	0.228	0.257	**0.263**	**0.267**
	MBPP-S					
Codex002	0.55	**0.583**	0.57	0.572	0.580	**0.589**
Codex001	0.494	0.532	0.515	0.523	**0.535**	**0.546**
Code-Cushman	0.436	0.467	0.456	0.457	**0.472**	**0.488**
Llama-13B	0.247	**0.284**	0.27	0.261	0.266	**0.277**
Llama-30B	0.325	0.357	0.348	0.253	**0.363**	**0.373**
	MBPP					
Codex002	0.536	0.563	0.512	0.58	**0.587**	**0.594**
Codex001	0.475	0.505	0.503	0.505	**0.520**	**0.525**
Code-Cushman	0.305	0.343	0.319	0.386	**0.405**	**0.420**
Llama-13B	0.185	**0.202**	0.197	0.183	0.195	**0.199**
Llama-30B	0.262	0.276	0.273	0.276	**0.287**	**0.294**

Table 1: Accuracy of generated code for HumanEval, MBPP, MBBP-S. Best results are colored in **first**, **second**.

4.2 UCS shows strong improvements for Code Generation

As showcased in Tables 1 and 2, the application of the UCS, WUCS, and Consensus-WUCS methods leads to substantial improvements in the accuracy as well as mean reciprocal rank of code generation across various models and datasets.

In the HumanEval dataset, UCS variants consistently outperform the traditional methods, namely Random and mean log probability. For instance, the Codex002 model exhibits a substantial accuracy improvement from 0.435 (Random) to 0.568 (Consensus-WUCS). Even the less performing models, such as Llama-13B and Llama-30B, exhibit noticeable accuracy gains when our proposed methods are employed.

Similar trends are observed in the MBPP-S and MBPP datasets. UCS, WUCS, and Consensus-WUCS consistently improve the accuracy across all models. Specifically, the Consensus-WUCS method consistently dominates Random and mean log probability ranking in all categories, and almost always outperforms WUCS as well. Of particular note is the performance of WUCS, which surpasses the mean log probability method in every model and dataset combination. In fact it is the best method for all dataset and model combinations except LLama-13B model for MBBP and MBPP-S. UCS, which does not require token probabilities and relies only on the generations, also demonstrates a consistent superiority over the random reranking.

Consensens-WUCS and WUCS are also almost always better than the Centroid based approach with Consensus-WUCS outperforming it 13/15 times. A discussion of the mean reciprocal ranking performance is deferred to the Supplement but the trend is similar.

4.3 UCS shows consistent improvements for open-ended generation

The performance of UCS, WUCS, and Consensus-WUCS on the Xsum and MiniF2F datasets demonstrates the versatility and efficacy of our proposed methods across varied tasks and models. The results, shown in Table 3, paint a promising picture for these reranking techniques.

In the case of the MiniF2F dataset, evaluated using the BLEU metric, Consensus-WUCS outperforms all other methods for the Codex002 model except for Centroid. For the Llama-13B, Llama-30B, and GPT-J models, the top performers are closely matched, with Consensus-WUCS, WUCS, and UCS all delivering competitive scores.

Turning to the Xsum dataset, we see a similar trend. For the Rouge-2 metric, Consensus-WUCS achieves the highest score for the Codex002 and both LLama models, and ties for the best score with WUCS for the Llama-13B model. In the GPT-J model, UCS performs slightly better than the WUCS and Consensus-WUCS. Nonetheless, all these methods surpass Random, and Mean-logp reranking methods and almost always surpass Centroid.

	Centroid	Mean-logp	UCS	WUCS	Consensus-WUCS
HumanEval					
Codex002	0.515	0.604	0.615	**0.630**	**0.633**
Codex001	0.432	0.484	0.488	**0.507**	**0.517**
Code-Cushman	0.4	0.428	0.434	**0.451**	**0.454**
Llama-13B	0.231	0.221	0.242	**0.248**	**0.25**
Llama-30B	0.29	0.286	0.324	**0.327**	**0.327**
MBPP-S					
Codex002	0.64	0.626	**0.67**	0.643	**0.647**
Codex001	0.594	0.575	0.594	**0.599**	**0.605**
Code-Cushman	0.527	0.521	0.531	**0.541**	**0.549**
Llama-13B	**0.355**	0.331	0.340	0.344	**0.347**
Llama-30B	0.425	0.408	0.337	**0.436**	**0.438**
MBPP					
Codex002	0.631	0.549	0.651	**0.655**	**0.659**
Codex001	0.574	0.58	0.587	**0.596**	**0.598**
Code-Cushman	0.435	0.29	0.479	**0.494**	**0.503**
Llama-13B	0.269	0.3	0.261	**0.305**	**0.304**
Llama-30B	0.346	0.332	0.351	**0.358**	**0.359**

Table 2: Mean reciprocal rank of generations for HumanEval, MBPP, MBBP-S. Best results are colored in **first**, **second**.

	Centroid	Random	Mean-logp	UCS	WUCS	Consensus-WUCS
MiniF2F						
Codex002	**0.582**	0.558	0.529	0.556	0.558	**0.562**
Llama-13B	**0.249**	0.243	0.242	0.246	0.247	**0.248**
Llama-30B	**0.264**	**0.26**	0.256	0.256	0.257	0.257
GPT-J	**0.248**	0.242	0.24	0.247	**0.248**	**0.248**
Xsum Rouge2						
Codex002	**0.218**	0.197	0.214	0.21	0.215	**0.219**
Llama-13B	0.103	0.092	0.103	0.104	**0.106**	**0.106**
Llama-30B	0.12	0.107	0.122	0.121	**0.122**	**0.123**
GPT-J	0.069	0.065	0.066	**0.071**	**0.07**	0.069
Xsum RougeL						
Codex002	**0.363**	0.339	0.351	0.348	0.353	**0.356**
Llama-13B	0.207	0.196	0.203	0.209	**0.21**	**0.209**
Llama-30B	0.227	0.214	0.228	0.23	**0.231**	**0.231**
GPT-J	0.175	0.172	0.166	**0.18**	**0.178**	0.175

Table 3: Performance on Xsum and MiniF2F datasets. Best results are colored in **first**, **second**.

With the Rouge-L metric, UCS variants show the best performance for the all models except Codex002. For the Llama-30B model, WUCS and Consensus-WUCS share the top spot, while UCS achieves the best score for the GPT-J model. Once again, these methods generally outperform Centroid, Random, and Mean-logp reranking methods.

In total, Consensus-WUCS gets the top spot in 5/12 comparisons, WUCS in 4/12, UCS in 2/12, and Centroid in 5/12 primarily due to MiniF2F.

4.4 UCS variants are competitive with Code Reviewer Ranker

The comparison with the Code Reviewer Ranker baseline, specifically with the Normalized Reviewer (NR) and Normalized Coder-Reviewer (NCR) variants, is in Table 5 (Supplement). As the state of the art in code reranking, these methods represent a strong baseline. Our results demonstrate that the WUCS and Consensus-WUCS methods are highly competitive, often outperforming both NR and NCR, despite the fact that NR and NCR require a second forward pass, which doubles the inference cost and adds latency overhead. A fuller discussion of the results is in the Supplement.

4.5 Improvements are consistent across different generation temperatures

In Figure 2 (Supplement) we show how UCS reranking behaves for MBPP as the decoding sampling temperature increases. While accuracy can vary across temperatures, the ranking of the different methods remains consistent. Consensus-WUCS dominates in terms of accuracy for most of the temperature regimes until you hit the temperature of 1. Importantly, for lower temperatures where we get the best results, Both Consensus-WUCS as well as WUCS get the best accuracy. While just UCS is on par with mean log-probability ranking until a temperature of 0.4 after which it falls behind, we note that UCS does not use any probability information about the generation and thus a fair comparison would be to that of random ranking which it is consistency better than for almost the entire temperature range.

4.6 Varying the maximum n-gram length does not change results

As mentioned in Section 3, UCS only considers unigrams. Here we consider Ngram Consistency Score – the more generalized version. To account for the fact that a sentence will have fewer n-grams, the more n increases, we multiply $p(t_j^{i,k})$ by $\frac{|g_i|}{|g_i|-|t_j^{i,k}|-1}$ where $t_j^{i,k}$ is now the kth appearance of the jth n-gram in the ith generation. In Figure 3 (Supplement), we show how the ranking behaves as the n increases. As can be seen, while there is a slight improvement going from $n = 1$ to $n = 4$, the improvement flattens after that point. 4-grams is also what is conventionally used when computing BLEU score so it is interesting that the same value ends up being optimal in the drastically different setting of code generation with each word being a token instead of an English word.

4.7 Increasing number of samples maintains reranking strength

In Figure 4 (Supplement), we show how the performance changes for MBPP and Xsum as the number of samples increases. All variants of UCS are able to maintain accuracy (although Consensus-WUCS sees a drop in the beginning for Xsum but maintains its performance subsequently) even as the number of samples increases from 5 to 100. Meanwhile, the mean log probability ranking drastically declines in terms of accuracy, quickly falling below even random selection. This is likely due to the tendency of mean log probability ranking to choose degenerate sequences Holtzman et al. [2019] which UCS variants seem to be able to avoid.

4.8 GCS^{ranked} comparison

In Figure 1, we show how the model performance changes as k for $pass@k$ increases. We compare GCS vs GCS^{ranked}. While the performance of GCS declines quickly, GCS^{ranked} maintains good performance even at larger values of k for all code generation datasets.

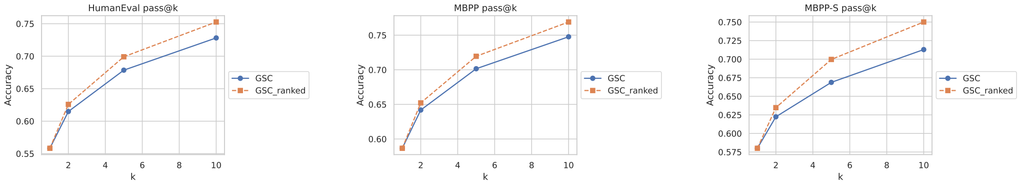

Figure 1: $pass@k$ for $k > 1$ for HumanEval, MBPP, MBPP-S

5 Related Work

5.1 Advanced decoding

There are also several advanced decoding methods to improve model generation quality. In Vijayakumar et al. [2016], the log probabilities of tokens are adjusted depending on their similarity to previously chosen words. However this method requires generations to be done jointly with coordination. This can cause significant infrastructure and latency overhead if the model itself is distributed across multiple GPUs or machines which is far from unusual for LLMs. In Li et al. [2022a] they use an auxiliary LLM model to contrast the token probabilities against the primary LLM and use them for decoding. In Su et al. [2022], they have a degeneration penalty to penalize generation of already generated tokens. A significant caveat of all such methods is that they also require access to the decoding procedure of the LLM. You cannot just take the generations from an LLM API. Our approach also bears some similarity to Mangu et al. [2000] where they compute a consensus hypothesis by doing multiple alignment of sentence lattices.

5.2 Auxiliary reranker

In Mizumoto and Matsumoto [2016], they use a perceptron based reranker to rerank model generated translations. SummaReranker [Ravaut et al., 2022] use mixture of experts training to train their reranker to optimize for multiple automated evaluation metrics (like ROUGE or BLEU score) at once. PairReranker [Jiang et al., 2022b] uses automated evaluation metrics to rank model generations and then select the top few best and worse and train a model to classify the better summary between a pair of summaries. All of the previous reranking methods however require training an auxiliary model.

5.3 Code generation reranking

There have also been multiple reranking proposals for code generation in particular. A unique characteristic of code (as oppposed to text) is that code can be executed. Thus several methods have tried to exploit that property for reranking. MBR-Exec [Shi et al., 2022] and AlphaCode [Li et al., 2022b] both execute the generated codes on unit tests. They rank the different codes according to how many other codes are semantically equivalent to them (i.e. have the same results on the given unit tests). CodeT [Chen et al., 2022] uses LLMs to generate both code and candidate unit tests. They then find sets of generated codes such that the product of the size of the set and the size of the *unit test* set the codes agree on is maximized. More recently, Coder-Reviewer Ranker [Zhang et al., 2022] applies the well known Maximum Mutual Information objective Li et al. [2015] to code generating LLMs by using the strong few shot and zero prompting capabilities of LLMs to obtain the query likelihood.

6 Conclusion and Future work

We analyze the self-consistency method for problems that have fixed answers and develop a framework to extend it to open-ended generations. We establish connections between our framework and other code generation reranking functions and prove that if the optimal generation is present in our generation set, we can always recover it as well as prove bounds on how close we can get to the optimal generation under certain settings.

Our simulated tests reveal our ability to consistently recover the best or close to best possible generation in the set. We introduce several lightweight similarity functions and show that they give strong and consistent improvements over state of the art baselines. Notably, our Unigram Consistency Score (UCS) function, the most minimal of our similarity functions, requires only access to raw generations to effectively rerank. We show that the UCS variants uniformly enhance the performance of code and text generation and are competitive with strong baselines like Coder Reviewer Reranker despite them needing a lot more compute resources as well as time. For code geneartion, we also leverage the fact that our reranking metric is based on pairwise similarity to improve performance for pass@k for $k > 1$. Additionally, we conduct multiple variations on our primary experiments to ascertain the robustness and reliability of our performance.

7 Broader Impact and Limitations

As a paper that tries to improve the performance of Large Language Models (LLMs), it inherits the risk and rewards of LLMs in general. LLMs have shown themselves highly relevant and useful for a number of tasks but in particular code generation. Our method shows particularly strong improvements for that task and thus we hope will have a broad impact. Nevertheless, we did not evaluate our method on whether it increases its propensity to select biased or toxic generations which we leave to future work.

8 Acknowledgements

We would like to thank Danica Sutherland and Ameya Velingker for useful discussions regarding the project.

References

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

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

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

N. Ho, L. Schmid, and S.-Y. Yun. Large language models are reasoning teachers. *arXiv preprint arXiv:2212.10071*, 2022.

A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

C.-Y. Hsieh, C.-L. Li, C.-K. Yeh, H. Nakhost, Y. Fujii, A. Ratner, R. Krishna, C.-Y. Lee, and T. Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*, 2023.

A. Q. Jiang, S. Welleck, J. P. Zhou, W. Li, J. Liu, M. Jamnik, T. Lacroix, Y. Wu, and G. Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022a.

D. Jiang, B. Y. Lin, and X. Ren. Pairreranker: Pairwise reranking for natural language generation. *arXiv preprint arXiv:2212.10555*, 2022b.

J. Li, M. Galley, C. Brockett, J. Gao, and B. Dolan. A diversity-promoting objective function for neural conversation models. *arXiv preprint arXiv:1510.03055*, 2015.

X. L. Li, A. Holtzman, D. Fried, P. Liang, J. Eisner, T. Hashimoto, L. Zettlemoyer, and M. Lewis. Contrastive decoding: Open-ended text generation as optimization. *arXiv preprint arXiv:2210.15097*, 2022a.

Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097, 2022b.

Y. Liu and P. Liu. Simcls: A simple framework for contrastive learning of abstractive summarization. *arXiv preprint arXiv:2106.01890*, 2021.

L. Mangu, E. Brill, and A. Stolcke. Finding consensus in speech recognition: word error minimization and other applications of confusion networks. *Computer Speech & Language*, 14(4):373–400, 2000.

T. Mizumoto and Y. Matsumoto. Discriminative reranking for grammatical error correction with statistical machine translation. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1133–1138, 2016.

S. Narayan, S. B. Cohen, and M. Lapata. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745*, 2018.

L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

S. Polu, J. M. Han, K. Zheng, M. Baksys, I. Babuschkin, and I. Sutskever. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*, 2022.

M. Ravaut, S. Joty, and N. F. Chen. Summareranker: A multi-task mixture-of-experts re-ranking framework for abstractive summarization. *arXiv preprint arXiv:2203.06569*, 2022.

F. Shi, D. Fried, M. Ghazvininejad, L. Zettlemoyer, and S. I. Wang. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*, 2022.

Y. Su, T. Lan, Y. Wang, D. Yogatama, L. Kong, and N. Collier. A contrastive framework for neural text generation. *arXiv preprint arXiv:2202.06417*, 2022.

A. K. Vijayakumar, M. Cogswell, R. R. Selvaraju, Q. Sun, S. Lee, D. Crandall, and D. Batra. Diverse beam search: Decoding diverse solutions from neural sequence models. *arXiv preprint arXiv:1610.02424*, 2016.

M. J. Wainwright. *High-dimensional statistics: A non-asymptotic viewpoint*, volume 48. Cambridge university press, 2019.

X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

S. Welleck, I. Kulikov, S. Roller, E. Dinan, K. Cho, and J. Weston. Neural text generation with unlikelihood training. *arXiv preprint arXiv:1908.04319*, 2019.

Y. Wu, A. Q. Jiang, W. Li, M. Rabe, C. Staats, M. Jamnik, and C. Szegedy. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022.

T. Zhang, T. Yu, T. B. Hashimoto, M. Lewis, W.-t. Yih, D. Fried, and S. I. Wang. Coder reviewer reranking for code generation. *arXiv preprint arXiv:2211.16490*, 2022.

	Aqua	**Multiarith**	**StrategyQA**
Codex001	1.028	1.044	1.025
Codex002	-	1.071	1.033
LaMDA-137	1.019	1.044	1.039
UL2-20B	0.99	0.999	0.999

Table 4: Ratio of average UCS score for correct generations by average UCS score for incorrect generations.

Supplementary Material

A Codex-001/Codex-Cushman results on Xsum/MiniF2F

Unfortunately due to the unexpected shutdown of the OpenAI API, we were unable to obtain results for Codex-001 and Codex-Cushman on the Xsum and MiniF2F datasets.

B Proofs

B.1 Proof of Theorem 2.1

Proof. This is true by definition for $k = 1$. For $k > 1$, let us assume that the number of categories $L = 3$. If the best generation g agrees with $\mathbf{v}$ on only one of the elements, then wlog, let that be the 1st one. Then the agreement score is $(p_1 + p_2')/2$ where $p_2' < p_2$. Let the agreement score for a generation g' that does not agree at all with $\mathbf{v}$ be $(p_1' + p_2'')/2$. However if for example $p_1 = 0.34, p_1' = 0.32, p_2' = 0.01, p_2'' = 0.32$, then g' will be selected over g. □

B.2 Proof of Theorem 2.2

Proof. It is true by assumption for $k = 1$. Assume it is true for $k = t$. Then that means that given the self consistency assumption that $a_t(\mathbf{u}_b, \mathbf{v})$ is the highest possible where a_t is the agreement until $k = t$. Then for $t + 1$, we know that $\sum_{i \neq b} \mathbb{I}(\mathbf{u}_b^{t+1} = \mathbf{u}_i^{t+1}$ is the highest (again by self-consistency assumption). Thus a_{t+1} is also the highest proving the theorem. □

B.3 Proof of Theorem 2.3

Formally, let $\mathbf{u}_i^j \sim Bernoulli(p_j)$. Let $b = \arg\max_i \sum^j p_j \cdot \mathbf{u}_i^j$ (i.e. the sequence selected by our method). Then we want a bound on $\mathbb{E}[\sum_j^k \mathbf{u}_b^j]$.

Proof. Let $q_i = \sum_j \mathbf{u}_i^j$. As all are iid, $\mathbb{E}[q_i] = \sum_j p_j$. We can upper bound this by upper bounding $\mathbb{E}[\max_i q_i]$. Note that $\mathbf{u}_i^j$ is subgaussian with parameter 1/2 as it's bounded in $[0, 1]$. Thus q_i is subgaussian with parameter $\sqrt{k}/2$. Thus $\mathbb{E}[\max q_i - \mathbb{E}[q_j]] \leq \sqrt{\frac{k \log k}{2}} \implies \mathbb{E}[\max q_i] \leq \sum_i p_i + \sqrt{\frac{k \log k}{2}}$

Negating $\mathbf{u}$, we can get the lower bound of $\mathbb{E}[\max q_i] \geq \sum_i p_i - \sqrt{\frac{k \log k}{2}}$ Wainwright [2019]

□

C Simulation results

We setup our simulation as follows. Let d be the number of predicates, n the number of generations, and l the number of categories. Then for each predicate, we uniformly at random sample a categorical distribution and then generate $\mathbf{u}_i$ from that distribution. We then apply our criterion of picking the $\mathbf{u}_b$ that has the highest average fractional agreement with all other $\mathbf{u}_i$ and measure (1) the % of times we are able to retrieve the generation that has the best agreement with $\mathbf{v}$ (2) the % agreement $\mathbf{u}_b$ has with the best possible generation out of the set. We vary d, l between 2 and 50, and n between 25 and 250. All our results are based on 1000 samples. The results are in Figures 5 and 6.

For the first metric, we are able to retrieve the best generation a very high fraction of the time when l is < 5 even when d goes to higher values. Even when l is larger, we are still able to retrieve the best generation a non-trivial fraction of times – and notably our performance does not degrade much as n goes from 25 to 250.

Turning our attention to the second metric, we are able to consistently get a generation close to the best generation. This is especially true for small l where even when d increases to large values, we are able to get close to 100% agreement with the best generation. Even at high values of l however, we get relatively good agreement with the best generation – especially compared to picking a random generation – a heuristic we consistently beat.

D Experimental baselines

As mentioned earlier, we could not obtain Codex-001 and Codex-Cushman results on Xsum and MiniF2F due to the unexpected API shutdown. For the BLEU and Rouge-2 metrics, we report the values divided by 100. In terms of our baselines, we have

1. **Random selection** - we randomly select a generation from the set of generations
2. **Ranking by mean log probability** - we take the average log probability across the tokens in the generation and select the generation with the highest mean log probability
3. **Ranking using Centroid** - we take the generation with the lowest mean distance to all other generations in our confidence weighted unigram space as used in WUCS.
4. **Coder Reviewer Ranker** - This method has two variants – Normalized Reviewer (NR), and Normalized Coder Reviewer (NCR). NR computes the mean per token $\log p(x|y)$, where y is the generation and x is the prompt, and then ranks based on this metric. On the other hand, NCR merges the mean log probability ranking with NR, ranking according to $\log p(x|y) + \log p(y|x)$. As the state of the art in code reranking, these methods represent a strong baseline.

E Comparison with Coder-Reviewer Ranker

The results are in Table 5. Consensus-WUCS consistently outperforms NR and often surpasses NCR as well.

In the HumanEval dataset, Consensus-WUCS yields the highest accuracy for the Llama-13B and Llama-30B models. Similarly, in the MBPP-S dataset, Consensus-WUCS delivers superior performance for the Llama-13B and Llama-30B models, and closely matches the NCR for Codex models. In the MBPP dataset, the Consensus-WUCS method ranks as the best for Code-Cushman, Llama-13B, and Llama-30B models.

Notably in 40% of the experiments (6 out of 15), Consensus-WUCS outperforms all other methods, including the highly competitive NCR. Furthermore, Consensus-WUCS ranks second in 8 out of the 15 experiments, reinforcing its strong performance across diverse models and datasets.

Our results present evidence of the effectiveness of WUCS and Consensus-WUCS, which hold their own against much more heavyweight state-of-the-art methods and frequently deliver superior performance.

F Ada model embeddings also give a boost

To understand how generalizable the intuition behind the GCS metric (as opposed to the UCS metric) is for other similarity functions, we took the generations and used the text-ada-embedding-002 model by OpenAI to generate embedding vectors for the generations. We then used cosine similarity between the generations as the similarity function and used $GCS_{\text{Cosine Similarity}}$ to rank. The results are in Table 6. Using OpenAI embeddings as well results in improved performance over Random selection as well as mean log probability ranking validating our intuition that choosing the generation that is on average, the most similar to all other generations is a good ranking metric. That said,

	WUCS	Consensus-WUCS	N. Reviewer	N. Coder-Reviewer
	HumanEval			
Codex002	0.558	**0.568**	0.524	**0.576**
Codex001	0.426	**0.445**	0.42	**0.482**
Code-Cushman	0.373	**0.381**	0.358	**0.385**
Llama-13B	**0.187**	**0.192**	0.164	0.181
Llama-30B	**0.263**	**0.267**	0.219	0.241
	MBPP-S			
Codex002	0.58	**0.589**	0.559	**0.595**
Codex001	0.535	**0.546**	0.509	**0.55**
Code-Cushman	0.472	**0.488**	0.455	**0.512**
Llama-13B	**0.266**	**0.277**	0.228	**0.266**
Llama-30B	**0.363**	**0.373**	0.302	0.325
	MBPP			
Codex002	0.587	**0.594**	**0.631**	0.592
Codex001	0.52	0.525	**0.532**	**0.545**
Code-Cushman	**0.405**	**0.42**	0.398	0.339
Llama-13B	0.195	**0.199**	0.185	**0.2**
Llama-30B	0.287	**0.294**	**0.289**	0.283

Table 5: Comparison with Coder-Reviewer Reranker. Best results are colored in **first**, **second**.

	Random	Mean-logp	Ada	Consensus-WUCS
HumanEval	0.437	**0.533**	0.487	**0.568**
MBPP	0.533	0.416	**0.579**	**0.594**
MBBP-S	0.549	0.568	**0.601**	**0.589**
MiniF2F (BLEU)	0.558	0.556	**0.584**	**0.562**
Xsum (Rouge-2)	0.197	0.214	**0.219**	**0.219**

Table 6: Performance of cosine similarity of ada embedding as the similarity function. Metric is accuracy for HumanEval, MBPP, MBPP-S and BLEU for MiniF2F. Best results are colored in **first**, **second**.

this particular similarity function underperforms UCS, especially for code generation so we did not investigate it further.

G Normalizing inner product degrades performance

Neural generation models are well known to generate repetitive sequences Zhang et al. [2022], Welleck et al. [2019]. In Welleck et al. [2019], they modify the standard log-likelihood object for language models to minimize the probability of tokens immediately preceding the current token. This effectively pushes the model to generate unique new tokens and they show significant improvements in their model after they do this. If we normalize the inner product, then we would be effectively "canceling out" the contribution to the similarity score by having more unique tokens.

We evaluated the effect of normalizing the inner product by the vector norms. To understand better whether our performance is just an effect of selecting longer and more diverse sequences or whether the similarity metric itself is useful as well, we ran ablations where we evaluated ranking based on the longest sequence, as well as based on mean across the elements of $\mathbf{v}_i$ as defined in Section 3 – which takes into account the sequence diversity. The results are in Table 7 in the Supplement. Normalization results in a decline in performance. Furthermore neither ranking by the longest sequence nor ranking by sequence diversity is sufficient to give the results we see as neither result in a consistent improvement even against the Random selection baseline.

	Random	WUCS	WUCS-normalized	Longest	Most Diverse
	HumanEval				
Codex002	0.435	**0.558**	0.462	0.441	**0.51**
Codex001	0.345	**0.426**	**0.382**	0.338	0.369
Llama-30B	0.207	**0.263**	**0.235**	0.208	0.215
	Random	**WUCS**	**WUCS-normalized**	**Longest**	**Most Diverse**
	MBPP				
Codex002	0.536	**0.587**	**0.576**	0.529	0.52
Codex001	0.475	**0.52**	**0.517**	0.475	0.457
Llama-30B	0.262	**0.287**	**0.278**	0.263	0.245
	Random	**WUCS**	**WUCS-normalized**	**Longest**	**Most Diverse**
	Xsum				
Codex002	0.197	**0.215**	**0.211**	0.197	0.188
Llama-30B	0.107	**0.122**	**0.12**	0.107	0.116
GPT-J	0.065	**0.07**	**0.07**	0.065	0.069

Table 7: Impact of normalization. Best results are colored in **first**, **second**.

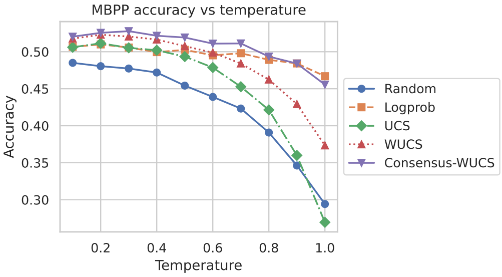

Figure 2: Accuracy for MBPP as the decoding sampling temperature increases.

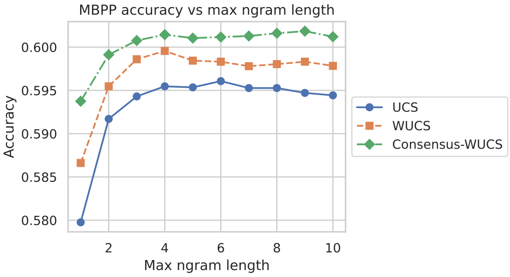

Figure 3: Accuracy for MBPP as the n in n-gram increases.

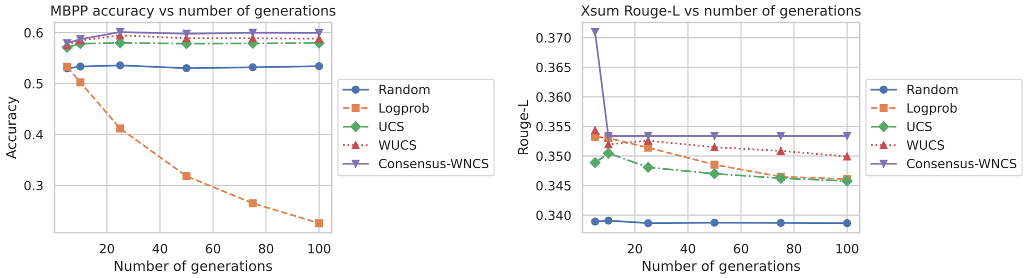

Figure 4: Accuracy for MBPP and Rouge-2 for Xsum as the number of generations increase.

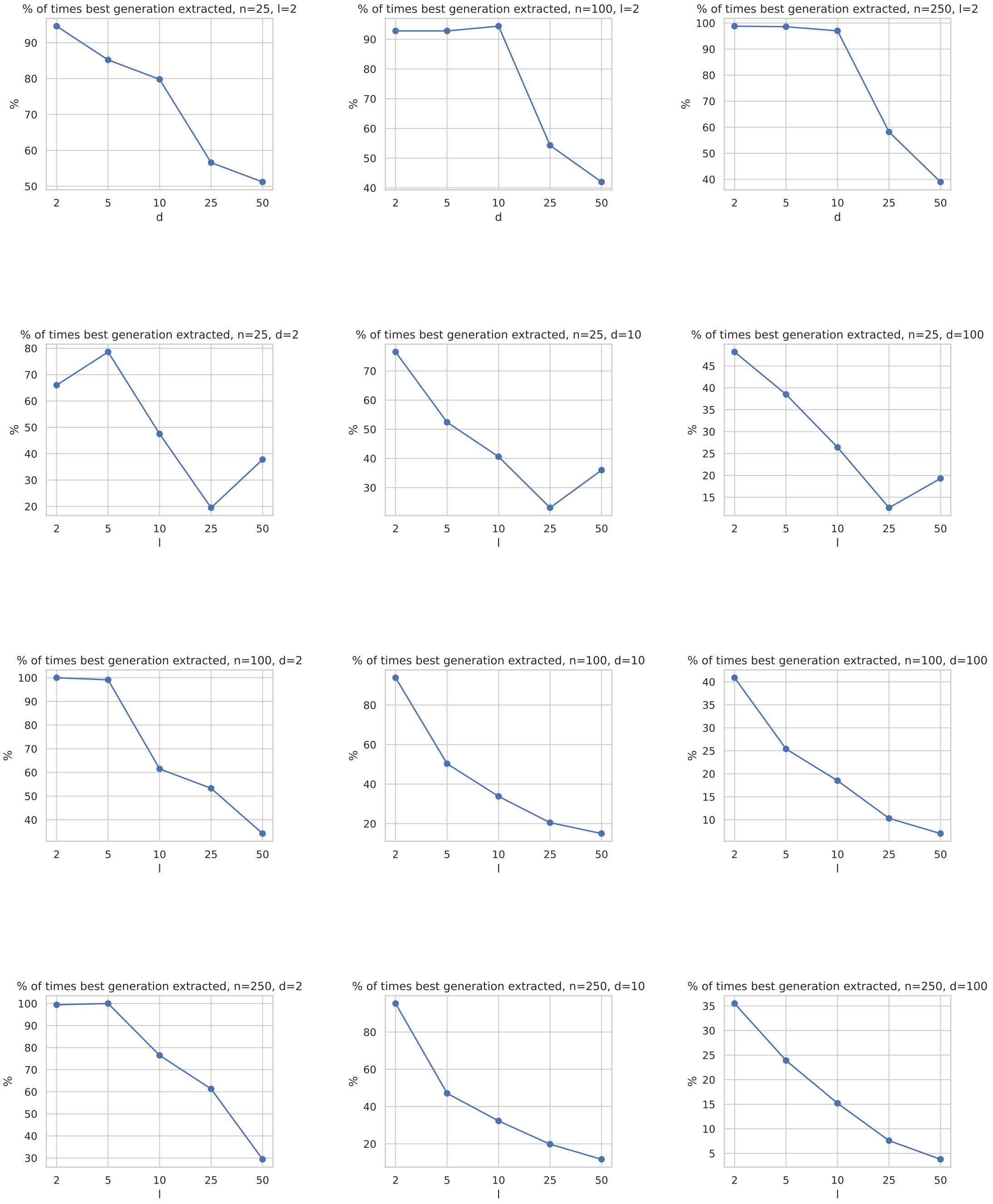

Figure 5: The above figures show what percentage of the time we are able to retrieve the best generation out of the set of generations that we have

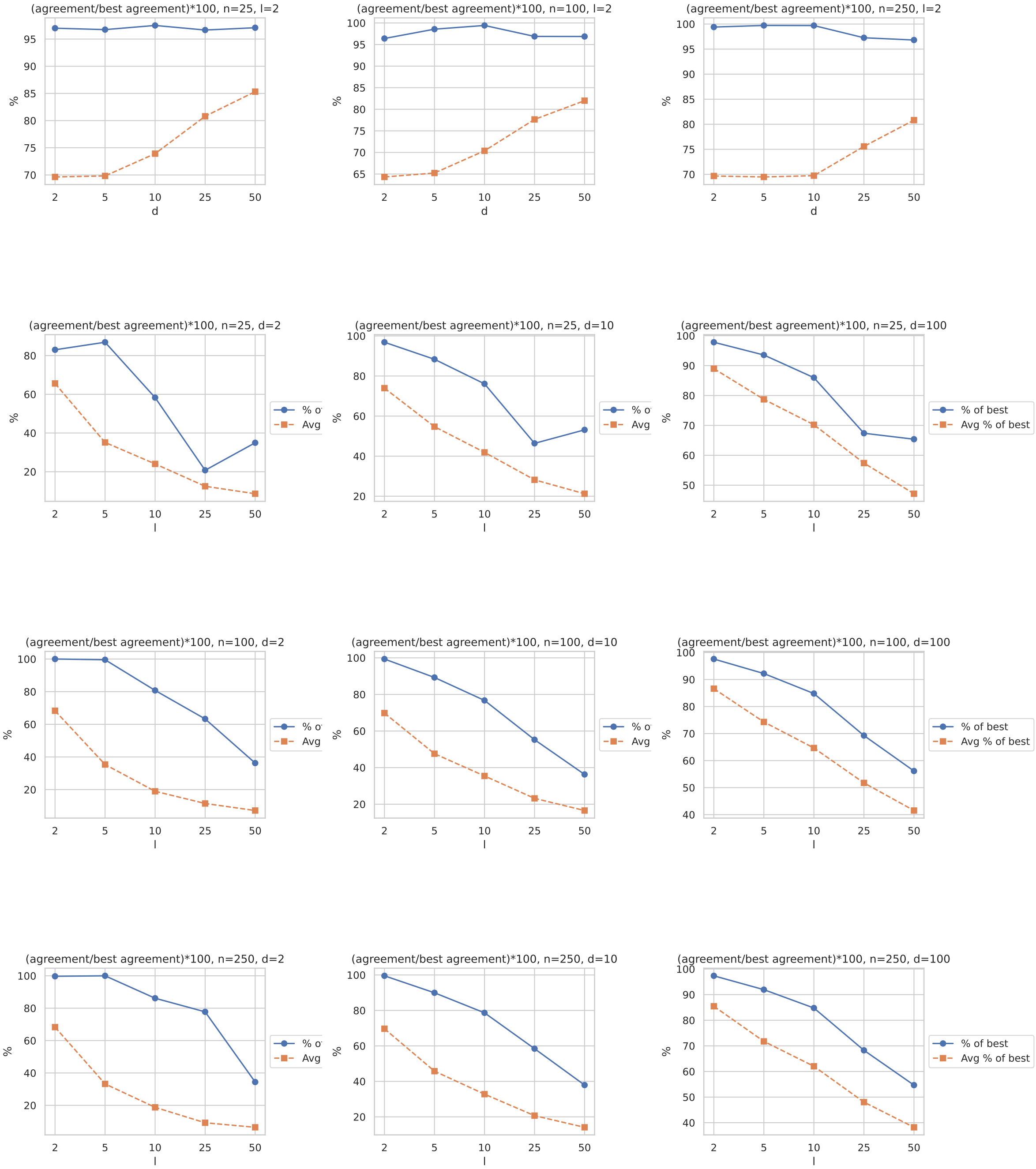

Figure 6: The above figures show what % the best generation as per the highest fractional agreement heuristic and a randomly selected generation agree with the best generation of the set