Testing the generalizability of MBR-exec and further improvements to code generation accuracy

Aidan Walsh
abwalsh@princeton.edu
and **Rebecca Zhu**
rebeccazhu@princeton.edu
and **Jeremy Dapaah**
jdapaah@princeton.edu

Abstract

Recent work in Natural Language Processing has presented alternative approaches to decoding the output of large language models (LLMs), specifically those fine-tuned to produce code based off of a natural language prompt. One such approach, MBR-exec, performs a Minimum Bayes Risk assessment on the execution results of multiple sample outputs of code, and has been found previously to significantly improve the execution accuracy of Codex. In this work, we test the generalizability of MBR-exec on another code generation model, CodeGen, and additionally motivate the discussion on how to best evaluate the output of code generation models. We propose a linear interpolation model of multiple metrics that further improves the downstream accuracy of both Codex and CodeGen.[1]

1 Introduction

Recent work in machine translation focuses on the use on transformer architectures (Vaswani et al., 2017). These models have been effective in generalizing the sequence-to-sequence paradigm to problems such as text summarization (Liu and Lapata, 2019), question answering (Lukovnikov et al., 2020), and even tasks without natural language as output, such as code generation. Code generation models can be useful as they can lead to faster development. By automating the production of boilerplate code, software developers can focus their larger efforts on more complex design and developmental tasks. One of the most well-known for this task is OpenAI's Codex. Codex is a code generation model built on GPT-3, and is used to power tools such as GitHub Co-pilot.

In order to improve Codex's execution accuracy, Shi et al. introduce execution result–based minimum Bayes risk decoding (MBR-exec), an alternate decoding scheme which samples the model multiple times and ranks the outputted code samples based on their execution output on unit tests (Austin et al., 2021) (Shi et al., 2022). They found that the ranking strategy significantly increased downstream execution accuracy of the model. In this work, we apply the MBR-exec ranking system to CodeGen, another code generation model, and find that its increase generalizes to other models. In addition, we also propose a linear interpolation model using the MBR-exec score, the average log probability of the code samples, and n-gram matching scores such as BLEU (Papineni et al., 2002) and METEOR (Banerjee and Lavie, 2005). We find that using this model as a ranking scheme as opposed to solely the MBR-exec score further increases the accuracy of the model.

2 Background

2.1 MBR-exec

Execution-result based minimum Bayes risk decoding (MBR-exec) is a reranking strategy proposed by Meta researchers Shi et al. Generally, minimum Bayes risk functions act as a method of selecting the sample which minimizes a loss function of the option with respect to the sample space. In the case of MBR-exec, the loss function for each code sample is a sum of how many other sample programs where the execution results differ in any way. As indicated in the formula below, we see that the more dissimilar a program's execution outputs are to the other samples, the higher the MBR score. The MBR is calculated with the following equation, where $\mathcal{P}$ is the set of program samples and $\mathcal{T}$ is the set of unit tests for the program:

$$\hat{p} = \arg\min_{p_i \in \mathcal{P}} \sum_{p_j \in \mathcal{P}} \max_{t \in \mathcal{T}} \mathbb{1}[p_i(t) \neq p_j(t)]$$

Thus, the MBR-exec score is an integer $\in [0, |\mathcal{P}| - 1)$, where a program can only have a MBR of zero if it matches all other programs on all test cases.

[1]Our code and data can be found at https://github.com/Aidan-Walsh/mbr-linear

If two programs have matching MBR-exec scores, then the program with the greater average log probability is chosen (explained in 2.2.3).

2.2 Evaluation Metrics

2.2.1 BLEU

BLEU (Papineni et al., 2002) is a metric introduced by Kishore Papineni and his colleagues in 2002 for evaluating natural language translation attempts. It functions by calculating the geometric sum of multiple n-gram similarities scores between a candidate translation and human generated references in the target language. BLEU gained recognition as the de-facto method of evaluation machine translation, especially in contexts between natural languages. Typically a translation with a score of about 30-40 is understandable or good, while translations with a score of up to 60 are at the level of fluent and adequate human translations. Shi et al. uses MBR-BLEU as a metric that computes the BLEU score between two generated programs. Unlike MBR-exec, it instead chooses the generated program that most overlaps textually with other generated programs, with respect to BLEU. They use MBR-tokenBLEU which tokenizes based on whitespace and MBR-charBLEU (also called MBR-BLEU) which tokenizes on a character level. We also use these schemes in our work. Refer to the tables in our appendix to see the calculations of BLEU and MBR-exec.

2.2.2 METEOR

BLEU, while an industry standard, is often ineffective at deriving the intent via an n-gram model. As a result, reasonable translations can often suffer from a poor score, while lower quality translations that match more tokens will score higher. An alternative translation evaluator is METEOR (Banerjee and Lavie, 2005), which places a heavier emphasis on recall than the BLEU algorithm. Additionally, METEOR takes semantic similarity into account, while BLEU operates off exact matches. We propose using MBR-METEOR (tokenized by characters) and MBR-tokenMETEOR (tokenized by whitespace) as potential decoding schemes. They perform identically to MBR-BLEU, however they use the METEOR metric instead.

2.2.3 Log Likelihood

We perform two decoding schemes utilizing the log probabilities of generated code tokens: maximizing likelihood (ML) and maximizing average log likelihood (MALL). In ML, given a set of programs, we select the program with the largest log likelihood:

$$\hat{p} = \arg\max_{p \in \mathcal{P}} \Pi_{i=1}^{n_p} P(w_{p,i}|C, w_{p,1}, ..., w_{p,i-1})$$

where n_p is the number of tokens in a generated program p, and $w_{p,i}$ is the i'th token. C is the context/prompt that we provide. In MALL, we follow other work (Chen et al., 2021) and choose the program that has the highest average log likelihood since ML favors shorter programs:

$$\hat{p} = \arg\max_{p \in \mathcal{P}} \frac{1}{n_p} \sum_{i=1}^{n_p} \log P(w_{p,i}|C, w_{p,1}, ..., w_{p,i-1})$$

We are able to access probabilities of tokens by accessing the logits of the final output layer of the transformers.

2.2.4 Executability

Since MBR-exec runs generated code on test cases, this gives it a special advantage: it tests for executability and only selects code that runs. Thus, we perform Shi et al.'s ablation study on Codex that compares MBR-exec (execution results) with decoding schemes that test executability. These decoding schemes are called "executability-mbr-bleu" and "executability-logprob" in our results section. They perform the same calculations as "mbr-bleu" and "logprob", except they also only select generated code that can be executed.

2.3 Codex and CodeGen

Codex is a descendant of OpenAI's GPT-3 (Brown et al., 2020), fine-tuned for code generation tasks and is trained on 159 gigabytes of python code from 54 million Github repositories. In partnership with Microsoft, OpenAI used Codex to create Github Co-pilot, a code completion tool integrated in IDEs (OpenAI, 2021). Codex is capable of producing code in multiple languages, including JavaScript, Go, Perl, PHP, Ruby, Swift and TypeScript, and Shell, but it is most capable in Python, the language on which it was fine-tuned. Codex has since been deprecated in favor of newer models, but the execution results for Shi et al. are available in the project repository.[2]

CodeGen is a family of code generation models released by Salesforce, Inc, and made available

[2] https://github.com/facebookresearch/MBR-exec#setup

on HuggingFace (Nijkamp et al., 2022). Out of the family, a model can be selected based on two parameters: the size of the model, as well as the corpus it was trained on. The models can be fine tuned for specific tasks using the open source JaxFormer training library, released in the same paper.

2.4 MBPP

The Mostly Basic Programming Problems dataset (MBPP) contains 974 Python functions, text descriptions of those programs, and test cases to check for functional correctness. Their difficulty is designed to be solvable by entry-level programmers (Austin et al., 2021). 500 natural language prompts and their corresponding reference are portioned in the dataset for testing.

3 Methodology

3.1 CodeGen Model Selection

To evaluate the generalizability of MBR-exec, we need a new code generation model to test the decoding scheme on. We elected to use CodeGen because it was open-source and easy to integrate into the architecture of the MBR-exec codebase. In selecting a model out of the CodeGen family, we took multiple factors into consideration. Given the time constraint of the project, we couldn't choose too large of a model, as we needed to have enough time to sample the model extensively enough that we could draw conclusions from our data. Conversely, we also wanted to select a model that was large enough to generate reasonable code outputs. We elected to use the version of CodeGen with 2 billion parameters. We also needed to decide which corpus our model should train on. Our options were The Pile, an English database of size 0.8 TB, a corpus of code in multiple languages, *multi*, or a corpus of Python code called BigPython, *mono*. We elected to choose the model trained on mono, as our test cases would be testing our model on the creation of Python code.

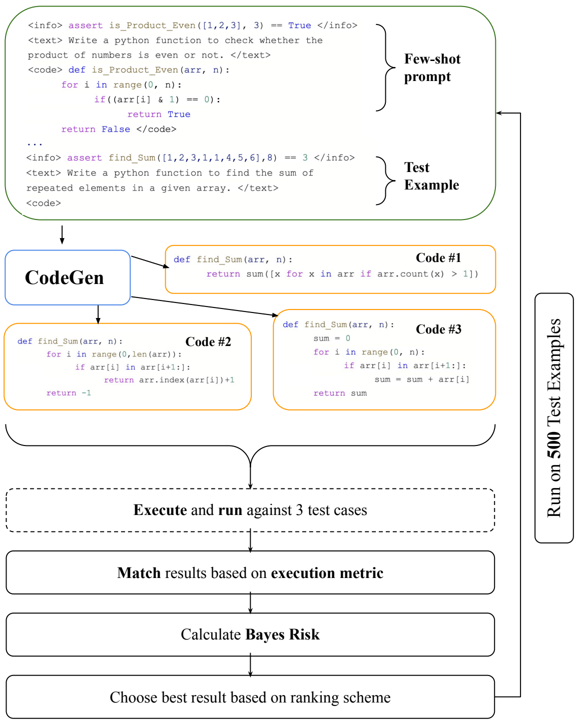
3.2 Few-shot prompts

In the original paper, the prompts given to Codex were structured as 3-shot prompts. 3-shot, or generally k-shot prompts, give the model context on what their output should look like by giving k full examples of successful outputs. In the case of code generation with the MBPP dataset, refer to figure 1 for an example of a few-shot prompt where the "..." indicates 2 repeats of the structure above it.

From the prompt, the model would add the code and conclude its output with the </code> token, which it has learned from the k shots prior. We apply this same prompt structure to our input for CodeGen.

3.3 Data Collection

Querying CodeGen requires GPU access to be tractable for this project. While the checkpoint is open-source and available to use from HuggingFace, it requires multiple hours of GPU time to query enough samples for each of the 500 test outputs in the MBPP. We collected the data over multiple days of GPU runtime on Google Colab, saving the outputs in a shared drive. For each of the 500 test cases provided by the MBPP dataset, we collected 45 code outputs from CodeGen for temperatures 0.3 and 0.6. For temperature 0.9, we collected 10 samples. This was achieved by using 10 random seeds for the queries to CodeGen, where each random seed generated approximately 5 code outputs. Thus, when sampling, this enables us to sample up to 45 code generations for temperatures 0.3 and 0.6 and up to 10 code generations for temperature 0.9. The temperatures 0.3,0.6, and 0.9 are spaced far enough apart to show us trends in CodeGen output as temperature changes.



```
<info> assert is_Product_Even([1,2,3], 3) == True </info>
<text> Write a python function to check whether the
product of numbers is even or not. </text>
<code> def is_Product_Even(arr, n):
      for i in range(0, n):
            if((arr[i] & 1) == 0):
                  return True
      return False </code>
...
<info> assert find_Sum([1,2,3,1,1,4,5,6],8) == 3 </info>
<text> Write a python function to find the sum of
repeated elements in a given array. </text>
<code>
```

```
def find_Sum(arr, n):
      return sum([x for x in arr if arr.count(x) > 1])
```

```
def find_Sum(arr, n):
      for i in range(0,len(arr)):
            if arr[i] in arr[i+1:]:
                  return arr.index(arr[i])+1
      return -1
```

```
def find_Sum(arr, n):
      sum = 0
      for i in range(0, n):
            if arr[i] in arr[i+1:]:
                  sum = sum + arr[i]
      return sum
```

Few-shot prompt
Test Example
CodeGen
Code #1
Code #2
Code #3
Execute and run against 3 test cases
Match results based on execution metric
Calculate Bayes Risk
Choose best result based on ranking scheme
Run on 500 Test Examples

Figure 1: Methodology for calculating MBR-exec and choosing best generated program $\hat{p}$ using a few-shot prompt

3.4 Linear Interpolation

To develop a new decoding scheme, we derived a linear interpolation model using the various metrics included in the CodeGen API return values, such as logits, and calculated values ourselves. This included the average log probability of the sequence (a_5), the MBR-exec score (a_4), MBR-BLEU (a_2) and MBR-tokenBLEU (a_1), as well as MBR-tokenMETEOR (a_3):

$$\hat{p} = \arg\max_{p \in P} \ a_p \cdot b \text{ s.t. } a_p, b \in \mathcal{R}^5, \text{ where}$$

$$a_p = \left[a_1, a_2, a_3, a_4, a_5\right]; b = \left[b_1, b_2, b_3, b_4, b_5\right]$$

where all b_i serve as hyperparameters and our goal is to find a program p that maximizes the dot product between a_p and b. All a_i are also normalized from 0 to 1. For each generated program p, a_p is fixed, but all $b_i \in b$ can be tuned such that we maximize execution accuracy among selected programs. To find such b_i, we used a naive grid search algorithm:

```
score = 0 # highest average score
for b1 in [potential values...]:
    for b2 in [potential values...]:
        ...
            temp = 0
            # run number of times to get average score
            # with hyperparameters
            for i in range(num_iterations):
                # i is the seed used to get random sample
                # of generated code
                temp += run_with_hyperparams([b1,b2,...], i)
            if temp/num_iterations > score:
                hyperparams = [b1,b2,...]
                score = temp/num_iterations

return hyperparams
```

Figure 2: Pseudocode for Grid Search algorithm

3.5 Contextual Logits

We also propose a "context" decoding scheme that uses the token log probabilities contextually, rather than just as a sum in ML or as an average in MALL. To do so, "context" uses a weighted average where each token log probability is given weight according to hyperparameters. Since CodeGen only provided the ID numbers of its tokens - not the string values, we were only able to perform this on Codex. Similar to our linear interpolation scheme, we defined 4 hyperparameters: one that rewarded/penalized longer code outputs (c_1), one that increased/reduced weight for tokens involving numbers (c_2), one that increased/reduced weight for tokens involving only characters(c_3), and one that increased/reduced weight for tokens involving only punctuation such as brackets, parenthesis, semi-colons etc. (c_4). A similar grid search algorithm defined in figure 2 was used to find these hyperparameters. Let us define the array of hyperparameters as c, s.t. $c = [c_1, c_2, c_3, c_4]$.

4 Results and Analysis

4.1 Reproducing Shi et al.

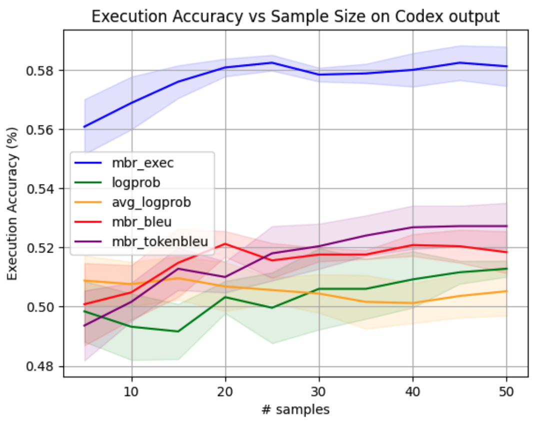


Figure 3: Shi et al.'s proposed decoding schemes on Codex output with a temperature of 0.3. Shown results are the mean and standard deviation after 5 tests for sample sizes from 5 to 50.

Figure 3 indicates that MBR-exec far outperforms the other decoding schemes, up to approximately 6%. Shi et al.'s paper also shows that MBR-exec is slightly above 58% when using a sample size of 50. When using a sample size of 25, both our data and Shi et al. indicate that the mean execution accuracy for MBR-exec is 58.2% with a standard deviation of 0.3%. Refer to appendix A, graph 11, to see Shi et al.'s graph of similar results.

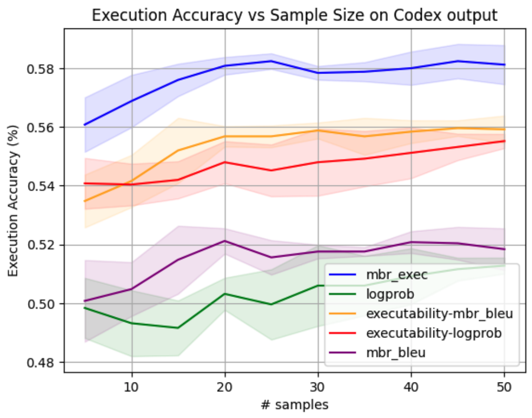


Figure 4: Decoding schemes on Codex output that involve testing executability with a temperature of 0.3. Shown results are the mean and standard deviation after 5 tests for sample sizes from 5 to 50.

Figure 4 indicates that testing for executability of Codex's generated programs significantly boosts performance. Comparing to figure 3, MBR-BLEU and logprob see an increase in almost 4% across all sample sizes when executability is tested. Shi et al.'s paper makes similar findings across sample sizes up to 120. MBR-exec still outperforms all other decoding schemes by at least 2%. Refer to appendix A, graph 12, to see Shi et al.'s graph of similar results.

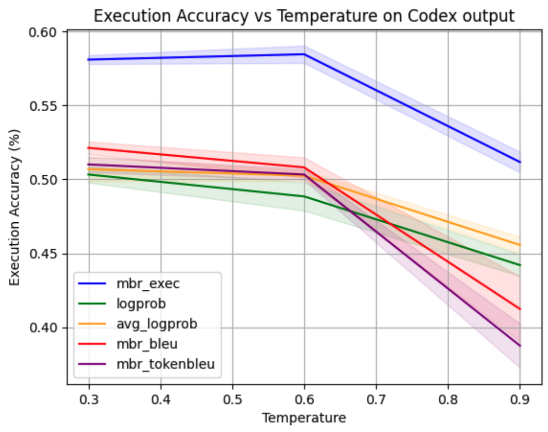


Figure 5: Decoding schemes on Codex output with temperatures 0.3,0.6,0.9. Shown results are the mean and standard deviation after 5 tests for sample sizes 25.

Figure 5 indicates a general trend that as temperature increases from 0.3 to 0.9, performance generally decreases. For all decoding schemes except MBR-exec, performance slightly decreases from t=0.3 to t=0.6, then sharply decreases from t=0.6 to t=0.9. An exception would be MBR-exec where its performance very slightly increases from t=0.3 to t=0.6, then sharply decreases. Shi et al.'s paper produces very similar results that sees identical trends. A higher temperature indicates more variability in output, so the use of log probabilities is likely ineffective for t=0.9, since Codex will be outputting code with lower and more variable probabilities. The variability will also reduce the likelihood of there being a popular code output, thus reducing the performance of any variations of MBR. MBR-exec possibly performs worse with t=0.3, because since there is less variation, the reduced variation likely produces a popular incorrect output, that is less popular when t = 0.6. Refer to appendix A, graph 13, to see Shi et al's graph of similar results.

4.2 CodeGen Results

Figure 6 indicates that MBR-exec still outperforms all other decoding schemes, as long as t=0.6. MBR-exec performs up to 5% better than the next best decoding schemes (avg-logprob and logprob perform the exact same) for larger sample sizes. Compared to figure 3 which focuses on Codex, this 5% difference is very similar to the 6% difference. On the other hand, MBR-tokenBLEU and MBR-BLEU perform worse than schemes using the log-probability, where the opposite is shown in figure 3.

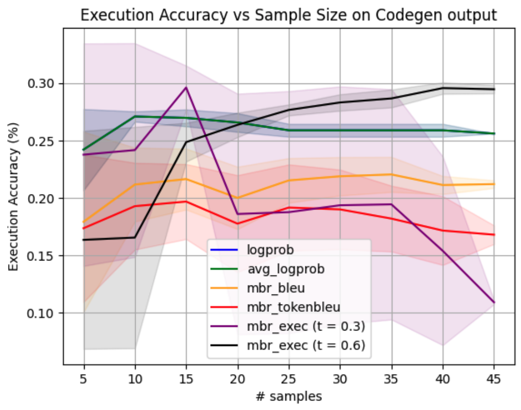


Figure 6: Shi et al's proposed decoding schemes on CodeGen output with a temperature of 0.3 unless otherwise specified. Shown results are the mean and standard deviation after 5 tests for sample sizes from 5 to 45.

This could be because CodeGen is only using 2B parameters, so it is more likely to produce incorrect code than Codex, so its popular textual output is incorrect, and so it is better to rely on the token log probabilities. MBR-exec, when t=0.3, sees a massive reduction in performance (and large variance) as sample size increases. This is likely, again, due to CodeGen's reduced performance that is only able to produce a correct popular output when there is more variability when t=0.6. This is similar to the explanation for the increase in performance seen for MBR-exec in figure 3, however, there is a much larger difference for CodeGen. **This indicates that MBR-exec's performance is maximized when t=0.6 for text-to-code generators.**

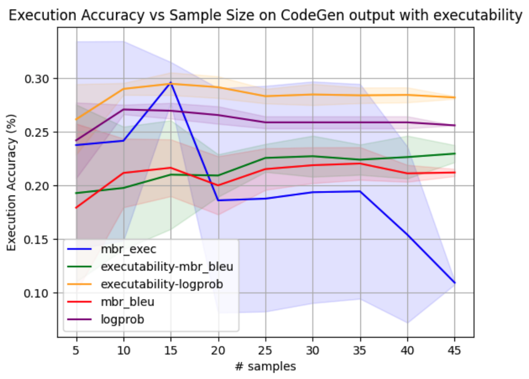


Figure 7: Decoding schemes on CodeGen output that involve testing executability with a temperature of 0.3. Shown results are the mean and standard deviation after 5 tests for sample sizes from 5 to 45.

Figure 7 indicates that testing executability for CodeGen, like for Codex in figure 4, also boosts performance. Logprob sees a boost in approximately 3%, similar to the boost for Codex, and is only 2% worse than MBR-exec when t=0.6 and sample sizes are large. This is very similar to the difference in figure 4. MBR-BLEU also sees a boost in performance by about 2% compared to the 4% boost in figure 4. **Thus, we can conclude that executability improves decoding schemes by 2-4% for both Codex and CodeGen, where MBR-exec is still more effective** (when t = 0.6 for CodeGen, but can be both 0.3 or 0.6 for Codex). The only notable difference is that log probabilities are more effective than MBR-BLEU for CodeGen.

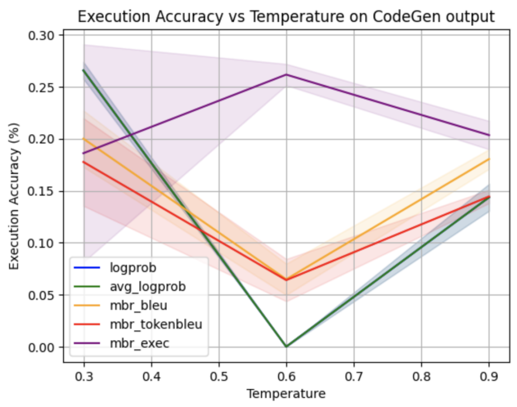


Figure 8: Decoding schemes on CodeGen output with temperatures 0.3,0.6,0.9. Shown results for temperatures 0.3 and 0.6 are the mean and standard deviation after 5 tests for sample sizes 20. For 0.9, 5 tests were run with a sample size of 5.

Figure 8 indicates that performance for all decoding schemes, except MBR-exec, is reduced when t = 0.6. This is the same trend seen for Codex in figure 5; however, performance is more drastically reduced when t=0.6, particularly logprob and avg-logprob which see performance drop to 0% and the other schemes drop to below 10%. This is likely because the variation when t=0.6 produces output with low log-likelihoods and so using schemes involving log-likelihoods is very ineffective. The same can be said for tokens of generated code that MBR-BLEU an MBR-tokenBLEU use since t=0.6 likely generates unlikely code. Surprisingly, when t=0.9, performance is better than when t=0.6. For MBR variations, this could be because when t=0.9, the correct output still dominates proportionally. For logprob, this could be the way that we wrote our code: we ignored negative infinity values for log probabilities which were prevalent for t=0.6 and 0.9. These negative infinity values were not present for Codex but sometimes appeared in CodeGen's logits. Thus, looking at all all non-negative-infinity log-probabilities was likely detrimental for t=0.6 since they appeared infrequently and disproportionately but appeared proportionally for all CodeGen output when t=0.9. MBR-exec still performed best when t=0.6 and lost performance as temperature increased, like in figure 5.

Method	Codex	CodeGen
Sample (3-shot)	0.4759 ± 0.016	0.1586 ± 0.11
MBR-exec	0.5824 ± 0.0027	0.1860 ± 0.10
Interpolation	$\mathbf{0.5844} \pm \mathbf{0.0059}$	$\mathbf{0.2780} \pm \mathbf{0.03}$

Table 1: Results for t = 0.3. Sampling is accuracy with no decoding so is essentially decoding with sample size 1 and we report mean and standard deviation after 125 tests. For MBR-exec and interpolation, results shown are after 5 tests with sample sizes of 25.

4.3 Adding our Ablations

Table 1 indicates that our proposed interpolation decoding scheme is more effective than MBR-exec for both Codex and CodeGen when t=0.3. CodeGen saw an improvement in performance by almost 10% and Codex saw an improvement in performance by 0.2%. Our interpolation hyperparameters were chosen for Codex, then were extended to be used on CodeGen, where the large improvement can be seen. **Thus, our interpolation boosted the performance of MBR-exec and can be extended across text-to-code generators**. Furthermore, the poor performance seen by MBR-exec on some temperatures (as seen on CodeGen when t=0.3) can be remedied by our interpolation scheme.

Method	Codex	CodeGen
Sample (3-shot)	0.4412 ± 0.016	0.05103 ± 0.085
MBR-exec	$\mathbf{0.5844} \pm \mathbf{0.0059}$	0.2616 ± 0.010
Interpolation	0.5820 ± 0.0087	$\mathbf{0.2770} \pm \mathbf{0.011}$

Table 2: Results for t = 0.6. We report mean and standard deviation after 125 tests for sampling. For MBR-exec and interpolation, results shown are after 5 tests with sample sizes of 20.

Table 2 indicates that interpolation performed better than MBR-exec by 1.54% on CodeGen but performed worse by 0.24% for Codex. Performance increase across all text-to-code generators for all temperatures is not expected since the hyperparameters were tuned for t=0.3. Seeing that the performance still increased for CodeGen, however, **we can claim that interpolation may increase performance across different temperatures for text-to-code generators.** Interestingly, interpolation for t=0.3 performed the exact same as MBR-exec for t=0.6, potentially indicating a global maximum for execution accuracy for all decoding schemes. Interpolation on CodeGen when t=0.3 also performed better than interpolation when t=0.6, where their results for MBR-exec saw a reversed relationship, **indicating that interpolation remedies any issues in MBR-exec across temperatures in text-to-code generators**.

Figure 9 shows that interpolation consistently performs better on Codex than MBR-exec when t=0.3. Our other proposed schemes, MBR-

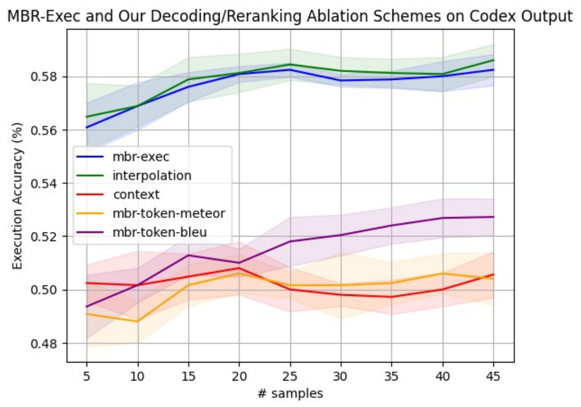


Figure 9: Our proposed ablations on Codex output with t = 0.3. Shown results are the mean and standard deviation after 5 tests for sample sizes from 5 to 45.

tokenMETEOR and Context perform worse than MBR-tokenBLEU where they only ever get up to 51% accuracy compared to MBR-tokenBLEU's 53% and interpolation's 58.4%.

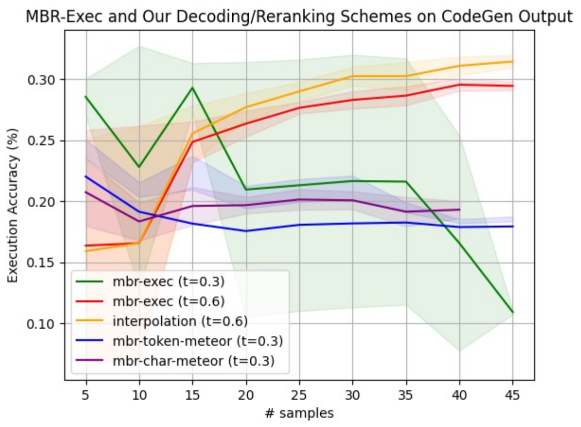


Figure 10: Our proposed ablations on CodeGen output with t = 0.3 or 0.6. Shown results are the mean and standard deviation after 5 tests for sample sizes from 5 to 45.

Figure 10 shows that interpolation consistently performs better (around 2%) than MBR-exec for both t=0.3 and t=0.6 on CodeGen, with a smaller difference shown in figure 9 (around 0.2%). Similar to Codex in figure 9, MBR-tokenMETEOR and MBR-METEOR (MBR-charMETEOR) also perform worse than interpolation and MBR-exec. Looking at figure 4, they also perform worse than MBR-BLEU, like the trend seen in figure 9. **Thus, we can conclude that MBR-METEOR and MBR-tokenMETEOR consistently perform worse than MBR-BLEU and MBR-exec, but interpolation consistently performs the best among all decoding schemes.**

4.4 Hyperparameters

4.4.1 Linear Interpolation

The grid search resulted in the following b: $[-0.5, 0.5, -0.5, 100, 2]$. The -0.5 for b_1, b_3 indicates that we actually penalized the MBR-tokenBLEU and MBR-tokenMETEOR scores of generated programs. We notice that these are token scores, not character scores. This lets interpolation perform better likely because program similarity on a token (separated by whitespace) level encourages and only focuses on identicalness of program structure and function use. In reality, correct programs can be incredibly diverse, so token MBR scores such as MBR-tokenMETEOR and MBR-tokenBLEU may be detrimental in some cases to the finding of the best generated program $\hat{p}$. The 0.5 indicates that a small value is given to MBR-BLEU, the 100 indicates that most of the weight is given to MBR-exec, and the 2 indicates that some weight goes to the average log probability. Since MBR-exec uses its primary calculation MBR-exec and then uses the average log probability for tie-breaking, it makes sense that most of the weight would be given to MBR-exec, then to the average log probability. Compared to MBR-exec, linear interpolation does require the tuning of hyperparameters, but as we have seen in figure 9 and 10, it can be extended across text-to-code generators, and from tables 1 and 2, it can be extended across temperatures. Thus, after finding one set of hyperparameters, they can be reused and do not need to be recalculated. Furthermore, while linear interpolation requires more computation, with parallelization, it can perform at the same speed as MBR-exec.

4.4.2 Context

Due to limits computationally, a full grid-search algorithm was not able to be utilized, however, we found a (mostly) maximizing c such that $c = [4, 10, 1, 0]$. The 4 means that we rewarded longer code outputs, the 10 indicates that we massively increased the weight for tokens that involved numbers, the 1 means we did nothing to the weight for tokens only involving characters, and the 0 indicates that we completely ignored punctuation in our weighted average. According to figure 9, Context underperformed compared to the other baselines, peaking at approximately 51% accuracy, however, it sees a positive gradient after the samples are at 35. With more samples and fine-tuning, it could outperform the other baselines such as MBR-tokenBLEU, but its difference from MBR-exec and interpolation indicate that it may require more hyperparameters or modifications for it to outperform these two schemes.

4.5 Qualitative Analysis

Referring to Appendix A, tables 3 and 4, we can qualitatively analyze the output of Codex and CodeGen. Looking at both tables, we find that Codex tends to produce much longer code output than CodeGen and its methodology is a lot more diverse even for lower temperatures. This possibly leads to the better performance of Codex, since CodeGen likely oversimplifies by writing shorter code. We see that CodeGen with a temperature of 0.3 produces the same code 3 times, however Codex produces 3 different code outputs. This likely led to poor performance for MBR-exec when being used on CodeGen with a temperature of 0.3, since the inaccuracies of CodeGen were likely prevalent in all homogenous outputs. When comparing the difference in code across temperatures, both Codex and CodeGen experiment with more functions: Codex uses join, and uses substrings cleverly, and CodeGen uses a while loop and strip. This diversity in code likely boosts the performance, especially in CodeGen, since it will increase the chance of producing code that passes all the test cases. CodeGen, though, still seems more stubborn by reusing replace and using a similar structure in its code in every output. This length, diversity, and experimentation in Codex is likely due to its 12 billion parameters compared to CodeGen's 2 billion parameters. Both Codex and CodeGen are trained on public python code from Github, however their corpora are technically different (BigPython vs Codex's unlabelled dataset). Since CodeGen's output is a lot less diverse and more homogenous across temperatures, this could possibly mean that while CodeGen looks at a lot of code, much of the training code could be semantically similar or lack diversity. For example, most of the training code could be written by more intermediate programmers, rather than a mix of skill types, or the training code could be mostly pulled from academia.

5 Conclusion and Future Work

In summary, this study provides evidence that the MBR-exec decoding scheme increases the execution accuracy of code generation models, and variations, such as interpolation, further increase execution accuracy. Both interpolation and MBR-exec can be extended across different text-to-code generators, such as Codex and CodeGen, as well as extended across temperatures. Furthermore, the accuracy of the text-to-code generator Codex has been shown to be significantly more effective than CodeGen (2 billion parameters). Due to our limited access to GPUs and RAM, limitations include the usage and testing on CodeGen: future work could analyze CodeGen's models that use 6 billion and 16 billion parameters. Future work could also investigate other interpolation models for code translation such as one that uses the model output as context in the linear interpolation scheme. One such method of doing so could be using the CodeBERTScore metric, which uses embeddings to take the structure of the code into account.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen

Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Yang Liu and Mirella Lapata. 2019. Text summarization with pretrained encoders.

D. Lukovnikov, A. Fischer, and J. Lehmann. 2020. Pre-trained transformers for simple question answering over knowledge graphs.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint*.

OpenAI. 2021. Openai codex.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.

6 Appendix A

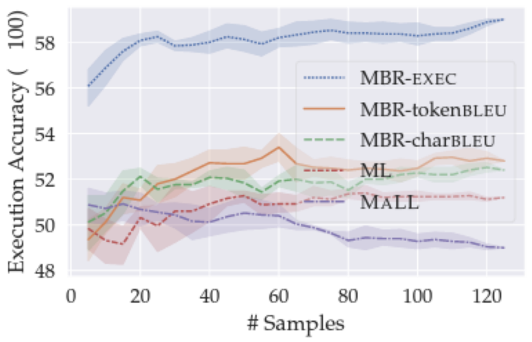


Figure 11: Shi et al.'s results using selection criteria on Codex with temperature 0.3. They ran 5 tests for each sample size from 5 to 120 where they report the mean and standard deviation.

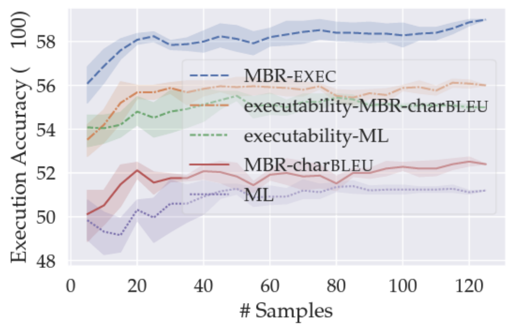


Figure 12: Shi et al.'s results using selection criteria with executability tests on Codex with temperature 0.3. They ran 5 tests for each sample size from 5 to 120 where they report the mean and standard deviation.

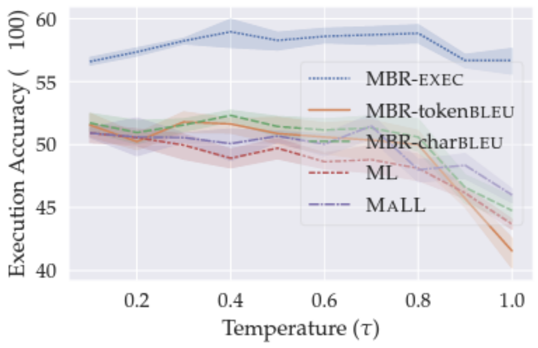


Figure 13: Shi et al.'s results using selection criteria tests on Codex for all temperatures. They ran 5 tests for each temperature where their sample size was 25. They report the mean and standard deviation

Codex	CodeGen

def remove_Occ(string, char):
    string = list(string)
    if char in string:
        string.remove(char)
        string.remove(char)
    return ''.join(string)
	def remove_Occ(str, ch):
	    while str.find(ch)!= -1:
	        str = str.replace(ch, \"\")
	    return str

def remove_Occ(string, character):
    st1 = string.replace(character,\"\",1)
    st2 = st1.replace(character,\"\",1)
    return st2
	def remove_Occ(str, ch):
	    str = str.replace(ch, \"\")
	    return str.strip()

def remove_Occ(text, character):
    return text[text.find(character)
    + 1:text.rfind(character)]
	def remove_Occ(string, char):
	    return string.replace(char, \"\")

Table 3: Output of Codex and CodeGen with t = 0.6 when asked to "Write a python function to remove first and last occurrence of a given character from the string."

label	Codex	CodeGen	MBR-exec	MBR-BLEU

c1	def remove_Occ(str1,char):
	    return str1.
	    replace(char, '', 1).
	    replace(char, '', 1).
	    replace(char, '', 1).
	    replace(char, '', 1)
		def remove_Occ(s,c):
		    return s.
		    replace(c,\"\")
			(0, -1.541)	BLEU(c1,c2)
				+BLEU(c1,c3)
				=0.8897

c2	def remove_Occ(string, char):
	    string = string.
	    replace(char, \"\", 1)
	    string = string.
	    replace(char, \"\", 1)
	    return string
		def remove_Occ(str, char):
		    return str.
		    replace(char, '')
			(0, -1.325)	BLEU(c2,c1)
				+BLEU(c2,c3)
				=1.2075

c3	def remove_Occ(string, char):
	    first = string.
	    find(char)
	    last = string.
	    rfind(char)
	    return string[:first]
	    + string[last + 1:]
		def remove_Occ(str, ch):
		    return str.
		    replace(ch, '')
			(0, -1.983)	BLEU(c3,c2)
				+BLEU(c3,c1)
				=1.0239

Table 4: Output of Codex and CodeGen with t = 0.3 when asked to "Write a python function to remove first and last occurrence of a given character from the string." The MBR-Exec and MBR-BLEU scores reported are for Codex, given that the sample size is the 3 samples shown. For MBR-exec, they match on all test cases, hence the 0, and so the one with the highest average log probability is chosen. Thus, the second one is chosen in MBR-exec. For MBR-BLEU, the highest one is chosen, so the second one is also chosen.