A SYSTEMATIC EVALUATION OF LARGE LANGUAGE MODELS OF CODE

Frank F. Xu, Uri Alon, Graham Neubig, Vincent J. Hellendoorn
School of Computer Science
Carnegie Mellon University
{fangzhex,ualon,gneubig}@cs.cmu.edu, vhellendoorn@cmu.edu

ABSTRACT

Large language models (LMs) of code have recently shown tremendous promise in completing code and synthesizing code from natural language descriptions. However, the current state-of-the-art code LMs (e.g., Codex (Chen et al., 2021)) are not publicly available, leaving many questions about their model and data design decisions. We aim to fill in some of these blanks through a systematic evaluation of the largest existing models: Codex, GPT-J, GPT-Neo, GPT-NeoX-20B, and CodeParrot, across various programming languages. Although Codex itself is not open-source, we find that existing open-source models do achieve close results in some programming languages, although targeted mainly for natural language modeling. We further identify an important missing piece in the form of a large open-source model trained exclusively on a multi-lingual corpus of code. We release a new model, PolyCoder, with 2.7B parameters based on the GPT-2 architecture, that was trained on 249GB of code across 12 programming languages on a single machine. In the C programming language, *PolyCoder outperforms all models including Codex*. Our trained models are open-source and publicly available at https://github.com/VHellendoorn/Code-LMs, which enables future research and application in this area.

1 INTRODUCTION

Language models (LMs) assign probabilities to sequences of tokens, and are widely applied to natural language text (Bengio et al., 2003; Baevski & Auli, 2018; Brown et al., 2020). Recently, LMs have shown impressive performance in modeling also source code, written in programming languages (Hindle et al., 2016; Hellendoorn & Devanbu, 2017; Alon et al., 2020; Karampatsis et al., 2020). These models excel at useful downstream tasks like code completion (Raychev et al., 2014) and synthesizing code from natural language descriptions (Desai et al., 2016). The current state-of-the-art large language models for code, such as Austin et al. (2021), have shown significant progress for AI-based programming assistance. Most notably, one of the largest of these models, Codex (Chen et al., 2021) has been deployed in the real-world production tool GitHub Copilot[1], as an in-IDE developer assistant that automatically generates code based on the user's context.

Despite the great success of large language models of code, *the strongest models are not publicly available*. This prevents the application of these models outside of well-resourced companies and limits research in this field for low-resourced organizations. For example, Codex provides non-free access to the model's *output* through black-box API calls,[2] but the model's weights and training data are unavailable. This prevents researchers from fine-tuning and adapting this model to domains and tasks other than code completion. The lack of access to the model's internals also prevents the research community from studying other key aspects of these models, such as interpretability, distillation of the model for more efficient deployment, and incorporating additional components such as retrieval.

Several medium to large-sized pre-trained language models are publicly available, such as GPT-Neo (Black et al., 2021), GPT-J (Wang & Komatsuzaki, 2021) and GPT-NeoX (Black et al., 2022).

[1] https://copilot.github.com/
[2] https://openai.com/blog/openai-codex/

Despite being trained on a mixture of a wide variety of text including news articles, online forums, and just a modest selection of (GitHub) software repositories (Gao et al., 2020), these language models can be used to generate source code with a reasonable performance Chen et al. (2021). In addition, there are a few open-source language models that are trained solely on source code. For example, CodeParrot (Tunstall et al., 2022) was trained on 180 GB of Python code.

Given the variety of model sizes and training schemes involved in these models and lack of comparisons between these, the impact of many modeling and training design decisions remains unclear. For instance, we do not know the precise selection of data on which Codex and other private models were trained; however, we do know that some public models (e.g., GPT-J) were trained on a mix of natural language and code in multiple programming languages, while other models (e.g., CodeParrot) were trained solely on code in one particular programming language. Multilingual models potentially provide better generalization, because different programming languages share similar keywords and properties, as shown by the success of *multilingual* models for natural language (Conneau & Lample, 2019) and for code (Zügner et al., 2021). This may hint that *multilingual* LMs can *generalize* across languages, outperform monolingual models and be useful for modeling low-resource programming languages, but this is yet to be verified empirically.

In this paper, we present a systematic evaluation of existing models of code – Codex, GPT-J, GPT-Neo, GPT-NeoX, and CodeParrot – across various programming languages. We aim to shed more light on the landscape of code modeling design decisions by comparing and contrasting these models, as well as providing a key missing link: thus far, no large open-source language model was trained exclusively on code from *multiple programming languages*. We provide three such models, ranging from 160M to 2.7B parameters, which we release under the umbrella name "PolyCoder". First, we perform an extensive comparison of the training and evaluation settings between PolyCoder, open-source models, and Codex. Second, we evaluate the models on the HumanEval benchmark (Chen et al., 2021) and compare how do models of different sizes and training steps scale, and how different temperatures affect the generation quality. Finally, since HumanEval only evaluates the natural language to Python synthesis, we curate an unseen evaluation dataset[3] in each of the 12 languages, to evaluate the perplexity of different models. We find that although Codex is allegedly focused on Python (Chen et al. (2021) §3.1), Codex performs surprisingly well in other programming languages too, and even better than GPT-J and GPT-NeoX that were trained on the Pile (Gao et al., 2020). Nonetheless, in the C programming language, *our PolyCoder model achieves a lower perplexity than all these models, including Codex.*

Although most current models perform worse than Codex, we hope that this systematic study helps future research in this area to design more efficient and effective models. More importantly, through this systematic evaluation of different models, we encourage the community to study and release medium-large scale language models for code, in response to the concerns expressed by Hellendoorn & Sawant (2021):

> *[...] this exploding trend in cost to achieve the state of the art has left the ability to train and test such models limited to a select few large technology companies—and way beyond the resources of virtually all academic labs.*

We believe that our efforts are a significant step towards democratization of large language models of code.

2 Related Work

At the core of code modeling lies ongoing work on pretraining of language models (LMs). Large-scale pretraining of LMs has had an astounding impact on natural language processing in recent years (Han et al., 2021). Figure 1 provides an overview of how different models compare in size and availability.

2.1 Pretraining Methods

We discuss three popular pretraining methods used in code language modeling. An illustration of these methods are shown in Figure 2.

[3]The exact training set that Codex was trained on is unknown.

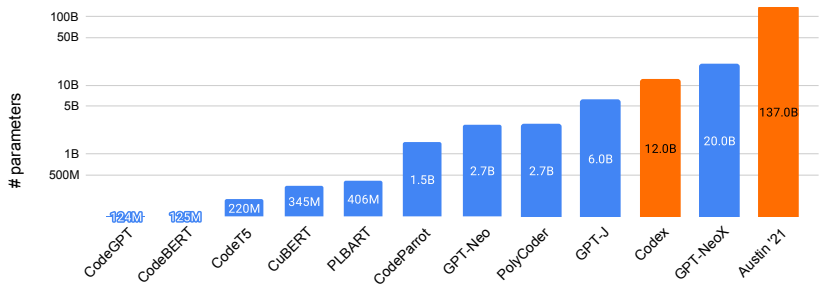


Figure 1: Existing language models of code, their sizes and availability (open source vs. not open-source).

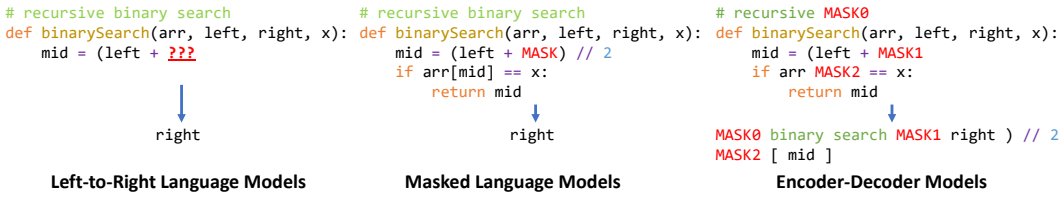


Figure 2: Three types of pretrained language models.

Left-to-Right Language Models (Figure 2, left) Auto-regressive, Left-to-right LMs, predict the probability of a token given the previous tokens. In code modeling, CodeGPT (124M) (Lu et al., 2021), CodeParrot (1.5B) (Tunstall et al., 2022), GPT-Neo (2.7B) (Black et al., 2021), GPT-J (6B) (Wang & Komatsuzaki, 2021), Codex (12B) (Chen et al., 2021), GPT-NeoX (20B) (Black et al., 2022), and Google's (137B) (Austin et al., 2021) belong to this category. The left-to-right nature of these models makes them highly useful for program generation tasks, such as code completion. On the other hand, as code is usually not written in a single, left-to-write pass, it is not trivial to leverage context that appears "after" the location of the generation. In this paper, we focus on this family of models and will discuss the existing models in more detail in the following sections.

Masked Language Models (Figure 2, middle) While auto-regressive language models are powerful for modeling the probability of sequences, their unidirectional nature makes them less suitable for producing effective whole-sequence representations for downstream tasks such as classification. One popular bidirectional objective function used widely in representation learning is masked language modeling (Devlin et al., 2018), where the aim is to predict masked text pieces based on surrounding context. CodeBERT (125M) (Feng et al., 2020) and CuBERT (345M) (Kanade et al., 2020) are examples of such models in code. In programming contexts, these methods provide useful representations of a sequence of code for downstream tasks such as code classification, clone detection, and defect detection.

Encoder-decoder Models (Figure 2, right) An encoder-decoder model first uses an encoder to encode an input sequence, and then uses a left-to-right LM to decode an output sequence conditioned on the input sequence. Popular pretraining objectives include masked span prediction (Raffel et al., 2019) where the input sequence is randomly masked with multiple masks and the output sequence are the masked contents in order, and denoising sequence reconstruction (Lewis et al., 2019) where the input is a corrupted sequence and the output is the original sequence. These pretrained models are useful in many sequence-to-sequence tasks (Raffel et al., 2019). In code, CodeT5 (220M) (Wang et al., 2021), and PLBART (406M) (Ahmad et al., 2021) use the two objectives mentioned above respectively, and performs well in conditional generation downstream tasks such as code commenting, or natural language to code generation.

2.2 PRETRAINING DATA

Some models (e.g. CodeParrot and CodeT5) are trained on GitHub code only, with corpora extracted using either Google BigQuery's GitHub dataset [4], or CodeSearchNet (Husain et al., 2019). Others (e.g., GPT-Neo and GPT-J) are trained on "the Pile" (Gao et al., 2020), a large corpus containing a blend of natural language texts and code from various domains, including Stack Exchange dumps, software documentations, and popular (>100 stars) GitHub repositories. The datasets on which other proprietary models (Codex, Google's) were trained on are unknown. One goal of our study is to try to shed light on what corpora might be the most useful for pretraining models of code.

3 EVALUATION SETTINGS

We evaluate all models using both extrinsic and intrinsic benchmarks, as described below.

Extrinsic Evaluation One of the most popular downstream tasks for code modeling is code generation given a natural language description. Following Chen et al. (2021), we evaluate all models on the HumanEval dataset. The dataset contains 164 prompts with descriptions in the form of code comments and function definitions, including argument names and function names, and test cases to judge whether the generated code is correct. To generate code given a prompt, we use the same sampling strategy as Chen et al. (2021), using softmax with a temperature parameter $\texttt{softmax}(x/T)$. We evaluate using a wide range of temperatures $T = [0.2, 0.4, 0.6, 0.8]$ to control for the confidence of the model's predictions. Similarly to Codex, we use nucleus sampling (Holtzman et al., 2019) with top-$p = 0.95$. We sample tokens from the model until we encounter one of the following stop sequences that indicate the end of a method:[5] '`\nclass`', '`\ndef`', '`\n#`', '`\nif`', or '`\nprint`'. We randomly sample 100 examples per prompt in the evaluation dataset.

Intrinsic Evaluation To evaluate the intrinsic performance of different models, we compute the perplexity for each language on an unseen set of GitHub repositories. To prevent training-to-test data leakage for models such as GPT-Neo and GPT-J, we remove repositories in our evaluation dataset that appeared in the GitHub portion of the Pile training dataset [6]. To evaluate Codex, we use OpenAI's API [7], choosing the `code-davinci-001` engine. We note that the data that this model was trained on is *unknown*, so we cannot prevent data leakage from the training to the test set for Codex. We sampled 100 random files for each of the 12 programming languages in our evaluation dataset. To make perplexity comparable across different tokenization methods used in different models, we use Pygments [8] to equally normalize the log-likelihood sum of each model, when computing perplexity.[9]

4 COMPARED MODELS

4.1 EXISTING MODELS

As discussed in Section 2, we mainly focus on auto-regressive left-to-right pretrained language models, most suitable for code completion tasks.

We evaluate Codex, as it is currently deployed in real-world and has impressive performance in code completion (Chen et al., 2021). Codex uses the GPT-3 language model (Brown et al., 2020) as its underlying model architecture. Codex was trained on a dataset spanning 179GB (after deduplication) covering over 54 million public Python repositories obtained from GitHub on May 2020. As reflected in its impressive results in other programming languages than Python, we suspect that Codex was also trained on large corpora of additional programming languages. The model available for querying through a non-free API.

[4] `https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code`

[5] The absence of whitespace, which is significant in Python, signals an exit from the method body.

[6] `https://github.com/EleutherAI/github-downloader`

[7] `https://beta.openai.com/docs/engines/codex-series-private-beta`

[8] `https://pygments.org/docs/lexers/`

[9] Every model uses its original tokenizer for predicting the next token. We use the shared tokenizer only for computing the perplexity given the log-likelihood sum.

As for open-source models, we compare GPT-Neo, GPT-J and GPT-NeoX, the largest variants having 2.7, 6 and 20 billion parameters, respectively. GPT-NeoX is the largest open-source pretrained language models available. These models are trained on the Pile dataset, so they are a good representatives of models that were trained on both natural language texts from various domains and source code from GitHub. We also compare CodeParrot with at most 1.5 billion parameters, a model that was only trained on Python code from GitHub. CodeParrot follows the process used in Chen et al. (2021) that obtained over 20M files Python files from Google BigQuery Github database, resulting in a 180GB dataset, which is comparable to Codex's *Python* training data, but the model itself is much smaller.

There was no large open-source language model trained almost exclusively on code from multiple programming languages. To fill this gap, we train a 2.7 billion model, PolyCoder, on a mixture of repositories from GitHub in 12 different programming languages.

Language	Repositories	Files	Size Before Filtering	Size After Filtering
C	10,749	3,037,112	221G	55G
C#	9,511	2,514,494	30G	21G
C++	13,726	4,289,506	115G	52G
Go	12,371	1,416,789	70G	15G
Java	15,044	5,120,129	60G	41G
JavaScript	25,144	1,774,174	66G	22G
PHP	9,960	1,714,058	21G	13G
Python	25,446	1,550,208	24G	16G
Ruby	5,826	674,343	5.0G	4.1G
Rust	4,991	304,842	5.2G	3.5G
Scala	1,497	245,100	2.2G	1.8G
TypeScript	12,830	1,441,926	12G	9.2G
Total	147,095	24,082,681	631.4G	253.6G

Table 1: Training corpus statistics.

4.2 POLYCODER'S DATA

Raw Code Corpus Collection GitHub is an excellent source for publicly available source code of various programming languages. We cloned the most popular repositories for 12 popular programming languages with at least 50 stars (stopping at about 25K per language to avoid a too heavy skew towards popular programming languages) from GitHub in October 2021. For each project, each file belonging to the majority-language of that project was extracted, yielding the initial training set. This initial, unfiltered dataset spanned 631GB and 38.9M files.

Data Preprocessing The detailed data preprocessing strategy comparison with other models are analyzed in Table 2. In general, we tried to follow Codex's design decisions, although there is a fair bit of ambiguity in the description of its data preprocessing.

Deduplication and Filtering Similarly to Codex and CodeParrot, very large (>1MB) and very short (<100 tokens) files were filtered out, reducing the size of the dataset by 33%, from 631GB to 424GB. This only reduced the total *number* of files by 8%, showing that a small number of files were responsible for a large part of the corpus.[10]

Allamanis (2019) has shown that code duplication that commonly manifests in datasets of code adversely effects language modeling of code. Therefore, we deduplicated files based on a hash of their content, which reduced the number of files by nearly 30%, and the dataset size by additional 29%, leaving 24.1M files and 254GB of data.

Overall, the filtering of very large and very short files plus deduplication, reduced the number of files by 38%, and the dataset size by 61%, roughly on par with the 70% dataset size reduction reported by CodeParrot. A key difference that remains is that other approaches use more fine-grained filtering

[10]Codex additionally mentions removing "auto-generated" files, but the definition of this was not clear, so we omitted this step.

	PolyCoder	CodeParrot	Codex
Dedup	Exact	Exact	Unclear, mentions "unique"
Filtering	Files $>$ 1 MB, $<$ 100 tokens	Files $>$ 1MB, max line length $>$ 1000, mean line length $>$ 100, fraction of alphanumeric characters $<$ 0.25, containing the word "auto-generated" or similar in the first 5 lines	Files $>$ 1MB, max line length $>$ 1000, mean line length $>$ 100, auto-generated (details unclear), contained small percentage of alphanumeric characters (details unclear)
Tokenization	Trained GPT-2 tokenizer on a random 5% subset (all languages)	Trained GPT-2 tokenizer on train split	GPT-3 tokenizer, add multi-whitespace tokens to reduce redundant whitespace tokens

Table 2: Comparison of data preprocessing strategies of different models.

strategies, such as limiting the maximum line length or average line length, filtering of probable auto-generated files, etc. For example, Chen et al. (2021) have filtered only 11% of their training data.

The dataset statistics are shown in Table 1, showcasing data sizes per language before and after filtering. Our dataset contains less Python code (only 16G) than Codex or CodeParrot, and instead covers many different programming languages.

Tokenizer We train a GPT-2 tokenizer (using BPE (Sennrich et al., 2015)) on a random 5% subset of all the pretraining data, containing all the languages. Codex uses an existing trained GPT-3 tokenizer, with the addition of multi-whitespace tokens to reduce the sequence length after tokenization, as consecutive whitespaces are more common in code than in text.

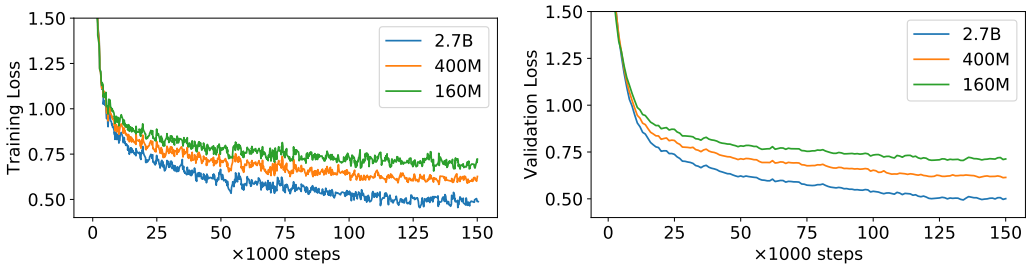
4.3 POLYCODER'S TRAINING

Considering our budget, we chose the GPT-2 (Radford et al., 2019) as our model architecture. To study the effect of scaling of model size, we train 3 different sized models, with 2.7 billion, 400 million and 160 million parameters, as the largest 2.7B model being on par with GPT-Neo for fair comparison. The 2.7 billion model is a 32 layer, 2,560 dimensional Transformer model, with a max context window of 2048 tokens, trained with a batch size of 128 sequences (262K tokens). The model is trained for 150K steps. The 400 million model is a 24 layer, 1,024 dimensional variant, and the 160 million model is a 12 layer, 768 dimensional variant, otherwise idem. We use GPT-NeoX toolkit [11] to train the model efficiently in parallel with 8 Nvidia RTX 8000 GPUs on a single machine. The wall time used to train the largest 2.7B model is about 6 weeks. In its default configuration, this model should train for 320K steps, which was not feasible with our resources. Instead, we adjusted the learning rate decay to half this number and trained for up to 150K steps (near-convergence). The training and validation loss curves for different sized models are shown in Figure 3. We see that even after training for 150K steps, the validation losses are still decreasing. This, combined with the shorter training schedule and faster learning rate decay, strongly signals that the models are still under-fitting and could benefit from longer training.

We compare the training setting and hyperparameters with CodeParrot and Codex in Table 3. Due to high computational costs, we were unable to perform hyperparameter search. Most hyperparameters are the same as those used in their respective GPT-2 model training [12] to provide a good default with regards to the corresponding model size. Some key differences include context window sizes to allow for more tokens as context, batch sizes and tokens trained, as well as model initialization with or without natural language knowledge.

	PolyCoder (2.7B)	CodeParrot (1.5B)	Codex (12B)
Model Initialization	From scratch	From scratch	Initialized from GPT-3
NL Knowledge	Learned from comments in the code	Learned from comments in the code	Natural language knowledge from GPT-3
Learning Rate	1.6e-4	2.0e-4	1e-4
Optimizer	AdamW	AdamW	AdamW
Adam betas	0.9, 0.999	0.9, 0.999	0.9, 0.95
Adam eps	1e-8	1e-8	1e-8
Weight Decay	-	0.1	0.1
Warmup Steps	1600	750	175
Learning Rate Decay	Cosine	Cosine	Cosine
Batch Size (#tokens)	262K	524K	2M
Training Steps	150K steps, 39B tokens	50K steps, 26B tokens	100B tokens
Context Window	2048	1024	4096

Table 3: Comparison of design decisions and hyper-parameters in training different models of code.



(a) Training

(b) Validation

Figure 3: Training and validation loss during the 150K step training process.

5 Results

5.1 Extrinsic Evaluation

The overall results are shown in Table 4.[13] The numbers are obtained by sampling with different temperatures and picking the best value for each metric. Among existing models, PolyCoder is worse than similarly sized GPT-Neo and the even smaller Codex 300M. Overall, PolyCoder lies after Codex, GPT-Neo/J, while performing stronger than CodeParrot. PolyCoder, which was trained only on code, falls behind a similar sized model (GPT-Neo 2.7B) trained on the Pile, a blend of natural language texts and code. Looking at the rightmost columns in Table 4 offers a potential explanation: in terms of total Python tokens seen during training, all models substantially exceed ours. This in partly because they use a higher proportion of Python code (we aimed to balance data volume across programming languages), and in part because of resource limitations, which lead to PolyCoder not observing its entire training data. In addition, the natural language blend in the training corpus may help code language modeling as well, especially with code-related texts such as Stack Exchange dumps being included.

Compared to GPT-Neo (2.7B), PolyCoder has seen fewer Python tokens, but more code tokens in other programming languages, hinting that transfer from other languages to Python helps to achieve a similar performance. This suggests that future research could benefit from blending code in different programming languages, as well as natural language text.

[11] `https://github.com/EleutherAI/gpt-neox`

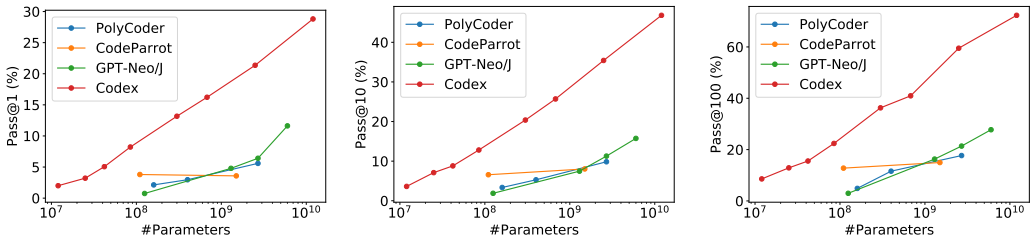
[12] `https://github.com/EleutherAI/gpt-neox/tree/main/configs`

[13] Due to the large model size of GPT-NeoX (20B) and limited computational budget, we did not include it in the HumanEval experiment.

Model	Pass@1	Pass@10	Pass@100	Tokens Trained	Code Tokens	Python Tokens
PolyCoder (160M)	2.13%	3.35%	4.88%	39B	39B	2.5B
PolyCoder (400M)	2.96%	5.29%	11.59%	39B	39B	2.5B
PolyCoder (2.7B)	5.59%	9.84%	17.68%	39B	39B	2.5B
CodeParrot (110M)	3.80%	6.57%	12.78%	26B	26B	26B
CodeParrot (1.5B)	3.58%	8.03%	14.96%	26B	26B	26B
GPT-Neo (125M)	0.75%	1.88%	2.97%	300B	22.8B	3.1B
GPT-Neo (1.3B)	4.79%	7.47%	16.30%	380B	28.8B	3.9B
GPT-Neo (2.7B)	6.41%	11.27%	21.37%	420B	31.9B	4.3B
GPT-J (6B)	11.62%	15.74%	27.74%	402B	30.5B	4.1B
Codex (300M)	13.17%	20.37%	36.27%	100B*	100B*	100B*
Codex (2.5B)	21.36%	35.42%	59.50%	100B*	100B*	100B*
Codex (12B)	28.81%	46.81%	72.31%	100B*	100B*	100B*

*Codex is initialized with another pretrained model, GPT-3.

Table 4: Results of different models on the HumanEval benchmark, and the number of different types of tokens seen during the training process.

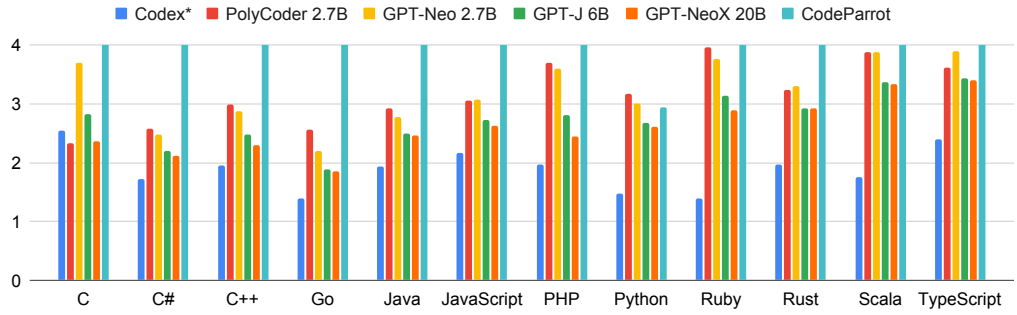


(a) Pass@1 (b) Pass@10 (c) Pass@100

Figure 4: The scaling effect of HumanEval performance on different models.

Scaling Effect To further understand the effect of the number of model parameters with respect to HumanEval code completion performance, we show the Pass@1, Pass@10 and Pass@100 percentage with respect the the model size in Figure 4. We can see that the performance of the Codex models are significantly better than all the other open-source models across all numbers of parameters. The performance on HumanEval benchmark increases linearly with the magnitude (log scale) of the number of parameters in the model. Similar scaling effects could be found on PolyCoder and GPT-Neo/J models. Interestingly, the CodeParrot models that are trained only on Python seem to have reached a saturating performance with respect to increasing number of parameters, where the training corpus being focused on Python may have some effect. With higher number of parameters (2.7B), PolyCoder's performance is trending worse than that of GPT-Neo/J. Comparing GPT-Neo/J that is trained on Pile dataset containing a blend of text, Stack Exchange dumps and GitHub data, with PolyCoder that are trained on only GitHub repositories of popular programming languages, we hypothesize that the added text, especially texts in technical and software engineering domains, may be crucial for the larger model to boost the performance. We also compare the performance difference between the model trained after 100K steps versus the model after 150K steps in Appendix A, and find that training for longer helps the larger model more as it is still under-fitted.

Temperature Effect All the above results are obtained by sampling the language model with different temperatures and picking the best value for each metric. We are also interested in how different choices of temperature affects the final generation quality. We summarize the results in Figure 5. The general trend is for Pass@1, lower temperatures are better, and for Pass@100, a higher temperature will help, while for Pass@10 a temperature in the middle is better suited. We hypothesize that this is because a higher temperature during generation makes the model less confident in its predictions and thus allow for more exploration and more diverse outputs, resulting in better accuracy at Pass@100. Too high a temperature (0.8) is also hurtful if the model is capable enough.



* Since the exact training set of Codex is unknown, it may include files from these test sets rendering Codex's results overly-optimistic.

Figure 6: Perplexity comparison on our evaluation dataset of different models on different programming languages. Note that the y-axis is capped at 4; CodeParrot's entropy on all languages other than Python is much higher than shown here (see Table 5).

On the contrary, a lower temperature makes the model output very confident in its prediction and thus will be better suited for generating very few correct examples, and thus the better performance for Pass@1. In Appendix B we repeat these experiments with the smaller models as well. This suggests the importance of temperature and the need to tune it individually for different generation scenarios.

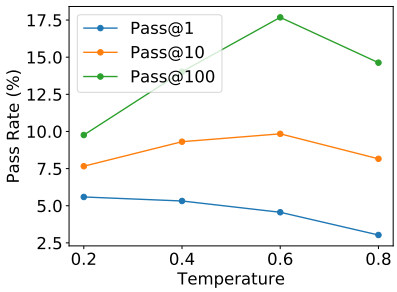


Figure 5: HumanEval performance with different softmax temperatures during generation.

5.2 Intrinsic Evaluation

The perplexity results on the evaluation datasets are shown in Figure 6, with detailed numbers in Appendix C. The plot caps the perplexity score to 4 as CodeParrot performs poorly in languages other than Python. It is important to note that although Codex's perplexities are lower than other models in most languages, Codex might have been trained on the test sets, and its results are thus over-optimistic.

Notably, *PolyCoder outperforms Codex and all other models in the C language*. Comparing the open-source models only, PolyCoder performs better than the similarly sized GPT-Neo 2.7B in C, JavaScript, Rust, Scala and TypeScript.

In the other 11 languages other than C, all other open-source models, including ours, are significantly worse (higher perplexity) than Codex. We hypothesize that this is due to the fact that PolyCoder is trained on an imbalanced mixture of different languages, with C and C++ being closely related and the two most dominant in the entire training corpus (Section 4.2). Thus, the larger volume in total (because of long files) makes C the most "favored" language by PolyCoder. The reason why PolyCoder does not outperform Codex in C++ is possibly due to the complexity of C++ language and Codex's significantly longer context window size (4096, compared to PolyCoder's 2048), or because Codex is possibly trained on more C++ training data.

With the same pretraining corpus, the gain from a 2.7B model (GPT-Neo) to a 6B model (GPT-J) is significant over all languages. However, when increasing the model size further to 20B, the improvement varies across different languages. For example, the performance on Go, Java, Rust, Scala, TypeScript do not increase significantly when the model size increases by 3 times. This suggests that for some programming languages, and given the amounts of data, the capacity of GPT-J is sufficient. Interestingly, these languages seem to coincide with languages where PolyCoder outperforms a similarly sized model trained on Pile. This may hint that for the languages in which larger models do not provide additional gains, training the model only using code may be enough or slightly more helpful than training on both natural language and code.

We can see that comparing different models, perplexity trends for Python correlates well with the HumanEval benchmark performance of the extrinsic evaluation (Section 5.1). This suggests that perplexity is a useful and low-cost metric to estimate other, downstream, metrics.

6 CONCLUSION

In this paper, we perform a systematic evaluation of large language models for code. The performance generally benefits from larger models and longer training time. We also believe that the better results of GPT-Neo over PolyCoder in some languages show that training on natural language text *and* code can benefit the modeling of code. To help future research in the area, we release PolyCoder, a large open-source language model for code, trained exclusively on code in 12 different programming languages. In the C programming language, *PolyCoder achieves lower perplexity than all models including Codex*.

REFERENCES

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2655–2668, Online, June 2021. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2021.naacl-main.211.

Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 143–153, 2019.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *International Conference on Machine Learning*, pp. 245–256. PMLR, 2020.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*, 2018.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. URL https://doi.org/10.5281/zenodo.5297715. If you use this software, please cite it using these metadata.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. GPT-NeoX-20B: An open-source autoregressive language model. 2022.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Alexis Conneau and Guillaume Lample. Cross-lingual language model pretraining. *Advances in Neural Information Processing Systems*, 32:7059–7069, 2019.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 345–356, 2016.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Liang Zhang, Wentao Han, Minlie Huang, et al. Pre-trained models: Past, present and future. *AI Open*, 2021.

Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773, 2017.

Vincent J. Hellendoorn and Anand Ashok Sawant. The growing cost of deep learning for source code. *Commun. ACM*, 65(1):31–33, dec 2021. ISSN 0001-0782. doi: 10.1145/3501261. URL https://doi.org/10.1145/3501261.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pp. 5110–5121. PMLR, 2020.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 1073–1085. IEEE, 2020.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL https://openreview.net/forum?id=6lE4dQXaUcb.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 419–428, 2014.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Lewis Tunstall, Leandro von Werra, and Thomas Wolf. *Natural Language Processing with Transformers*. " O'Reilly Media, Inc.", 2022.

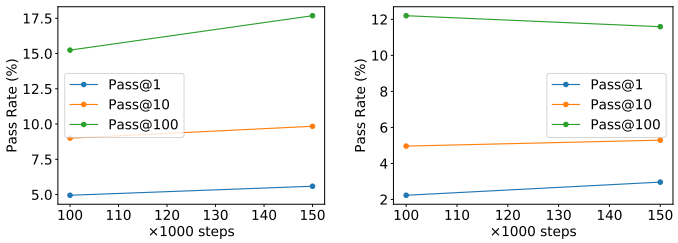
Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. `https://github.com/kingoflolz/mesh-transformer-jax`, May 2021.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=Xh5eMZVONGF`.

A Scaling Effect: Trained Longer

We compare the performance difference between the model trained after 100K steps versus the model after 150K steps in Figure 7. We can see that in the larger 2.7B model, by training the model longer till 150K steps, the performance increases uniformly, with Pass@100 increasing the most. However, for a smaller model such as the 400M model, by training the model longer till 100K steps, the improvements are subdued and Pass@100 drops. This suggests that with the larger model, training for longer may provide additional boost in performance. This echoes with the observation from the training curve (Figure 3) as well.



(a) 2.7B Model (b) 400M Model

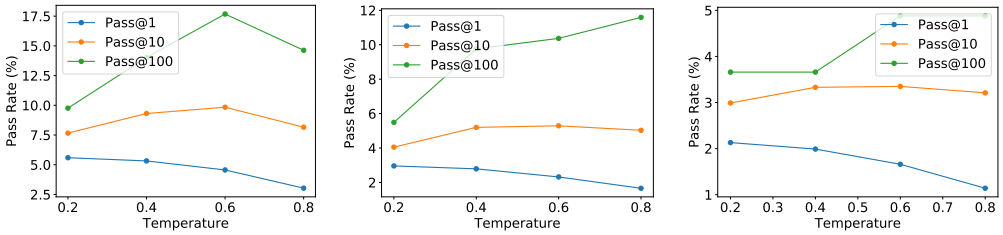
Figure 7: HumanEval performance comparison after training the model for longer.

B Temperature Effect: Smaller Models

We show how temperature affects HumanEval performance on model of all three sizes in Figure 8. We find that for a larger model, e.g., the 2.7B model, a temperature as high as 0.8 is actually hurting the performance for Pass@100, suggesting that if the model is good enough, a very high temperature may cause the outputs to be too diverse, thus hurting the correctness. This suggests the importance of temperature and the need to tune it individually for different model capacity and different generation scenarios.

C Detailed Perplexity Results

We show the detailed perplexity of different models on different languages in Table 5. The number of tokens shown in the table is obtained after tokenizing the code in each language using their respective lexers, by Pygments. This number of tokens is used to normalize the perplexity scores to make them comparable across models. Note that CodeParrot is only trained on Python data and thus performs poorly in other languages.



(a) 2.7B Model (b) 400M Model (c) 160M Model

Figure 8: HumanEval performance using different softmax temperatures during generation.

Language	#tokens	Codex*	PolyCoder 2.7B	GPT-Neo 2.7B	GPT-J 6B	GPT-NeoX	CodeParrot
C	55,333	2.55	2.33	3.69	2.82	2.37	19.23
C#	67,306	1.72	2.58	2.49	2.20	2.12	7.16
C++	69,627	1.95	2.99	2.87	2.47	2.32	8.48
Go	79,947	1.39	2.57	2.19	1.89	1.85	10.00
Java	65,484	1.94	2.92	2.78	2.49	2.47	6.79
JavaScript	54,620	2.17	3.06	3.07	2.73	2.62	9.23
PHP	45,682	1.98	3.70	3.61	2.81	2.45	19.91
Python	79,653	1.47	3.18	3.00	2.68	2.61	2.95
Ruby	46,537	1.39	3.96	3.77	3.13	2.89	14.26
Rust	107,717	1.96	3.24	3.30	2.92	2.92	8.68
Scala	65,756	1.75	3.87	3.88	3.37	3.33	12.91
TypeScript	55,895	2.40	3.61	3.90	3.43	3.41	12.54

* Since the exact training set of Codex is unknown, it might have been trained on these test sets, and Codex's results are over-optimistic.

Table 5: Perplexity of different models for different programming languages on our evaluation dataset.