🎅SantaCoder: don't reach for the stars!🌟

Loubna Ben Allal* Hugging Face
Raymond Li* ServiceNow Research
Denis Kocetkov* ServiceNow Research
Chenghao Mou Independent
Christopher Akiki Leipzig University and ScaDS.AI
Carlos Munoz Ferrandis Hugging Face
Niklas Muennighoff Hugging Face
Mayank Mishra IBM Research
Alex Gu MIT
Manan Dey SAP
Logesh Kumar Umapathi Saama Technologies
Carolyn Jane Anderson Wellesley College
Yangtian Zi Northeastern University
Joel Lamy Poirier ServiceNow Research
Hailey Schoelkopf EleutherAI
Sergey Troshin University of Amsterdam
Dmitry Abulkhanov Huawei Noah's Ark Lab
Manuel Romero Independent
Terry Yue Zhuo Monash University
Francesco De Toni UWA
Bernardo García del Río Flowrite
Qian Liu Sea AI Lab
Shamik Bose Canopy Software
Urvashi Bhattacharyya Discover Dollar Pvt Ltd
Michael Lappert Berner Fachhochschule
Ian Yu PIISA
Paulo Villegas Telefonica I+D
Jia Li Independent
David Lansky Independent
Huu Nguyen Ontocord, LLC
Danish Contractor IBM Research
Luis Villa Independent
Daniel Fried Carnegie Mellon University
Dzmitry Bahdanau ServiceNow Research
Yacine Yernite Hugging Face
Sean Hughes ServiceNow
Arjun Guha Northeastern and Roblox
Harm de Vries‡ ServiceNow Research
Leandro von Werra‡* Hugging Face

Abstract

The BigCode project is an open-scientific collaboration working on the responsible development of large language models for code.[1] This tech report describes the progress of the collaboration until December 2022, out-

*Corresponding authors (denoted by ‡) can be contacted at `contact@bigcode-project.org`
[1]See `https://www.bigcode-project.org`

lining the current state of the Personally Identifiable Information (PII) redaction pipeline, the experiments conducted to de-risk the model architecture, and the experiments investigating better preprocessing methods for the training data. We train 1.1B parameter models on the Java, JavaScript, and Python subsets of The Stack (Kocetkov et al., 2022) and evaluate the models on MultiPL-E (Cassano et al., 2022), a text2code benchmark available in 18 programming languages. We find that more aggressive filtering of near-duplicates can further boost performance and, surprisingly, that selecting files from repositories with 5+ GitHub stars deteriorates performance significantly. Our best model outperforms previous open-source multilingual code generation models (InCoder-6.7B and CodeGen-Multi-2.7B) in both left-to-right generation and infilling on the Java, JavaScript, and Python portions of MultiPL-E, despite being a substantially smaller model. All models are released under an OpenRAIL license at `https://hf.co/bigcode`.

1 Introduction

Over the last two years, we have witnessed tremendous progress in the development of code generating AI assistants (Chen et al., 2021; Chowdhery et al., 2022; Nijkamp et al., 2022; Fried et al., 2022; Li et al., 2022; Athiwaratkun et al., 2022). Machine learning models are now capable of assisting professional developers through the synthesis of novel code snippets, not only from surrounding code fragments, but also from natural language instructions. The models powering these code completion systems are usually referred to as Large Language Models for Code—or code LLMs—and are created by training large transformer neural networks (Vaswani et al., 2017) on big corpora of source code. However, there is a lack of transparency in the research community on the development of these models due to their commercial value and the legal uncertainty around distributing training data and models. Some groups have released model weights (Fried et al., 2022; Nijkamp et al., 2022) or provided access to the model through a paid API service (Chen et al., 2021; Athiwaratkun et al., 2022), but these papers did not release the full training data or the preprocessing methods that were used.

BigCode is an open scientific collaboration working on the responsible development of large language models for code, empowering the machine learning and open-source communities through open governance. Various BigCode working groups focus on relevant subtopics such as collecting datasets, implementing methods for training code LLMs, developing an evaluation suite, and discussing ethical best practices for these powerful models. For example, the Legal, Ethics, and Governance working group has explored questions on data licensing, attribution of generated code to original code, the redaction of Personally Identifiable Information (PII), and the risks of outputting malicious code. In earlier work, the BigCode community released The Stack v1.1 (Kocetkov et al., 2022), a 6.4 TB dataset of permissively licensed source code in 384 programming languages. That work also introduced "Am I in The Stack",[2] a governance tool for developers to check whether their source is part of the dataset, and an opt-out form for those who wish to have their code removed from the dataset.[3]

In this tech report, we summarize the learnings of the BigCode community in developing the Santa models, a set of 1.1B-parameter models trained on the Java, JavaScript, and Python subsets of The Stack and evaluated on MultiPL-E (Cassano et al., 2022). We describe the first steps of the community towards developing larger code models and report experiments to de-risk the model architecture and the data processing pipeline. Specifically, the contributions of this report can be summarized as follows:

- We describe the current state of the PII redaction pipeline. We detail how we create a PII benchmark of 400 code files, describe the filters for detecting emails, ip

[2] `https://huggingface.co/spaces/bigcode/in-the-stack`
[3] `https://www.bigcode-project.org/docs/about/the-stack/`

addresses, and secret keys, and analyze its performance on the annotation benchmark. All experiments in this work are conducted on the PII-redacted version of The Stack.

- We run ablations for Multi Query Attention (MQA) (Shazeer, 2019; Chowdhery et al., 2022; Li et al., 2022) and Fill-in-the-Middle (FIM) (Fried et al., 2022; Bavarian et al., 2022). MQA can significantly speed-up inference for larger batch sizes, while FIM enables code models to do infilling tasks. We find that both changes only slightly deteriorate downstream performance compared to baseline models.
- We investigate the impact of 4 preprocessing methods on the training data: filtering files from repositories with 5+ GitHub stars, filtering files with a high comments-to-code ratio, more aggressive filtering of near-duplicates, and filtering files with a low character-to-token ratio. We observe modest impact of the new filters except for the stars filter, which deteriorates performance on text2code benchmarks significantly. This is an interesting result given that previous work has explicitly filtered for GitHub Stars as a proxy for data quality (Gao et al., 2020).
- Using the findings from these experiments, we train a final 1.1B parameter model, dubbed SantaCoder, on Python, JavaScript, and Java. This model obtains comparable or stronger performance than previous open-source multilingual models, InCoder-6.7B and CodeGen-Multi-2.7B, on code generation and infilling tasks on the MultiPL-E benchmark for these three languages, despite being substantially smaller.

2 Related Work

Code LLMs Recently, there has been an increasing amount of research on using large-scale transformer models to analyze or generate source code. Many studies have focused on using decoder-only models with a causal language modeling objective (Chen et al., 2021; Austin et al., 2021; Nijkamp et al., 2022; Christopoulou et al., 2022; Izadi et al., 2022; Xu et al., 2022; Athiwaratkun et al., 2022), while other studies have investigated encoder (Feng et al., 2020a; Kanade et al., 2020) and encoder-decoder architectures (Li et al., 2022; Ahmad et al., 2021; Wang et al., 2021; Roziere et al., 2021). Bavarian et al. (2022); Fried et al. (2022) propose to use decoder-only models for code-infilling tasks using a causal masking mechanism, and Bavarian et al. (2022) argues that training with such a fill-in-the middle (FIM) objective does not harm the model's ability to do left-to-right generation. Shazeer (2019) proposes Multi Query Attention (MQA), an architectural change to the transformer neural network in which key and value embeddings are shared across attention heads, resulting in lower memory requirements and faster inference for large batch settings. Multi Query Attention was implemented in AlphaCode (Li et al., 2022) and PaLM (Chowdhery et al., 2022).

Evaluating text to code The correctness of generated code can be tested using *unit tests*, a method for approximating semantic equivalence. Textual similarity metrics have also been used to evaluate code (Feng et al., 2020b; Ren et al., 2020); however, they have been shown to correlate only weakly with code correctness (Austin et al., 2021; Chen et al., 2021).

Many single-language benchmarks for evaluating code completion exist (Kulal et al., 2019; Iyer et al., 2018; Zhong et al., 2017; Yu et al., 2018; Austin et al., 2021; Hendrycks et al., 2021; Chen et al., 2021; Austin et al., 2021; Athiwaratkun et al., 2022; Lai et al., 2022). Two of the most popular benchmarks for Python are HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), which consist of a natural language description of a function and a set of unit tests.

MultiPL-E (Cassano et al., 2022) extends two popular benchmarks for code completion, MBPP and HumanEval, to 18 additional languages. The doctests, function signatures, and unit tests for each benchmark suite are automatically compiled to new languages. Python-specific terminology in the prompt is automatically replaced with the equivalent terminology used for each programming language. MBXP (Athiwaratkun et al., 2022) is a concurrent

benchmark that uses a similar approach, but differs in the details of type inference, prompt construction, and evaluation. In particular, MBXP uses the same set of assertions in the prompt that it uses to test the correctness of generated solutions. In contrast, MultiPL-E keeps the tests hidden from the model and only uses them to test correctness.

Evaluating other tasks Code generation models have also been used to solve a variety of tasks (Tufano et al., 2020; Feng et al., 2020b; Ahmed & Devanbu, 2022; Hellendoorn et al., 2018; Pradel et al., 2020). CodeXGLUE (Lu et al., 2021) is a set of 14 datasets for evaluating code generation models. The tasks include code-to-code tasks like clone detection, code repair, and code translation; text-to-code tasks like code search and code generation; and code-to-text tasks like generating documentation. The programming languages included vary by task; the most common are Python and Java.

3 Opt-out process

Developers who do not wish their source code to be used for training code LLMs are given the opportunity to opt-out of The Stack (Kocetkov et al., 2022). We received 9 opt-out requests before the cut-off date for removing data (31 October 2022). These individuals accounted for 299 repositories. Of these, 161 were already excluded from The Stack v1.0 (because they did not have a permissive license), and 138 were in The Stack v1.0. We honored the requests to opt-out and removed these repositories from The Stack v1.1. After the cut-off date (31 October 2022), we have received more requests for requests and we will remove these repositories prior to releasing The Stack v1.2.

4 Redacting Personally Identifiable Information

We describe our first efforts to redact PII from The Stack.

4.1 PII benchmark

We construct a PII benchmark by annotating the following entities on a small subset of The Stack: names, emails, usernames, passwords, IP addresses, API keys, and SSH keys. We pre-filtered 400 samples from a total of 4000 code files that were likely to contain Personally Identifiable Information (PII). We first select 4000 code files from 11 programming languages, with a total of 800 samples for Python and C++, 400 samples for Java, JavaScript, TypeScript, and PHP, and 160 samples for C, C#, Markdown, Go, and Ruby. To detect keys in these samples, we used the detect-secrets tool[4] with all default plugins activated. In addition, we used regular expressions to detect emails, IPv4 and IPv6 addresses, see Appendix C.1. Twelve members of the BigCode community annotated the files on the LightTag platform,[5] with one annotator assigned per file. After the annotation phase, one member reviewed all the annotation tags. To further increase annotation quality, we ran our initial PII detection tools on the annotated files and manually corrected any incorrect annotations identified as false positives or false negatives.

4.2 PII detection and redaction

For the first iteration of the PII redaction pipeline, we focus on emails, IP addresses, and keys, and leave the detection of names, usernames, and passwords for future work.

Emails We use a regular expression to detect emails, see Appendix C.1. We replace detected emails with [random 5 character string]@example.com.

IP addresses We use regular expressions for IPv4 and IPv6 IP addresses, see Appendix C.1. In addition, we check if the detected IP addresses have a valid format using the

[4] `https://github.com/Yelp/detect-secrets`
[5] `https://www.lighttag.io/`

`ipaddress` python package. We also do not select IP addresses of the format a.b.c.d where a, b, c and d are single digit numbers, except if the words "dns" or "server" appear in the neighboring context (100 characters before or after). These detected addresses were mostly false positives, consisting of package and release versions. Lastly, we do not anonymize private IP addresses[6] and popular DNS servers, as we don't consider them sensitive information. See Appendix C.2 for the full list.

We replace detected IP addresses with one of 5 randomly generated IP addresses.

Keys We employed the `detect-secrets` tool to identify secret keys in the code files. To this end, we kept all the regex and entropy based plugins, including the AWS key detector, the GitHub Token detector, the Azure storage key detector, and the Base64 High Entropy String detector. You can find the full list of plugins at `https://github.com/bigcode-project/bigcode-dataset/blob/6b3f54751b6e38e1ed70f2307331d6943ba39eae/pii/utils/keys_detection.py#L19`.
We deactivated keyword detectors because they were detecting commonly used words like "password" rather than actual secret keys. To remove false positives, we activated filters like UUIDs and string-like secret filtering, see the full list at `https://github.com/bigcode-project/bigcode-dataset/blob/6b3f54751b6e38e1ed70f2307331d6943ba39eae/pii/utils/keys_detection.py#L11`.
We also observed that entropy detectors sometimes detected human-readable text like paths and URLs as secrets, even when adjusting the entropy threshold. To address this issue, we added a `gibberish`[7] detector filter on top of detect-secrets to verify that the detected string was actually gibberish. Additionally, we noticed that hashes were sometimes falsely detected as secret keys. To mitigate this problem, we added a hash filter that verifies the size of the detected string and checks for the presence of keywords like "sha", "md5", "hash", and "byte" in the neighboring context. Finally, to avoid corrupting any files, we prevent the removal of keys from files where words like "sha" or "hash" are mentioned in more than 2% of the number of lines.

4.3 Performance analysis

Evaluation on PII benchmark We evaluated our PII detection pipeline on the benchmark we annotated. The 400 files contained 214 emails, 99 IP addresses and 34 secret keys. Figure 1 shows the precision and recall for each PII entity. Email and IP address detection perform well, with a precision and recall above 90% for emails and above 80% for IP addresses. While key detection also achieves almost 80% precision, its recall is much lower (slightly above 50%). We found that the key detection pipeline was especially sensitive to the precision-recall trade-off, as including more plugins or disabling some filters detected more keys but also increased the number of false positives.

PII detection on The Stack We run the PII pipeline on the Python, Java and JavaScript subsets of The Stack v1.1 (Kocetkov et al., 2022). Table 1 shows some statistics on the number of files containing PII and the total number of secrets found. Some files containing PII are not modified if they contain only private IP addresses or popular DNS servers, as explained in the previous section. The number of files containing PII is significantly lower for JavaScript compared to Python and Java, but this could be due to the fact that JavaScript files were filtered based on line length and percentage of alphanumeric characters before running PII detection. We also observe that Python and JavaScript have a higher number of secrets per file compared to Java.

To better understand these results, we computed the relevant percentiles in Table 2. We can see that Java indeed has fewer secrets per file, and that almost 0.1% of the files contain a large number of secrets (about 100). We found that some of these files contained multiple instances of PII, such as emails stored in some form of database, or are files containing long encodings and key-like strings that are split into multiple keys. Finally, we also plot the distributions of detected secrets by entity type in Figure 2. For this graph, we filtered

[6] They are non-internet facing IP addresses used in internal networks

[7] `https://github.com/domanchi/gibberish-detector`

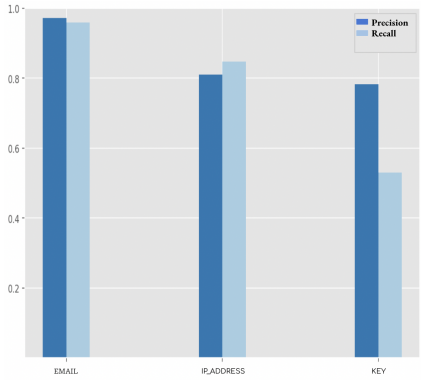

Figure 1: Precision and recall of PII detection tools.

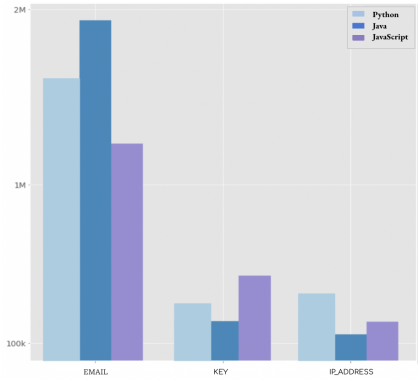

Figure 2: Distribution of PII detected in The Stack for Python, Java and JavaScript.

Language	# files	# files with PII	# secrets	# modified files
Python	15,148,604	1,224,632	3,255,053	1,040,809
Java	25,124,914	1,588,453	2,757,169	1,506,766
JavaScript*	23,670,848	835,198	2,468,183	744,842

Table 1: Statistics from running PII detection on The Stack. JavaScript files initially went through line-length filtering. Modified files are those altered during PII redaction.

out files with more than 100 secrets, but this did not change the distribution of PII across languages. We observe that IP addresses are most often found in Python, keys in JavaScript and emails in Java.

5 Experiments

5.1 Dataset, model, and training details

Dataset The base training dataset for the experiments in this paper contains 268 GB of Python, Java and JavaScript files from The Stack v1.1 (Kocetkov et al., 2022) after removing data from opt-out requests, near-deduplication, PII-redaction (see Section 4), and filtering based on line-length and percentage of alphanumeric characters. This dataset was also decontaminated by removing files that contained test-samples from the following benchmarks: HumanEval (Chen et al., 2021), APPS (Hendrycks et al., 2021), MBPP (Austin et al., 2021) and MultiPL-E (Cassano et al., 2022).

Tokenizer Seeing as the Santa models were the first models our community would train, our design choices for the tokenizer were modulated by a conservative approach, partly based on insights developed during the development of InCoder (Fried et al., 2022). We train a Hugging Face Tokenizer (MOI et al., 2022) using the Byte-Pair Encoding (BPE) algorithm on raw bytes with a vocabulary size of 49,152 tokens. This tokenizer was trained on 600,000 rows (Around 2.6 GB) of data—200,000 for each language—which were pre-tokenized using a digit splitter and the default GPT-2 pre-tokenizer regex before being converted to bytes.

Training details Our base model is a 1.1B-parameter decoder-only transformer with FIM and MQA trained in `float16`. It has 24 layers, 16 heads and a hidden-size of 2048. The model is trained for 300K iterations with a global batch-size of 192 using Adam (Kingma & Ba, 2015) with $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$ and a weight-decay of 0.1. A total of 118B tokens are seen in training. The learning-rate is set to 2×10^{-4} and follows a cosine decay after warming up for 2% of the training steps. Each training run takes 3.1 days to complete

Language	mean	median	95th percentile	99th percentile	99.9th percentile
Python	2.7	1	6	23	135
Java	1.7	1	3	11	63
JavaScript	3.3	1	7	30	197

Table 2: Statistics of the number of detected PII per file in The Stack.

Language	Base	Stars	Comments-to-code	Near-dedup	Tokenizer fertility
Python	75.6 GB	26.6 GB	65.6 GB	62.0 GB	72.5 GB
Java	110 GB	35.8 GB	92.7 GB	88.4 GB	105.5 GB
JavaScript	82.7 GB	20.8 GB	57.5 GB	65.1 GB	76.4 GB

Table 3: Data volume after additional filtering of the Python, Java, JavaScript subsets of The Stack.

on 96 Tesla V100 GPUs for a total of 1.05×10^{21} FLOPs. The final model described in Section 6.2 uses twice the amount of compute.

5.2 Architecture ablations

We perform ablation experiments to de-risk the model architecture and training objective. Specifically, we investigate Fill-in-the-Middle (Bavarian et al., 2022) and Multi Query Attention (MQA) (Shazeer, 2019).

FIM vs No-FIM Recent works (Fried et al., 2022; Bavarian et al., 2022) have shown that autoregressive language-models can learn to infill code snippets by random transformation of the training data. Bavarian et al. (2022) argue that such data transformations do not harm the left-to-right generative capabilities of the model. Following Bavarian et al. (2022), we implement FIM as a random transformation of the input sequence and split each training document into three parts uniformly at random: prefix, middle and suffix. Each part is prepended with a corresponding sentinel token, then documents are rearranged to put the middle part at the end of the sequence. The autoregressive training objective is unchanged. We use context-level FIM, apply transformations at the character level, use a FIM-rate of 0.5 and SPM+PSM joint training. We compare our base model to a model that was trained with the standard left-to-right objective only (No-FIM).

Multi Query Attention vs Multi Head Attention Shazeer (2019) proposes Multi Query Attention (MQA), an architectural change to transformer that shares key and value embeddings across attention heads. Compared to Multi Head Attention (MHA), this lowers the memory bandwidth requirements at generation time and results in faster inference. We compare our base model to a similar model using MHA instead, with the same hyperparameters otherwise. Note that the MHA model has more parameters (1.3B) than the base model in this setting.

5.3 Data filtering ablations

We experiment with a number of preprocessing methods applied to the base dataset, described in Section 5.1. Note that the filters are applied on top of the other filters such as near-deduplication, line length filtering, etc.

GitHub stars Do popular repositories contain good quality code? We use GitHub stars as a proxy metric. We set the minimum threshold to 5 stars, as we believe that a lower number of stars would not be an indicator of popularity. This filter removes more than 60% of the data (in terms of volume), see Table 3. Note that more than 40% of the files do not have stars and that setting the threshold to 10 stars would remove an additional 5% of the data.

Language	Attention	FIM	HumanEval	MBPP
Java	Multi Query Attention	✓	0.35	0.54
	Multi Head Attention	✓	0.36	0.55
	Multi Query Attention	✗	0.37	0.55
JavaScript	Multi Query Attention	✓	0.33	0.64
	Multi Head Attention	✓	0.37	0.67
	Multi Query Attention	✗	0.37	0.65
Python	Multi Query Attention	✓	0.36	0.67
	Multi Head Attention	✓	0.38	0.70
	Multi Query Attention	✗	0.39	0.68

Table 4: Pass@100 results for the architecture ablations on HumanEval and MBPP.

Comment-to-code ratio Good code should be well documented. With this assumption, we filter files with a high comments-to-code ratio. We use the `ast` and `tokenize` modules to extract docstrings and comments from Python files, and `Pygments` to extract comments from Java and JavaScript files. We then analyze the comment-to-code character ratio. We find that about 20% of Python and Java files and 45% of JavaScript files have no comments. We use a minimum threshold of 1%, removing an additional 3% of files in each language. We also find that files with a ratio above 80% have poor quality, so we filter them out, eliminating 2% of data in all languages. Overall, this comment-to-code filter removes 20% of the data in terms of volume.

More near-deduplication Previous work (Kocetkov et al., 2022) has demonstrated the effectiveness of deduplication in boosting the performance of code LLMs. Based on this finding, we investigate whether more aggressive near-deduplication can further improve performance. To this end, we conduct experiments on a 100K subset of the base dataset. In the original deduplication pipeline, we implemented a false positive check on top of the `MinHash LSH`[8] output. This added processing time, but was necessary due to a high false positive rate of around 15%. To remove more duplicates while maintaining a low false positive rate and a low false negative rate, we switch to using 5-gram for min-hashing, and 0.7 for the Jaccard Similarity threshold, without any additional false positive checks after the initial near-deduplication. As a result, we see additionally 16%–20% fewer files than the original already-deduplicated base dataset (see Table 3), and a decrease in both the estimated false positive rate (from 15% to 5%) and the estimated false negative rate for documents with high similarities (from 35% to 24%).

Tokenizer fertility Can we use the tokenizer to remove low-quality files from the dataset? We experiment with filtering files with a low character-to-token ratio[9]. For each language, we find that files with a ratio below the 5th percentile are usually of poor quality, but increasing the threshold may eliminate some good-quality files. We therefore set the cutoff value for this ratio to the following values: 2.5 for Python, 2.9 for Java, and 2.6 for JavaScript. This filters out roughly 4% to 5% of data. Note that these values depend highly on the tokenizer and the data. This filter may also be biased against files with non-English comments.

5.4 Evaluation

Text2code evaluation The text2code task involves generating the body of a function from a prompt that includes a function description, the function signature (its name and arguments), and optionally a handful of example inputs and outputs. Every problem is accompanied by a set of hidden test cases, which are used to determine if the generated function is correct. We use the MultiPL-E text2code benchmark Cassano et al. (2022),

[8] `https://github.com/ekzhu/datasketch`

[9] We slightly abuse the term tokenizer fertility in this work as it usually refers to the average number of subwords per token, where a token is determined by the true tokenizer of the programming language. See e.g. (Rust et al., 2021)

Model	Java	JavaScript	Python
Baseline	0.64	0.61	0.42
GitHub stars	0.54	0.57	0.37
Comments-to-code	0.62	0.59	0.44
More near deduplication	0.66	0.57	0.45
Tokenizer fertility	0.67	0.65	0.45
Final	0.62	0.60	0.44

Table 5: Fill-in-the-middle results for the data filtering ablations on MultiPL-HumanEval. Each number reports the fraction of lines where the model exactly reproduces a single line of code that is held out from the body of a function in a held out problem.

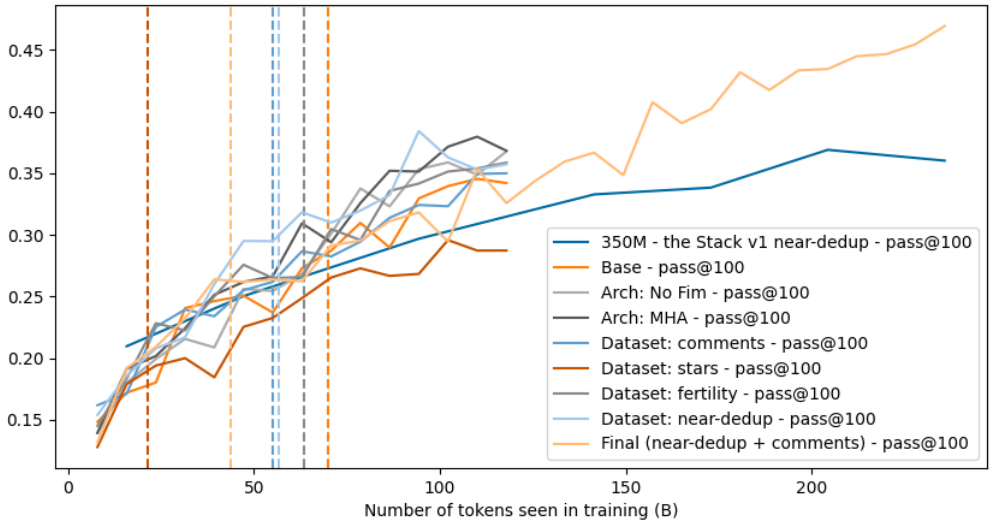

Figure 3: HumanEval pass@100 performance throughout training for all models. Note that evaluation shown here is based on OpenAI Python prompts and might differ (slightly) from the MultiPL-E prompts used in the rest of this paper.

which is derived from HumanEval Chen et al. (2021) and MBPP Austin et al. (2021) (the "sanitized" subset of MBPP.). Whereas the latter two benchmarks target Python, MultiPL-E has a suite of compilers that translate HumanEval and MBPP to 18 other programming languages. Since our models are only trained on Java, JavaScript, and Python, we only evaluate them on these three languages.

We use the methodology of Chen et al. (2021) and we calculate pass@k rates for ($k = 1, 10, 100$) for every problem. Intuitively, pass@1 estimates the likelihood a model will generate a correct solution in a single attempt, whereas pass@10 and pass@100 estimate the likelihood that the model will generate a correct solution given 10 and 100 attempts respectively. Following the literature, we sample 200 completions at temperatures 0.2 and 0.8 and use 0.2 to estimate pass@1 and 0.8 for pass@10 and pass@100.

Fill-in-the-middle evaluation To evaluate fill-in-the-middle, we use the single-line exact match metric, which was introduced by Fried et al. (2022) and also employed by Bavarian et al. (2022). For every benchmark problem, we mask out a single line of text from the function body (i.e., not from the function description or signature), and prompt the model to fill in that line of code. We exclude blank lines and comments, and count the number of times the model produces exactly the masked out line. This benchmark requires working solutions for problems, which MultiPL-E does not have. (A text2code benchmark like MultiPL-E only needs hidden tests.) Instead, of writing solutions by hand, we use solutions generated by a code generation model, which is the approach of Athiwaratkun et al. (2022).

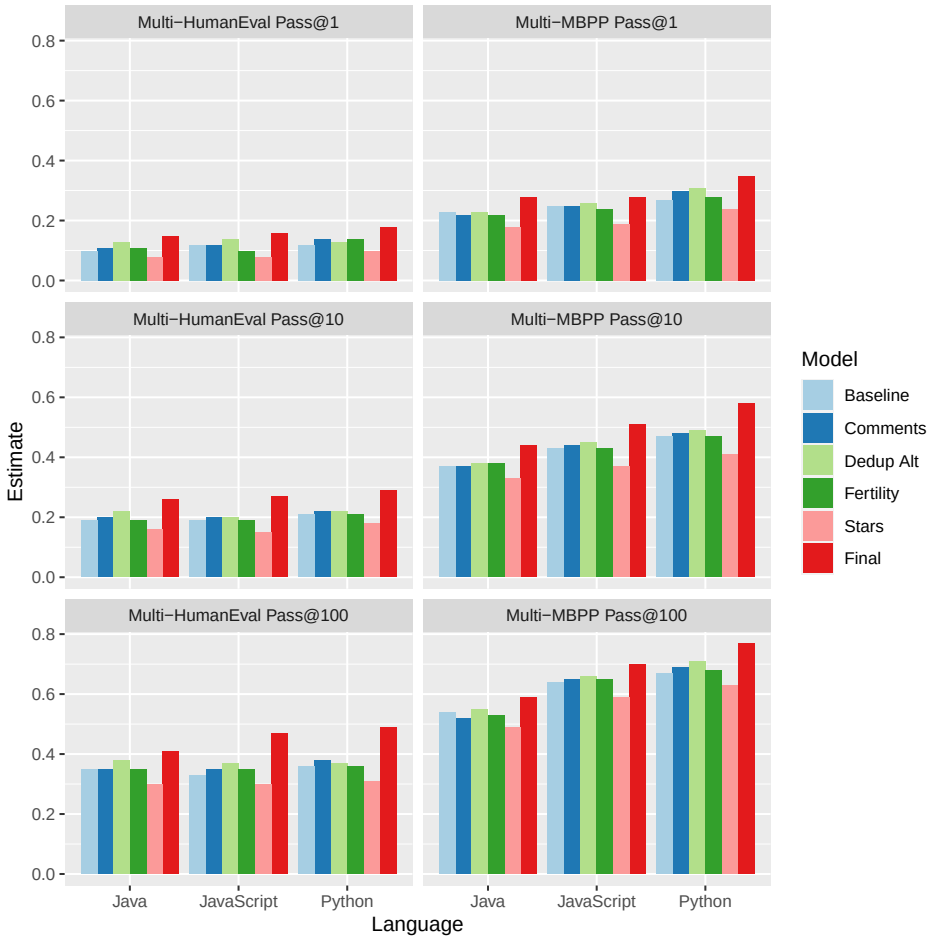

Figure 4: Pass@k rates on Multi-HumanEval and Multi-MBPP by model and language

Specifically, we use working solutions produced by `code-davinci-002` at temperature 0.8. Note that this approach does not produce solutions to every problem, since not all problems are solvable. Moreover, for uniformity, we use this approach for Python as well, even though hand-written Python solutions exist for our benchmarks. We only report fill-in-the-middle evaluations for the data filtering ablations.

6 Results

6.1 Ablations

For the architecture ablations, we report the results on text2code benchmarks in Table 4. For the data filtering ablations, we show the text2code results in Figure 4 and report the fill-in-the middle evaluations in Table 5. We show the HumanEval performance throughout all training runs in Figure 3. You can find the full results tables of the text2code experiments are Appendix A.

Slight drop in performance for MQA We see a small drop in performance for Multi Query Attention (MQA) compared to Multi Head Attention (MHA). As shown in Table 4, the MHA model improves pass@100 with 1-4% on HumanEval and with 1-3% on MBPP. We specifically observe noticeable improvements for the JavaScript versions of the text2code benchmarks. However, it should be noted that the MHA model has more parameters (1.3B) than the MQA model (1.1B), and a head-to-head comparison might, therefore, not be

Model	Size	Left-to-right pass@100			Fill-in-the-middle ex. match		
		Java	JavaScript	Python	Java	JavaScript	Python
InCoder	6.7B	0.36	0.38	0.47	0.49	0.51	0.31
CodeGen-multi	2.7B	0.42	0.39	0.39	✗	✗	✗
CodeGen-mono	2.7B	✗	✗	0.57	✗	✗	✗
Codex[10]	2.5B	✗	✗	0.60	✗	✗	✗
SantaCoder	1.1B	0.41	0.47	0.49	0.62	0.60	0.44

Table 6: Comparing the performance of the final version of SantaCoder with InCoder (Fried et al., 2022), CodeGen (Nijkamp et al., 2022), and Codex (Chen et al., 2021) on left-to-right (HumanEval pass@100) and fill-in-the-middle benchmarks (HumanEval line filling, exact match).

entirely fair. We think that the inference speed-ups of MQA might outweigh the small drop in performance.

FIM for cheap We observe a minor drop in performance of the FIM model compared to the No-FIM model. Specifically, we see that the pass@100 performance of the FIM model is 2-4% lower on HumanEval and 1% lower on MBPP. While Bavarian et al. (2022) presented evidence for the existence of a FIM-for-free property (i.e., arguing that autoregressive models can be trained with FIM without harming left-to-right capabilities), we do find a small but consistent drop of FIM models on left-to-right text2code benchmarks.

Modest impact of near-deduplication, comments, and fertility filter On text2code benchmarks, we observe small gains for the near-deduplication and comment-to-code filters and a neutral effect of the tokenizer filter. The near-deduplication filter improves HumanEval performance by 1-3% and MBPP by 1-4% across the three programming languages. The comment-to-code filter improves HumanEval performance by 0-2% but decreases MBPP performance in certain cases (Java). See Appendix A for the full results table. On fill-in-the-middle benchmarks, we see that the tokenizer fertility filter performs well, improving performance by 2-4% across the three languages. The near-duplication and comments filters have a mixed effect, improving fill-in-the-middle performance for Python but deteriorating performance for JavaScript.

GitHub stars deteriorate performance Surprisingly, we find that the GitHub stars filter performs poorly. On HumanEval and MBPP, the pass@100 performance consistently drops by 3-6% across the three languages. On the fill-in-the-middle benchmark, the performance drops by 5-11% (Table 5). Note that the stars filter removes the most data (over 60%) and, therefore, raises the question whether the performance difference is due to the smaller dataset. However, as can be seen in Figure 3, HumanEval pass@100 diverged early on in training, indicating that the drop in performance is not only due to data size but also data quality.

6.2 Final model

Based on the insights from the architecture and dataset ablations, we train a final model, which we call SantaCoder, with MQA and FIM and the two data filters that yielded the best results: more near-deduplication and comments-to-code filter. We train this model for 600K iterations (236B tokens) and keep all other hyper-parameters the same.

Improved text2code performance Doubling the training iterations leads to much stronger text2code performance on MultiPL-E, significantly boosting performance across all benchmarks and programming languages (see Figure 4). Looking at the performance

[10]This is the performance of a Codex model reported by Chen et al. (2021). It is not clear if this model is available via the OpenAI API.

throughout training (Figure 3), it is likely that longer training can further increase performance. Surprisingly, we find that the final training run did not improve the fill-in-the-middle evaluations (see Table 5).

Comparison to InCoder, CodeGen, and Codex Table 6 compares our SantaCoder model to comparably-sized code generation models from previous work on the MultiPL-E benchmark, using the methodology described in Section 5.4. We find that our model generally outperforms previous open-source multi-language code generation models despite being smaller, outperforming the InCoder 6.7B Fried et al. (2022) model on both left-to-right generation and single line fill-in-the-middle infilling across languages, and obtaining comparable or stronger performance to CodeGen-multi 2.7B Nijkamp et al. (2022).

7 Conclusion

We described the progress of the BigCode project until December 2022. The community took its first steps towards redacting PII and demonstrated that regular expressions are reasonably effective at detecting emails and IP addresses. Future work should focus on increasing the precision and recall of secret keys, as well as detecting other sensitive information such as names, usernames, and password. Using the PII-redacted version of The Stack, we conducted a series of architectural and data filtering ablations. One of our main findings was that filtering for Github stars consistently decreased performance across all benchmarks and programming languages. Using the findings of these ablation studies, we trained a final 1.1B model—dubbed SantaCoder—for 236B tokens and showed it's able to outperform previous multi-lingual code models (InCoder-6.7B and CodeGen-Multi-2.7B) on both left-to-right generation and infilling tasks. We anticipate that larger architectures and more training data will be able to produce stronger multilingual, infilling-capable models, and plan to continue to scale the findings from our investigations here.

8 Contributions

Model license Carlos Munoz Ferrandis, Christopher Akiki, Danish Contractor, Harm de Vries, Huu Nguyen, Leandro von Werra, Luis Villa, Sean Hughes, Yacine Jernite, David Lansky

PII redaction Loubna Ben Allal, Jia Li, Paulo Villegas, Harm de Vries, Leandro Von Werra, Christopher Akiki, Ian Yu, Michael Lappert, Urvashi Bhattacharyya, Shamik Bose, Bernardo García del Río, Francesco De Toni, Terry Yue Zhuo, Qian Liu, Manuel Romero

Dataset Denis Kocetkov, Chenghao Mou, Loubna Ben Allal, Leandro von Werra, Dmitry Abulkhanov, Christopher Akiki, Raymond Li

Tokenizer Christopher Akiki, Sergey Troshin, Dmitry Abulkhanov, Daniel Fried, Leandro von Werra, Harm de Vries

Training and architecture Raymond Li, Daniel Fried, Hailey Schoelkopf, Joel Lamy Poirier, Qian Liu, Niklas Muennighoff, Dzmitry Bahdanau, Harm de Vries, Leandro von Werra

Opt out Sean Hughes, Carlos Munoz Ferrandis, Christopher Akiki, Denis Kocetkov, Harm de Vries, Huu Nguyen, Leandro von Werra, Luis Villa

Evaluation Arjun Guha, Yangtian Zi, Carolyn Jane Anderson, Loubna Ben Allal, Raymond Li, Niklas Muennighoff, Manan Dey, Logesh Kumar Umapathi, Leandro von Werra, Harm de Vries

Inference Mayank Mishra, Alex Gu, Joel Lamy Poirier, Leandro von Werra, Harm de Vries

Acknowledgement We thank ServiceNow and HuggingFace for the provided compute resources.

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.

Toufique Ahmed and Premkumar Devanbu. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 2022. doi: 10.1145/3510003.3510049.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multilingual evaluation of code generation models, 2022. URL https://arxiv.org/abs/2210.14868.

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.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle, 2022. URL https://arxiv.org/abs/2207.14255.

Loubna Ben Allal, Niklas Muennighoff, and Leandro Von Werra. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. A scalable and extensible approach to benchmarking nl2code for 18 programming languages, 2022. URL https://arxiv.org/abs/2208.08227.

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. Evaluating large language models trained on code. 2021.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022. doi: 10.48550/arXiv.2204.02311. URL https://doi.org/10.48550/arXiv.2204.02311.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jiansheng Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. Pangu-coder: Program synthesis with function-level language modeling, 2022. URL https://arxiv.org/abs/2207.11280.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020a. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020b. doi: 10.48550/ARXIV.2002.08155. URL https://arxiv.org/abs/2002.08155.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis, 2022. URL https://arxiv.org/abs/2204.05999.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800GB dataset of diverse text for language modeling, 2020.

Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Laria Reynolds, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, September 2021. URL `https://doi.org/10.5281/zenodo.5371628`.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep Learning Type Inference. In *Fse*, 2018.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*, 2021. doi: 10.48550/ARXIV.2105.09938. URL `https://arxiv.org/abs/2105.09938`.

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.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*, 2018.

Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: Multi-token code completion by jointly learning from structure and naming sequences. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pp. 401–412, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510172. URL `https://doi.org/10.1145/3510003.3510172`.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1412.6980`.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The Stack: 3 TB of permissively licensed source code. *Preprint*, 2022.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf`.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Anthony MOI, Nicolas Patry, Pierric Cistac, Pete, Funtowicz Morgan, Sebastian Pütz, Mishig, Bjarte Johansen, Thomas Wolf, Sylvain Gugger, Clement, Julien Chaumond, Lysandre Debut, François Garillot, Luc Georges, dctelus, JC Louis, MarcusGrass, Taufiquzzaman Peyash, 0xflotus, Alan deLevie, Alexander Mamaev, Arthur, Cameron, Colin Clement, Dagmawi Moges, David Hewitt, Denis Zolotukhin, and Geoffrey Thomas. huggingface/tokenizers: Rust 0.13.2, November 2022. URL `https://doi.org/10.5281/zenodo.7298413`.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *arXiv preprint*, 2022.

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Esecfse*, 2020.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020. URL `https://arxiv.org/abs/2009.10297`.

Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. *arXiv preprint arXiv:2102.07492*, 2021.

Phillip Rust, Jonas Pfeiffer, Ivan Vulić, Sebastian Ruder, and Iryna Gurevych. How good is your tokenizer? on the monolingual performance of multilingual language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 3118–3135, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.243. URL `https://aclanthology.org/2021.acl-long.243`.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *CoRR*, abs/1911.02150, 2019. URL `http://arxiv.org/abs/1911.02150`.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.10297*, 2020. doi: 10.48550/ARXIV.2009.05617. URL `https://arxiv.org/abs/2009.05617`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL `https://aclanthology.org/2021.emnlp-main.685`.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, pp. 1–10, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392730. doi: 10.1145/3520312.3534862. URL `https://doi.org/10.1145/3520312.3534862`.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task, 2018. URL https://arxiv.org/abs/1809.08887.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning, 2017. URL https://arxiv.org/abs/1709.00103.

A Full text2code results

We report the full results of all experiments. Table 7 and 8 show the full results for the data filtering ablations on HumanEval and MBPP, respectively. Table 9 and 10 reports the full results for the architecture ablations on HumanEval and MBPP, respectively.

Language	Model	Pass@1	Pass@10	Pass@100
Java	Baseline	0.1	0.19	0.35
	GitHub stars	0.08	0.16	0.3
	Comments-to-code ratio	0.11	0.2	0.35
	More near deduplication	0.13	0.22	0.38
	Tokenizer fertility	0.11	0.19	0.35
JavaScript	Baseline	0.12	0.19	0.33
	GitHub stars	0.08	0.15	0.3
	Comments-to-code ratio	0.12	0.2	0.35
	More near deduplication	0.14	0.2	0.37
	Tokenizer fertility	0.1	0.19	0.35
Python	Baseline	0.12	0.21	0.36
	GitHub stars	0.1	0.18	0.31
	Comments-to-code ratio	0.14	0.22	0.38
	More near deduplication	0.13	0.22	0.37
	Tokenizer fertility	0.14	0.21	0.36

Table 7: Full results for data filtering ablations on HumanEval

Language	Model	Pass@1	Pass@10	Pass@100
Java	Baseline	0.23	0.37	0.54
	GitHub stars	0.18	0.33	0.49
	Comments-to-code ratio	0.22	0.37	0.52
	More near deduplication	0.23	0.38	0.55
	Tokenizer fertility	0.22	0.38	0.53
JavaScript	Baseline	0.25	0.43	0.64
	GitHub stars	0.19	0.37	0.59
	Comments-to-code ratio	0.25	0.44	0.65
	More near deduplication	0.26	0.45	0.66
	Tokenizer fertility	0.24	0.43	0.65
Python	Baseline	0.27	0.47	0.67
	GitHub stars	0.24	0.41	0.63
	Comments-to-code ratio	0.3	0.48	0.69
	More near deduplication	0.31	0.49	0.71
	Tokenizer fertility	0.28	0.47	0.68

Table 8: Full results for data filtering ablations on HumanEval

Language	Attention	FIM	Pass@1	Pass@10	Pass@100
Java	Multi Query Attention	✓	0.1	0.19	0.35
	Multi Head Attention	✓	0.12	0.21	0.36
	Multi Query Attention	✗	0.11	0.21	0.37
JavaScript	Multi Query Attention	✓	0.12	0.19	0.33
	Multi Head Attention	✓	0.13	0.21	0.37
	Multi Query Attention	✗	0.14	0.21	0.37
Python	Multi Query Attention	✓	0.12	0.21	0.36
	Multi Head Attention	✓	0.13	0.24	0.38
	Multi Query Attention	✗	0.14	0.23	0.39

Table 9: Full results for architecture ablations on HumanEval

Language	Attention	FIM	Pass@1	Pass@10	Pass@100
Java	Multi Query Attention	✓	0.23	0.37	0.54
	Multi Head Attention	✓	0.23	0.38	0.55
	Multi Query Attention	✗	0.23	0.37	0.55
JavaScript	Multi Query Attention	✓	0.25	0.43	0.64
	Multi Head Attention	✓	0.26	0.46	0.67
	Multi Query Attention	✗	0.23	0.44	0.65
Python	Multi Query Attention	✓	0.27	0.47	0.67
	Multi Head Attention	✓	0.31	0.49	0.7
	Multi Query Attention	✗	0.28	0.47	0.68

Table 10: Full results for architecture ablations on HumanEval

Model Family	Variant	BLEU
InCoder	6.7B	16.04
CodeGen-Mono	16B	20.56
SantaCoder	Baseline	17.67
SantaCoder	No-FIM	17.71
SantaCoder	MHA	17.72
SantaCoder	Bf16	17.67
SantaCoder	GitHub Stars	18.04
SantaCoder	Comments-to-code	17.81
SantaCoder	More near deduplication	17.65
SantaCoder	Tokenizer fertility	17.64
SantaCoder	Final	18.13

Table 11: CodeXGLUE (Lu et al., 2021) Python Docstring generation smoothed 4-gram BLEU scores using the same methodology as Fried et al. (2022) (L-R single). Models are evaluated zero-shot, greedily and with a maximum generation length of 128.

B Docstring generation

In addition to code completion benchmarks, we also report results on docstring generation. To this end, we evaluate our models on CodeXGLUE code-to-text Lu et al. (2021), which is a benchmark constructed from CodeSearchNet Husain et al. (2019). We use the bigcode-evaluation-harness library Ben Allal et al. (2022), which is derived from lm-evaluation-harness Gao et al. (2021). Models are prompted with a Python function signature and asked to output a corresponding docstring. Results are shown in Table 11.

Findings We find all BigCode Santa variants with 1.1B parameters to outperform the 6.7B InCoder model (Fried et al., 2022), which we attribute to differences in the training datasets. Among BigCode models, variants trained on more Python perform better: The *stars* variant with 32% of Python in its training corpus outperforms the *tokenizer fertility* variant with only 28.5% of Python (see proportions in Table 3). The `bfloat16` is the same as the *no-fim* variant, except for the latter being trained in `float16`. There's no notable performance difference between the two, likely because at our small scale of 1.1B parameters we did not face any training instabilites.

Qualitative examples Below is an example prompt from CodeXGLUE. Model generations and the correct solution are in Table 12.

```
def dailymotion_download(url, output_dir='.', merge=True, info_only=False,
    **kwargs):
    """
```

Model Family	Variant	Generation
InCoder	6.7B	Download a video from Dailymotion.
CodeGen-Mono	16B	Downloads Dailymotion videos by URL.
SantaCoder	Baseline	Download Dailymotion videos.
SantaCoder	FIM	Download a video from a dailymotion video.
SantaCoder	MHA	Download a video from a Dailymotion video.
SantaCoder	bf16	Download video from dailymotion.com.
SantaCoder	GitHub stars	Download media from dailymotion.com
SantaCoder	Comments-to-code	Download a video from Dailymotion.
SantaCoder	More near deduplication	Download a dailymotion video.
SantaCoder	Tokenizer fertility	Download a video from Dailymotion.
Correct solution		Downloads Dailymotion videos by URL.

Table 12: CodeXGLUE (Lu et al., 2021) Python Docstring generation examples.

C PII

C.1 Regular expressions

Email addresses We used the following regular expression to detect emails.

```
email_pattern = r'''
    (?<= ^ | [\b\s@,?!;:)('".\p{Han}<] )
    (
      [^\b\s@?!;,:)('"<]+
      @
      [^\b\s@!?;,/]*
      [^\b\s@?!;,/:)('">.]
      \.
      \p{L} \w{1,}
    )
    (?= $ | [\b\s@,?!;:)('".\p{Han}>] )
'''
```

We replace detected emails with [random 5 character string]@example.com.

IP addresses We used the following regular expressions to detect IPv4 and IPv6 addresses.

```
ipv4_pattern = r"(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
    (?:\.(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)){3}"
ipv6_pattern = r"(?:[0-9a-fA-F]{1,4}:){7,7}[0-9a-fA-F]{1,4}|(?:[0-9a-fA-F
    ]{1,4}:){1,7}:|(?:[0-9a-fA-F]{1,4}:){1,6}:[0-9a-fA-F]{1,4}|(?:[0-9a-fA-
    F]{1,4}:){1,5}(?::[0-9a-fA-F]{1,4}){1,2}|(?:[0-9a-fA-F]{1,4}:)
    {1,4}(?::[0-9a-fA-F]{1,4}){1,3}|(?:[0-9a-fA-F]{1,4}:){1,3}(?::[0-9a-fA-
    F]{1,4}){1,4}|(?:[0-9a-fA-F]{1,4}:){1,2}(?::[0-9a-fA-F]{1,4}){1,5}|[0-9
    a-fA-F]{1,4}:(?:(?::[0-9a-fA-F]{1,4}){1,6})|:(?:(?::[0-9a-fA-F]{1,4})
    {1,7}|:)|fe80:(?::[0-9a-fA-F]{0,4}){0,4}%[0-9a-zA-Z]{1,}|::(?:ffff
    (?::0{1,4}){0,1}:){0,1}(?:(?:25[0-5]|(?:2[0-4]|1{0,1}[0-9]){0,1}[0-9])
    \.){3,3}(?:25[0-5]|(?:2[0-4]|1{0,1}[0-9]){0,1}[0-9])|(?:[0-9a-fA-F
    ]{1,4}:){1,4}:(?:(?:25[0-5]|(?:2[0-4]|1{0,1}[0-9]){0,1}[0-9])\.)
    {3,3}(25[0-5]|(?:2[0-4]|1{0,1}[0-9]){0,1}[0-9])"
ip_pattern = (
    r"(?:^|[\b\s@?,!;:\'\")(.\p{Han}])("
    + r"|".join([ipv4_pattern, ipv6_pattern])
    + ")(?:$|[\s@,?!;:'\"(.\p{Han}])"
)
```

Data pre-filtering This is the regular expression we used to pre-filter the annotation dataset for data containing emails.

```
email_pattern = r'([^\s@,?!;:\'\"=)(]+@[^,\s!?;,\'\"=]{3,}[\.][^\s\b\'\"@
    ,?!;:)(.]+)'
```

For IP addresses, we used the same regular expression as the one used for PII detection.

C.2 List of private IP addresses and popular DNS servers

- 8.8.8.8
- 8.8.4.4
- 1.1.1.1
- 1.0.0.1

- 76.76.19.19
- 76.223.122.150
- 9.9.9.9
- 149.112.112.112
- 208.67.222.222
- 208.67.220.220
- 8.26.56.26
- 8.20.247.20
- 94.140.14.14
- 94.140.15.15