Decoding Data Quality via Synthetic Corruptions: Embedding-guided Pruning of Code Data

Yu Yang[1,2] **Aaditya K. Singh**[2] **Mostafa Elhoushi**[2] **Anas Mahmoud**[2]
Kushal Tirumala[2] **Fabian Gloeckle**[2] **Baptiste Rozière**[2] **Carole-Jean Wu**[2]
Ari S. Morcos[2] **Newsha Ardalani**[2]
[1]UCLA [2]Meta AI
yuyang@cs.ucla.edu
{melhoushi,adityasingh,nasmahmoud,ktirumala,
fgloeckle,broz,carolejeanwu,new}@meta.com

Abstract

Code datasets, often collected from diverse and uncontrolled sources such as GitHub, potentially suffer from quality issues, thereby affecting the performance and training efficiency of Large Language Models (LLMs) optimized for code generation. Previous studies demonstrated the benefit of using embedding spaces for data pruning, but they mainly focused on duplicate removal or increasing variety, and in other modalities, such as images. Our work focuses on using embeddings to identify and remove "low-quality" code data. First, we explore features of "low-quality" code in embedding space, through the use of synthetic corruptions. Armed with this knowledge, we devise novel pruning metrics that operate in embedding space to identify and remove low-quality entries in the Stack dataset. We demonstrate the benefits of this *synthetic corruption informed pruning (SCIP)* approach on the well-established HumanEval and MBPP benchmarks, outperforming existing embedding-based methods. Importantly, we achieve up to a 3% performance improvement over no pruning, thereby showing the promise of insights from synthetic corruptions for data pruning.

1 Introduction

Machine learning, and in particular Large Language Models (LLMs), are transforming a wide range of industries. Their capabilities extend even to specialized tasks like code generation and medical diagnostics, thus amplifying their societal and economic impact [1]. In this race for higher performance, some training datasets have swelled to petabyte size, sourced from extensive repositories like the Common Crawl. While significant effort has gone into optimizing the computational aspects of training LLMs, such as hardware acceleration and algorithmic improvements [2], the question of data efficiency is still relatively under-explored. Data efficiency is not merely a computational concern but is intrinsically tied to the quality of the training data. The use of large, but ineffective, datasets can result in protracted training times, higher energy consumption, and ultimately, models that are expensive to deploy and maintain [3].

Code datasets, usually compiled from diverse, open-source platforms like GitHub, are often riddled with inconsistencies, errors, or low-quality code snippets. These issues not only undermine the model's final performance but also affect the efficiency and effectiveness of the training process. The presence of such low-quality data essentially "pollutes" the learning environment, leading to suboptimal results. Therefore, improving data quality is not merely an ancillary task but a fundamental requirement for achieving the full potential of code-generating LLMs. A recent study [4] showcased the benefits of so-called "textbook-quality" data in enhancing model efficiency for code-generation

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

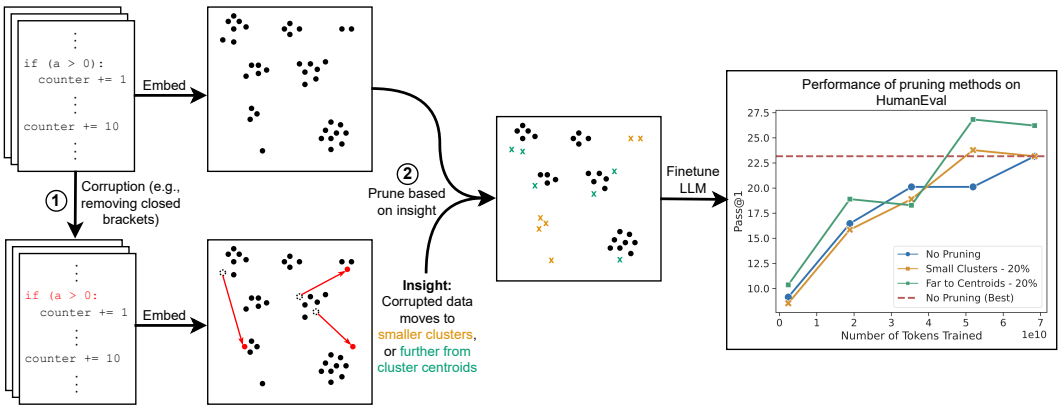


Figure 1: Schematic of SCIP. First, we synthetically corrupt code data, which tends to move code embeddings to smaller clusters or further from cluster centroids. Then, we use this insight to propose a new pruning metric, resulting in improved training efficiency and better end performance.

tasks. However, their strategy relies heavily on generating closed-source data with GPT-3.5 and then filtering it based on GPT-4 [5] predictions, both of which are proprietary models, thus making this approach less accessible for many researchers due to high costs and difficulty of reproducibility. Furthermore, another study [6] highlighted potential issues with training on generated outputs. This emphasizes the need for open-source techniques to identify valuable data in existing, large-scale, natural corpora.

Building upon these identified challenges and gaps in existing research, we focus on easy-to-use, accessible pruning methods for the large open-source Stack dataset [7]. To this end, we take inspiration from recent approaches to data pruning in the domains of image [3] and multimodal models [8], which make use of pre-trained embedding spaces to identify useful or duplicate data, to keep or prune, respectively. In the hitherto unexplored domain of code, we introduce synthetic corruption informed pruning (SCIP): First, we identify what constitutes "low-quality" data in embedding space through controlled corruption of existing data, and find that corrupted code tends to reside in smaller clusters and often be farther from cluster centroids. Then, we introduce a pruning strategy, based on these insights, that ranks data points based on their cluster size and distance to the nearest centroid, aiming to remove a predefined fraction of the data. Using these embedding-based methods for pruning low-quality code, we demonstrate improvements in performance and training efficiency on widely used benchmarks [9, 10].

2 What Does Low-Quality Mean for Code Data?

2.1 Definition of Low-Quality Data

Let $\mathcal{D}$ be the original dataset, $\mathcal{Q} \subseteq \mathcal{D}$ be a subset, and $\mathcal{D}_{\text{test}}$ be the test set. Let $x_{\text{test},i}$ be the i-th test example in $\mathcal{D}_{\text{test}}$. First, we define a general metric M, which could potentially be pass@k [9] or any other quality metric. We then define $M(\theta(\mathcal{D}), \mathcal{D}_{\text{test}})$ as the expectation of a particular metric (for example, pass@k$_i$) over all $x_{\text{test},i}$ in $\mathcal{D}_{\text{test}}$ when training on dataset $\mathcal{D}$ with model parameters θ:

$$M(\theta(\mathcal{D}), \mathcal{D}_{\text{test}}) = \mathbb{E}_{x_{\text{test},i} \in \mathcal{D}_{\text{test}}}[\text{pass@k}_i]$$

The set $\mathcal{Q}$ is defined as "low-quality" if the following inequality holds:

$$M(\theta(\mathcal{D}), \mathcal{D}_{test}) < M(\theta(\mathcal{D} \setminus \mathcal{Q}), \mathcal{D}_{test})$$

In simpler terms, $\mathcal{Q}$ is considered "low-quality" data if removing it from $\mathcal{D}$ improves the score of the general metric M on $\mathcal{D}_{\text{test}}$.

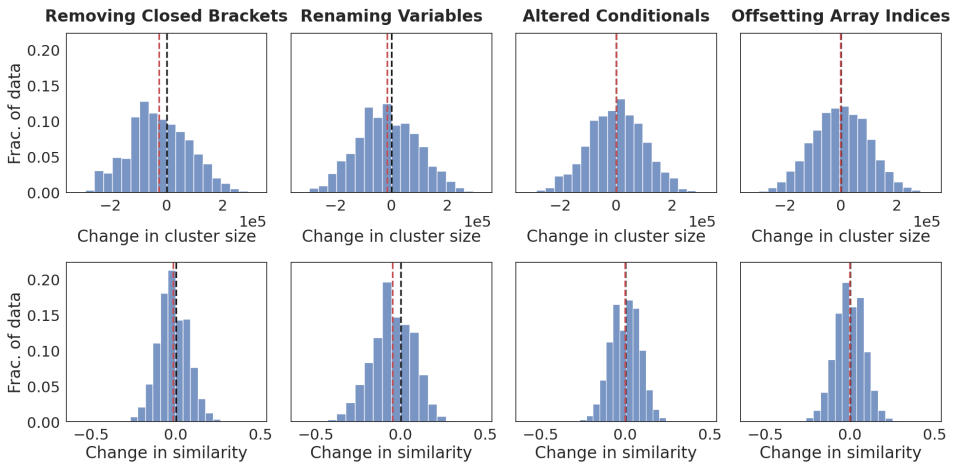


Figure 2: Corrupted data tends to reside in smaller clusters (top row) and farther from centroids (bottom row) when compared to the original, uncorrupted data. The effects are more pronounced for syntax errors (left two columns) as compared to content errors (right two columns). Red dotted line indicates mean, black dotted line indicates 0. More details and analysis can be found in Appendix B.2.

2.2 SCIP: Two-Step Framework for Identifying Low-Quality Data

To systematically identify low-quality data, we propose a two-step framework, illustrated in Figure 1. The first step involves the creation of data with known errors, serving as markers for low-quality data. From this first step, we gather insights on how corruption affects embeddings (obtained with a pretrained model), and use this knowledge to prune data with similar embedding properties.

Synthetic Corruption Generation To identify and prune "low-quality" code data, it's important to understand its possible forms. We consider two main domains: syntax errors and content errors. Synthetic corruption has the benefit of creating *matched* pairs of higher and lower quality data, making it more controlled than alternative approaches which could be confounded by style.

- **Data with Syntax Errors:** Syntax errors are clear indicators of bad code, preventing a file from executing successfully. Such issues can be as common as unmatched parentheses or as nuanced as referencing undeclared variables. To intentionally introduce these errors for the sake of our experiments, we employ two main corruptions: removing closed brackets (specifically, `')'`, `']'`, `'}'`) and renaming variables to syntactically invalid names.

- **Data with Content Errors:** Although such code may run without immediate issues, its output might diverge from the intended result due to underlying logical errors. To simulate this, we either alter conditional operators (through negation) or offset array indices (changing `'i'` to `'i+1'`) to disrupt data access patterns.

More specifics can be found in Appendix B. Through these synthetic corruptions, we ensure a systematic introduction of both syntax and content errors, aiding in a more comprehensive identification of "low-quality" data. By focusing on a representative sample of errors, we effectively set the stage for the next step: identifying and pruning "low-quality" data in large-scale datasets.

Data Pruning Informed by Synthetic Corruptions In the embedding space of a pre-trained code embedding model, StarEncoder [11], we see that synthetic corruption exhibits a distinct change: corruption moves points to smaller clusters or further out from centroids, as compared to the original, uncorrupted code (Fig. 2). These insights shape our pruning strategy. By focusing on data in smaller clusters and distant from centroids, we aim to efficiently identify and remove low-quality data from the original dataset. A formal version of the algorithm, with pseudocode can be found in Appendix C.

Table 1: Pass@1 performance on HumanEval and MBPP for different pruning methods with 20% files pruned.

	No pruning	Random Pruning	SSL Prototype	SemDeDup	D4	**Small Clusters**	**Far from Centroids**	**Combined Small+Far**
HumanEval	25.0%	24.0%	23.8%	20.7%	23.2%	23.2%	<u>26.8%</u>	**28.0%**
MBPP	<u>33.4%</u>	31.9%	32.2%	32.4%	31.2%	**35.0%**	30.8%	33.0%

3 Pruning Low-quality Data for More Efficient Training

3.1 Experiment Setup

Dataset. Our experiments utilize the Stack v1.1 dataset [7], which is sourced from GitHub repositories published from 2015 to 2022, and specifically designed for code generation tasks. Although the dataset includes code from 358 different programming languages, we narrow our focus solely to Python to ensure a more controlled study. This results in a dataset of 12.6M files and 20.4B tokens.

Model and Training Details. Following the methodology of the current state-of-the-art open-source model, Code Llama [12], we fine-tune a 1.5B LLaMA [13] model instead of training from scratch. The model has 48 layers, 24 heads per layer, and inner dimension of 1536. All experiments are run on 32 NVIDIA A100 GPUs with fully-sharded data parallel [14]. We use a learning rate of 3e-4, a batch size of 576, a sequence length of 2048, and train for 56,000 steps ($\sim$67B tokens).

3.2 Evaluation

Our evaluation employs two well-established benchmarks in the code generation field: HumanEval [9] and MBPP [10]. The primary metric for evaluation across these benchmarks is "pass@k," which measures the percentage of test cases that are correctly solved within the top-k generated code snippets. For baselines, we compare to no pruning, random pruning (averaged over 3 seeds), and three other pruning methods using embeddings, based on prior work in other modalities: SSL-prototypes [3], SemDeDup [8], and D4 [15]. Additional details can be found in Appendix D.

3.3 Results

In Table 1, our proposed methods – pruning data that are "Far from Centroid" and within "Small Clusters" – yield clear performance improvements on HumanEval and MBPP, respectively. However, better performance on one benchmark often comes at the expense of the other, perhaps due to the different natures of these tasks. Motivated by the strong performance of our two suggested methods, we experimented with a combined method: first pruning files from small clusters, then files far from centroids, with the ratio between these defined by a parameter α. We found that $\alpha = 0.8$ performed best (see Appendix C). Impressively, this combined method achieves the best performance of all methods tried on HumanEval, a full 3% above no pruning and better than all prior work on embedding-based pruning, while also remaining competitive with no pruning on MBPP.

We also observe in Fig. 1 that "Far from Centroid" and "Small Clusters" both achieve an efficiency speedup (both methods achieve the baseline pass@1 rate in fewer training steps). Further insights into the qualitative attributes of pruned data are presented in Fig. 4."

4 Conclusions

We introduce SCIP, a systematic method to identify and remove "low-quality" code data from large datasets. Building on the insights of the value of high-quality data presented in earlier studies [4], our work goes further by offering accessible, open-source, and cost-effective pruning techniques through the use of embedding spaces. We go beyond prior work in embedding-based pruning [3, 8, 15] by motivating heuristics through identification of "low-quality" data via synthetic corruptions: we systematically create code discrepancies, both in syntax and content, to understand their influence on the embedding space. Our findings reveal that syntax errors lead to significant shifts away from cluster centroids and into smaller clusters. Leveraging these observations, we designed pruning methods that consider both distances to centroids and cluster sizes to effectively identify and remove low-quality

data. Applying these pruning methods leads to better performance on code generation benchmarks, showing the promise of insights from synthetic corruptions for improving pruning techniques.

More broadly, our results underscore the significance of rigorous data curation. Beyond just code, more rigorously examining "low-quality" data could lead to more informed pruning techniques. Similar to how code can have both syntax and content discrepancies, natural language data too can have structural (e.g., grammatical) and semantic (e.g., factually incorrect) anomalies. In future work, the strategies and methodologies established here of using synthetically corrupted data as a pruning signal could be extended and adapted to general natural language datasets, ensuring models trained on them produce more accurate, reliable, and coherent outputs.

Acknowledgments

We would like to sincerely thank Jack Lanchantin for the insightful discussions, and Shubham Toshniwal, Koustuv Sinha, and Alberto Bietti for generously sharing their valuable insights drawn from their previous research.

References

[1] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. Gpts are gpts: An early look at the labor market impact potential of large language models, 2023.

[2] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher R'e. Flashattention: Fast and memory-efficient exact attention with io-awareness. *ArXiv*, abs/2205.14135, 2022.

[3] Ben Sorscher, Robert Geirhos, Shashank Shekhar, Surya Ganguli, and Ari Morcos. Beyond neural scaling laws: beating power law scaling via data pruning. *Advances in Neural Information Processing Systems*, 35:19523–19536, 2022.

[4] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

[5] OpenAI. Gpt-4 technical report, 2023.

[6] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. The curse of recursion: Training on generated data makes models forget, 2023.

[7] 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.

[8] Amro Kamal Mohamed Abbas, Kushal Tirumala, Daniel Simig, Surya Ganguli, and Ari S. Morcos. Semdedup: Data-efficient learning at web-scale through semantic deduplication. In *ICLR 2023 Workshop on Multimodal Representation Learning: Perks and Pitfalls*, 2023.

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

[10] 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.

[11] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha

Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! 2023.

[12] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[13] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[14] FairScale authors. Fairscale: A general purpose modular pytorch library for high performance and large scale training. `https://github.com/facebookresearch/fairscale`, 2021.

[15] Kushal Tirumala, Daniel Simig, Armen Aghajanyan, and Ari S Morcos. D4: Improving llm pretraining via document de-duplication and diversification. *arXiv preprint arXiv:2308.12284*, 2023.

[16] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. Unsupervised learning of visual features by contrasting cluster assignments. *Advances in neural information processing systems*, 33:9912–9924, 2020.

[17] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[18] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

A Related Work

Embedding-based Data Pruning for LLMs With the advent of LLMs, data quality characterization has become even more critical. SemDeDup [8] exemplifies how quality embeddings can expedite LLM learning with reduced data. By removing semantic duplicates, SemDeDup enhances both training efficiency and downstream performance for LLMs. Extending this further, D4 (Document De-Duplication and Diversification) [15] combines SemDeDup with SSL Prototypes [3] and outperforms using both SemDeDup and SSL Prototypes independently. These findings offer insights into continual model improvement beyond data scale. While these methods use embedding spaces to prune, the heuristics employed on top (pruning more "prototypical" examples or close by points) are hand-designed. In this work, we extend prior work by showing the promise of synthetically corrupted data as a signal for embedding space heuristics for pruning.

Code Generation Remarkable advancements have been made in recent years in the development of code-generating AI assistants [12]. These models, known as code LLMs, are crafted by training large transformer neural networks on extensive corpora of source code, empowering them to perform sophisticated code completions. They can generate code not only from surrounding code fragments but also from natural language instructions. To evaluate code LLMs, researchers use unit tests to check if the generated code behaves as expected. Two popular benchmarks for evaluating Python code generation are HumanEval [9] and MBPP [10], which include descriptions of functions in plain language along with corresponding sets of unit tests.

B Synthetically Corrupted Data

B.1 Creation

To effectively identify and prune "low-quality" code data, it's important to understand its possible forms. We categorize them into two main domains: syntax errors and content errors.

Data with Syntax Errors Syntax errors are clear indicators of problematic code, preventing a code snippet from executing successfully. Such issues can be as common as unmatched parentheses or as nuanced as referencing undeclared variables. To intentionally introduce these errors for the sake of our experiments, we employ two main corruptions:

1. **Removing Closed Brackets:** By omitting closing brackets, including parentheses `)`, square brackets `]`, and curly braces `}`, from code, we introduce errors that would halt execution. For example, the code segment `for i in range(10):` might be changed to `for i in range(10:`.
2. **Renaming Variables:** Altering variable names at random intervals ensures that they no longer match their original declarations. For instance, a variable declared as `counter = 0` might be used later in the code as `counter += 1`, which we would change to `ctr += 1`, creating a reference to an undeclared variable.

Data with Content Errors Although such code may run without immediate issues, its output might diverge from the intended result due to underlying logical errors. To simulate this, we adopt two principal corruptions:

1. **Altered Conditionals:** Switching common relational operators alters the flow of code without introducing blatant syntax errors. For example, conditions like `if a == b:` can be transformed to `if a != b:`.
2. **Offsetting Array Indices:** Adjusting indices by adding or subtracting a unit disrupts data access patterns. A line such as `value = array[i]` might become `value = array[i+1]`, potentially leading to unintended behavior or out-of-bound errors.

Importantly, we note that synthetic corruption yields matched pairs of good (original) and bad (corrupted) code. We corrupt and embed each document in the Stack, with one of the above corruptions at a time. Note that, future work could consider drawing insights from a smaller percentage of corrupted data, but we opted for the full datasets for simplicity. On the whole, this

Table 2: Fraction of files pairs that changed non-negligibly after each corruption.

	Removing Closed Brackets	Renaming Variables	Altered Conditionals	Offsetting Array Indices
Changed cluster	0.77	0.89	0.02	0.23
Changed distance from cluster centroid	0.49	0.57	0.02	0.20

clustering step is relatively inexpensive compared to model training, taking about 2% of the time. In the next section, we look at the effects of these corruptions in embedding space.

B.2 Effects of Synthetic Corruptions

Building off prior work [3, 8, 15] on embedding-based pruning, we started by clustering the embeddings of our original, unperturbed data. On a single GPU, this clustering step takes on the order of 10 minutes. We used $k = 100$ clusters, identified using the K-means algorithm with cosine similarity as the metric. When corrupting a file, we then consider the following two things:

1. Distance to its cluster centroid before and after corruption
2. Size of new cluster if the corrupted file lies in a different cluster. If so, the size difference between the two clusters (in terms of # of files)

Our main results are presented in Fig. 2. For visualization pruposes, we excluded all points with negligible change (points that stay in the same cluster or points whose distance to centroid changes by less than 0.01). The fraction of file-pairs remaining after this exclusion is presented in Table 2. From Table 2, we can see that content corruptions (right two columns) lead to a way smaller fraction of file pairs that show significant changes in embedding space, which partially explains the weaker signal we see in Fig. 2 for these corruptions. On the other hand, syntactical corruptions have effects on most files, especially leading to many cluster changes.

C Algorithm

Given a dataset $\mathcal{D}$, we cluster the data into $K = 100$ clusters with centroids $C = \{c_1, c_2, \ldots, c_{100}\}$. Let $e(x)$ denote the embedding of a code snippet x. We pre-normalize all the embeddings to have magnitude 1. The distance metric we use is cosine-similarity, $d_C(x, y) = 1 - x^\top y$, where x, y are both unit norm.

Distance to Centroid: For each code snippet $x \in \mathcal{D}$, we determine the nearest centroid and compute its distance as:

$$c_{\text{min}}(x) = \arg\min_{c_i \in C} d_C(e(x), c_i)$$

$$d(x) = d_C(e(x), c_{\text{min}}(x))$$

Cluster Size: For each centroid c_i, we determine the size of its associated cluster, $s(c_i)$ which equals the number of points assigned to it. For any snippet x, the associated cluster size is given by $s(c_{\text{min}}(x))$.

Pruning Strategy: To prune the dataset, we first rank the code snippets based on their associated cluster size $s(c_{\text{min}}(x))$ and their distance $d(x)$. We then prune the top $p = 20\%$ of the dataset. To interpolate between pruning based on cluster size and distance, we specify a hyperparameter α. We then prune $\alpha p\%$ of data based on cluster size (removing smaller clusters) and $(1 - \alpha)p\%$ of the remaining data based on distance (removing points further from centroids). We experiment with multiple values of α, with full results in Table 3. This pruning mechanism ensures that the data points that are most similar to our synthetically corrupted data, in terms of spatial properties, are removed, thus refining $\mathcal{D}$ to a cleaned version $\mathcal{D}_{\text{clean}}$.

The pseudocode can be found at Algorithm 1.

Table 3: Effects of different percentages of pruning from small clusters and from points far from cluster centroids.

α	0.0	0.2	0.5	0.7	0.8	1.0
HumanEval	26.8%	22.6%	23.8%	23.8%	28.0%	23.2%
MBPP	30.8%	31.6%	33.2%	31.8%	33.0%	35.0%

Algorithm 1 Embedding-guided Weighted Pruning of Code Data

Require: Dataset $\mathcal{D}$, fraction p to prune, embedding function $e(\cdot)$, weight α between [0, 1]
Ensure: Pruned dataset $\mathcal{D}_{\text{pruned}}$

Cluster $\mathcal{D}$ into K clusters with centroids $C = \{c_1, c_2, \ldots, c_K\}$
Calculate cluster sizes $s(c_i)$ for $c_i \in C$
for each x in $\mathcal{D}$ **do**
    $c_{\text{min}}(x) \leftarrow \arg\min_{c_i \in C} d_C(e(x), c_i)$ ▷ Find closest centroid
    $d(x) \leftarrow d_C(e(x), c_{\text{min}}(x))$ ▷ Compute distance to closest centroid
end for
Rank $\mathcal{D}$ based on $s(c_{\text{min}}(x))$ in ascending order
$\mathcal{D}_{\text{prune\_by\_size}} \leftarrow$ top $\alpha \times p\%$ of $\mathcal{D}$ based on cluster size ranking
Rank remaining $\mathcal{D} \setminus \mathcal{D}_{\text{prune\_by\_size}}$ based on $d(x)$ in descending order
$\mathcal{D}_{\text{prune\_by\_distance}} \leftarrow$ top $(1-\alpha) \times p\%$ of $\mathcal{D} \setminus \mathcal{D}_{\text{prune\_by\_size}}$ based on distance ranking
$\mathcal{D}_{\text{pruned}} \leftarrow \mathcal{D} \setminus (\mathcal{D}_{\text{prune\_by\_size}} \cup \mathcal{D}_{\text{prune\_by\_distance}})$ ▷ Remove the pruned data
return $\mathcal{D}_{\text{pruned}}$

D Evaluation

D.1 Metric

The pass@k metric evaluates the functional correctness of the generated code. Specifically, a code sample is considered "correct" if it successfully passes a set of unit tests. For each problem, k code samples are generated, and the problem is considered "solved" if any of those samples pass the unit tests. The metric pass@k ultimately reports the total fraction of problems that are solved.

We define $\text{pass@k}_i(n, c, k)$ for each $x_{\text{test},i}$ as:

$$\text{pass@k}_i(n, c, k) = \begin{cases} 1.0, & \text{if } n - c < k, \\ 1.0 - \prod_{j=n-c+1}^{n} \left(1 - \frac{k}{j}\right), & \text{otherwise.} \end{cases}$$

Here, n is the total number of samples in $\mathcal{D}_{\text{test}}$, c is the number of correct samples for the i-th test case, and k is the value for which pass@k_i is calculated.

D.2 Datasets

HumanEval HumanEval [9] consists of 164 hand-crafted programming tasks aimed at assessing functional correctness. The hand-crafted nature minimizes overlap with GitHub-based training data. Each task includes essential elements like function signature, docstring, and an average of 7.7 unit tests, targeting skills like language comprehension, reasoning, and algorithms.

MBPP: Mostly Basic Programming Problems MBPP [10] features 974 crowd-sourced Python programs, varying from simple calculations to tasks requiring specialized knowledge. Each problem has a problem statement, a Python function, and three test cases, along with a ground-truth solution. For our experiments, we use an edited subset of 426 questions that adhere to standard Python conventions and are unambiguous.

We follow the standard procedure to evaluate models in zero-shot on HumanEval and 3-shot on MBPP. Example prompt and answers are provided in Figure Fig. 3.

```python
def change_base(x: int, base: int):
    """Change numerical base of input number x to base.
    return string representation after the conversion.
    base numbers are less than 10.
    >>> change_base(8, 3)
    '22'
    >>> change_base(8, 2)
    '1000'
    >>> change_base(7, 2)
    '111'
    """
----------END PROMPT----------
    ret = ""
    while x > 0:
        ret = str(x % base) + ret
        x //= base
    return ret
```

(a) HumanEval example

```
You are an expert Python programmer, and here is your task: Write a function to find the similar elements from the given two tuple lists. Your code should pass these tests:

assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)
assert similar_elements((1, 2, 3, 4),(5, 4, 3, 7)) == (3, 4)
assert similar_elements((11, 12, 14, 13),(17, 15, 14, 13)) == (13, 14)
[BEGIN]
def similar_elements(test_tup1, test_tup2):
  res = tuple(set(test_tup1) & set(test_tup2))
  return (res)
[DONE]
You are an expert Python programmer, and here is your task: Write a python function to identify non-prime numbers. Your code should pass these tests:

assert is_not_prime(2) == False
assert is_not_prime(10) == True
assert is_not_prime(35) == True
[BEGIN]
import math
def is_not_prime(n):
    result = False
    for i in range(2,int(math.sqrt(n)) + 1):
        if n % i == 0:
            result = True
    return result
[DONE]
You are an expert Python programmer, and here is your task: Write a function to find the largest integers from a given list of numbers using heap queue algorithm. Your code should pass these tests:

assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],3)==[85, 75, 65]
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],2)==[85, 75]
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],5)==[85, 75, 65, 58, 35]
[BEGIN]
import heapq as hq
def heap_queue_largest(nums,n):
  largest_nums = hq.nlargest(n, nums)
  return largest_nums
[DONE]
You are an expert Python programmer, and here is your task: Write a function that matches a word at the beginning of a string. Your code should pass these tests:

assert text_match_string(" python")==('Not matched!')
assert text_match_string("python")==('Found a match!')
assert text_match_string("  lang")==('Not matched!')
[BEGIN]
----------END PROMPT----------
import re
def text_match_string(text):
        patterns = '^\w+'
        if re.search(patterns,  text):
                return 'Found a match!'
        else:
                return 'Not matched!'
```

(a) MBPP example

Figure 3: Example prompt and solutions for (a) HumanEval and (b) MBPP. “END PROMPT” is added in artificially for reader’s clarity – that line does not appear in the actual prompt or solution.

D.3 Baselines

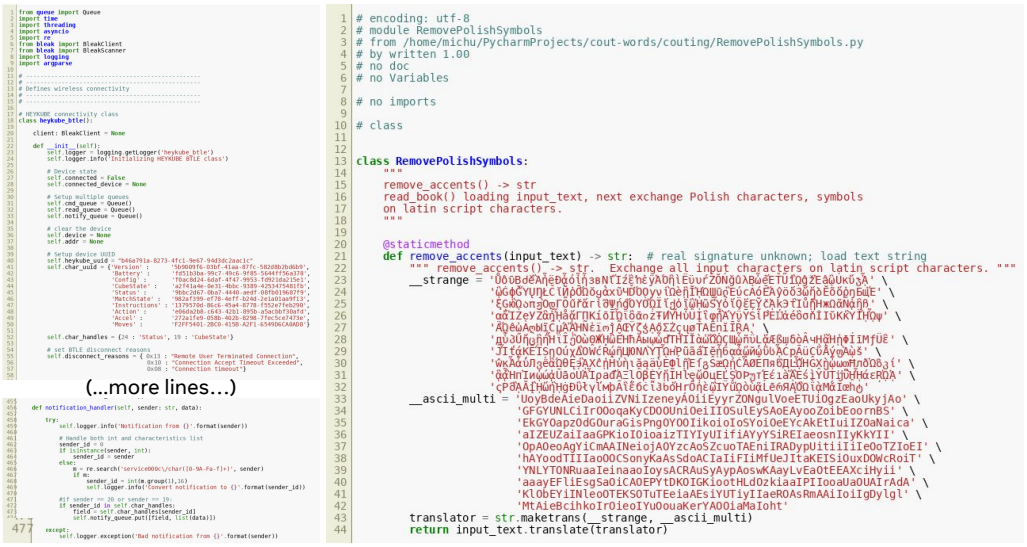
SSL-prototypes SSL prototypes [3] presents a data pruning based on the underlying theory for perceptrons. This method makes three predictions relevant to broader neural network applications and benchmark dataset training. Firstly, when the initial dataset size is large, emphasizing the most challenging examples will be more helpful compared to random data pruning. Secondly, when data pruning retains a fixed fraction of the toughest examples, the outcome should exhibit power law scaling consistent with that of random pruning as the original dataset size grows. Lastly, optimizing test error over both the initial dataset size and the retained fraction can potentially produce a Pareto optimal curve that surpasses the power law scaling concerning the pruned dataset size. This is achieved by more aggressive pruning for larger initial dataset sizes. To devise a self-supervised pruning metric the SSL prototypes method employs k-means clustering within the embedding space of a pre-trained self-supervised model, such as SWaV [16]. A data point’s difficulty is determined by its cosine distance to the closest cluster centroid or prototype, implying that “easy” examples align closely with the prototype, while “hard” ones deviate significantly. This self-supervised prototype metric either matches or outperforms some of the best supervised metrics up to retaining 70-80% of the data for image datasets.

SemDeDup The process of detecting perceptual duplicates might be straightforward in input space, but identifying semantic duplicates presents a unique challenge. This is primarily because semantic duplicates can be considerably different in pixel or token spaces. The SemDeDup method [8]tackles

this issue by employing the embedding space of a large pre-trained foundational model, which offers a semantically-rich distance metric. To detect and eliminate semantically similar entries, the algorithm first embeds each data point using foundational models, such as CLIP [17] for images and OPT [18] for text. Subsequent clustering of these embeddings is done using k-means. Within each cluster, pairwise cosine similarities are computed, setting a threshold above which entries are flagged as semantic duplicates. Only the entry with the least cosine similarity to the cluster's centroid is retained, while the others are pruned.

D4 D4 [15]: While working with large datasets, encountering clusters of redundant or templated text is common. Such clusters, which may not be filtered out by methods like MinHash, create dense regions in the embedding space. This density can influence clustering algorithms like k-means to allocate clusters to duplicated text, which can compromise the efficiency of methods like SSL Prototypes, where many clusters may be dominated by duplicates rather than topic-based coherence. Recognizing this, the D4 strategy was introduced. It starts by applying the SemDeDup method on the entire dataset produce a de-duplicated dataset. This pruned dataset is then clustered using K-Means. Subsequently, SSL Prototypes is applied. The resulting strategy ensures a global and local diversification of data. This method is abbreviated as D4, denoting "Document De-Duplication and Diversification".

E Inspecting the Pruned Data



(a) Top 2 examples pruned for their low similarity from the centroids. They both contain a lot of meaningless symbols.

(b) Top 2 examples pruned from the smallest clusters. They either repeat multiple similar functions, or contain extremely long lines with not instructive file paths.

Figure 4: Examples of pruned data points.