K-Specialist Approach to Code Generation

Karthik Suresh [1] **Hafeez Ali Anees Ali** [1] **Calvin Kranig** [1]

Abstract

We introduce a new model training and inference pipeline involving the use of K-means clustering and a novel dataset to perform SFT (Supervised Fine Tuning) for code generation. We adopt a number of techniques to generate data samples synthetically with an emphasis on data quality and complexity. Benchmarking our dataset against other SFT datasets for code generation, we find that ours has the highest complexity scores, as evidenced by higher Cyclomatic and Halstead Complexity measures while underperforming on the Diversity benchmark. During the training phase, we train phi-2 (2.7B) (Hughes) and CodeLlama-Python-7B (Rozière et al., 2023) using a novel procedure. We leveraged our collected data to train a K-means clustering model using embeddings from a Sentence Embedding model. By training the K-means model on embeddings from our collected SFT dataset, we are able to split the SFT data into K splits. We use these K data splits to train K LoRA adapters. Using our method, our best model, phi-2 ($K = 10$), achieves **53.54%** *pass@1* on the HumanEval benchmark, which is comparable to the *pass@1* performance of the CodeLlama-Python-34B variant. Moreover, we see an increase in performance as we increase K while keeping the number of data points the same.

1. Introduction and Related Work

Code generation has gained importance with a plethora of closed-source models such as GPT-4 (OpenAI, 2023), Anthropic Claude (ant) , etc., and more recently, open-source models such as CodeLlama (Rozière et al., 2023), phi-2 (Hughes) pushing the boundaries of generating working code corresponding to a question or description posed by a user. Before LLM-based approaches became popular, the area known as program synthesis (Gulwani et al., 2017) leveraged symbolic approaches (Wang et al., 2017; Feng et al., 2018) for code generation.

[1]University of Wisconsin - Madison, Wisconsin, USA.

The standard paradigm in evolving models for code generation is to pre-train on a large corpus of code and natural language data, which is usually leveraged from GitHub or StackOverflow data. Different methods use data preprocessing techniques to ensure the quality and usability of the data. For example, the authors of Nijkamp et al. (2023) first obtain the raw data on the scale of hundreds of gigabytes. Thereafter, they perform filtering, deduplication, tokenization, shuffling, and concatenation.

Once the model is pretrained, instruction fine-tuned models can be evolved by performing Supervised Fine Tuning (SFT) on some instructions relevant to the downstream task. Inclusion of such a step can significantly improve performance (Ouyang et al., 2022; Wei et al., 2022; Touvron et al., 2023; Zhou et al., 2023)

Dataset composition for this Supervised Fine-tuning phase may prove essential to the downstream performance of the model on the task at hand. Plenty of work has been done in characterizing the dataset composition of data in the SFT phase across various axes, such as Data Quantity (Zhou et al., 2023; Chen et al., 2023; Song et al., 2023), Data Quality (Cao et al., 2023; Li et al., 2023; Wang et al., 2023), Data Diversity (Wan et al., 2023; Lu et al., 2023), Data Complexity (Xu et al., 2023; Luo et al., 2023; Mukherjee et al., 2023), etc. The majority of these works delve into aspects that are important in a conversational model, such as helpfulness, creativity, multi-turn dialogue, etc., which may not be too relevant to the code generation task.

Furthermore, much research hasn't been done on or documentation provided for the supervised fine-tuning datasets used for code understanding and generation tasks, especially for Python. Many of these datasets are proprietary and not released to the public, which leads us to speculate on what they must have included in their training data and if there may be any data leakage. We synthetically curate our own dataset, emphasizing dataset quality and complexity of question-code pairs.

Inspired by the popularity and success of MoE (Mixture-of-Experts) models (Fedus et al., 2022; Mixtral, 2023), we have developed an MoE-like model pipeline. Our approach is more similar to work like (Wang et al., 2022c) that uses a Mixture-of-Adapters referring to a mixture of PEFT methods such as LoRA (Hu et al., 2021) and Houlsby Adapters

(Houlsby et al., 2019). We show that we are able to improve performance on the HumanEval dataset (Chen et al., 2021) as we scale up the number of "experts"[1].

To summarize, we make the following contributions in our research work:

- Build an SFT dataset focused on code generation in Python, keeping in mind dataset quality and complexity
- To benchmark existing SFT datasets for code generation along with our own, in terms of diversity and complexity of instructions
- To develop training and inference pipeline taking inspiration from Mixture-of-Experts and Mixture-of-Adapters implementations that show improved performance with an increased number of "experts"

2. Dataset Curation

Dataset curation for SFT is an important step in achieving significant performance gains on downstream tasks (Ouyang et al., 2022; Wei et al., 2022; Touvron et al., 2023; Zhou et al., 2023). The datasets existing today for Code generation are not well-documented; more recently, many of these datasets have been closed to the public and have not been released. We aim here to benchmark some of the existing datasets that are available to the public and curate our own dataset with a focus on data quality and inclusion of challenging questions.

2.1. Seed Data Collection

While curating our dataset, we start by collecting some seed data points manually and through available public datasets.

We manually collect a set of 58 high-quality question-code pairs from GeeksForGeeks [2] and Leetcode [3] that fall under the tags Easy, Medium, and Hard, with increased emphasis on the Medium and Hard questions. We collect these data points from a wide variety of topics, such as Arrays, Sorting, Searching, Dynamic Programming, etc., to ensure diversity in our seed dataset. We term this Seed set as $S1$

We also collect seed data from the XLCoST (Zhu et al., 2022) dataset, which has its data points scraped from GeeksForGeeks. We chose this dataset as it provides far more challenging questions when compared to alternatives. However, the dataset only has title-code pairs for each data point. The title here corresponds to a title given to a programming question on GeeksForGeeks. For example, one of the problems has the title "Leader in an array". This is an underspecified question that does not detail what constitutes the Leader of an array. To solve this, we manually collect a set of 20 examples where we provide the title, question, and the corresponding code. For every other data point, we feed in the problem's title along with two few-shot examples to GPT-3.5 and elicit generations for the question description and corresponding code. Although the XLCoST dataset constitutes the solution to the question, we synthetically generate the solution because the generated question description may constitute a different idea of what a title (like Leader in an array) might be when compared to the actual question description on GeekForGeeks. So, to avoid any such disparities in question-code pairs, we prompt GPT-3.5 to generate both the question description and the code solution jointly.

Some of the titles for the data points may be highly underspecified. These were usually seen to have a low character count. As a processing step, we removed the lower quartile of data points (as measured by string length) from the XLCoST dataset. This constitutes our second seed data source, termed $S2$.

2.2. Synthetic Data Generation

Using $S1$ and $S2$ we generate more data points using two methods inspired by Wang et al. (2022b); Xu et al. (2023); Eldan & Li (2023):

2.2.1. Method 1

Here, we pass in few-shot examples from $S1$ to GPT-3.5 and get a response generation. The samples obtained herein are added back to the seed dataset, and the data generation loop is run again to generate more examples with this expanded seed set. All the data points obtained during these two data generation cycles added with the seed dataset give rise to the set $S1^*$.

2.2.2. Method 2

Inspired primarily by the method used in Eldan & Li (2023) and the strategies employed in Xu et al. (2023); Wu et al. (2023), we hope first to extract a few keywords that characterize programming problems. We manually collect a small set of high-quality question-code-keyword pairs. We then pass in the questions from $S2$ to GPT-3.5 and prompt it to generate keywords relevant to the question using few shots from the above-mentioned small set. For the question, "*Maximize first array element by performing given operations at most K times*", the generated keyword list was ['Array', 'Operation', 'Maximize', 'Integer']. We collect a list of keywords in this manner for all questions in the dataset. After

[1] We use "experts" to loosely define the different specialist models that we train, not to be confused with the experts in Mixture-of-Experts

[2] https://www.geeksforgeeks.org/

[3] https://leetcode.com/

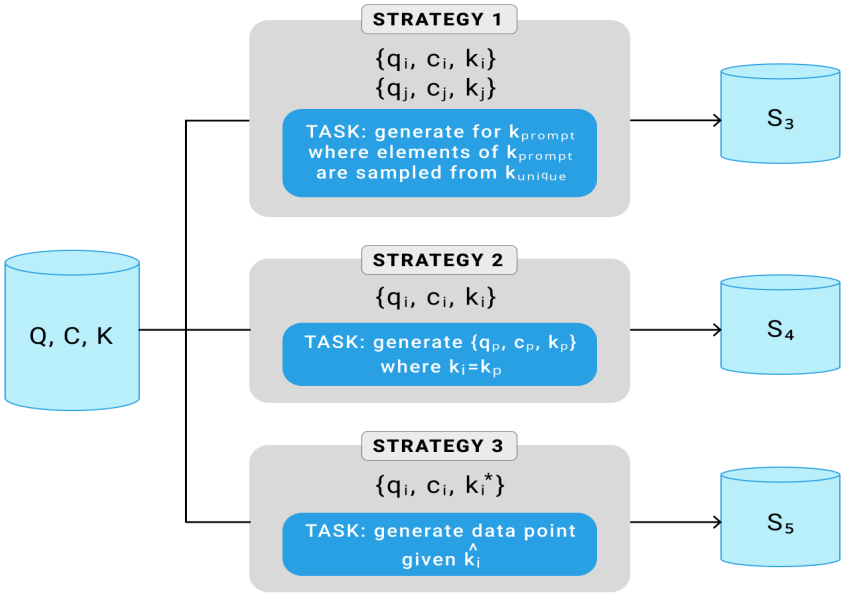

Figure 1. Three Data Generation Strategies as detailed in Section 2.2.2. Strategy 1 resulted in many cases of model refusal to use the given keyword list to produce a meaningful question-code pair. Strategy 2 generated data points that were similar to the in-context examples with some slight modifications. Strategy 3, which was a combination of elements from Strategy 1 and Strategy 2, resulted in complex and unique data points being generated.

removing duplicates, the final list of keywords had a length of roughly ∼2000, and is termed k_{unique}.

Given a set of questions $Q = \{q_1, q_2, q_3, \ldots, q_n\}$, Corresponding set of codes $C = \{c_1, c_2, c_3, \ldots, c_n\}$, and set consisting of the list of corresponding keywords $K = \{k_1, k_2, k_3, \ldots, k_n\}$. Here, each element k_i is a list of keywords associated with the question q_i and code c_i, and n refers to the size of dataset $S2$. We implement three strategies that are also diagrammatically outlined in Figure 1:

1. We sample 2 in-context examples $\{q_i, c_i, k_i\}$, $\{q_j, c_j, k_j\}$ $(i \neq j)$ from $S2$ and add it to the prompt. We then sample from k_{unique} to generate an n-length list of keywords k_{prompt} where $n \sim \text{Uniform}(5, 15)$. We then prompt the model to generate examples based on k_{prompt}. The prompt also includes instructions that convey that not all of these keywords need to be used to generate the new data points. This strategy resulted in mixed results. Half of the generations suggested that the model could not generate an example with the list of keywords given as they were a seemingly random combination of keywords. The other half resulted in acceptable generations that included a wide variety of question-code pairs. Overall, this seemed hit or miss, depending on if we had sampled the right keywords to make k_{prompt}. We discard the erroneous generations and add the rest to a set termed $S3$.

2. We sample an example $\{q_i, c_i, k_i\}$ from our dataset and prompt it to produce another data point $\{q_p, c_p, k_p\}$ that adheres to the condition $k_i = k_p$. In other words, we use an example in the dataset to generate another data point that adheres to the same keyword list. We saw that this type of prompting produced similar questions with minor changes compared to the in-context question. We call the resultant data $S4$

3. The drawback of the first strategy was the model's refusal to generate certain combinations that appear in the keyword list as they may be seemingly random and unrelated. The second strategy resulted in similar data points that differed in small ways. For our final strategy, we combined these two approaches. Starting with strategy 2, we sample an in-context example $\{q_i, c_i, k_i\}$. We create a perturbation of k_i called k_i^*, an extension of k_i with m^* elements sampled from k_{unique} where $m^* \sim \text{Uniform}(0, 5)$. We then create another perturbation called $\hat{k_i}$ by perturbing k_i with $\hat{m}$ elements from k_{unique}, where $\hat{m} \sim \text{Uniform}(0, 5)$. We include $\{q_i, c_i, k_i^*\}$ as our in-context example and prompt the model to generate a question-code pair for $\hat{k_i}$. This strategy successfully generated unique examples that borrowed elements from the in-context seed

example keyword list k_i and added some keyword elements from $\hat{k_i}$ as well. The dataset resulting from this is termed $S5$.

2.3. Deduplication

For the final step, we merged $S1^*, S2, S3, S4$, and $S5$ to construct a combined dataset. Existing research (Kandpal et al., 2022; Lee et al., 2021; Silcock et al., 2022), suggests that deduplication is an essential step in the data preprocessing pipeline. We perform semantic-based deduplication, where we keep only one copy of data points that have a cosine embedding similarity (as calculated by the *e5-base-v2* model (Wang et al., 2022a)) greater than 0.95. If we have n examples with a cosine similarity greater than 0.95 with a single data point, we discard these n examples and keep only the single data point. We repeat this process for all data points in our dataset. This brought the dataset size from ∼35K to ∼20K. We choose a high threshold as embedding models are known to overestimate similarity.

3. Dataset Diversity and Complexity

We take Code generation datasets with python questions and calculate certain metrics. The datasets we take into consideration are Code Alpaca (Chaudhary, 2023), Evol Code Alpaca (Theblackcat102), Evol Instruct Code (Nickrosh), and two datasets from Wei et al. (2023) namely, Magicoder Instruct and Magicoder Evol Instruct. The Evol datasets are obtained from following the Evol-Instruct directions from Xu et al. (2023) and Luo et al. (2023). Table 1 and 2 show the metrics pertaining to diversity and complexity, respectively.

3.1. Diversity metrics

Dataset	Question MTLD	Question Spread	Code Spread
Evol Code Alpaca	**59.07**	**1.28**	1.24
Evol Instruct Code	42.45	1.27	1.25
Magicoder Instruct	36.31	1.23	1.22
Magicoder Evol Instruct	**59.07**	**1.28**	1.24
Code Alpaca	30.92	1.22	**1.29**
Ours	40.36	1.26	1.23

Table 1. Diversity Results as measured by MTLD (Measure of Textual Lexical Diversity) (McCarthy & Jarvis, 2010), Question Spread and Code spread

MTLD (McCarthy & Jarvis, 2010): Measure of Textual Lexical Diversity, put simply, is how many different words are used. We got the inspiration to use it from (Ding et al., 2023), who use it to measure the diversity of their instruction set for conversational data.

Intra-dataset Embedding Cosine similarity:

While MTLD gives us lexical diversity based on the words in natural language, this gives us an idea of semantic diversity. We sample instead of taking all the points in the dataset as this computation is very expensive and time-consuming. We take $N = 1000$ in our calculations. We first take the question and code and convert it into sentence embeddings using an *e5-base-v2* model. Using the formulation below, we calculate the average intra-dataset cosine similarity metric, which is termed as "Question Spread" and "Code Spread" for question and code respectively in Table 1. Intra-dataset cosine similarity for a dataset of size N is given by the reciprocal of the average cosine similarities of embeddings amongst each other:

$$\sum_{j=1}^{N}\sum_{i=1}^{N}\frac{N\cdot(N-1)}{\text{cos\_sim}(\mathbf{e_i},\mathbf{e_j})}$$

where $i \neq j$; $\text{cos\_sim}(\mathbf{e_i},\mathbf{e_j})$ is the cosine similarity between $\mathbf{e_i}$ and $\mathbf{e_j}$; $\{\mathbf{e_1},\mathbf{e_2},\ldots,\mathbf{e_n}\}$ are the embeddings for the different questions/codes from the dataset as generated by the *e5-base-v2* model

3.2. Complexity metrics

Dataset	Halstead Complexity			Cyclomatic	Data Points
	Vocabulary	Length	Difficulty	Avg. Complexity	
Evol Code Alpaca	10.14	15.96	1.99	2.85	111,272
Evol Instruct Code	9.60	14.42	1.96	2.69	78,264
Magicoder Instruct	5.70	7.93	0.99	2.41	75,197
Magicoder Evol Instruct	10.14	15.96	1.99	2.85	111,183
Code Alpaca	3.72	4.91	0.77	1.41	20,022
Ours	**14.10**	**22.07**	**2.99**	**3.87**	20,252

Table 2. Halstead Complexity Measures and Cyclomatic Complexity

We measure the "Complexity" of the Python code in Code SFT datasets. "Complexity" in this sense, relates to how hard a particular piece of code is to write or understand. A variety of metrics can be used to characterize this:

Cyclomatic Complexity: The average cyclomatic complexity of all Python programs in the datasets under consideration

Halstead Measures: Halstead's Measures were originally formulated to identify measurable properties of software and the relations between them. We use the vocabulary, length, and difficulty metrics from the Halstead Measures. Vocabulary is the count of unique operators and operands in the code, while Length corresponds to the total number of operators and operands in the code. The difficulty here refers to the measure of a program's difficulty in writing and understanding.

Zhao et al. (2023); Xu et al. (2023); Luo et al. (2023) suggest that increased complexity in instruction datasets may give

improvements in performance. From Table 2, we can see that our curated dataset has challenging problems that have complexity metrics far above the other datasets. However, we also see that we are still lacking in comparison when it comes to diversity of instructions.

Model	*pass@1*	*pass@10*
Closed-Source Models		
GPT-3.5	48.10%	-
GPT-4	**67.00%**	-
PaLM-Coder	36.00%	-
Codex	28.8%	46.8%
Open-Source Models		
CodeLlama-Python-7B	38.4%	70.30%
CodeLlama-Python-13B	43.30%	77.40%
CodeLlama-Python-34B	53.70%	**82.80%**
Llama-2-70B	29.9%	-
CodeT5+	29.30%	-
StarCoder Prompted	40.8%	-
phi-2	48.00%*[0]	-
Ours		
phi-2 (K=1)	49.27%	70.97%
phi-2 (K=5)	50.00%	71.81%
phi-2 (K=10)	53.54%	70.77%
CodeLlama-Python-7B (K=1)	41.95%	68.33%
CodeLlama-Python-7B (K=5)	44.63%	69.40%
CodeLlama-Python-7B (K=10)	42.44%	69.50%

Table 3. Human Eval Results for our model architecture in comparison to others. We use a Temperature of 0.2 while evaluating *pass@1* and use a Temperature of 0.8 while evaluating *pass@10*

4. Training and Inference

We propose a new model training and inference pipeline. The components of the pipeline include (1) Sentence Embedding Model, (2) K-means clustering model, (3) K-LoRA Model Training, (4) Inference Strategy

4.1. Sentence Embedding Model

The first part of our pipeline involves the use of a Sentence Embedding Transformer model. These are models that are trained to produce semantically meaningful embeddings given any string. We use the *e5-base-v2* model (Wang et al., 2022a) to embed the question into a fixed 768-dimensional vector. The idea here is to capture the semantic meaning of the question. We pass in the set of questions $Q = \{q_1, q_2, q_3, \ldots, q_n\}$ to the Sentence embedding model to get the corresponding embedding vectors $\mathbf{E} = \{\mathbf{e_1}, \mathbf{e_2}, \mathbf{e_3}, \ldots, \mathbf{e_n}\}$ where n is the size of our fine-tuning dataset.

4.2. K-means Clustering Model

We then train a K-means clustering model to perform unsupervised clustering on the embedding vectors $\mathbf{E}$ obtained from the sentence embedding model. The idea here is to cluster the data points into K clusters based on the semantic meaning of the question and then train a QLoRA (Dettmers et al., 2023) Adaptation for each of these K data slices. By doing this, we hypothesize that each model can be a "specialist" in a small sub-group of semantically similar questions and yield better performance overall. This fitted model is saved for use during inference.

4.3. K-LoRA Model Training

Once the Clustering model has assigned clusters to each data point in the training data, we proceed to train K LoRA weight updates, one for each cluster. The total number of data points remains the same, now split between K clusters.

We use QLoRA (Dettmers et al., 2023), a quantization-enabled variant of LoRA (Hu et al., 2021), as a PEFT (Parameter Efficient Fine Tuning) method to train our Pre-trained Language Models (PLMs). This elicits close to full-FT performance while using only a fraction of the parameters. We perform this parameter efficient fine-tuning on two models: The phi-2 model (Hughes) with 2.7 Billion parameters and the 7 Billion parameter variant of the CodeLlama-Python model (Rozière et al., 2023). We used the Huggingface chat templating feature [4] to convert the question-code pairs into prompt styles that are suited to each model. The time taken to train our K adaptations is just a little higher than the training time for a single adaptation on all the instances in the training set. This is because the data is split into K slices, and the time taken to train K LoRA Adaptations on these slices is equivalent to training one model on all the slices together. Once a LoRA Adapter is trained, it can be saved, and another adapter can be trained using the same base PLM, an operation involving little additional latency according to Hu et al. (2021).

We perform fine-tuning for 2 epochs with a learning rate of $2e{-}4$. The training batch size was 4 for the phi-2 model and 2 for the Codellama-Python-7B model. We used a LoRA *rank* value of 128, *alpha* value of 256, and a *dropout* value of 0.1. We choose relatively high LoRA parameters by taking insights from previous work (Raschka, 2023). We opted for mixed-precison training while keeping the sequence length at 1024 tokens. When multiple LoRA checkpoints

[4] https://huggingface.co/docs/transformers/main/en/chat_templating

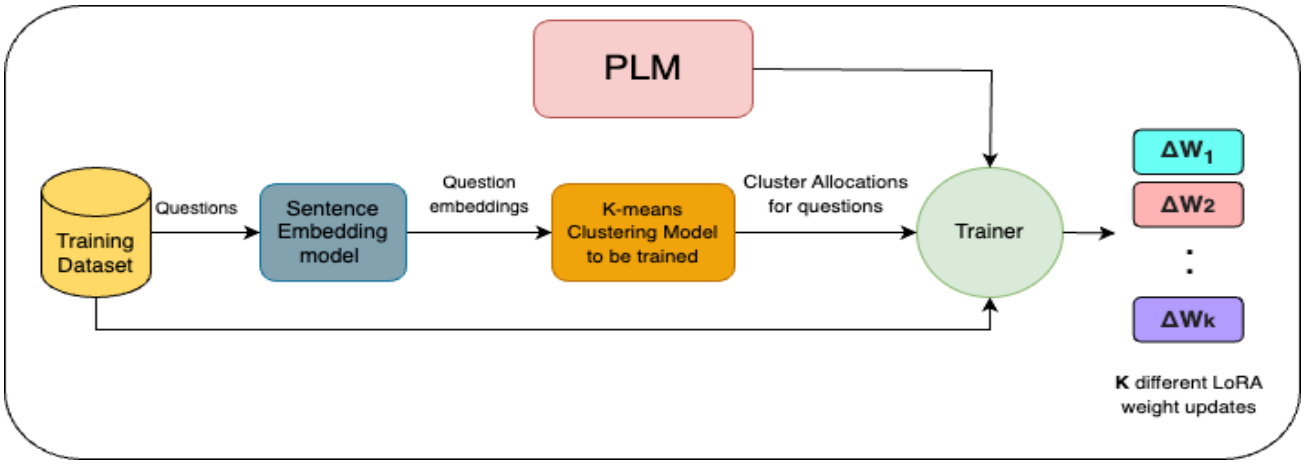

Figure 2. Training pipeline we followed to train our model. The Training dataset we used corresponds to the dataset we collected as detailed in Section 2. The PLM (Pretrained Language model) which we use as our base for training is one of two models: The phi-2 model (2.7B) (Hughes) and CodeLlama-7B (Rozière et al., 2023) Python variant. The Sentence Embedding model we use is e5-base-v2 (Wang et al., 2022a). The K different ΔW's correspond to K different LoRA weight updates

exist, we choose the one with the lowest eval loss from our held-out validation set. LoRA checkpoints are saved every 200 steps. All other hyperparameters used for finetuning are given in Appendix A. All training and inference was done using a 1xA100 80GB.

4.4. Inference Strategy

For batched inference, we can pre-cluster the data points and sequentially load the K LoRA adapters to merge them with the PLM. A weight update arising from using LoRA can be reversed by subtracting the weight update matrix from the model. By doing this, we can efficiently swap in and swap out LoRA weights with the PLM to perform generations for each data cluster sequentially. For Online inference (producing generations in real-time when the request is received), all K model weights (after performing the LoRA weight update) need to be loaded, requiring a higher memory footprint. Moreover, the embedding and clustering model latency may play a bigger role here than in batched inference.

5. Results

We evaluate our model's code generation performance on the HumanEval benchmark (Chen et al., 2021). We calculate the *pass@1* and *pass@10* for two models that we fine-tuned using the methods detailed in previous sections: CodeLlama-7B and phi-2. We choose $K \in \{1, 5, 10\}$. $K = 1$ corresponds to standard fine-tuning and inference, where all the questions are assigned to the same cluster and involve only one set of LoRA weight updates. $K = 5$ and $K = 10$ correspond to scenarios where the evaluation questions are assigned to one of 5 or 10 pre-computed clusters, respectively. These cluster allocations are performed using a trained K-means clustering model that is fit on the training dataset.

Table 3 shows that as we increase K, we get increased *pass@1* performance while keeping the number of training data points the same. We do, however, see that there is a point of diminishing returns when we train the CodeLlama-Python-7B with K=10 total clusters. Since the dataset is split into ten smaller parts, the model may be overfitting to these data points, and its generalization ability decreases. We don't see the same in phi-2, probably because we are training fewer parameters (2.7B) when compared to our CodeLlama-Python model (7B) for the same dataset size.

We do see that the pass@10 performance decreases significantly from the PLM it is trained on, and there is fluctuation when changing the value of K. We leave it to future work to investigate what causes this performance drop and fluctuation along with methods to remedy it. Our most performant model corresponds to phi-2 (K=10) achieves 53.54% *pass@1* on HumanEval, which is almost on par with CodeLlama-Python-34B.

6. Limitations and Future Work

***pass@10* performance drop and fluctuation:** Table 3 shows that while *pass@1* increases with K, *pass@10* shows no clear trend. In fact, the *pass@10* even in cases where $K = 1$, shows performance degradation when compared to the *pass@10* of the PLM it was trained on. We don't exactly know why this is happening, but we leave it to future work

to try and understand this

Relatively less diversity in training dataset: The training dataset curated by us had less diversity as measured by semantic and syntactic metrics when compared to other datasets. This may lead to subpar results when training the PLM as well as fitting the K-means clustering model (Wan et al., 2023; Lu et al., 2023). Performing the deduplication step with a stricter similarity threshold might help our case here.

Large GPU footprint and increased latency during Online inference: Our model architecture requires K times the original Model GPU footprint while performing Online inferences. Moreover, there is increased latency arising from the Sentence Embedding and K-means clustering model.

Future work includes investigating the *pass@10* performance drop and increasing diversity in our dataset. Training on a combination of data from other SFT datasets mentioned in table 1 and 2 can also be a path to improve performance.

7. Conclusion

We curate a dataset with challenging question-code pairs by means of synthetic data generation from GPT-3.5. Novel prompt techniques were evolved to elicit complex combinations of existing questions. We also came up with a novel model architecture based on training K-LoRA weights for K slices of our dataset as labeled by a fitted K-Means clustering model. We fit this clustering model on Sentence embedding of questions (as calculated by an e5-base-v2 model (Wang et al., 2022a)) from our training dataset. During evaluation time, we compute the Sentence embedding of the incoming question, then assign it to a cluster and pass it on to the corresponding finetuned model trained on the data points from that assigned cluster. Our finetuned phi-2 2.7B $(K = 10)$ model achieves a *pass@1* of **53.54%** on HumanEval, comparable to that of a 34B parameter CodeLlama-Python model.

8. Course Evaluation

Indication of whether we completed the course evaluation (**Yes**) or not (**No**)

Karthik Suresh: **Yes**
Hafeez Ali Anees Ali: **Yes**
Calvin Kranig: **Yes**

References

Claude 2 — anthropic.com. https://www.anthropic.com/index/claude-2. [Accessed 18-12-2023].

Cao, Y., Kang, Y., and Sun, L. Instruction mining: High-quality instruction data selection for large language models. *arXiv preprint arXiv:2307.06290*, 2023.

Chaudhary, S. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca, 2023.

Chen, H., Zhang, Y., Zhang, Q., Yang, H., Hu, X., Ma, X., Yanggong, Y., and Zhao, J. Maybe only 0.5% data is needed: A preliminary exploration of low training data instruction tuning. *arXiv preprint arXiv:2305.09246*, 2023.

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

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

Ding, N., Chen, Y., Xu, B., Qin, Y., Zheng, Z., Hu, S., Liu, Z., Sun, M., and Zhou, B. Enhancing chat language models by scaling high-quality instructional conversations, 2023.

Eldan, R. and Li, Y. Tinystories: How small can language models be and still speak coherent english?, 2023.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022.

Feng, Y., Martins, R., Bastani, O., and Dillig, I. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018.

Gulwani, S., Polozov, O., Singh, R., et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4 (1-2):1–119, 2017.

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pp. 2790–2799. PMLR, 2019.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Hughes, A. Phi-2: The surprising power of small language models — microsoft.com.

Kandpal, N., Wallace, E., and Raffel, C. Deduplicating training data mitigates privacy risks in language models. In *International Conference on Machine Learning*, pp. 10697–10707. PMLR, 2022.

Lee, K., Ippolito, D., Nystrom, A., Zhang, C., Eck, D., Callison-Burch, C., and Carlini, N. Deduplicating training data makes language models better. *arXiv preprint arXiv:2107.06499*, 2021.

Li, X., Yu, P., Zhou, C., Schick, T., Zettlemoyer, L., Levy, O., Weston, J., and Lewis, M. Self-alignment with instruction backtranslation. *arXiv preprint arXiv:2308.06259*, 2023.

Lu, K., Yuan, H., Yuan, Z., Lin, R., Lin, J., Tan, C., Zhou, C., and Zhou, J. instag: Instruction tagging for analyzing supervised fine-tuning of large language models, 2023.

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

McCarthy, P. M. and Jarvis, S. Mtld, vocd-d, and hd-d: A validation study of sophisticated approaches to lexical diversity assessment. *Behavior research methods*, 42(2): 381–392, 2010.

Mixtral. Mixtral of experts — mistral.ai. https://mistral.ai/news/mixtral-of-experts/, 2023.

Mukherjee, S., Mitra, A., Jawahar, G., Agarwal, S., Palangi, H., and Awadallah, A. Orca: Progressive learning from complex explanation traces of gpt-4. *arXiv preprint arXiv:2306.02707*, 2023.

Nickrosh. nickrosh/Evol-Instruct-Code-80k-v1 · Datasets at Hugging Face — huggingface.co. https://huggingface.co/datasets/nickrosh/Evol-Instruct-Code-80k-v1.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

OpenAI. Gpt-4 technical report, 2023.

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

Raschka, S. Finetuning LLMs with LoRA and QLoRA: Insights from Hundreds of Experiments - Lightning AI — lightning.ai. https://lightning.ai/pages/community/lora-insights/, 2023.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2023.

Silcock, E., D'Amico-Wong, L., Yang, J., and Dell, M. Noise-robust de-duplication at scale. Technical report, National Bureau of Economic Research, 2022.

Song, C., Zhou, Z., Yan, J., Fei, Y., Lan, Z., and Zhang, Y. Dynamics of instruction tuning: Each ability of large language models has its own growth pace. *arXiv preprint arXiv:2310.19651*, 2023.

Theblackcat102. theblackcat102/evol-codealpaca-v1 · Datasets at Hugging Face — huggingface.co. https://huggingface.co/datasets/theblackcat102/evol-codealpaca-v1.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023.

Wan, F., Huang, X., Yang, T., Quan, X., Bi, W., and Shi, S. Explore-instruct: Enhancing domain-specific instruction coverage through active exploration. *arXiv preprint arXiv:2310.09168*, 2023.

Wang, L., Yang, N., Huang, X., Jiao, B., Yang, L., Jiang, D., Majumder, R., and Wei, F. Text embeddings by weakly-supervised contrastive pre-training, 2022a.

Wang, X., Dillig, I., and Singh, R. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N. A., Khashabi, D., and Hajishirzi, H. Self-instruct: Aligning

language model with self generated instructions. *arXiv preprint arXiv:2212.10560*, 2022b.

Wang, Y., Mukherjee, S., Liu, X., Gao, J., Awadallah, A. H., and Gao, J. Adamix: Mixture-of-adapter for parameter-efficient tuning of large language models. *arXiv preprint arXiv:2205.12410*, 1(2):4, 2022c.

Wang, Y., Wang, X., Li, J., Chang, J., Zhang, Q., Liu, Z., Zhang, G., and Zhang, M. Harnessing the power of david against goliath: Exploring instruction data generation without using closed-source models. *arXiv preprint arXiv:2308.12711*, 2023.

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. Finetuned language models are zero-shot learners, 2022.

Wei, Y., Wang, Z., Liu, J., Ding, Y., and Zhang, L. Magicoder: Source code is all you need, 2023.

Wu, S., Lu, K., Xu, B., Lin, J., Su, Q., and Zhou, C. Self-evolved diverse data sampling for efficient instruction tuning. *arXiv preprint arXiv:2311.08182*, 2023.

Xu, C., Sun, Q., Zheng, K., Geng, X., Zhao, P., Feng, J., Tao, C., and Jiang, D. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*, 2023.

Zhao, Y., Yu, B., Hui, B., Yu, H., Huang, F., Li, Y., and Zhang, N. L. A preliminary study of the intrinsic relationship between complexity and alignment. *arXiv preprint arXiv:2308.05696*, 2023.

Zhou, C., Liu, P., Xu, P., Iyer, S., Sun, J., Mao, Y., Ma, X., Efrat, A., Yu, P., Yu, L., et al. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206*, 2023.

Zhu, M., Jain, A., Suresh, K., Ravindran, R., Tipirneni, S., and Reddy, C. K. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*, 2022.

A. Hyperparameters

Parameter	phi-2	codellama
per_device_train_batch_size	4	4
per_device_eval_batch_size	1	1
gradient_accumulation_steps	2	2
learning_rate	0.0002	0.0002
max_grad_norm	0.3	0.3
weight_decay	0.001	0.001
lora_alpha	256	256
lora_dropout	0.1	0.1
lora_r	128	128
max_seq_length	1024	1024
use_4bit	true	true
use_nested_quant	false	false
bnb_4bit_compute_dtype	float16	float16
bnb_4bit_quant_type	nf4	nf4
num_train_epochs	2	2
Mixed-precision training	true	true
gradient_checkpointing	true	true
optim	paged_adamw_32bit	paged_adamw_32bit
lr_scheduler_type	cosine	cosine
warmup_ratio	0.05	0.05
Parameters Trained	262,364,160	262,410,240
Target Modules	['Wqkv','out_proj','fc1','fc2']	['gate_proj', 'down_proj', 'up_proj', 'q_proj', 'v_proj', 'k_proj', 'o_proj']

Table 4. Hyperparameters

B. Complexity Metrics Explanation

B.0.1. CYCLOMATIC COMPLEXITY

Construct	Effect on CC	Reasoning
if	1	An `if` statement is a single decision.
elif	1	The `elif` statement adds another decision.
else	0	The `else` statement does not cause a new decision. The decision is at the `if`.
for	1	There is a decision at the start of the loop.
while	1	There is a decision at the `while` statement.
except	1	Each `except` branch adds a new conditional path of execution.
finally	0	The finally block is unconditionally executed.
with	1	The `with` statement roughly corresponds to a try/except block (see PEP 343 for details).
assert	1	The `assert` statement internally roughly equals a conditional statement.
Comprehension	1	A list/set/dict comprehension of generator expression is equivalent to a for loop.
Boolean Operator	1	Every boolean operator (and, or) adds a decision point.

Table 5. Cycolmatic Complexity corresponds to the number of linearly independent paths through the code

The formula for cyclomatic complexity is given by:

$$V = E - N + 2P$$

Where:

V: Cyclomatic Complexity

E: Number of edges in the control flow graph

N: Number of nodes in the control flow graph

P: Number of connected components (usually 1 for a single program)

B.0.2. HALSTEAD METRICS

$$\begin{aligned}
\text{Program vocabulary: } & \eta = \eta_1 + \eta_2 \\
\text{Program length: } & N = N_1 + N_2 \\
\text{Calculated program length: } & N^{\wedge} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \\
\text{Volume: } & V = N \log_2 \eta \\
\text{Difficulty: } & D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2} \\
\text{Effort: } & E = D \cdot V
\end{aligned}$$