What Makes Good In-context Demonstrations for Code Intelligence Tasks with LLMs?

Shuzheng Gao[1†], Xin-Cheng Wen[1], Cuiyun Gao[1*], Wenxuan Wang[2], Hongyu Zhang[3], Michael R. Lyu[2]

[1] School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China
[2] Department of Computer Science and Engineering, The Chinese University of Hong Kong, China
[3] School of Big Data and Software Engineering, Chongqing University, China

szgao98@gmail.com, xiamenwxc@foxmail.com, gaocuiyun@hit.edu.cn, hyzhang@cqu.edu.cn, {wxwang,lyu}@cse.cuhk.edu.hk

Abstract—Pre-trained models of source code have gained widespread popularity in many code intelligence tasks. Recently, with the scaling of the model and corpus size, large language models have shown the ability of in-context learning (ICL). ICL employs task instructions and a few examples as demonstrations, and then inputs the demonstrations to the language models for making predictions. This new learning paradigm is training-free and has shown impressive performance in various natural language processing and code intelligence tasks. However, the performance of ICL heavily relies on the quality of demonstrations, e.g., the selected examples. It is important to systematically investigate how to construct a good demonstration for code-related tasks. In this paper, we empirically explore the impact of three key factors on the performance of ICL in code intelligence tasks: the selection, order, and number of demonstration examples. We conduct extensive experiments on three code intelligence tasks including code summarization, bug fixing, and program synthesis. Our experimental results demonstrate that all the above three factors dramatically impact the performance of ICL in code intelligence tasks. Additionally, we summarize our findings and provide takeaway suggestions on how to construct effective demonstrations, taking into account these three perspectives. We also show that a carefully-designed demonstration based on our findings can lead to substantial improvements over widely-used demonstration construction methods, e.g., improving BLEU-4, EM, and EM by at least 9.90%, 175.96%, and 50.81% on code summarization, bug fixing, and program synthesis, respectively.

I. INTRODUCTION

Recently, there has been an increasing focus on code intelligence research, aiming at reducing the burden on software developers and enhancing programming productivity [1], [2]. With the large-scale open-source code corpora and the progress of deep learning techniques, various neural source code models have been developed and have achieved state-of-the-art performance on a variety of code intelligence tasks including code summarization [3], bug fixing [4], and program synthesis [5].

In recent years, the advent of pre-training techniques has significantly advanced progress in this area. For instance, CodeBERT [6], a BERT-based model pre-trained on
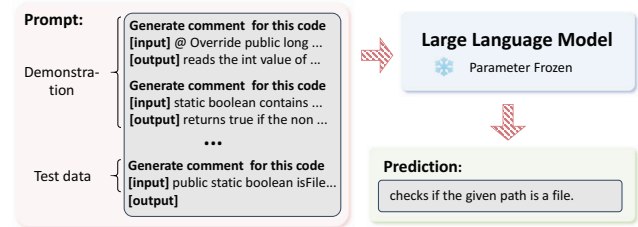

Fig. 1: An example of in-context learning on code summarization task.

both natural and programming language data, has demonstrated promising performance in various code intelligence tasks [4], [7]. Other subsequent pre-trained code models such as PLBART [8] and CodeT5 [9] further achieve much improvement over CodeBERT. However, the size and training data of the above models are limited, which may hinder the models from achieving their potential [10]. In these years, we have witnessed explosive growth in the size of pre-trained models. Various billion-level large language models are proposed such as GPT-3 [11] and PALM-E [12]. For instance, the size of the pre-trained model PALM-E [12] (562B) in 2023 is over two thousand times larger than the largest model BERT [13] (223M) in 2018.

As the size of language models and training data continues to increase, large language models (LLMs) demonstrate various emergent abilities. One such ability is in-context learning (ICL) [11], [14], which allows models to learn from just a few examples within a specific context. As shown in Fig. 1, ICL utilizes a demonstration including task instructions and a few examples to describe the task, which is then concatenated with a query question to form an input for the language model to make predictions. The most significant difference between ICL and traditional tuning methods such as fine-tuning [6] is that it is training-free and does not need parameter updates. The training paradigm enables ICL to be directly used upon any LLMs and significantly reduces the training costs of adapting models to new tasks [11]. Recent studies show that ICL has achieved impressive results in various fields, including logic

---

reasoning [15], dialogue system [16], and program repair [17]–[19], and can even outperform the supervised fine-tuning methods trained on large task-specific data.

Although ICL has been proven useful in code intelligence tasks, the performance of ICL is known to strongly rely on the quality of demonstrations [20], [21]. Existing studies [17], [19] mainly construct demonstrations by randomly selecting and arranging the demonstration examples. To the best of our knowledge, there is a lack of an in-depth investigation of ICL for code intelligence tasks. Considering the impressive performance of ICL, it is worth understanding the impact of demonstration design and investigating the challenges of applying ICL for code intelligence tasks. In this work, we systematically analyze how different demonstration construction methods influence the performance of ICL on code intelligence tasks, aiming at answering the following question: *What makes good in-context demonstrations for code intelligence tasks with LLMs?* By analyzing the design space of in-context demonstrations, our study mainly focuses on three aspects of in-context demonstrations, including the selection, order, and number of demonstration examples. We conduct an experimental study on three popular code intelligence tasks including code summarization, bug fixing, and program synthesis. Specifically, we mainly investigate the following four research questions (RQs):

1) What kind of selection methods are helpful for ICL in code intelligence tasks?
2) How should demonstration examples be arranged for ICL in code intelligence tasks?
3) How does the number of demonstration examples in a prompt impact the performance of ICL in code intelligence tasks?
4) How is the generalizability of our findings?

To answer the first RQ, we compare a wide range of demonstration selection methods such as random selection, similarity-based selection, and diversity-based selection. We also experiment with different retrieval methods in the similarity-based selection to find which retrieval method is more helpful for ICL in code intelligence tasks. To answer the second RQ, we compare random ordering with two other ordering methods including similarity and reverse similarity, towards exploring the impact of different ordering methods. To answer RQ3, we change the number of demonstration examples in the prompt and investigate whether the performance of ICL also grows with the increase in the number of demonstration examples. To answer the last RQ, we experiment on different LLMs and validate the findings we achieve in the above RQs.

Key Findings. Based on the extensive experiments, our study reveals several key findings:

1) Both similarity and diversity in demonstration selection are important factors for ICL in code intelligence tasks. They not only enhance the overall performance but also lead to more stable predictions.
2) The order of demonstration examples has a large impact

on the performance of ICL. In most cases, placing similar samples at the end of a prompt achieves better results.
3) Increasing the number of demonstration examples can be beneficial for ICL, provided that the examples are not cut off due to the input length limitation of LLMs. Careful attention should be paid to this issue, as the length of code is generally longer than natural language.

We also show that a carefully-designed demonstration based on the achieved findings can lead to substantial improvements over the widely-used demonstration construction methods [17], [19], [22], e.g., improving BLEU-4, EM, and EM by at least 9.90%, 175.96%, and 50.81% on code summarization, bug fixing and program synthesis, respectively.

Contributions. In summary, the main contributions of this work are as follows:

1) To the best of our knowledge, this paper represents the first systematic study on how to construct effective demonstrations for code intelligence tasks.
2) Our comprehensive exploration of demonstration design highlights a range of findings for improving ICL's performance in code intelligence tasks.
3) We discuss the implications of our findings for researchers and developers and future work for code intelligence tasks in the era of large language models.

II. BACKGROUND

A. Large Language Models

LLMs have become a ubiquitous part of Natural Language Processing (NLP) due to their exceptional performance [11], [23]. These models typically follow the Transformer [24] architecture and are trained on large-scale corpora using self-supervised objectives such as masked language modeling [13]. The size of LLMs has increased significantly in the past few years. For example, the parameters of recent LLMs like GPT-3 [11] and PALM-E [12] are over one hundred billion. Apart from the LLMs for general purposes, there are also LLMs with billion-level parameters trained on code corpora, such as AlphaCode [25], and Codex [2]. The OpenAI's Codex is a large pre-trained code model that is capable of powering Copilot. AlphaCode [25] is a 41-billion-large model trained for generating code in programming competitions like Codeforces. Recently, LLMs like ChatGPT [26] and GPT-4 [23] have also shown impressive performance in many code intelligence tasks.

Apart from proposing new LLMs, how to effectively leverage them has also become an important research topic. A prevalent method is to fine-tune the model and update its parameters on downstream datasets [13]. Recently, prompt-based fine-tuning has been proposed, which aims to convert the training objective of downstream tasks into a similar form as the pre-training stage [27], [28]. Considering the cost of tuning the whole model, various Parameter Efficient Tuning methods have been proposed, such as Adapter [29], Lora [30], and prefix tuning [31]. These methods keep most of the parameters in the model frozen and only tune a small portion of them.
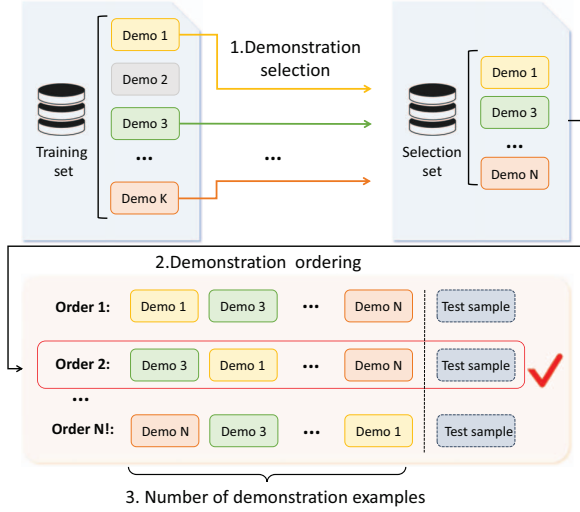
Fig. 2: Illustration of design space of in-context demonstrations.

B. In-context Learning

Tuning a large pre-trained model can be expensive and impractical for researchers, especially when limited fine-tuned data is available for certain tasks. ICL offers a new alternative that uses language models to perform downstream tasks without requiring parameter updates [11], [14]. It leverages a demonstration in the prompt to help the model learn the input-output mapping of the task. This new paradigm has achieved impressive results in various tasks such as logic reasoning and program repair [15], [17], [19].

Specifically, as shown in Fig. 1, ICL employs N demonstration examples $\{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$ and further reconstructs them into reconstructed examples $\{(x'_1, y'_1), (x'_2, y'_2), ..., (x'_N, y'_N)\}$ by natural language instructions and prompt template, where x_i, y_i, x'_i, y'_i are the input, output, reconstructed input, and reconstructed output, respectively. Typically, The value of N is relatively small, i.e., fewer than 50 samples, which is significantly smaller than the size of the training set in previous fine-tuned methods [6], [9]. This setting is referred to as *few-shot in-context learning*. Specially, when the value of N is zero, it is called the *zero-shot in-context learning setting*. Then, ICL concatenates the reconstructed demonstration examples d_1 to d_N literally into demonstration $D = x'_1 \parallel y'_1 \parallel x'_2 \parallel y'_2 \parallel ... \parallel x'_N \parallel y'_N$, and further adds the test sample at the end to construct the input prompt $P = D \parallel x'_{test}$, where $\parallel$ denotes the literal concatenation operation. This prompt is finally fed into the language model for predicting the label y_{test} for test samples.

Previous studies in NLP have shown that the performance of ICL is strongly dependent on the quality of the demonstration. For example, Liu et al. [20] show that selecting demonstration examples with higher similarity or increasing the number of demonstration examples can improve ICL's performance. The results in [21] show that the order of demonstration examples also has a large impact on the results. Following previous studies, we summarize three key factors to consider when designing a demonstration for ICL: the selection, ordering, and number of demonstration examples, as shown in Fig. 2.

We would like to further clarify that there are two types of demonstration in ICL: **task-level** demonstration and **instance-level** demonstration [32], [33]. The task-level demonstration uses the same demonstration examples for all test samples and does not take the difference of each test sample into consideration, while the instance-level demonstration selects different demonstration examples for different test samples. Although instance-level demonstrations generally perform better than task-level demonstrations, it requires a labeled training set in advance for retrieval. The task-level demonstration is more flexible as it can be used in scenarios where very few data are labeled, or no labeled data are available by selecting few representative data for human labeling [33]. In this paper, we investigate both the task-level and instance-level demonstration construction methods for code intelligence tasks.

III. EXPERIMENTAL EVALUATION

A. Research Questions

We design experiments to investigate the impact of the selection, ordering, and number of demonstrations on ICL for code intelligence tasks. Our research aims to answer the following questions:

RQ1: What kind of selection methods are helpful for ICL in code intelligence tasks?

RQ2: How should demonstration examples be arranged for ICL in code intelligence tasks?

RQ3: How does the number of demonstration examples in a prompt impact the performance of ICL in code intelligence tasks?

RQ4: How is the generalizability of our findings?

In RQ1, we aim at verifying whether selecting similar and diverse demonstration examples is helpful. Besides, we also compare different retrieval methods to analyze the impact of different similarity measurement methods for ICL. RQ2 aims at investigating the influence of ordering methods by comparing random ordering with similarity-based ordering. In RQ3, we want to explore whether increasing the number of examples could bring better performance for ICL. In RQ4, we evaluate whether the findings achieved in RQ1-RQ3 are also applicable to different LLMs for verifying the generalizability of the findings.

B. Evaluation tasks

We conduct experiments on three popular code intelligence tasks: code summarization, bug fixing, and program synthesis.

1) Code Summarization: Code summarization, also known as code comment generation, aims to generate useful comments automatically for a given code snippet [7]. Recent work mainly formulates it as a sequence-to-sequence neural machine translation (NMT) task and involves pre-trained techniques to achieve better performance [9], [34].

TABLE I: Statistics of the benchmark datasets.

Task	Datasets	Train	Dev	Test
Code Summarization	CSN-Java	164,923	5,183	10,955
	TLC	69,708	8,714	6,489
Bug Fixing	B2F$_{small}$	46,628	5,828	5,831
	B2F$_{medium}$	53,324	6,542	6,538
Program Synthesis	CoNaLa	2,389	-	500

Datasets. To evaluate the performance of code summarization, we use two widely-used datasets: CodeSearchNet (CSN) [35] and TLCodeSum (TLC) [7]. CSN is a large-scale source code dataset mined from open-source GitHub repositories. It contains code summarization data in six programming languages, i.e., Java, Go, JavaScript, PHP, Python, and Ruby. The dataset is split into training, validation, and test sets in the proportion of 8:1:1. In this study, considering our time and resource limitation, we use the Java portion of the filtered CSN dataset in CodeBERT [6], which contains 181,061 samples across the training, validation, and test sets for evaluation. TLC has 87,136 code-comment pairs crawled from 9,732 open-source Java projects on GitHub with at least 20 stars. The code snippets are all at the method level and the comments of corresponding Java methods are considered as code summaries. The portion of training, validation, and test set is also 8:1:1. As reported in previous work, there are duplicated data in the training and test set. Therefore, we follow previous work [36] and remove the duplicated data, and finally get a test set with 6,489 samples.

Metrics. We use three widely-adopted metrics for code summarization evaluation: BLEU-4 [37], ROUGE-L [38] and METEOR [39] for evaluation. These metrics evaluate the similarity between generated summaries and ground-truth summaries and are widely used in code summarization [3], [36], [40].

2) Bug Fixing: Bug fixing is the task of automatically fixing bugs in the given code snippet. It helps software developers find and fix software errors [4], [41].

Datasets. The dataset for bug fixing is B2F which is collected by Tufano et al. [4] from bug-fixing commits in GitHub. We use the multi-model version proposed in MODIT [42] for experiments as it contains both the code changes and the fix instruction. The model is given both the buggy code and natural language fix guidance to predict the fixed code. We follow their original setting to split the dataset into two parts B2F$_{medium}$ and B2F$_{small}$ based on the length of code tokens (the code length of B2F$_{medium}$ is between 50 and 100 tokens and that of B2F$_{small}$ is below 50 tokens).

Metrics. We follow previous work [43] and use Exact Match (EM) and BLEU-4 for both datasets.

3) Program Synthesis: Program synthesis is the task of generating source code based on the given natural language description. It provides practical assistance to developers and enhances their productivity [2].

Datasets. For program synthesis, we use the CoNaLa [44] dataset for evaluation. This dataset consists of 2,889 ⟨intent, code⟩ pairs mined from Stack Overflow in Python. We directly

TABLE II: Prompt template for each task. Here text in the form of {#xxx} will be filled in actual inputs from the dataset.

Task	Template
Code Summarization	Generate comment (summarization) for this code [input] {#code} [output] {#comment}
Bug Fixing	Fix the bug according to the guidance [input] {#buggy code} <s> {#instruction} [output] {#fixed code}
Program Synthesis	Generate code based on the requirement [input] {#requirement}[output] {#code}

use the original partition of the dataset, which includes 2,389 samples for training and 500 samples for testing.

Metrics. We follow previous work [43] and evaluate the performance of program synthesis with four metrics including Exact Match (EM), CodeBLEU (CB), Syntax Match (SM), and Dataflow Match (DM). EM measures whether the code generated by the model is identical to the goal code. CB [45] is a modified version of BLEU designed specifically for code, which leverages syntax and semantic information such as Abstract Syntax Tree (AST) and data flow to measure the similarity of two code snippets. SM and DM are two components that calculate the matching subtrees and data flow edges' proportion, respectively.

C. Implementation

We utilize the OpenAI Codex (code-davinci-002) API [2] in our paper for all experiments in the first three RQs. In RQ4, we further use the API of GPT-3.5 (text-davinci-003) [11] and ChatGPT (gpt-3.5-turbo) [26] for experiments. As for the hyperparameters of the APIs, following the previous work [46], [47], we set the temperature to 0 to get the deterministic output. The frequency_penalty and presence_penalty are also set to 0. The input length limitation of Codex, GPT-3.5, and ChatGPT is 8,001, 4,096, and 4,097 tokens, respectively. Hence we cut off the input code of each demonstration example to $\frac{8001}{N+1}$, $\frac{4096}{N+1}$, and $\frac{4097}{N+1}$ tokens, respectively, where N represents the number of demonstration examples. Empirically, it took approximately 6 hours to evaluate 1,000 examples for Codex. To avoid excessive time costs, we randomly sample a small test set (2,000 samples) for each dataset with over 2,000 test samples. We use four examples in the demonstration in RQ1 and RQ2, and further discuss the impact of the number of demonstration examples in RQ3. The templates used in this study are shown in Table II. We also show some examples in our GitHub repository[1]. We conduct all the experiments on a server with 2 Nvidia RTX 3090 GPUs. The GPUs are used in the dense retrieval process.

IV. EXPERIMENTAL RESULTS

A. RQ1: Demonstration Selection

1) Experimental design: We first explore the impact of demonstration selection methods on ICL for code-related tasks. To provide a comprehensive study, we adopt different

---

[1]https://github.com/gszsectan/ICL/tree/master/prompts

TABLE III: Experimental results of different demonstration selection methods on Code Summarization. "Avg" and "CV" denote the average results and Coefficient of Variation over three different orders, respectively.

Approach	Code Summarization											
	CSN						TLC					
	BLEU-4		ROUGE-L		METEOR		BLEU-4		ROUGE-L		METEOR	
	Avg	CV	Avg	CV	Avg	CV	Avg	CV	Avg	CV	Avg	CV
Task-level Demonstration												
Random	19.64	1.44	35.46	1.88	15.30	1.54	17.29	0.71	34.28	0.61	12.48	0.67
KmeansRND	**20.71**	0.82	**38.03**	0.44	**16.34**	0.83	**17.91**	1.19	**35.69**	1.60	**13.48**	0.91
Instance-level Demonstration												
BM-25	**22.35**	0.46	38.31	0.56	**17.01**	0.78	**36.96**	0.84	**51.42**	0.79	**24.22**	0.99
SBERT	22.27	0.23	**38.39**	0.42	16.91	0.22	36.42	0.61	50.47	0.40	23.86	0.68
UniXcoder	22.11	0.61	38.23	0.53	16.81	0.23	36.77	0.52	51.11	0.29	24.08	0.79
CoCoSoDa	21.92	0.46	37.85	0.22	16.78	0.24	36.91	0.69	50.69	0.53	24.08	0.39
Oracle (BM-25)	27.69	0.43	46.17	0.14	20.26	0.22	43.16	0.15	59.17	0.09	28.09	0.16

TABLE IV: Experimental results of different demonstration selection methods on Bug Fixing.

Approach	Bug Fixing							
	$B2F_{medium}$				$B2F_{small}$			
	BLEU-4		EM		BLEU-4		EM	
	Avg	CV	Avg	CV	Avg	CV	Avg	CV
Task-level Demonstration								
Random	**86.96**	0.16	7.26	16.18	71.18	0.56	9.95	6.33
KmeansRND	86.91	0.17	**9.03**	5.45	**72.89**	1.36	**10.37**	3.86
Instance-level Demonstration								
BM-25	**88.05**	0.09	**21.85**	1.78	**77.54**	0.13	**30.45**	0.96
SBERT	87.98	0.06	19.00	2.88	76.26	0.16	26.15	0.87
UniXcoder	87.87	0.09	19.14	2.00	77.52	0.07	29.93	0.51
CoCoSoDa	87.73	0.07	19.23	0.74	76.45	0.07	27.40	1.04

kinds of demonstration selection methods for the three code intelligence tasks.

For task-level demonstration, we need to select a group of demonstration examples for the whole test set, as illustrated in Section II-B. To explore the influence of different in-context demonstration examples on the performance of ICL, we randomly select three groups of demonstration examples from the training set, and evaluate their performance on different tasks, denoted as *Random*. Besides, we further investigate whether improving the diversity of demonstration examples is beneficial to ICL. We select the demonstration examples by first dividing the whole samples into N clusters and then randomly selecting one sample from each cluster, namely *KmeansRND*. Specifically, we use UniXcoder [48] for vectorization and use the K-means++ algorithm [49] for clustering, where K is set to N that represents the number of demonstration examples. Similar to *Random*, we also investigate the performance of different groups of examples for *KmeansRND* and conduct the selection process three times, resulting in three groups of demonstration examples.

For instance-level demonstration, we need to select examples for each test sample, as illustrated in Section II-B. Following [20], we formulate the selection process as a retrieval problem and compare the performance of different retrieval-based methods including:

1) **BM-25**: BM-25 is a classic sparse retrieval method in the information retrieval field. It has also been widely used in many code intelligence models [50], [51].
2) **SBERT**: SBERT [52] is a popular sentence modeling

method and has been widely used in text retrieval [52], [53]. Specifically, in this paper, we use the version that is further trained on the code-related dataset to obtain code and text representations [54].
3) **UniXcoder**: UniXcoder [48] is a unified cross-modal pre-trained model that is pre-trained with three sequence modeling tasks and two contrastive learning-based tasks. It shows promising performance on zero-shot code-to-code search.
4) **CoCoSoDa**: CoCoSoDa [55] is a state-of-the-art code search model that utilizes contrastive learning for code and text representation learning.

For BM-25, we implement with the gensim package [56] by retrieving samples with the highest similarity from the training set. For dense retrieval methods, we directly use these pre-trained models in the replication packages released by the authors without further tuning. Based on the code/text representations output by the pre-trained models, we select the training samples presenting the highest cosine similarities with the test sample. We also follow the previous work [32] and create a method called *Oracle*, which selects demonstration examples by calculating the similarity between the output of the test sample and the output of all training set examples. The Oracle method is usually regarded as an upper bound of the performance, considering that the output of the test sample is not available in practice. The retrieval process in Oracle is implemented by BM-25, since BM-25 shows the best performance compared with other dense retrieval methods as shown in Table III-V.

To avoid the influence of different orders of demonstration examples, we run each experiment three times with different orders and report the average results on each metric. Besides, we further evaluate the sensitivity of each method to different orders by Coefficient of Variation (CV) [57]. The CV is calculated by σ/μ, where σ is the standard deviation and μ is the mean. A lower CV indicates smaller data variation. It takes the magnitude of data into account and has been widely used to measure the data dispersion in many fields such as economics and software engineering [57], [58].

2) Analysis: We present the experimental results in Table III-V. For each metric, we report the average results over

TABLE V: Experimental results of different demonstration selection methods on Program Synthesis.

Approach	Program Synthesis							
	CB		SM		DM		EM	
	Avg	CV	Avg	CV	Avg	CV	Avg	CV
Task-level Demonstration								
Random	**28.36**	1.30	44.37	0.83	**39.70**	1.33	16.00	1.60
KmeansRND	28.03	1.47	**44.41**	0.54	37.31	1.54	**17.03**	1.06
Instance-level Demonstration								
BM-25	**30.37**	0.91	**46.22**	0.84	40.75	1.06	**18.53**	0.50
SBERT	29.08	0.70	44.91	0.31	39.81	3.01	16.13	2.54
UniXcoder	28.96	0.50	43.93	0.67	37.96	1.12	16.00	3.53
CoCoSoDa	29.42	0.82	44.62	0.70	**40.91**	1.12	16.30	0.86

three random orders and CV which measures their sensitivity to different orders. In Fig. 3, we show the distribution of results with different groups of examples for Random and KmeansRND.

Diversity of examples is beneficial for task-level demonstration. As can be seen in Table III-V and Fig. 3, by comparing the results on Random and KmeansRND, we can find that in most cases improving the diversity of task-level demonstrations can not only improve the average performance of ICL but also reduce the fluctuation brought by different groups of examples. For example, as shown in Table III, comparing the results of code summarization on CSN, the average improvements of KmeansRND over Random are 5.45%, 7.25%, and 6.80% with respect to BLEU-4, ROUGE-L, and METEOR, respectively. Besides, we can also find that the performance of different in-context demonstration examples of Random varies a lot, and improving the diversity of selected examples can reduce this variation in general. For example, as shown in Fig. 3 (a), the gap between the best and worst BLEU-4 score of Random is about 2.5 while that of KmeansRND is only about 0.6. This indicates that improving the diversity of selected demonstration examples is beneficial for building task-level demonstration.

> **Finding 1:** Diversity of examples is helpful for the demonstration selection of ICL. It can help improve overall performance and lead to a more stable prediction regarding different groups of examples.

BM-25 is a simple and effective method for instance-level demonstration. By comparing the results of different instance-level demonstration methods, we can find that the simple BM-25 method can achieve comparable or even better performance than other dense retrieval methods on demonstration selection in ICL. For example, the average EM of BM-25 on Program Synthesis is 18.53, which outperforms three strong dense retrieval methods SBERT, UniXcoder, and CoCoSoDa by 14.88%, 15.81%, and 13.68%, respectively. This result indicates that BM-25 serves as an effective baseline approach and could be taken into account in future studies of demonstration selection for code intelligence tasks.
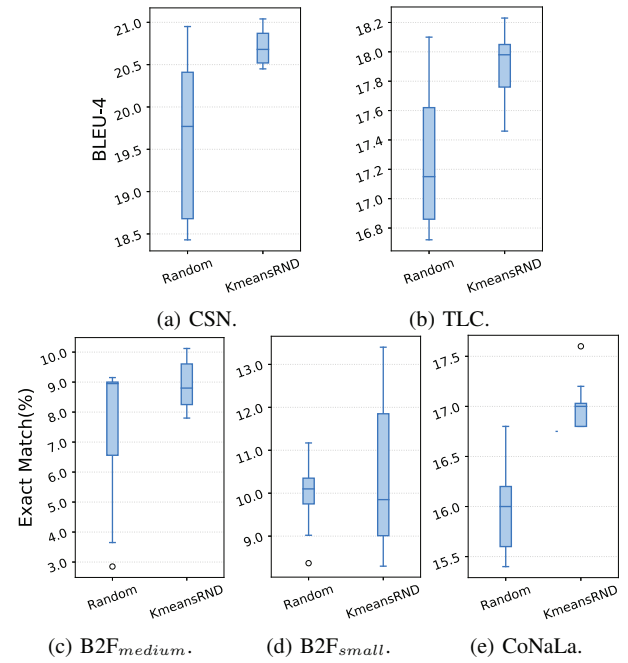

(a) CSN.   (b) TLC.

(c) B2F$_{medium}$.   (d) B2F$_{small}$.   (e) CoNaLa.

Fig. 3: Comparison of the performance distribution of Random and KmeansRND regarding different groups of examples on three tasks.

> **Finding 2:** The retrieval methods for demonstration selection can impact the performance of ICL, among which BM-25 is a simple and effective method.

Instance-level demonstration outperforms task-level demonstration greatly. As shown in Table III-V, we can find that instance-level demonstration can achieve much better performance in all tasks. Specifically, the instance-level selection methods improve the best task-level demonstration's exact match results by at least 141.97% and 193.64% on B2F$_{medium}$ and B2F$_{small}$, respectively. These results indicate that selecting similar demonstration examples specifically for each test sample can benefit ICL in code intelligence tasks a lot.

The task-level demonstration is more sensitive to the order than the instance-level demonstration. By comparing the CV of task-level demonstration and instance-level demonstration, we can find that the performance of instance-level demonstration is generally more stable than task-level demonstration regarding different example orders. Specifically, as shown in Table IV, the CV of BLEU-4 of task-level demonstration KmeansRND to the order is 0.17 and 1.36 on two bug fixing datasets, which is much larger than that of instance-level demonstration methods (e.g., 0.09 and 0.13 for BM-25, respectively). This indicates that selecting examples by similarity is more robust to the changes in the demonstration order and we should carefully arrange the order of

TABLE VI: Experimental results of different demonstration ordering methods.

Approach		Code Summarization (CSN)			Bug Fix (B2F$_{small}$)		Program Synthesis (CoNaLa)			
		BLEU-4	ROUGE-L	METEOR	BLEU-4	EM	CB	SM	DM	EM
Random	Random	20.46	36.71	16.17	**72.40**	9.52	27.72	44.46	37.53	15.53
	Similarity	**21.04**	**37.86**	**16.26**	72.02	**9.93**	**28.47**	**44.87**	37.79	**16.00**
	Reverse Similarity	19.78	33.71	15.64	71.44	9.02	27.62	44.48	**37.96**	15.20
KmeansRND	Random	20.67	**37.64**	15.97	72.29	8.60	26.64	42.97	**37.24**	**16.87**
	Similarity	**20.69**	37.62	16.05	**72.90**	**10.15**	**27.20**	42.97	36.93	16.40
	Reverse Similarity	20.55	37.43	**16.20**	72.05	9.78	27.09	**43.74**	37.19	16.60
BM-25	Random	**22.35**	**38.31**	**17.01**	77.54	30.45	30.37	46.22	40.75	**18.53**
	Similarity	22.23	38.12	**17.01**	**77.76**	**30.95**	**30.83**	**46.41**	**41.33**	17.60
	Reverse Similarity	22.13	38.26	16.91	77.60	29.80	30.01	45.72	39.60	18.20

demonstration examples when using task-level demonstration

> **Finding 3:** Compared with task-level demonstration, instance-level demonstrations can achieve much better performance and are generally more robust to the changes in the demonstration order.

Apart from the above, we can also observe in Table III that the best demonstration selection method BM-25 still has a large gap with the Oracle. This indicates that these retrieval methods may fail to select semantic similar examples and there exists a large space for further improvement concerning the demonstration selection method for code intelligence tasks.

B. RQ2: Demonstration Order

1) Experimental setup: In RQ1, we have found that the order of demonstration examples impacts the performance of ICL on code intelligence tasks, especially on task-level demonstration. Therefore, in this section, we explore how to better arrange the demonstration examples in ICL. Inspired by the finding that the task-level demonstration is more sensitive to the example order than the instance-level demonstration, we suppose that the order of similarities between each demonstration example and test sample plays an important role in ICL.

To verify this, in this RQ, we compare random order with two basic ordering methods, i.e., *Similarity* and *Reverse Similarity*. In the Similarity method, we compare the similarity of each example with the test sample and the example with a higher similarity will be placed closer to the test sample. On the contrary, for the Reverse Similarity method, the demonstration examples will be placed in descending order according to their similarity to the test sample. We experiment with three demonstration selection methods here. As illustrated in RQ1, since the order arrangement is important for task-level demonstration, we use both the Random and KmeansRND for experiments. As for instance-level demonstration, we conduct experiments on BM-25, since it shows the best performance among all the instance-level demonstration selection methods.

2) Analysis: From the results in Table VI, we can find that placing the demonstration examples in ascending order based on their similarity to the test sample performs generally better than the reverse. Specifically, *Similarity* consistently outperforms *Reverse Similarity* on code summarization and

bug fixing by at least 0.45% and 0.21% with respect to BLEU-4 and EM, respectively. By further comparing all the results together, we can observe that similarity achieves the best performance in most cases. Specifically, it achieves the best performance in 62.96% (17/27) metrics and tasks. However, we can also observe that there are some cases in which both *Similarity* and *Reverse Similarity* perform worse than the average results of using random order, indicating that more complex demonstration ordering methods can be explored by the future work.

> **Finding 4:** The different orders of demonstration examples can impact the performance of ICL. Arranging the demonstration examples based on their similarity to the test sample in ascending order can achieve relatively better results in most cases.

C. RQ3: The Number of Demonstration Examples

1) Experimental setup: In this section, we investigate whether the increase in the number of examples will improve the performance of ICL on code intelligence tasks. We vary the number of demonstration examples from 1 to 64. We use BM-25 and *Similarity* as demonstration selection and demonstration ordering methods, respectively, based on the above findings.

2) Analysis: As shown in Fig. 4, we can find that the performance of ICL on all the tasks increases with the number of demonstration examples at first. However, when the number of examples is above 16, the results on different tasks show different trends. For example, for bug fixing, the performance achieves the peak when the number of demonstration examples is 32 and suffers from a significant drop when further increasing the number to 64. As for program synthesis, the performance keeps increasing and tends to be stable when the number exceeds 32. We believe that the different trends are caused by the *truncation problem* [59], [60]. As illustrated in Section III-C, when increasing the number of examples, the length of the whole demonstration will increase and the examples might be cut off to avoid exceeding the length limitation of LLMs. Specifically, for the B2F$_{small}$ dataset, all the examples are complete without cutting off when the number of examples is below 32. However, when the number becomes 32, 2.33% demonstration examples are cut off. When
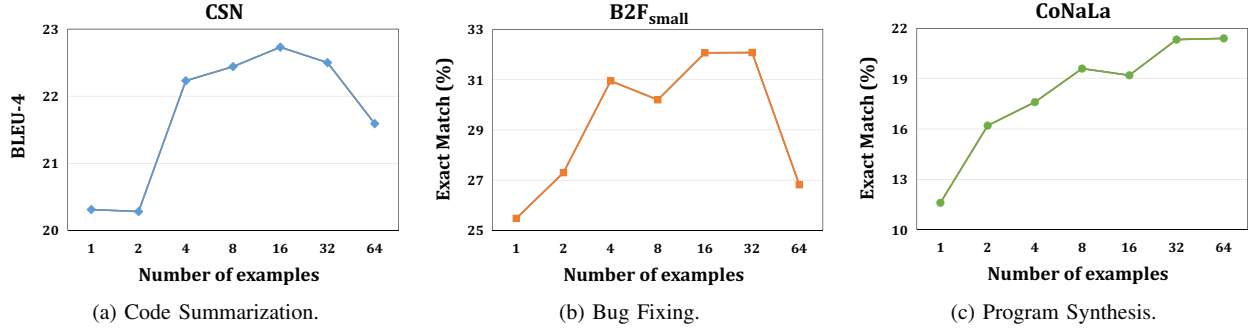
Fig. 4: Experimental results of ICL with different number of demonstration examples.

TABLE VII: Experiments of generalization of findings on GPT3.5 and ChatGPT.

Approach		CB		EM	
Selection	Order	Avg	CV	Avg	CV
GPT-3.5					
Random	Random	26.60	3.01	12.32	4.73
KmeansRND	Random	28.26	1.93	13.60	1.65
UniXcoder	Random	30.06	0.53	13.73	1.13
BM-25	Random	**30.81**	1.05	14.40	1.81
BM-25	Similarity	30.69	0.00	**15.20**	0.00
ChatGPT					
Random	Random	28.17	1.98	11.88	4.24
KmeansRND	Random	28.25	2.31	12.92	1.78
UniXcoder	Random	29.33	1.85	**14.32**	2.87
BM-25	Random	28.95	5.75	13.47	1.82
BM-25	Similarity	**30.03**	0.00	14.20	0.00
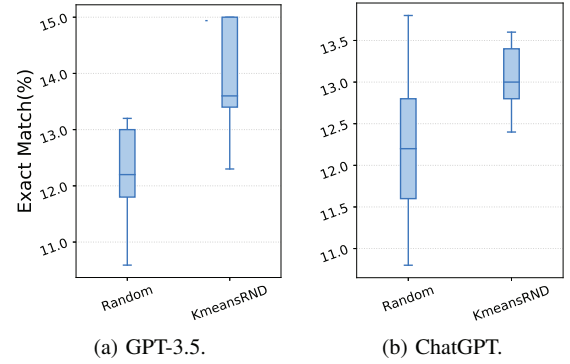
Fig. 5: Comparison of the performance distribution of Random and KmeansRND regarding different groups of examples on GPT-3.5 and ChatGPT.

further increasing the number to 64, the truncation problem happens on over 80% examples and 44.32% characters in those examples are discarded, resulting in a dramatic performance degradation. Since the length of samples in CSN and B2F$_{small}$ datasets is much larger than that of the CoNaLa dataset, i.e., 557, 492, 101 characters per sample for CSN, B2F$_{small}$, and CoNaLa, respectively, the truncation problem does not appear on program synthesis even though the number grows to 64. Therefore, balancing the number of examples and the ensuing truncation problem is important for ICL.

Since the code is generally much longer than natural language [35], the truncation problem is easier to appear in code intelligence tasks. Besides, more examples will also lead to a larger cost of using external API and the inference time [61]. A smaller number of examples may be more appropriate for code intelligence tasks. From the results (Fig. 4), we can also find that the performance with four demonstration examples is good enough, achieving 96.48%, 97.80%, and 94.80% of the best performance on the three tasks with respect to EM, BLEU-4, and CodeBLEU, respectively. Therefore, considering the above trade-off, using four examples in the demonstration is a good choice for code intelligence tasks.

> **Finding 5:** More demonstration examples in the prompt will not always lead to better performance considering the truncation problem. To save costs, it is suggested that four examples are used in the demonstration.

D. RQ4: The Generalization of Findings

1) Experimental setup: In this section, we evaluate the generalization of our findings on different LLMs. Apart from Codex, we experiment on two other LLMs including GPT-3.5 [11] and ChatGPT [26]. To validate the finding 1-4, we experiment with the following combinations of demonstration selection and ordering methods: Random+Random, KmeansRND+Random, UniXcoder+Random, BM-25+Random, and BM-25+Similarity. As for the finding 5 in RQ3, we use BM-25+Similarity as the selection and ordering method and vary the number of demonstration examples from 1 to 128 to validate whether the truncation will lead to performance degradation. Due to the cost limit, we choose the program synthesis task for evaluation.

We also measure how much improvement could our findings bring by comparing the performance of ICL with a carefully-

TABLE VIII: Comparison of different demonstration construction methods on three LLMs.

| Approach		Code Summarization (CSN)			Bug Fix (B2F$_{small}$)		Program Synthesis (CoNaLa)			
		BLEU-4	ROUGE-L	METEOR	BLEU-4	EM	CB	SM	DM	EM
Codex	Zero-shot	1.82	4.27	4.19	34.65	1.43	8.71	9.26	23.81	0.20
	Baseline demonstration	17.37	32.04	13.43	69.07	9.70	27.54	44.56	37.07	14.20
	Carefully-designed demonstration	**22.73**	**39.52**	**17.35**	**77.54**	**32.25**	**32.07**	**48.03**	**42.88**	**21.40**
GPT-3.5	Zero-shot	6.34	15.05	14.08	2.81	0.15	0.06	0.26	0.00	0.20
	Baseline demonstration	14.55	21.53	13.81	62.87	9.15	26.36	36.94	41.67	10.00
	Carefully-designed demonstration	**15.99**	**26.78**	**16.70**	**71.70**	**25.25**	**30.69**	**43.95**	**44.78**	**15.20**
ChatGPT	Zero-shot	3.63	11.40	13.16	2.32	0.05	25.70	37.64	54.44	3.40
	Baseline demonstration	10.76	20.02	14.83	41.57	4.60	27.62	41.83	46.85	9.40
	Carefully-designed demonstration	**11.90**	**23.31**	**16.93**	**53.92**	**18.15**	**30.03**	**45.04**	**44.26**	**14.20**
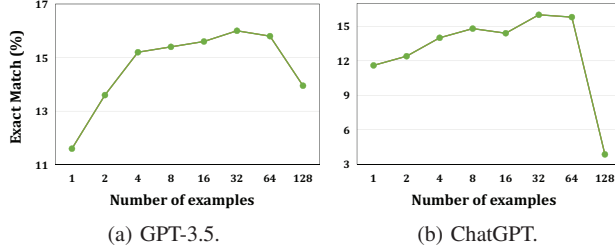
Fig. 6: Experimental results of different number of demonstration examples on GPT-3.5 and ChatGPT.

designed demonstration, ICL with the widely-used demonstration construction method [17], [19], [22], and zero-shot ICL. In the carefully-designed demonstration, we use BM-25 and Similarity as demonstration selection and ordering methods and employ four demonstration examples; while for the widely-used baseline demonstration construction method, we use the settings in previous work [17], [19], [22] and randomly select two demonstration examples from the training set with random order. As for zero-shot ICL, as illustrated in section II-B, no demonstration example is used and the model predicts only based on the instruction.

2) Analysis: We present the average results and CV of GPT-3.5 and ChatGPT in Table VII. In Fig. 5 and Fig. 6, we present the performance distribution of different groups of examples and the impact of the number of examples on these two LLMs, respectively. The comparison of different demonstrations is shown in Table VIII. Due to the space limitation, we only present the performance on EM and CB and the results on other metrics can be found in our replication package. From these results, we can observe that our findings can also be applied to GPT-3.5 and ChatGPT.

As shown in Table VII and Fig. 6, we can observe that KmeansRND+Random not only outperforms Random+Random on the average results, but also has a more stable prediction distribution regarding different groups of examples. Taking GPT-3.5 as an example, KmeansRND+Random improves Random+Random by 6.24% and 10.39% with respect to CB and EM, respectively. This indicates that diversity is also beneficial for the demonstration construction of these two models (**finding 1**). Similarly, by comparing BM-25+Random and UniXcoder+Random, we can also find that

BM-25 can achieve similar performance and even outperforms UniXcoder on GPT-3.5 by 2.50% and 4.88% with respect to CB and EM, respectively. This shows that BM-25 is also a simple and effective demonstration selection method in these two models (**finding 2**). Besides, on GPT-3.5 and ChatGPT, instance-level demonstrations also consistently outperform task-level demonstrations and achieve lower CV to different orders in general. It indicates that selecting demonstration examples by similarity is also beneficial for these two LLMs (**finding 3**). As for the impact of example order, we can also find that BM-25+Similarity consistently improves BM-25+Random on all metrics and LLMs, e.g., improving the average EM by 5.56% and 5.42% on GPT-3.5 and ChatGPT, respectively (**finding 4**). As for the impact of numbers, we can observe similar trends on GPT-3.5 and ChatGPT in Fig. 6, the EM first increases with the number of demonstration examples. As the number further increases to 128, 25.05% examples suffer from the truncation problem, resulting in a sudden degradation (**finding 5**).

Table VIII shows the comparison of different demonstrations. We can also observe that the performance of zero-shot ICL is very poor on all tasks, which indicates the importance of using demonstration examples to guide the LLM to understand the task. Besides, by comparing the performance of the carefully-designed demonstration with the baseline demonstration, we can find that ChatGPT with a carefully-designed demonstration outperforms the baseline demonstration by at least 10.59%, 294.57%, and 51.06% on code summarization, bug fixing, and program synthesis with respect to BLEU-4, EM, and EM, respectively. The results indicate the importance of constructing a good demonstration, and the generalizability of the findings.

V. DISCUSSION

A. Implications of Findings

In this section, we discuss the implications of our work for researchers and developers.

Researchers: Our research demonstrates that the performance of few-shot in-context learning is highly dependent on the design of demonstrations. With well-constructed demonstrations, ICL can achieve much better performance. Our experimental results also show potential research directions in the era of LLM and ICL for the code intelligence community. Specifically:

- As shown in the results of RQ1, current state-of-the-art code retrieval models still have a large gap with the Oracle, indicating that these models fail to select examples with the highest semantic similarities. Therefore, effective code representation models for zero-shot code-to-code search are worth studying. Besides, designing example selection strategies based on the prior knowledge of each task or the properties of source code are also interesting directions that are worth exploring.

- Placing similar examples in the back of all examples leads to relatively better performance than random and reverse placings. However, such improvement is not consistent. Therefore, how to automatically design a better ordering method for code intelligence tasks needs to be further investigated.

- Different from natural language text, the length of a code snippet is often much longer. This limits the number of examples in the prompt and could bring large computation and time costs for LLMs. Therefore, incorporating program slicing and reduction techniques into ICL to reduce the costs is worth investigating.

Developers: In-context learning is a paradigm that allows for learning from a few examples in the prompt without requiring parameter updates. This new approach has also fascinated the language-model-as-a-service community. Our findings indicate that the selection, order, and number of demonstration examples have significant impacts on the performance of ICL for code intelligence tasks. Based on our findings, we conclude the following insights and takeaways for developers to use LLM in their work:

- Including demonstration examples in the prompt, which help the model understand the task and guide the output format.

- Using a retrieval method to select demonstration examples when a labeled training set is available. For the retrieval methods, consider using BM-25 as it is a simple yet effective method.

- Improving the diversity of task-level demonstration examples with clustering to obtain more accurate and stable predictions.

- When arranging the order of demonstration examples, placing similar samples at the end of the list is a good choice in most cases.

- Using as many demonstration examples as possible, but be mindful of the maximum length limitation to avoid truncation issues. To save costs, it is also suggested that four examples are used in the demonstration.

B. Threats to Validity

We identify three main threats to validity of our study:

1) **Potential data leakage.** In this paper, we conduct experiments by using the API of OpenAI Codex, GPT-3.5, and ChatGPT. However, since they are closed-source models, their parameters and training sets are not publicly available, which raises concerns about potential data leakage. Specifically, there is a possibility that the model has

already been trained on the test set and merely memorizes the results instead of predicting them. However, we can observe from our experiments that the model's performance in a zero-shot setting is catastrophic, indicating a low probability of direct memorization of the dataset. Moreover, all experiments in our paper were conducted using these models and we use the relative performance improvement to measure the effectiveness of different demonstration construction strategies. Therefore, the findings of our paper remain convincing.

2) **The selection of tasks.** In this study, we investigate constructions of the demonstration on representative three tasks including code summarization, bug fixing, and program synthesis. These tasks cover different types such as Code → Text, Code+Text → Code, and Text → Code. Hence, we believe the finding of our paper can generalize to a wide arrange of code intelligent tasks. In the future, we plan to conduct experiments on other types of tasks such as Code → Class tasks (e.g., vulnerability detection) and Code → Code tasks (e.g., code translation).

3) **The selection of models.** In this paper, we select three LLMs for experiments. Nonetheless, there are other LLMs available, such as CodeGen [62] and CodeGeeX [63]. In the future, we plan to conduct experiments on a broader range of LLMs to verify the generalizability of our findings.

4) **The selection of languages.** For each task, we select one popular dataset for evaluation. The datasets of three tasks only contain two programming languages, i.e., Java and Python. In the future, we will validate the effectiveness of demonstration construction methods in other languages.

VI. RELATED WORK

A. Pre-trained Models of Code

Recently, with the development of pre-trained techniques, the pre-trained models of code have been widely used and achieved state-of-the-art performance on various software engineering tasks. One such model is CodeBERT [6], which is an encoder-only pre-trained model on six programming languages with two self-supervised tasks. Another model, CodeT5 [9] is an encoder-decoder pre-trained model following the same architecture as T5. CodeGPT [64] is a decoder-only model that pre-trains on programming languages dataset and has the same architecture as GPT-2. PLBART [8] uses denoising sequence-to-sequence pretraining for both program understanding and generation purposes. UniXCoder [48] involves multi-modal contrastive learning and cross-modal generation objective to learn the representation of code fragments.

Apart from these smaller pre-trained models in academic circles, many pre-trained code models with much larger sizes have been proposed in the industry in recent years. Codex [2] is a large code pre-trained model proposed by OpenAI that supports the service of Copilot. In addition to Codex, the models recently released by OpenAI, such as ChatGPT [26] and GPT-4 [23], are also pre-trained on source code data

and demonstrate impressive programming abilities. Alpha-Code [25] is trained for generating code for programming competitions like Codeforces, using 715G data and 41B parameters. CodeGen [62] is a large pre-trained model for multi-turn program synthesis with more than 16B parameters, while CodeGeeX [63] is a recently proposed open-source multilingual code generation model with 13B parameters.

B. In-context Learning

Large language models have revolutionized natural language processing (NLP) in recent years. Based on large pre-training data and model sizes, LLMs show impressive emergent abilities that have not been observed in small models [10]. Brown et al. [11] first show that GPT-3 has the ability to learn from a few examples in the context without parameter update. Liu et al. [20] first explore selecting the closest neighbors as the in-context examples. Recently, Levy et al. [65] propose to improve the diversity of in-context examples and achieve better performance on NLP compositional generalization tasks. Lu et al. [21] find that the order of in-context examples has a large impact on the performance and propose two methods LocalE and GlobalE based on the entropy. Recently, a series of work [15], [66] focus on the complex reasoning tasks and propose chain-of-thought prompt by guiding the model to output its reasoning path.

In addition to NLP, there has been increasing interest in applying in-context learning to code intelligence tasks [17], [19], [22], [47], [67], [68]. For example, Xia et al. [17] evaluate the effectiveness of LLMs on program repair. Nashid et al. [47] propose to use the BM-25 to retrieve similar examples and construct the demonstrations for assert generation and program repair. However, these works mainly focus on the evaluation of LLMs on one or two tasks and do not discuss the construction of in-context demonstrations in-depth. In contrast, our work aims at conducting a systematic study of designing better demonstrations for ICL in code intelligence tasks.

VII. Conclusion and future work

In this paper, we experimentally investigate the impact of different demonstration selection methods, different demonstration ordering methods, and the number of demonstration examples on the performance of in-context learning for code intelligence tasks. Our research demonstrates that a carefully-designed demonstration for ICL outperforms simpler demonstrations a lot. We summarize our findings and provide suggestions to help researchers and developers construct better demonstrations for code intelligence tasks. In the future, we will explore more aspects of source code on the performance of in-context learning such as the quality of the code and the naturalness of the code. Additionally, we will also further verify our findings on other large language models. Our source code and full experimental results are available at https://github.com/shuzhenggao/ICL4code.

Acknowledgment

References

[1] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. ACM, 2022, pp. 39–51.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021.

[3] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020*. Association for Computational Linguistics, 2020, pp. 4998–5007.

[4] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 25–36.

[5] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Association for Computational Linguistics, 2017, pp. 440–450.

[6] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.

[7] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. ijcai.org, 2018, pp. 2269–2275.

[8] W. U. Ahmad, S. Chakraborty, B. Ray, and K. 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, NAACL-HLT 2021, Online, June 6-11, 2021*. Association for Computational Linguistics, 2021, pp. 2655–2668.

[9] Y. Wang, W. Wang, S. R. Joty, and S. 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, EMNLP 2021*. Association for Computational Linguistics, 2021, pp. 8696–8708.

[10] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *Trans. Mach. Learn. Res.*, vol. 2022, 2022.

[11] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[12] D. Driess, F. Xia, M. S. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu *et al.*, "Palm-e: An embodied multimodal language model," *CoRR*, vol. abs/2303.03378, 2023.

[13] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the*

Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers). Association for Computational Linguistics, 2019, pp. 4171–4186.

[14] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, and Z. Sui, "A survey for in-context learning," *arXiv preprint arXiv:2301.00234*, 2022.

[15] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *NeurIPS*, 2022.

[16] Y. Hu, C. Lee, T. Xie, T. Yu, N. A. Smith, and M. Ostendorf, "In-context learning for few-shot dialogue state tracking," in *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*. Association for Computational Linguistics, 2022, pp. 2627–2643.

[17] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1482–1494.

[18] Y. Peng, S. Gao, C. Gao, Y. Huo, and M. R. Lyu, "Domain knowledge matters: Improving prompts with fix templates for repairing python type errors," *CoRR*, vol. abs/2306.01394, 2023.

[19] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs?: An evaluation on quixbugs," in *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*. IEEE, 2022, pp. 69–75.

[20] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for gpt-3?" in *Proceedings of Deep Learning Inside Out: The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures, DeeLIO@ACL 2022, Dublin, Ireland and Online, May 27, 2022*. Association for Computational Linguistics, 2022, pp. 100–114.

[21] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*. Association for Computational Linguistics, 2022, pp. 8086–8098.

[22] J. Y. Khan and G. Uddin, "Automatic code documentation generation using GPT-3," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 174:1–174:6.

[23] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023.

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017, pp. 5998–6008.

[25] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[26] ChatGPT, "Chatgpt," https://chat.openai.com/, 2022.

[27] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Comput. Surv.*, vol. 55, no. 9, pp. 195:1–195:35, 2023.

[28] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 2022, pp. 382–394.

[29] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-efficient transfer learning for NLP," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, vol. 97. PMLR, 2019, pp. 2790–2799.

[30] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[31] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proceedings of the 59th Annual Meeting of the*

Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021. Association for Computational Linguistics, 2021, pp. 4582–4597.

[32] O. Rubin, J. Herzig, and J. Berant, "Learning to retrieve prompts for in-context learning," in *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*. Association for Computational Linguistics, 2022, pp. 2655–2671.

[33] H. Su, J. Kasai, C. H. Wu, W. Shi, T. Wang, J. Xin, R. Zhang, M. Ostendorf, L. Zettlemoyer, N. A. Smith, and T. Yu, "Selective annotation makes language models better few-shot learners," 2023.

[34] S. Gao, H. Zhang, C. Gao, and C. Wang, "Keeping pace with ever-increasing data: Towards continual learning of code intelligence models," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 30–42.

[35] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019.

[36] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, "On the evaluation of neural code summarization," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1597–1608.

[37] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318.

[38] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81.

[39] S. Banerjee and A. Lavie, "METEOR: an automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005*. Association for Computational Linguistics, 2005, pp. 65–72.

[40] S. Gao, C. Gao, Y. He, J. Zeng, L. Nie, X. Xia, and M. R. Lyu, "Code structure-guided transformer for source code summarization," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 23:1–23:32, 2023.

[41] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[42] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 443–455.

[43] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, "Natgen: generative pre-training by "naturalizing" source code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 2022, pp. 18–30.

[44] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 2018, pp. 476–486.

[45] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020.

[46] Z. Cheng, T. Xie, P. Shi, C. Li, R. Nadkarni, Y. Hu, C. Xiong, D. Radev, M. Ostendorf, L. Zettlemoyer, N. A. Smith, and T. Yu, "Binding language models in symbolic languages," 2023.

[47] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2450–2462.

[48] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational*

Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022. Association for Computational Linguistics, 2022, pp. 7212–7225.

[49] D. Arthur and S. Vassilvitskii, "k-means++: the advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007.* SIAM, 2007, pp. 1027–1035.

[50] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020.* ACM, 2020, pp. 1385–1397.

[51] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: Exemplar-based neural comment generation," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020.* IEEE, 2020, pp. 349–360.

[52] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019.* Association for Computational Linguistics, 2019, pp. 3980–3990.

[53] Y. Luan, J. Eisenstein, K. Toutanova, and M. Collins, "Sparse, dense, and attentional representations for text retrieval," *Trans. Assoc. Comput. Linguistics*, vol. 9, pp. 329–345, 2021.

[54] Sentence-transformers, "st-codesearch-distilroberta," https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base, 2021.

[55] E. Shi, Y. Wang, W. Gu, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cocosoda: Effective contrastive learning for code search," pp. 2198–2210, 2023.

[56] Gensim, "Gensim package," https://github.com/RaRe-Technologies/gensim, 2010.

[57] C. E. Brown, *Coefficient of Variation.* Springer Berlin Heidelberg, 1998, pp. 155–157.

[58] M. Wei, N. S. Harzevili, Y. Huang, J. Wang, and S. Wang, "CLEAR: contrastive learning for API recommendation," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 2022, pp. 376–387.

[59] Z. Dai, Z. Yang, Y. Yang, J. G. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers.* Association for Computational Linguistics, 2019, pp. 2978–2988.

[60] A. Bulatov, Y. Kuratov, and M. Burtsev, "Recurrent memory transformer," in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022.

[61] OpenAI-pricing, "Openai-pricing," https://openai.com/pricing, 2022.

[62] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[63] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x," *CoRR*, vol. abs/2303.17568, 2023.

[64] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021.

[65] I. Levy, B. Bogin, and J. Berant, "Diverse demonstrations improve in-context compositional generalization," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023.* Association for Computational Linguistics, 2023, pp. 1401–1422.

[66] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *NeurIPS*, 2022.

[67] T. Ahmed and P. T. Devanbu, "Few-shot training llms for project-specific code-summarization," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022.* ACM, 2022, pp. 177:1–177:5.

[68] G. Poesia, A. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, and S. Gulwani, "Synchromesh: Reliable code generation from pre-trained language models," in *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022.* OpenReview.net, 2022.