Selective Shot Learning for Code Explanation

Paheli Bhattacharya and Rishabh Gupta
Bosch Research and Technology Centre, India
{paheli.bhattacharya, gupta.rishabh}@in.bosch.com

Abstract

Code explanation plays a crucial role in the software engineering domain, aiding developers in grasping code functionality efficiently. Recent work shows that the performance of LLMs for code explanation improves in a few-shot setting, especially when the few-shot examples are selected intelligently. State-of-the-art approaches for such Selective Shot Learning (SSL) include token-based and embedding-based methods (Geng et al., 2024). However, these SSL approaches have been evaluated on proprietary LLMs, without much exploration on open-source Code-LLMs. Additionally, these methods lack consideration for programming language syntax. To bridge these gaps, we present a comparative study and propose a novel SSL method (SSL_{ner}) that utilizes entity information for few-shot example selection. We present several insights and show the effectiveness of SSL_{ner} approach over state-of-the-art methods across two datasets. To the best of our knowledge, this is the first systematic benchmarking of various few-shot examples selection approaches using open-source Code-LLMs for the code explanation task.

1 Introduction

Code understanding and explanation (MacNeil et al., 2023), also known as code summarization (Ahmed and Devanbu, 2022; Iyer et al., 2016) and code comment generation (Hu et al., 2018; Sharma et al., 2022), is an important problem in the domain of software engineering. It involves generating concise and informative explanations for pieces of source code. This provides the developers with a quick understanding of its functionality aiding in code maintenance, search and retrieval (Ye et al., 2020). For programmers new to a particular programming language, code summaries serve as valuable documentation to familiarize them with the new environment efficiently (MacNeil et al., 2023). Automating the task of code documentation through comments and explanations can therefore prove beneficial in many ways.

Large Language Models (LLMs) have proven their efficiency in a variety of NLP tasks. LLMs have shown promising results in several software engineering tasks like code generation (Li et al., 2023; Yin et al., 2023), translation (Huang et al., 2023), test case generation (Schäfer et al., 2023) and code explanation (Geng et al., 2024; Ahmed and Devanbu, 2022; MacNeil et al., 2023; Bhattacharya et al., 2023; Ahmed et al., 2024). While using LLMs for the code explanation task, it has been shown that few-shot prompting achieves better results than zero-shot prompting (Geng et al., 2024; Ahmed et al., 2024). Hence, selecting examples for few-shot learning is an important design criteria. We use the term Selective Shot Learning (SSL) when few-shot examples are chosen intelligently, instead of being random. SSL approaches for code explanation include token-based and embedding-based methods (Geng et al., 2024) without taking into account the language syntax.

Recent work in the area of code explanation have only considered proprietary LLMs like Codex (Geng et al., 2024; MacNeil et al., 2023), Code-davinci-002 (Ahmed and Devanbu, 2022), Text-Davinci-003 (Ahmed et al., 2024), GPT-3 (MacNeil et al., 2023) and GPT-3.5-turbo (Ahmed et al., 2024). However there is a huge gap in proper benchmarking and performance evaluation of several competing, open-source Code-LLMs like CodeLlama (Rozière et al., 2023), StarCoder (Li, 2023) for the code explanation task.

To this end, the contributions of the paper are:

- We explore several open-source Code-LLMs for the task of code explanation, across two datasets covering different levels of descriptions (inline and method-level). We make the dataset and code publicly available at https://github.boschdevcloud.com/HXT2KOR/code-explanation.

Proceedings of the Fourth Workshop on Generation, Evaluation and Metrics (GEM² 2025), pages 151–160
July 31 - August 1, 2025

• We assess the performance of several selective-shot learning approaches, including token-based and embedding-based approaches. Additionally we propose a novel Selective-shot Learning method using NER (SSL_{ner}) that includes code-based entity information for example selection.

• We draw several interesting insights – for e.g., we find that the performance of the medium-sized LLMs (StarCoder 15B) increase more rapidly compared to the larger-sized LLM (CodeLlama 34B) and SSL_{ner} to be the best performing SSL approach and being interpretable.

2 Related Work

The **Code Explanation** (MacNeil et al., 2023) task is a well studied problem in the domain of software engineering (Haiduc et al., 2010; Moreno et al., 2013; Eddy et al., 2013). With the advent of deep learning, methods combining neural architectures (Cai et al., 2020; Ahmad et al., 2020; Sharma et al., 2022) along with software engineering approaches like AST trees (Hu et al., 2018) have been proposed.

Large Language Models have shown exceptional performance in a plethora of NLG tasks (Yang et al., 2023). The zero-shot and few-shot capabilities of these model make them highly adaptable to many NLP tasks. Generic, open-source LLMs like LLama-2 (Touvron et al., 2023), Alpaca (Taori et al., 2023) are trained on open internet datasets. CodeLLMs such as StarCoder (Li, 2023), CodeUp (Jiang and Kim, 2023), CodeLlama (Rozière et al., 2023) and Llama-2-Coder (Manuel Romero, 2023) have been either trained or fine-tuned on code-specific datasets containing source codes covering around 80+ programming languages.

The **Large Language Models**, when used for the **Code explanation** task, has shown some encouraging results. The recent approaches (MacNeil et al., 2023; Geng et al., 2024; Ahmed and Devanbu, 2022; Ahmed et al., 2024) demonstrate that the LLMs performs better in the few-shot setup when good examples of the task are provided. Hence, deciding the relevant examples is an important design criteria while using LLMs for the code explanation task. Existing approaches involve token-based, embedding-based (Geng et al., 2024) and BM-25 along with repository information, data flow graph, AST tree etc. (Ahmed et al., 2024). However, these methods do not explore the efficacy of CodeLLMs.

There has been **systematic evaluations** of transformer models (CodeT5 and CodeBERT) (Mondal et al., 2023) and open source Code-LLMs (Bhattacharya et al., 2023) for code summarization, LLMs on code search (Diera et al., 2023) and non-CodeLLMs like GPT, Bard for code documentation generation (Dvivedi et al., 2024).

This work addresses the lack of systematic benchmarking of selective shot learning (SSL) strategies for code explanation. It analyzes four open-source CodeLLMs across two datasets and three SSL methods, without using auxiliary tools like AST or data-flow graphs (Ahmed et al., 2024).

3 Dataset

In order to perform an extensive evaluation of the performance of the different open source Code-LLMs on the code explanation task, we consider two types of datasets which have different levels of codes and explanations – Inline level and Function level. We describe each of them in detail:

(i) Inline level: This involves explaining particular lines of codes. Inline documentation improves readability and maintainability of a code. We experiment with the CoNaLa: The Code/Natural Language Challenge dataset (Yin et al., 2018). The dataset contains manually curated $(code\ snippet, code\ explanation)$ pairs. The code snippets are in the Python programming language. The code explanation is a natural language description that explains the task *code snippet* is performing. Table 1 shows the statistics of the dataset. There are $1,666$ and 350 samples in the train and test sets respectively. The average length of code snippet and their explanations is approximately 14 tokens.

(ii) Function level: This involves explaining specific functions or methods. We experiment with the TLC dataset (Mu et al., 2023), a widely-used dataset for the code comment generation task. The TLC dataset has additional labels for each data sample that implies the intents of the code – “how to use”, “property”, “why”, “how it is done” and “what”. Since the code snippets in TLC dataset are function level codes, we find in Table 1 that the length of the code snippets are longer than the ones in the CoNaLa dataset. However the length of the explanations is on average 12 tokens which is comparable to CoNaLa. The test data size is 4,236 samples, with a minimum for the “how-to-

Table 1: Statistics of the two datasets – CoNaLa and TLC – experimented within this paper. CoNaLa contains inline level codes written in Python. TLC contains function level codes written in Java. TLC is further subdivided into 5 different subdomains (code intents). CoNaLa contains shorter codes compared to TLC. The average length of the comments are comparable for the two datasets.

Code Level	**Language**	**Dataset**	**Sub-domain**	**# Samples**		**Average length**			
				train	test	train		test	
						Code	Comment	Code	Comment
Inline	Python	CoNaLa	–	1666	350	13.92	14.68	14.35	14.06
Function	Java	TLC	How-to-use	838	37	75.14	12.75	65.41	12.97
			Property	5,016	292	69.96	12.86	73.5	12.59
			Why	5,935	297	82.29	12.47	83.38	12.34
			How-it-is-done	11,478	507	89.5	14.63	89.94	14.32
			What	28,991	2158	87.26	11.8	86.56	11.12

use" intent with 37 samples and a maximum of 2158 samples for the "what" intent.

4 Selective-Shot Learning Approaches

In this section we elaborate the different approaches for selecting relevant demonstrations for the code explanation task. The general pipeline is shown in Figure 1. It is assumed that there is a database containing $(code\ snippet, code\ explanation)$ pairs (referred to as training data) from which relevant examples will be selected. Similarity is computed between the input code snippet (q) and all $code\ snippets$ (d_i) in the database, using the approaches $Selection_{token}$, $Selection_{semantic}$ and SSL_{ner} described next. From each approach, we find the most relevant k code snippets, along with their explanations, and curate a prompt which is then passed on to an LLM to generate the explanation for q.

4.1 Token-based selection

In the token-based selection strategy proposed in (Geng et al., 2024) the query code q and the all code snippets d_i are first preprocessed by removing the keywords defined in the programming languages and converting all the tokens to lower case. The preprocessed q and d_i's are then converted to a list of tokens $tokens_{target}$ and $tokens_{candidate}$ respectively. Then a Jaccard similarity is computed between the two token lists to get the resulting token based similarity.

$Selection_{token} = \frac{|\ tokens_{target} \cap tokens_{candidate}\ |}{|\ tokens_{target} \cup tokens_{candidate}\ |}$.

The value of $Selection_{token}$ ranges from 0 to 1. A larger value of indicates a higher similarity between the query code and the candidate code from the retrieved set. Based on the similarity value, the d_i's are ranked in decreasing order and then the top-k most similar code snippet and their corresponding explanation is added as few-shot demonstrations.

4.2 Embedding-based selection

In the embedding-based approach proposed in (Geng et al., 2024), the query code q and all code snippets d_i in the database are encoded as vectors $\overrightarrow{d_i}$ and $\overrightarrow{q}$ respectively using the CodeBERT embedding model. The $Selection_{semantic}$ score is then the cosine similarity computed between the embeddings $\overrightarrow{d_i}$ and $\overrightarrow{q}$. The value of $Selection_{semantic}$ lies between 0 to 1. A larger value indicates a higher similarity. Based on the similarity value, the d_i's are ranked in decreasing order and then the top-k most similar code snippets and their corresponding explanations are added as few-shot demonstrations.

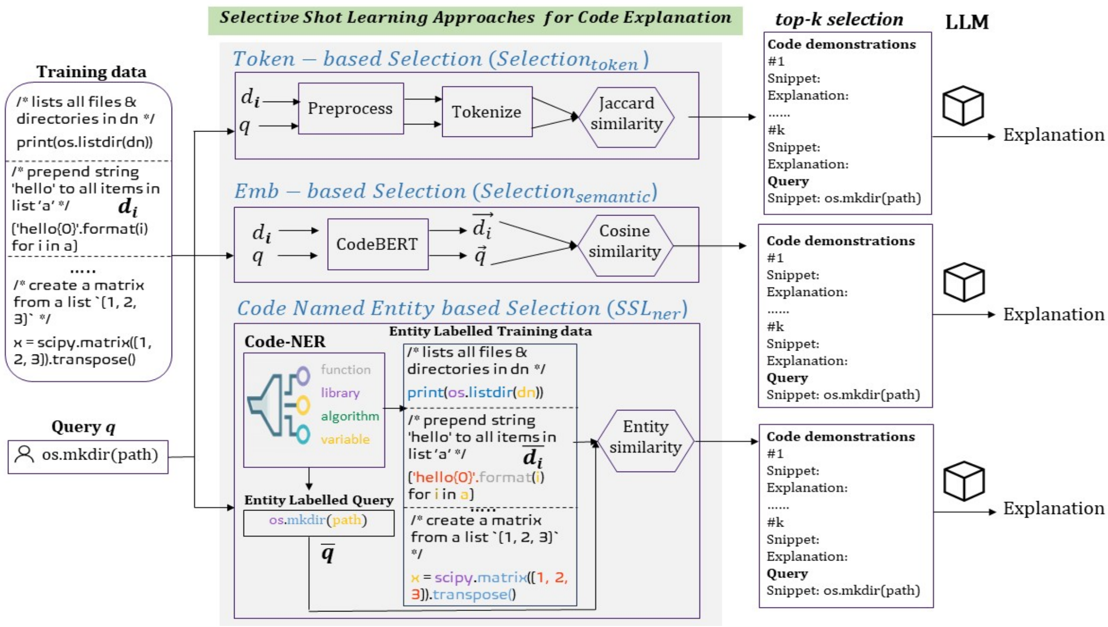
4.3 Code Named Entity based Selection

In this section, we present a novel method, Selective-shot Learning using Named Entity Recognition ($\boldsymbol{SSL_{ner}}$), that utilizes code-based named entities to select examples. It has two submodules Code Entity Extraction and Entity-based similarity, described subsequently.

Code entity extraction – This is the entity extraction module that returns a set of entities E from the programming language domain. We use UniversalNER (Zhou et al., 2023), an LLM that extracts entities from a wide variety of domains including programming. 20 different entities like function, library, data structure, algorithms etc. are supported in the model. For instance, given a code snippet `print(os.listdir(dname))`, this module will label `print` and `listdir` as 'function', `os` as library and `dname` as 'variable'. Figure 1 shows that the training data samples and the query code are passed through the code entity extraction module and each of them are labelled with entity information.

Entity-based similarity – This is the entity similarity module to find how similar are the list of entities which are extracted from the code snippets.

Figure 1: The workflow of the code explanation pipeline using Selective Shot Learning (SSL) approaches. In the input we have a query code snippet q whose explanation needs to be generated and a training database containing $(code\ snippet, code\ explanation)$ pairs from which the few-shot examples need to be selected. The training data samples are ranked according to their similarity with q, where similarity can be computed using either $Selection_{token}$, $Selection_{semantic}$ or SSL_{ner}. From the ranked list, top-k examples are selected and given as a prompt along with q to an LLM which then generates the explanation.



Given two code snippets q and d, the similarity:

$$sim_{ne}(q, d) = \sum_{i=1}^{|E|} w_{e_i} * s_{e_i(q,d)} \tag{1}$$

where $e_i \in E$ is a particular entity type; $s_{e_i}(q, d) = jaccard(e_{i_q}, e_{i_d})$ is the jaccard similarity between e_{i_q}, e_{i_d} (the entities of type e_i in q and d respectively) and w_{e_i} is the weight for an entity type e_i in similarity estimation. We assign $w_{e_i} = 0$ for e_i = 'data type', 'variable' and 'value' because the entities of these types may not play a major role in similarity estimation. For others we set $w_{e_i} = 1$.

To summarize, $\boldsymbol{SSL_{ner}}$ takes the input code snippet q and the training database containing documented code pairs in the form of $(code\ snippet, code\ explanation)$. These pairs are then ranked in decreasing order of similarity values $sim_{ne}(d, q)$ calculated using Eq. 1. The top-k most similar code snippets along with their explanations are selected, appended with the prompt and sent to an LLM to generate the explanation of the input code snippet q.

In the example (Figure 1), given a query code snippet `os.mkdir(path)` and $k = 2$, the similar codes that are likely to get retrieved are `print(os.listdir(dname)` and `r+=[e for e in os.listdir(folder) if e.endswith('.c')]`, since both these code snippets use the os library. The query code snippet `os.mkdir(path)` also uses the same library and hence is more similar to those two code snippets than others (e.g. `x=scipy.matrix([1,2,3]).transpose()`) in the training set. The code samples along with their explanations now forms the demonstrations in the prompt.

5 Experimental Setup

In this section we describe the experimental design choices used in this paper.

Evaluation: We use the BLEU, METEOR and ROUGE-L scores for evaluating the model generated explanations with respect to the ground truth explanations. These are the most widely used metrics for the task (Geng et al., 2024; Hu et al., 2018; Ahmed et al., 2024).

Large Language Models: We evaluate the performance of the different approaches by providing prompts to the following LLMs – Llama-2-Coder-7B, CodeUp-13B-Chat, StarCoder (15.5B) and CodeLlama-34B-Instruct. We use $k = 10$ examples as suggested by previous works (Geng et al., 2024; Ahmed and Devanbu, 2022) for better performance. For the UniversalNER LLM, we set max_new_tokens=64, do_sample=False, temperature=0.1. For all CodeLLMs, we set max_new_tokens = 32, do_sample = False and temperature = 0.7.

For the TLC dataset, there are five intents as described in Section 3. (Geng et al., 2024) uses these intents in the prompt construction. For instance, for a test query code from the intent "how-to-use" they use the prompt: "Describe the usage or the expected set-up of using the method". However, we find that including such intent-specific keywords in the prompt does not affect the performance of the open source code LLMs. We therefore do not include the description of the intents in the prompt.

The zero-shot prompt templates used in our experiments are as follows:

CodeLlama: `[INST] <>You are an expert in Programming. Below is a line of python code that describes a task. Return only one line of summary that appropriately describes the task that the code is performing. You must write only summary without any prefix or suffix explanations. Note: The summary should have minimum 1 words and can have on an average 10 words. <>{code} [/INST]`

Llama2-Coder, StarCoder and CodeUp:

```
#Human: You are a helpful code summarizer. Please describe in simple english the purpose of the following Python code snippet: {code}
#Assistant:
```

6 Results

The empirical results of the code explanation task on the CoNaLa dataset are presented in Table 2. For the five code intents in the TLC dataset the results are given in Tables 3–7. We frame research questions addressing the pivotal points in using LLMs for the task of code explanation and also the effects of different exemplar selection strategies.

RQ1: The effectiveness of open-source CodeLLMs for the task of code explanation using the vanilla In-context learning technique. The first two rows for each open source code LLM (LLama2-Coder, CodeUp, StarCoder and CodeLlama) in Tables 2, 3–7 show the performance of zero-shot and randomly selected examples for few-shot prompting techniques (*few shot (random)*).

Table 2: The performance of the approaches using four LLMs for the code explanation task on the CoNaLa dataset. We report the % improvement of SSL_{ner} over the baseline approaches $Selection_{token}$ and $Selection_{semantic}$.

Model	Approach	BLEU	ROUGE-L	METEOR
Llama2-Coder (7B)	zero shot	0.292	0.298	0.236
	few shot (random)	0.364	0.373	0.323
	$Selection_{token}$	0.393	0.401	0.36
	$Selection_{semantic}$	0.405	0.415	0.379
	SSL_{ner}	**0.408**	**0.419**	**0.386**
CodeUp (13B)	zero shot	0.31	0.35	0.203
	few shot (random)	0.345	0.372	0.291
	$Selection_{token}$	0.382	0.403	0.343
	$Selection_{semantic}$	0.402	0.417	0.368
	SSL_{ner}	**0.412**	**0.424**	**0.384**
StarCoder (15B)	zero shot	0.291	0.33	0.216
	few shot (random)	0.373	0.402	0.335
	$Selection_{token}$	0.411	0.435	0.385
	$Selection_{semantic}$	0.429	0.449	0.407
	SSL_{ner}	**0.435**	**0.451**	**0.416**
CodeLlama (34B)	zero shot	0.354	0.374	0.254
	few shot (random)	0.369	0.38	0.321
	$Selection_{token}$	0.389	0.397	0.357
	$Selection_{semantic}$	0.395	0.403	0.375
	SSL_{ner}	**0.399**	**0.405**	**0.381**

Table 3: The performance of all the approaches using four LLMs for the code explanation task over the **How-to-use** intent in the TLC dataset. We report the % improvement of SSL_{ner} over the baseline approaches $Selection_{token}$ and $Selection_{semantic}$.

Model	Approach	BLEU	ROUGE-L	METEOR
Llama2-Coder (7B)	zero shot	0.186	0.126	0.123
	few shot (random)	0.291	0.275	0.236
	$Selection_{token}$	0.324	0.315	0.291
	$Selection_{semantic}$	0.347	0.34	0.317
	SSL_{ner}	**0.358**	**0.355**	**0.323**
CodeUp (13B)	zero shot	0.187	0.132	0.15
	few shot (random)	0.319	0.302	0.274
	$Selection_{token}$	0.342	0.357	0.336
	$Selection_{semantic}$	0.391	0.381	0.367
	SSL_{ner}	**0.395**	**0.395**	**0.372**
StarCoder (15.5B)	zero shot	0.194	0.138	0.107
	few shot (random)	0.259	0.265	0.216
	$Selection_{token}$	0.365	0.393	0.351
	$Selection_{semantic}$	0.402	0.426	0.371
	SSL_{ner}	**0.411**	**0.431**	**0.378**
CodeLlama (34B)	zero shot	0.198	0.136	0.173
	few shot (random)	0.237	0.229	0.196
	$Selection_{token}$	0.242	0.206	0.263
	$Selection_{semantic}$	0.263	0.219	0.285
	SSL_{ner}	**0.27**	**0.223**	**0.292**

In both the CoNaLa and TLC datasets we observe CodeLlama to perform the best in the zero shot prompting setting. This is because the model is the largest in size (34B) compared to other models Llama2-Coder (7B), CodeUp (13B) and StarCoder (15.5B). Additionally, CodeLlama is further fine-tuned on Llama-2 while CodeUp and StarCoder has been trained for scratch on code data.

Interestingly, for the few shot prompting, we

Table 4: The performance of all the approaches using four LLMs for the code explanation task over the **why** intent in the TLC dataset. We report the % improvement of SSL_{ner} over the baseline approaches $Selection_{token}$ and $Selection_{semantic}$.

Model	Approach	BLEU	ROUGE-L	METEOR
Llama2-Coder (7B)	zero shot	0.201	0.142	0.118
	few shot (random)	0.261	0.221	0.196
	$Selection_{token}$	0.304	0.287	0.264
	$Selection_{semantic}$	0.346	0.318	0.288
	SSL_{ner}	**0.352**	**0.324**	**0.298**
CodeUp (13B)	zero shot	0.212	0.129	0.16
	few shot (random)	0.257	0.231	0.21
	$Selection_{token}$	0.276	0.262	0.244
	$Selection_{semantic}$	0.296	0.289	0.268
	SSL_{ner}	**0.301**	**0.297**	**0.276**
	Gain (%) over $Selection_{token}$	9.06	13.36	13.11
	Gain (%) over $Selection_{semantic}$	1.69	2.77	2.99
StarCoder (15.5B)	zero shot	0.196	0.159	0.127
	few shot (random)	0.278	0.279	0.242
	$Selection_{token}$	0.296	0.313	0.268
	$Selection_{semantic}$	0.315	0.331	0.297
	SSL_{ner}	**0.338**	**0.342**	**0.303**
CodeLlama (34B)	zero shot	0.225	0.186	0.216
	few shot (random)	0.253	0.191	0.238
	$Selection_{token}$	0.313	0.294	0.315
	$Selection_{semantic}$	0.348	0.338	0.343
	SSL_{ner}	**0.361**	**0.344**	**0.35**

Table 5: The performance of all the approaches using four LLMs for the code explanation task over the **property** intent in the TLC dataset. We report the % improvement of SSL_{ner} over the baseline approaches $Selection_{token}$ and $Selection_{semantic}$.

Model	Approach	BLEU	ROUGE-L	METEOR
Llama2-Coder (7B)	zero shot	0.245	0.226	0.197
	few shot (random)	0.323	0.341	0.305
	$Selection_{token}$	0.356	0.362	0.324
	$Selection_{semantic}$	0.391	0.405	0.359
	SSL_{ner}	**0.401**	**0.416**	**0.372**
CodeUp (13B)	zero shot	0.263	0.202	0.22
	few shot (random)	0.429	0.42	0.404
	$Selection_{token}$	0.469	0.491	0.474
	$Selection_{semantic}$	0.528	0.517	0.505
	SSL_{ner}	**0.542**	**0.532**	**0.522**
StarCoder (15.5B)	zero shot	0.269	0.243	0.223
	few shot (random)	0.456	0.476	0.446
	$Selection_{token}$	0.467	0.479	0.474
	$Selection_{semantic}$	0.544	0.524	0.531
	SSL_{ner}	**0.558**	**0.535**	**0.538**
CodeLlama (34B)	zero shot	0.252	0.215	0.254
	few shot (random)	0.3	0.246	0.267
	$Selection_{token}$	0.337	0.328	0.377
	$Selection_{semantic}$	0.376	0.375	0.427
	SSL_{ner}	**0.379**	**0.382**	**0.432**

Table 6: The performance of all the approaches using four LLMs for the code explanation task over the **How-it-is-done** intent in the TLC dataset. We report the % improvement of SSL_{ner} over the baseline approaches $Selection_{token}$ and $Selection_{semantic}$.

Model	Approach	BLEU	ROUGE-L	METEOR
Llama2-Coder (7B)	zero shot	0.187	0.193	0.157
	few shot (random)	0.271	0.267	0.235
	$Selection_{token}$	0.324	0.342	0.318
	$Selection_{semantic}$	0.357	0.372	0.348
	SSL_{ner}	**0.366**	**0.387**	**0.358**
CodeUp (13B)	zero shot	0.204	0.185	0.181
	few shot (random)	0.292	0.297	0.259
	$Selection_{token}$	0.32	0.336	0.294
	$Selection_{semantic}$	0.36	0.366	0.325
	SSL_{ner}	**0.369**	**0.371**	**0.327**
StarCoder (15.5B)	zero shot	0.243	0.193	0.146
	few shot (random)	0.331	0.338	0.327
	$Selection_{token}$	0.411	0.437	0.394
	$Selection_{semantic}$	0.449	0.486	0.427
	SSL_{ner}	**0.463**	**0.491**	**0.436**
CodeLlama (34B)	zero shot	0.262	0.211	0.232
	few shot (random)	0.275	0.241	0.257
	$Selection_{token}$	0.325	0.325	0.309
	$Selection_{semantic}$	0.365	0.357	0.354
	SSL_{ner}	**0.373**	**0.367**	**0.368**

Table 7: The performance of all the approaches using four LLMs for the code explanation task over the **What** intent in the TLC dataset. We report the % improvement of SSL_{ner} over the baseline approaches $Selection_{token}$ and $Selection_{semantic}$.

Model	Approach	BLEU	ROUGE-L	METEOR
Llama2-Coder (7B)	zero shot	0.153	0.162	0.128
	few shot (random)	0.285	0.274	0.242
	$Selection_{token}$	0.334	0.342	0.306
	$Selection_{semantic}$	0.352	0.358	0.317
	SSL_{ner}	**0.358**	**0.363**	**0.325**
CodeUp (13B)	zero shot	0.178	0.162	0.221
	few shot (random)	0.312	0.41	0.368
	$Selection_{token}$	0.352	0.382	0.352
	$Selection_{semantic}$	0.392	0.41	0.373
	SSL_{ner}	**0.407**	**0.425**	**0.381**
StarCoder (15.5B)	zero shot	0.2	0.18	0.131
	few shot (random)	0.291	0.327	0.274
	$Selection_{token}$	0.327	0.395	0.317
	$Selection_{semantic}$	0.365	0.403	0.354
	SSL_{ner}	**0.374**	**0.416**	**0.362**
CodeLlama (34B)	zero shot	0.193	0.183	0.234
	few shot (random)	0.203	0.216	0.27
	$Selection_{token}$	0.28	0.287	0.287
	$Selection_{semantic}$	0.301	0.316	0.335
	SSL_{ner}	**0.318**	**0.322**	**0.341**

observe that the improvements over the zero-shot strategy are much more profound in the smaller sized models (Llama2-Coder, CodeUp and StarCoder) compared to CodeLlama. For instance, one can note from Table 4 that while CodeLlama $(0.225, 0.186, 0.216)$ performs better than StarCoder $(0.196, 0.159, 0.127)$ in the zero shot setting, the latter outperforms the former in the few shot setting, i.e., StarCoder in random few-shot gives $(0.278, 0.279, 0.242)$ and CodeLlama gives $(0.253, 0.191, 0.238)$. This could be attributed to the fact that since CodeLlama is a larger model, in-context examples does not add much to its existing, inherent knowledge. Smaller size models benefit further by providing in-context examples.

RQ2 : Does the performance of open-source Code LLMs improve when provided with relevant in-context examples?

Given that few shot learning improves performance over zero-shot, we now analyse if the quality of the few-shot examples affect the quality of the code explanations generated. As described in Section 4.3, we experiment with three few-shot example selection strategies – $Selection_{token}$, $Selection_{semantic}$ and SSL_{ner}.

For both the datasets, we observe that token and

Table 8: An example showing a code snippet, its ground truth explanation, top 3 examples selected from the baseline method ($Selection_{token}$) and our approach (SSL_{ner}) and the corresponding outputs generated by CodeLlama.

Snippet	Ground Truth	$Selection_{token}$		SSL_{ner}	
		Top 3 Examples	**LLM Output**	**Top 3 Examples**	**LLM Output**
open(filename, 'w'). close()	erase all the contents of a file'filename'	snippet: open('filename', 'w').write ('\\n'.join('%s %s' % x for x in mylist)) comment: create a file 'filename' with each tuple in the list 'mylist' written to a line snippet: hashlib.md5(open('filename.exe', 'rb').read()).hexdigest() comment: calculate the md5 checksum of a file named 'filename.exe' snippet: [w for w in open('file.txt') if not re.search('[aeiou]{2}', w)] comment: Create a list containing words that contain vowel letter followed by the same vowel in file 'file.text'	The code creates a new file with the name 'filename' and closes it.	snippet: f = open('test/test.pdf', 'rb') comment: read a binary file 'test/test.pdf snippet: f = open(os.path.join(__location__, 'bundled-resource.jpg')) comment: open a file 'bundled-resource. jpg' in the same directory as a python script snippet: f = open('words.txt', 'rU') comment: open the file 'words.txt' in 'rU' mode	The code is erasing the contents of a file 'filename'.

embedding-based demonstration selection strategies improve the code explanation generation performance across all the LLMs. For instance in the CoNaLa dataset (Table 2) the BLEU scores for LLama2-Coder, CodeUp, StarCoder and CodeLlama increase by $12\%, 19\%, 17\%$ and 8% respectively when compared with random few shot and SSL_{ner}. Similar to what we observed above, the improvements are more pronounced in the medium sized models, CodeUp and StarCoder, as compared to CodeLlama which is a 34B model. For the TLC dataset we observe this trend for intents "how-to-use", "property" and "what" (Tables 3, 5, 7).

RQ3 : How do the token-based demonstration selection strategies compare?

We now analyse two token based demonstration selection strategies $Selection_{token}$ and SSL_{ner}.

For CoNaLa dataset (Table 2), we find that SSL_{ner} shows a better performance as compared to $Selection_{token}$. For instance, in the BLEU metric the improvements reported are $3.8\%, 7.85\%, 5.84\%$ and 2.57% respectively for Llama2-Coder, CodeUp, StarCoder and CodeLlama. The improvements are statistically significant as measured paired Student's T-test at 95%.

Table 8 shows an example code snippet from the CoNaLa dataset, its ground truth explanation, the top 3 examples selected using $Selection_{token}$ and SSL_{ner} and the corresponding outputs generated by the LLM model CodeLlama. The main intent of the example code snippet is to 'erase' the contents of a file. The explanation generated by the SSL_{ner} example selection strategy is more similar to the ground truth than the one by $Selection_{token}$. The examples selected by SSL_{ner} are more concretely on 'file opening' alone but $Selection_{token}$ selects examples that although have a notion of 'opening the file' but is followed by subsequent, complex actions like calculating the checksum, performing string operations etc. This is likely to confuse the model thereby providing an erroneous explanation.

In the TLC dataset, we find that the improvements of SSL_{ner} over $Selection_{token}$ are more notable. For instance, the gain % achieved by SSL_{ner} over $Selection_{token}$ for the intent "what" (which has the highest number of test samples, 2158, ref. Table 1) using CodeLlama and StarCoder in BLEU, ROUGE and METEOR are $(13\%, 13.5\%, 13.9\%)$ and $(14.6\%, 9.6\%, 11.82\%)$ respectively. These improvements are statistically significant.

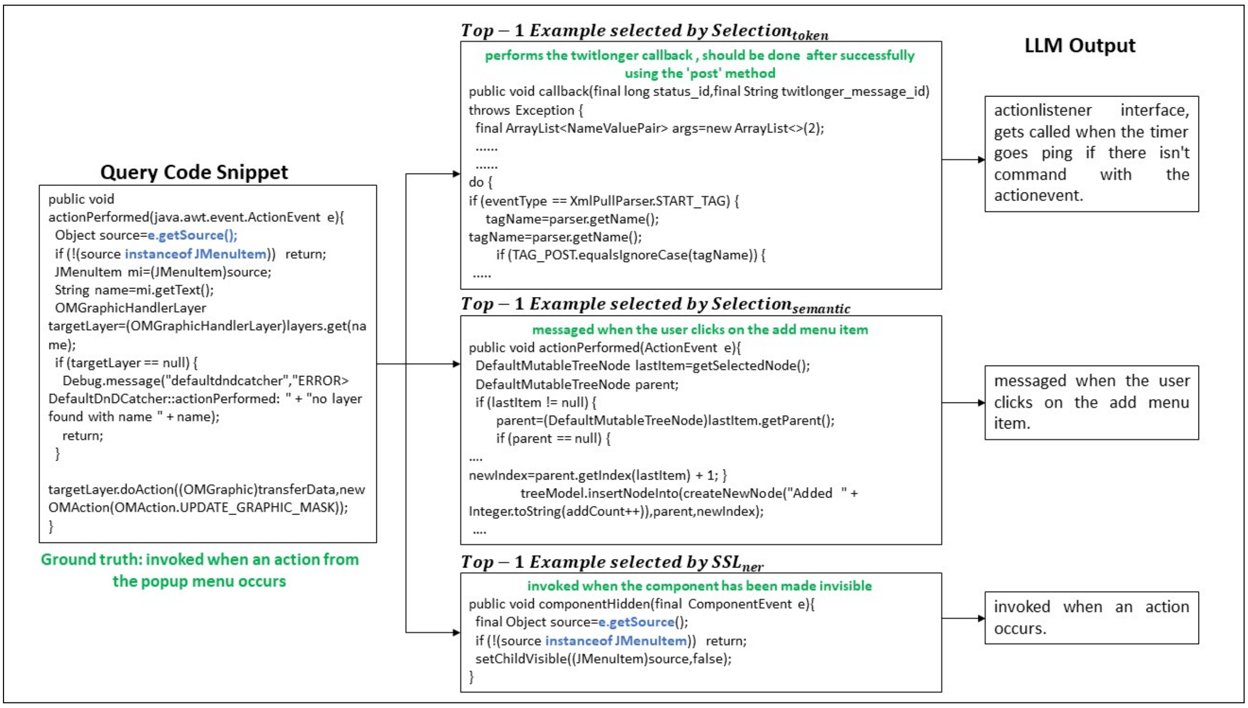
Hence we conclude that SSL_{ner} selects more relevant and consise demonstrations compared to the simpler $Selection_{token}$ approach. The method is interpretable through the matches in different code entities like libraries, functions and classes. The method is also customizable as per end-user needs via the code entity weights. For instance, if the user wants demonstration examples to be more similar in terms of *class* and not much in terms of *functions* and *libraries*, the importance can be adjusted by tuning the weight parameter w_{e_i} suitably, where e_i is a particular entity.

RQ4 : How do the token-based and embedding-based strategies compare?

We perform a comparative study between $Selection_{token}$, SSL_{ner} (both token-based) and $Selection_{semantic}$ (embedding-based). For the CoNaLa dataset, we find the best performance is observed in StarCoder (ref. Table 2). The improvements over the best token-based method SSL_{ner} and $Selection_{semnatic}$ are trivial and is not statistically significant. Similar observations hold for the five intents in the TLC dataset (Tables 3 – 7).

We now look at a qualitative example from the

Figure 2: An example demonstrating the Query Code method, the top 1 demonstration example selected by $Selection_{token}$, $Selection_{semantic}$ and SSL_{ner} along with the LLM (StarCoder) generated output for each method, respectively.



TLC dataset (intent: "use") in Figure 2. Due to the lengthy function-level codes and page limitation, we omit portions of the selected codes in the middle. The query code has the ground truth "invoked when an action from the popup menu occurs". We show the top 1 example selected by each SSL-approach $Selection_{token}$, $Selection_{semantic}$ and SSL_{ner} and the corresponding explanations of the query code generated by StarCoder for each demonstration example.

For $Selecction_{token}$ we find that the explanation generation is not accurate and straight-forward. It is also difficult to understand the points of similarity between the demonstration example and the query code. $Selection_{semantic}$ gives a much better explanation of the query code compared to $Selection_{token}$ as it hints at some user clicks and action occurring thereafter. The reason behind the selection of this example is difficult to interpret as there are no direct links observable. For instance the query code uses methods like `getSource()` and classes like `OMGraphicHandler`. The example from $Selection_{semantic}$ consists of classes like `DefaultMutableTreeNode` and methods like `getRoot()`. For SSL_{ner} we find the example consisting of similar methods `getSource()` and class `JMenuItem`. The explanation generated by the LLM using this demonstration example is hence similar to the ground truth explanation, although it misses the word "popup" .

7 Conclusion and Future Work

In this paper, we perform a comparative study of several open-source Code LLMs, SSL methods and experiment with two datasets having varying levels of explanations for the code explanation task. We perform a thorough analysis of the methods and the performances of the different CodeLLMs that lead to different interesting insights.

Additionally, we introduce a new Selective-shot Learning method SSL_{ner} based on code-based NER . Empirical results suggest SSL_{ner} to be the best token-based demonstration selection strategy while being inherently interpretable and customizable through the code entities.

There are several avenues to extend this work. Possibilities of combining SSL_{ner} with embeddings may be studied. We also plan to experiment with repository level code explanations. Fine-tuning the LLMs by using the relevant examples selected by SSL_{ner} is likely to improve performance. We leave its consideration to future research.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007.

Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5.

Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1004–1004. IEEE Computer Society.

Paheli Bhattacharya, Manojit Chakraborty, Kartheek NSN Palepu, Vikas Pandey, Ishan Dindorkar, Rakesh Rajpurohit, and Rishabh Gupta. 2023. Exploring large language models for code explanation. *arXiv preprint arXiv:2310.16673*.

Ruichu Cai, Zhihao Liang, Boyan Xu, Zijian Li, Yuexing Hao, and Yao Chen. 2020. TAG : Type auxiliary guiding for code comment generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 291–301.

Andor Diera, Abdelhalim Dahou, Lukas Galke, Fabian Karl, Florian Sihler, and Ansgar Scherp. 2023. Gencodesearchnet: A benchmark test suite for evaluating generalization in programming language understanding. In *GenBench: The first workshop on generalisation (benchmarking) in NLP*, page 12.

Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2024. A comparative analysis of large language models for code documentation generation. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, page 65–73, New York, NY, USA. Association for Computing Machinery.

Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 13–22. IEEE.

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *46th International Conference on Software Engineering*.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working conference on reverse engineering*, pages 35–44. IEEE.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, page 200–210. Association for Computing Machinery.

Yufan Huang, Mengnan Qi, Yongqiang Yao, Maoquan Wang, Bin Gu, Colin Clement, and Neel Sundaresan. 2023. Program translation via code distillation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10903–10914.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics 2016*, pages 2073–2083. Association for Computational Linguistics.

Juyong Jiang and Sunghun Kim. 2023. Codeup: A multilingual code generation llama2 model with parameter-efficient instruction-tuning. `https://huggingface.co/deepse`.

Haau-Sing Xiaocheng Li, Mohsen Mesgar, André FT Martins, and Iryna Gurevych. 2023. Python code generation by asking clarification questions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14287–14306.

Raymond Li. 2023. Starcoder: may the source be with you! `https://huggingface.co/bigcode/starcoder`.

Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 931–937.

Manuel Romero. 2023. Llama-2-coder-7b.

Debanjan Mondal, Abhilasha Lodha, Ankita Sahoo, and Beena Kumari. 2023. Understanding code semantics: An evaluation of transformer models in summarization. In *GenBench: The first workshop on generalisation (benchmarking) in NLP*, page 65.

Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International conference on program comprehension (ICPC)*, pages 23–32. IEEE.

Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2023. Developer-intent driven code comment generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 768–780. IEEE.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, and 1 others. 2023. Code llama: Open foundation models for code. https://huggingface.co/codellama.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *Preprint*, arXiv:2302.06527.

Rishab Sharma, Fuxiang Chen, and Fatemeh Fard. 2022. Lamner: code comment generation using character language model and named entity recognition. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 48–59.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html*, 3(6):7.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, and 1 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, and Xia Hu. 2023. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *arXiv preprint arXiv:2304.13712*.

Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging code generation to improve code retrieval and summarization via dual learning. In *Proceedings of The Web Conference 2020*, pages 2309–2319.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, pages 476–486. ACM.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. 2023. Natural language to code generation in interactive data science notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 126–173. Association for Computational Linguistics.

Wenxuan Zhou, Sheng Zhang, Yu Gu, Muhao Chen, and Hoifung Poon. 2023. Universalner: Targeted distillation from large language models for open named entity recognition.