Better Context Makes Better Code Language Models: A Case Study on Function Call Argument Completion

Hengzhi Pei[1]*, **Jinman Zhao**[2], **Leonard Lausen**[2], **Sheng Zha**[2], **George Karypis**[2]

[1] University of Illinois Urbana-Champaign
[2] Amazon Web Services
hpei4@illinois.edu, {jinmaz,lausen,zhasheng,gkarypis}@amazon.com

Abstract

Pretrained code language models have enabled great progress towards program synthesis. However, common approaches only consider in-file local context and thus miss information and constraints imposed by other parts of the codebase and its external dependencies. Existing code completion benchmarks also lack such context. To resolve these restrictions we curate a new dataset of permissively licensed Python packages that includes full projects and their dependencies and provide tools to extract non-local information with the help of program analyzers. We then focus on the task of function call argument completion which requires predicting the arguments to function calls. We show that existing code completion models do not yield good results on our completion task. To better solve this task, we query a program analyzer for information relevant to a given function call, and consider ways to provide the analyzer results to different code completion models during inference and training. Our experiments show that providing access to the function implementation and function usages greatly improves the argument completion performance. Our ablation study provides further insights on how different types of information available from the program analyzer and different ways of incorporating the information affect the model performance.

1 Introduction

Following their counterparts in the natural language domain, we see rapid adoption of language models in the code domain, e.g. Code-GPT (**?**), GPT-J (**?**), Codex (**?**), PLBART (**?**), CodeT5 (**?**), CodeGen (**?**), to name a few. Such *code language models* have demonstrated amazing abilities in achieving new state of the art in various code-related tasks such as clone detection (**????**), code completion (**????**) and code translation (**???**). Notably, pretrained large code language models have been reported to solve programming problems at a high rate (**?**) or at a rate similar to average human contestants (**?**).

Despite their success, these code language models are usually restricted to file-level local code context, and thus miss richer context from project files and libraries. This is in contrast to the more common software development setting where the current piece of code only makes sense under the context and constraints posed by the codebase that it resides in. As a result, models can generate outputs that appear locally plausible but contradict the constraints imposed by the codebase that they operate in.

Machine-learning-for-code datasets and benchmarks also lack such project-level environments for training and evaluating code language models. For example, a common choice for evaluating code completion models is Py150 (**?**) which contains Python source files from Github. However, its train-test split is at file level, making it infeasible to obtain dependency information or incorporate project-level analysis. Similar problems applies to other existing datasets in the code domain (more in Section 5). This motivates us to create a new dataset that better captures what happens in the real-world where software is developed by incorporating external dependencies and spans multiple files. Another missed opportunity is the availability program analyzer smarts that are common in today's integrated development environments (IDEs). Our dataset provides an extensible foundation for the integration of program analyzer information into machine learning for code models.

We collect thousands of permissively licensed Python packages and create a development environment for each of them along with their dependencies. We utilize a language server which can provide language-specific smarts based on project-level analysis. We query the language server to extract environment information relevant to a certain code location. Our setup supports extracting all analyzer information that is available to a developer through an IDE, potentially beneficial to a wide range of code-related tasks.

As a case study, we focus on the task of *function call argument completion*, a special case of code completion where we predict what arguments should be passed through a given function call. This task is well suited to analyze the impact of missing cross-file and cross-project information as understanding function calls requires information that spans file boundaries. A developer who needs to decide what to pass to a function first needs to understand the meaning and the expected usage. They may also benefit from looking at other usage examples of the same function, especially those from within the current working project. In this regard, we build a task-specific dataset by querying the language server to extract function definition, implementation and usage information for each function call instance. We show in Section 4.1

*Work done while interning at Amazon Web Services.

that function call argument completion is more challenging than the average code completion cases and it cannot be made trivial through copying similar occurrences (Table 8).

We conduct extensive experiments on feeding function-call-related information to various pretrained code language models for function call argument completion under different settings, with or without further training the models with this additional type of information. We concatenate different types of analyzer-induced context together with the original local code context as model inputs. We evaluated the performance of several state-of-the-art pretrained code language models, including decoder-only models CodeGPT, CodeGPT-adapt, CodeGen, and encoder-decoder models CodeT5, PLBART, UnixCoder.

We find that providing code language models with analyzer-induced context universally improves their accuracy of call argument prediction. Both the improvements and the best performances are greater with task-specific fine-tuning. We also found that under similar settings, encoder-decoder models perform better than decoder-only models for this task, which is not the case for general code completion (**?**). See more in Section 4.2. The ablation study (Section 4.3) further reveals the unique effects of function implementation and function usage information on model completion performances.

Our main contributions are: *i*) We collect PYENVS, a large number of permissively licensed Python packages along with their isolated development environments which supports querying for project-related information. *ii*) We build CALLARGS, a new dataset for function call argument completion which provides function definition, implementation, and usage information for each function call instance. *iii*) We conduct extensive experiments on feeding analyzer-induced code context as input into various pretrained code language models for function call argument completion, and report various findings. Our code and data will be made available at https://github.com/amazon-research/function-call-argument-completion.

2 Dataset Creation

We describe the formation of PYENVS, a sizable collection of permissively licensed Python projects along with their development environments that support queries to an industry-level program analyzer. We showcase the creation of CALLARGS, a dataset for function call argument completion, which we use in the later experiments. The setup can be used to create analyzer-annotated datasets for other code-related tasks.

2.1 PYENVS for Analyzer-Annotated Code Tasks

We look up the top 5000 most downloaded projects[1] from Python Package Index (PyPI). To ensure that our environments can be redistributed, we only keep the projects where the project itself and all its dependent packages are permissively licensed[2]. For each project we then create a virtual environment[3] including the project code and all its dependencies obtained via the Python Package Installer (`pip`)[4]. For each virtual environment, we locate the source code of the project and its dependencies using the metadata provided by `pip` in the installation directories. In all, we obtain 2814 packages along with their virtual environments.

This setup is reminiscent of a human developer's work environment and enables us and other researchers to build on tools designed for human developers to obtain additional information for machine learning for code models. The Language Server Protocol[5] has been proposed to standardize how such tools and IDEs communicate and is commonly used by IDEs such as VSCode. For this purpose, we choose the Jedi Language Server[6] based on the popular Jedi auto-completion, static analysis and refactoring library for Python as the program analyzer to extract auxiliary information of each project. It enables us to obtain the location of a function's implementation as well as the locations of other points in the project that use the function. We set up the language server with project-specific workspaces based on the virtual environments, enabling the language server to provide the same level of information that is accessible by a developer using an IDE.

2.2 CALLARGS: Function Call Argument Completion Dataset

Based on the above, we further construct a new function call argument completion dataset CALLARGS. We parse each Python file in a project into abstract syntactic trees and extract local information related to each function call, which includes: 1) the function name, 2) the location of function arguments, 3) the location of the function call, and 4) the local context's location of the function call. Here the local context means the function body in which the given function call occurs. In our dataset, we only consider function calls that occur in a function body.

For analyzer-induced information, we query the language server and get the response about the function definition and signature information of the given function call (details in Appendix **??**).

We ignore some function calls which we regard as less meaningful for argument predictions. For example, function calls related to error messages and logging, function calls which the language server fails to find their definitions, or function calls without any arguments. More details about our criteria can be found in Appendix **??**.

To prevent information leakage from project dependencies, we make an effort to ensure both project-level and dependency-level isolation between the training and the test set. We propose an isolation strategy for the tasks that access information from direct dependencies of each project such as our function call argument completion task. We treat the Python standard library built-in packages such as `os` and

[1] https://hugovk.github.io/top-pypi-packages/

[2] MIT, Apache, BSD, CC0, ZPL 2.1, ISCL, PSF (Python Software Foundation License), HPND, or Unlicense.

[3] https://docs.python.org/3/tutorial/venv.html

[4] https://github.com/pypa/pip

[5] https://microsoft.github.io/language-server-protocol/

[6] https://github.com/pappasam/jedi-language-server

`pickle` as public information whose use does not necessitate isolation between training and test set. For third-party dependencies (i.e. other projects), we ensure that if a project is a dependency of a project in the training set or itself is in the training set, then it can not be part of the test set or a dependent of any project in the test set; and vice versa. More details can be found in Appendix **??**.

We randomly sample validation and test set and select training sets to respect the said isolation. We carried out the process with different sample sizes and chose the resulting split where the ratio among training:validation:test is roughly 10:1:1. The statistics is shown in Table 1.

Split	No. of projects	No. of files	No. of tokens	No. of function calls
Train	1578	13790	36.2M	364752
Validation	145	2496	5.2M	42841
Test	145	1701	2.5M	49085

Table 1: Statistics of the CALLARGS dataset.

3 Task & Method

In this section, we formulate the call-argument-completion task, and describe how we incorporate the static analyzer information as function implementation context and function usage context to code language models.

3.1 Task Formulation

Example: `<PREDICT>` denotes the prediction location.

```
1 def _set_arguments(self,arguments,context):
2   positional, named = arguments
3   variables = context.variables
4   args, kwargs = self.arguments.map(
5         <PREDICT>
6   self._set_variables(args, kwargs,
        variables)
7   context.output.trace(lambda: self.
        _trace_log_args_message(variables))
```

Table 2: An argument completion example for `self.arguments.map()`. For unidirectional prediction, the local context consists of the left context (Line 1-4) only; for in-filling prediction, the local context is given as the left local context (Line 1-4) and right local context (Line 6-7).

We define our call argument completion task to be: given the available context (local, project-level, or beyond) of a given function call, predict the complete list of (positional and keyword) arguments to be passed to the function call.

We treat code as sequences of code tokens. We start with the base case where only the in-file local context around the function call location is available. Assume a code-token sequence $X = [x_1, x_2, ..., x_n]$ where $X_f = [x_j, x_{j+1}, ...x_{l-1}]$ is a function-call occurrence and $X_a = [x_l, x_{l+1}, ...x_r]$ is the list of arguments passed to the function call. In our task, we restrict X to be a Python function that contains at least one function call. We refer to the tokens before the target arguments $X_L = [x_i]_{i<l}$ as the left local context, and the tokens after the target arguments $X_R = [x_i]_{i>r}$ as the right local context of the function call X_f.

We consider two variations of the task. In *unidirectional prediction*, we model $P(X_a \mid X_L)$. This is a widely adopted scenario for code completion (e.g. **????**), which simulates the case that a developer is writing code from the beginning to end, where only the local context up to the prediction point is available. In *in-filling prediction*, we model $P(X_a \mid X_L, X_R)$ which simulates the case that a developer is editing an existing piece of source code, where both the left and right context is present. This setting is similar to cloze tests (**?**) which aim to predict the token for the blank with the context of the blank. Our particular setting is more difficult, as the model needs to continuously generate more than one tokens. Table 2 shows an example for the unidirectional prediction and the in-filling prediction.

For both cases, we use a language model to generate predictions for the call arguments. For unidirectional prediction, a decoder-only model $P(X_a \mid X_L) = \prod_{i=l}^{r} p(x_i \mid x_{<i})$ can be used. For both unidirectional and in-filling prediction, an encoder-decoder model $P(X_a \mid X_L, X_R) = \prod_{i=l}^{r} p(x_i \mid x_{<i}, X_L, X_R)$ where X_L and X_R (if available) are passed into the encoder.

3.2 Static Analyzer Information

With the presence of additional information from the static analyzer, we incorporate it as additional context A into the models: $P(X_a|X_L, A)$ for unidirectional prediction and $P(X_a|X_L, X_R, A)$ for in-filling prediction. We use as context the function implementation information and function usage information for function call argument completion.

Function implementation context Imp is the Python function definition of the given function call X_f. It reveals the formats, the constraints, and the intention of the current function call. We retrieve this information with the function definition location from the language server.

Function usage context $\{U_i\}$ collects local contexts surrounding the calls of the same function within the project. Specifically, we include the in-file usages that occur before the prediction location and the in-project usages that occur in a different file. The usage context provides project-specific examples that helps the model to better induce the usage for the current function call. We retrieve this context by grouping the function call instances that share the same definition.

Some function calls, especially those defined in the Python standard libraries, can appear many times within the codebase of a project. Therefore, we design a similarity-based ranking criteria and only select top usages to provide to a model. For a target function call and a usage u of the same call, we calculate the similarity as $|S_l \cap S_u|/|S_l|$ where S_l is the set of local call left context tokens and S_u is the set of the left context tokens for u.

3.3 Incorporating Methods

We concatenate analyzer-induced code context with the local context as the model input, which leverages the power of the pretrained language models to implicitly understand and extract information from each part. Table 3 describes the input templates for the decoder-only models and the encoder-decoder models. We set a length budget for each piece of information so that the total input does not exceed model capacity. If a context exceeds the allocated length, we drop the function implementation and the right local context from the right, and drop the usage context and the left local context from the left to preserve the most relevant information.

We consider the case both with or without task-specific fine-tuning. We call the former setting *concatenating during inference* (CDI). In this setting, the model is never exposed to such auxiliary information. Since auxiliary information in our case can be efficiently retrieved by the static analyzer, we further consider fine-tuning a model with augmented inputs. We hypothesize that this helps a model better understand the relationship between each piece of information.

4 Experiment

In this section, we design experiments to answer the following research questions (RQs).

RQ1 *How do general code completion models perform on function call argument completion?*
We conjecture that general code completion models without project-specific context do not work well on the tasks where external context is critical. To this regard, we test several pretrained code language models on our CALLARGS dataset.

RQ2 *To what extent are the analyzer-induced information helpful for pretrained language models to do function call argument completion?*
To explore this point, we conduct experiments under two settings. We first test if a general code completion model would perform better on our CALLARGS dataset under the CDI setting. Then, we test for unidirectional and in-filling prediction on our CALLARGS dataset after task-specific fine-tuning.

RQ3 *What are the roles and contributions of different types of analyzer-induced information?*
Specifically, we study the impact of function implementation context and function usage context on our CALLARGS dataset. We conduct an ablation study to break down the performance gain from using function implementation context and function usage context. We further explore how the choice of the number and the length of usage context affect model performance.

4.1 RQ1

To answer RQ1, we evaluate various decoder-only models pretrained on Python source code: CodeGPT (**?**), CodeGPT-adapted (**?**) and CodeGen (**?**). Since the models are pretrained using different tokenization strategies and different pretraining datasets, we fine-tune them using general code completion, aka next-token prediction at *all* tokens, over CALLARGS to reduce the impact from data and domain shift. Specifically, we follow the preprocessing step in Py150 (**?**) to standardize the inputs, split the files from CALLARGS training set into code fragments of equal lengths of 1024 tokens, and use the standard causal language modeling loss. Project-level information is not presented during this process.

We train for 10 epochs with a batch size of 32 and a learning rate of 5e-5 using AdamW optimizer (**?**). We apply early stopping when validation set perplexity does not improve for 3 epochs, and chose the checkpoint with the best validation set perplexity. We evaluate the token-level accuracy for general code completion on the files from CALLARGS test set. For each argument completion instance, we ask the model to generate arguments from the left local context only (unidirectional prediction). We use beam search of size 5 until a matching parenthesis is generated. We measure the exact match accuracy (EM) and Levensthein edit similarity (EditSim) between the ground truths and the model completions.

The results are shown in Table 4. We find that, with only local context, although general code completion models can achieve good token-level accuracy (Token-level Acc 72 – 79), they do not perform well on call argument completion (EM 36 – 45). This suggests that call arguments are more difficult to predict than general locations (cf. **?**). One possible reason is that general code completion on average involves easier prediction locations such as boiler-plate code and code idioms.

4.2 RQ2

We evaluate the performance of function call argument completion with the presence of analyzer-induced contexts. When incorporating those contexts, we fill in the respective input slots as described in Section 3.3.

Concatenating during inference. We directly concatenate analyzer-induced context as input to the same models described in Section 4.1. We allocate at least a quarter of the total length budget for analyzer-induced context. We use no more than 3 function usages for each instance unless otherwise specified.The results are shown in Table 4.

We find that both function implementation (w/ implementation) and function usages (w/ usages) universally improve the EM and EditSim across all the models tested, indicating that pretrained code language models benefit from auxiliary contexts even without exposure to them during training. The gains from function usages is much greater (average EM improvements 9.27 vs 1.63), suggesting that if presented only at inference time, similar contexts help code completion more (cf. similar observations from **?**).

Task-specific fine-tuning. Next, we fine-tune different pretrained code language models specifically for call argument completion, with or without the presence of auxiliary contexts. For decoder-only models, we use CodeGPT (**?**), UnixCoder (**?**) and CodeGen (**?**). For encoder-decoder models, we use CodeT5 (**?**), PLBART (**?**) and UnixCoder [7] (**?**).

[7] UnixCoder is pretrained with both denoising objectives and unidirectional language modeling so it can be used as both an encoder-decoder model and a decoder-only model.

Type	Input format
Decoder-only	<s> $[Imp]$ </s> $[U_m]$ </s> ... </s> $[U_1]$ </s> $[X_L]$
Encoder-Decoder	<s> $[X_L]$ <PREDICT> $[X_R]$ </s> $[Imp]$ </s> $[U_1]$ </s> ... </s> $[U_m]$ </s>

Table 3: The input formats for decoder-only and encoder-decoder models. <s>, </s>, and <PREDICT> are special tokens. <PREDICT> suggests the location to fill in with the call arguments.

Model (Token-level Acc)	Context	EM	EditSim
CodeGPT (72.18)	local context	36.29	63.50
	w/ implementation	37.40	64.75
	w/ usages	44.99	73.15
CodeGPT-adapted (72.59)	local context	37.24	64.76
	w/ implementation	38.25	65.98
	w/ usages	46.58	74.36
UnixCoder-base (75.88)	local context	38.45	66.11
	w/ implementation	40.04	67.85
	w/ usages	47.93	75.40
CodeGen-MONO (78.73)	local context	43.45	69.69
	w/ implementation	46.26	72.46
	w/ usages	52.99	78.52

Table 4: The call-argument completion performance of several general code completion models on CALLARGS. The token-level accuracy is for general code completion.

We conduct experiments for both the unidirectional and in-filling prediction. For the latter, the length budget of the right local context is one-third of the length budget of the left local context. The length budget for function implementation and the average length of each function usage are set as one-eighth of the total input length. We train our models for 10 epochs with a batch size of 64 and a learning rate of 2e-5. We use early stopping based on the perplexity over the validation set. The results are shown in Table 5.

Comparing the unidirectional results from decoder-only models (Table 5, top section) to those in Table 4, we find that task-specific fine-tuning EM and EditSim are on average 4.49 and 5.52 higher across the models using local context only. The presence of function implementations and function usages further greatly improves the argument completion performance compared to using local context only, with an average 13.28 gain for EM and an average 8.76 gain for EditSim across all models and settings. The best results are achieved by CodeT5-base with 62.73 EM and 84.33 EditSim for unidirectional prediction, and by CodeT5-base with 69.28 EM and 88.08 EditSim for in-filling prediction.

Compared to unidirectional prediction, the in-filling prediction metrics are on average 6.83 and 4.57 higher for EM and EditSim using the same model and the same source of information. We conjecture that this is because the right context reveal the use of the function call result, or because it provides more information for the model to relate to previously seen similar patterns.

For models with similar numbers of trainable parameters, the results from the encoder-decoder models are usually better than the decoder-only models when using the same contexts. For example, for unidirectional prediction, the encoder-decoder version of UnixCoder is better than its decoder-only version in both metrics. This suggests that the encoder-decoder architecture can be a powerful design for code auto-completion, probably thanks to its better ability at leveraging the input contexts. The results also suggest that the pretraining objectives are important for model's performance. For example, CodeT5-small (60M) pretrained with mask span prediction is better than PLBART-base (139M) pretrained without it, despite that the former has fewer parameters.

4.3 RQ3

We conduct an ablation study on the impact of function implementation context and function usage context. We use CodeT5 to study how different types of contexts, and more closely, how the use of usage contexts, would influence the argument completion performance. We choose CodeT5 because its pretraining tasks align well with our task, and as evidenced by our results in the previous subsection, CodeT5 achieves the best results across similar model sizes. The total input length is 512 unless otherwise specified.

Effect of implementation and usage information. Table 6 shows the completion results with different auxiliary contexts. We find that both function implementation and function usage information are beneficial for call argument completion. Adding usage information leads to higher performance gain.

We also report in Table 6 Surface-level Positional Matching (SPM*), which checks the rate where the predicted arguments can match the parameters in the function definition. We see that function implementation is more helpful in improving SPM*, suggesting the importance of accessing function definition in getting the number of arguments and the keyword prefixes right.

Effect of the number and length of function usage contexts. We vary the number and the length budget of function usages used. The results are shown in Table 7. We find that longer context helps as enlarging the model input length from 512 to 1024 improves the argument completion performance. Using more function usage information does not bring significant improvements when the total input length is fixed. This may be because the coverage of exact or similar usages would saturate as the number of usages increases as we can see in Appendix **??**. A better way to leverage more usage information is a meaningful future direction.

Usage copying. One concern is if the models are simply copying the arguments from other usages. Therefore, we evaluate the performance of copying the top usage if the similarity is above a certain threshold. Otherwise, the model

Task (model type)	Model (# of trainable parameters)	Context	EM	EditSim	Input length
Unidirectional (decoder-only)	CodeGPT (124M)	local context	41.74	70.01	512
		+implementation&usages	54.19	78.88	924
	CodeGPT-adapted (124M)	local context	41.65	70.05	512
		+implementation&usages	54.20	79.03	924
	UnixCoder-base (126M)	local context	42.52	71.33	512
		+implementation&usages	58.22	81.46	924
	CodeGen-MONO (355M)	local context	47.49	74.74	512
		+implementation&usages	62.55	83.71	924
(encoder-decoder)	CodeT5-small (60M)	local context	43.65	71.89	512
		+implementation&usages	59.16	82.34	1024
	CodeT5-base (223M)	local context	47.16	74.44	512
		+implementation&usages	62.73	84.33	1024
	PLBART-base (139M)	local context	38.96	68.17	512
		+implementation&usages	51.26	76.77	1024
	UnixCoder-base (126M)	local context	46.33	73.29	512
		+implementation&usages	60.53	82.85	924
In-filling (decoder-only)	CodeT5-small (60M)	local context	51.63	77.60	512
		+implementation&usages	62.47	85.07	1024
	CodeT5-base (223M)	local context	56.59	80.44	512
		+implementation&usages	69.28	88.08	1024
	PLBART-base (139M)	local context	46.68	73.96	512
		+implementation&usages	57.25	80.94	1024
	UnixCoder-base (126M)	local context	54.31	78.53	512
		+implementation&usages	66.20	86.05	924

Table 5: Performance of different models with task-specific fine-tuning on CALLARGS. Unidirectional results are grouped by model types: decoder-only (top) and encoder-decoder (middle). In-filling results are from encoder-decoder models (bottom).

Task	Context	EM	EditSim	SPM*
Unidirectional	local context	47.16	74.44	89.22
	+implementation	52.66	78.66	97.91
	+usages	58.95	81.99	94.76
	+implementation&usages	61.59	83.73	98.64
In-filling	local context	56.59	80.44	91.8
	+implementation	60.22	82.97	98.27
	+usages	65.57	85.86	95.88
	+implementation&usages	67.59	87.10	98.77

Table 6: Performance of CodeT5-base with different auxiliary contexts.

output is used. We set the threshold using the validation set. In this experiment, the model is fine-tuned with implementation context but not usage context. We also check whether concatenating the usage information at inference directly (CDI) would give similar performances. The result is shown in Table 8.

We find that the threshold copying indeed improves the prediction, which confirms the existence and merit of exact matches. However, the model can better leverage additional patterns and relations (CDI) than simple copying (threshold). On the other hand, using function usage information during the model training (+implementation&usages) brings the most performance gain. It indicates the nontrivial ability of our best performing models to attend usage examples and compose appropriate arguments from them.

Task	EM	EditSim	Input length	Usages
Unidirectional	61.59	83.73	512	(3, 64)
	62.73	84.33	1024	(3, 128)
	63.00	84.46	1024	(6, 64)
	62.87	84.46	1024	(8, 64)
In-filling	67.59	87.10	512	(3, 64)
	69.28	88.08	1024	(3, 128)
	69.26	88.02	1024	(6, 64)
	69.00	87.92	1024	(8, 64)

Table 7: Performance of CodeT5-base when using different numbers and lengths of usage contexts. "Usages" column indicates the number of function usages used and the average length budget for each usage.

5 Related Work

Datasets. The lack of cross-file and cross-project (e.g. dependencies) information is a general issue in current evaluation datasets for code. In terms of code completion, common choices are Py150 (**?**) for Python and Github Java Corpus (**?**) for Java. Both datasets are constructed at file level, where source files are isolated from their project and dependencies and no consideration of project separation is taken in constructing training and test sets. **?** constructed a code completion dataset from CodeNet (**?**), which contains coding problems and solutions from online judge websites and also lacks project context. **?** presented a real-world Python method generation task based on CodeSearchNet (**?**) but the

Task	Context	EM	EditSim
Unidirectional	+implementation	52.66	78.66
	w/ usages (threshold)	56.26	80.54
	w/ usages (CDI)	57.61	81.12
	+implementation&usages	61.59	83.73
In-filling	+implementation	60.22	82.97
	w/ usages (threshold)	62.55	84.40
	w/ usages (CDI)	64.24	85.16
	+implementation&usages	67.59	87.10

Table 8: Comparing using function usage information during training and during inference for CodeT5-base.

auxiliary information they extract still comes from within a local file. **?** constructed a completion dataset based on top Python repositories on GitHub and released the URLs for these repositories. However, those repositories are not write-protected and can change over time. Besides, setting up the dependency environments at scale for further analysis is not easy. Both make their dataset difficult to reproduce. In the contrast, we release the code and the dependencies for the projects to ensure reproducibility. Apart from code completion, datasets for other code tasks such as Cloze test (e.g. **?**), code refinement (e.g. **???**), and generating code from text descriptions (e.g. **???**), are often small and mostly without project-level code context. Beyond-local information is beneficial for programmers to solve programming tasks in real-world settings. The lack of such information in the current dataset would restrict the progress into high-level semantic understanding and reasoning in the code domain.

Code language models. Encouraged by the success of pretrained language models in natural language processing (**????**) and the promise of naturalness in code (**??**), we have seen rising adaptations of language models for code. For example, CuBERT (**?**) and CodeBERT (**?**) are pretrained based on masked language modeling. GPT-C (**?**) and CodeGPT (**?**) are both pretrained based on unidirectional language modeling. PLBART (**?**) and CodeT5 (**?**) are pretrained encoder-decoder structures which adopts denoising objectives and can support code understanding and code generation. UnixCoder (**?**) combines the above three pretraining objectives for a unified pretrained model.

Code completion. Code completion is an essential feature for modern IDEs and an important topic for code intelligence. In recent years, deep neural networks (**?????**), especially pretrained language models (**??**) become the mainstream solution to this task. Still, incorporating additional information proved beneficial. One popular choice is abstract syntax tree, e.g. **???**. However, **?** suggested that pretrained code language models may have already encoded the syntax. Other proposals seek to use data flow graph, control graph, and various graph relations, e.g. **??**. However, information is still restricted from a single file. We instead try to enhance the model with out-of-file information, similar to what is accessible in a development environment.

For project-level analyzer induced information, **?** described a way to use a static analyzer to refine completion candidates from neural methods. **?** considered leveraging the project-wise contexts via embeddings for better function call completion performance. Other than code completion, project-level information has been utilized for methods name prediction (**?**) and generating code from text descriptions (**?**). However, none of them tested their approaches with pretrained code language models. In terms of incorporating additional context through concatenation, **?** reported improvements from prioritize certain parts of in-file context. Recently, **?** proposed to enhance code language models by concatenating similar code fragments retrieved by a neural network. Despite the general similarity, we 1) use a simple lightweight way to retrieve auxiliary information instead of training a heavy retriever; 2) do not restrict ourselves on similar code fragments and show that dissimilar code fragments (function implementation) can be helpful; 3) explore task-specific fine-tuning with retrieved information for better completion.

6 Conclusion

We curated PyEnvs, a collection of permissively licensed Python packages along with isolated development environments. Upon that, we built a function call argument completion dataset CallArgs containing analyzer-induced information for each function call. We experimented feeding auxiliary information as additional input context to various pretrained code language models for call argument completion during training and inference. Results show that access to the function implementation and function usages universally improves the model performances. We further provide insights on the effect of different types of models and different types of additional information on this task. In the future, we can use PyEnvs to construct new datasets for other code-related tasks to further study the benefits from cross-file and cross-project information.

Minus voluptatem tenetur at inventore odio iusto explicabo autem, iure repellendus saepe, officiis nihil aliquam debitis dolor minima suscipit et aperiam, dolor quia odio fugiat molestiae ea laudantium ipsam expedita aperiam.Sequi libero accusamus sapiente, quas ut dolores debitis perspiciatis non voluptatibus, quasi debitis eos adipisci nemo sit praesentium?Dolorem laudantium obcaecati numquam saepe perspiciatis voluptatem veritatis, quo molestiae fuga corrupti officia quidem eaque assumenda?Porro odit corporis doloribus aut recusandae delectus quaerat eligendi, amet officia ab fugiat eaque sit totam pariatur minus dignissimos.Corporis vel ab, nesciunt accusantium quibusdam ratione consequatur asperiores, earum alias aliquid repellat iusto quia nulla.Nesciunt voluptatibus perspiciatis, quod nulla amet ipsum neque vitae necessitatibus perferendis, nam quos molestias voluptates dignissimos ex doloremque.Nulla impedit optio exercitationem, repudiandae architecto molestias et sunt, similique fugit quam iure iste?Impedit fugit inventore laudantium laborum tempora possimus perspiciatis eligendi incidunt, temporibus similique laudantium consectetur soluta dignissimos necessitatibus modi distinctio, nesciunt magnam in eius?Praesentium molestias quo in soluta, tempora fugit accusantium eius ali-

quam, eos vero laboriosam adipisci ducimus recusandae sit in aperiam amet voluptatum, iusto doloremque dolores nihil quod reprehenderit corporis dolorum illo vel.