ReACC: A Retrieval-Augmented Code Completion Framework

Shuai Lu[1], **Nan Duan**[1], **Hojae Han**[2*], **Daya Guo**[3*],
Seung-won Hwang[2], **Alexey Svyatkovskiy**[4]
[1]Microsoft Research Asia [2]Seoul National University
[3]Sun Yat-sen University [4]Microsoft Devdiv
{shuailu,nanduan,alsvyatk}@microsoft.com
{stovecat,seungwonh}@snu.ac.kr
guody5@mail2.sysu.edu.cn

Abstract

Code completion, which aims to predict the following code token(s) according to the code context, can improve the productivity of software development. Recent work has proved that statistical language modeling with transformers can greatly improve the performance in the code completion task via learning from large-scale source code datasets. However, current approaches focus only on code context within the file or project, i.e. internal context. Our distinction is utilizing "external" context, inspired by human behaviors of copying from the related code snippets when writing code. Specifically, we propose a retrieval-augmented code completion framework, leveraging both lexical copying and referring to code with similar semantics by retrieval. We adopt a stage-wise training approach that combines a source code retriever and an auto-regressive language model for programming language. We evaluate our approach in the code completion task in Python and Java programming languages, achieving a state-of-the-art performance on CodeXGLUE benchmark.

1 Introduction

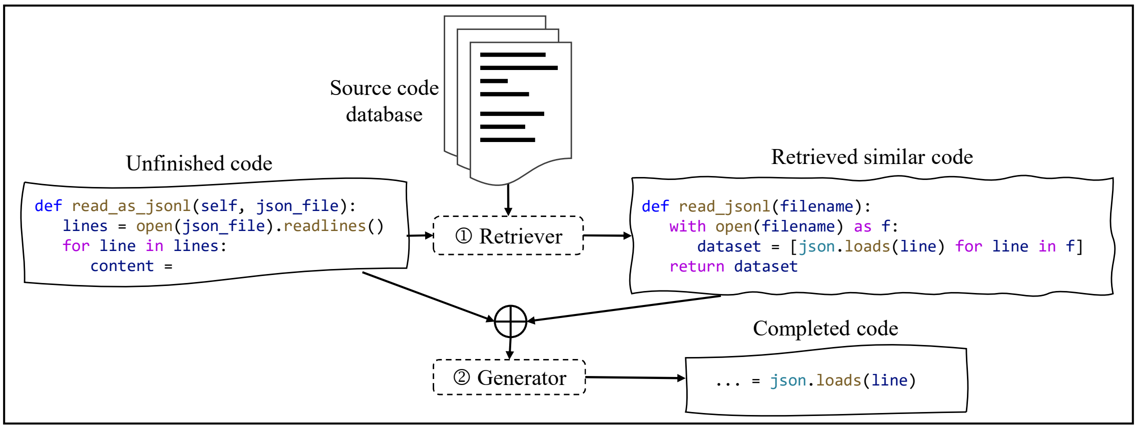
With the growth of software engineering field, large-scale source code corpus gives a chance to train language models in code domain (Hindle et al., 2016; Tu et al., 2014). And benefiting from the large transformer models (Vaswani et al., 2017) and pre-training techniques (Devlin et al., 2018; Radford et al., 2018), a rapid progress has been made in many code-related tasks like code search (Feng et al., 2020; Guo et al., 2020), code summarization (Clement et al., 2020; Ahmad et al., 2020), bug fixing (Mashhadi and Hemmati, 2021; Drain et al., 2021) and code completion (Svyatkovskiy et al., 2020; Liu et al., 2020; Kim et al., 2021; Clement et al., 2020).

Code completion is considered as an essential feature towards efficient software development in modern Integrated Development Environments (IDEs). The task is formulated by predicting the following code token(s) based on the code context. Traditionally, code completion requires real-time program analysis and recommends type-correct code tokens (Tu et al., 2014). Recently, statistical language models trained on large-scale source code data have shown high accuracy in the code completion task. Primitive approaches take the given context only (Liu et al., 2016; Karampatsis et al., 2020), some methods use richer information, e.g., adding code token types (Liu et al., 2020), abstract syntax tree (AST) structures (Li et al., 2018; Kim et al., 2021), or extended hierarchical context (Clement et al., 2021). However, one key limitation of existing methods is the scope of information they utilize; all the information is bounded in the given input file. This is unnatural from human perspective, as studies demonstrate that programmers tend to reuse an existing code snippet by copying part of code with or without minor modifications to accelerate software development (Roy and Cordy, 2008; Baker, 2007), leading a software repository usually containing 7-23% cloned codes (Svajlenko and Roy, 2015).

Motivated by this phenomenon, in this paper, we argue the utility of extending the information scope beyond the input file, i.e., into a large codebase. We conjecture that using codes with similar semantics as auxiliary information are beneficial to predict the following code tokens. Therefore, we propose ReACC – a **Re**trieval-**A**ugmented **C**ode **C**ompletion framework (See Figure 1). The code completion task under our framework can be re-formulated by, given a source code corpus for search and an unfinished code snippet to complete, using the unfinished code as a query to retrieve similar code snippets from search corpus, and predicting the following code tokens by reusing the

[*] Work done during internship at Microsoft.

Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics
Volume 1: Long Papers, pages 6227 - 6240
May 22-27, 2022 ©2022 Association for Computational Linguistics



```
def read_as_jsonl(self, json_file):
    lines = open(json_file).readlines()
    for line in lines:
        content =
```

```
def read_jsonl(filename):
    with open(filename) as f:
        dataset = [json.loads(line) for line in f]
    return dataset
```

```
... = json.loads(line)
```

Figure 1: An illustration of ReACC framework. Given an unfinished code snippet to complete, ReACC first retrieves the similar code from source code database. Then the similar code is concatenated with the unfinished code, the completed code will be generated based on them.

retrieved code. ReACC consists of two core components: (1) a dual-encoder model served as the code-to-code search **retriever** (2) an auto-regressive language model served as the code completion **generator**. ReACC adopts the stage-wise training strategy which is widely used in other tasks like open-domain question answering (Karpukhin et al., 2020; Izacard and Grave, 2021), natural language to code generation (Hashimoto et al., 2018; Parvez et al., 2021), etc.

The simplest technique for retrieving code is to build a sparse vector retriever like TF-IDF or BM25 (Robertson and Zaragoza, 2009) which are both based on keyword matching algorithms. The sparse retriever can capture lexical information and is sensitive to the names of code identifiers. The dense retriever, on the contrary, can capture syntactic and semantic information by mapping a code snippet to a dense vector. In the code completion task, the code retriever is expected to comprehend the source code's intent in order to retrieve the semantically similar codes. On the other hand, considering programmers are prone to copy-and-paste existing code, the retriever should evaluate lexical similarity as well. To that end, we adopt the hybrid retriever (Karpukhin et al., 2020; Ma et al., 2021), which combines results of dense and sparse retriever. We employ a dual-encoder model architecture as the dense retriever since the cross-encoder model has a high computational complexity. To achieve a better understanding ability, we initialize our dense retriever with GraphCodeBERT (Guo et al., 2020), which is a pre-trained BERT-based programming language understanding model. Then we continue pre-training the retriever by contrastive learning to enhance sentence embedding. As the labeled data containing similar code pairs is rare, we utilize various transformations to generate programs with similar functionality for data augmentation.

We implement the generator with a decoder-only transformer model. To incorporate the external information from retrieved similar code, we concatenate the obtained code and code context as input. The generator is initialized by CodeGPT-adapted (Lu et al., 2021) which is a domain-adaptation model from GPT-2 (Radford et al., 2018) pre-trained on code corpus.

We evaluate our ReACC framework on two benchmark datasets – CodeXGLUE (Lu et al., 2021) and CodeNet (Puri et al., 2021), in Python and Java programming languages. ReACC achieves a state-of-the-art performance on both datasets. The experimental results demonstrate that external source code retrieved by our retriever is useful for auto-completing the partial code.

To summarize, our main contributions are:

- We propose a retrieval-augmented method to assist the code auto-completion task. [1]
- To adapt to the code completion scenario, where the retrieval query is an unfinished code snippet, we propose the partial code-to-code search task and create datasets for evaluation.
- We adopt semantic-preserving transformations for data augmentation to pre-train the code retrieval model.

[1] Our codes are available at https://github.com/celbree/ReACC

2 Related Work

2.1 Code completion

Code completion is an essential task for code intelligence. Hindle et al. (2016) are the first to use language model for code completion by N-gram technique. Deep neural networks (Liu et al., 2016; Alon et al., 2020; Karampatsis et al., 2020) and pretraining approaches (Liu et al., 2020; Svyatkovskiy et al., 2020) are later frequently utilized to accomplish this. Besides considering source code as code token sequences, some research focuses on completing an abstract syntax tree (AST) by anticipating the next node in the flattened tree (Li et al., 2018; Kim et al., 2021). Guo et al. (2021) complete codes by generating sketches, i.e. code snippets with "holes". Svyatkovskiy et al. (2021) and Clement et al. (2021), on the other hand, investigate ways to improve the efficiency and long-range modeling in the code completion task, respectively.

All of these works employ previously written code context as inputs, along with AST structural information or token types. But none of them has attempted to leverage existing external code as auxiliary information.

2.2 Retrieval on code intelligence

Contrastive learning on code Inspired by the great success of contrastive learning in other domains (Wu et al., 2018; Reimers and Gurevych, 2019; Fang et al., 2020; Chen et al., 2020; He et al., 2020; Radford et al., 2021; Gao et al., 2021), researchers have deployed this technique to source code for better code fragment understanding. Jain et al. (2020) and Bui et al. (2021) propose ContraCode and Corder, respectively. Both models use the self-supervised contrastive learning framework and generate code snippets as data augmentations via compiler-based semantic-preserving transformations. Their models have shown the effectiveness of contrastive learning in code clone detection, code search and code summarization tasks. SynCoBERT (Wang et al., 2022) and UniXcoder (Guo et al., 2022) are both pre-training models that utilize multi-modal data, including code, comment, and AST, for better code fragment representation through contrastive learning.

Retrieval for code-related tasks Many code intelligence tasks benefit from information retrieval (Xia et al., 2017). A common scenario for information retrieval in code domain is code search with natural language description as a query (Arwan et al., 2015; Gu et al., 2018; Cambronero et al., 2019). As for other code intelligence tasks, Hayati et al. (2018) propose an action subtree retrieval method called ReCode for generating general-purpose code. Hashimoto et al. (2018) propose a retrieve-and-edit framework for code autocompletion and code generation. Luan et al. (2019) propose Aroma, which utilizes code-to-code structural search and intersecting candidate code snippets to recommend relevant code given another code snippet as a query. Both Wei et al. (2020) and Li et al. (2021) leverage the retrieve-and-edit/refine framework to improve model's performance in code summarization. Parvez et al. (2021) propose REDCODER, using a dense retriever trained on paired NL-code pairs to retrieve relevant comments or codes as a supplement for code summarization or code generation tasks.

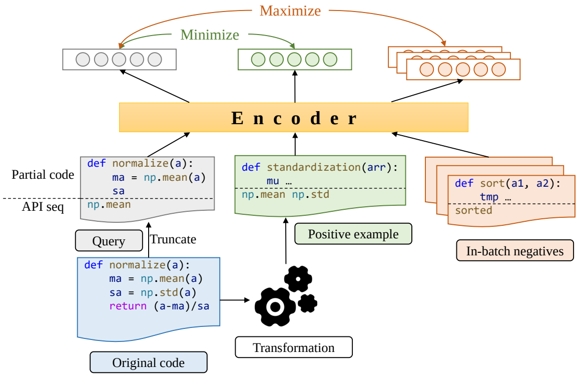
In most circumstances where a dense retriever is utilized, a natural language comment is treated as a query to retrieve code. In the code completion scenario, however, we focus on using code as query, particularly partial code, which is a more difficult task since there are few labeled data with semantically similar code pairs and in partial code search, semantics in query is incomplete.

3 Approach

We first introduce the formulation of retrieval-augmented code completion task. Then we give detailed descriptions on the retriever and generator in ReACC. We show how we continue pre-training GraphCodeBERT (Guo et al., 2020) with contrastive learning on code and how we address the problem that there is no labeled data for positive instances of similar programs in section 3.2. In section 3.3 we talk about the way to aggregate retrieved code and code context in the generator.

3.1 Task Formulation

Assume that we have a source code database containing a large collection of software repositories, which consist of D source code files, $f_1, f_2, ..., f_D$. Following the Dense Passage Retriever (DPR) model (Karpukhin et al., 2020), we split each of the files into code fragments of equal lengths as the basic retrieval units. Such splitting not only leads to a better retrieval results as stated by Karpukhin et al. (2020), but also supports extreme long code files where each part of a file represents differ-



```
def normalize(a):
    ma = np.mean(a)
    sa
np.mean
```

```
def standardization(arr):
    mu …
np.mean np.std
```

```
def sort(a1, a2):
    tmp …
sorted
```

```
def normalize(a):
    ma = np.mean(a)
    sa = np.std(a)
    return (a-ma)/sa
```

Figure 2: Illustration on the training process of the retriever in our proposed framework ReACC.

ent semantics. Thus we get M code fragments as the retrieval database $C = \{c_1, c_2, ..., c_M\}$. Let $X = \{x_1, x_2, ..., x_k\}$ be the unfinished code written previously, a retriever $\mathbf{R} : (X, C) \rightarrow C$ retrieves the most similar code fragment c_s in C. The generator $\mathbf{G}$ predicts the following code token(s) $Y = \{x_{k+1}, ..., x_{k+n}\}$, where $n = 1$ in the token-level code completion task, based on context and retrieved code. Formally, $P(Y) = \prod_{i=1}^{n} P(x_{k+i}|c_s, x_{1:k+i-1})$.

3.2 Retriever

The retrieval module in ReACC is expected to retrieve semantically equivalent code given an incomplete code. We adopt the hybrid retriever (Karpukhin et al., 2020; Ma et al., 2021) framework by combining scores of sparse and dense retriever. The sparse retriever we use is BM25 (Robertson and Zaragoza, 2009) based on the implementation of ElasticSearch[2]. As a term-based retrieval method, BM25 considers each code fragment as a code token sequence and employs bag-of-words representations. The matching score computed by BM25 indicts lexical similarity between the query and document. As for the dense retriever, it maps each code fragment to a d-dimension dense vector. We construct it in this paper based on the DPR model (Karpukhin et al., 2020). Figure 2 illustrates the training process of the dense retriever of ReACC. In the following, we will walk through it in detail.

Dense Retriever Our dense retriever consists of two bidirectional transformer-based encoders E_C and E_Q. E_C encodes each code fragment in the retrieval database C and builds indexes for them. The query is encoded by E_Q. We take the representation of `[CLS]` token as output and the similarity is computed by $sim(q, c) = E_C(c)^T E_Q(q)$. Since both E_C and E_Q take source code as inputs with the only difference being whether they are partial or not, the dual encoders share weights in ReACC. At the training stage, following DPR (Karpukhin et al., 2020), we adopt in-batch negatives to calculate the contrastive loss by InfoNCE (Oord et al., 2018):

$$\begin{aligned} &L(q, c^+, c_1^-, c_2^-, ..., c_m^-) \\ &= -\,log \frac{e^{sim(q,c^+)}}{e^{sim(q,c^+)} + \sum_{i=1}^{m} e^{sim(q,c_i^-)}} \end{aligned} \tag{1}$$

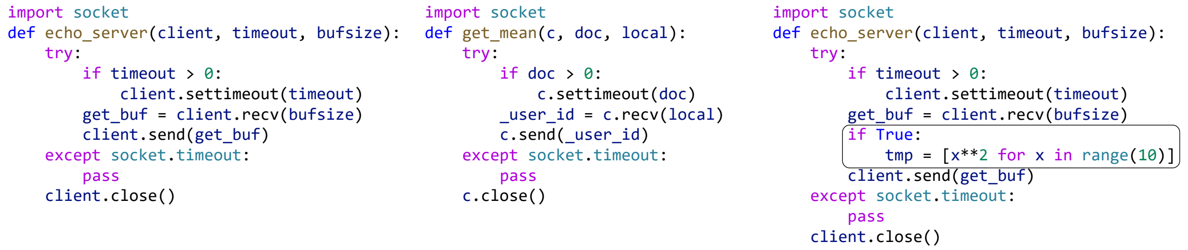
However, unlike DPR, we don't employ "hard" negatives which are retrieved from BM25. Because programmers tend to copy tokens directly, a code with distinct semantics but substantial lexical similarity can help with code completion.

Data Augmentation The purpose of contrastive learning of the dense retriever in ReACC is to learn a representation of code fragments that keeps codes with similar or equivalent semantics close and dissimilar codes far apart. It requires numerous positive and negative code pairs. However, it is difficult to identify similar programs based on an unlabeled code corpus, e.g., certain widely used datasets (Allamanis and Sutton, 2013; Raychev et al., 2016; Husain et al., 2019) mined from GitHub repositories.

Searching semantically equivalent code requires extra code compilation and execution costs (Massalin, 1987; Churchill et al., 2019), which is unrealistic in a large database. Instead of searching, an alternative way is to create code snippets with same functionalities for data augmentation. To do so, we apply several semantic-preserving transformations to the original source code to construct a set of variants. There exists several attempts to apply such transformation to code (Jain et al., 2020; Rabin et al., 2021; Bui et al., 2021). In this paper, we mainly adopt identifier renaming and dead code (unreachable or unused code) insertion. Figure 3 shows an example of performing such transformations to a Python code.

- **Identifier renaming** is a method of renaming an identifier with another. We only rename variable and method names as other identifiers cannot be changed arbitrarily like built-in types or API calls. Different from previous works, we preserve

[2] https://github.com/elastic/elasticsearch



```python
import socket
def echo_server(client, timeout, bufsize):
    try:
        if timeout > 0:
            client.settimeout(timeout)
        get_buf = client.recv(bufsize)
        client.send(get_buf)
    except socket.timeout:
        pass
    client.close()
```

original python code

```python
import socket
def get_mean(c, doc, local):
    try:
        if doc > 0:
            c.settimeout(doc)
        _user_id = c.recv(local)
        c.send(_user_id)
    except socket.timeout:
        pass
    c.close()
```

After renaming all variables

```python
import socket
def echo_server(client, timeout, bufsize):
    try:
        if timeout > 0:
            client.settimeout(timeout)
        get_buf = client.recv(bufsize)
        if True:
            tmp = [x**2 for x in range(10)]
        client.send(get_buf)
    except socket.timeout:
        pass
    client.close()
```

After inserting dead code

Figure 3: An example of applying semantic-preserving transformations to Python code.

part of the lexical information while modifying the names at the same time based on the consideration that identifier names typically convey the meanings for humans and lexical similarity contributes a lot for retrieving (It is verified in section 4.4). To do so, we mask all the identifiers in a program and leverage GraphCodeBERT (Guo et al., 2020) to predict each identifier like in the masked language model task. The top-10 predictions (excluding the original identifier) are selected as the candidate set for renaming.

- **Dead code insertion** is to insert a dead code into a code fragment at a proper location. Dead code is a code snippet which can never be reached (Xi, 1999) or is reachable but whose result can never be used in any other computation (Debray et al., 2000). In software engineering, dead code insertion is one of the most common techniques for code obfuscation (You and Yim, 2010), whose goal is to modify a code to make it hard to understand but remain its functionality, which is similar to our goal. We first randomly select variable names which don't appear in this program and then use them to form a statement from a predefined set of dead code (See Appendix A for details), such as assignment, method invocations, looping statement, conditional statement and so on. We traverse the AST and identify all the statements. Then we choose a statement at random and insert the dead code after it, leading a new subtree in the AST.

Input Format We integrate both the code token sequence and the API usage sequence as inputs. API usage sequence is highly related to the functionality of a code snippet (Gu et al., 2016; Hu et al., 2018). To improve the code representation, we extract the API sequence and append it to the source code token sequence. Finally, we use a random truncation of the original code as the query and the entire created program as the positive example during training to address the problem on how to retrieve based on incomplete semantics.

3.3 Generator

The output of retriever is the retrieved code c_s. Considering c_s is queried by code context x while our target is the following code of x, so we propose *fragment alignment* – using the next fragment c'_s of c_s in the same file (we have split each file into code fragments for retrieval as discussed in Section 3.1) for completing the next fragment of x. Thus, the input sequence for the generator is the concatenation of c'_s and x: $x' = c'_s \oplus x$.

The generator module in ReACC supports any model architecture that can perform code completion task. In our experiments, we adopt CodeGPT-adapted (Lu et al., 2021), which is a decoder-only transformer model pre-trained on Python and Java datasets from CodeSearchNet (Husain et al., 2019) via casual language model. CodeGPT-adapted has shown promising results in the code completion task in CodeXGLUE benchmark (Lu et al., 2021) on two widely used code completion datasets.

4 Experiments: Code Clone Detection

In order to evaluate the effectiveness of the code-to-code retrieval module in ReACC, we perform code clone detection task which aims to retrieve semantic equivalent programs. In this section, we describe how we create the test dataset for this task and how we evaluate the performance of ReACC's retriever.

4.1 Dataset

CodeNet (Puri et al., 2021) dataset consists of a large collection of programs which are derived from online judge websites. We respectively create a code clone detection evaluation dataset from CodeNet in Python and Java with zero-shot setting.

Dataset	Language	Task	Train	Valid	Test	Desc.
CodeNet (Puri et al., 2021)	Python	Clone	-	-	15,594	Solutions for 2,072 problems
	Java	Clone	-	-	14,426	Solutions for 1,599 problems
	Python	Completion	2,636,118	32,984	10,000	For line-level completion
CodeXGLUE (Lu et al., 2021)	Python	Completion	95,000	5,000	50,000 / 10,000	Use PY150
	Python†	Completion	95,000	5,000	- / 20,000	Applying eWASH
	Java	Completion	12,934	7,176	8,268 / 3,000	Use JavaCorpus

Table 1: Dataset statistics. The two numbers in **Test** of CodeXGLUE denote the examples for token-level and line-level code completion, respectively. † is a newly created test set, see the text for details.

We collect code solutions for thousands problems and solutions for the same problem are considered as semantically equivalence. The data statistics are shown in Table 1.

Retrieval Training Set The dense retriever in ReACC is pre-trained on CodeSearchNet dataset (Husain et al., 2019), a large-scale source code corpus extracted from GitHub repositories. We employ 1.6M Java methods and 1.2M Python functions from it.

4.2 Baseline Methods

CodeBERT (Feng et al., 2020) is a pre-trained model for programming language, which is trained on NL-PL pairs from CodeSearchNet dataset in six programming languages.

GraphCodeBERT (Guo et al., 2020) is also pre-trained on CodeSearchNet NL-PL pairs and considers the inherent structure of code i.e. data flow.

4.3 Experiment Setup

The retrieval encoder is initialized with GraphCodeBERT. It is continual pre-trained with both masked language model objective and contrastive learning. We use in-batch negatives with a batch size of 256. With a learning rate of 5e-5, We train the retriever for Python and Java for 30 epochs each.

We implement the code clone detection experiment in the *partial search* way, which is ideally adapted to code completion scenarios as it accepts a partial program as a query while maintaining the same goal.

4.4 Results

Table 2 shows the results in the zero-shot code clone detection task on CodeNet dataset, with the *partial search* setting. Models are measured by MAP@K (Mean Average Precision at K), which is the evaluation metric in the CodeXGLUE clone detection task, and precision at 1, as we only care about the most similar code for code completion. From the comparison with other transformer-based encoders, we can see CodeBERT and GraphCodeBERT can hardly retrieve equivalent code. While our model significantly outperforms them, which indicts our model is capable of retrieving the semantically equivalent code even when the query's semantics is incomplete.

We also find that BM25 performs splendidly in this task, which is quite different from the performance on other tasks like open-domian QA (Karpukhin et al., 2020), code summarization (Parvez et al., 2021), etc. The findings suggest that semantically related codes are likely to be lexically similar, which leads lexical similar to contribute more for retrieval, making code-to-code search easier than text-to-code or question-to-passage search using the term-based retrieval method.

5 Experiments: Code Completion

In this section, we evaluate ReACC on end-to-end code completion.

5.1 Dataset

CodeXGLUE (Lu et al., 2021) is a benchmark dataset containing 14 datasets for 10 diversified code intelligence tasks. We use PY150 dataset (Raychev et al., 2016) in Python and GitHub Java Corpus dataset (Allamanis and Sutton, 2013) in Java from it for code completion task. Table 1 shows the data statistics.

5.2 Baseline Methods

CodeGPT/CodeGPT-adapted (Lu et al., 2021) are both pre-trained on Python and Java datasets from CodeSearchNet. CodeGPT is trained from scratch while CodeGPT-adapted is a domain adaptation model which is initialized by GPT-2 (Radford et al., 2019).

PLBART (Ahmad et al., 2021) is based on BART (Lewis et al., 2020) architecture which employs denoising sequence-to-sequence (Seq2Seq)

Model	Python		Java	
	MAP@100	Precision	MAP@100	Precision
CodeBERT	1.47	4.75	1.15	4.58
GraphCodeBERT	5.31	15.68	4.54	16.05
BM25	**10.32**	23.17	8.67	25.85
ReACC-retriever	9.60	**27.04**	**9.31**	**27.55**

Table 2: Results on zero-shot code clone detection dataset created from CodeNet.

Model	PY150			JavaCorpus		
	Perplexity	Exact Match	Edit Sim	Perplexity	Exact Match	Edit Sim
GPT-2	-	41.73	70.60	-	27.50	60.36
CodeGPT	2.502	42.18	71.23	4.135	28.23	61.81
CodeGPT-adapted	2.404	42.37	71.59	3.369	30.60	63.45
CodeT5-base	-	36.97	67.12	-	24.80	58.31
PLBART	-	38.01	68.46	-	26.97	61.59
ReACC-bm25	2.312	46.07	73.84	3.352	30.63	64.28
ReACC-dense	2.329	45.32	73.95	3.355	30.30	64.43
ReACC-hybrid	**2.311**	**46.26**	**74.41**	**3.327**	**30.70**	**64.73**

Table 3: Results on the code completion task in CodeXGLUE

	Exact Match	Edit Sim
GPT-2	37.08	68.71
CodeGPT	37.21	69.00
CodeGPT-adapted	38.77	70.07
X-CodeGPT	39.41	70.97
ReACC-bm25	**40.24**	71.65
ReACC-dense	39.67	71.80
ReACC-hybrid	40.15	**72.01**

Table 4: Results on the new testset created from PY150 in CodeXGLUE

pre-training and is pre-trained on unlabeled data across PL and NL.

CodeT5 (Wang et al., 2021) is also an encoder-decoder pre-trained model which adapts T5 (Raffel et al., 2019) architecture and considers the identifier-aware token type information in code.

X-CodeGPT is a variant of CodeGPT which adapts eWASH (Clement et al., 2021) to CodeGPT. Clement et al. (2021) propose eWASH, a method for leveraging the syntax hierarchy of source code to give the model wider field of vision in a file and achieving a new SOTA performance on the CodeXGLUE code completion task. We reproduce their method and develop X-CodeGPT by adapting eWASH to CodeGPT-adapted.

5.3 Experiment Setup

Fine-tune We fine-tune CodeGPT-adapted on PY150 and GitHub Java Corpus datasets, respectively, and use it as the generator in ReACC. The number of epochs for training PY150 is 30 and Java Corpus is 10, with a batch size of 96 and a learning rate of 2e-5. Except for X-CodeGPT, all other baseline models are fine-tuned with the same settings.

As for X-CodeGPT, we pre-train it with a training set extracted from CodeSearchNet in eWASH format, where each example is a function body with its corresponding extended context, as described by Clement et al. (2021). Since eWASH requires codes parsed into ASTs but codes in CodeXGLUE have been tokenized and cannot be parsed, we build a new dataset from PY150 to fine-tune X-CodeGPT on CodeXGLUE. As a result, we download the origin files in PY150 and create a new dataset that retains the train/valid/test split, as seen in Table 1.

Evaluation Following Lu et al. (2021), we conduct two code completion scenarios, token-level and line-level completion, to measure models' ability of predicting one and more tokens. Perplexity is the evaluation metric for token-level completion, whereas exact match accuracy (EM) and edit similarity are used for line-level completion. For token-level completion, based on the consideration of efficiency, instead of applying retrieval at each step, we retrieve similar codes based on current context after predicting the first 100 tokens, and leverage it for further prediction.

Retrieval Database We use the training set of PY150 and Java Corpus as retrieval database for test. We don't use the contrastive pre-training corpus (i.e., CodeSearchNet) in order to avoid the duplication between CodeXGLUE and CodeSearchNet as they are both extracted from GitHub.

	Exact Match	Edit Sim
CodeGPT-adapted	46.38	74.10
ReACC-bm25	55.88	79.62
ReACC-dense	64.21	84.57
ReACC-hybrid	**64.74**	**84.93**

Table 5: Results on the code completion task created from CodeNet Python dataset

Hybrid Retriever A linear combination of scores from BM25 and our dense retriever forms a hybrid retriever. Specifically, we calculate the score by $sim(q, c) + \alpha \cdot BM25(q, c)$ and let $\alpha = 0.9$ based on the results on dev set for both PY150 and Java Corpus datasets.

5.4 Results

Table 3 and Table 4 compare different baseline models on code completion task in the CodeXGLUE Python and Java datasets. ReACC framework with the hybrid retriever outperforms consistently than other baselines on all datasets, which proves our conjection that the "external" context is beneficial to the code completion task. The comparison with X-CodeGPT in Table 4 demonstrates that utilizing "external" context could be more useful than making the most of the current code file. Among three configurations of the retriever in ReACC, hybrid retriever performs best on almost all metrics except the exact match score in the new test set of PY150.

From Table 3, we can observe that comparing the two datasets, the improvement in the PY150 dataset is greater than that in the Java Corpus dataset. The reason for this is that the retrieval database for Java (i.e., the training set) is much smaller. The CodeXGLUE Java Corpus dataset contains only 12,934 files for training so that it's more difficult to retrieve similar code from them.

Another finding is that BM25 shows comparable results with dense retriever and even performs better in perplexity and exact match metrics. The findings indict that the code completion task can benefit from both semantically and lexically similar codes.

5.5 Analysis

ReACC in specific domain Both PY150 and Java Corpus datasets are extracted from GitHub repositories which are distributed in a wide domain. As some people frequently write codes in a more specific domain, e.g., data mining/pattern recognition domain for Kaggle[3] users, algorithm domain for ACM community, etc. To evaluate ReACC in a specific code domain, we construct a code completion Python dataset from CodeNet, which can be considered in algorithm domain. Table 5 reveals that ReACC significantly outperforms CodeGPT-adapted in CodeNet by 10% and 18% absolute improvement in edit similarity and exact match, respectively. According to the findings, ReACC is more effective in a specific domain. We also notice that ReACC with dense retriever outperforms BM25 significantly in CodeNet. It can be explained by the fact that in algorithm domain, semantically similar code may be more valuable than for code completion lexically similar code.

	EM	Edit Sim
ReACC-dense	**45.32**	**73.95**
Retriever		
- identifier renaming	44.91	73.14
- dead code insertion	45.11	73.57
- API sequence	44.77	73.01
- query truncation	43.93	72.65
Generator		
- fragment alignment	45.08	73.56

Table 6: Ablation study for both retriever and generator module. Experiments are run in CodeXGLUE PY150 dataset.

Ablation study To further understand how our training options affect model performance, we conduct ablation experiments. As seen in Table 6, when data argumentation and training strategies in retriever or generator are eliminated, the metrics degrade. The most essential factor among them is query truncation. Comparing the two semantic-preserving transformations, identifier renaming contributes more than dead code insertion.When fragment alignment is removed from generator, i.e. using the retrieved code snippet itself for generator, performance suffers slightly.

ReACC vs GitHub Copilot GitHub Copilot[4] is a powerful technique for code completion which uses OpenAI Codex (Chen et al., 2021) as the model backend. We run some qualitative examples with its extension in VSCode, which are shown in the Appendix B. It worth noting that Codex is more powerful than CodeGPT since it is a large-scale pre-trained model that is trained on all source codes in GitHub based on GPT-3 (Brown et al., 2020). However, in some cases, ReACC with CodeGPT as

[3] https://www.kaggle.com/

[4] https://copilot.github.com/

the generator outperforms Copilot. And in 6 Copilot itself can benefit from ReACC when it takes advantage of ReACC's retriever, which indicates the effectiveness of retrieval-augmented method for strong generative models.

6 Conclusion

We propose ReACC, a retrieval-augmented code completion framework that utilizes "external" context for the code completion task by retrieving semantically and lexically similar codes from existing codebase. We pre-train a dual-encoder as a retriever for partial code search, which retrieves code fragments given a partial code. Our method can adopt any architecture that can perform code completion as the generator. On the CodeXGLUE benchmark, ReACC achieves a state-of-the-art performance in the code completion task.

Acknowledgements

This work is supported by Microsoft Research Asia and IITP grants (2021-0-01696, High Potential Individuals Global Training Program)

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.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. 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*, pages 2655–2668.

Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *International Conference on Machine Learning*, pages 245–256. PMLR.

Achmad Arwan, Siti Rochimah, and Rizky Januar Akbar. 2015. Source code retrieval on stackoverflow using lda. In *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, pages 295–299. IEEE.

Brenda S Baker. 2007. Finding clones with dup: Analysis of an experiment. *IEEE Transactions on Software Engineering*, 33(9):608–621.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 511–521.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 964–974.

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

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1027–1040.

Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: Multi-mode translation of natural language and python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9052–9065.

Colin B Clement, Shuai Lu, Xiaoyu Liu, Michele Tufano, Dawn Drain, Nan Duan, Neel Sundaresan, and Alexey Svyatkovskiy. 2021. Long-range modeling of source code files with ewash: Extended window access by syntax hierarchy. *arXiv preprint arXiv:2109.08780*.

Saumya K Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems (TOPLAS)*, 22(2):378–415.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–8.

Hongchao Fang, Sicheng Wang, Meng Zhou, Jiayuan Ding, and Pengtao Xie. 2020. Cert: Contrastive self-supervised learning for language understanding. *arXiv preprint arXiv:2005.12766*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. 2021. Learning to complete code with sketches. In *International Conference on Learning Representations*.

Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 10073–10083.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. *arXiv preprint arXiv:1808.10025*.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9729–9738.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Communications of the ACM*, 59(5):122–131.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2269–2275.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Gautier Izacard and Édouard Grave. 2021. Leveraging passage retrieval with generative models for open domain question answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 874–880.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1073–1085. IEEE.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880.

Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. Editsum: A retrieve-and-edit framework for source code summarization.

Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 4159–25.

Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. 2016. Neural code completion.

Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 473–485.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28.

Xueguang Ma, Kai Sun, Ronak Pradeep, and Jimmy Lin. 2021. A replication study of dense passage retriever. *arXiv preprint arXiv:2104.05740*.

Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. *arXiv preprint arXiv:2103.11626*.

Henry Massalin. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. *arXiv preprint arXiv:2103.00020*.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

Nils Reimers and Iryna Gurevych. 2019. 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)*, pages 3982–3992.

Stephen Robertson and Hugo Zaragoza. 2009. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc.

Chanchal K Roy and James R Cordy. 2008. An empirical study of function clones in open source software. In *2008 15th Working Conference on Reverse Engineering*, pages 81–90. IEEE.

Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating clone detection tools with bigclonebench. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 131–140. IEEE.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443.

Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 329–340. IEEE.

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Xin Wang, Fei Mi Yasheng Wang, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2022. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 349–360. IEEE.

Zhirong Wu, Yuanjun Xiong, Stella Yu, and Dahua Lin. 2018. Unsupervised feature learning via non-parametric instance-level discrimination. *arXiv preprint arXiv:1805.01978*.

Hongwei Xi. 1999. Dead code elimination through dependent types. In *International Symposium on Practical Aspects of Declarative Languages*, pages 228–242. Springer.

Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering*, 22(6):3149–3185.

Ilsun You and Kangbin Yim. 2010. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE.

A Predefined Dead Code

We define a set of dead code to choose from for both Python and Java. We focus on four kinds of common statements, i.e., declaration statement, expression statement, conditional statement and looping statement. Examples are shown in figure 4. To generate a dead code snippet, we can use one kind of them or combine different statements together.

B Qualitative Examples

Figure 5 and figure 6 show qualitative examples of generated code by different models. ReACC + Copilot denotes ReACC framework with Copilot as the generator.

Category	Python	Java
Declaration	`var1 = 1`	`int var1 = 1;`
	`var1, var2, var3 = 1, [2,3], "name"`	`String var2 = "abc";`
		`int[] var3;`
Expression	`var1+var2`	`var1 += 2;`
	`var2.extend(var3)`	`var2.append("def");`
	`sorted(var2)`	`var3[0]--;`

Conditional statement	`var1 = 1`
	`if var1 < 10:`
	`    # several random simple`
	`statements here`
	`else:`
	`    # statements here`
		`int var = 4;`
		`if (var1 < 10){`
		`    // compound statement`
		`}`

	`var1 = 1 if True else 2`	

Looping statement	`for var1 in range(10):`
	`    # statements here`
		`for (int var = 0; var < 10; var++) {`
		`    // compound statement`
		`}`

	`var1 = 10`
	`while var1 > 0:`
	`    # statements here`
	`    var1 -= 1`
		`int var1 = 5;`
		`while (var1 > 0){`
		`    // compound statement`
		`    var1--;`
		`}`

Figure 4: Examples of predefined set of dead code. Vars are randomly selected from other files. Literals like strings and integers are also generated at random. For conditional and looping statements, several simple statements (i.e., declaration and expression) are generated to fill the body.

Input	`from __future__ import unicode_literals`
	`import calendar`
	`import datetime`
	`from django.utils import http as http_utils`
	`from daydreamer.tests.views.core import http`
	`class TestCase(http.TestCase):`
	`    def format_etag(self, etag):`
	`        return`	

Retrieved code	`from datetime import datetime`
	`from django.test import TestCase`
	`from django.utils import unittest`
	`from django.utils.http import parse_etags, quote_etag, parse_http_date`
	`FULL_RESPONSE = ''`
	`ETAG = ''`
	`EXPIRED_ETAG = ''`
	`class ConditionalGet(TestCase):`
	`    ...`	

CodeGPT	`etag`	Edit Sim: 26
ReACC	`http_utils.format_etag(etag)`	Edit Sim: 87
Copilot	`'"%s"' % etag`	Edit Sim: 25
ReACC + Copilot	`'"%s"' % etag`	Edit Sim: 25
Ground Truth	`http_utils.quote_etag(etag)`	

Figure 5: An qualitative example from PY150 test set. The input code comes from https://github.com/skibblenybbles/django-daydreamer/blob/master/daydreamer/tests/views/behaviors/http/base.py

Input

```
import ...
logger = borg.get_logger(__name__, default_level = "INFO")

def evaluate_split(run_data, alpha, split, train_mask, test_mask):
    training = run_data.masked(train_mask).collect_systematic([4])
    testing = run_data.masked(test_mask).collect_systematic([4])
    model = borg.models.MulEstimator(alpha = alpha)(training, 10, training)
    score = numpy.mean(borg.models.run_data_log_probabilities(model, testing))

    ...

def main(out_path, bundle, workers = 0, local = False):
    def yield_jobs():
        run_data = borg.storage.RunData.from_bundle(bundle)
        validation =
```

Retrieved code

```
import ...
logger = borg.get_logger(__name__, default_level = "INFO")

def evaluate_split(run_data, model_name, mixture, independent, instance_count,
train_mask, test_mask):
    testing = run_data.masked(test_mask).collect_systematic([4])
    training_all = run_data.masked(train_mask)
    training_ids = sorted(training_all.ids, key = lambda _: numpy.random.rand())

    ...

def main(out_path, experiments, workers = 0, local = False):
    logger.info("", len(experiments))
    get_run_data = borg.util.memoize(borg.storage.RunData.from_bundle)
    def yield_jobs():
        for experiment in experiments:
            logger.info("preparing experiment: %s", experiment)
            run_data = get_run_data(experiment["run_data"])
            validation = sklearn.cross_validation.ShuffleSplit(len(run_data), 32,
test_fraction = 0.1, indices = False)
            max_instance_count = numpy.floor(0.9 * len(run_data)) - 10
            instance_counts = map(int, map(round,
numpy.r_[10:max_instance_count:24j]))
            ...
```

CodeGPT	`borg.storage.RunData.from_bundle(run_data)`	Edit Sim: 40
ReACC	`sklearn.cross_validation.ShuffleSplit(len(run_data), 32, test_fraction=0.1, indices=False)`	Edit Sim: 77
Copilot	`run_data.masked(run_data.get_validation_mask())`	Edit Sim: 41
ReACC + Copilot	`sklearn.cross_validation.ShuffleSplit(len(run_data), 32, test_fraction = 0.1, indices = False)`	Edit Sim: 77
Ground Truth	`sklearn.cross_validation.KFold(len(run_data), 10, indices=False)`	

Figure 6: An qualitative example from PY150 test set. The input code comes from https://github.com/borg-project/borg/blob/master/borg/experiments/mul_over_alpha.py