CodeQA: A Question Answering Dataset for Source Code Comprehension

Chenxiao Liu, Xiaojun Wan
Wangxuan Institute of Computer Technology, Peking University
The MOE Key Laboratory of Computational Linguistics, Peking University
{jslcx,wanxiaojun}@pku.edu.cn

Abstract

We propose CodeQA, a free-form question answering dataset for the purpose of source code comprehension: given a code snippet and a question, a textual answer is required to be generated. CodeQA contains a Java dataset with 119,778 question-answer pairs and a Python dataset with 70,085 question-answer pairs. To obtain natural and faithful questions and answers, we implement syntactic rules and semantic analysis to transform code comments into question-answer pairs. We present the construction process and conduct systematic analysis of our dataset. Experiment results achieved by several neural baselines on our dataset are shown and discussed. While research on question-answering and machine reading comprehension develops rapidly, few prior work has drawn attention to code question answering. This new dataset can serve as a useful research benchmark for source code comprehension.

1 Introduction

Question Answering (QA) is the task of answering questions given a context about which the questions are being asked. With the advancement of deep learning and the availability of large-scale data, QA has received increasing attention from researchers. In recent years, QA has been applied into broad application domains, such as news (Hermann et al., 2015; Trischler et al., 2016), science (Khot et al., 2018; Hardalov et al., 2020), movies (Miller et al., 2016), medical field (Pampari et al., 2018), etc. Among QA's wide applications, code QA is an appealing application scenario on account of the distinctive nature of code differing from text.

In this study, we focus on generating QA pairs for source code for the purpose of source code comprehension. QA-based source code comprehension is the ability to read a code snippet and then answer questions about it, which requires understanding both source code and natural language. Take the question "*What does the code insert at the specified index?*" together with a code snippet in Table 1 as an example. To answer the question, one probably first reads the source code carefully, figures out that the method adds the given parameter "child" at the specified index into the object "children". Thus one gives a proper answer: "*The given child.*".

```
public void insertChildAt(Element child,
     int index){
    setChildParent(child);
    children.add(index, child);
}
```
Question: What does the code insert at the specified index?
Answer: The given child.

Table 1: A question-answer pair for a sample code snippet in the QA-based source code comprehension task.

Compared with code summarization task (Haiduc et al., 2010) that generates comments for codes, QA-based code comprehension task introduces more specific guidance and more explicit signals for models on what to generate. It provides more granularity levels ranging from method to variable, not just regarding several lines of code as a whole. Besides, it is easier to be evaluated since the output is more succinct, constrained and targeted (Kryściński et al., 2019).

QA-based source code comprehension has direct use in education to facilitate programming learning, where a system automatically answers questions about codes that someone has read. A more general use is to help improve software maintenance since it can advance the readability of code. Moreover, it can provide diverse information that can be leveraged to help perform a wide range of software engineering tasks, such as bug detection, specification inference, testing and code synthesis.

However, constructing a code QA dataset for source code comprehension is very challenging. Naturally occurring QA pairs on the Web are often complicated, noisy and contain information that

Findings of the Association for Computational Linguistics: EMNLP 2021, pages 2618–2632
November 7–11, 2021. ©2021 Association for Computational Linguistics

cannot be inferred from the source code. We tried to collect naturally occurring QA pairs from source code management platforms like Github, QA sites for programmers like StackOverflow, programming online judge platforms like Leetcode. But these QA pairs usually rely on knowledge apart from source code, which makes it problematic to disentangle modeling weakness from data noise. An alternative is to have experienced developers write QA pairs for source code from scratch, which is inefficient and cost-intensive.

In this study, we introduce a new data construction process to address the above challenges and propose CodeQA, a free-form question-answering dataset. First, to ensure that QA pairs are natural and clean, we utilize code comments as data source. We pick out two large-scale well-studied datasets from Github - a Java dataset and a Python dataset. Then we select comments that are suitable to generate QA pairs from these datasets. Targeting at generating various types of QA pairs such as Wh-questions and Yes/No questions, we implement syntactic rules and semantic analysis to transform comments into QA pairs. More specifically, comments are transformed into dependency trees and converted to fit question templates that are invoked by semantic role labels. We also analyze the verbal group of comments and generate Yes/No questions. After that, QA pairs with ambiguous answers and unbalanced counts of Yes/No are filtered.

Due to the varied nature of code comments, CodeQA covers a variety of information containing in codes, ranging from method to variable. We analyze our dataset and classify all the generated QA pairs into four categories: functionality, purpose, property and workflow. Our experiments with several baseline models demonstrate that neural models struggle to generate correct answers. These results suggest that our dataset could serve as a useful groundwork for QA-based source code comprehension.

Prior work (Bansal et al., 2021) built a dataset for code QA, but only a third of their questions are free-form. The main differences between our work and prior work are: first, all of our questions are free-form; second, our questions have diverse textual expressions and they are asking about information of various granularity in code, while questions in prior work are mostly fixed and on the single granularity.

Therefore, the contributions of this paper are as follows.

- We propose the QA-based source code comprehension task and introduce a large-scale question-answering dataset containing 119,778 QA pairs for 56,545 Java codes, and 70,085 QA pairs for 44,830 Python codes. As far as we know, it is **the first** diverse free-form QA dataset specially built for source code comprehension. The dataset is available at https://github.com/jadecxliu/CodeQA.

- We present a data construction process to generate code QA pairs based on code comments and advance a taxonomy to classify code QA pairs into four categories.

- We provide several baselines to evaluate the QA-based source code comprehension task. Experimental results demonstrate this dataset could serve as a useful benchmark for model and metric development.

2 Related Work

2.1 Question Answering

Question answering has a long history and has attracted increasing attention in recent years. Question Answering tasks are usually divided into four categories (Chen, 2018; Qiu et al., 2019; Liu et al., 2019a): cloze tests, multiple-choice, span prediction and free-form answering. A few examples of QA datasets in each category are CNN & Daily Mail (Hermann et al., 2015), RACE (Lai et al., 2017), SQuAD (Rajpurkar et al., 2016), MS MARCO (Nguyen et al., 2016). Compared with other categories, free-form answering tasks show their superiority in the dimensions of understanding, flexibility, and application, which are the closest to practical application. However, the flexibility of the answer form brings difficulty to build datasets (Liu et al., 2019a). On these datasets, earlier work in question answering employed rule-based and machine-learning-based methods. Recent deep-learning techniques leveraged neural networks with different attention mechanisms and pre-trained text representation (Yamada et al., 2020; He et al., 2020), improving the ability of extracting contextual information and context-question interaction.

In code QA, Bansal et al. (2021) designed a context-based QA system for *basic* questions about subroutines and evaluated the system by an RNN-based encoder-decoder network. They define the "*basic*" question as a question about a small detail of a method, such as "What are the parameters to the method?". These questions can be solved by parsing code without source code comprehension. Remaining questions are similar to code summarization, such as "What does method do?". Compared with existing work, we construct a more complex and attractive QA dataset for the purpose of code comprehension, which requires the understanding of both source code and natural language.

2.2 Code Summarization

Code summarization is the task of creating readable summaries that describe the functionality of a code snippet. Neural source code summarization approaches frame the problem as a sequence generation task (Iyer et al., 2016) and use encoder-decoder networks with attention mechanisms. Some approaches utilized the structural information of code, such as Code2seq proposed by Alon et al. (2018), DeepCom proposed by Hu et al. (2018a), ast-attendgr proposed by LeClair et al. (2019). Structural information can be also encoded into tree structure encoders such as Tree-LSTM (Shido et al., 2019), Tree-Transformer (Harer et al., 2019), and Graph Neural Network (LeClair et al., 2020). Besides, other techniques like reinforcement learning (Wan et al., 2018), dual learning (Wei et al., 2019), retrieval-based techniques (Zhang et al., 2020), language-agnostic representation learning (Zügner et al., 2021) further enhance the code summarization models. Recently, neural architectures like Transformer (Vaswani et al., 2017) and large pre-trained models (Peters et al., 2018; Radford et al., 2018; Devlin et al., 2018; Liu et al., 2019b) have brought improvements on code summarization task. Representative works are transformer designed for code (Ahmad et al., 2020), CodeBERT(Feng et al., 2020), which is a model pre-trained on the CodeSearchNet (Husain et al., 2019) dataset for programming and natural languages.

Among these noteworthy works, two datasets including a Java dataset (Hu et al., 2018b) and a Python dataset (Barone and Sennrich, 2017) are popular when conducting experiments. We construct our QA dataset based on the two datasets.

3 The Code QA Task

In this work, we focus on building a dataset and setting up baselines for the following code QA task: Given a source code c and a natural language question q about c, a free-form textual answer a is required to be generated. The textual answer a may be a word, a phrase or a sentence, and it usually cannot be directly extracted from the source code. This task is different from traditional machine reading comprehension, as code is very different from text and we need programming knowledge to understand it. The source code and the natural language question are actually in two different languages and a QA system should have the ability to understand both the code language and the natural language. Moreover, the system needs to generate an answer faithful to the question and the corresponding code, rather than extract some tokens from the code. Therefore, the task is very challenging and it is very meaningful and urgent to construct and release a large-scale dataset for this task.

4 Dataset Construction

4.1 Data Source

We construct our code QA dataset based on two code summarization datasets. The first one is a parallel corpus consisting of about a hundred thousand Python methods with descriptions written by their own programmers collected from Github (Barone and Sennrich, 2017). Each source code object contains a "docstring" (documentation string), which is retained at runtime as metadata. Programmers use docstrings to describe the functionality, interface or usage of code objects. Docstrings are extracted as natural language descriptions for code summarization tasks. The second is a parallel corpus of over seventy thousand Java code-comment pairs from Github (Hu et al., 2018b). The dataset contains the Java methods and the corresponding Javadoc comments. These comments describe the functionality of Java methods and are taken as code summaries.

Code comments like Python docstrings and Javadocs can be viewed as the source of QA pairs. As code comments are deemed faithful to the code snippets, the QA pairs generated from the code comments are also faithful to the code snippets. Note that the code comments are only used for generating QA pairs, but not provided in the final code QA dataset. The comment taxonomy constructed by prior work (Zhai et al., 2020) illustrates

Potential Answer	Question Template	Sample Comment	Generated QA
subject (nsubj)	**Wh** mainAux otherAux verb obj modifiers?	The aliases will be associated with the index when it gets created.	Q: What will be associated with the index when it gets created? A: The aliases.
direct object (dobj)	**Wh** mainAux nsubj otherAux verb modifiers?	The code trims all occurrences of the supplied leading character from the given string.	Q: What does the code trim from the given string? A: All occurrences of the supplied leading character.
open clausal complement (xcomp)	**Wh** mainAux nsubj verb modifiers?	The function tries to request new data if the dialog is open and a stream is set.	Q: What does the function try if the dialog is open and a stream is set? A: To request new data.
Temporal (TMP)	**When** mainAux nsubj otherAux verb obj modifiers?	The code takes a screenshot after every test.	Q: When does the code take a screenshot? A: After every test.
Locative (LOC)	**Where** mainAux nsubj otherAux verb obj modifiers?	This method positions the stream at the first central directory record.	Q: Where does this method position the stream? A: At the first central directory record.
Manner (MNR)	**How** mainAux nsubj otherAux verb obj modifiers?	The code creates an instance of a class using the specified classloader.	Q: How does the code create an instance of a class? A: Using the specified classloader.
Cause (CAU)	**Why** mainAux nsubj otherAux verb obj modifiers?	The code always returns true since we wanna get all vars in scope.	Q: Why does the code always return true? A: Since we wanna get all vars in scope.
Purpose (PNC and PRP)	**For what purpose** mainAux nsubj otherAux verb obj modifiers?	The function adds and removes entries from the statements collection to munge wikibase rdf exports into a more queryable form.	Q: For what purpose does the function add and remove entries from the statements collection? A: To munge wikibase rdf exports into a more queryable form.

Table 2: A few templates to describe the construction of QA pairs.

that the content of comment can be classified into five perspectives: *what*, *why*, *how-it-is-done*, *property* and *how-to-use*. The diverse perspectives of content provide rich information to dig up and be transformed into QA pairs. Thus, we can generate question-answer pairs for code snippets by identifying potential answers in code comments, such as asking about the constraints or intentions of the key components of code occurring in comments.

4.2 Comment Selection

Not all comments are suitable for generating QA pairs. We thus define a selection process to help filter out noisy comments. Most comments lack the subject, such as "attach votes count to each object of the queryset", which starts with a verb and the hidden meaning is "the code attaches votes count to each object of the queryset". Incomplete sentences add difficulty to parsing in the following stage. So if a comment lack the subject, we add "the code" at the beginning of the sentence. Besides, we filter incomplete comments (still under development) or comments unrelated to the corresponding code. These comments are clued by keywords including "TODO", "license", "ownership", etc. (Pascarella and Bacchelli, 2017).

4.3 Question and Answer Formulation

Typically, questions can be divided into several types (Day and Park, 2005): General questions, with Yes/No answers; Wh-questions, starting with what, where, when and so on; Choice questions, where there are some options inside the question. Since the questions in this work are converted from code comments, we focus on general questions and wh-questions, and leave choice questions as future work.

To obtain wh-questions and general questions from code comments, we implement rule-based and template-based methods to convert syntactic and semantic representations into QA pairs.

For wh-questions, we make use of dependency parsing (DP) and semantic role labeling (SRL). For one thing, we transform comments into dependency trees in the format of Universal Dependencies (UD) (De Marneffe et al., 2014) by using the allennlp parser (Gardner et al., 2018). We extract a potential answer where a verb is headed by a few dependency nodes in the dependency tree with the help of semantic role labels (SRL) according to the Propbank

1.0 specifications (Palmer et al., 2005). The SRL model is provided by Gardner et al. (2018). For another, we extract the roles of each predicate occurred in the comment by SRL. According to Propbank, the roles are *proto-agent*, *proto-patient*, *location*, *direction*, *manner*, *extent*, *cause*, etc. Roles like *location*, *direction* are classified as modifiers, which can formulate our answers. In the end, we use some of the predefined handwritten templates in Dhole and Manning (2020) to generate QA pairs. Table 2 presents a few templates and examples to describe the construction of questions and answers. The first three rows are from dependency heuristics and others are from SRL heuristics. The detailed process of wh-question formulation is provided in Appendix A.

With respect to general questions, we analyze the verbal group of the comment and generate a Yes/No question for every predicate that has a finite verb. We generate multiple Yes/No questions for each predicate if a comment contains multiple predicates. First, we select a clause for the current predicate and may rearrange the sequential position of semantic role labeling arguments. The standard declarative word order is preserved when generating a QA. When copular, modals, or cases if an auxiliary be/have/do is already present, we do not provide do-support. Otherwise, we add do-support and may move adjunct arguments relative to the main verb. As the negation label of the main verb in the verbal group indicates the polarity, according to Flor and Riordan (2018), we do not transfer the negation into generated question, but flip the answer from "yes/no" to "no/yes". For example, from "windows don't have a mode like linux cli example", we derive the question "Do windows have a mode like linux cli example?" and the answer "No".

Since some comments can not be successfully parsed to generate QA pairs, we construct 115,807 QA pairs from 44,867 code snippets in Python dataset, and 203,229 QA pairs from 56,583 code snippets in Java dataset.

4.4 Postprocessing

To generate high-quality code QA pairs, we filter the QA pairs that have ambiguous answers, such as answers only containing pronouns. Since some comments start with "the method" "this function" and we have added "the code" at the beginning of some comments when preprocessing, some generated QA pairs question about the subject and get answers like "this method". We also filter these QA pairs as they do not provide specific information about code snippets.

	Java	**Python**
Size of training set	95,778	56,085
Num. of unique codes	43,339	34,641
Num. of unique tokens in codes	27,504	108,571
Num. of unique tokens in questions	13,097	11,401
Num. of unique tokens in answers	13,820	12,723
Avg. Num. of tokens per code	119.52	48.97
Avg. Num. of tokens per question	9.48	8.15
Avg. Num. of tokens per answer	4.74	4.07
Size of dev set	12,000	7,000
Size of test set	12,000	7,000

Table 3: CodeQA dataset statistics.

	Java	**Python**
Dev answer repetition rate against Train	5.51%	1.17%
Test answer repetition rate against Train	3.63%	1.56%
Span answer	1.47%	4.99%
Extraction answer	3.71%	12.57%

Table 4: The repetition rate and extraction rate of answers.

Besides, the original ratio of Yes questions to all questions is too high, with a heavily uneven proportion of Yes questions to No questions in our dataset. So we delete the majority of Yes questions to achieve a relative balance between Yes and No questions. Then we split each dataset into training, development and test sets in proportion with 8 : 1 : 1 after shuffling the pairs.

5 Dataset Analysis

In this section, we introduce the overall statistics of our dataset and verify the free-form characteristic of our dataset. To explore the distribution of different kinds of source code QA pairs, we propose a taxonomy of four types of source code comprehension.

5.1 Overall Statistics

Table 3 describes the basic statistics of our CodeQA dataset. In Java dataset, there are 95,778 training pairs, 12,000 development pairs and 12,000 test pairs. In Python dataset, there are 56,085 training pairs, 7,000 development pairs and 7,000 test pairs.

We calculate the percentage of answers in the development or test set that can be found in the training set (excluding Yes/No answers). Besides,

Type	Example QA	Percentage
Functionality	Q: What does the code instantiate? A: An input data placeholder variable.	63%
Purpose	Q: For what purpose does this method return a new name? A: To address the key value pair in the context of that user.	9%
Property	Q: When is this event triggered on the slave instances? A: Every time a stats report is to be sent to the locust master.	17%
Workflow	Q: How does a maximal independent set of nodes in g return? A: By repeatedly choosing an independent node of minimum degree.	11%

Table 5: Distribution of different QA pairs among 100 randomly chosen samples.

Question Type	Percentage
What	67.24%
How	8.93%
Where	5.85%
When	6.89%
Why	1.02%
For what purpose	5.08%
Yes/No	2.86%
Other	2.13%

Table 6: Distribution of questions after automatic partitioning.

we calculate the percentage of answers in all sets that are spans of the code, and the percentage of answers whose each token could be extracted from the code, as shown in Table 4. The statistics attest to the free-form nature of our dataset.

5.2 Categorization

We are not aware of an agreed-upon typology of all code QA types. Categorizations of different types of code summarization exist (Pascarella and Bacchelli, 2017; Zhai et al., 2020), but the provided categorizations differ and are manually classified by coders in general. Bansal et al. (2021) generated QA pairs about code and divided the questions into six types, including basic extractive question types like "What is the return type of method?", "What are the parameters of method?" and question types equivalent to code summarization, i.e. "What does method do?". After consulting both prior works and a separate part of the training data, we characterize the data into the following four types.

These types consist of (1) Functionality. It provides a definition of the range of functions that the subject and/or its interface can perform. (2) Purpose. It explains the reason why the subject is provided or the design rationale of the subject. (3) Property. It declares properties of the subject, such as pre-conditions and post-conditions of a method or some statements. Pre-conditions specify the constraints that should satisfy in order to use the subject while post-conditions indicate the result of using the subject. (4) Workflow. It describes how the subject is done, which means implementation details like the design or the workflow of the subject. The subject mentioned in the four types can either be the whole method or a key component of the code, e.g. a statement, a variable.

To get a better understanding of the categorizations of code QA pairs, we sampled 100 QA pairs in the development set of Python, and then manually labeled the examples with the categories shown in Table 5. The results show that more than half of questions target functionality, while 37% questions ask about purpose, property, or workflow.

To show the diversity in QA pairs, we also automatically categorize all the QA pairs in Table 6. We can see that What question makes up 67.24% of the data; 27.77% of the questions are How/Where/When/Why/For what purpose; 2.86% are Yes/No; and the remaining 2.13% are other types.

6 Experiments

In this section, we first introduce four baseline models for this task. Then we compare and analyze the results of different models under both automatic and human evaluation metrics.

6.1 Baselines

We present baseline results on CodeQA by examining four existing typical approaches. Since no previous work specifically designs a model for QA-based source code comprehension, we make some modifications to each of the existing approaches. Note that we do not employ retrieval models, for the answer repetition rate is quite low as shown in Table 4. Details about hyperparameter settings of all baselines are provided in the Appendix B.

- **Seq2seq**: A Seq2seq model (Sutskever et al., 2014) with attention and copy mechanism

	Dev					Test				
	BLEU	ROUGE	METEOR	EM	F1	BLEU	ROUGE	METEOR	EM	F1
Seq2seq	28.86	20.13	5.21	4.11	21.02	28.29	18.88	4.79	3.23	19.75
Dual Encoder	30.80	22.78	7.21	5.95	23.57	29.51	20.39	6.27	4.06	21.16
Transformer	31.54	23.67	7.69	6.52	24.39	30.22	21.26	6.77	4.41	22.04
CodeBERT	33.45	31.80	11.57	7.80	32.69	32.40	28.22	10.10	6.20	29.20
CodeBERT*	37.32	33.06	13.36	11.00	34.38	35.19	33.56	10.99	9.00	32.07
Human*	64.18	66.10	29.04	37.00	63.22	62.97	63.44	28.19	34.00	64.06

Table 7: Performance of various models on Java dataset. **means the result is obtained on only 100 questions sampled in the respective dataset. We evaluate the 100 answers generated by CodeBERT and that given by an experienced programmer respectively. The results are used only for comparing CodeBERT and Human.*

	Dev					Test				
	BLEU	ROUGE	METEOR	EM	F1	BLEU	ROUGE	METEOR	EM	F1
Seq2seq	29.70	22.18	6.36	2.95	23.67	30.09	21.69	6.38	2.84	23.27
Dual Encoder	28.57	17.62	4.43	2.22	18.78	28.90	17.79	4.53	2.22	18.97
Transformer	30.85	23.96	7.91	3.37	25.30	30.69	23.26	7.80	3.35	24.59
CodeBERT	33.84	30.27	11.51	5.50	31.57	34.86	30.28	12.51	4.93	31.56
CodeBERT*	34.02	29.89	12.18	8.00	35.89	34.78	30.93	13.45	8.00	34.10
Human*	56.21	60.65	26.32	34.00	61.98	55.96	59.55	25.49	32.00	61.59

Table 8: Performance of various models on Python dataset. * *has the same meaning as in the above table.*

(See et al., 2017). While originally designed for text-to-text generation, it is commonly used in free-form question-answering as well (Nguyen et al., 2016). The input of the model is in the form of "[CLS] Question [SEP] Code". Since models using original code tokens could perform better than models using abstract syntax tree (AST) sequences (Ahmad et al., 2020), we employ the code tokens as input for all baseline models.

- **Dual Encoder**: A seq2seq model with two encoders. The model first builds a code representation and a question representation by its code-info encoder and question-info encoder respectively. After that, it concatenates the two representations for the decoder. Both encoders and decoder are similar to the architecture in the above Seq2seq model.

- **Transformer**: A Transformer encoder-decoder model (Vaswani et al., 2017) with relative position representations (Shaw et al., 2018) and copy attention. The input is a sequence containing a question and a code separated by [SEP]. Since the semantic representation of a code does not rely on the absolute positions of its tokens, the Transformer ignores the directional information and encodes pairwise relationship (Ahmad et al., 2020).

- **CodeBERT**: A Transformer encoder-decoder model where the encoder are initialized with CodeBERT (Feng et al., 2020). Following BERT (Devlin et al., 2018) and RoBERTa (Liu et al., 2019b), CodeBERT is a bimodal model pre-trained with natural language and programming languages including Python and Java, etc. We fine-tune the model parameters on our dataset and predict answers given an input sequence consisting of a question and a code. Note that when training a version of CodeBERT by traversing the AST of code, model does not bring improvements on generation tasks (Feng et al., 2020). Thus we do not transform the code into tree structure.

6.2 Automatic Evaluation Metrics

The model output is evaluated using several automatic metrics: BLEU (Papineni et al., 2002), ROUGE-L (Lin, 2004), METEOR (Banerjee and Lavie, 2005), Exact Match (EM) and F1.

6.3 Model Performance

Tables 7, 8 present the results with different models for the QA-based source code comprehension task on Java and Python datasets, respectively. CodeBERT performs the best, followed by Transformer. It is not surprising since pre-trained models on programming codes and texts are more powerful in encoding code representations and bridging the gap between code language and natural language,

	Java				Python			
	Fluency		Correctness		Fluency		Correctness	
	Dev	Test	Dev	Test	Dev	Test	Dev	Test
Seq2seq	2.09	2.21	1.66	1.62	2.23	2.24	1.61	1.63
Dual Encoder	2.23	2.19	1.79	1.57	2.38	2.44	1.58	1.64
Transformer	2.37	**2.41**	1.84	1.75	2.40	2.44	1.70	1.72
CodeBERT	**2.53**	2.40	**1.92**	**1.89**	**2.45**	**2.50**	**1.78**	**1.79**

Table 9: Human evaluation results. Scores of each aspect range from 1 to 3 and higher scores are better.

```
public IOContainer append(IOObject[]
    output){
    List<IOObject> newObjects = new
        LinkedList<>();
    for(int i = _NUM; i < output.length;
         i++){
        newObjects.add(output[i]);
    }
    newObjects.addAll(ioObjects);
    return new IOContainer(newObjects);
}
```

Q: What is added to the given ioobjects?
A: all ioobjects of this container
Seq2seq: an array
Dual Encoder: the given objects
Transformer: a new array
CodeBERT: an iocontent container

```
def get_page_args():
    pages = {}
    for arg in request.args:
        re_match = re.findall("page_(.*)
            ", arg)
        if re_match:
            pages[re_match[0]] = int(
                request.args.get(arg))
    return pages
```

Q: What does the code get?
A: page arguments
Seq2seq: the page
Dual Encoder: the args
Transformer: the page of the pages
CodeBERT: page arguments from the request

Table 10: Examples of different models' performance on Java and Python datasets.

thus improving the performance. We see that all the models get low Exact Match scores, indicating that answering the questions with the same token string as the gold answer is rather difficult.

6.4 Human Performance

We assess human performance on CodeQA's development and test sets. Due to the large scale of the dataset, we sampled 100 questions from each set and asked two experienced programmers to give answers according to code snippets on two sets respectively. To make a comparison between model performance and human performance, we pick up the best model (CodeBERT) and evaluate its output on the same 100 questions. As shown in the last two rows of Tables 7, 8, the model' performance has a significant gap compared with human's.

6.5 Human Evaluation

Besides automatic evaluation, we randomly sampled 100 QA pairs from the development and test sets of CodeQA respectively, and asked two programmers to evaluate outputs of baselines on two sets respectively in the following aspects. *Fluency* measures if an answer is grammatically correct and is fluent to read. *Correctness* measures if an answer is targeting the given question and code. Each reviewer gives a score between 1 and 3 for each aspect, with 3 indicating the best quality.

As shown in Table 9, CodeBERT gets the competitive performance in most metrics. All models get relatively poor performances on the aspect of correctness compared with fluency. The low scores of correctness indicate that it is quite challenging for models to do well in code QA.

6.6 Qualitative Analysis

We provide a couple of examples in Table 10 to demonstrate the outputs from different baselines (more qualitative examples are provided in Appendix C). In the Java example, CodeBERT captures the key component "io container" while other models generate imprecise concepts. As the Python code tries to get all arguments of a page, the first three baselines generate answer either about "page" or about "args" while CodeBERT contains both concepts. The examples reveal that, in comparison to the Seq2seq model and the Transformer model, CodeBERT generates more detailed and accurate answers.

7 Conclusion

In this paper, we build the first diverse free-form question answering dataset for code by transforming code comments into QA pairs. We also provide several neural baselines, and demonstrate that CodeQA could lay the foundation for further research on QA-based source code comprehension.

For future work, how to expand the number of question types and generate more high-quality QA pairs are the major challenges. Besides, we will explore more powerful QA models to better leverage information from code and capture interaction between code and question.

Acknowledgments

This work was supported by National Natural Science Foundation of China (61772036), Beijing Academy of Artificial Intelligence (BAAI) and Key Laboratory of Science, Technology and Standard in Press Industry (Key Laboratory of Intelligent Press Media Technology). We appreciate the anonymous reviewers for their helpful comments. Xiaojun Wan is the corresponding author.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Aakash Bansal, Zachary Eberhart, Lingfei Wu, and Collin McMillan. 2021. A neural question answering system for basic questions about subroutines. *arXiv preprint arXiv:2101.03999*.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*.

Danqi Chen. 2018. *Neural reading comprehension and beyond*. Stanford University.

Richard R Day and Jeong-suk Park. 2005. Developing reading comprehension questions. *Reading in a foreign language*, 17(1):60–73.

Marie-Catherine De Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, and Christopher D Manning. 2014. Universal stanford dependencies: A cross-linguistic typology. In *LREC*, volume 14, pages 4585–4592.

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*.

Kaustubh D Dhole and Christopher D Manning. 2020. Syn-qg: Syntactic and shallow semantic rules for question generation. *arXiv preprint arXiv:2004.08694*.

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*.

Michael Flor and Brian Riordan. 2018. A semantic role-based approach to open-domain automatic question generation. In *Proceedings of the thirteenth workshop on innovative use of NLP for building educational applications*, pages 254–263.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. 2018. Allennlp: A deep semantic natural language processing platform. *arXiv preprint arXiv:1803.07640*.

Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *2010 acm/ieee 32nd international conference on software engineering*, volume 2, pages 223–226. IEEE.

Momchil Hardalov, Todor Mihaylov, Dimitrina Zlatkova, Yoan Dinkov, Ivan Koychev, and Preslav Nakov. 2020. Exams: A multi-subject high school examinations dataset for cross-lingual and multilingual question answering. *arXiv preprint arXiv:2011.03080*.

Jacob Harer, Chris Reale, and Peter Chin. 2019. Tree-transformer: A transformer-based method for correction of tree-structured data. *arXiv preprint arXiv:1908.00449*.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2020. Deberta: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654*.

Karl Moritz Hermann, Tomáš Kočiskỳ, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching machines to read and comprehend. *arXiv preprint arXiv:1506.03340*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred api knowledge.

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*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083.

Tushar Khot, Ashish Sabharwal, and Peter Clark. 2018. Scitail: A textual entailment dataset from science question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Wojciech Kryściński, Nitish Shirish Keskar, Bryan McCann, Caiming Xiong, and Richard Socher. 2019. Neural text summarization: A critical evaluation. *arXiv preprint arXiv:1908.08960*.

Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. 2017. Race: Large-scale reading comprehension dataset from examinations. *arXiv preprint arXiv:1704.04683*.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 184–195.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Shanshan Liu, Xin Zhang, Sheng Zhang, Hui Wang, and Weiming Zhang. 2019a. Neural machine reading comprehension: Methods and trends. *Applied Sciences*, 9(18):3698.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019b. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Alexander Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. 2016. Key-value memory networks for directly reading documents. *arXiv preprint arXiv:1606.03126*.

Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. Ms marco: A human generated machine reading comprehension dataset. In *CoCo@ NIPS*.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational linguistics*, 31(1):71–106.

Anusri Pampari, Preethi Raghavan, Jennifer Liang, and Jian Peng. 2018. emrqa: A large corpus for question answering on electronic medical records. *arXiv preprint arXiv:1809.00732*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 227–237. IEEE.

Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*.

Boyu Qiu, Xu Chen, Jungang Xu, and Yingfei Sun. 2019. A survey on neural machine reading comprehension. *arXiv preprint arXiv:1906.03824*.

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

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.

Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368*.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*.

Adam Trischler, Tong Wang, Xingdi Yuan, Justin Harris, Alessandro Sordoni, Philip Bachman, and Kaheer Suleman. 2016. Newsqa: A machine comprehension dataset. *arXiv preprint arXiv:1611.09830*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *arXiv preprint arXiv:1910.05923*.

Ikuya Yamada, Akari Asai, Hiroyuki Shindo, Hideaki Takeda, and Yuji Matsumoto. 2020. Luke: deep contextualized entity representations with entity-aware self-attention. *arXiv preprint arXiv:2010.01057*.

Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. Cpc: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1359–1371.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1385–1397. IEEE.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318*.

A Wh-question Formulation Details

To generate wh-questions, we first parse comments into dependency trees in the format of Universal Dependencies (UD) by using the allennlp parser. Dependency trees are syntactic tree structures, where syntactic units are connected via links. We extract the clause of a verb headed by a few dependency nodes which can serve as answers with the help of PropBank's predicate-argument structure (SRL). The clause is treated as a combination of a subject, an object, the head verb and other non-core arguments. Furthermore, the clause can be refined with modals, auxiliaries and negations if found around the verb. The SRL model is provided by Gardner et al. (2018). Then templates in Dhole and Manning (2020) are used to generate QA pairs. The templates convert *What* to *Who/Whom*, *When* or *Where* depending on the named entity of the answer. To ensure subject-predicate concord, templates modify *do* to *does* or *did* relying on the tense and number of the subject.

Algorithm 1 illustrates the heuristic rules of dependency parsing.

Algorithm 1 Heuristic Rules of DP

```
 1: $\{d_0, ..., d_n\} \leftarrow DP(w_0...w_n)$
 2: for $i = 0\ to\ n$ do
 3:     if $parent(d_i)\ is\ not\ null$ then
 4:         $d_v \leftarrow parent(d_i)$
 5:         $\{A_0, ..., A_{CAU}, A_{TMP}\} \leftarrow SRL(d_v)$
 6:         $subj \leftarrow A_0$
 7:         if $d_i \in A_1$ then
 8:             $obj \leftarrow A_1$
 9:         else
10:             $obj \leftarrow A_2$
11:         $A_x \leftarrow \sum(A_3, ..., A_{TMP})$
12:         $verb \leftarrow \{d_v, modals, negation\}$
13:         $template \leftarrow dep_{type} \leftarrow d_i$
14:         $QA \leftarrow template(subj, obj,$
                    $A_x, verb)$
```

Then we extract the roles of each predicate occurred in the comment by the SRL model provided by Gardner et al. (2018). Semantic roles include the generalized core arguments of predicates labeled as A0, A1, etc., with a set of adjunct modifiers. According to Propbank 1.0, the roles are *proto-agent*, *proto-patient*, *location*, *direction*, *manner*, *extent*, *cause*, etc. Roles like *location*, *direction* are classified as modifiers, which can formulate our answers. We make use of a set of predefined handwritten templates in Dhole and Manning (2020), which convert a comment into an interrogative statement by rearranging the arguments according to the modifier.

Algorithm 2 describes the heuristic rules of semantic role labeling.

Algorithm 2 Heuristic Rules of SRL

```
 1: $\{SRL_0, ..., SRL_s\} \leftarrow SRL(w_0...w_n)$
 2: for $i = 0\ to\ s$ do
 3:     if $SRL_i\ contains\ A_0\ or A_1\ and \geq 1\ A_m$
    then
 4:         $\{A_0, ..., A_{CAU}, A_{TMP}\} \leftarrow SRL_i$
 5:         if $A_x = modifier$ then
 6:             for $A_x \in SRL_i$ do
 7:                 $subj \leftarrow A_0$
 8:                 $A_x^- \leftarrow \sum(A_3, ...,$
                        $A_{TMP} - A_x)$
 9:                 $verb \leftarrow \{A_v, modals,$
                        $negation\}$
10:                 $template \leftarrow modifier_{type}$
                        $\leftarrow A_x$
11:                 $QA \leftarrow template(subj, A_x,$
                        $A_x^-, verb)$
```

B Baseline Details

- **Seq2seq**: bidirectional RNN with number of layers = 2, hidden size = 512, batch size = 32, beam size = 4, learning rate = 0.002, dropout = 0.2, Adam optimizer (Kingma and Ba, 2014).

- **Dual Encoder**: Both the code-info encoder and the question-info encoder have 2 layers. Other hyper-parameters are same as Seq2seq.

- **Transformer**: Transformer model with number of layers = 6, number of heads = 8, hidden size = 512, batch size = 32, beam size = 4, initial learning rate = 0.0001, dropout = 0.2, Adam optimizer.

- **CodeBERT**: The encoder is the pre-trained CodeBERT, while the decoder is a transformer structure with number of layers = 6, number of heads = 12. Other hyper-parameters: batch size = 64, beam size = 10, learning rate = 5e-5, Adam optimizer.

For each of the first three models, we train the model for a maximum of 200 epochs on a Nvidia

1080 Ti GPU and perform early stop if the validation performance does not improve for 20 consecutive iterations. We fine-tune CodeBERT for 20 epochs on 3 Nvidia 1080 Ti GPUs and select the checkpoint with best BLEU score.

C Qualitative Examples

```
public Select<T> sortAsc(String[] columns){
    for(String column:columns){
        mSortingOrderList.add(column + STRING);
    }
    return this;
}
```
Question: How does the specified columns sort?
Answer: in asc order
Seq2seq: in ascending order
Dual Encoder: in desc order
Transformer: in desc order
CodeBERT: in ascending order

```
public static int count(String string, String mark){
    if(!TextUtils.isEmpty(string) && !TextUtils.isEmpty(mark)){
        int count = _NUM;
        int index = string.indexOf(mark);
        while(index != -_NUM){
            count++;
            string = string.substring(index + mark.length());
            index = string.indexOf(mark);
        }
        return count;
    }
    return _NUM;
}
```
Question: What does the code count?
Answer: how many marks existed in string
Seq2seq: the string
Dual Encoder: the number of elements in the string
Transformer: the number of occurrences of this string
CodeBERT: the number of times the given string

```
public synchronized void create(long seqno)
        throws ReplicatorException, InterruptedException{
    if(file.exists()){
        throw new THLException(STRING + file.getName());
    }
    try{
        dataOutput = new BufferedFileDataOutput(file, bufferSize);
    } catch(IOException e){
        throw new THLException(STRING + file.getName(), e);
    }
    mode = AccessMode.write;
    try{
        write(MAGIC_NUMBER);
        write(MAJOR_VERSION);
        write(MINOR_VERSION);
        write(seqno);
        flush();
    } catch(IOException e){
        throw new THLException(STRING + file.getName(), e);
    }
    baseSeqno = seqno;
    if(logFlushTask != null) logFlushTask.addLogFile(this);
}
```
Question: What does the code create?
Answer: a new log file
Seq2seq: a new instance
Dual Encoder: a flie
Transformer: a file
CodeBERT: a log file

Table 11: Qualitative examples of different models' performance on Java dataset.

```python
def html_to_xhtml(html):
    try:
        html = html.getroot()
    except AttributeError:
        pass
    prefix = "{%s}" % XHTML_NAMESPACE
    for el in html.iter(etree.Element):
        tag = el.tag
        if tag[0] != '{':
            el.tag = prefix + tag
```

Question: How do all tags in an html tree convert to xhtml?
Answer: by moving them to the xhtml namespace
Seq2seq: recursively
Dual Encoder: with the given html
Transformer: using xhtml tags
CodeBERT: by removing their xhtml namespace

```python
def table_extend(tables, keep_headers=True):
    from copy import deepcopy
    for ii, t in enumerate(tables[:]):
        t = deepcopy(t)
        if t[0].datatype == 'header':
            t[0][0].data = t.title
            t[0][0]._datatype = None
            t[0][0].row = t[0][1].row
            if not keep_headers and (ii > 0):
                for c in t[0][1:]:
                    c.data = ''
        if ii == 0:
            table_all = t
        else:
            r1 = table_all[-1]
            r1.add_format('txt', row_dec_below='-')
            table_all.extend(t)
    table_all.title = None
    return table_all
```

Question: What does the code extend?
Answer: a list of simple tables
Seq2seq: a table
Dual Encoder: a list of 0
Transformer: the tables
CodeBERT: a list of tables

```python
def calc_angle(v1, v2, v3):
    v1 = v1 - v2
    v3 = v3 - v2
    return v1.angle(v3)
```

Question: What does the code calculate?
Answer: the angle between 3 vectors representing 3 connected points
Seq2seq: the angle of the v1
Dual Encoder: the v2
Transformer: the angle angle of an error angle
CodeBERT: the angle between two numbers

Table 12: Qualitative examples of different models' performance on Python dataset.