CodeQueries: A Dataset of Semantic Queries over Code

Surya Prakash Sahu[*]
Indian Institute of Science
India
suryaprakash@iisc.ac.in

Madhurima Mandal[†]
Indian Institute of Science
India
madhurimam@iisc.ac.in

Shikhar Bharadwaj[‡]
Indian Institute of Science
India
shikharb@alum.iisc.ac.in

Aditya Kanade
Microsoft Research
India
kanadeaditya@microsoft.com

Petros Maniatis
Google DeepMind
USA
maniatis@google.com

Shirish Shevade
Indian Institute of Science
India
shirish@iisc.ac.in

ABSTRACT

Developers often have questions about semantic aspects of code they are working on, e.g., “Is there a class whose parent classes declare a conflicting attribute?”. Answering them requires understanding code semantics such as attributes and inheritance relation of classes. An answer to such a question should identify code spans constituting the answer (e.g., the declaration of the subclass) as well as supporting facts (e.g., the definitions of the conflicting attributes). The existing work on question-answering over code has considered yes/no questions or method-level context. We contribute a labeled dataset, called CodeQueries, of semantic queries over Python code. Compared to the existing datasets, in CodeQueries, the queries are about code semantics, the context is file level and the answers are code spans. We curate the dataset based on queries supported by a widely-used static analysis tool, CodeQL, and include both positive and negative examples, and queries requiring single-hop and multi-hop reasoning.

To assess the value of our dataset, we evaluate baseline neural approaches. We study a large language model (GPT3.5-Turbo) in zero-shot and few-shot settings on a subset of CodeQueries. We also evaluate a BERT style model (CuBERT) with fine-tuning. We find that these models achieve limited success on CodeQueries. CodeQueries is thus a challenging dataset to test the ability of neural models, to understand code semantics, in the extractive question-answering setting.

CCS CONCEPTS

• **Software and its engineering** → **Software post-development issues**; • **Computing methodologies** → *Artificial intelligence.*

KEYWORDS

Code understanding, developer productivity, neural modeling, extractive question-answering

ACM Reference Format:
Surya Prakash Sahu, Madhurima Mandal, Shikhar Bharadwaj, Aditya Kanade, Petros Maniatis, and Shirish Shevade. 2024. CodeQueries: A Dataset of Semantic Queries over Code. In *17th Innovations in Software Engineering Conference (ISEC 2024), February 22–24, 2024, Bangalore, India.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3641399.3641408

1 INTRODUCTION

Extractive question-answering in natural-language settings is a venerable domain of NLP, requiring detailed reasoning about a single reasoning step (“single hop” [39]) or multiple reasoning steps (“multi-hop” [48]). In the context of programming languages, neural question answering over code has not grown to similar complexity: tasks are either binary yes/no questions [20] or range over a localized context (e.g., a source-code method) [2, 26].

Recent results show promise towards neural program analyses around complex concepts such as program invariants [43, 44], inter-procedural properties [10], and even evidence of deeper semantic meaning [22]. However, there do not exist semantically rich question-answering datasets requiring reasoning over code, especially for questions with large scope (entire files) and high complexity (e.g., multi-hop reasoning). Also, given the criticality of program analysis, it is pertinent to judge neural approaches not only on the answer to a question, but also on the reasoning or evidence for that answer.

In this work, we set out to build a labeled dataset, called *CodeQueries*[1], for extractive question-answering over code. The queries are described in English and the context is provided by the contents of a source-code file. If a file does not contain code spans matching the queried pattern then the answer spans is an empty set. These are *negative examples. Positive examples* provide *answer spans* in the file. Some queries require reasoning about multiple facts. For them, the *supporting facts* are also identified as code spans in the file. As an example, consider a query about existence of “conflicting attributes in base classes”. Figure 1 shows a positive example labeled with answer and supporting-fact spans. The subclass `ThreadedTCPServer` inherits from the two base classes `ThreadingMixin` and `TCPServer`, both of which define the method `acceptConnection`. Since both superclasses define the same method, there is a conflict in resolving the method `acceptConnection` invoked on instances of `ThreadedTCPServer`. As shown in the figure, the declaration of the subclass constitutes the answer span and the declarations of the conflicting attribute in the superclasses constitute supporting facts.

[*] Now at Observe.
[†] Now at Myntra.
[‡] Now at Google Research.

[1] https://huggingface.co/datasets/thepurpleowl/codequeries

ISEC 2024, February 22–24, 2024, Bangalore, India

ACM ISBN 979-8-4007-1767-3/24/02.
https://doi.org/10.1145/3641399.3641408

```python
class TCPServer:
    def __init__(self, service, ...): ...

    # Supporting Fact 1
    def acceptConnection(self, conn): ...

    def handleConnection(self, conn): ...

class ThreadingMixin:
    # Supporting Fact 2
    def acceptConnection(self, conn): ...

# Answer Span
class ThreadedTCPServer(ThreadingMixin, TCPServer):
    pass
```

Figure 1: Example code labeled with the answer and supporting-fact spans for the conflicting-attributes query.

There are two difficulties in constructing such a dataset: 1) identifying semantic queries that are representative of developers' requirements and 2) deriving labels. We overcome these difficulties by basing our dataset creation on queries supported by a widely-used static analysis tool, CodeQL[2] [1]. We identify 52 public CodeQL queries that produce highest number of answers on files in a common corpus of Python code [41]. Each CodeQL query identifies a semantic aspect of code related to correctness, reliability, maintainability or security of code through program analysis. Among the 52 queries, 15 require *multi-hop reasoning* and 37 require *single-hop reasoning*. For instance, the example in Figure 1 requires multi-hop reasoning across three classes.

Each CodeQL query is evaluated by the CodeQL engine on a relational representation of code (similar to how a database query is evaluated by a database engine). We extract answer and supporting-fact spans from the analysis results. Since there can be multiple files in the corpus with code that matches a query, we can gather multiple positive examples per query; e.g., several instances of conflicting attributes from different source-code files. We also include code on which the queries do not return any answer spans (negative examples) so that a model can learn to predict when the code does not have the queried pattern (e.g., absence of a buggy code pattern). These are analogous to the no-answer [9] or unanswerable scenarios [38]. The English descriptions of the CodeQL queries, provided in the CodeQL documentation, are used in the natural-language queries in our dataset. For example, the "conflicting attributes in base classes" query[3] is of the form "When a class subclasses multiple base classes, attribute lookup is performed from left to right amongst the base classes. ... this means that if more than one base class defines the same attribute ... may not be the desired behavior ...". Thus, a neural model will be required to analyze code semantics from the analysis intent described in natural language. Figure 2 shows the data preparation setup. CodeQueries contains 34,662 positive examples and 52,613 negative examples.

To assess the value of our dataset, we consider various baseline neural approaches, varying in architectural choices, evaluation methods and the presence of supporting facts. Specifically, we study the ability of a large language model (GPT3.5-Turbo), that has seen extensive natural language and code, to answer semantic queries with various amounts of prompting on a subset of CodeQueries. We also study a much smaller but more custom model, fine-tuned from CuBERT [23].

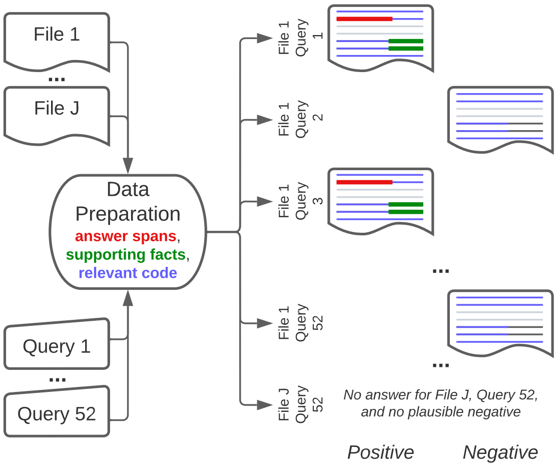


Figure 2: Methodology for preparing the CodeQueries dataset. All source-code files are analyzed against each of the 52 CodeQL queries to gather multiple positive and negative examples for that query. We derive answer spans, supporting-fact spans and code relevant for answering the query for each example. The details are discussed in Section 3.

We find that these models achieve limited success on CodeQueries. With zero-shot prompting, GPT3.5-Turbo achieves exact match with the ground-truth answer spans (within pass@10) on 20.84% of positive examples and detects that 26.77% negative examples do not contain answer spans. The model performance increases to 32.66% and 70.08% respectively when prompted with few-shot examples. The CuBERT model when fine-tuned with limited data achieves exact match on only 3.74% positive examples. CodeQueries is thus a challenging dataset that can be used for evaluating current and future neural approaches, on their ability to understand code semantics, in the extractive question-answering setting. It can further help understand opportunities to improve model performance. We have released our code, data and model checkpoints to facilitate future work on the proposed problem of answering semantic queries over code at https://github.com/thepurpleowl/codequeries-benchmark.

2 RELATED WORK

Natural-language questions and queries about code. CoSQA [20] includes yes/no questions to determine whether a web search query and a method match. Bansal et al. [2] and CodeQA [26] are two recent works on question-answering over code. Both consider a method as the code context, and programmatically extract question-answer pairs specific to the method from the method body and comments. Bansal et al. [2] generate questions about method signatures (e.g., what are parameter types), (mis)matches between a function

[2] https://codeql.github.com/
[3] https://codeql.github.com/codeql-query-help/python/py-conflicting-attributes/

and a docstring, and function summaries. CodeQA is generated from code comments using rule-based templates. The answers are natural-language sentences extracted from code comments. The context in our case is larger, file-level; queries are about semantic aspects of code and may require long chains of reasoning; and answers are spans over code. CS1QA [24] is a dataset of question-answering in an introductory programming course and proposes classification of the question into pre-defined types, identification of relevant source-code lines and retrieval of related QAs. In an orthogonal direction, natural language queries have been used for code retrieval [6, 15, 16, 19, 21, 49].

Learning-based program analysis. Use of program analysis helps improve software quality. However, implementing analysis algorithms requires expertise and efforts. There is increasing interest in using machine learning for program analysis. Recent work in this direction includes learning program invariants [43, 44], rules for static analysis [4], intra- and inter-procedural data flow analysis [10], specification inference [3, 8], reverse engineering [11], and type inference [18, 28, 31, 35, 36, 46]. These techniques target specific analysis problems, use specialized program representations or customize learning methods. Our work targets semantic queries over code and presents a uniform extractive question-answering setup for them, wherein the developer intent is expressed in natural language. Our queries cover diverse program analyses involving forms of type checking, control-flow and data-flow analyses, and many other checks (see the supplementary document[4]). Pashakhanloo et al. [33, 34] advocate the use of relational representations of code, as used in CodeQL, in neural modeling and use them on classification tasks. GitHub has recently launched an experimental service[5] that uses feature-based machine learning to classify JavaScript and TypeScript code with regards to four common vulnerabilities.

Question-answering over text. Various datasets for extractive question-answering over text requiring single-hop [39] and multi-hop [48] reasoning have been proposed. Our dataset consists of queries requiring single- and multi-hop reasoning over code. Along the lines of prior work [9, 38], we include negative examples in which the queries cannot be answered with the given context, though the context contains plausible answers [48]. In order to enable explainability, we include supporting facts [48] in our dataset. We experiment on file-level code which may contain parts that are not relevant to the query. This is analogous to distractor paragraphs [48] and requires the models to deal with spurious information.

3 DATASET PREPARATION

In this section, we describe our methodology for dataset preparation. An example in our dataset is a tuple (Q, C, A, SF) where Q is a query, C is the contents of a Python file, A is the set of answer spans (i.e., code fragments of C that constitute the answer) and SF is the set of supporting-fact spans.

Single-hop and multi-hop queries. We evaluated the queries (formalized in the CodeQL query language) from a standard suite of CodeQL [37] on the redistributable subset [23] of the ETH Py150 dataset of Python code [41] (the ETH Py150 Open dataset). These queries are written by experts and identify coding issues pertaining to correctness, reliability, maintainability or security of code. We evaluated each query on individual Python files (Figure 2). To get a reasonable number of positive examples for each query, we selected queries with at least 50 answer spans in the training split of the ETH Py150 Open dataset. We inspected the definition[6] of a query to check whether answering it requires a single reasoning step or multiple reasoning steps, and classified the query accordingly as a *single-hop* or *multi-hop* query. Out of the 52 queries, 15 are multi-hop and 37 are single-hop. We call these *positive queries.* Note that the formal CodeQL queries are used only for preparing the dataset. We use the English description of a query as the corresponding natural-language query in our dataset.

Positive and negative examples. By evaluating a positive query, we identify files containing code spans that satisfy the query definition. These are positive examples for the query. Naively, any code on which a query does not return an answer could be viewed as a negative example; for instance, in the case of conflicting attributes (Figure 1), it would be trivial to answer that there are no conflicting attributes if the code does not contain classes. In natural-language question answering, Yang et al. [48] recommend that unanswerable contexts should contain *plausible, but not actual, answers*; otherwise, it is simple to distinguish between answerable and unanswerable contexts [47]. Therefore, we manually derive the CodeQL queries required to obtain negative examples *with plausible answers.* We ensure that a *negative query* identifies code similar to the original (positive) query but which does not satisfy the key properties required for producing an answer to the original query. For example, the negated version of the conflicting-attributes query finds code containing a class with multiple inheritance (similar to Figure 1) such that the base classes do *not* have conflicting attributes. Suppose `hasMultipleInheritance(c,p1,p2)` and `haveConflict(p1,p2)` respectively identify a subclass `c` with two parent classes `p1` and `p2`, and check if they have conflicting attributes. The positive query will have `hasMultipleInheritance(c,p1,p2)` and `haveConflict(p1,p2)`, whereas the negative query will have `hasMultipleInheritance(c,p1,p2)` and `not haveConflict(p1,p2)`. Using results of the negative queries, we derive negative examples. While the positive queries are already available publicly, we are releasing the negative queries.

Answer and supporting-fact spans. We identify the answer and supporting-fact spans from the results produced by the CodeQL engine for each of the positive queries. These spans are of a variety of syntactic patterns, making it non-trivial for a model to identify the right candidates for answering the queries. In all, there are 42 different syntactic patterns of spans such as class declarations, `with` statements, and list comprehensions. We give the statistics of syntactic patterns of spans in the supplementary document[4]. Note that negative examples do not have answer or supporting-fact spans.

Dataset statistics. Table 1 gives the dataset statistics according to the splits of the ETH Py150 Open dataset. We place an example

[4] https://github.com/thepurpleowl/codequeries-benchmark/blob/main/Codequeries_Statistics.pdf

[5] https://github.blog/2022-02-17-code-scanning-finds-vulnerabilities-using-machine-learning/

[6] https://codeql.github.com/codeql-query-help/python/

Table 1: Dataset statistics.

		Train	Validation	Test
Positive	Min	34	2	14
	Max	11,490	1,249	6,439
	Total	**20,783**	**2,319**	**11,560**
Negative	Min	29	1	17
	Max	17,592	1,893	9,892
	Total	**31,676**	**3,464**	**17,473**

derived from a Python file in the same split as the file. The Min/-Max entries give the number of minimum/maximum examples over individual queries, whereas Total is the sum of examples across all queries. We observed that the query to identify "unused imports" produced maximum examples. We report additional dataset statistics in the supplementary document[7].

Relevant code blocks. A CodeQL query produces answers based only on specific parts of code within a file, e.g., a set of classes within the file or a set of methods within a class in the file. We inspect the query definitions[6] and automate extraction of the query-relevant parts from a file. Given the query results, we programmatically obtain the code blocks needed for arriving at the same results for the query. We call them *relevant code blocks.* A code block is either a method, all class-level statements (such as attribute definitions) within a class or module-level statements that do not belong to any class or method. In Section 4.2, we describe how this information is used to help the CuBERT model scale to large files by filtering out irrelevant code blocks using a classifier. Note that we implement the static analysis to identify relevant blocks only for purposes of generating labeled training data. However, in practice, developers can provide such data manually without having to implement such analysis.

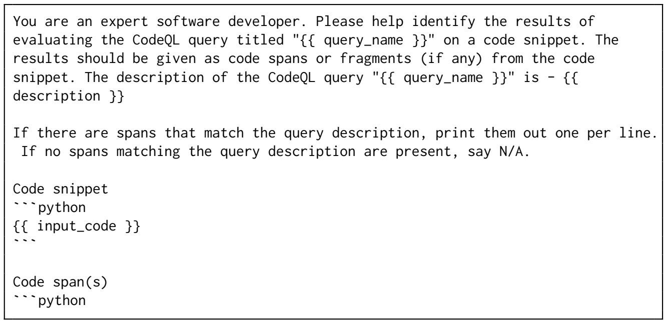
4 EXPERIMENT DESIGN

CodeQueries is intended as a dataset to analyze semantic understanding of neural models through extractive question-answering over code. In this work, we evaluate a large language model (LLM) with prompting and a contextual embedding model with fine-tuning, to assess the difficulty level of our dataset. A full-scale benchmarking of the existing models is *not* an objective of this work.

4.1 Prompting a Large Language Model

Large language models [7, 14, 25, 29, 30, 45] have shown impressive ability on coding tasks and are capable of zero-shot and few-shot inference [5]. We use the GPT3.5-Turbo model [30] from OpenAI in different settings described below. The complete prompt templates are provided in the supplementary material.

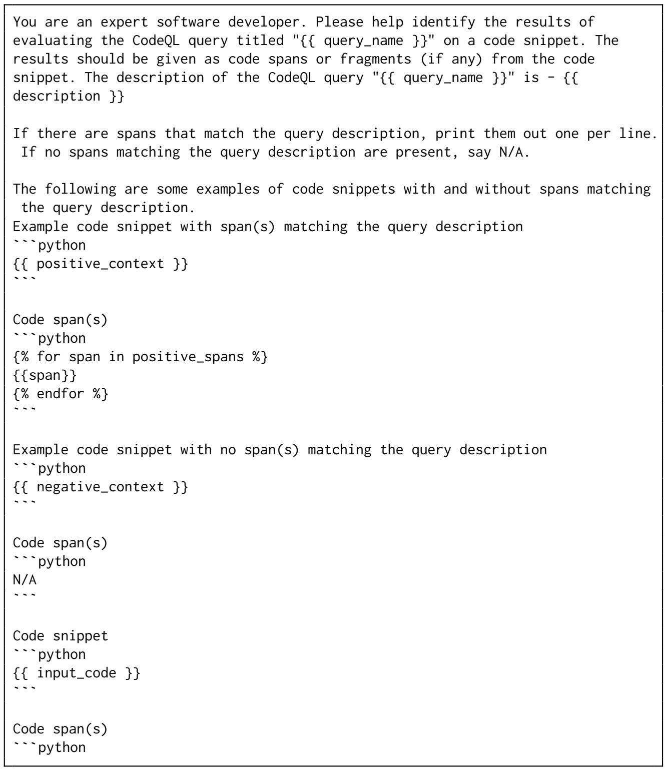
Zero-shot prompting. In this setting, we provide the name of the CodeQL query and its English description, both taken from the CodeQL documentation, and instruct to output answer spans for given code. We require the model to output "N/A" if it judges that the code does not have an answer. The contents of a file are provided as the code to be analyzed. The prompt template is presented in Figure 3.



~~~
You are an expert software developer. Please help identify the results of
evaluating the CodeQL query titled "{{ query_name }}" on a code snippet. The
results should be given as code spans or fragments (if any) from the code
snippet. The description of the CodeQL query "{{ query_name }}" is - {{
description }}

If there are spans that match the query description, print them out one per line.
 If no spans matching the query description are present, say N/A.

Code snippet
```python
{{ input_code }}
~~~

Code span(s)
```python
```

Figure 3: Zero-shot prompt template.



~~~
You are an expert software developer. Please help identify the results of
evaluating the CodeQL query titled "{{ query_name }}" on a code snippet. The
results should be given as code spans or fragments (if any) from the code
snippet. The description of the CodeQL query "{{ query_name }}" is - {{
description }}

If there are spans that match the query description, print them out one per line.
 If no spans matching the query description are present, say N/A.

The following are some examples of code snippets with and without spans matching
 the query description.
Example code snippet with span(s) matching the query description
```python
{{ positive_context }}
~~~

Code span(s)
```python
{% for span in positive_spans %}
{{span}}
{% endfor %}
```

Example code snippet with no span(s) matching the query description
```python
{{ negative_context }}
```

Code span(s)
```python
N/A
```

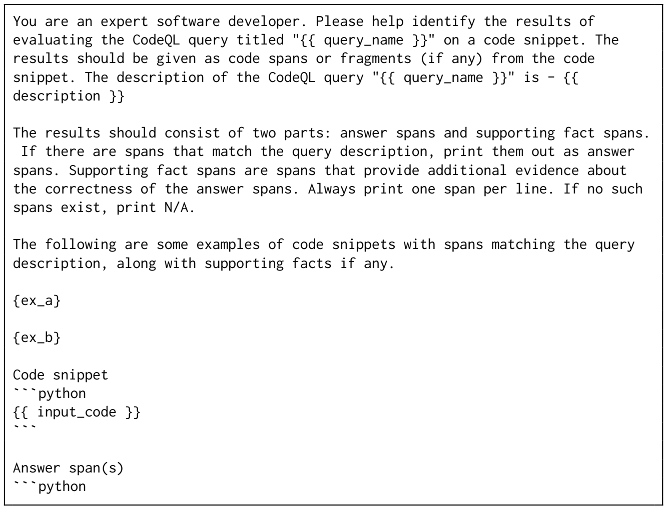
Code snippet
```python
{{ input_code }}
```

Code span(s)
```python
```

Figure 4: Few-shot prompt template with BM25 retrieval.

Few-shot prompting with BM25 retrieval. We provide the same instructions to the model as in the zero-shot prompting, but in addition, include a positive and a negative labeled example in the prompt. For a query Q, we retrieve labeled examples for Q from the training split that are similar to the code to be analyzed, using the BM25 method [42]. In addition to term frequency and inverse document frequency, BM25 considers the ratio between term occurrences and the overall document length, enabling the retrieval of code examples where the snippets from the input code are more prominent. The prompt template is presented in Figure 4. Similar to the zero-shot setting, we require the model to output the answer spans or "N/A". To ensure that we do not overflow the prompt,

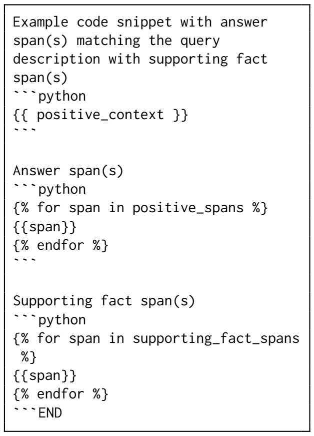
[7] https://github.com/thepurpleowl/codequeries-benchmark/blob/main/Codequeries_Statistics.pdf



~~~
You are an expert software developer. Please help identify the results of
evaluating the CodeQL query titled "{{ query_name }}" on a code snippet. The
results should be given as code spans or fragments (if any) from the code
snippet. The description of the CodeQL query "{{ query_name }}" is - {{
description }}

The results should consist of two parts: answer spans and supporting fact spans.
 If there are spans that match the query description, print them out as answer
spans. Supporting fact spans are spans that provide additional evidence about
the correctness of the answer spans. Always print one span per line. If no such
spans exist, print N/A.

The following are some examples of code snippets with spans matching the query
description, along with supporting facts if any.

{ex_a}

{ex_b}

Code snippet
```python
{{ input_code }}
~~~

Answer span(s)
```python
```

Figure 5: Few-shot prompt template with supporting facts.

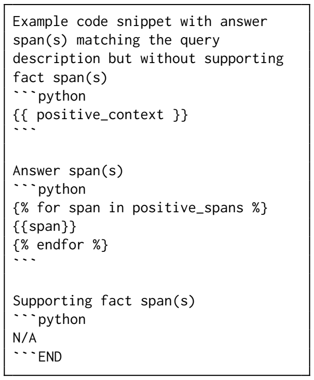


~~~
Example code snippet with answer
span(s) matching the query
description with supporting fact
span(s)
```python
{{ positive_context }}
~~~

Answer span(s)
```python
{% for span in positive_spans %}
{{span}}
{% endfor %}
```

Supporting fact span(s)
~~~python
{% for span in supporting_fact_spans
 %}
{{span}}
{% endfor %}
```END
~~~

Figure 6: "ex_a" sub-template in few-shot prompt with supporting facts.



~~~
Example code snippet with answer
span(s) matching the query
description but without supporting
fact span(s)
```python
{{ positive_context }}
~~~

Answer span(s)
```python
{% for span in positive_spans %}
{{span}}
{% endfor %}
```

Supporting fact span(s)
~~~python
N/A
```END
~~~

Figure 7: "ex_b" sub-template in few-shot prompt without supporting facts.

we minimize the examples by keeping only code blocks that are relevant to the query (see Section 3, relevant code blocks). This optimization is used in the next setting as well.

Few-shot prompting with supporting facts. As discussed in Section 3, we extract supporting facts from the CodeQL results. In this setting, we evaluate the ability of the LLM to produce both answer and supporting-fact spans. In CodeQueries, only positive examples have answer and supporting facts, and therefore this setting is applicable only to the positive examples. The answers to some queries can be determined through local reasoning and they do not have additional supporting facts. Our prompt provides instructions to produce answer and supporting facts, and an example with answer and supporting-fact spans. For examples without supporting facts, we mark supporting facts as "N/A". The prompt template is presented in Figure 5.

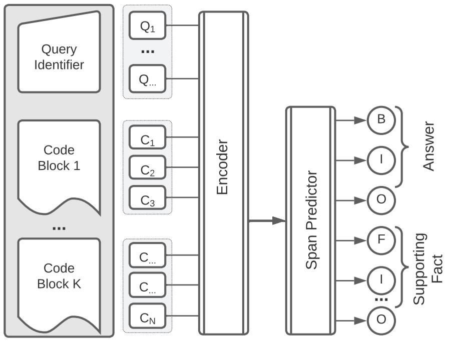


Figure 8: The span prediction setup.

4.2 Fine-tuning a Contextual Embedding Model

Span prediction problem. We reformulate the extractive question-answering problem as a problem of classifying code tokens. Let $\{B, I, O\}$ respectively indicate **B**egin, **I**nside and **O**utside labels [40]. An answer span is represented by a sequence of labels such that the first token of the answer span is labeled by a B and all the other tokens in the span are labeled by I's. We use an analogous encoding for supporting-fact spans, but we use the F label instead of B to distinguish facts from answers. Any token that does not belong to a span is labeled by an O. We thus represent multiple answer or supporting-fact spans by a single sequence over $\{B, I, O, F\}$ labels. We call this the *span prediction problem.* Note that this does not allow overlap between spans, which we have empirically found not to be a problem in our dataset.

Model selection. We can fine-tune BERT-style, encoder-based contextual models [13, 17, 23] to solve the span prediction problem. These models come with different input size restrictions. For CuBERT and CodeBERT, checkpoints are available for input length of 512. For CuBERT, a checkpoint for input length of 1024 is also available. The CuBERT checkpoint with input length of 1024 is denoted as CuBERT-1K. The GraphCodeBERT model allocates input length of 512 for code tokens and 128 for data-flow graph nodes. We use all these available checkpoints for experimentation. As a non-pre-trained baseline, we train a Transformer encoder with input length of 1024 from scratch. We used the CuBERT vocabulary for this Transformer encoder but trained the token embeddings in an end-to-end manner. We use hyper-parameters and configurations discussed in *Training setup* of this section. While it is possible to evaluate other models, it is not the primary focus of this paper. Hence, in our experiments, we use the best-performing (see Section 5.2) CuBERT model.

Span prediction model. Figure 8 shows the span prediction setup. The input to the model is the unique name of a query (marked as query identifier in the figure) and the code. The whole sequence is preceded with the [CLS] token, similar to BERT [12]. The symbols Q_i and C_j denote subword tokens of the query identifier and code, respectively. For simplicity, we do not explicitly show the special delimiter tokens such as [CLS]. The input sequence is fed to the encoder. The span prediction layer consists of a token classifier that performs a four-way classification over the labels $\{B, I, O, F\}$. It is

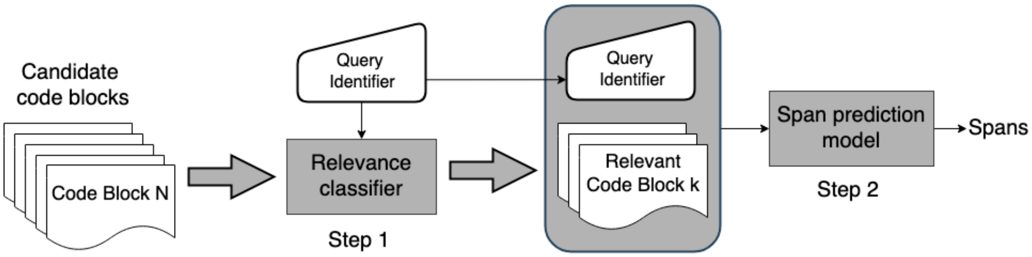


Figure 9: Two-step procedure to handle large-size code containing possibly irrelevant code blocks. In step 2, the span prediction model follows the approach illustrated in Figure 8.

applied to the encoding of every code token in the last layer of the encoder. For negative examples, all tokens are to be classified as O.

Two-step procedure of relevance classification and span prediction. We found that not all code is relevant for answering a given query. Additionally, in many cases, the entire file contents do not fit in the input to the model. As discussed in Section 3, we identify the relevant code blocks programmatically using the CodeQL results during data preparation. We use this information to devise a two-step procedure (see Figure 9) to deal with the problem of scaling to large-size code:

Step 1: We first apply a *relevance classifier* to every block in the given code and select code blocks that are likely to be relevant for answering a given query.

Step 2: We then apply the span prediction model (Figure 8) to the set of selected code blocks to predict answer and supporting-fact spans.

Training: Let F be a file and R be the set of code blocks in F that are relevant for a query Q. Other blocks in F are irrelevant. We train a classifier that given Q and a code block b predicts whether b is relevant or not. We fine-tune a CuBERT checkpoint as the relevance classifier. Instead of training the span prediction model on the entire contents of a file F, we train it on code blocks relevant for Q within F. The code blocks identified as relevant during data preparation are used for training. We fine-tune the models by minimizing the cross-entropy loss.

Training setup: The pre-trained CuBERT encoder model checkpoints are available for input length of 512 and 1024. We use the 1024-length checkpoint for span prediction and the 512-length checkpoint for relevance classification. For span prediction, the token encodings from the final hidden layer of an encoder are passed through a dropout layer with a dropout probability of 0.1 followed by a classification layer. We initially experimented with up to 10 epochs and learning rates in the order of e-5 and e-6 for these models. We observed that the models reached minimum validation loss with the following configurations and used them. Fine-tuning is performed for 5 epochs for the 512-length models and for 3 epochs for the 1024-length models, with a learning rate of 3e-5. Based on the memory constraints, we used batch sizes of 4 and 16 for sequence lengths 1024 and 512 respectively. All the models are trained by minimizing the cross-entropy loss using the AdamW optimizer [27] and linear scheduling without any warmup. The best checkpoint is decided based on least validation loss. We used the same hyper-parameters for fine-tuning the CuBERT 1024 span prediction model with a limited number of files (Section 5.2). For the relevance classification model, we fine-tuned the pre-trained CuBERT model with input length limit of 512. The pooled output is passed through a dropout layer with dropout probability of 0.1 and a 2-layer classifier with a hidden dimension of 2048. We fine-tuned it for 5 epochs with a learning rate of 3e-6 and used weighted crossentropy (with weights 1/2 for irrelevant/relevant class) as the loss function. The best checkpoint is decided based on the least validation loss. We used the same hyper-parameters except for the learning rate (2e-6) for fine-tuning the CuBERT 512 relevance classification model with a limited number of files (Section 5.2).

Compute: All experiments are performed on a 64 bit Debian system with an NVIDIA Tesla A100 GPU having 40GB GPU memory and 85GB RAM.

Inference: At inference time, given a query Q and a file comprising code blocks $\{b_1, \ldots, b_n\}$, we generate a set of n examples by concatenating Q and the contents of each of b_i. The relevance classifier is applied on each of these examples and all blocks classified as relevant are selected. The selected blocks and the query are passed to the span prediction model as shown in Figure 9.

4.3 Evaluation Metrics

We measure the performance of the model in terms of *exact match*. An exact match occurs when the set of predicted answer spans is same as the set of ground-truth answer spans. When supporting facts are predicted, the exact match also requires that the set of predicted supporting-fact spans is same as the set of ground-truth supporting-fact spans. For a relevance classification model, we measure the usual classification metrics: accuracy, precision, and recall.

5 EXPERIMENTAL RESULTS

5.1 Evaluation of the LLM with Zero-shot and Few-shot Prompting

Sampled test data. Due to a limited inference budget, we evaluate the LLM (GPT3.5-Turbo) on a sample of the test split. Considering the available prompt size of 4096 tokens in the used LLM, we sampled files that can fit into the input along with the examples of few-shot prompts, i.e., files having less than 2000 tokens are considered. For each of the 52 queries, we select a maximum of 20 test files with 10 each from positive and negative examples. We refer to this as the *sampled test data*.

Table 2: Percentage exact match achieved by GPT3.5-Turbo on the sampled test data.

(a) Zero-shot prompting and few-shot prompting with BM25 retrieval for answer span prediction.

	Zero-shot prompting		Few-shot prompting with BM25 retrieval	
Pass@k	Positive	Negative	Positive	Negative
1	9.82	12.83	**16.45**	**44.25**
2	13.06	17.42	**21.14**	**55.53**
5	17.47	22.85	**27.69**	**65.43**
10	20.84	26.77	**32.66**	**70.08**

(b) Few-shot prompting with supporting facts for answer and supporting-fact span prediction.

	Few-shot prompting with supporting facts
Pass@k	Positive
1	21.88
2	28.06
5	34.94
10	39.08

Results on the sampled test data. We experiment on the sampled test data with various prompts and obtain 10 generations at temperature of 0.8 per inference. We use the *pass@k* measure [7] for k draws from n generations, for $k \in \{1, 2, 5, 10\}$ and $n = 10$.

Table 2a shows the results of zero-shot prompting and few-shot prompting with BM25 retrieval for answer span prediction. In zero-shot prompting, the LLM gets only 9.82% and 12.83% exact match on positive and negative examples respectively with pass@1. For $k = 10$, these increase to 20.84% and 26.77% respectively. The few-shot prompting shows improvement over zero-shot prompting at all values of k. The improvement on negative examples is particularly significant. We believe that this is because both a positive and a negative example are provided in the prompt. The negative example has a plausible but incorrect candidate answer (see Section 3). The difference in the two examples helps the LLM detect the negative examples more accurately.

Table 2b shows the results of few-shot prompting with supporting facts on answer and supporting-fact span prediction. As discussed in Section 4.1, this setting is applicable only to positive examples. We see that the LLM achieves exact match of 21.88%–39.08% for different values of k. Note that for the experiment in Table 2a, the model is required to distinguish between positive and negative examples, which is not the case in this setting. The additional annotation of supporting facts in the examples in the prompt seems to help the model in predicting both answers and supporting facts.

Observations. With zero-shot prompting, the LLM was able to identify correct spans in positive examples for simple queries, e.g., 80% exact match for the query “Flask app is run in debug mode”, but achieved no exact match on complex queries like “Inconsistent equality and hashing”. It faces similar problems with the negative examples. Some of these failure cases are fixed with few-shot prompting where explicit spans of positive/negative examples in the prompt provide additional information about the intent and differences between positive/negative examples. For many queries including “Inconsistent equality and hashing”, few-shot prompts having examples with supporting facts are able to generate correct answer spans along with the correct supporting facts. As general observations, for both single-hop and multi-hop queries, we see shorter and more accurate code generation with few-shot prompts compared to zero-shot prompts.

5.2 Evaluation of the Fine-tuned Contextual Embedding Models

Model selection. In Table 4, we see all the models have excellent exact-match accuracy for the negative examples; meaning that they are successful in identifying unanswerable contexts. On the positive examples, the finetuned models achieve accuracy in the range of 59.77−72.51%. Better performance of GraphCodeBERT suggests additional information provided with data-flow nodes tokens helps. Predicting spans for positive examples requires accurately identifying both the beginning token and all the other tokens that form the span, whereas for a negative example it suffices to predict that no token belongs to a span. We believe that the relative gap in the performance of the models between positive and negative examples stems from this difference. We also provide query-wise performance of best preforming CuBERT-1K model in the supplementary document.

Evaluation setup. We fine-tune the relevance classification and span prediction models from the pre-trained CuBERT checkpoints

Table 3: Percentage exact match achieved by the models fine-tuned from CuBERT.

(a) Answer and supporting-fact span prediction on the complete test data.

Variants	Positive	Negative
Two-step(20, 20)	3.74	95.54
Two-step(all, 20)	7.81	97.87
Two-step(20, all)	33.41	96.23
Two-step(all, all)	**52.61**	**96.73**
Prefix	36.60	93.80
Sliding window	51.91	85.75

(b) Results on the sampled test data from Section 5.1.

	Answer span prediction		Answer & supporting-fact span prediction
Variants	Positive	Negative	Positive
Two-step(20, 20)	9.42	92.13	8.42
Two-step(all, 20)	15.03	94.49	13.27
Two-step(20, all)	32.87	96.26	30.66
Two-step(all, all)	51.90	95.67	49.30

Table 4: Percentage exact match achieved by the considered fine-tuned models.

Models	Positive	Negative
Transformer	22.50	**97.57**
CuBERT	59.77	97.38
CodeBERT	62.67	95.96
GraphCodeBERT	61.08	97.40
CuBERT-1K	**72.51**	96.79

for 512 and 1024 token lengths respectively. Each of them is trained jointly on all 52 queries (see *Training Setup* in Section 4.2). We train two variants of each of these models: 1) one on *all* files in the training split and 2) another on *10 positive and 10 negative files per query* as a representative of the practical setting in which only a few labeled examples are available. We denote the resultant two-step procedure (classification followed by span prediction) by *two-step*(x, y) indicating that the relevance classifier is trained with x files and the span predictor is trained with y files from the training data, for $x, y \in \{20, \text{all}\}$.

Results on the complete test data. As these models are run locally, we can evaluate them on the complete test data (unlike the LLM). Table 3a gives results of the two-step procedure on the complete test data. The *two-step*(all, all) setup which uses all the training data for both the relevance classification and span prediction performs the best, getting 52.61% and 96.73% exact match on positive and negative examples. However, it relies on existence of a large set of labeled examples for training, which may not be available in practice. The most practical setting, *two-step*$(20, 20)$, is able to get exact match on only 3.74% positive examples. Among the $\{B, I, O, F\}$ labels, the label **O**utside is very frequent compared to the other labels and hence, the token classifier is biased towards predicting it and that explains why the exact match is high for the negative examples in all settings.

The relevance classifier trained with 20 files achieves accuracy, precision, and recall scores of 91.37, 79.72, and 89.61, respectively. Training it with all files increases the scores to 96.38, 95.73, and 90.10 respectively. We evaluated two simple substitutes to relevance classification in the two-step procedure. We considered a *prefix* setup in which the maximum file prefix that can fit the input is selected. Another setup is a *sliding window* setup in which a file is split by the input size of the model into different chunks forming independent examples and the results are aggregated across the chunks. Table 3a shows the results obtained by the span prediction model, trained on *all* data, in conjunction with *prefix*/*sliding window*. We see that *two-step*(all, all) performs better than them.

Results on the sampled test data. Table 3b gives results of the two-step procedure on the sampled test data from Section 5.1. We see that *two-step*$(20, 20)$ has comparable performance to the LLM in pass@1 in zero-shot prompting on answer-span prediction over positive examples (Table 2a). It underperforms the LLM for higher values of k and in few-shot prompting, including for predicting both answer and supporting-fact spans (Table 2b). Increasing the training budget to *all* examples improves the performance of the fine-tuned models. As discussed earlier, the high performance on negative examples is an artifact of the skew in the token labels towards the **O**utside label.

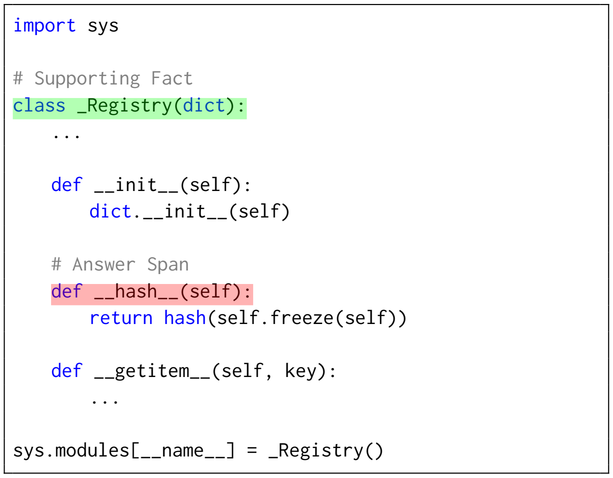
Observations. For some queries like "Imprecise assert" a single file may contain multiple candidate answer spans, e.g., multiple assert statements. With limited training, the relevance classifier had low recall, missing out on some of the relevant candidates. Training with more data allows the relevance classifier to avoid considering irrelevant code blocks as relevant, which can be observed in the significant increase in precision score. For single-hop queries, most of the code blocks in a file would be irrelevant. Training with more data resulted in a significant boost (≥ 10%) in accuracy score for 15 single-hop queries. For some queries such as "Module is imported with 'import' and 'import from"', there is less ambiguity in relevant versus irrelevant blocks and those queries did not benefit much from larger training data.

The span prediction model trained on limited data achieves some success only on a few queries where the answer spans follow specific syntactic patterns, e.g., "Deprecated slice method" whose answer spans contain one of `__getslice__`, `__setslice__` or `__delslice__`. On these queries, training on larger data does not improve the model performance much. In general, the span prediction works better on single-hop queries than multi-hop queries, even when trained on all data.

6 DISCUSSION

6.1 Examples of Successful and Unsuccessful Span Predictions

In this section, we present examples of both successful and unsuccessful predictions of various two-step and LLM prompting setups.



```python
import sys

# Supporting Fact
class _Registry(dict):
    ...

    def __init__(self):
        dict.__init__(self)

    # Answer Span
    def __hash__(self):
        return hash(self.freeze(self))

    def __getitem__(self, key):
        ...

sys.modules[__name__] = _Registry()
```

Figure 10: Positive example code labeled with the answer and supporting-fact spans for the "Inconsistent equality and hashing" query.

Figure 10[8] is a positive example of the multi-hop query "Inconsistent equality and hashing" where the `__hash__` method is

[8] Part of `CenterForOpenScience/scrapi/scrapi/registry.py` file in the ETH Py150 Open dataset

implemented, but __eq__ method is not implemented. Zero-shot prompting fails to generate the answer spans, whereas few-shot prompting with BM25 retrieval and few-shot prompting with supporting facts generate the correct answer span. Among two-step setups, only two-step setups with span prediction models trained with all data, i.e., *two-step*(20, *all*) and *two-step*(*all*, *all*), were able to predict the correct spans.

```python
...

class _AnyLocation:
    ...

# Supporting Fact
class XPathQuery:
    def __init__(self, queryStr):
        ...

    # Answer Span
    def __hash__(self):
        return self.queryStr.__hash__()

__internedQueries = {}
...
```

Figure 11: Positive example code labeled with the answer and supporting-fact spans for the "Inconsistent equality and hashing" query.

```python
...
from helpers import unittest

from luigi.contrib.ssh import RemoteContext

class TestMockedRemoteContext(unittest.TestCase):

    def test_subprocess_delegation(self):
        ...
        # Answer Span 1
        self.assertTrue("ssh" in self.last_test)
        # Answer Span 2
        self.assertTrue("-i" in self.last_test)
        # Answer Span 3
        self.assertTrue(".../key.pub" in self.last_test)
        # Answer Span 4
        self.assertTrue("...@..." in self.last_test)
        # Answer Span 5
        self.assertTrue("ls" in self.last_test)

        subprocess.Popen = orig_Popen

    def test_check_output_fail_connect(self):
        ...
```

Figure 12: Positive example code labeled with the answer spans for the "Imprecise assert" query.

Figure 11[9] is another positive example of the same query, for which all prompting strategies and two-step setups except few-shot prompting with supporting facts failed to predict the answer span.

```python
from __future__ import unicode_literals
...

class TypedChoiceFieldTest(SimpleTestCase):
    ...
    def test_typedchoicefield_5(self):
        ...
        self.assertEqual('', f.clean(''))

    def test_typedchoicefield_6(self):
        ...
        self.assertIsNone(f.clean(''))

    def test_typedchoicefield_has_changed(self):
        ...
        self.assertFalse(f.has_changed(None, ''))
        ...
        self.assertTrue(f.has_changed('', 'a'))
        ...
```

Figure 13: Negative example code for the "Imprecise assert" query.

```python
from __future__ import unicode_literals
...

class Indent(object):
  ...
  def __init__(self, type, size):
    ...

  def __hash__(self):
    return (self.type, self.size).__hash__()

  def __eq__(self, other):
    return hash(self) == hash(other)

  ...

class GherkinParser(object):
  ...

class GherkinFormatter(object):
  ...
```

Figure 14: Negative example code for the "Inconsistent equality and hashing" query.

Figure 12[10] is a positive example of the single-hop query "Imprecise assert". For this example, all prompting strategies, i.e., zero-shot prompting, few-shot prompting with BM25 retrieval, and few-shot prompting with supporting facts, were able to generate the correct

[9] Part of kuri65536/python-for-android/python-modules/twisted/twisted/words/xish/xpath.py file in the ETH Py150 Open dataset

[10] Part of spotify/luigi/test/test_ssh.py file in the ETH Py150 Open dataset.

Table 5: Comparison of exact match and BLEU metric scores on the sampled test data.

(a) Zero-shot prompting and few-shot prompting with BM25 retrieval for answer span prediction.

	Zero-shot prompting		Few-shot prompting with BM25 retrieval	
Pass@k	Positive EM	Positive BLEU	Positive EM	Positive BLEU
1	9.82	18.75	**16.45**	**20.22**
2	13.06	23.74	**21.14**	**25.92**
5	17.47	29.88	**27.69**	**33.68**
10	20.84	34.45	**32.66**	**39.22**

(b) Few-shot prompting with supporting facts for answer and supporting-fact span prediction.

	Few-shot prompting with supporting facts	
Pass@k	Positive EM	Positive BLEU
1	21.88	37.71
2	28.06	46.20
5	34.94	54.59
10	39.08	59.33

(c) Results of fine-tuned model on the sampled test data from Section 5.1.

	Answer span prediction		Answer & supporting-fact span prediction	
Variants	Positive EM	Positive BLEU	Positive EM	Positive BLEU
Two-step(20, 20)	9.42	12.75	8.42	14.73
Two-step(all, 20)	15.03	19.19	13.27	19.46
Two-step(20, all)	32.87	34.25	30.66	35.16
Two-step(all, all)	51.90	54.89	49.30	54.59

answer span. Among two-step setups, only two-step setups with span prediction models trained with all data, i.e., *two-step*(20, *all*) and *two-step*(*all*, *all*), were able to predict the correct spans.

Figure 13[11] is a negative example of the single-hop query "Imprecise assert". For this example, zero-shot prompting fails to generate 'N/A', whereas few-shot prompting with BM25 retrieval was able to generate the 'N/A', denoting the absence of the desired span. Among two-step setups, all setups except *two-step*(20, 20), were able to predict the absence of spans.

Figure 14[12] is a negative example of the multi-hop query "Inconsistent equality and hashing". For this example, zero-shot prompting and few-shot prompting with BM25 retrieval were not able to generate the required 'N/A'. Among two-step setups, all setups except *two-step*((20, 20), were able to predict the absence of any desired answer spans.

6.2 Choice of evaluation metric.

The exact match (EM) metric is commonly used in prior work on natural language extractive QA benchmarks like SQuAD and HotpotQA. Being a strict metric, lower scores with exact match can overshadow overall model performance. Metrics used for evaluating a generated sentence with respect to a reference sentence, such as BLEU [32], can act as a soft replacement for exact match metric. In Table 5, we review the model performance with sampled test data from Section 5.1 by reporting the BLEU scores along with the corresponding exact match scores from Table 2a, Table 2b, and Table 3b. Since there are no ground truth spans for negative examples, BLEU can only be computed for positive examples. We see that BLEU scores correspond well with EM.

7 CONCLUSIONS AND FUTURE WORK

We presented the CodeQueries dataset to test the ability of neural models to understand code semantics on the proposed problem of answering semantic queries over code. Our dataset consists of 52 queries spanning those many distinct program analysis tasks over Python code. It requires a model to perform single- or multi-hop reasoning, understand structure and semantics of code, distinguish between positive and negative examples, and accurately identify answer and supporting-fact spans. We are releasing our data preparation code that can be extended to support more queries and more programming languages. Our evaluation shows that CodeQueries is challenging for the best-in-class generative and embedding approaches under different prompting or fine-tuning settings. We consider file-level context but there is scope to increase it to include entire code repositories. We are considering extensions to our dataset to include more semantic queries and more programming languages.

REFERENCES

[1] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[2] Aakash Bansal, Zachary Eberhart, Lingfei Wu, and Collin McMillan. 2021. A Neural Question Answering System for Basic Questions about Subroutines. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE.

[3] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active Learning of Points-to Specifications. *SIGPLAN Not.* 53, 4 (2018).

[4] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2017. Learning a Static Analyzer from Data. In *Computer Aided Verification - 29th International Conference*. Springer.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] José Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the*

[11] Part of `django/django/tests/forms_tests/field_tests/test_typedchoicefield.py` file in the ETH Py150 Open dataset.

[12] Part of `waynemoore/sublime-gherkin-formatter/lib/gherkin.py` file in the ETH Py150 Open dataset.

Foundations of Software Engineering.

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

[8] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin T. Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.

[9] Christopher Clark and Matt Gardner. 2017. Simple and effective multi-paragraph reading comprehension. *arXiv preprint arXiv:1710.10723* (2017).

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F. P. O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research)*. PMLR.

[11] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics.

[14] Google. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403* (2023).

[15] Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R Lyu. 2021. CRaDLe: Deep code retrieval based on semantic dependency learning. *Neural Networks* 141 (2021), 385–394.

[16] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE*. ACM.

[17] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[18] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery.

[19] Geert Heyman and Tom Van Cutsem. 2020. Neural Code Search Revisited: Enhancing Code Snippet Retrieval through Natural Language Intent. *CoRR* abs/2008.12193 (2020). arXiv:2008.12193

[20] Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20, 000+ Web Queries for Code Search and Question Answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP*. Association for Computational Linguistics.

[21] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436

[22] Charles Jin and Martin Rinard. 2023. Evidence of Meaning in Language Models Trained on Programs. arXiv:2305.11169 [cs.LG]

[23] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning*. PMLR.

[24] Changyoon Lee, Yeon Seonwoo, and Alice Oh. 2022. CS1QA: A Dataset for Assisting Code-based Question Answering in an Introductory Programming Course. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2026–2040.

[25] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[26] Chenxiao Liu and Xiaojun Wan. 2021. CodeQA: A Question Answering Dataset for Source Code Comprehension. In *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics.

[27] Ilya Loshchilov and Frank Hutter. 2017. Fixing Weight Decay Regularization in Adam. *CoRR* abs/1711.05101 (2017). arXiv:1711.05101

[28] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. 2021. Type4Py: Deep Similarity Learning-Based Type Inference for Python. *CoRR* (2021). arXiv:2101.04470

[29] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *arXiv preprint arXiv:2305.02309* (2023).

[30] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL]

[31] Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. 2020. OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints. *CoRR* abs/2004.00348 (2020). arXiv:2004.00348

[32] 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*. Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[33] Pardis Pashakhanloo, Aaditya Naik, Hanjun Dai, Petros Maniatis, and Mayur Naik. 2022. Learning to Walk over Relational Graphs of Source Code. In *Deep Learning for Code Workshop*.

[34] Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. 2021. CodeTrek: Flexible Modeling of Code using an Extensible Relational Representation. In *International Conference on Learning Representations*.

[35] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*. 2019–2030.

[36] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: neural type prediction with search-based validation. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[37] Query Suite. 2022. https://github.com/github/codeql/blob/main/python/ql/src/codeql-suites/python-lgtm.qls.

[38] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know What You Don't Know: Unanswerable Questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

[39] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100, 000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. The Association for Computational Linguistics.

[40] Lance A. Ramshaw and Mitch Marcus. 1995. Text Chunking using Transformation-Based Learning. In *Third Workshop on Very Large Corpora*.

[41] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices* 51, 10 (2016).

[42] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[43] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems*.

[44] Charles Sutton, David Bieber, Kensen Shi, Kexin Pei, and Pengcheng Yin. 2023. Can Large Language Models Reason About Program Invariants?. In *Proceedings of the International Conference on Machine Learning*.

[45] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[46] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations*. OpenReview.net.

[47] Dirk Weissenborn, Georg Wiese, and Laura Seiffe. 2017. Making Neural QA as Simple as Possible but not Simpler. In *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017)*. Association for Computational Linguistics.

[48] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

[49] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*. ACM Press. https://doi.org/10.1145/3178876.3186081