Learning to Answer Semantic Queries over Code

Surya Prakash Sahu
Indian Institute of Science

Madhurima Mandal
Indian Institute of Science

Shikhar Bharadwaj
Indian Institute of Science

Aditya Kanade
Microsoft Research India

Petros Maniatis
Google Brain

Shirish Shevade
Indian Institute of Science

Abstract

During software development, developers need answers to queries about semantic aspects of code. Even though extractive question-answering using neural approaches has been studied widely in natural languages, the problem of answering semantic queries over code using neural networks has not yet been explored. This is mainly because there is no existing dataset with extractive question and answer pairs over code involving complex concepts and long chains of reasoning. We bridge this gap by building a new, curated dataset called CodeQueries, and proposing a neural question-answering methodology over code.

We build upon state-of-the-art pre-trained models of code to predict answer and supporting-fact spans. Given a query and code, only some of the code may be relevant to answer the query. We first experiment under an ideal setting where only the relevant code is given to the model and show that our models do well. We then experiment under three pragmatic considerations: (1) scaling to large-size code, (2) learning from a limited number of examples and (3) robustness to minor syntax errors in code. Our results show that while a neural model can be resilient to minor syntax errors in code, increasing size of code, presence of code that is not relevant to the query, and reduced number of training examples limit the model performance. We are releasing our data and models[1] to facilitate future work on the proposed problem of answering semantic queries over code.

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" [Rajpurkar et al., 2016]) or multiple reasoning steps ("multi-hop" [Yang et al., 2018]). In the context of programming languages, neural question answering has not grown to similar complexity: tasks are either binary yes/no questions [Huang et al., 2021] or range over a localized context (e.g., a source-code method) [Bansal et al., 2021, Liu and Wan, 2021].

Motivated by the recent promise of neural program analyses for learning complex concepts such as loop invariants [Si et al., 2018] and even inter-procedural data flow analysis [Cummins et al., 2021], in this work we study extractive question-answering over code, for questions with a large scope (entire files) and complexity including both single- and multi-hop reasoning. Given the criticality of program analysis, we formulate our problem as one that extracts not only an *answer span*, but also *supporting facts* that elucidate the reasoning behind the answer and render it more interpretable.

Figure 1 shows an illustrative example (compressed for space). We elide some code with "...". The Python module exhibits a buggy behavior: the subclass `ThreadedTCPServiceServer` inherits

[1] `https://github.com/thepurpleowl/codequeries-benchmark`

```
 1 class TCPServiceServer:
 2     def __init__(self, service, ...): ...
 3
 4     def serve(self, address): ...
 5
 6     # Supporting Fact 1
 7     def acceptConnection(self, conn): ...
 8
 9     def handleConnection(self, conn): ...
10
11 class ThreadingMixin:
12     # Supporting Fact 2
13     def acceptConnection(self, conn): ...
14
15 # Answer Span
16 class ThreadedTCPServiceServer(
17     ThreadingMixin, TCPServiceServer):
18     pass
```

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

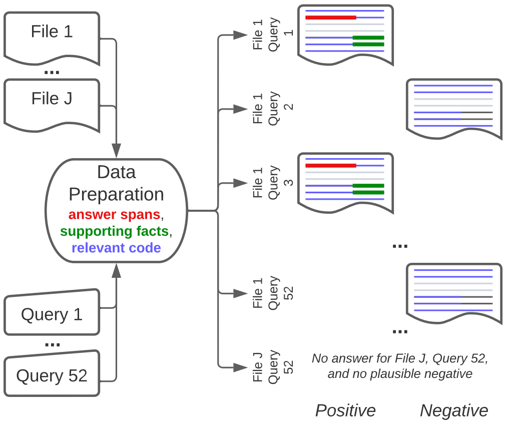


Figure 2: Data preparation setup. All source-code files are analyzed against each of the 52 queries to gather multiple positive and negative examples for that query. We derive answer spans, supporting-fact spans and code relevant for answering the query for each example. The details are discussed in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

from the two base classes `ThreadingMixin` and `TCPServiceServer`, both of which define method `acceptConnection`, which causes a conflict during multiple inheritance. Given this code, a neural model capable of answering the query about “Conflicting attributes in base classes” should point at the declaration of the subclass (lines 16–17) as the answer span, and the conflicting declarations of the `acceptConnection` method at lines 7 and 13 as the supporting facts.

Neural approaches to this problem are challenging, since (1) there are no existing datasets with such complex extractive question and answer pairs; and (2) the underlying program analyses can include complex concepts with long chains of reasoning, spanning multiple methods and classes. For instance, the conflicting-attributes query requires multi-hop reasoning. For the code in Figure 1, it can be answered only by simultaneously reasoning about all the three classes, the inheritance relations among them, and their methods.

We address the first challenge by building a new, curated dataset, CodeQueries. We use an industry-leading symbolic code-query engine and language, CodeQL[2] [Avgustinov et al., 2016], to produce this dataset. Similar to a database engine interpreting a database query, a CodeQL query is interpreted by the CodeQL engine on source code. We run 52 public CodeQL queries on every file in a common corpus of Python code [Raychev et al., 2016]. This gives us *positive examples* comprising a query and code as input, and the answer and supporting-facts as output. 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 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 [Clark and Gardner, 2017] or unanswerable scenarios [Rajpurkar et al., 2018]. We call them *negative examples*. Figure 2 shows the data preparation setup. CodeQueries contains 171,346 examples, among which 71,603 are positive and 99,743 are negative. With this set of 52 queries, the goal is to train a neural model on the train split comprising examples from all the 52 queries and evaluate it to answer the test split examples (over code not seen during training) of those queries. Thus, from training examples of the conflicting-attributes query, the model should learn to identify presence or absence of conflicting attributes in new code.

We address the second challenge by building a neural extractive question-answering methodology using Transformers [Vaswani et al., 2017] to encode input, and span predictors to produce output. On the encoding side, we start with state-of-the-art pre-trained encoders of code (CuBERT [Kanade et al., 2020], CodeBERT [Feng et al., 2020] and GraphCodeBERT [Guo et al., 2020]). On the span-prediction side, we design an output layer that predicts BIO-style tagging [Ramshaw and Marcus,

[2] `https://codeql.github.com/`

1995] (short for **B**egin, **I**nside, **O**utside) over code tokens, to produce answer and supporting-fact spans that can belong to multiple methods and classes. We use different labels to distinguish the beginning of answer spans from the beginning of supporting-fact spans.

With these Transformer-based baselines, we study various pragmatic considerations undergirding a practical solution. First, given a query and code, only some of the code may be relevant. We experiment under an ideal setting where only the relevant code is given to the model at inference time, and our models do well. Our best model identifies answer and supporting-fact spans that match the ground truth on 86.70% examples (72.51% on positive examples and 96.79% on negative examples). However, as we show, given larger code, simple heuristic splitting of the code does not provide good results. We therefore design a two-step procedure in which a classifier predicts which parts of the code are relevant to a query. In the second step, spans are predicted over the relevant parts of the code identified in the first step. The two-step procedure improves over the heuristic methods but is inferior to the ideal setting when only the relevant parts from the large-size code are given to the model. Second, we recognize that, in practice, a developer may have a limited number of labeled examples. We study how robust our methodology is to limited supervision. Third, to further support the practical use of our approach, we study its tolerance to minor syntax errors. The results show that while a neural model can be resilient to minor syntax errors in code, reduced number of training examples limits the model performance.

One may wonder why it is fruitful to study neural approaches to a problem that is symbolically "solved", as evidenced by the existence of frameworks like CodeQL. The answer is twofold. First, while these frameworks provide powerful mechanisms for querying source code, they come with upfront cost, e.g., learning a specialized query language, writing detailed formal queries, understanding helper functions, maintaining the queries if the query language evolves, and making sure that the code is free of even minor syntactic errors. In contrast, our work shows that even without sophisticated analysis techniques, a form of *program analysis by example*, in which developers only supply examples of code labeled with answer and supporting-fact spans for a query, can lead to good learned extractive performance. Second, answers to queries can be obtained even during the software development process, when minor syntactic errors may still exist in code. Symbolic techniques cannot process the input with such errors and fail to provide analysis results.

The main contributions of this work are as follows:

- We propose the problem of answering semantic queries over code. It is grounded in the real-world usage of code-query languages. Solving it requires single-hop or multi-hop reasoning, and understanding structure and semantics of code.
- We prepare a dataset, CodeQueries, which contains 171,346 labeled examples across a diverse set of 52 queries evaluated on Python code.
- We build upon strong pre-trained models of code. We experiment both under an ideal setting and under various pragmatic considerations. Our results show that neural models can be useful for answering code queries but more work is needed to effectively tackle the challenges of learning from fewer examples and scaling to large-size code.
- We have released our data and models to facilitate future work on the proposed problem of answering semantic queries over code.

2 Related Work

Learning-based program analysis. Use of program analysis helps improve developer productivity and 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 loop invariants [Si et al., 2018], rules for static analysis [Bielik et al., 2017], intra- and inter-procedural data flow analysis [Cummins et al., 2021], specification inference [Bastani et al., 2018, Chibotaru et al., 2019], reverse engineering [David et al., 2020], and type inference [Hellendoorn et al., 2018, Pandi et al., 2020, Pradel et al., 2020, Wei et al., 2020, Mir et al., 2021]. These techniques target specific analysis problems with specialized program representations or learning methods. Our work targets semantic queries over code and presents a uniform extractive question-answering setup for them. Our queries cover diverse program analyses involving forms of type checking, control and data flow analysis, among others (see Appendix F of the list of queries). In another line of work, Pashakhanloo et al. [2021, 2022] 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 feature[3] that uses machine learning to classify JavaScript and TypeScript code with regards to four common vulnerabilities. Similar to our work, they built the training set using pre-existing CodeQL queries (written to detect those vulnerabilities). They expect a classifier to catch cases missed by incomplete or stale CodeQL queries. They use relational representation of code built by the CodeQL engine and take help from human experts in feature selection[4]. In contrast, we learn directly on source code. They perform binary classification to surface security alerts, whereas our goal is extractive question-answering to aid developers in code understanding. Despite these technical differences, we share the motivation that machine learning can be used to ease the burden of manually writing or maintaining detailed, formal code queries.

Natural-language questions and queries about code. CoSQA [Huang et al., 2021] includes yes/no questions to determine whether a web search query and a method match. Bansal et al. [2021] and CodeQA [Liu and Wan, 2021] 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. [2021] generate questions about method signatures (e.g., what are parameter and return types), (mis)match between a function and a docstring, and natural-language function summary. CodeQA is generated from code comments using rule-based templates. The answers are natural-language sentences extracted from code comments using NLP techniques. The code in our case can be larger, encompassing multiple methods and classes; queries are about semantic aspects of code and can need long chains of reasoning; and answers are spans over code. In an orthogonal direction, natural language queries have been used for code retrieval [Gu et al., 2018, Husain et al., 2019, Cambronero et al., 2019, Heyman and Cutsem, 2020, Gu et al., 2021].

Question-answering over text. Various datasets for extractive question-answering over text requiring single-hop [Rajpurkar et al., 2016] and multi-hop [Yang et al., 2018] reasoning have been proposed. Our dataset consists of queries requiring single- and multi-hop reasoning over code. Along the lines of [Clark and Gardner, 2017, Rajpurkar et al., 2018], we include negative examples in which the queries cannot be answered with the given context, though the context contains plausible answers [Yang et al., 2018]. For improving explainability, we also include in our dataset and models prediction of supporting facts [Yang et al., 2018]; supporting-fact supervision might also be helpful in alternative chain-of-thought methodologies [Wei et al., 2022]. We experiment on large-size code which may contain parts that are not relevant to the query. This is analogous to distractor paragraphs [Yang et al., 2018] and requires the models to deal with spurious information.

3 Dataset Preparation

Query evaluation. To prepare the CodeQueries dataset, we evaluated the queries from a standard suite of CodeQL [Query Suite] on the open, redistributable subset [Kanade et al., 2020] of the ETH Py150 dataset of Python programs [Raychev et al., 2016] (the ETH Py150 Open dataset). These queries are written by experts and identify coding issues pertaining to correctness, reliability, maintainability and 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. This gave us a suite of 52 queries. The query definitions build upon specialized CodeQL libraries and have 17–689 lines of code with an average of 61 lines. We inspected the definition 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 (see Appendix C for examples). We identify the answer and supporting-fact spans from the results produced by the CodeQL engine for each of the queries. These spans come from a wide 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 Appendix H.

In practice, a developer may want to validate the absence of a pattern (e.g., validate that there is no unused variable). We therefore handle unanswerable scenarios also. We call examples that do

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

[4] `https://github.blog/2022-02-17-leveraging-machine-learning-find-security-vulnerabilities/`

	Training	Validation	Test
Files	74,749	8,302	41,457
1-hop positive examples	29,202	3,216	16,019
N-hop positive examples	13,893	1,528	7,745
1-hop negative examples	40,111	4,244	22,346
N-hop negative examples	19,756	2,195	11,091
Total examples	102,962	11,183	57,201

Table 1: Dataset statistics.

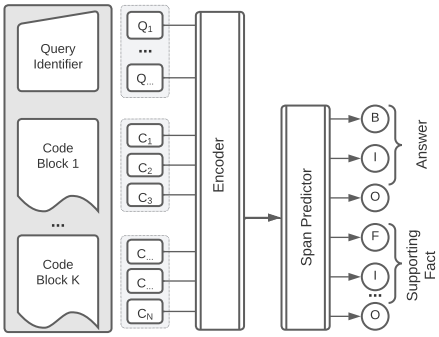


Figure 3: The span prediction setup.

not contain answer spans *negative examples*. 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. [2018] recommend that unanswerable contexts should contain *plausible, but not actual, answers*; otherwise, it is simple to distinguish between answerable and unanswerable contexts [Weissenborn et al., 2017]. Therefore, to obtain negative examples *with plausible answers*, we manually derive logical negations of the CodeQL queries and evaluate them on the ETH Py150 Open dataset. 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 for 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. Using results of the negative queries, we derive negative examples. See Appendix D for examples of positive and negative queries.

Deriving labeled examples. A CodeQL query produces answers based only on specific parts of code, e.g., a single class within a file or a single method within a class. We inspect the query definitions and automate extraction of the query-relevant parts from code. We treat each method as a *code block*. Any code belonging to a class which is not within a method of that class (e.g., declarations of class variables) or any code belonging a file which is not within any class or method (e.g., `import` statements) form separate code blocks. Given the locations of answer and supporting-fact spans for a query, we programmatically obtain the code blocks needed for arriving at the same results for the query. We call them *relevant code blocks*. See Appendix E for additional details.

We represent a *labeled example* in our dataset as a tuple (Q, C, A, SF) where Q is the unique identifier of a query, C is a set of relevant code blocks over which the query is evaluated, A is the set of answer spans over C and SF is the set of supporting-fact spans over C. Each span is a tuple (b, i, j) which identifies the code block $b \in C$ and the start and end indexes, i and j respectively, over tokens in b. By definition, the sets A and SF are empty for negative examples. The query identifier Q is used for uniquely identifying which query among the 52 queries is being applied on C. The formal query definitions are used only for preparing the dataset.

Table 1 gives the statistics of the CodeQueries dataset according to the splits of the ETH Py150 Open dataset. We place the examples derived from a Python file in the same split as the file. We train the neural models on the train split comprising examples from the 52 queries and evaluate them to answer the test split examples of those queries. The table also gives the number of positive and negative examples for single-hop and multi-hop queries (see Appendix F for detailed query-wise statistics).

4 Modeling and Metrics

4.1 Span Prediction Model

Input representation and encoding. Given a query identifier Q and a set of code blocks C, we prepare an input sequence by concatenating Q and the code blocks in C. The code blocks are ordered by their order of appearance in the code files they are extracted from. They are separated by a special `[SEP]` token and the entire sequence is preceded with the `[CLS]` token, similar to BERT [Devlin et al., 2019]. We use the pre-trained CuBERT [Kanade et al., 2020] and CodeBERT [Feng et al., 2020] models as encoders of input sequences. These models use subword vocabularies. Therefore, the input sequences are represented as sequences of subword tokens from the respective vocabularies.

We also use the GraphCodeBERT model [Guo et al., 2020], which additionally embeds data-flow information about code. Our code representation is the same as GraphCodeBERT: code blocks are concatenated and separated by a special delimiter, followed by the sequence of data-flow graph nodes for each of the code blocks; data-flow and variable-alignment edges are represented as attention masks over node sequences and code blocks.

Output representation and span prediction layer. Let $\{B, I, O\}$ respectively indicate **B**egin, **I**nside and **O**utside labels [Ramshaw and Marcus, 1995]. 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 either kind of span is labeled by an O. This allows us to represent multiple answer or supporting-fact spans for the given code blocks in a single sequence over $\{B, I, O, F\}$ labels.

The span prediction layer consists of a token classifier that performs a four-way classification over the labels $\{B, I, O, F\}$. It is applied to the encoding of every code token in the last layer of the encoder. We finetune the model by minimizing the cross-entropy loss. Note that in the case of negative examples, all tokens are to be classified as O. Figure 3 shows the setup for span prediction. 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]` and the data-flow information encoded in the GraphCodeBERT based model.

4.2 Two-step Procedure of Relevance Classification and Span Prediction

As discussed in Section 3, we identify the relevant code blocks programmatically using the CodeQL result during data preparation for an ideal setting in which the model is only invoked on relevant code. However, at inference time on new code, this relevance information is unknown, and a developer may provide large-size code (e.g., an entire file) which could contain code irrelevant to the query. We devise a two-step procedure to deal with this.

Given a query identified by Q and code C comprising code blocks $\{b_1, \ldots, b_n\}$, we generate a set of n examples by concatenating Q and each of b_i. A classifier takes each of the examples and predicts whether b_i is relevant to provide an answer for Q. We call this the *relevance classification problem*. We use the relevant blocks identified as part of the construction of our dataset, along with irrelevant blocks, for training the classifier. We finetune pre-trained models of code for classification. These relevance classifiers are distinct from the span prediction models.

Our two-step procedure to answer a query identified by Q on a large-size code involves first applying a relevance classifier to every block in the given code w.r.t. Q. In the second step, all the blocks classified as relevant in the first step are used as input for the span prediction model.

4.3 Evaluation Metrics

We measure the performance of a span prediction model as the percentage of examples for which the set of predicted answer spans is same as the set of ground-truth answer spans, and the set of predicted supporting-fact spans is same as the set of ground-truth supporting-fact spans. We call this metric the *exact match*. For a relevance classification model, we measure the usual classification metrics.

5 Experimental Results

We use CuBERT, CodeBERT, and GraphCodeBERT as encoders of the input sequences. For all of them, checkpoints for input length 512 are available. GraphCodeBERT allows an additional 128 tokens for data-flow information. For CuBERT, a checkpoint for length of 1024 is also available. We experiment with all of these. For span prediction, the token encodings are followed by a dropout layer and a single-layer classifier. As a non-pre-trained baseline, we train a Transformer encoder from scratch with input of length 1024 for span prediction. For relevance classification, we finetuned the CuBERT model for length 512 with a dropout layer followed by two feedforward layers. We used the AdamW optimizer [Loshchilov and Hutter, 2017] and selected learning rates through initial

Models	**All**	**Positive**	**Negative**
Transformer	66.38	22.50	**97.57**
CuBERT	81.76	59.77	97.38
CodeBERT	82.13	62.67	95.96
GraphCodeBERT	82.31	61.08	97.40
CuBERT-1K	**86.70**	**72.51**	96.79

Table 2: Exact-match results in the ideal setting.

Setting	**All**	**Positive**	**Negative**
Prefix	72.28	36.60	93.80
Sliding window	73.03	51.91	85.75
Two-step	**80.13**	**52.61**	**96.73**
File-level ideal	82.47	59.60	96.26

Table 3: Exact-match results on large-size code.

experimentation. The model checkpoints were selected by the least validation loss. Appendix A provides the complete details of our training setup.

5.1 Experimentation under an Ideal Setting with only Relevant Code

We now evaluate our models on test split examples containing only relevant code; this is the *ideal* evaluation setting since the models do not have to deal with irrelevant code.

Exact-match results. Table 2 reports the exact-match metrics for all the models: Column *All* refers to results on all (both positive and negative) examples, *Positive* and *Negative* refer to results only on positive or negative examples. CuBERT and CuBERT-1K refer to the 512 and 1024 length models respectively. The CuBERT-1K model achieves the maximum exact match of 86.70% on all examples. We use the *best performing model*, CuBERT-1K, to further characterize the dataset and assess model performance. See Appendix I for examples of successful and unsuccessful span predictions.

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%. 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. CuBERT and GraphCodeBERT improve upon CodeBERT over negative examples but at the cost of reduced performance on positive examples. The significant gap between the baseline Transformer model, which is trained from scratch, and the finetuned models shows that pre-training provides a clear advantage on this dataset.

Query-wise analysis. We carried out query-wise analysis of the predictions of the CuBERT-1K model. We summarize the queries with best and worst results (see Appendix G for detailed results). Among the multi-hop queries: On positive examples, the model struggles the most on the top-2 queries (Q9 and Q7) by the average number of tokens in examples (see Table 6, Appendix F); and the query Q12 has the highest exact match due to the simplicity of the query. On negative examples, the query Q8 works best at the cost of positive examples; and the worst performing query Q4 has the smallest number of negative examples (see Table 6, Appendix F).

Among the single-hop queries, there are 6 queries with fewer than 100 positive examples. Of these, except for the query Q49, all others are among the worst performing. The queries Q36 and Q38 are simple and are the best performing on positive examples. On negative examples, the model achieves very high exact match (>95%) for several queries; and the query Q42 has the lowest exact match.

Effect of number of spans. Figure 4 is a Radar chart of span-wise distribution of exact match for the CuBERT-1K model. The number of spans per example range from 0 to > 20. The number in parentheses against a span label is the count of examples in the test split with those many spans, e.g., 33,437 negative examples with “0-Span". The concentric circles indicate exact match in the range of $[0, 1]$ in steps of 0.2. The exact match of the predicted spans is shaded in light Red color. Both the number of examples and the exact match decrease with increasing number of spans.

Ablation with respect to supporting facts. We perform an ablation study to investigate the effect of the presence of supporting facts in the data. We finetune CuBERT-1K with only answer spans in targets. We compare it against the CuBERT-1K model from Table 2, which is finetuned on both answer and supporting-fact spans. We compute exact match with respect to answer spans only. While the model finetuned without supporting facts achieves 86.96% exact match on all examples, the model finetuned with supporting facts achieves exact match of 87.41%. Thus, training with supporting facts

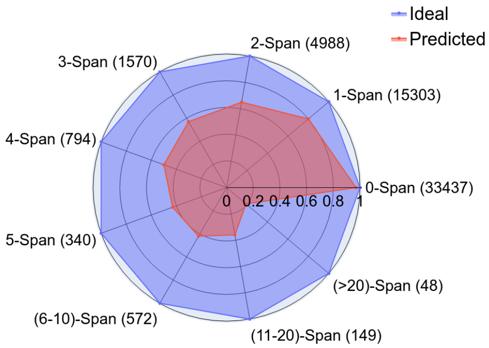


Figure 4: Span-wise distribution of exact match.

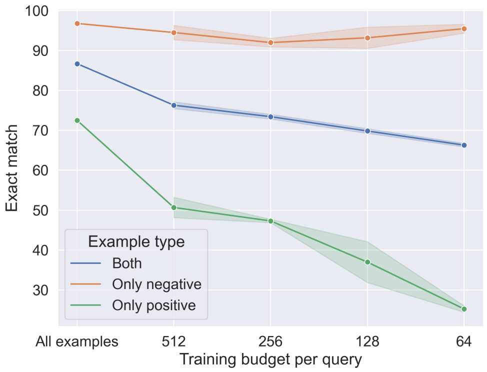


Figure 5: Learning from limited examples per query.

provides additional supervision and helps with slightly improved exact match with respect to answer spans. The full results are available in Table 4 (Appendix B).

Training on all queries versus individual queries. We investigate the effect of training a single model for all queries (our default choice) versus training a separate model for each query. Appendix B and Figure 6 provide the details and metrics for this experiment. The model trained on all queries outperforms the query-specific models on 46 of 52 queries, showing significant positive transfer.

5.2 Experimentation under Pragmatic Considerations

5.2.1 Scaling to Large-Size Code

Section 5.1 evaluates models on *only relevant code*. However, a developer may provide large-size code (e.g., entire files) during inference which could contain code not relevant to the query. To evaluate this pragmatic setting, we prepare a *file-level test dataset* from the test split of the ETH Py150 Open dataset. Note that the CodeQL engine may return multiple answers for a given query on code within a file. For preparing CodeQueries, they are treated as distinct examples with their own distinct sets of relevant code blocks. We now drop this distinction, and consider all answer and supporting-fact spans for a query over a file together as ground truth for a single example comprising the entire file. This dataset is substantially more challenging because of large size and more spans per example. This file-level dataset contains 44,423 test examples, among those, 16,712 are positive (with 9,408 being single-hop and 7,304 being multi-hop) and the remaining 27,711 are negative (with 16,710 being single-hop and 11,001 being multi-hop). Out of these, 75.33% examples have more than 1024 tokens (the input size limit of our largest model, CuBERT-1K) and the average number of tokens is 5407. Note that even if an example has less than 1024 tokens, it may contain code irrelevant for the query unlike in the ideal setting of Section 5.1.

We evaluate the two-step procedure in Section 4.2 on the file-level test dataset against *two heuristic baseline procedures*: (1) *Prefix*: Take the maximum sized prefix of the file that the model can handle as input. (2) *Sliding window*: Take the maximum sized non-overlapping chunks of the file w.r.t. the input size of the model, perform inference on them independently and combine the results. We also consider an analog of the ideal setting at the file level, which we call (3) *file-level ideal*: this takes the union of relevant code blocks for a query from a file across all ground-truth spans in it.

For the two-step procedure, we train a relevance classification model as outlined in Section 4.2 and use it to first select a set of blocks from a file for a given query. The span prediction model is then applied on the selected set of blocks to answer the query. We use the best performing span-prediction model from Section 5.1 (the CuBERT-1K model) as it is *without additional training*.

Table 3 shows the exact match results for all the procedures. The performance of the span prediction model in the file-level ideal setting is provided for reference. Because the model is supplied only relevant blocks, its exact match is higher than the three procedures. However, its exact match on positive examples is 59.60%. The average number of code tokens per example in the ideal setting of Section 5.1 is 1297, whereas due to the file-level scope, the average number of code tokens in the examples of the *file-level* ideal setting is 1660. Thus, the increase in size of code adversely affects the model performance. Further, the average number of ground-truth spans on positive examples go up from 1.83 in the ideal setting of Section 5.1 to 2.57 in the file-level dataset, making the task of achieving exact match w.r.t. all the spans more difficult.

The two-step procedure outperforms both the heuristic procedures, but is understandably inferior to the file-level ideal setting. Relevance classification achieves accuracy, precision and recall of 96.38, 95.73 and 90.10 respectively. Thus, the classifier may include spurious code blocks (false positives) or filter out relevant code blocks (false negatives). This, in addition to the large size of examples and more spans, limits its exact match on positive examples to only 52.61%. The performance of all procedures on negative examples remains, unsurprisingly, high. In sliding-window, the model sees code selected based on number of tokens from the beginning of the file. This results in arbitrary splits, causing the model to predict spurious spans and dragging down exact match on negative examples.

5.2.2 Learning from a Limited Number of Examples

In practice, a developer may have a limited number of labeled examples. We now assess the ability of the CuBERT-1K model to answer the queries when trained with decreasing budget. We finetune the model on all the queries but restrict the number of examples per query to at most 512, 256, 128 or 64. We select an equal number of positive and negative examples within the restricted budget. Figure 5 compares models trained with the different training budgets on the entire test split of CodeQueries. We have repeated the experiments three times for the restricted budgets and the variance is shown in the figure. The exact match over all examples drops from 86.70% to 66.29% (mean over three experiments) when we go from training with all available examples to 64 examples per query. This is mainly due to the drop in performance on the positive examples. As the occurrences of the **O**utside label far outnumber other labels, the models gravitate towards predicting the label O for all tokens and find it difficult to accurately predict the spans in positive examples. This is seen in the consistently better performance on the negative examples even with decreasing budget.

5.2.3 Robustness to Minor Syntactic Errors

Symbolic techniques, like CodeQL, build specialized representations of code for analysis and fail in the presence of even minor syntax errors in code. We may be able to apply neural networks that require only tokenization on such code. To validate robustness of neural models to minor errors, we consider representative errors committed by developers, such as improper indentation, omission of curly braces, and absence of keywords and operators, as possible code perturbations. We perform these perturbations on the examples in the test split of CodeQueries to form perturbed examples and evaluate the CuBERT-1K model from Section 5.1 on them *without additional training*.

For each example, we sample up to three lines from the code and apply a perturbation to each. To maintain correspondence to ground truth, we do not perturb the ground truth spans. We bias sampling towards lines in a prefix of the code, so that perturbed tokens are not pruned away when examples are pruned to length of 1024. We discard around 3.5% examples for which we could not get perturbed code that tokenizes within a small, fixed sampling budget. The CuBERT-1K model achieved 83.29% exact match on all perturbed examples. The same model has achieved 86.70% exact match on the original, error-free dataset (see Table 2). The full results are available in Table 5. We leave training on perturbed examples [Jain et al., 2020, Allamanis et al., 2021] to future work.

6 Conclusions and Future Work

We presented the CodeQueries dataset which tests the ability of neural models for code understanding on the proposed problem of answering semantic queries over code. It requires the models to perform single- or multi-hop reasoning. Despite a diverse set of 52 queries requiring varied program analyses, the proposed models perform reasonably well if relevant code is given. At the same time, our evaluation under pragmatic considerations indicates that scalability to entire files and learning from a limited number of examples have much room for improvement. We plan to explore models with the ability to handle larger contexts (e.g., [Dai et al., 2019]), better training of relevance classifier and span-prediction inspired pre-training objectives (e.g., [Joshi et al., 2020, Ram et al., 2021]) in the future. We could also add many more semantic queries and programming languages to our dataset.

Our work can make it easier for regular developers, without the time or expertise to write the formal queries, to formulate semantic queries through examples. If this line of work succeeds, it may reduce the demand for experts involved in developing the symbolic program analysis techniques.

References

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *arXiv preprint arXiv:2105.12787*, 2021.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

Aakash Bansal, Zachary Eberhart, Lingfei Wu, and Collin McMillan. A neural question answering system for basic questions about subroutines. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2021.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Active learning of points-to specifications. *SIGPLAN Not.*, 53(4), 2018.

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. Learning a static analyzer from data. In *Computer Aided Verification - 29th International Conference*. Springer, 2017.

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

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

Christopher Clark and Matt Gardner. Simple and effective multi-paragraph reading comprehension. *arXiv preprint arXiv:1710.10723*, 2017.

Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F. P. O'Boyle, and Hugh Leather. 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, 2021.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

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

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 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, 2019.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics, 2020.

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

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

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

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2018.

Geert Heyman and Tom Van Cutsem. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *CoRR*, abs/2008.12193, 2020.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 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, 2021.

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

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

Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics*, 8:64–77, 2020.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*. PMLR, 2020.

Chenxiao Liu and Xiaojun Wan. Codeqa: A question answering dataset for source code comprehension. In *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics, 2021.

Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Deep similarity learning-based type inference for python. *CoRR*, 2021.

Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. Opttyper: Probabilistic type inference by optimising logical and natural constraints. *CoRR*, abs/2004.00348, 2020.

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

Pardis Pashakhanloo, Aaditya Naik, Hanjun Dai, Petros Maniatis, and Mayur Naik. Learning to walk over relational graphs of source code. In *Deep Learning for Code Workshop*, 2022.

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

Query Suite. `https://github.com/github/codeql/blob/main/python/ql/src/codeql-suites/python-lgtm.qls`.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 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, 2016.

Pranav Rajpurkar, Robin Jia, and Percy Liang. 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, 2018.

Ori Ram, Yuval Kirstain, Jonathan Berant, Amir Globerson, and Omer Levy. Few-shot question answering by pretraining span selection. *arXiv preprint arXiv:2101.00438*, 2021.

Lance A. Ramshaw and Mitch Marcus. Text chunking using transformation-based learning. In *Third Workshop on Very Large Corpora*, 1995.

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

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems*, 2018.

tree-sitter project. `https://github.com/tree-sitter/tree-sitter`, 2021. Retrieved August 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models, 2022. URL `https://arxiv.org/abs/2201.11903`.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*. OpenReview.net, 2020.

Dirk Weissenborn, Georg Wiese, and Laura Seiffe. 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, 2017.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 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, 2018.

A Training Setup

This section documents the setup used for training the models discussed in Section 5. The pre-trained encoder 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 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. We omit the architectural details of these pre-trained models and refer the reader to the respective papers for the details. The inputs are pruned or padded after tokenization using the respective subword vocabularies. As a non-pre-trained baseline, we train a Transformer encoder with input length of 1024 from scratch. We tried 3-6 layers, 8 or 16 attention heads, and 512 or 1024 as the embedding dimension. The best performing model has a hidden dimension of 512, 2048 as dimension of hidden layer of feed-forward layer, 8 attention heads and 3 encoder layers. We used the CuBERT vocabulary for this Transformer encoder but trained the token embeddings from scratch with dimension of 512.

For span prediction, the token encodings from the final hidden layer of an encoder are passed through a dropout layer with dropout probability 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: Finetuning is performed for 5 epochs for the 512-length models and for 3 epochs for the 1024-length models. The learning rate of 3e-5 is used for CuBERT and CodeBERT, and 5e-5 is used for GraphCodeBERT. The non-pre-trained baseline Transformer model is trained for 40 epochs with the learning rate of 3e-5. The checkpoints are selected by least validation loss. 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 [Loshchilov and Hutter, 2017] and linear scheduling without any warmup. We finetuned the CuBERT 1024 model on specialized datasets for ablation of supporting facts and query-specific models (Section 5.1) and under limited number of per-query examples (Section 5.2.2). We used the same hyper-parameters for these experiments as stated for the CuBERT 1024 model above.

For the relevance classification model, we finetuned 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 hidden-dimension of 2048. We finetuned it for 5 epochs with learning rate of 3e-6 and weighted crossentropy (with weights 1/2 for irrelevant/relevant class) as loss function. The best checkpoint is decided based on least validation loss.

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

B Additional Results

	Answer Match		
Models	**All**	**Positive**	**Negative**
CuBERT-1K - trained without supporting facts	86.96	73.67	96.41
CuBERT-1K - original, trained with supporting facts	**87.41**	**73.76**	**97.11**

Table 4: Results of ablation with respect to supporting facts. "Answer Match" is exact match but based only on answer span predictions.

Training on all queries versus individual queries. For queries with small number of examples, predicting both answer and supporting facts is harder than predicting only the answer spans. We therefore train and evaluate the models for answer span prediction.

We finetune the CuBERT-1K model for each of the 52 queries separately, i.e, we group the training examples by queries and train 52 models (one model on each of these groups). We call these models *query-specific models*. We finetune a single model, called the *multi-query model*, on the training examples of all queries. We group the test examples by queries and compare the performance of the 52 query-specific models, against the multi-query model, on their respective test examples.

Figure 6 shows the performance of these models. The queries are arranged on X-axis in the decreasing order of the number of training examples. As the number of training examples decreases, the performance of the multi-query model is much better than the performance of the query-specific models. The average number of training examples when the multi-query model is better performing than the query-specific models is 434, and the least number of training examples in a query where a query-specific model is better performing than the multi-query model is 842. This indicates that training on multiple queries increases the model performance as compared to training on individual queries. The number of queries among the 52 queries where the multi-query model performs better than the query-specific models is 46 (these queries are marked by x) as compared to 6 queries where training individually is better (these queries are marked by •). This shows that multi-query learning is better performing and also convenient as a single model can answer several queries, i.e, a single model is effectively performing several program analysis tasks.

Test Data	**All**	**Positive**	**Negative**
Perturbed	83.29	67.16	94.59
Original	86.70	72.51	96.79

Table 5: Results of robustness to minor syntax errors. These are exact-match results of span prediction with the CuBERT-1K model on the perturbed test dataset (top) and the original unperturbed dataset (bottom, same as in Table 2).

C Single-hop and Multi-hop Examples

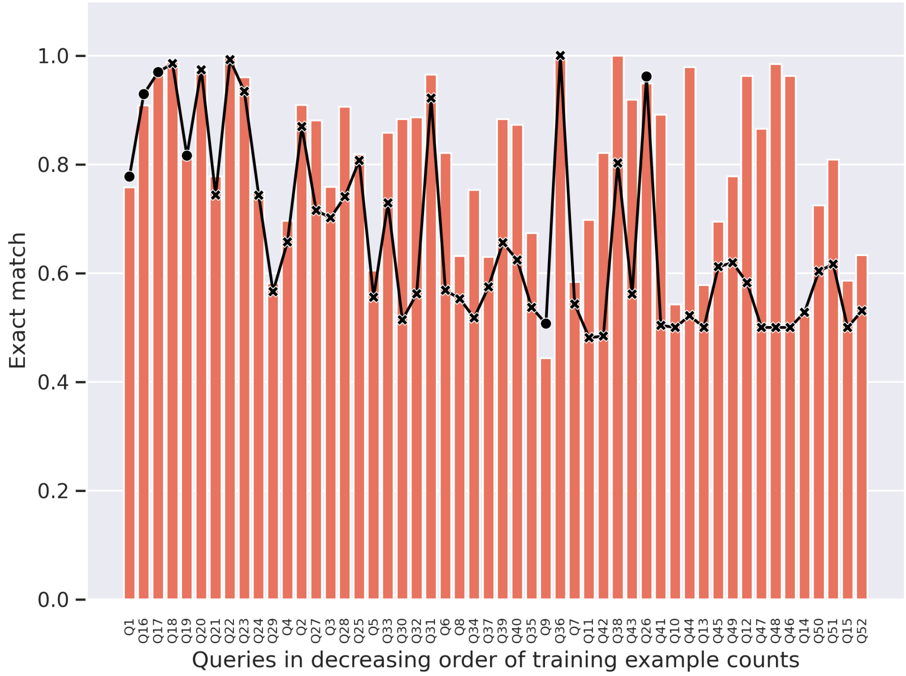
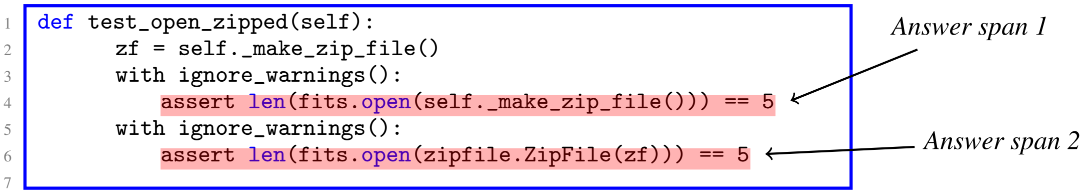


Figure 6: Comparison of training on all queries versus individual queries. The Orange bar plot shows the exact match (scaled to [0,1]) of the multi-query model (trained on all queries) on the test splits of the 52 queries arranged on X-axis in the decreasing order of the number of training examples. The line plot shows the exact match of the 52 query-specific models on their test sets. As seen, the multi-query model outperforms the query-specific models on 46 of 52 queries, showing significant positive transfer across queries. In the line plot, the 6 queries on which the query-specific models perform better than the multi-query model are marked by ● and the 46 queries on which the multi-query model performs better are marked by x.



```
def test_open_zipped(self):
      zf = self._make_zip_file()
      with ignore_warnings():
          assert len(fits.open(self._make_zip_file())) == 5
      with ignore_warnings():
          assert len(fits.open(zipfile.ZipFile(zf))) == 5
```

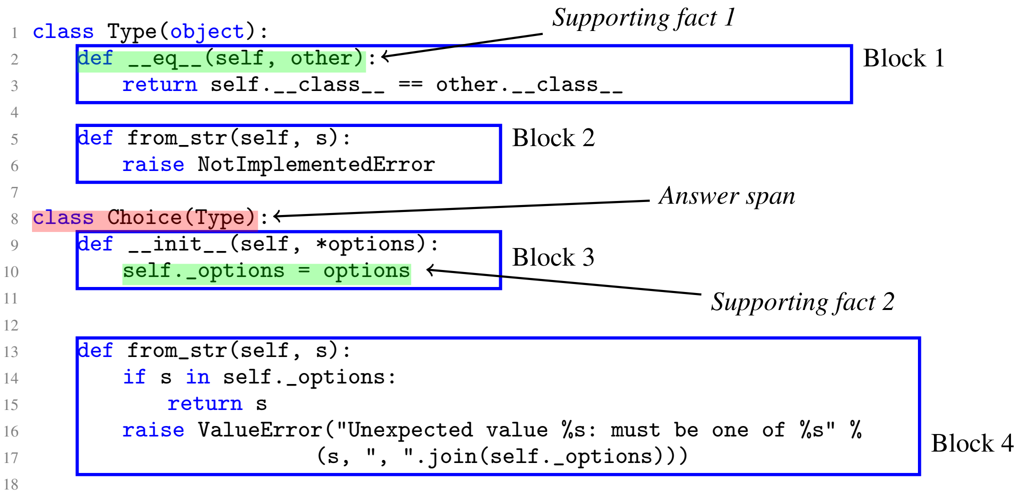
Answer span 1

Answer span 2

Figure 7: A positive example with multiple answer spans for the single-hop query "An `assert` statement has a side-effect."

Figure 7[5] shows a positive example for the single-hop query "An `assert` statement has a side-effect". This query aims to find `assert` statements that can potentially cause side-effects. It can be answered by finding `assert` statements and analyzing them to find potential side-effects, for example, the presence of function calls. In Figure 7, both the `assert` statements call `fits.open` which has side-effects and are identified as the ground truth answer spans.

[5] Part of `spacetelescope/PyFITS/pyfits/tests/test_core.py` file in the ETH Py150 Open dataset.



```
class Type(object):
    def __eq__(self, other):
        return self.__class__ == other.__class__

    def from_str(self, s):
        raise NotImplementedError

class Choice(Type):
    def __init__(self, *options):
        self._options = options


    def from_str(self, s):
        if s in self._options:
            return s
        raise ValueError("Unexpected value %s: must be one of %s" %
                        (s, ", ".join(self._options)))
```

Figure 8: A positive example with one answer span and two supporting-fact spans for the multi-hop query "`__eq__` not overridden when adding attributes".

```
def test_tbstyle_short(testdir):
      p = testdir.makepyfile("""
          def pytest_funcarg__arg(request):
              return 42
          def test_opt(arg):
              x = 0
              assert x
      """)
      result = testdir.runpytest("--tb=short")
      s = result.stdout.str()
      assert 'arg = 42' not in s
      assert 'x = 0' not in s
      result.stdout.fnmatch_lines([
          "*%s:5*" % p.basename,
          " assert x",
          "E assert*",
      ])
      result = testdir.runpytest()
      s = result.stdout.str()
      assert 'x = 0' in s
      assert 'assert x' in s
```

Figure 9: A negative example with multiple plausible, but no actual, answers for the single-hop query "An `assert` statement has a side-effect." It has four assert statements but none of them has a side-effect.

Figure 8[6] shows a positive example for the multi-hop query "`__eq__` not overridden when adding attributes". If a subclass adds attributes that are not present in its superclasses, it should override the `__eq__` method. Otherwise, the equality function might not work properly. More than one code block is needed to find answers to this query. This example requires analyzing code blocks from the subclass and its superclass. Figure 8 shows the answer span and the supporting-fact spans. The declaration of the `Choice` class is the answer span in this example. The highlighted supporting facts from different code blocks indicate that this class has added an attribute (`_options`) which is absent in its superclass (`class Type`), and the superclass has defined the `__eq__` method, which should

[6] Part of `rllab/rllab/rllab/envs/box2d/parser/xml_attr_types.py` file in the ETH Py150 Open dataset.

```python
class TagNode(Node):
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return (self.__class__ == other.__class__ and
                self.name == other.name)

    def __repr__(self):
        return '<TagNode: %s>' % self.name

class RegexTagNode(TagNode):
    def __init__(self, name, regex):
        self.name = name
        self.regex = regex

    def __eq__(self, other):
        return (self.__class__ == other.__class__ and
                self.name == other.name and
                self.regex == other.regex)

    def __repr__(self):
        return '<RegexTagNode %s: %s>' % (self.name, self.regex)

```

Figure 10: A negative example with a plausible, but not actual, answer for the multi-hop query "`__eq__` not overridden when adding attributes". The subclass `RegexTagNode` adds a new attribute `regex` not present in its base class `TagNode` but also overrides the `__eq__` method.

be overridden. Code blocks like `__init__` methods from the superclass and subclass may indicate declaration of additional attributes in the subclass. `__eq__` methods are also helpful in answering this query, as they indicate if `__eq__` method needs to be overridden. Apart from these, other code in the superclass and subclass may also indicate the presence of additional attributes, hence all code from the superclass and subclass is relevant.

D Positive and Negative Queries

By evaluating the original CodeQL queries, we obtain positive results that provide answers to the queries. We call these queries *positive queries*. To obtain negative results, we modify the positive queries to identify spans that are plausible, but not actual, answers with respect to the positive queries. We call these modified queries as *negative queries*.

We refer the reader to the query definition of the "An `assert` statement has a side-effect" query[7], which is a single-hop query. It defines a `predicate func_with_side_effects` to check if a given expression contains a function and another `predicate call_with_side_effect` to check if a given expression contains a call which can have a side-effect on the functionality of the rest of the code. The `predicate probable_side_effect` uses `func_with_side_effects` and `call_with_side_effect` to check if a given expression has a side-effect. The CodeQL queries have syntax similar to SQL and use the `select` and `where` clauses. The predicates are used in these queries to identify appropriate answer spans. In this case, the `where` clause checks if an `assert` statement has an expression `e` that has a probable side-effect `probable_side_effect(e)`. As discussed earlier, Figure 7 is a positive example for this query.

To obtain the corresponding negative query, we simply change the `where` clause to use `not probable_side_effect(e)` while keeping everything else the same. When evaluated, it gives us plausible yet incorrect answers with respect to the original, positive query since the answer does

[7] `https://github.com/github/codeql/blob/main/python/ql/src/Statements/SideEffectInAssert.ql`

contain an assert but it does *not* have a side-effect. Figure 9[8] shows a negative example we get by evaluating the negative query.

We now refer the reader to the query definition of the "__eq__ not overridden when adding attributes" query[9], which is a multi-hop query. It defines a predicate class_stores_to_attribute to check which attributes are declared by a class. The predicate should_override_eq checks that a given class does not declare __eq__ method but its superclass declares __eq__ method. The where clause selects class definitions of classes which declare additional attributes, have superclasses which declare the __eq__ method, but the class itself *does not* declare the __eq__ method.

To obtain the corresponding negative query, we define a predicate should_override_eq_and_does_override_eq to check that both the given class and its superclass declare the __eq__ method. The definition of this predicate is similar to that of should_override_eq except that instead of not cls.declaresAttribute("__eq__") we use cls.declaresAttribute("__eq__"), that is, we remove not in the first conjunct of should_override_eq. The where clause in the negative query uses should_override_eq_and_does_override_eq to select the class definition of a class which declares additional attributes, has superclasses which declares the __eq__ method and the class itself *does* declare the __eq__ method. Figure 10 [10] shows a negative example we get by evaluating the negative query.

E Details about Deriving Labeled Examples

We explain the procedure to derive labeled examples starting from the results returned by the CodeQL engine. Given the answer and supporting-fact spans of a single-hop query, we select the method to which the spans belong as the code to be queried upon. The locations of the spans within the method are treated as the ground truth. A class may contain class-level statements that do not fall within any method, e.g., declarations of class variables. If the spans fall within class-level statements, we collate all the class-level statements together as the code to be queried. Similarly, a file may contain file-level statements that do not fall within any class or method, e.g., the import statements. We collate the file-level statements together to form an example for a single-hop query if the spans fall within file-level statements. We generically refer to the statements of a method (including the method declaration), the class-level statements within a class (including the class declaration), or the file-level statements within a file as a *code block*. Single-hop queries involve analyzing single code blocks in isolation.

A multi-hop query analyzes multiple code blocks to produce an answer. However, the result returned by the CodeQL engine does not explicitly identify which code blocks were required for the analysis. We therefore inspect the CodeQL query definitions to identify the steps used by the query to select code blocks for analysis. For example, the conflicting-attributes query selects a class and all its superclass(es). Another query may analyze code within only a single class. Given the locations of answer and supporting-fact spans within code for a query, we programmatically obtain the set of code blocks needed for arriving at the same results for the given query. We implement this procedure using the tree-sitter parsing library [tree-sitter project, 2021]. An example for a multi-hop query consists of a set of code blocks and the ground truth spans.

As discussed in Section 3, we write negative queries to obtain negative examples. The procedure to derive negative examples from the results of negative queries is the same as described above. However, the spans identified in the results of negative queries are plausible, but not actual, answers for the corresponding original queries. Therefore, the resulting examples are labeled with an empty set of spans as ground truth, meaning that a model should not return any spans on the corresponding code with respect to the original (positive) query.

[8] Part of pytest-dev/pytest/testing/test_terminal.py file in the ETH Py150 Open dataset.

[9] https://github.com/github/codeql/blob/main/python/ql/src/Classes/DefineEqualsWhenAddingAttributes.ql

[10] Part of codysoyland/surlex/src/surlex/grammar.py file in the ETH Py150 Open dataset.

F Query-wise Dataset Statistics

We report the query-wise statistics for multi-hop queries, aggregated across all splits, in Table 6. We report the statistics for *All Examples*, *Positive* examples and *Negative* examples. *Count* gives the number of examples. We sort all the tables from here on by the descending order of count of all examples. Under all examples, we give the average length of the input sequences in terms of sub-tokens. Here, the sub-tokenization is performed using the CuBERT vocabulary. For positive examples, we report the average number of answer (abbreviated as *Ans.*) spans and supporting fact (abbreviated as *SF*) spans. The column *Avg. Spans after Pruning* is the average number of spans (both answer and supporting fact spans combined) after the sub-tokenized sequence is pruned to the length of 1024 for the CuBERT-1K model. Note that the number of answer or supporting fact spans is zero for negative examples and are hence omitted. We highlight the minimum and maximum values per column in bold face.

Table 6: Query-wise statistics for the multi-hop queries.

Index	**Query Name**	**All Examples**		**Positive**				**Negative**
		Count	Avg. Length	Count	Avg. Ans. Spans	Avg. SF Spans	Avg. Spans after Pruning	Count
Q1	Unused import	**48,555**	3220.68	**19,178**	**2.10**	**0**	2.04	**29,377**
Q2	Missing call to `__init__` during object initialization	1,540	287.18	770	**1.00**	2.02	2.98	770
Q3	Use of the return value of a procedure	1,013	1078.05	432	1.44	1.04	1.98	581
Q4	`__eq__` not overridden when adding attributes	857	1837.72	778	**1.00**	3.87	**3.29**	**79**
Q5	Wrong number of arguments in a call	794	872.75	355	1.28	1.04	2.08	439
Q6	Comparison using is when operands support `__eq__`	540	1063.45	230	1.52	**0**	1.21	310
Q7	Signature mismatch in overriding method	531	3610.39	247	1.32	1.22	1.40	284
Q8	Non-callable called	422	1867.71	162	1.65	1.44	2.36	260
Q9	`__init__` method calls overridden method	393	**5457.02**	193	1.29	**4.23**	2.81	200
Q10	Conflicting attributes in base classes	364	2757.73	182	1.07	3.05	1.75	182
Q11	`__iter__` method returns a non-iterator	329	**213.20**	209	**1.00**	1.08	2.08	120
Q12	Flask app is run in debug mode	243	343.55	123	**1.00**	**0**	**0.98**	120
Q13	Inconsistent equality and hashing	241	1657.24	121	**1.00**	1.00	1.78	120
Q14	Wrong number of arguments in a class instantiation	212	1317.06	99	1.21	0.91	1.84	113
Q15	Incomplete ordering	**174**	2252.13	**87**	**1.00**	1.31	2.05	87
Aggregate		56,208	2985.95	23,166	1.94	0.34	2.10	33,042

Table 7 gives the query-wise statistics for single-hop queries aggregated across all splits. The column headings have the same meaning as those of Table 6. We highlight the minimum and maximum values per column in bold face.

Table 7: Query-wise statistics for the single-hop queries.

Index	Query Name	All Examples		Positive				Negative
		Count	Avg. Length	Count	Avg. Ans. Spans	Avg. SF Spans	Avg. Spans after Pruning	Count
Q16	Unused local variable	**34,202**	464.42	**14,326**	1.43	**0**	1.36	**19,876**
Q17	Except block handles `BaseException`	22,596	561.92	10,377	1.27	**0**	1.16	12,219
Q18	Imprecise assert	16,311	306.26	6,060	2.05	**0**	2.00	10,251
Q19	Variable defined multiple times	10,300	758.36	3,694	1.82	3.49	3.51	6,606
Q20	Testing equality to None	5,454	573.88	2,354	1.36	**0**	1.24	3,100
Q21	Unreachable code	4,612	790.50	2,098	1.20	**0**	1.06	2,514
Q22	First parameter of a method is not named `self`	4,088	177.55	2,044	1.00	**0**	1.00	2,044
Q23	Unnecessary pass	2,517	388	1,114	1.26	**0**	1.14	1,403
Q24	Module is imported with `import` and `import from`	1,959	605.6	953	1.06	**0**	1.03	1,006
Q25	Module is imported more than once	1,051	899.96	489	1.17	1.20	1.98	562
Q26	Comparison of constants	909	795.2	78	**10.73**	**0**	**9.50**	831
Q27	Implicit string concatenation in a list	871	1819.25	319	1.75	**0**	1.43	552
Q28	Suspicious unused loop iteration variable	823	713.69	387	1.12	**0**	1.02	436
Q29	Duplicate key in dict literal	715	**2588.83**	150	3.99	**3.82**	2.09	565
Q30	Unnecessary `else` clause in loop	676	576.45	338	1.02	**0**	1.01	338
Q31	First argument to super() is not enclosing class	657	180.22	326	1.02	**0**	1.01	331
Q32	Redundant assignment	597	1045.49	260	1.30	**0**	1.24	337
Q33	An assert statement has a side-effect	537	458.88	206	1.65	**0**	1.59	331
Q34	Nested loops with same variable	520	920.02	241	1.16	1.05	1.90	279
Q35	Import of deprecated module	515	709.62	243	1.12	**0**	1.05	272
Q36	NotImplemented is not an Exception	476	138.42	237	1.01	**0**	1.01	239
Q37	Redundant comparison	463	908.08	188	1.47	1.30	2.52	275
Q38	Deprecated slice method	439	117.18	215	1.04	**0**	1.04	224
Q39	Constant in conditional expression or statement	423	610.60	164	1.58	**0**	1.34	259
Q40	Comparison of identical values	415	707.73	164	1.53	**0**	1.47	251
Q41	`import ⋆` may pollute namespace	397	1536.95	197	1.02	**0**	0.99	200
Q42	Unnecessary delete statement in function	382	582.99	199	**1.00**	1.00	1.92	183
Q43	Illegal raise	374	553.21	173	1.16	**0**	1.1	201
Q44	Insecure temporary file	287	440.16	139	1.09	**0**	1.07	148
Q45	Modification of parameter with default	238	827.17	96	1.48	1.02	2.26	142

Index	Query Name	All Examples		Positive				Negative
		Count	Avg. Length	Count	Avg. Ans. Spans	Avg. SF Spans	Avg. Spans after Pruning	Count
Q46	Should use a `with` statement	228	368.79	113	1.02	**0**	1.00	115
Q47	Special method has incorrect signature	222	**101.90**	111	**1.00**	**0**	1.00	111
Q48	Non-standard exception raised in special method	205	174.25	102	1.01	**0**	1.01	103
Q49	Use of `global` at module level	193	606.58	72	1.69	**0**	1.44	121
Q50	Modification of dictionary returned by locals()	173	776.82	73	1.34	**0**	1.18	100
Q51	Incomplete URL substring sanitization	168	826.67	70	1.43	**0**	1.31	98
Q52	Unguarded next in generator	**145**	512.22	**67**	1.22	**0**	1.21	**78**
Aggregate		115,138	538.71	48,437	1.46	0.31	1.53	66,701

G Query-wise Results

We now present query-wise exact match results for all, positive and negative examples. Tables 8 and 9 give results for the multi-hop and single-hop queries for the CuBERT-1K model. The average performance on the single-hop queries is better than the multi-hop queries due to the complexity of analysis inherent in multi-hop reasoning.

Table 8: Query-wise results for the multi-hop queries.

Index	Query Name	Exact Match		
		All Examples	Positive	Negative
Q1	Unused import	78.30	52.10	95.35
Q2	Missing call to `__init__` during object initialization	88.06	85.45	90.67
Q3	Use of the return value of a procedure	75.32	50.34	96.49
Q4	`__eq__` not overridden when adding attributes	56.18	63.90	**11.90**
Q5	Wrong number of arguments in a call	48.97	14.81	76.30
Q6	Comparison using is when operands support `__eq__`	82.11	64.63	95.37
Q7	Signature mismatch in overriding method	51.58	2.97	92.50
Q8	Non-callable called	59.21	8.82	**100.00**
Q9	`__init__` method calls overridden method	**34.27**	**2.82**	65.28
Q10	Conflicting attributes in base classes	47.46	16.95	77.97
Q11	`__iter__` method returns a non-iterator	79.25	96.77	54.55
Q12	Flask app is run in debug mode	**97.47**	**97.50**	97.44
Q13	Inconsistent equality and hashing	64.06	71.88	56.25
Q14	Wrong number of arguments in a class instantiation	48.61	23.53	71.05
Q15	Incomplete ordering	63.79	48.28	79.31
Aggregate		76.82	52.19	94.01

Table 9: Query-wise results for the single-hop queries.

Index	Query Name	Exact Match		
		All Examples	Positive	Negative
Q16	Unused local variable	91.78	82.90	98.34
Q17	Except block handles `BaseException`	96.56	92.71	99.82

Index	Query Name	Exact Match		
		All Examples	Positive	Negative
Q18	Imprecise assert	98.64	96.25	99.95
Q19	Variable defined multiple times	78.56	50.13	95.44
Q20	Testing equality to None	97.31	93.82	**100.00**
Q21	Unreachable code	78.07	56.04	96.68
Q22	First parameter of a method is not named `self`	99.93	99.86	**100.00**
Q23	Unnecessary pass	95.79	91.28	99.39
Q24	Module is imported with `import` and `import from`	79.85	67.67	91.24
Q25	Module is imported more than once	72.18	50.58	91.62
Q26	Comparison of constants	93.79	42.86	95.84
Q27	Implicit string concatenation in a list	83.17	62.60	98.31
Q28	Suspicious unused loop iteration variable	86.79	79.38	93.04
Q29	Duplicate key in dict literal	59.69	10.71	97.26
Q30	Unnecessary `else` clause in loop	86.92	74.53	99.07
Q31	First argument to super() is not enclosing class	95.65	92.98	98.28
Q32	Redundant assignment	91.24	83.53	97.25
Q33	An assert statement has a side-effect	84.52	62.90	98.92
Q34	Nested loops with same variable	53.53	31.71	73.86
Q35	Import of deprecated module	72.63	42.05	99.02
Q36	NotImplemented is not an Exception	**100.00**	**100.00**	**100.00**
Q37	Redundant comparison	56.91	12.99	89.42
Q38	Deprecated slice method	**100.00**	**100.00**	**100.00**
Q39	Constant in conditional expression or statement	88.31	67.92	99.01
Q40	Comparison of identical values	83.46	58.00	98.80
Q41	`import` ⋆ may pollute namespace	93.02	87.50	98.46
Q42	Unnecessary delete statement in function	72.63	77.55	**67.39**
Q43	Illegal raise	86.73	85.11	88.24
Q44	Insecure temporary file	95.65	90.91	**100.00**
Q45	Modification of parameter with default	60.00	**9.09**	92.31
Q46	Should use a `with` statement	95.00	92.50	97.50
Q47	Special method has incorrect signature	82.43	78.38	86.49
Q48	Non-standard exception raised in special method	95.31	96.88	93.75
Q49	Use of `global` at module level	92.06	79.17	100.00
Q50	Modification of dictionary returned by locals()	75.86	47.83	94.29
Q51	Incomplete URL substring sanitization	76.71	39.29	**100.00**
Q52	Unguarded next in generator	**42.86**	13.04	69.23
Aggregate		91.55	82.34	98.16

H Statistics of Syntactic Patterns of Spans

In our dataset, the answer and supporting-fact spans cover various types of programming language constructs. Hence, in Table 10, we tabulate the number of spans in terms of syntactic patterns of Python constructs in decreasing order of their frequency in the combined data of all three splits. To find the pattern of a span, we have used `tree-sitter` tree-sitter project [2021] to get the closest ancestor node which encloses the tokens appearing in the span. Two special entries in the table are block and module. A *block* node can represent any block of code, i.e., a block of code, a function, a class. Sometimes the closest ancestor node is the root node of the source code, for those cases *module* node is used as a representative node.

Table 10: Statistics of syntactic patterns of spans.

Syntactic Pattern	Count	Syntactic Pattern	Count	Syntactic Pattern	Count
import statement	43,013	raise statement	375	module	56
assignment	32,422	function parameters	373	dictionary keys	47
call	15,978	assert statement	368	break statement	43
except clause	13,269	delete statement	358	while statement	43
function definition	8,937	if statement	243	argument list	34
non-boolean binary operator	5,319	sequence expressions	192	with statement	26

Span Type	Count	Span Type	Count	Span Type	Count
class attributes	2,844	identifier	186	parenthesized expression	14
class definition	2,882	decorator	138	boolean operator	13
block	2,331	print statement	126	elif clause	12
pass statement	1,451	global statement	125	expression list	12
string literal	1,279	list comprehension	101	lambda	11
for statement	1,164	subscript	72	conditional expression	8
concatenated string	558	not operator	71	yield	5
return statement	395	try statement	65	continue statement	3
				Aggregate	134,962

I Examples of Successful and Unsuccessful Span Predictions

In this section, we present examples of both successful and unsuccessful predictions of the CuBERT-1K model for the query “An `assert` statement has a side-effect.”. Figure 9 is a negative example where there are four `assert` statements, but none of them cause a side-effect. The model does not predict any answer spans as the `assert` statements are simple enough to deduce absence of side-effect. This is a case of *successful prediction on a negative example* (true negative). Figure 11[11]. shows *a negative example for which the model is unsuccessful* (false positive). It incorrectly predicts Line 24 as an answer span, possibly because of the peculiar numerical expressions in the code.

Figure 12[12] is *a positive example where the model successfully predicted the answer span* (true positive). The presence of the `open` method might have helped the model to identify the side-effect. Figure 13[13] shows *a positive example where the model fails to predict the correct span* (false negative). The complex, multi-line expression in the statement might have made it difficult for the model to predict the correct answer span.

[11] Part of `fredrik-johansson/mpmath/mpmath/tests/test_elliptic.py` file in the ETH Py150 Open dataset

[12] Part of `kvesteri/flask-storage/tests/test_mock.py` file in the ETH Py150 Open dataset.

[13] Part of `getsentry/raven-python/tests/functional/tests.py` file in the ETH Py150 Open dataset.

```
1  def test_jtheta_issue_79():
2      # near the circle of covergence q = 1 the convergence slows
3      # down; for q > Q_LIM the theta functions raise ValueError
4      mp.dps = 30
5      mp.dps += 30
6      q = mpf(6)/10 - one/10**6 - mpf(8)/10 * j
7      mp.dps -= 30
8      # Mathematica run first
9      # N[EllipticTheta[3, 1, 6/10 - 10^-6 - 8/10*I], 2000]
10     # then it works:
11     # N[EllipticTheta[3, 1, 6/10 - 10^-6 - 8/10*I], 30]
12     res = mpf(’32.0031009628901652627099524264’) + \
13          mpf(’16.6153027998236087899308935624’) * j
14     result = jtheta(3, 1, q)
15     # check that for abs(q) > Q_LIM a ValueError exception is raised
16     mp.dps += 30
17     q = mpf(6)/10 - one/10**7 - mpf(8)/10 * j
18     mp.dps -= 30
19     try:
20         result = jtheta(3, 1, q)
21     except ValueError:
22         pass
23     else:
24         assert(False)
25
26     # bug reported in issue 79
27     mp.dps = 100
28     ...
```

Figure 11: A negative example for the query “An `assert` statement has a side-effect.”. It has no assert statement which has a side-effect, but the model incorrectly predicts Line 24 as an answer span.

```
def test_reads_file_object_and_saves_in_dict(self):
        storage = MockStorage()
        io = StringIO()
        io.write(’file contents’)
        storage.save(’key’, io)
        assert storage.open(’key’).read() == ’file contents’
```

Figure 12: A positive example for the query “An `assert` statement has a side-effect.” where the model predicts the ground-truth span (highlighted in Red) successfully.

```
def test_absolute_import(self):
        string = ’from __future__ import absolute_import’
        kwargs = {
            ’stdout’: open(’/dev/null’, ’a’),
            ’stderr’: open(’/dev/null’, ’a’),
        }
        for filename in find_files(ROOT, ’*.py’):
            assert not call([’grep’, string, filename], **kwargs), \
                "Missing %r in %s" % (string, filename[len(ROOT) - 5:])
```

Figure 13: A positive example for the query “An `assert` statement has a side-effect.” where the model fails to predict the ground-truth span (highlighted in Red).