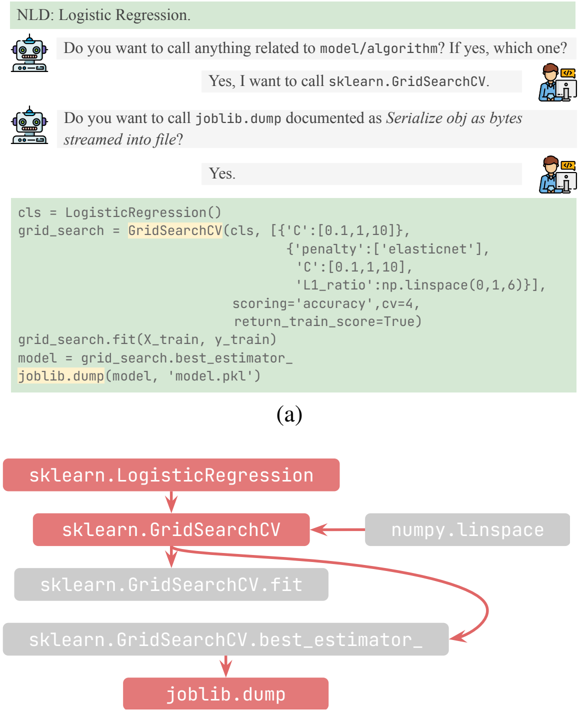
Python Code Generation by Asking Clarification Questions

Haau-Sing Li[1], **Mohsen Mesgar**[2*], **André F. T. Martins**[3,4,5], **Iryna Gurevych**[1]
[1]Ubiquitous Knowledge Processing Lab (UKP Lab)
Department of Computer Science and Hessian Center for AI (hessian.AI), TU Darmstadt
[2]Bosch Center for Artificial Intelligence, Renningen, Germany
[3]Instituto Superior Técnico and LUMLIS (Lisbon ELLIS Unit)
[4]Instituto de Telecomunicações, Lisbon, Portugal [5]Unbabel
hli@ukp.tu-darmstadt.de

Abstract

Code generation from text requires understanding the user's intent from a natural language description and generating an executable code snippet that satisfies this intent. While recent pretrained language models demonstrate remarkable performance for this task, these models fail when the given natural language description is under-specified. In this work, we introduce a novel and more realistic setup for this task. We hypothesize that the under-specification of a natural language description can be resolved by asking clarification questions. Therefore, we collect and introduce a new dataset named **CodeClarQA** containing pairs of natural language descriptions and code with created synthetic clarification questions and answers. The empirical results of our evaluation of pretrained language model performance on code generation show that clarifications result in more precisely generated code, as shown by the substantial improvement of model performance in all evaluation metrics. Alongside this, our task and dataset introduce new challenges to the community, including when and what clarification questions should be asked. Our code and dataset are available on GitHub.[1]



NLD: Logistic Regression.

Do you want to call anything related to `model/algorithm`? If yes, which one?

Yes, I want to call `sklearn.GridSearchCV`.

Do you want to call `joblib.dump` documented as *Serialize obj as bytes streamed into file*?

Yes.

```
cls = LogisticRegression()
grid_search = GridSearchCV(cls, [{'C':[0.1,1,10]},
                                  {'penalty':['elasticnet'],
                                   'C':[0.1,1,10],
                                   'L1_ratio':np.linspace(0,1,6)}],
                           scoring='accuracy',cv=4,
                           return_train_score=True)
grid_search.fit(X_train, y_train)
model = grid_search.best_estimator_
joblib.dump(model, 'model.pkl')
```

(a)

(b)

Figure 1: (a) An example of NLD-code pair that requires further clarification. We highlight operations that need clarification. (b) The generated graph of the example. Each node is an operation, with key operations marked in red and the rest in gray. Edges show the data flow.

1 Introduction

Text-to-code generation aims to understand a user's intention represented by a natural language description (NLD) to generate a code that satisfies the user's intention. Models for this task are a crucial component of digital pair-programmers, which assist data scientists (Agashe et al., 2019; Liu et al., 2021), software developers (Chen et al., 2021; Xu et al., 2022), and computer programming educators (Li et al., 2022).

Recent work addresses this task using pretrained language models (PLMs) fine-tuned on large-scale code data in general-purpose programming languages, such as Python and Java (Chen et al., 2021; Li et al., 2022; Nijkamp et al., 2022; Chowdhery et al., 2022; Xu et al., 2022; Lahiri et al., 2022).

Although these models are successful, they fail to resolve the case of an NLD that lacks enough specifications. Figure 1a depicts an example of under-specified NLD (shown in yellow). The problem of missing specifications in NLDs not only widely occurs in real-world use cases (Lahiri et al., 2022; Chaurasia and Mooney, 2017) but is also important for training text-to-code generation models. Although important, alleviating the under-specification of NLDs is challenging for two rea-

[*] Work done while being a postdoc at UKP Lab.
[1] https://github.com/UKPLab/codeclarqa

*Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics
Volume 1: Long Papers*, pages 14287–14306
July 9-14, 2023 ©2023 Association for Computational Linguistics

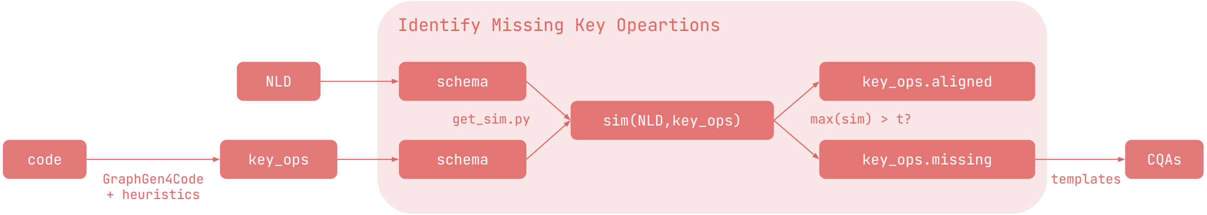


Figure 2: Our method for creating the CodeClarQA dataset. We identify key operations and corresponding documentation from the code. We represent them in a latent space using their schemata, letting us compute similarity scores of all schema element pairs between an NLD and the documentation of a key operation. If there is an element pair with a similarity score lower than a threshold t the key operation is missing in NLD. We adopt templates to create CQAs for missing key operations.

sons. First, missing specifications can happen at various levels, including individual operations, argument values of the operations, and sub-tasks consisting of several operations decomposed from the entire source code file as the task. Second, it is not obvious how to identify if an NLD carries information about specifications at any level mentioned above or not.

In this paper, we introduce interactivity into text-to-code generation for specifications on the level of individual operation calls in Python. We hypothesize that by gathering more specifications using these interactions, we can alleviate the incompleteness of NLD's specifications and thus generate a more precise code (Figure 1a). To train and evaluate such models, we introduce the CodeClarQA dataset collected through a novel method to synthetically generate clarification questions and answers (CQAs) for an NLD-Code pair. To map the operations to natural language, we retrieve the API documentation. If there is a low similarity between the NLD and the operation's documentation, we identify the operation as a missing specification and generate a clarification question (CQ). The answers to CQs are selected from the given code. Furthermore, we propose a pipeline to demonstrate the use case of our dataset for developing NLD-Code generation models. Our pipeline consists of three modules – a clarification need predictor, a CQ generator, and a code generator. For each module, we introduce models that can serve as baselines.

To evaluate the quality of our dataset, we conduct a human evaluation. We also evaluate the models we proposed for each component of our pipeline. Our empirical results show that by conditioning PLM-based code generation models on CQAs in our dataset, the performance of these models increases, indicating the correctness of our hypothesis and our collected dataset. Alongside, our experimental results show that advanced PLMs (e.g., RoBERTa, BART, T5, and CodeT5) struggle to achieve high performance under the interactive code generation pipeline. This important observation demonstrates the difficulty of our dataset for the recent PLMs, introducing new challenges for these models.

2 Creating the CodeClarQA Dataset

We aim to make the process of code generation interactive such that by asking CQs about a given NLD, we resolve NLD under-specification before generating any code. To do so, we design a novel method to synthetically collect CQAs for a given NLD-Code pair, leading to the new dataset, which we call CodeClarQA. Figure 2 shows a general view of our data creation method.

2.1 Identifying Key Operations

Key operations correspond to sub-tasks decomposed from the code snippet as the task. For instance, the code in Figure 1 can be decomposed into three sub-tasks: call the logistic model, use grid search to fit different logistic models, and save the best model. The corresponding key operations are *sklearn.LogisticRegression*, *sklearn.GridSearchCV*, and *joblib.dump*. Ideally, an NLD should provide enough information about key operations in a code. If the NLD does not do so, it lacks sufficient specifications. Thus, our first step for generating CQAs for a given NLD-code pair is to identify the key operations required to generate the code from NLD.

To identify key operations, we represent the code as a graph. Data flow graphs are an effective structural representation of code semantics in a code. Given this fact, we use the graph defined by the GraphGen4Code toolkit (Abdelaziz et al., 2021), the state-of-the-art toolkit for generating a graph from a source code, including API-related operations and the data flow. This makes it easy for us to identify key operations. Figure 1a shows the graph representation. Non-leaf nodes represent key operations. Edges indicate the data flow of the operations. For each NLD-Code pair, we parse the code to generate a graph.

Given a graph, we identify nodes with the following properties as key operations: **(i) For operations that are one object/function and its methods/fields, we treat the object/function as a key operation.** This is coherent with one hypothesis behind the design of GraphGen4Code, where an object/function is first initiated before fields/methods thereof are applied. For instance, *sklearn.GridSearchCV* is the key operation among all operations related to it, as other operations apply a method (*.fit*) or read a field (*.best_estimator_*) of it (Figure 1b). **(ii) For a multiple-operation line of code, the last operation on the data flow path is a key operation.** For instance, *sklearn.GridSearchCV* and *numpy.linspace* are in the same line. *sklearn.GridSearchCV* is a key operation since *sklearn.GridSearchCV* is the line's highest-level operation (Figure 1b). See Appendix A for details of the procedure of identifying key operations.

2.2 Is a Key Operation Missing in NLD?

Given a set of key operations required to generate a code, we should identify if the given NLD provides any information about these operations. To do so, for each key operation, we propose to align the schema of textual documentation of a key operation with the schema of a given NLD. A schema (Majumder et al., 2021) is defined as a set of important elements of a document. Every schema element is either in the form of (*verb*, *key-phrase*, *relation*) or (*key-phrase*), where *key-phrase* is extracted using YAKE (Campos et al., 2020), and *verb* and *relation* are obtained by searching through the closest verb and its dependency relation using the dependency tree (Qi et al., 2020). An example of (*verb*, *key-phrase*, *relation*) is (transforms, final estimator, *obl*), and an example of (*key-phrase*) is (pipeline).

For each key operation required to generate a code, we compute similarity scores for all schema element tuples using elements from the NLD and the documentation. For each pair of schema elements, we use a pretrained text encoder (Reimers and Gurevych, 2019) to compute similarity scores between these phrases as key information. Note that we combine *verb* and *key-phrase* if the schema element is in the triplet form before computing the similarity score. Eventually, we identify a key operation is missing in the NLD if the highest similarity score of all schema element pairs is lower than the threshold t. Each key operation is then labeled as *aligned* or *missing*. We perform a grid search to find the best t on a validation set, maximizing the F1 score. See Appendix B for an example.

2.3 Generating CQAs for Missing Key Operations

We formulate CQs as multiple-choice questions and yes/no questions. The former needs an answer with yes/no following a choice of an API call. The latter requires only an answer with yes/no.

Multiple-choice. We collect all extracted key operations from the dataset, mentioned or missing, that contain 1023 different API sub-modules, methods, and fields. We then extract the last tokens from each operation name, filter out all stop words, and keep operations that share the same last token of their names. For instance, *sklearn.partial_fit* and *sklearn.fit* share the same last token as *fit*. Note that we hypothesize that for operations with the same name but from a different library, e.g., *keras.fit* and *sklearn.fit*, they refer to the same operation. We generate multiple-choice questions for these key operations if they are missing. To do so, we use the template *Do you want to call anything related to* LAST_TOKEN*? If yes, which one?*.

Yes/No. For operations that do not belong to multiple-choice questions, we generate a yes/no question using the template *Do you want to call* OPERATION_NAME *documented as* DOC*?*. For instance, a CQ about *numpy.logspace* is generated as "Do you want to call *numpy.logspace* documented as *Return numbers spaced evenly on a log scale?*"

2.4 Dataset

We use NLD-Code pairs in the notebookCDG dataset (Liu et al., 2021) to create CQAs because of the high code quality ensured by votes of the Jupyter Notebooks and the high NLD quality ensured by the preprocessing method based on the study of markdown documentation (Wang et al., 2021a). We first identify key operations (§2.1) and label them as either *aligned* or *missing* (§2.2). Finally, we select NLD-Code pairs with at most five

	Total	Train	Dev	Test
# NLD-Code Samples	19368	17431	968	969
Avg. NLD Length	12.43	12.45	12.22	12.30
Avg. Code Length	44.40	44.52	44.52	42.25
# Samples w/ CQAs	12339	11098	630	611
# Samples w/o CQAs	7029	6333	338	358
# CQAs	17506	15711	923	872
# Multiple-Choice Qs	8952	8008	474	470
# Yes/No Qs	8554	7703	449	402
# Operations	817	749	227	215
# Packages	89	82	33	28

Table 1: Some statistics of our dataset.

missing key operations, duplicate missing key operations, and create CQAs (§2.3). Table 1 shows dataset statistics.

3 Pipeline for CQ-driven Code Generation

Our system generates precise code by asking CQs before generating. To do so, it uses an interactive code generation pipeline that includes three modules: (i) a clarification need predictor, (ii) a CQ ranker, and (iii) a code generator.

Given an NLD, the clarification need predictor predicts the need to ask CQs, with labels *Need* and *No Need*. If there is a need for asking CQs, the CQ ranker selects n CQs. We set n as five to push these models to choose CQs with the most information gains. Given the NLD, CQs and corresponding answers, the code generator generates a code.

4 Experiments

Having proposed our dataset (§2) and a pipeline for interactive code generation (§3), we next evaluate the quality of the dataset creation method by focusing on §2.2 and results of use our dataset to evaluate recent PLM-based models for each pipeline module for interactive code generation, before assessing the quality of the pipeline. The dataset evaluation analyzes the effectiveness of identifying key operations, while experiments on the pipeline aim to validate our hypothesis that interactiveness helps code generation and evaluate task difficulty.

4.1 Dataset Evaluation

To evaluate our dataset creation method, we randomly split our dataset into train/validation/test sets. We asked two Ph.D. students in computer science to annotate each NLD-Code pair in the validation and test sets. The annotation for each NLD-Code pair is a binary label, indicating if the NLD misses any key operation from the code. These annotations let us (i) study the properties of our dataset and (ii) evaluate the quality of our method for finding missing key operations using different text encoders. See Appendix §D for more details.

Setting. The validation and test set consist of 100 NLD-Code pairs respectively. The Fleiss Kappa is 0.74 (0.83 for the validation and 0.66 for the test set). We randomly chose one annotator's annotation as reference labels. See Appendix §E for more analysis on annotation results.

4.2 Clarification Need Predictor

In order to label when CQs were needed, we learned a binary classifier. This classifier predicts, for an NLD, whether it needs further clarification. The classifier was trained on the NLD-Code pairs in the training portion of the **CodeClarQA** dataset.

Setting. We fine-tune baseline pretrained transformer classifiers, including BERT (Devlin et al., 2019), RoBERTa (Liu et al., 2019), and the encoder of BART (Lewis et al., 2020). To include models trained on NLD data, we also fine-tune the encoder of PLBART (Ahmad et al., 2021). Models are fine-tuned on the training set with NLDs as the input. We fine-tune each model for 10 epochs with learning rate 5×10^{-5} and pick the best-performing model on accuracy. We compare the models on the test set using accuracy, precision, recall, and F1.

4.3 CQ Ranker

Given an NLD, a CQ ranker should recommend potential key operations by asking CQs. We formulate this as a ranking task, where we select a subset of CQs from a universal set of CQs. We use all created CQs using our method mentioned in §2 as the universal set.

Setting. We follow Aliannejadi et al. (2021) and fine-tune cross-encoders on all NLD-CQ pairs and experiment with models used in §4.2. Given an NLD-CQ pair, each model is trained to do binary classification. At inference time, all CQs in the universal set are paired with a given NLD and ranked by model score. Given an NLD, positive samples CQs created in the dataset. To create negative samples, we experiment with random negative sampling and BM25 (Robertson et al., 1995). The number of negative samples selected is the average number of positive samples. Each model is trained for 10 epochs with learning rate 5×10^{-5}.

Model	Acc	P	R	F1
SentenceT5$_{\text{large}}$(0.83)	87.40	71.43	80.65	75.76
SentenceT5$_{\text{xl}}$(0.82)	87.40	68.57	82.76	75.00
GTR$_{\text{large}}$(0.70)	88.19	71.43	83.33	76.92
GTR$_{\text{xl}}$(0.69)	88.98	68.57	**88.89**	77.42
MiniLM$_{\text{L12}}$-all-v1(0.47)	87.40	74.29	78.79	76.47
MiniLM$_{\text{L12}}$-all-v2(0.49)	87.40	73.53	78.12	77.61
DistilRoBERTa(0.49)	88.19	74.29	83.33	76.92
RoBERTa$_{\text{large}}$-all(0.46)	89.76	80.00	82.35	81.16
MPNet$_{\text{base}}$-all-v1(0.40)	84.25	80.00	68.29	73.68
MPNet$_{\text{base}}$-all-v2(0.42)	86.61	**82.86**	72.5	77.33
MPNet$_{\text{base}}$qa-dot(0.62)	89.76	**82.86**	80.56	81.69
MPNet$_{\text{base}}$qa-cos(0.41)	**90.55**	**82.86**	82.86	**82.86**

Table 2: Results for identifying missing key operations on our test set using different text encoders. The numbers in parenthesis refer to the threshold optimized on the human-annotated validation set for F1 score.

We evaluate model performance with the test set on $R@k, k \in \{1, 3, 5, 10\}$.

4.4 Code Generator

The key hypothesis of our work is that interactive code generation systems outperform non-interactive ones. In this experiment, we conduct a proof-of-concept experiment to validate this hypothesis, assuming a perfect interactive system with perfectly asked CQs and answers. We fine-tune models with and without oracle CQAs from our dataset. Note that for both yes/no and multiple-choice questions, we have only positive answers in our dataset.

Setting. We experiment with models mentioned by Zhou et al. (2022) for fine-tuning, including GPT-Neo-{125M, 1.3B} (Black et al., 2021), T5 (Raffel et al., 2020), and CodeT5 (Wang et al., 2021b). We include CodeParrot-{110M,1.5B} (Tunstall et al., 2022). Note that for CodeParrot-110M, we use the model fine-tuned on text-to-code generation.[2] Moreover, we finetune PLBART-base (Ahmad et al., 2021). We train each model for 40 epochs with learning rate 5×10^{-5}. Each experiment takes up to 6 hours on a single A100 GPU. We evaluate models on BLEU score (Papineni et al., 2002), CodeBLEU score (Ren et al., 2020), and Exact Match (EM). Note that we don't include state-of-the-art execution-based metrics (Huang et al., 2022; Yin et al., 2022), since it requires us to include code context into the dataset, which leverages the difficulty of dataset construction. As we don't include code context into the dataset, code predictions are more likely to fail on e.g. variable naming, which affects the execution results but does not necessarily lead to poor code quality.

Error	Freq		ER (%)	
	Dev	Test	Dev	Test
Taxonomy (FP)	3 (.33)	3 (.50)	7.32	8.57
Element Pair (FP)	3 (.33)	3 (.50)	7.32	8.57
Argument (FN)	4 (.57)	4 (.67)	4.08	4.35

Table 3: Statistics of the most common FP and FN predictions. Freq refers to frequency, with relative frequency included in the parenthesis. Error rates (ER) are computed on the corresponding predictions.

4.5 End-to-end Pipeline Evaluation

To assess the performance of the entire pipeline (§3), we use the best-performing models for each module. We pass an NLD to the clarification need predictor. Given a positive prediction, we pass the NLD to the CQ ranker. For each NLD, we select the top-k ($k \in \{1, 3, 5\}$) ranked CQs by the CQ ranker. We compare them to CQs created using our approach and select overlapping CQs. Finally, we concatenate the NLD and all selected CQs with corresponding answers and feed them to the code generator.

5 Results

5.1 Dataset Evaluation

We first evaluate the effect of different text encoders on the performance of our method for identifying missing operations. Table 2 shows the results. We achieve the best performance using MPNet$_{\text{base}}$qa-cos text encoder. We then use our annotations to analyze the predictions of this model. Table 3 shows the results of this analysis in terms of False Positive (FP) and False Negative (FN) errors. For the sake of brevity, we report the full list in Appendix §D.

The "Taxonomy" and "Element Pair" error types take up to 7.32% and 8.57% of all operations predicted as *aligned* in the validation/test sets, respectively.

The rare case of FP predictions suggests that our approach to generating CQAs effectively creates CQAs for missing key operations. The *Taxonomy* error relates the differences related to the taxonomy of terms that could not be identified, taking up to about 8.57%. The *Element Pair* error relates to

[2] https://huggingface.co/datasets/codeparrot/github-jupyter-parsed

Type	Category	Example
FP	Taxonomy	**NLD**: We've addressed a lot of the issues holding us back when using a linear model... **Code Line:** LCV = LassoCV() **Doc:** Lasso CV: Lasso linear model with iterative fitting along a regularization path.
FP	Element Pair	**NLD**: ...we concatenate the two sets while remembering the index so we can split it later again. **Code Line:** train_features = train.drop(['SalePrice'], axis=1) **Doc:** drop: Make new Index with passed list of labels deleted.
FN	Argument	**NLD**: Transforming some numerical variables. **Code Line:** all_data['MSSubClass'] = all_data['MSSubClass'].apply(str) **Doc:** apply: Apply a function along an axis of the Data Frame.

Table 4: Examples of predictions in identifying missing key operations. We provide true positive (TP), false positive (FP), and false negative (FN) examples. **Category** refers to the assigned category of prediction by human evaluation. Key operations and schema element pairs with the highest similarity scores are highlighted.

Model	Acc	Precision	Recall	F1
$\text{RoBERTa}_{\text{base}}$	64.94	65.91	**95.29**	77.43
$\text{BART}_{\text{base}}$	70.33	74.78	79.95	77.24
$\text{PLBART}_{\text{base}}$	71.05	**75.86**	79.34	77.56
$\text{BERT}_{\text{base}}$	**71.49**	75.72	80.73	**78.13**

Table 5: Results of clarification need prediction. All numbers are averaged across four runs.

		$R@k(\%)$			
Model	NS	1	3	5	10
BM25		0.43	0.79	0.79	1.22
$\text{RoBERTa}_{\text{base}}$	*BM25*	0.09	0.26	0.73	1.61
	Rand.	5.3	12.98	18.45	27.02
$\text{BERT}_{\text{base}}$	*BM25*	5.21	13.17	17.47	24.39
	Rand.	4.57	12.41	17.82	26.88
$\text{PLBART}_{\text{base}}$	*BM25*	4.41	10.84	15.85	23.19
	Rand.	10.64	17.91	22.24	30.15
$\text{BART}_{\text{base}}$	*BM25*	4.88	10.98	14.66	21.44
	Rand	**13.62**	**21.34**	**26.19**	**34.88**

Table 6: Results of CQ ranking on the test set. Column headers indicate the k value in $R@k$ in percentages. *NS* refers to the negative sampling strategy. All numbers are averaged across four runs.

the cases where non-relevant schema elements are aligned, taking up to about 8.57%. The *Argument* error represents the alignment between arguments, taking up only 4.08%/4.35% of all negative predictions from the validation/test set. Table 4 shows examples of these errors.

For the taxonomy error, our method identifies a schema element match of *linear models* but fails to predict the difference between a *lasso linear model* and a *linear model* in the taxonomy of machine learning terms. This finding shows a potential direction of future work, in which *aligned* operations might require clarification to be distinguished from operations with similar names. The example of *Argument* error reflects the case where a complete semantics of the operation needs both the documentation and the argument values. As we proposed to compare documentation and the NLD, we miss out on arguments that can complement the semantics of the operation. The corresponding example shows that the operation *.apply* 's semantics is incomplete without the argument *str*. This is reflected in the design of our method, as we use API documentation which reflects the semantics of the API call, while argument values are not documented.

The *Element Pair* error example shows that (make, index, *obj*) from the documentation's schema is aligned with (index) from NLD's schema. In contrast, the key operation from the documentation should be either *drop* or *deleted*.

5.2 Clarification Need Predictor Evaluation

Table 5 summarizes the results of different classifiers. Most tested models obtain relatively high performances except for $\text{RoBERTa}_{\text{base}}$, which overfits the imbalanced data where 63.71% samples have positive labels, as shown by the high recall but low precision. Moreover, $\text{BERT}_{\text{base}}$ has the best performance on both accuracy and F1 score.

5.3 CQ Ranker Evaluation

We report the results of our experiments on CQ generation in Table 6. The results confirm that our design of selecting CQs is reasonable, with the best-performing model showing similar results to the "Question Relevance" task designed by Aliannejadi et al. (2021). However, we hypothesize that our task is more challenging, as the lexical overlap between the NLD and the correctly selected CQs is low due to our design of dataset creation

Method	BLEU	CodeBLEU	EM(%)
T5$_{\text{base}}$	7.88	14.65	0.88
T5$_{\text{base}}$+*CQAs*	12.43	19.04	2.09
GPT-Neo$_{\text{125M}}$	11.89	24.75	0.00
GPT-Neo$_{\text{125M}}$+*CQAs*	15.63	26.97	0.00
GPT-Neo$_{\text{1.3B}}$	13.95	26.57	0.00
GPT-Neo$_{\text{1.3B}}$+*CQAs*	19.64	31.05	0.00
CodeParrot$_{\text{110M}}$	12.61	26.42	0.10
CodeParrot$_{\text{110M}}$+*CQAs*	17.97	31.01	0.00
CodeParrot$_{\text{1.5B}}$	12.04	26.02	0.10
CodeParrot$_{\text{1.5B}}$+*CQAs*	17.77	30.74	0.00
PLBART$_{\text{base}}$	24.63	28.04	12.00
PLBART$_{\text{base}}$+*CQAs*	38.91	38.54	**18.03**
CodeT5$_{\text{base}}$	27.03	32.66	10.84
CodeT5$_{\text{base}}$+*CQAs*	**39.13**	**38.99**	13.93

Table 7: Code generation results without and with created CQAs in our dataset. All numbers are averaged across four runs.

which looks for key operations with documentation that has no keyword matches to the NLD. This requires the model to utilize the under-specified NLD and infer the topic of the task and the user's intent before providing suggestions by asking CQs.

Our hypothesis is strongly supported by the low recall of the BM25 ranker, which ranks CQs based on their lexical similarities with NLD. Moreover, we find that models trained with the BM25 negative sampler always perform lower than the ones trained with the random sampler, which also supports our hypothesis because the BM25 negative sample is expected not to select CQs that have high lexical overlap with the NLD as negative samples, while they have a higher chance of asking key operations that are "mentioned".

5.4 Code Generator Evaluation

We train recent models using only the NLD-Code pairs or with NLD, Code, and CQAs in the **CodeClarQA** dataset. The experimental setup aims to test our hypothesis that interactiveness helps code generation by running code generation models with "perfect" clarifications. Note that this only serves as proof of concept, as CQAs contain operation names in the target source code, leading to data leakage because the names of the API calls exist in the CQs.

Model	k	BLEU		CodeBLEU		EM(%)	
		✗	✓	✗	✓	✗	✓
PLBART$_{\text{base}}$	1	19.51	19.82	22.43	22.16	5.24	6.94
	3	15.57	21.33	22.69	23.51	4.15	7.53
	5	13.20	**22.07**	21.77	**24.07**	3.95	**7.92**
CodeT5$_{\text{base}}$	1	19.14	24.04	24.14	25.56	5.37	7.15
	3	14.58	25.45	25.28	26.63	4.36	7.51
	5	13.03	**26.27**	25.13	**27.24**	4.33	**7.82**

Table 8: Pipeline evaluation. ✗refers to experiments with top k CQs directly appended to the NLD, and ✓refers to experiments with CQs selected as noted in §4.5. All numbers are averaged across four runs.

Table 7 shows that all models fine-tuned with CQs have substantially better performance, with the largest gap of 14.28 in BLEU, 10.5 in CodeBLEU, and 6.03 in EM reached by PLBART$_{\text{base}}$, which supports our hypothesis that interactions help code generation. Moreover, all models pretrained on code data have better performances, with CodeT5$_{\text{base}}$ and PLBART$_{\text{base}}$ as the best-performing models we tested.

5.5 Pipeline Evaluation

We use BERT$_{\text{base}}$ clarification need predictor, BART$_{\text{base}}$ CQ ranker with random negative sampling, and PLBART$_{\text{base}}$ trained with *CQAs*. Given the question ranker's predictions, we select CQAs from the test sample with CQ included in the top-k ($k \in \{1, 3, 5\}$) list yielded by the CQ ranker. Besides concatenating selected CQs to NLDs, we also concatenate CQs without selecting them, treating them as "unanswered clarifications".

We report the results of pipeline evaluation in Table 8. We find that model performances on all evaluation metrics substantially increased with more highly-ranked CQs being included and "answered" by comparing highly-ranked CQs and the CQAs in the dataset. Moreover, we also find the opposite trend for "un-answered clarifications" where models perform worse with more highly-ranked CQs included (but not "answered"). This aligns with the challenge of asking CQs mentioned in §5.3.

Last but not least, we compare the pipeline inference results in Table 8 to the results in Table 7. Notably, our pipeline underperforms models trained on data with only NLDs and code. This is expected, as we use code generators that are fine-tuned on all CQAs, and the results of ranking CQs suggest that the task of asking CQs is challenging (§5.3).

6 Analysis

Intuitively, asking CQs helps code generation because it provides more specifications, thus aligning model generations to desired and better-quality outputs. To test if this hypothesis stands under the context of our proposed task and pipeline, we analyze

Model		Recall			
		micro		macro	
		✗	✓	✗	✓
PLBART$_{\text{base}}$		31.08		32.89	
	+top 1	14.39	25.14	15.50	23.69
	+top 3	18.69	30.79	19.33	29.00
	+top 5	17.31	32.65	18.20	30.79
	+CQAs	**92.72**		**92.23**	
CodeT5$_{\text{base}}$		37.44		39.17	
	+top 1	15.45	28.27	17.07	27.51
	+top 3	17.72	33.62	18.90	32.47
	+top 5	17.32	35.67	18.60	34.39
	+CQAs	**92.71**		**92.63**	

Table 9: Micro and macro recalls of identified missing key operations. ✗refers to experiments with top k CQs directly appended to the NLD, and ✓refers to experiments with CQs selected as noted in §4.5. All numbers are averaged across four runs.

Model	Recall		ρ	
		BLEU	CodeBLEU	EM(%)
PLBART$_{\text{base}}$	micro	0.929^*	0.949^*	0.915^*
	macro	0.932^*	0.962^*	0.923^*
CodeT5$_{\text{base}}$	micro	0.918^*	0.938^*	0.910^*
	macro	0.909^*	0.949^*	0.911^*

Table 10: Pearson correlation coefficient (ρ) between recalls and results from Table 7 and Table 8. Results marked with * are statistically significant ($p < 0.001$).

model generations quantitatively and qualitatively.

Recall of identified missing key operations. Table 9 shows the recall of missing key operations from predictions. We find that training with clarifications includes substantially more missing key operations, while the pipeline still does not outperform models trained on data with only NLDs and code, similar to Table 8. Furthermore, we report Pearson correlation between the recall of missing key operations and code generation results (See Table 10), finding high and positive correlations which support our hypothesis that asking CQs helps code generation through clarified key operations.

Case study. We examine predictions and provide an example in Table 11. We find that training with oracle CQAs leads to predictions close to the ground truth, especially on operations, with only differences at argument-level specifications, which is expected as we focus on clarifications on operations. However, the task is challenging as the top 5 ranked CQs do not include CQs in the reference CQAs, leading to the pipeline prediction including a call of confusion matrix but missing *AdaBoostClassifier* and *cross_val_predict*.

7 Related Work

CQ generation. Aliannejadi et al. (2019, 2021) define CQs based on facets/aspects of the text input's topic, guiding annotators to write CQs based on the facet information. Eberhart and McMillan (2022) ask CQs for query refinement based on facets/aspects from existing NLDs in a dataset. Our work is distinguished from the above works as our method does not require a predefined collection of facets/aspects of the text inputs. The advantage of our method is that we collect NLDs as specifications from code.

More generally, two main focuses of work on CQ generation are (i) disambiguation of terms (Xu et al., 2019; Guo et al., 2021) and (ii) providing more information (Rao and Daumé III, 2018; Guo et al., 2021; Majumder et al., 2021; Nakano et al., 2022). With the goal of disambiguation of terms, Xu et al. (2019) utilize the knowledge base to create CQs that disambiguate different entities that share the same entity names. Guo et al. (2021) included CQs of coreference resolution that disambiguate pronouns. Rao and Daumé III (2018); Guo et al. (2021) define CQs to gather information missing from textual input. Majumder et al. (2021) ask CQs on missing information from the item description but existing in similar items, defined as missing schema. Nakano et al. (2022) construct pseudo-CQs by eliminating a part of a sentence and transforming it into a CQ and a corresponding answer. Our work adopts the definition of CQs as asking for new information and is distinguished from these works by defining a new type of information as key operations for a code, which are challenging to be defined and identified if they are included in the original text query.

Text-to-Code generation. Text-to-code generation was first defined through learning on the parallel corpus of NLD-Code pairs (Allamanis et al., 2015; Miceli Barone and Sennrich, 2017; Yin et al., 2018). To study programming in practice with dependency between different code snippets, Iyer et al. (2018) introduced a more challenging task that studies generation based on NLD and programming context. Agashe et al. (2019) address the task of generating code cells on Jupyter Notebook given previous markdown and code cells.

NLD: Confusion Matrix for the Best Model.	
Reference CQAs	**CQ1**: Do you want to call anything related to ’model/algorithm’? If yes, which one? **A1**: "Yes, I want to call ’sklearn.AdaBoostClassifier’ **CQ2**: Do you want to call anything related to ’predict’? If yes, which one? **A2**: Yes, I want to call ’sklearn.cross_val_predict’

Ground Truth	ada = AdaBoostClassifier(n_estimators=200, random_state=0, learning_rate=0.05)
	result = cross_val_predict(ada, X, Y, cv=10)
	sns.heatmap(confusion_matrix(Y, result), cmap=’winter’, annot=True, fmt=’2.0f’)
	plt.show()

CodeT5$_{\text{base}}$	y_pred = model.predict(X_test)
	y_pred_classes = np.argmax(y_pred, axis=1)
	y_true = np.argmax(y_test, axis=1)
	import scikitplot as skplt
	skplt.metrics.plot_confusion_matrix(y_true, y_pred_classes, title=’Confusion Matrix for Best Model’)
	plt.show()

CodeT5$_{\text{base}}$+*top 5*	print(confusion_matrix(y_test, gbc.predict(X_test))[1])
	print(classification_report(y_test, gbc.predict(X_test))[1])

CodeT5$_{\text{base}}$+*CQAs*	ada = AdaBoostClassifier()
	result = cross_val_predict(ada, X, Y, cv=10)
	sns.heatmap(confusion_matrix(Y, result), cmap=’winter’, annot=True, fmt=’2.0f’)
	plt.show()

Table 11: Example of predictions CodeT5$_{\text{base}}$ without asking CQs, with pipeline predictions, and with oracle CQAs. Missing operations and schema element pairs with the highest similarity scores are highlighted. Note that *top 5* ranked CQs do not include CQs in reference CQAs.

Our work also sources NL-Code pairs collected from Jupyter Notebooks (Liu et al., 2021). We do not consider dependency between different code/markdown cells when creating CQA, because including previous cells will change the necessity of asking some CQs and make our CQA creation method less controllable.

Recent research also focuses on generating code utilizing API knowledge or existing source code. Xu et al. (2020) augment data with samples created by documentation. Parvez et al. (2021) retrieve samples from the training set or an archived code database. Zhou et al. (2022) use retrieval-augmented generation approach by retrieving documentation from source code API usage. In contrast, we design the task of retrieving CQs and consider interactivity between the model and the user.

8 Conclusion and Future Work

In this paper, we introduced a new challenge of asking clarification questions for code generation for Python, along with a method to generate a dataset to create clarification questions and answers that do not require human annotations over the whole dataset. We release our collected dataset CodeClarQA, which consists of clarification questions and answers on API usage. We further proposed a pipeline system implemented by recent text and code encoders to evaluate model performances on this challenge. Our experimental results confirm that clarification questions and answers are strong information-gathering methods for better generation of code while deciding when to ask clarification questions and what questions to ask remains challenging. Future works include improving clarification questions for higher user engagement and question diversity; studying the lack of user intent completeness beyond the level of operations, e.g., lack of user intent completeness in arguments; and introducing conversational relations between clarification questions.

Limitations

Our method primarily focuses on operation-level specifications, while there are real-world use cases with other specifications. Moreover, our method of creating CQAs can only be scaled to all Python codes that involve heavy API usage. However, if a similar code knowledge graph generator of another language is developed, our method can also be scaled to the corresponding language. Our method is also limited in identifying specifications missing from the NLD, suggesting potential future work to create CQs about specifications “mentioned but not specified enough” in the NLD.

Ethical Concerns

One concern about the data is the issue of copyright. Liu et al. (2021) have checked the data policy of all 20 Kaggle competitions, in which none has copyright issues. Furthermore, they have contacted Kaggle's administrator and have made sure that the dataset collection procedure did not violate the platform's policy. We also check the license of open-source APIs when collecting documentation and make sure that there is no concern about copyright issues. Another concern about the data is that it might include privacy data. Again, we think that our data has a minimum risk of leakage of data with privacy concerns since we only collect data from the 20 Kaggle competitions where there is no concern of privacy data. The API documentation also has the minimum risk of containing data with privacy concerns.

Acknowledgements

We thank Xuye Liu and Dakuo Wang for providing the original dataset and source code for dataset preprocessing. We thank Nico Daheim, Ben Peters, Mert Tiftikci, Kexin Wang, Imbesat Hassan Rizvi, Dominic Petrak for their valuable feedback and suggestions on a draft of this paper. We thank the anonymous reviewers for their detailed and insightful comments.

This work has been funded by the LOEWE Distinguished Chair "Ubiquitous Knowledge Processing" (LOEWE initiative, Hesse, Germany), by EU's Horizon Europe Research and Innovation Actions (UTTER, contract 101070631), and by the Fundação para a Ciência e Tecnologia through contract UIDB/50008/2020.

References

Ibrahim Abdelaziz, Julian Dolby, Jamie McCusker, and Kavitha Srinivas. 2021. A toolkit for generating code knowledge graphs. In *Proceedings of the 11th on Knowledge Capture Conference*, K-CAP '21, page 137–144, New York, NY, USA. Association for Computing Machinery.

Ibrahim Abdelaziz, Julian Dolby, Jamie McCusker, and Kavitha Srinivas. 2022. Can machines read coding manuals yet? – a benchmark for building better language models for code understanding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2022)*.

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5436–5446, Hong Kong, China. Association for Computational Linguistics.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Mohammad Aliannejadi, Julia Kiseleva, Aleksandr Chuklin, Jeff Dalton, and Mikhail Burtsev. 2021. Building and evaluating open-domain dialogue corpora with clarifying questions. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4473–4484, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Mohammad Aliannejadi, Hamed Zamani, Fabio Crestani, and W. Bruce Croft. 2019. Asking clarifying questions in open-domain information-seeking conversations. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'19, page 475–484, New York, NY, USA. Association for Computing Machinery.

Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2123–2132, Lille, France. PMLR.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. If you use this software, please cite it using these metadata.

Ricardo Campos, Vítor Mangaravite, Arian Pasquali, Alípio Jorge, Célia Nunes, and Adam Jatowt. 2020. Yake! keyword extraction from single documents using multiple local features. *Inf. Sci.*, 509(C):257–289.

Shobhit Chaurasia and Raymond J. Mooney. 2017. Dialog for language to code. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 175–180, Taipei, Taiwan. Asian Federation of Natural Language Processing.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

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, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Zachary Eberhart and Collin McMillan. 2022. Generating clarifying questions for query refinement in source code search. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 140–151.

Meiqi Guo, Mingda Zhang, Siva Reddy, and Malihe Alikhani. 2021. Abg-coQA: Clarifying ambiguity in conversational question answering. In *3rd Conference on Automated Knowledge Base Construction*.

Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, and Nan Duan. 2022. Execution-based evaluation for data science code generation models. In *Proceedings of the Fourth Workshop on Data Science with Human-in-the-Loop (Language Advances)*, pages 28–36, Abu Dhabi, United Arab Emirates (Hybrid). Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Shuvendu K. Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2022. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950*.

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Xuye Liu, Dakuo Wang, April Wang, Yufang Hou, and Lingfei Wu. 2021. HAConvGNN: Hierarchical attention based convolutional graph neural network for code documentation generation in Jupyter notebooks. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4473–4485, Punta Cana, Dominican Republic. Association for Computational Linguistics.

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

Bodhisattwa Prasad Majumder, Sudha Rao, Michel Galley, and Julian McAuley. 2021. Ask what's missing and what's useful: Improving clarification question generation using global knowledge. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4300–4312, Online. Association for Computational Linguistics.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 314–319, Taipei, Taiwan. Asian Federation of Natural Language Processing.

Yuya Nakano, Seiya Kawano, Koichiro Yoshino, Katsuhito Sudoh, and Satoshi Nakamura. 2022. Pseudo ambiguous and clarifying questions based on sentence structures toward clarifying question answering system. In *Proceedings of the Second DialDoc Workshop on Document-grounded Dialogue and Conversational Question Answering*, pages 31–40, Dublin, Ireland. Association for Computational Linguistics.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

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

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A python natural language processing toolkit for many human languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 101–108, Online. Association for Computational Linguistics.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Sudha Rao and Hal Daumé III. 2018. Learning to ask good questions: Ranking clarification questions using neural expected value of perfect information. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2737–2746, Melbourne, Australia. Association for Computational Linguistics.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.1029*.

Stephen Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. 1995. Okapi at trec-3. In *Overview of the Third Text REtrieval Conference (TREC-3)*, pages 109–126. Gaithersburg, MD: NIST.

L. Tunstall, L. von Werra, and T. Wolf. 2022. *Natural Language Processing with Transformers: Building Language Applications with Hugging Face*. O'Reilly Media.

April Yi Wang, Dakuo Wang, Jaimie Drozdal, Xuye Liu, Soya Park, Steve Oney, and Christopher Brooks. 2021a. What makes a well-documented notebook? a case study of data scientists' documentation practices in kaggle. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI EA '21, New York, NY, USA. Association for Computing Machinery.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021b. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 1–10, New York, NY, USA. Association for Computing Machinery.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6045–6052, Online. Association for Computational Linguistics.

Jingjing Xu, Yuechen Wang, Duyu Tang, Nan Duan, Pengcheng Yang, Qi Zeng, Ming Zhou, and Xu Sun. 2019. Asking clarification questions in knowledge-based question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1618–1629, Hong Kong, China. Association for Computational Linguistics.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. 2022. Natural language to code generation in interactive data science notebooks. *arXiv preprint arXiv:2212.09248*.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. *arXiv preprint arXiv:2207.05987*.

Appendix

A Procedure of Identifying Key Operation.

We present our procedure for identifying key operations in Algorithm 1 as a detailed description of §2.1. Given an NLD-Code pair and all source codes from its corresponding notebook, our method first extracts operations for the entire notebook and selects operations corresponding to the code from the NLD-Code pair. We then identify key operations by keeping (i) operations from the same API submodule that have the shortest data flow path and (ii) operations that correspond to the last operation within the same line. Note that we also filter out operations that (i) are print functions, (ii) are numerical operations, and (iii) have no corresponding documentation.

B Preliminary Experiments on Identifying Missing Key Operations

We also considered code/documentation-trained models for computing similarities preliminarily. We experimented with RFLHU-BERTOverflow (Abdelaziz et al., 2022), which is trained on documentation-StackOverflowPosts pairs and performs similarly to the publicly unavailable RFLHU-CodeBERT in Abdelaziz et al. (2022). We obtained 75.59, 57.14, 55.56, and 56.34 in accuracy, precision, recall, and F1. This is substantially lower than all the results from Table 2.

C Example of Identifying if an Key Operation is Missing

We present an example of identifying if a key operation is missing figure 3. Given the key operations we have extracted (Figure 1b), we identify if a key operation is missing by comparing all its schema elements with schema elements of the NLD.

D Examples of Error Types

We analyzed predictions of MPNet$_{\text{base}}$qa-cos text encoder using our annotations. Table 12 shows examples of all types of FP and FN predictions we categorize. We also present in Table 13 the statistics of all FP and FN predictions.

E Annotation

We asked two Ph.D. students to annotate 200 NLD-Code pairs, respectively. It takes a volunteer about

Algorithm 1 Procedure of Extracting Key Operations

```
function EXTRACT KEY OPERATIONS AND DOCUMENTS(code_sample, code_notebook)
        ▷Step 1: Parse the code to build the graph and get operations for the notebook.
    ops_notebook ← GraphGen4Code(code_notebook).
        ▷Step 2: Find all operations corresponding to code from the NLD-Code pair
    ops_sample ← get_ops_sample(code_sample, code_notebook, ops_notebook)
        ▷Step 3: Pop operations of either import functions or numerical operations
    for op $\in$ ops_sample do
        if type(ops_sample) $\in$ {import_function, numerical_expression} then
            ops_sample.pop(op)
        ▷Step 4: Keep operations that are the last call of the same line of code
    for op $\in$ ops_sample do
        if op is not the last operation of the line then
            ops_sample.pop(op)
        ▷Step 5: Pop operations of print functions
    for op $\in$ ops_sample do
        if type(ops_sample) $\neq$ print_function then
            ops_sample.pop(op)
        ▷Step 6: Keep only operations that do not have other operations in its data flow path
    key_ops ← []
    for op $\in$ ops_sample do
        if $\{n \mid n \in \text{ops\_sample} \ \& \ n \in \text{op.path}\} = \varnothing$ then
            key_ops.append(op)
        ▷Step 7: Extract Documentations and keep only operations with documentation
     documentations ← []
    for op $\in$ key_ops do
        if op has no corresponding documentation then
            key_ops.pop(op)
        else
            op.documentation ← get_documentation(op)
    return key_ops
```

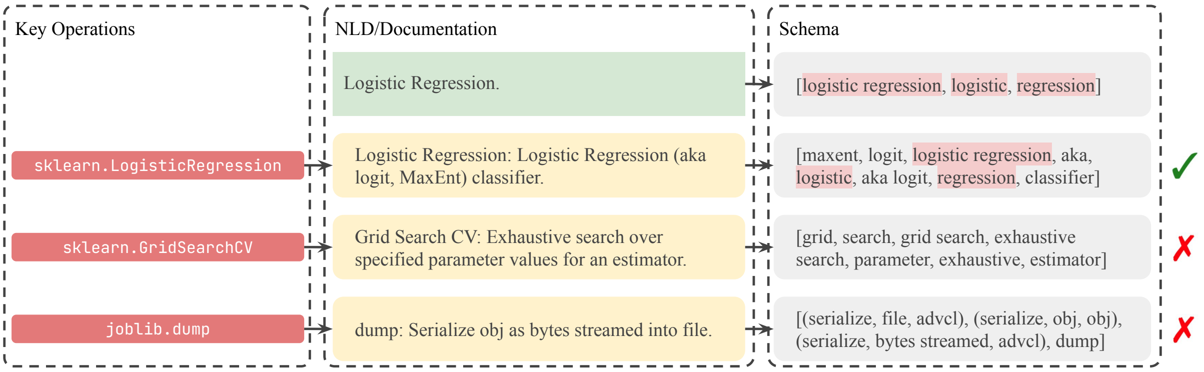

Key Operations
NLD/Documentation
Schema
Logistic Regression.
[logistic regression, logistic, regression]
sklearn.LogisticRegression
Logistic Regression: Logistic Regression (aka logit, MaxEnt) classifier.
[maxent, logit, logistic regression, aka, logistic, aka logit, regression, classifier]
✓
sklearn.GridSearchCV
Grid Search CV: Exhaustive search over specified parameter values for an estimator.
[grid, search, grid search, exhaustive search, parameter, exhaustive, estimator]
✗
joblib.dump
dump: Serialize obj as bytes streamed into file.
[(serialize, file, advcl), (serialize, obj, obj), (serialize, bytes streamed, advcl), dump]
✗

Figure 3: Example of identifying key operations with the example from Figure 1. ✓means that the key operation is *aligned*, and ✗means that the key operation is *missing*. Schema element pairs with the highest similarity scores are highlighted if the operation is predicted *aligned*.

2 hours to annotate. We show the guide in figure 4 and an example of annotation figure 5.

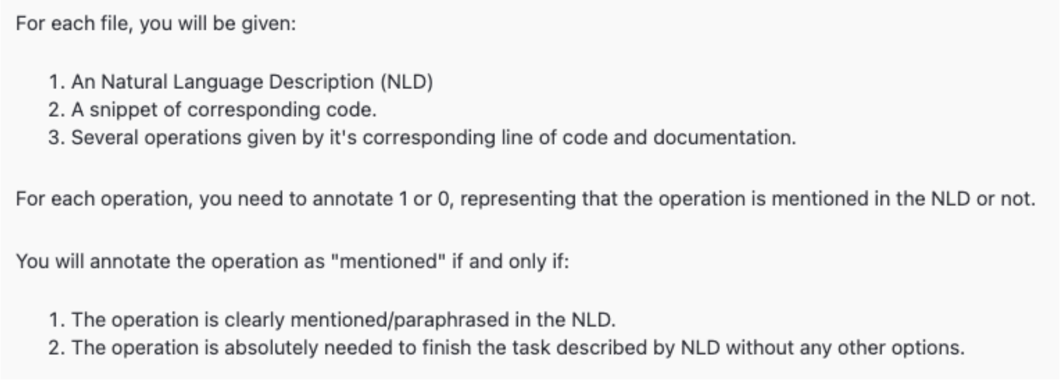
Discrepancy of annotation between development and test set. We noticed the discrepancy of Fleiss Kappa between the development and test set. We then asked annotators to provide reasons for different annotations. As a result, subjectivity is the main reason for differences between annotations. An example is shown in figure 5, where fitting the model is not directly mentioned yet can be inferred from the NLD. We also find that the test set contains more examples like this one, leading to a discrepancy of Fleiss Kappa between the development and the test set. We accept this difference as subjectivity is part of deciding *whether an operation is mentioned*.

F Examples of CodeClarQA Dataset

We present examples from our dataset in Table 14.

Type of Error	Example	Explanation
Taxonomy (FP)	**NLD**: We've addressed a lot of the issues holding us back when using a linear model... **Code Line:** LCV = LassoCV() **Doc:** Lasso CV: Lasso linear model with iterative fitting along a regularization path.	Lasso linear model should be distinguished from linear model.
Element Pair (FP)	**NLD**: ...we concatenate the two sets while remembering the index so we can split it later again. **Code Line:** train_features = train.drop(['SalePrice'], axis=1) **Doc:** drop: Make new Index with passed list of labels deleted.	Method identify drop as non-missing only by seeing index in both the NLD and the documentation.
Multiple Operations (FP)	**NLD**: Categorical Features. Let us look at the missing values in categorical features in detail. **Code Line:** categorical_features.isnull().sum().sort_values(ascending=False) **Doc:** sort values: Sort the Categorical by category value returning a new	Only isnull, sum, sort_values together refer to look at the missing values in categorical features.
Model (FP)	**NLD**: The variable importances from the boosted model on the reduced dataset. **Code Line:** sns.set_style('darkgrid') **Doc:** set style: Set the parameters that control the general style of the plots.	Method yields wrong prediction (positive) by comparing dataset and (set, plots, *obj*).
Argument (FN)	**NLD**: Transforming some numerical variables. **Code Line:** all_data['MSSubClass'] = all_data['MSSubClass'].apply(str) **Doc:** apply: Apply a function along an axis of the Data Frame.	apply(str) corresponds to the NLD, not apply itself.
Element Missing (FN)	**NLD**: The ' Age', ' Cabin', ' Embarked', ' Fare' columns have missing values. **Schema:** [embarked, missing, columns, age, cabin, fare] **Code Line:** full['Embarked'].fillna('S', inplace=True) **Doc:** fillna: Fill NA/ NaN values using the specified method. **Schema:** [(fill, fillna, *nsubj*), (fill, method, *obj*)]	Method fails to extract NA and NaN and compare them to missing.
Paraphrase (FN)	**NLD**: Train again for all data and submit. **Code Line:** rfc.fit(X_train_all, y_train_all) **Documentation:** fit: Fit the calibrated model.	Model cannot yield high similarity scores between train and fit.
Abbreviation (FN)	**NLD**: GBDT: . **Code Line:** gbdt = GradientBoostingClassifier(...) **Documentation:** Gradient Boosting Classifier: Gradient Boosting for classification.	Model cannot yield high similarity scores between gbdt and Gradient Boosting Classifier.

Table 12: Examples of all types of human evaluated errors in the human-annotated validation and test sets. We provide true positive (TP), false positive (FP), and false negative (FN) examples. Category refers to the assigned category of prediction by human evaluation. Key operations and schema element pairs with the highest similarity scores are highlighted.



For each file, you will be given:

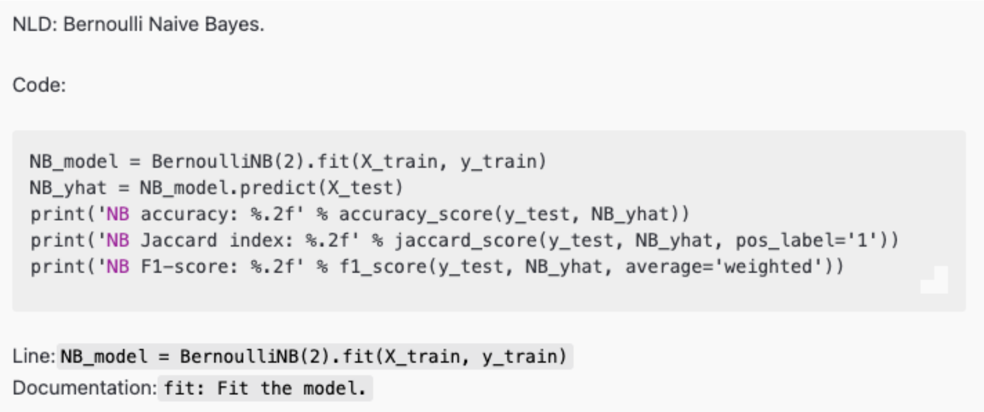
1. An Natural Language Description (NLD)
2. A snippet of corresponding code.
3. Several operations given by it's corresponding line of code and documentation.

For each operation, you need to annotate 1 or 0, representing that the operation is mentioned in the NLD or not.

You will annotate the operation as "mentioned" if and only if:

1. The operation is clearly mentioned/paraphrased in the NLD.
2. The operation is absolutely needed to finish the task described by NLD without any other options.

Figure 4: The annotation guide.



NLD: Bernoulli Naive Bayes.

Code:

```
NB_model = BernoulliNB(2).fit(X_train, y_train)
NB_yhat = NB_model.predict(X_test)
print('NB accuracy: %.2f' % accuracy_score(y_test, NB_yhat))
print('NB Jaccard index: %.2f' % jaccard_score(y_test, NB_yhat, pos_label='1'))
print('NB F1-score: %.2f' % f1_score(y_test, NB_yhat, average='weighted'))
```

Line: `NB_model = BernoulliNB(2).fit(X_train, y_train)`

Documentation: `fit: Fit the model.`

Figure 5: An example of annotation. Given an NLD-Code pair and key operations (with documentation) of it, an annotator is required to annotate each key operation as *aligned* or *missing*

Error Type	Freq		ER (%)	
	Dev	Test	Dev	Test
Taxonomy (FP)	3 (.33)	3 (.50)	7.32	8.57
Element Pair (FP)	3 (.33)	3 (.50)	7.32	8.57
Multiple Operations (FP)	2 (.22)	0 (.00)	4.87	0.00
Model (FP)	1 (.11)	0 (.00)	2.43	0.00
Argument (FN)	4 (.57)	4 (.67)	4.08	4.35
Element Missing (FN)	1 (.14)	1 (.17)	1.02	1.09
Paraphrase (FN)	1 (.14)	1 (.17)	1.02	1.09
Abbreviation (FN)	1 (.14)	0 (.00)	1.02	0.00

Table 13: Statistics of all FP and FN predictions. Error types are defined in 12. Freq refers to the frequency, with relative frequency included in the parenthesis. Error rates (ER) are computed on the corresponding predictions.

NLD: So, 18 categorical features and 10 numerical features to clean. We start with the numerical features, first thing to do is have a look at them to learn more about their distribution and decide how to clean them. > 2.2 Numerical features.
CQ1: Do you want to call ’pandas.head’ documented as ’Return the first ‘n‘ rows.’?"
A1: Yes.
CQ2: Do you want to call ’pandas.fillna’ documented as ’Fill NA/ NaN values using the specified method.’?
A2: Yes.
Code:

```
NAnum.head()
c[’MasVnrArea’] = c.MasVnrArea.fillna(0)
c[’LotFrontage’] = c.LotFrontage.fillna(c.LotFrontage.median())
c[’GarageYrBlt’] = c[’GarageYrBlt’].fillna(1980)
```

NLD: There are many libraries out there that support one-hot encoding but the simplest one is using method. This function is named this way because it creates dummy/indicator variables.
No CQAs.
Code:

```
Train_Master = pd.get_dummies(Train_Master, columns=[’Sex’, ’Pclass’, ’Embarked’], drop_first=True)
Train_Master.drop([’PassengerId’, ’Name’, ’Ticket’], axis=1, inplace=True) test_ids = Test_Master.loc[:, ’PassengerId’]
Test_Master = pd.get_dummies(Test_Master, columns=[’Sex’, ’Embarked’, ’Pclass’], drop_first=True)
Test_Master.drop([’PassengerId’, ’Name’, ’Ticket’], axis=1, inplace=True) Train_Master.head()
```

NLD: Need to look at the y_log relationship since that is what we will be predicting in the model.
CQ1: Do you want to call anything related to ’plot’? If yes, which one?
A1: Yes, I want to call ’matplotlib.plot’.
CQ2: Do you want to call ’matplotlib.scatter’ documented as ’A scatter plot of *y* vs. *x* with varying marker size and/or color.’?
A2: Yes.
Code:

```
NAnum.head()
y = np.exp(11.1618915) * np.exp(0.000570762509 * x_data)
plt.plot(x_data, np.log(y_data), ’o’)
plt.scatter(x_data, np.log(y), c=’red’)
```

NLD: Ensembling is a way to increase performance of a model by combining several simple models to create a single powerful model. I will use voting method in this kernal.
CQ1: Do you want to call anything related to ’model/algorithm’? If yes, which one?
A1: Yes, I want to call ’sklearn.RandomForestClassifier’.
CQ2: Do you want to call anything related to ’model/algorithm’? If yes, which one?
A2: Yes, I want to call ’sklearn.LogisticRegression’.
CQ3: Do you want to call anything related to ’model/algorithm’? If yes, which one?
A3: Yes, I want to call ’sklearn.DecisionTreeClassifier’.
CQ4: Do you want to call anything related to ’model/algorithm’? If yes, which one?
A4: Yes, I want to call ’sklearn.GaussianNB’.
CQ5: Do you want to call anything related to ’score’? If yes, which one?
A5: Yes, I want to call ’sklearn.cross_val_score’.
Code:

```
from sklearn.ensemble import VotingClassifier
estimators = [(’RFor’, RandomForestClassifier(n_estimators=100, random_state=0)), (’LR’, LogisticRegression(C=0.05, solver=’liblinear’)), (’DT’, DecisionTreeClassifier()), (’NB’, GaussianNB())]
ensemble = VotingClassifier(estimators=estimators, voting=’soft’)
ensemble.fit(train_X, train_Y.values.ravel())
print(’The accuracy for ensembled model is:’, ensemble.score(test_X, test_Y))
cross = cross_val_score(ensemble, X, Y, cv=10, scoring=’accuracy’)
print(’The cross validated score is’, cross.mean())
```

Table 14: Examples of the CodeClarQA dataset.

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
No section number, "limitations" section (page 8 and 9)

☑ A2. Did you discuss any potential risks of your work?
Section 5.1

☑ A3. Do the abstract and introduction summarize the paper's main claims?
No section number, "abstract" (page 1) and "introduction" (page 1,2)

☒ A4. Have you used AI writing assistants when working on this paper?
Left blank.

B ☑ Did you use or create scientific artifacts?

Section 2.1, 2.4

☑ B1. Did you cite the creators of artifacts you used?
Section 2.1, 2.4

☑ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
Section 2.1, 2.4, "Ethical Concerns" section (page 9)

☑ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
Section 2.1, 2.4

☑ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
No section number, "Ethical Concerns" section (page 9)

☑ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
Section 2.1, 2.4, "Ethical Concerns" section (page 9)

☑ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.
Section 2.4

C ☑ Did you run computational experiments?

Section 4

☑ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
Section 4

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

☑ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
Section 5

☑ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
Section 5

☑ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
Section 4

D ☑ Did you use human annotators (e.g., crowdworkers) or research with human participants?
Section 4.1

☑ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
Appendix D

☑ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
Section 4.1

☑ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
Section 4.1

□ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
Not applicable. The only requirement for annotators is that they are experts in coding python.

☒ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
There is no need to do it. The only requirement is that they are experts in coding python.