CodeExp: Explanatory Code Document Generation

Haotian Cui[1]*, **Chenglong Wang**[2], **Junjie Huang**[3], **Jeevana Priya Inala**[2]
Todd Mytkowicz[2], **Bo Wang**[1], **Jianfeng Gao**[2], **Nan Duan**[2]†
[1]University of Toronto, [2]Microsoft Research, [3]Beihang University
{ht.cui, bowang.wang}@mail.utoronto.ca,
huangjunjie@buaa.edu.cn
{chenwang, jinala, toddm, jfgao, nanduan}@microsoft.com,

Abstract

Developing models that can automatically generate detailed code explanation can greatly benefit software maintenance and programming education. However, existing code-to-text generation models often produce only high-level summaries of code that do not capture implementation-level choices essential for these scenarios. To fill in this gap, we propose the *code explanation generation* task. We first conducted a human study to identify the criteria for high-quality explanatory docstring for code. Based on that, we collected and refined a large-scale code docstring corpus and formulated automatic evaluation metrics that best match human assessments. Finally, we present a multi-stage fine-tuning strategy and baseline models for the task. Our experiments show that (1) our refined training dataset lets models achieve better performance in the explanation generation tasks compared to larger unrefined data (15× larger), and (2) fine-tuned models can generate well-structured long docstrings comparable to human-written ones. We envision our training dataset, human-evaluation protocol, recommended metrics, and fine-tuning strategy can boost future code explanation research. The code and annotated data are available at https://github.com/subercui/CodeExp.

1 Introduction

Code documentation improves program comprehension (Garousi et al., 2015) and reduces software maintenance cost (Chen and Huang, 2009). Recently, many automated code summary tools have been developed to reduce the effort of document creation: for example, Denigma[1] is an IDE extension for generating inline function summaries; GitHub Copilot Labs[2] is a code summary model built on top of Codex (Chen et al., 2021a) for generating explanations for AI generated code. However, these existing code summary tools focus on the generation of short high-level descriptions of source code semantics (Haiduc et al., 2010; Roy et al., 2021; Zhu and Pan, 2019), and code summary alone is insufficient to meet the software understanding and maintenance need: A recent survey shows 85% developers expect tools to generate method-level documentations explaining the functionalities, usage and design rationales of code (Hu et al., 2022). For example, as shown in Figure 1, the code summary captures only the high-level code functionality, while the explanatory docstring explains arguments, return values, and its computation process in a detailed and informative way. However, due to the lack of training and evaluation resources, few code explanation models have been developed.

In this work, we introduce the **code explanation generation** task. We provide (1) the training corpus, (2) fine-tuning strategy, and (3) human-evaluation protocol and recommended automatic evaluation metrics to support developing code explanation models. Our contributions include:

- We provide a python code-docstring corpus *CodeExp*, which contains (1) a large partition of 2.3 million raw code-docstring pairs, (2) a medium partition of 158 thousand pairs refined from the raw corpus using a learned filter, and (3) a partition of 13 thousand pairs with rigorous human annotations. Our data collection process leverages an annotation model learned from human annotations to automatically filter high quality code-docstring pairs from raw GitHub datasets.
- We propose a two-stage strategy for fine-tuning large language models using collected data — first with the raw data and then with medium-size refined data. Our experiments show that the best fine-tuned model achieves human-comparable per-

*Work was done during an internship at Microsoft.
†Corresponding author.
[1]https://denigma.app/
[2]https://github.com/github/feedback/discussions/8308

Findings of the Association for Computational Linguistics: EMNLP 2022, pages 2342 - 2354
December 7-11, 2022

Code:

```
def make_rng(rng_or_seed=None, default_seed=None, constructor=None):
    if (rng_or_seed is not None) and isinstance(rng_or_seed, RNG):
        rng = rng_or_seed
    elif (rng_or_seed is not None):
        rng = constructor(rng_or_seed)
    elif (default_seed is not None):
        rng = constructor(default_seed)
    else:
        rng = constructor(42)
    return rng
```

Summary:

```
Returns a random number generator.
```

Explanatory Docstring:

```
Returns a random number generator.
The RNG object is generated using the first of these cases that
produces a valid result:
1) rng_or_seed itself
2) constructor(rng_or_seed)
3) constructor(default_seed)
4) constructor(42)

Parameters:
rng_or_seed (int or RNG): If `rng_or_seed` is a random number
generator, then it is returned. If `rng_or_seed` is an integer, then
a random number generator is created using `constructor` and seeded
with `rng_or_seed`.
default_seed (int): Seed used if rng_or_seed is None.
constructor (function or class): Must return a RNG object.
constructor is called with rng_or_seed, default_seed or 42.

Returns:
An RNG object.
```

Figure 1: Example code paired with summary and explanatory docstring. Difference between two styles: Summaries outline the highest-level intent of the code. Docstrings are more **informative and detailed**, explaining the semantics of specific code pieces.

formance and highlights the importance of high-quality data for the code explanation task.

- We evaluated our models on seven automatic evaluation metrics and examined their consistency with respect to the human evaluation of 180 test examples. Our study shows that BLEU (Papineni et al., 2002) and METEOR (Banerjee and Lavie, 2005) best reflect code generation quality, and we recommend using them in future research.

2 Code Explanation Generation

Application scenarios. We focus on the generation of code explanations that describe both low-level and high-level code semantics. The automatic code explanation tool can benefit developers in many scenarios. For example, the tool can reduce software engineers developing effort by automatically generating function comments during development; it can help learners and codebase maintainers better understand undocumented code; it can also explain code generated by code generation models like Codex for the developer to better understand and verify its correctness. Note that in these scenarios, because the developer needs to understand both design rationale and implementation details of the code, an explanation covering detailed code semantics would be more appropriate than a short high-level description. For example, if a developer aims to create a test for the function make_rng in Figure 1 when maintaining a codebase, it is crucial to understand how the variable rng_or_seed is defined and used.

Task definition. Based on these observations, we define the code explanation task as the text generation given code snippets (functions), where the generated texts describe the code semantics. Concretely, a high quality code explanation should meet the following criteria: the description should be informative, covering important code behaviors, coherent with the semantics of source code, fluent, and grammatically correct. We follow these criteria to set up our annotation and evaluation protocols in Section 3.1 and 5, respectively.

Challenges. The first key challenge for developing code explanation models is the shortage of high-quality paired training and evaluation data. Despite the existence of large-scale public code corpora, directly using functions and their comments for modelling is not ideal because these comments can be oversimplified or misleading: Clement et al. (2020) found 44% of python function documents are very short in one-line style, and Wen et al. (2019) showed code changes rarely (<20%) trigger comments updates, potentially making the inconsistency between code and comments a severe issue.

Second, developing code explanation models is challenging. Besides the need for understanding code logic to generate a global summary, the explanation model also needs to generate detailed comments based on fine-grained local code structures (e.g., examine the control and data flow in order to explain how a variable is used).

Third, while prior work (Gros et al., 2020; Roy et al., 2021) empirically studied criteria of high quality code summaries, translating these criteria/guidelines into actionable automatic evaluation metrics for code explanation remains a challenge.

We next present how we collect high quality datasets, define evaluation metrics and fine-tune language models to address the above challenges.

3 CodeExp Data Collection

This section describes our data selection process. The code explanation corpus (*CodeExp*) consist of three sets of code-docstring pairs: (1) CodeExp(annotated) with 13K human annotated pairs. (2) CodeExp(refined) with 158K pairs filtered by a

trained model. (3) CodeExp(raw) with 2.3M unlabelled pairs.

3.1 Examine docstring quality with human annotations - CodeExp(annotated)

In order to understand how developers evaluate the quality of explanatory docstrings, we first conduct a user study to let developers annotate quality of code-docstring pairs using the code-doc-corpus collected by Barone and Sennrich (2017). The corpus contains 109,108 parallel python functions and docstrings. We automatically filtered the dataset to only include qualified examples. The filtered examples consist of the data pairs where (1) the number of code lines is within 6 to 30, (2) the number of lines in the docstring is larger than 3, and (3) the Cyclomatic Complexity[3] of the code is larger than 3. In total, this yields 13,186 valid examples. We hire external annotators to label this subset.

Annotation Our annotation protocol is developed through several pilots and further updated with hard examples as the annotation progresses. Annotators are asked to make three judgements for each pair of code and docstring: (1) **General adequacy:** The docstring should describe the main logic of the code, i.e. containing at least one sentence describing how the code handles the input or what it computes given the input. (2) **Coverage:** If the code contains outer-level if/else or try/except blocks, check whether the docstring describes the semantic of each block. (3) **Coherence:** Check whether the documentation (if any) of parameter types and returns match the code semantics.

For each pair of code and docstring, the human annotator is asked to give a score from 0 (worst) to 3 (best) for each step if it is applicable (only step 1 is always applicable); otherwise, the annotator leaves a blank score. For the coverage evaluation, the annotator will also mark the specific code spans and text spans that are associated with code blocks (Figure 2). Due to the concern of feasibility, we do not require the step 2 annotation to cover all aspects of details, but only the branching blocks.

We refer to the human-annotated data as the **CodeExp(annotated)** in later sections. We show the statistics of annotations in Appendix A.1. In the annotated result, we found the human-written docstrings mostly perform well in explaining the general logic and provide accurate type defines, but are less optimal considering the coverage requirement. In other words, examples have high scores for step 1 and step 3, but the scores for step 2 interestingly diverge. For the 11,900 code with branching blocks, 6,300 examples do not describe any block (step 2 score equals zero). The rest 5,400 examples (33%) describe at least one code block.


```
{
"docstring": "Pulls all flashed messages from the session and returns
them. ... whitelist of categories to limit return values",
"code": "
def get_flashed_messages(with_categories=False, category_filter=[]):
     flashes = _request_ctx_stack.top.flashes
     if (flashes is None):
          _request_ctx_stack.top.flashes = flashes = (
               session.pop('_flashes') if ('_flashes' in session)
               else [])
     if category_filter:
          flashes = list(filter(
               (lambda f: (f[0] in category_filter)), flashes))
     if (not with_categories):
          return [x[1] for x in flashes]
     return flashes
",
"step1": "3",
"step2": "2",
"step3": "3",
"match": [
     {
     "selectedCode": "if (not with_categories): \n return [x[1] for
                     x in flashes] ",
     "CodeSpan": [357,421],
     "selectedDoc": "``False`` gives just the message text)",
     "DocSpan": [685,723]
     },
     {
     "selectedCode": " if category_filter: \n flashes =
                      list(filter((lambda f: (f[0] in
                      category_filter)), flashes))",
     "CodeSpan": [254,352],
     "selectedDoc": "Filter the flashed messages to one or more
                    categories by providing those \n categories in
                    `category_filter`. This allows rendering
                    categories in \n separate html blocks. ",
     "DocSpan": [331,501]
     }
     ]
}
```


Figure 2: An annotated example of high-score code docstring pair. Integer scores of three steps are provided. The "CodeSpan" field extracts the specific code lines described by the texts of the "DocSpan" field.

3.2 CodeExp(raw) with Open-source Pairs

As it is economically infeasible to manually annotate a dataset large enough with code-docstring pairs for training a machine learning model, we instead create a suboptimal dataset with unlabelled pairs. Following (Husain et al., 2019), we pair the code function with its corresponding documentation to form a code-docstring pair. We leverage open-source python repositories in GitHub to collect CodeExp(raw). To ensure the code quality, we only keep repositories with more than 60 stars. This step yields around 55,000 repositories by December 2021. We downloaded all Python files with *'.py'* extensions and parsed the source code into abstract syntax trees (AST) with Tree-Sitter[4]. Then we include the functions provided with docstrings. Finally, we collected a corpus of 2,285,387 pairs of Python function code and docstrings. To the best of our knowledge, CodeExp(raw) is one of the largest

[3] https://radon.readthedocs.io/en/latest/intro.html#cyclomatic-complexity

[4] https://tree-sitter.github.io/tree-sitter/

datasets with parallel programming language (PL) and natural language (NL) till now. In comparison, CodeSearchNet (Husain et al., 2019) contains 457,461 samples of python code.

3.3 Data selection with a learned filter - CodeExp(refined)

The collected code and docstrings from Github are of mixed qualities and potentially introduce noise if used for training. Hence, we aim to refine a higher-quality subset from it for modeling. Our key insight here is to train a machine learning filter to mimic the human annotators based on the annotated dataset, and apply the learned filter to refine the raw data. The filter is fine-tuned from a pretrained BERT base model (uncased) on the collected human annotations. It takes as input the code and docstring pairs and predicts the step 1 and 2 scores [5]. The target scores are normalized to $[0, 1]$. We used 11208 examples, 85% of CodeExp(annotated), for training and the rest for validation. The model achieved mean square errors (mse) of 0.027 and 0.018 for step 1 and 2, respectively, on the validation set.

We apply the same workflow as in Section 3.1 to filter the raw 2.3M corpus of CodeExp(raw): we used the same complexity and length threshold to select candidate examples, and then applied the ML-filter. Finally, we selected the qualified data pairs with predicted step 1 and 2 scores greater than 1.0 (after scaling back). In this way, we collected 158,024 refined examples, named as the **CodeExp(refined)** partition.

Partition	#Examples	Quality	Annotated By
CodeExp(raw)	2,285,387	Mixed	-
CodeExp(refined)	158,024	High	Machine
CodeExp(annotated)	13,186	Mixed	Human

Table 1: Data statistics. The raw and refined partitions are collected from GitHub. The annotated partition is selected from code-doc-corpus (Barone and Sennrich, 2017).

Table 1 shows the statistics for the three partitions. Note that the annotated subset's quality is "mixed" because it contains both low and high scored examples annotated by human (we only use higher scored ones for testing in Section 4.2). Because the refined partition is refined by the learned filter, it better matches the definition of code explanations (Section 2). An example with human annotations is shown in Figure 2.

[5] Because most annotated examples have high step 3 scores, and low-score examples are inconsistent between annotators, we excluded step 3 score when training the ML filter.

Models	#Params	Pretrained w/
GPT-2-base (Radford et al., 2019)	117M	NL
GPT-Neo-13 (Black et al., 2021)	1.3B	NL, PL
GPT-Neo-27 (Black et al., 2021)	2.7B	NL, PL
CodeT5 (Wang et al., 2021)	220M	PL*

Table 2: Backbone models for fine-tuning. (*) CodeT5 was mainly trained with PL. The NL training data contained only the code comments.

4 Experiment Settings

In this paper, we formulate the code explanation generation as a sequence-to-sequence problem, where the source sequence is a function code (including both function signature and body), and the target sequence is an explanatory documentation string. We select four strong pretrained language models for programming language and fine-tune the models on our proposed CodeExp dataset. We next introduce the baseline models, experiment settings, and evaluation metrics.

4.1 Baseline models

We evaluate four popular pretrained language models in this paper: (1) *GPT2-base* (Radford et al., 2019), a transformer decoder model pretrained with only NL; (2) *GPT-Neo* (Black et al., 2021), a series of GPT-style decoders varying in parameter size and pretrained with the large PILE corpus (Gao et al., 2020) containing both NL and PL; (3) *CodeT5* (Wang et al., 2021), a transformer encoder-decoder model pretrained with NL and PL. (4) *Codex* (Chen et al., 2021b) the state-of-the-art model trained on Github code of multiple programming languages including python. We fine-tune these models with our collected data to test the performance except for Codex, with which we only report its zero-shot performance due to the inaccessibility of model weights.

4.2 Fine-tune

We fine-tune the baseline models with the collected data of different sizes and qualities (Table 2). This yields three fine-tuning strategies: **S1.** Fine-tune on all collected examples, i.e. CodeExp(raw); **S2.** Fine-tune on the refined subset, CodeExp(refined); **S3.** Fine-tune on the raw and refined partitions consecutively, in a "curriculum learning" manner. For each strategy, we leave out 1% of the raw partition and/or 5% of the refined partition for validation.

For evaluation, we select examples with high scores from the CodeExp(annotated) and remove duplicates if they also appear in the raw or refined partition (Appendix A.2). This generates a high-quality test set of 2,677 examples.

The pretrained models are fine-tuned using cross-entropy loss on 16 Nvidia V100 32GB GPUs. We select the checkpoint with the best perplexity score on the corresponding validation set for further evaluation. For strategy S3, the best checkpoint after fine-tuning on CodeExp(raw) is used as the starting point for the next phase on CodeExp(refined). The hyperparameters, including the max tokens, learning rate, batch size, epoch numbers, etc., are listed in Appendix Table 7. At inference time, we sample the top generated text using the default inference settings (Appendix A.3).

4.3 Automatic metrics

We adopt four existing metrics widely used in related tasks and propose two new metrics dubbed *CER* and *CodeBERTScore* to evaluate our models. The efficacy of each metric is verified in the next section against human evaluation. We include both statistical and recent model-based metrics.

Statistical Metrics BLEU (Papineni et al., 2002), ROUGE (Lin and Och, 2004), METEOR (Banerjee and Lavie, 2005) are three commonly used metrics in code summarization and machine translation. These methods compute the matching of n-grams between candidate and target texts in various manners. We use the sentence BLEU (with smoothing method 4) and METEOR interfaces provided by NLTK[6]. For ROUGE, we use the F1 score of ROUGE-1 and ROUGE-L.

We propose a new metric dubbed Common Entity Recall (*CER*). It first computes the number of common 1-grams shown in code, generated, and reference docstrings. Then it is divided by the number of common 1-grams of the code and reference docstring. The intuition is that we found the common 1-grams of the code and reference docstrings often contain important variable names, function identifiers, or important keywords (e.g., if, int).

Model-based Metrics BERTScore (Zhang et al., 2019) is an automatic metric that employs a BERT model to measure the similarity between generation and reference. The semantic similarity is computed as cosine similarities between the average token embeddings of generated and target texts.

As BERTScore is only for natural language, we propose CodeBERTScore as an adapted version of BERTScore for evaluating code-related tasks. This metric is built by replacing the language model in BERTScore with CodeBERT (Feng et al., 2020), a state-of-the-art language model pretrained with code and natural language. We use the average token embeddings of the 9th layer in CodeBERT to compute similarity.

5 Metric selection based on human evaluation

We conducted human evaluations on generated docstrings and compared the aforementioned automatic metrics against the human-eval results. The protocol of our human evaluation is designed to cover four important aspects: **A1**. General adequacy, **A2.** Coverage, **A3.** Coherence, **A4.** Fluency. Annotators rate for each aspect a score within 0-4 on the 5-point Likert scale (Likert, 1932). The detailed setup is listed in Appendix A.4. Aspects 1,3,4 have been used in both machine translation (Reiter, 2018) and code summarization studies (Song et al., 2019; Roy et al., 2021). The coverage aspect emphasizes the preference for informative explanations of code pieces. Notably, these scores provide reference-free assessments. Both the original reference docstring (identity is hidden) and generated ones are provided to annotators.

We calculate the adapted Kendal's τ (Graham et al., 2015) to measure the agreement between automatic metrics and human evaluation. For an arbitrary pair of two examples, it considers whether two metrics both prefer one example to the other. The τ value is the ratio difference of concordant (Con) and discordant (Dis) pairs,

$$\tau = \frac{|\#\texttt{Con} - \#\texttt{Dis}|}{|\#\texttt{Con} + \#\texttt{Dis} + \#\texttt{Tie}|}, \quad (1)$$

where # denotes the number of pairs. The concordant, discordant and tie pairs are calculated as in Table 3, where s_1 and s_2 are the scores for the two docstrings within a pair.

6 Results and discussion

6.1 Results of auto-metric evaluations

We evaluated the generated docstrings of all aforementioned models (in Section 4.1). We also in-

[6] https://www.nltk.org/

		Metric		
		$s_1 < s_2$	$s_1 = s_2$	$s_1 > s_2$
Human	$s_1 < s_2$	concordant	tie	discordant
	$s_1 = s_2$	-	-	-
	$s_1 > s_2$	discordant	tie	concordant

Table 3: Calculate concordant and discordant pairs.

cluded the *CodeT5* checkpoint for code summarization, i.e. *CodeT5-multi-sum*[7]. The evaluated results are shown in Table 4. In latter sections, we use the notion "[model name]-(raw), -(refined), -(r+r)" representing the model fine-tuned using the strategy S1, S2, and S3, respectively.

We observed all fine-tuned models have significant improvements over off-the-shell versions across all metrics, for example, the BLEU scores increased from below 0.4 (*GPT-Neo13*, *CodeT5-multi-sum*) to around 10.0 (*GPT-Neo13-(r+r)*, *CodeT5-(r+r)*). Since *GPT-Neo* and *CodeT5-multi-sum* are reported as strong summarization baselines (Wang et al., 2021), this result again highlights the large difference between summaries and explanatory docstrings.

Two-stage fine-tuning achieves best performances. For the three fine-tuning strategies (**S1-3** in Section 4), S3 yields the best performance for all baseline models. To recall, it trains on all collected examples (i.e. CodeExp(raw)) and the high-quality partition (i.e. CodeExp(refined)) consecutively. Comparing across models, fine-tuned CodeT5 models perform best for S2 and S3 fine-tuning. Especially, *CodeT5-(r+r)* achieves the highest scores with respect to 6 out of 7 metrics.

6.2 Results of human evaluations

We randomly select 180 examples from the test set and collect human evaluations using the protocol in Section 5. For each evaluated docstring, the annotator gives four scores for the aspects **A1-4** and an overall average score is computed as well. The evaluated docstrings are generated using 6 fine-tuned models given the selected 180 python functions. The models and respective results are shown in Table 5. For comparison, we also include the human-written reference docstrings in the original codebases and two strong baselines using the OpenAI Codex API, i.e. Codex-PyDoc and Codex-Py2NL. These two APIs generate docstrings and NL explanations, respectively, and we follow the official settings (Appendix A.5).

Comparing the overall scores in Table 5, the relative superiority of models are largely consistent with the results of auto-metrics (Table 4): (1) The fine-tuning strategies (S2, S3) using high-quality data partition outperforms S1. In fact, we found that fine-tuning using only CodeExp(raw) would often generate short one-line texts (like summarization) due to the majority of one-line docstring in the data, and this explains the annotators' low rating to CodeT5-(raw). (2) CodeT5-(refined) and CodeT5-(r+r) outperform other models.

Achieving human-comparable performance. We also observe that CodeT5-(refined) achieves comparable overall scores as the human-written references. Especially, it has a higher *Coverage* score than the references, indicating more detailed explanations. We plot in Figure 3 the (kernel density estimated) distribution of overall scores across 180 examples. The CodeT5-(refined) has the highest density accumulated at the high score range (3.5-4.0), and the distribution is very close to the distribution of reference docstrings.

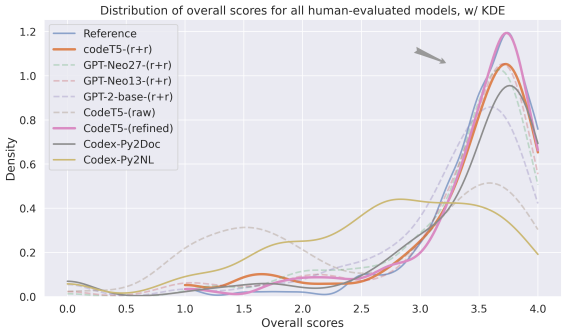

Figure 3: **Distribution of overall scores on 180 human-evaluated examples.** The best model has comparable performance with human-written reference docstrings.

We also observe the best fine-tuned models outperform *Codex-Py2Doc* and *Codex-Py2NL* with respect to all evaluation aspects, although the parameter size of Codex (12B) is 60 times larger than the fine-tuned CodeT5.

BLEU and METEOR are most consistent with human evaluations. To examine the consistency between automatic metrics and human evaluation, we calculated the adapted Kendal's τ (Section 5) and show the results in Table 6. We found BLUE and METEOR mostly match the human evaluations for aspects A1,3,4 and the overall score. For the aspect of A2, Coverage, the newly introduced

[7] https://huggingface.co/Salesforce/codet5-base-multi-sum

Model	ROUGE-1 f	ROUGE-L f	BLEU	CER	METEOR	BERTScore	CodeBERT Score
GPT-2-base							
- CodeExp(raw)	0.2557	0.2443	4.42	0.4580	18.55	83.87	77.46
- CodeExp(refined)	0.2462	0.2357	4.26	0.4890	19.00	83.55	77.48
- CodeExp(r+r)	0.2623	0.2520	5.19	0.4978	20.30	83.92	77.91
GPT-Neo13							
- w/o fine-tune	0.0694	0.0646	0.40	0.1294	5.71	78.19	67.62
- CodeExp(raw)	0.2894	0.2769	7.51	0.4805	21.85	84.46	78.31
- CodeExp(refined)	0.2954	0.2815	7.36	0.5570	23.58	84.24	78.46
- CodeExp(r+r)	**0.3265**	**0.3128**	**10.45**	0.5524	**26.31**	**84.78**	**79.26**
GPT-Neo27							
- w/o fine-tune	0.1480	0.1380	0.82	0.2285	10.06	78.40	71.99
- CodeExp(raw)	0.2953	0.2818	7.72	0.4895	22.42	84.31	78.34
- CodeExp(refined)	0.2955	0.2816	8.21	0.5285	23.66	84.33	78.41
- CodeExp(r+r)	**0.3298**	**0.3154**	**10.72**	**0.5560**	**26.87**	**84.86**	**79.28**
CodeT5							
- multi-sum	0.1507	0.1392	0.21	0.1240	5.58	82.35	74.29
- CodeExp(raw)	0.2652	0.2530	5.39	0.3851	17.51	83.84	77.42
- CodeExp(refined)	0.3175	0.3016	8.02	**0.5536**	24.38	84.67	78.90
- CodeExp(r+r)	**0.3415**	**0.3256**	**9.91**	**0.5695**	**26.87**	**84.98**	**79.52**

Table 4: Evaluate the generated explanation of fine-tuned models with various metrics. The **top three** scores for each metric are in bold text, and the top one scores are underlined.

Model	A1	A2	A3	A4	Overall
Reference	**3.617**	**2.994**	**3.628**	**3.822**	**3.515**
Codex-Py2Doc	3.394	**2.950**	3.378	3.556	3.319
Codex-Py2NL	2.489	2.528	2.922	2.683	2.656
GPT-2-base-(r+r)	2.972	2.883	3.406	3.539	3.200
GPT-Neo27-(r+r)	3.417	2.811	3.444	3.589	3.315
GPT-Neo13-(r+r)	3.283	2.900	3.439	**3.606**	3.307
CodeT5-(raw)	2.594	2.217	2.756	2.889	2.614
CodeT5-(refined)	**3.489**	**3.061**	**3.572**	**3.661**	**3.446**
CodeT5-(r+r)	**3.478**	2.933	**3.517**	3.578	**3.376**

Table 5: Aspect-wise and overall score of Human evaluations. Four aspects (Section 5): **A1**. General adequacy, **A2.** Coverage, **A3.** Coherence, **A4.** Fluency.

Metric	A1	A2	A3	A4	Overall
ROUGE-1f	0.389	0.293	0.275	0.463	0.311
ROUGE-Lf	0.377	0.292	0.275	0.462	0.304
BLEU	0.416	0.347	0.327	**0.554**	0.355
METEOR	**0.417**	0.368	**0.333**	0.518	**0.361**
BERTScore	0.278	0.218	0.251	0.471	0.263
CodeBERTScore	0.334	0.266	0.280	0.494	0.297
CER	0.346	**0.377**	0.328	0.454	0.339

Table 6: Kendall's τ calculated between 7 automatic metrics and the human evaluated scores.

CER has the highest τ. Interestingly, ROUGE scores show less alignment to human evaluations, although they have been widely used in code summarization. In summary, we recommend applying BLEU and METEOR to the task of code explanation generation.

6.3 Data quality matters

Reviewing both automatic and human evaluations, we find fine-tuning solely on the high-quality partition (CodeExp(refined)) significantly improves the performance compared to fine-tuning on CodeExp(raw). Taking as example the best performed model series, CodeT5: (1) For human evaluation, the CodeT5-(refined) achieves an overall score of 3.446 (ranking 1st) and improves 31.8% over the score 2.614 of CodeT5-(raw) (Table 5). (2) For automatic metrics, BLEU and METEOR are the two most faithful metrics recommended in Section 6.2. CodeT5-(refined) improves the BLEU score from 5.39 to 8.02 and METEOR score from 17.51 to 24.38, when compared to CodeT5-(raw). In fact, the preference of -(refined) over -(raw) can be observed for most automatic metrics and model types (Table 4). This pattern is particularly interesting considering that the refined partition is only 1/15 of the total size. One reason is that the majority of CodeExp(raw) are of short length and do not satisfy the requirements of code explanation documents (Section 3.3). Therefore, it brings about observable noise during the optimization. This result demonstrates the importance of data quality for code explanation generation.

6.4 Case Study

Figure 4 shows an example of code and generated docstring. More examples are listed in Appendix A.6. The CodeT5-(r+r) model successfully captures the main logic of "Indent text by a given number of characters". It also describes in detail the types and semantics of input parameters and returns. Interestingly, the model also captures the

condition for the first "ValueError", although a more faithful description should be "ValueError: if the number of characters differs from the number of lines".

Notably, achieving human-written quality does not mean perfect. In this example, the human-written reference missed the description for the ValueErrors. We found the coverage aspect is challenging for both human-written and model-generated docstrings in general. This has also been reflected by the A2 scores in Table 5.

Code:

```
def indent(text: str, indent_chars: Union[(str, List[str])] = ' ',
level: int = 1) -> str:
     text_lines = text.splitlines(keepends=True)
     if isinstance(indent_chars, str):
          indented_lines = [((indent_chars * level) + line)
                            for line in text_lines]
     elif isinstance(indent_chars, list):
          if (len(indent_chars) != len(text_lines)):
               raise ValueError('Must provide indentation chars for
                                 each line')
          indented_lines = [((ind * level) + line)
                            for (ind, line) in zip(indent_chars,
                            text_lines)]
     else:
          raise ValueError('Unrecognized indentation characters
                           value')
     return ''.join(indented_lines)
```

CodeT5-(r+r):

```
Indent text by a given number of characters.

Args:
    text (str): Text to indent.
    indent_chars (str or list[str], optional): Indentation
characters. Defaults to "".
    level (int, optional): Indent level. Defaults to 1.

Returns:
    str: Indented text.

Raises:
    ValueError: If the specified character is not a valid number of
characters.
```

Reference:

```
Indent lines of text in the string ``text`` using the indentation
character(s) given in ``indent_chars`` ``level`` times.

:param text: A string containing the lines of text to be indented.
:param level: The number of times to indent lines in ``text``.
:param indent_chars: The characters to use for indentation.  If a
string uses repetitions of that string for indentation.  If a list
of strings, uses repetitions of each string to indent each line.

:return: The indented text.
```

Figure 4: **Case study of generated docstring.** The CodeT5-(r+r) correctly captured some detailed code semantics (highlighted in cyan). An ambiguous span is highlighted in pink as well.

7 Related works

Several approaches have been proposed for method/function-level automatic code documentation. Early studies use template-based and information retrieval approaches to generate long-form documents (Wong et al., 2013; McBurney and McMillan, 2014; Moreno et al., 2013). Recent efforts have been mainly focused on the summarization. Barone and Sennrich (2017) collected a large parallel Python code and docstring corpus and trained LSTM-based machine translation model. DeepCom (Hu et al., 2018) introduced structure information of AST to help generate summaries for Java methods. Zhou et al. (2019) proposed ContectCC, which encodes the context information of external dependencies using the API calls in the source Java method. More recently, Ahmad et al. (2020) proposed a transformer approach for method level summarization. Zhang et al. (2020) built a retrieval-based approach using similar code snippets in the generation. The aforementioned studies employed several popular PL-NL corpora (Barone and Sennrich, 2017; Hu et al., 2018; Husain et al., 2019). The average length of these corpora is below 20 tokens and they target the summarization task where the docstrings are often one-liners. Apart from these efforts, few deep learning based approaches generate full-length of documentation. Clement et al. (2020) pretrained T5 models to generate python docstring in numerous styles. The OpenAI Codex (Chen et al., 2021b) trained GPT-3 for both code and document generation in six programming languages.

In these mentioned studies, machine translation (MT) metrics have been widely used for code comment assessments. Gros et al. (2020) questioned this adoption of reference-based MT metrics by examining and showing the semantic difference between code and NL. Roy et al. (2021) rigorously examined the consistency between automatic metrics and human assessments using Kendall's τ as well and recommended the usage of BLEU, METEOR, and chrF.

8 Conclusion

The code explanation generation is an important task for code understanding. On one hand, we show existing summarization methods do not directly apply to this task. On the other hand, we built data collection pipelines, explored consistency between automatic and human evaluations, and provided a framework for fine-tuning existing pre-trained models to generate explanatory docstrings of human-comparable quality. We highlight the importance of data quality by showing that fine-tuning on high-quality data exceeds the performance using raw data of 15 times larger scale. We expect the proposed infrastructures, including the annotated dataset, human-evaluation protocol, recommended metrics, and fine-tuning strategy, to boost future research for code explanation.

9 Limitations

The examined automatic metrics provide insufficient semantic verification for the generated docstrings. Also, the absolute τ values of automatic metrics are all below 0.5, which indicates limited consistency with respect to human evaluations. We look forward to potential factuality-based metrics better modeling the correctness and coverage of the explained semantics. Apart from evaluating stand-alone generations, a user study in the production/developer environment could more accurately reflect the effectiveness of AI-generated explanatory documents. As for the model performance, increasing the coverage over detailed code semantics remains a challenge for tested models. In fact, both generated and human-written docstrings have low coverage scores in the human assessments. Lastly, we tested various fine-tuning strategies in this work, while the large-scale pretraining for code explanation is as well worth exploring.

References

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

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

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

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.

Jie-Cherng Chen and Sun-Jen Huang. 2009. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6):981–992.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri 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, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. Evaluating large language models trained on code. Codex.

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

Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: multi-mode translation of natural language and python code with transformers. *arXiv preprint arXiv:2010.03150*.

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

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Golara Garousi, Vahid Garousi-Yusifoğlu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, and Brian Smith. 2015. Usage and usefulness of technical software documentation: An industrial case study. *Information and Software Technology*, 57:664–682.

Yvette Graham, Timothy Baldwin, and Nitika Mathur. 2015. Accurate evaluation of segment-level machine translation metrics. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1183–1191.

David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to comment "translation": Data, metrics, baselining & evaluation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 746–757. IEEE.

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

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM*

26th International Conference on Program Comprehension (ICPC), pages 200–20010. IEEE.

Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. 2022. Practitioners' expectations on automated code comment generation. In *2022 IEEE/ACM 44rd International Conference on Software Engineering (ICSE)*. IEEE.

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

Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology*.

Chin-Yew Lin and Franz Josef Och. 2004. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pages 605–612.

Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 279–290.

Laura Moreno, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Jsummarizer: An automatic generator of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 230–232. IEEE.

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.

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

Ehud Reiter. 2018. A structured review of the validity of bleu. *Computational Linguistics*, 44(3):393–401.

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1105–1116.

Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. 2019. A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access*, 7:111411–111428.

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

Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 53–64. IEEE.

Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 562–567. IEEE.

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

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. 2019. Augmenting java method comments generation with context information based on neural networks. *Journal of Systems and Software*, 156:328–340.

Yuxiang Zhu and Minxue Pan. 2019. Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352*.

A Appendices

A.1 Statistics of CodeExp annotation results

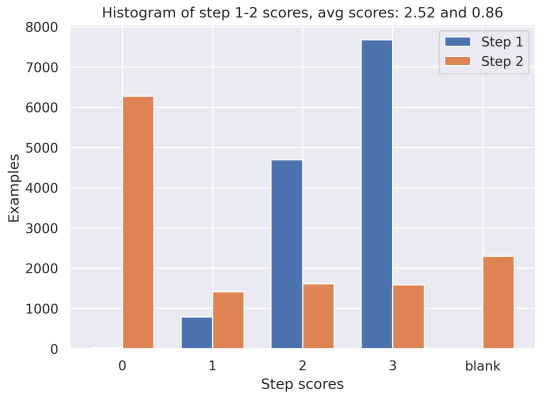

Figure A5: Histogram of Step 1. General adequacy and Step 2. Coverage. Blank score of step 2 indicates there is no branching if/else conditions in the code example.

A.2 Test set configuration

The test set mainly consists of examples in the CodeExp(annotated) partition. We select the group of data of high quality, i.e. with scores of all steps ≥ 1 (including blank scores), and remove duplicates that also appeared in the CodeExp(raw) and CodeExp(refined). This process generates 1744 examples. We also applied the same procedure of quality filtering and deduplication on a small held-out set of GitHub code-doc pairs. Altogether the test set contains 2677 examples.

In detail, the deduplication works by computing the Levenshtein distance between a candidate string of code/document and each code/document in the 2.3 million CodeExp(raw). To accelerate, We compare the first 300 characters of the candidate and target strings. If any computed distance is less than 5% of the total string length, the candidate code-doc example will be considered a duplicate and excluded from the test set.

A.3 Inference settings

At inference time, the generated tokens are sampled with temperature set to 0.1 for all models. The max generated token is set according to each model's capacity. Specifically, 512 tokens for CodeT5 and GPT-Neo models, 256 tokens for GPT-2-base.

A.4 Human evaluation settings

The human evaluation consists of four aspects. For each aspect, a question related to the docstring quality is asked, and the annotator is expected to give an integer score within 0-4, where 0 stands for "not satisfying the question at all" and 4 stands for "perfectly satisfying the question". The four aspects and questions are as follows:

1. General adequacy: Is it possible to gain a basic understanding of what the code does after reading the docstring?

2. Coverage: How much does the docstring cover important semantic details, including descriptions for input parameters, returns, exceptions and if/else, try/catch blocks?

3. Coherence: Is the information provided in the docstring correct and related to the code?

4. Fluency: Is the docstring grammatically correct and easy to read?

Notably, the first three aspects reflect those aspects of data annotations in Section 2.

Given one source code, each annotator must evaluate the generated docstrings of all models (including the reference docstring) to remove inter-model bias. The source of the docstring is hidden to the annotators. In total, ten annotators provided 6480 scores, 180 examples $\times$ 9 models $\times$ 4 aspects. All annotators have python developing experience of over 2 years.

A.5 Codex API settings

The Codex-Py2Doc stands for the Codex API example of "Write a Python docstring". The official prompt includes the `# Python 3.7` header, the source code, and appends at the end the prompting sentence `# An elaborate, high quality docstring for the above function: """`. The model stops generating when the stop token `#` or `"""` is generated. Similarly, the Codex-Py2NL denotes the Codex API of "Python to natural language". The prompt includes the `# Python 3` header, the source code, and the prompting line `# Explanation of what the code does \n\n #`.

We follow all official settings for these APIs, including setting the temperature to 0, top p to 1.0, frequency penalty to 0.0, and presence penalty to 0.0. We use the most capable engine available, "code-davinci-002" and increase the max generated tokens to 256. The default stop token `#` for Codex-Py2NL is removed in our settings because otherwise the API would only generate one line of text.

Model	max input tokens	optimizer	learning rate	batch size	#epochs
GPT-2-base-(raw)	1024	AdamW	start at 5e-5, linear decay to 0	16	3
GPT-2-base-(refined)	1024			16	3
GPT-Neo13-(raw)	2048			16	3
GPT-Neo13-(refined)	2048			16	3
GPT-Neo27-(raw)	2048			16	3
GPT-Neo27-(refined)	2048			16	3
CodeT5-(raw)	1024			32	10
CodeT5-(refined)	1024			32	10

Table 7: Fine-tuning hyperparameters. Note: (1)All max tokens are set to the upper limits of the model_max_token of the pretrained model from Huggingface.co (2)The two-stage fine-tuning for each model adopts the settings of -(raw) and -(refined) at each stage, respectively.

A.6 Examples of generated docstrings

See Figure A6.

Code:

```
def download_file_maybe_extract(url, directory, filename=None,
                                extension=None, check_files=[]):
    if (filename is None):
        filename = _get_filename_from_url(url)
    directory = str(directory)
    filepath = os.path.join(directory, filename)
    check_files = [os.path.join(directory, str(f)) for f in
    check_files]
    if ((len(check_files) > 0) and _check_download(*check_files)):
        return filepath
    if (not os.path.isdir(directory)):
        os.makedirs(directory)
    logger.info('Downloading {}'.format(filename))
    if ('drive.google.com' in url):
        _download_file_from_drive(filepath, url)
    else:
        with tqdm(unit='B', unit_scale=True, miniters=1,
                  desc=filename) as t:
            urllib.request.urlretrieve(url, filename=filepath,
                                       reporthook=_reporthook(t))
    _maybe_extract(compressed_filename=filepath,
                   directory=directory, extension=extension)
    if (not _check_download(*check_files)):
        raise ValueError('[DOWNLOAD FAILED] `*check_files` not
                         found')
    return filepath
```

Reference:

```
Download the file at ``url`` to ``directory``. Extract to
``directory`` if tar or zip.

Args:
    url (str or Path): Url of file.
    directory (str): Directory to download to.
    filename (str, optional): Name of the file to download;
Otherwise, a filename is extracted from the url.
    extension (str, optional): Extension of the file; Otherwise,
attempts to extract extension from the filename.
    check_files (list of str or Path): Check if these files exist,
ensuring the download succeeded. If these files exist before the
download, the download is skipped.

Returns:
    (str): Filename of download file.

Raises:
    ValueError: Error if one of the ``check_files`` are not found
following the download.
```

CodeT5-(r+r):

```
Download a file from a url, unless it is already present in
directory.

Args:
    url (str): URL to download file from.
    directory (str): Directory to extract the file to.
    filename (str, optional): Name of the file to download.
        If None, use filename from URL.
    extension (str, optional): Extension of the file.
    check_files (list, optional): List of files to check.
        If None, will download all files.

Returns:
    str: Path to the downloaded file.

Raises:
    ValueError: If the download fails.
```

CodeT5-(refined):

```
Download a file from url and extract it to directory.

Args:
    url (str): URL to download file from
    directory (str): Directory to save downloaded file to
    filename (str, optional): Name of the file to be downloaded. If
None, use the filename from url.
    extension (str, optional): Extension of the file to be extracted.
If None, use the extension of the file.
    check_files (list of str, optional): Files to be checked.
        If None, no files are checked.

Returns:
    str: Path to downloaded file.

Raises:
    ValueError: If the file is not found.
```

GPT-Neo27-(r+r):

```
Download a file from the given url to the given directory.

Args:
    url (str): URL to download
    directory (str): directory to download file
    filename (str, optional): name of the file to be downloaded
    extension (str, optional): file extension to be downloaded
    check_files (list, optional): list of file paths to be downloaded

Returns:
    str: path to the downloaded file
```

Codex-Py2Doc:

```
Downloads a file from a URL if it not already in the cache.

By default the file at the url `url` is downloaded to the cache_dir
`~/.keras`, placed in the cache_dir `~/.keras`, and given the
filename `filename`. The final location of a file `example.txt` would
therefore be `~/.keras/example.txt`.

Files in tar, tar.gz, tar.bz, and zip formats can also be extracted.
Passing a hash will verify the file after download. The command line
programs `shasum` and `sha256sum` can compute the hash.
```

Codex-Py2NL:

```
1. Download the file from the url
# 2. Check if the file is already downloaded
# 3. If not, download the file
# 4. Check if the file is already extracted
# 5. If not, extract the file
# 6. Check if the file is already downloaded and extracted
# 7. If not, raise an error
# 8. Return the filepath
```

Figure A6: An example function, human-written reference docstring, and selected generated docstrings. The best fine-tuned models captures both global and local function behaviors. Highlighted text: pink - notable errors in generation or reference. cyan - key information captured in generation.