DocChecker: Bootstrapping Code Large Language Model for Detecting and Resolving Code-Comment Inconsistencies

Anh T. V. Dau
FPT Software AI Center
Vietnam
anhdtv7@fpt.com

Jin L.C. Guo
McGill University
Canada
jguo@cs.mcgill.ca

Nghi D. Q. Bui
Fulbright University
Vietnam
nghi.bui@fulbright.edu.vn

Abstract

Comments in source code are crucial for developers to understand the purpose of the code and to use it correctly. However, keeping comments aligned with the evolving codebase poses a significant challenge. With increasing interest in automated solutions to identify and rectify discrepancies between code and its associated comments, most existing methods rely heavily on heuristic rules. This paper introduces **DocChecker**, a language model-based framework adept at detecting inconsistencies between code and comments and capable of generating synthetic comments. This functionality allows **DocChecker** to identify and rectify cases where comments do not accurately represent the code they describe. The efficacy of **DocChecker** is demonstrated using the Just-In-Time and CodeXGlue datasets in various scenarios. Notably, **DocChecker** sets a new benchmark in the Inconsistency Code-Comment Detection (ICCD) task, achieving 72.3% accuracy, and scoring 33.64 in BLEU-4 on the code summarization task. These results surpass other Large Language Models (LLMs), including GPT 3.5 and CodeLlama.

DocChecker is available for use and evaluation. It is available on https://github.com/FSoft-AI4Code/DocCheckerGitHub and as an Online Tool. A demonstration video of its functionality can be found on YouTube.

1 Introduction

Code summarization is a significant issue in software engineering due to its ability to produce explanatory comments for source code, which is essential for ensuring software quality. Identifying and resolving discrepancies between the source code and its corresponding comments is an essential obstacle. Inconsistencies resulting from code changes not being accurately reflected in comments or from initially imprecise descriptions can cause substantial problems in comprehending and managing software. An illustrative example of this inconsistency, sourced from the CodeSearchNet dataset, is depicted in Figure 1 (Husain et al., 2019). These disparities can cause software defects, degrade software quality, and lower developer productivity, as highlighted in recent studies (Wen et al., 2019; Tan et al., 2012; Panthaplackel et al., 2021; Steiner and Zhang, 2022). Moreover, the prevalent issue of code-comment conflicts in widely used datasets impacts the efficacy of code language models trained on them (Sun et al., 2022; Shi et al., 2022; Manh et al., 2023). Recent large models trained specifically for code understanding and generation tasks may be able to address these challenges (Wang et al., 2021; Nijkamp et al., 2022; Wang et al., 2023; Di Grazia and Pradel, 2023). However, these models' efficacy depends on the quality of the training data, emphasizing the importance of accurate and consistent code-comment pairs.

Code Function

```
func (s *storageZfs) ContainerMount(c container)
                    (bool, error) {
      return s.doContainerMount(c.Project(),
             c.Name(), c.IsPrivileged())
}
```

Comment

```
Things we don't need to care about
```

Figure 1: An example of code-comment inconsistency from the CodeSearchNet dataset.

To address these issues, we introduce DocChecker, a framework designed specifically for detecting inconsistencies between code and comments (**ICCD**)). Leveraging the capabilities of AI4SE and insights gained from advancements in code LLMs, DocChecker addresses the critical need for high-quality, consistent documentation in software development. The key idea is to leverage an encoder-decoder backbone network and then

Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics System Demonstrations, pages 187–194
March 17-22, 2024 ©2024 Association for Computational Linguistics

pre-train code-text pairs. This pre-training process employs a multi-faceted approach, including contrastive learning to *bootstrap* code and text features, binary classification to discern consistent from inconsistent pairs, and text generation to create coherent comments. The backbone of this system is UniXcoder (Guo et al., 2022), chosen for its effectiveness and efficiency in handling multi-modal content. DocChecker is specifically designed not only to detect but also to resolve inconsistencies between code and comments by generating replacement comments that accurately reflect the current state of codebase. Furthermore, compared to state-of-the-art CodeLLMs, our method excels significantly on ICCD and code summarization tasks. DocChecker outperforms StarCoder by 30% and surpasses GPT-3.5 and CodeLlama by 10% in terms of accuracy, even though such models are pre-trained on larger-scale datasets. In summary, the key contributions of DocChecker are:

- We propose DocChecker, a framework built on a code language model, jointly pre-trained with three objectives: contrastive learning between code and text, binary classification, and comment generation.
- The experiments show that DocChecker achieves state-of-the-art results on ICCD and code summarization, compared to existing methods and LLMs such as StarCoder, GPT-3.5, and CodeLlama.
- DockChecker is released as an easy-to-use package that can be deployed and installed on a local machine, facilitating its adoption in real-world software development scenarios.

2 Related Work

2.1 Pre-trained Code Language Models

Large language models have demonstrated remarkable success in code understanding and generation, giving rise to code models such as several notable ones, each specializing in different aspects of code processing. Encoder-decoder models like UniXCoder (Guo et al., 2022), CodeT5 (Wang et al., 2021), CodeT5+(Wang et al., 2023) excel in both understanding and generating code. Encoder-only models, such as CuBERT (Kanade et al., 2020) and CodeBERT (Feng et al., 2020), are adept at code-understanding tasks, with CuBERT focusing on Python and CodeBERT extending to six languages. Meanwhile, decoder-only models such as CodeLlama (Roziere et al., 2023), StarCoder (Li et al., 2023), and Magicoder (Wei et al., 2023), CodeGen(Nijkamp et al., 2022, 2023), and DeepSeek-Coder (Guo et al., 2024) specialize in code generation. Other models are trained based on additional structural features of source code, such as InferCoder (Bui et al., 2021a) and Corder (Bui et al., 2021b). These models are typically trained on large-scale datasets from Github (Kocetkov et al., 2022; Lu et al., 2021), with heuristic rules (Manh et al., 2023) used to select only high-quality parts for training

2.2 Detect Inconsistency Between Code and Comment

Source code comments are important in understanding the meaning of the code function. The significance of comments aligning with source code is divided into two categories: inconsistent code-comment detection and comment updates. Rabbi and Siddik (2020) measures the similarity between code functions and comments, identifying inconsistency when the score falls below a set threshold. Panthaplackel et al. (2021) develops a deep learning-based approach to comprehend and establish relationships between comments and code changes. Instead of using machine learning approaches, others propose rule-based methods for analysis. Ratol and Robillard (2017) introduces Fraco, an Eclipse plugin for fragile comment detection during identifier renaming, while Shi et al. (2022) develops an automated code-comment cleaning tool for accurate noise detection in the CodeSearchNet dataset Husain et al. (2019). Although rule-based methods are clear and straightforward, they struggle with new datasets and lack semantic understanding. Recent research explores automatic comment updating, with tools like CUP (Liu et al., 2021) and HebCUP (Lin et al., 2021) effective for simple changes (a single token change) but not for complex ones. In contrast, our framework excels at detecting and updating inconsistent code-comment pairs.

3 Overview of DocChecker

In this section, we describe DocChecker as a Python package and demonstrate its user interface. For full customization and detailed documentation of DocChecker, users can reference our GitHub repository.

```
from DocChecker.utils import inference
✓ 0.3s

code = """
    def split_ext(path, basename=True):
        # This method is called every time we finished the task.
        if basename:
            path = os.path.basename(path)
        return os.path.splitext(path)
    """
inference(raw_code=code,language='python',output_file_path='./out.json')
✓ 3.6s
```

Figure 2: Screenshot for the Input Example.

```
{} out.json > ...
[
    {
        "function_name": "split_ext",
        "code": "def split_ext(path, basename=True):\n        \n        if basena
        "docstring": " This method is called every time we finished the task.",
        "predict": "Inconsistent!",
        "recommended_docstring": "Splits a file extension into its components ."
    }
]
```

Figure 3: Screenshot for the Output file Example.

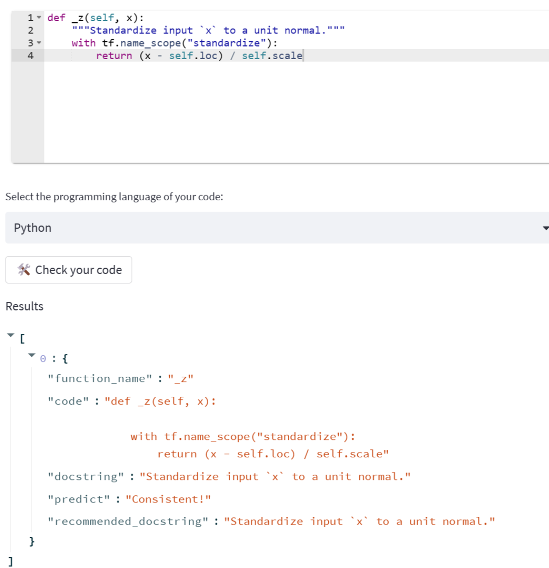

Figure 4: Screenshot for the user interface.

3.1 Python Package

We bundle DocChecker into an easy-to-use library that can be installed via Pypi.

Input: User must provide their source code file as well as the corresponding programming language. DocChecker is able to extract all code functions and their metadata (e.g. function name) by using the AST parser [1]. An example of how to use DocChecker is illustrated in Figure 2.

Output: DocChecker returns in the form of a list of dictionaries corresponding in number to input code functions, including the name of each function in raw code, code snippet, associated docstring, as well as its prediction, and the recommended docstring. If a code-text pair is considered as *"Inconsistent!"*, DocChecker will generate a complete docstring to replace the old ones; otherwise, it will keep the original version. Figure 3 is a screenshot that shows the result of DocChecker's prediction.

3.2 User Interface

We show a demo interface of DocChecker as depicted in Figure 4. It consists of a coding field for directly entering source code or uploading existing code files, a select widget specifying the programming language used for their code, and a button that triggers the query process. When the front-end receives the query result, it displays the previously mentioned list of dictionaries.

[1] We use tree-sitter as the parser `https://github.com/tree-sitter/tree-sitter`.

4 Building Blocks of DocChecker

This section outlines the architecture of DocChecker (see Section 4.1), the objectives guiding its pre-training (Section 4.2), and the specific setup used during pre-training (Section 4.3). Initially, the model undergoes pre-training focusing on contrastive learning and code-to-text generation, followed by fine-tuning for the specific Inconsistency Code-Comment Detection (ICCD) task.

4.1 Architecture

DocChecker's design is influenced by the effectiveness of pre-trained models. Instead of building from scratch, it utilizes existing pre-trained encoder-decoder models. For this project, we selected UniXcoder, an encoder-decoder model (Guo et al., 2022), as our backbone network due to its customizable nature and efficient performance with relatively fewer parameters (details in Section 6.3).

4.2 Pre-training Objectives

DocChecker's pre-training involves three primary objectives:

Code-Text Contrastive Learning (CTC): This aims to align the feature spaces of code and text encoders. We enhance model accuracy by emphasizing similarities in positive code-text pairs and differentiating them from negative pairs. Negative samples are generated following the methodology in (Li et al., 2021), focusing on hard negative pairs based on contrastive similarity.

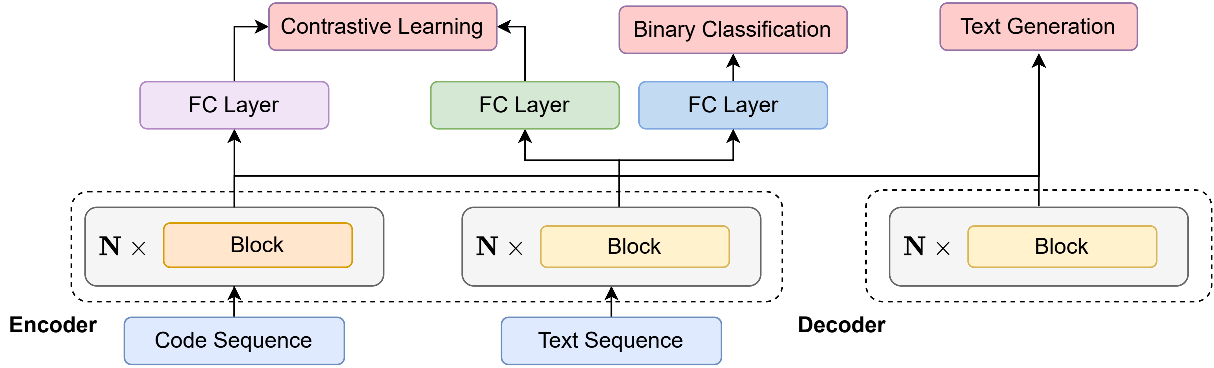

Figure 5: Overview of the DocChecker framework.

Binary Classification (BC): This objective assesses the alignment between code and text. The model distinguishes between consistent (positive) and inconsistent (negative) code-text pairs, enhancing its ability to detect inconsistencies.

Comment Generation (CG): This objective focuses on creating comments that explain a specific code snippet. Training the model to optimize cross-entropy loss in an autoregressive manner improves the model's ability to generate coherent comments.

In addition to these objectives, DocChecker benefits from multi-task learning, sharing the weights between the text encoder and decoder to improve text representation. Separate, fully-connected layers are utilized to capture task-specific differences and minimize task interference.

4.3 Pre-training Setup

DocChecker uses UniXcoder (Guo et al., 2022), which excels at multi-modal contexts and unified cross-modal models. Our goal is to make Dockchecker lightweight and easy to install on a local machine. With 12 hidden layers, 768 hidden sizes, and 3072 intermediate sizes, UniXcoder's architecture of 124M parameters meets our requirements. UniXCoder is also pre-trained on CodeXGLUE, which includes a variety of programming languages. This diverse dataset is critical to ensuring the model's performance in a wide range of software engineering scenarios.

5 Experiment Setup

In this section, we first present the tasks and the datasets used to assess the performance of DocChecker. Then, we describe the baselines and metrics used for evaluation.

5.1 Evaluation Tasks

DocChecker is evaluated for two tasks: ICCD and Code Summarization.

ICCD: For this task, given a comment C with a corresponding code method M, determine whether comment C is semantically out of sync with code function M. To address this challenge, we utilize the post-hoc setting in (Panthaplackel et al., 2021), where code changes that resulted in the mismatch are unknown; Only the current version of the code snippet and old comment are available. This setting is similar to our work, where we want to detect inconsistency for code-text pairs.

Code Summarization: This task aims to generate a natural language summary to explain a given piece of code. By summarizing key concepts and features into a concise format, code summarization addresses the challenge of comprehending programming constructs, especially as codebases continue to grow in complexity.

5.2 Datasets

As we assess the performance of DocChecker across two distinct tasks, we rely on two datasets: the Just-In-Time dataset for the ICCD task and the CodeXGLUE dataset for the code summarization task.

Just-In-Time Dataset ((Panthaplackel et al., 2021)): In this dataset, each sample is the comment-method pair from 2 versions: before and after updating (C_1, M_1) and (C_2, M_2) . In the post-hoc setting, $C = C_1$ and $M = M_2$. They assume that developers updated the comment because it became inconsistent as a result of code changes; they take C_1 to be inconsistent with M_2, consequently leading to a negative example. For positive examples, they additionally examine cases in

which $C_1 = C_2$ and assume that the existing comment has been revised to align consistently with the corresponding code snippet. For a more reliable evaluation, they manually check to get 300 clean examples from the test set and note it as the cleaned test set.

CodeXGLUE dataset (Lu et al. (2021)): This dataset comprises six programming languages: Python, Java, JavaScript, Ruby, Go, and PHP. They come from publicly available open source non-fork GitHub repositories, with each documentation representing the first paragraph.

5.3 Baselines:

Baselines for ICCD task: We select the following existing work to compare against DocChecker for its effectiveness on the ICCD task:

- **SVM (Corazza et al., 2018)**: This bag-of-words approach classifies whether a comment is coherent with the method using an SVM with TF-IDF vectors corresponding to the comment and method;
- **Deep-JIT**: (Panthaplackel et al., 2021) presents a method for detecting inconsistencies between natural language comments and source code. With different ways of encoding, they consider three types and note them as *SEQ, GRAPH, HYBRID*. Deep-JIT is the existing SOTA method on the Just-In-Time dataset.
- **Pretrained Language Models of Code**: We evaluate a range of language models specifically designed for code. Firstly, we focus on three prominent and powerful CodeLLMs: **GPT-3.5-Turbo**, **StarCoder** (15B) (Li et al., 2023), and **CodeLlama** (34B) (Roziere et al., 2023). These models are assessed using both zero-shot (0-shot) and few-shot (3-shot) prompting approaches. In the zero-shot setup, no examples from the Just-In-Time dataset are provided, while the few-shot experiment incorporates three code-text pairs with correct labels from the dataset in each prompt. These prompts are then applied to all selected LLMs. Additionally, we also incorporate a comparative analysis with two established pre-trained models: **CodeBERT** (Feng et al., 2020) and **CodeT5** (Wang et al., 2021).

Baselines for Code Summarization task: In this experiment, we focus on the fine-tuning setting and compare our method with smaller-scale LMs, including RoBERTa (Liu et al., 2019), CodeBERT (Feng et al., 2020) trained with masked language modeling; PLBART (Ahmad et al., 2021) is based on BART and pre-trained using denoising objective; CodeT5 (Wang et al., 2021), adapted from T5, takes into account important token-type information in identifiers; and the variant of UniXcoder (Guo et al., 2022) since we utilize UniXcoder as the backbone network.

Method	Cleaned Test set		Full Test Set	
	F1	Acc	F1	Acc
SVM	53.9	60.7	54.6	60.3
Deep-JIT$_{SEQ}$	63.0	60.3	66.3	62.8
Deep-JIT$_{GRAPH}$	65.0	62.2	67.2	64.6
Deep-JIT$_{HYBRID}$	63.3	55.2	66.3	58.9
CodeBERT	67.9	66.9	70.7	69.8
CodeT5	69.5	68.8	70.2	70.1
GPT-3.5 $_{0\text{-shot}}$	60.9	65.1	62.5	64.6
StarCoder $_{0\text{-shot}}$	43.7	43.1	45.2	43.9
CodeLlama $_{0\text{-shot}}$	70.2	68.7	62.6	61.8
GPT-3.5 $_{3\text{-shot}}$	66.4	67.0	66.1	61.4
StarCoder $_{3\text{-shot}}$	44.2	43.6	42.8	42.2
CodeLlama $_{3\text{-shot}}$	70.5	69.2	62.3	62.1
DocChecker	**73.1**	**70.7**	**74.3**	**72.3**

Table 1: Results for post hoc settings on the Just-In-Time dataset

5.4 Evaluation Metrics

Metrics for ICCD: We use two common classification metrics: F1 score (w.r.t. the positive label) and Accuracy (Acc) to report the performance of methods.

Metrics for Code Summarization: For this task, we use the smoothed BLEU-4 (Lin and Och, 2004) as the evaluation metric and report the overall score of six programming languages.

6 Evaluation Results

6.1 Effectiveness of DocChecker on ICCD

Table 1 presents results for all baselines under the post-hoc setting and LLMs. In general, we find that our model can significantly outperform all of the baselines. Despite CodeBERT and CodeT5 being pre-trained models with more parameters and showcasing efficiency in numerous downstream tasks, their performance is behind ours. DocChecker achieves a new SoTA of 72.3% accuracy and 74.3% F1 score on the full test set of Just-In-Time. On the other hand, although previous literature has empirically explored various capabil-

Method	Summarization BLEU-4
RoBERTa	16.57
CodeBERT	17.83
PLBART	18.32
CodeT5-small	19.14
CodeT5-base	19.55
UniXcoder	19.30
-w/o contras	19.20
-w/o cross-gen	19.27
-w/o comment	18.97
-w/o AST	19.33
-using BFS	19.24
-using DFS	19.25
DocChecker	**33.64**

Table 2: Results on the code summarization task.

ities of LLMs in diverse natural language processing and code generation tasks, billion-parameter LLMs such as StarCoder, GPT 3.5, and CodeLlama still struggle with ICCD, even with the construction of various types of prompts. In particular, DocChecker produces significant improvements of +10% accuracy and F1 score compared to the selected LLMs.

The experiment results suggest that DocChecker benefits from using a pre-trained language model with our novel pre-training objectives. It supports that our method effectively detects inconsistent samples in the code corpus.

6.2 The effectiveness of DocChecker on Code Summarization

Our results on this task are shown in Table 2. DocChecker is compared to a number of pre-trained code language models during our evaluation. Following DocChecker's pre-training for three aforementioned objectives, our method outperforms others significantly. DocChecker's BLEU-4 score is twice that of RoBERTa and CodeBERT. Furthermore, despite the fact that CodeT5-base uses a 12-layer encoder and a 12-layer decoder, which are twice as powerful as our architecture, its performance is significantly lower. DocChecker outperforms CodeT5 and the backbone network UniXcoder by +13 BLEU-4 scores.

6.3 Influence of the backbone network on DocChecker

DocChecker functions as a framework, so choosing an encoder-decoder model for the backbone network is flexible. This section demonstrates the effect of several pre-trained models on DocChecker's effectiveness. We use CodeBERT, CodeT5, and

Backbone Network	Cleaned test set F1	Cleaned test set Acc	Full test set F1	Full test set Acc
CodeBERT	68.2	67.1	71.5	70.4
CodeT5	70.1	69.5	71.9	71.5
UniXcoder	73.1	70.7	74.3	72.3

Table 3: Results of DocChecker pre-trained with different backbone networks on the Just-In-Time dataset.

Code Function

```
public <R> sendAndReceive(final Function
    <HttpResponse, R> responseHandler){
  return responseHandler.apply(send());
}
```

Original Comment

```
Syntax sugar.
```

Recommended comment from DocChecker

```
Send and Receive the response.
```

Code Function

```
Path renameToFinalName(FileSystem fs, Path tempPath)
    throws IOException, StageException {
    return fsHelper.renameAndGetPath(fs, tempPath);
}
```

Original Comment

```
This method should be called every time we finish writing
into a file and consider it done .
```

Recommended comment from DocChecker

```
Renames the given path to the final name .
```

Figure 6: Some inconsistent code-comment examples were collected from the CodeXGlue dataset and our recommended comment to replace.

UniXcoder as the backbone network for DocChecker. Each chosen backbone is pre-trained in the DocChecker framework and fine-tuned using the Just-In-Time dataset. The results in Table 3 show that the pre-trained models perform better after re-pre-training compared to their original versions. However, UniXcoder emerges as the most effective backbone model for this task, so we use it for all of our experiments.

6.4 Practical Application

Aside from demonstrating DocChecker's performance, we highlight its effectiveness in real-world scenarios. We consider the popular CodeXGlue dataset, which extracts functions and paired comments from Github repositories. Although this benchmark dataset is expected to be of high quality, noise is unavoidable due to variations in coding conventions and assumptions used in modern programming languages and IDEs. Using Doc-

Checker, we can filter the dataset's inconsistent code-comment samples and create new comprehensive summary sentences for them.

Figure 6 shows an example of an inconsistent sample identified by DocChecker in the CodeSearchNet dataset. The comment associated with the code snippet is misaligned and needs to be updated. Beyond detection, our method generates a detailed summary sentence for each sample, which replaces the outdated ones.

7 Conclusion

In this paper, we present DocChecker, a framework to filter and generate replacement comments for inconsistent code-comment pairs. The experimental results demonstrate the effectiveness of this method compared to SoTA existing methods and LLMs, showcasing its applicability in both academic and practical contexts. We have released DocChecker as an easy-to-use library, complemented by a user-friendly interface to enhance user interaction.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021a. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1186–1197. IEEE.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021b. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 511–521.

Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. 2018. Coherence of comments and method implementations: a dataset and an empirical investigation. *Software Quality Journal*, 26:751–777.

Luca Di Grazia and Michael Pradel. 2023. Code search: A survey of techniques for finding code. *ACM Comput. Surv.*, 55(11).

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

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. pages 7212–7225.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

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

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Junnan Li, Ramprasaath Selvaraju, Akhilesh Gotmare, Shafiq Joty, Caiming Xiong, and Steven Chu Hong Hoi. 2021. Align before fuse: Vision and language representation learning with momentum distillation. *Advances in neural information processing systems*, 34:9694–9705.

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

Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2021. Automated comment update: How far are we? In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 36–46.

Chin-Yew Lin and Franz Josef Och. 2004. Orange: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507.

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. *CoRR*, abs/1907.11692.

Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. 2021. Just-in-time obsolete comment detection and update. *IEEE Transactions on Software Engineering*, pages 1–1.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tu-

fano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Dung Nguyen Manh, Nam Le Hai, Anh T. V. Dau, Anh Minh Nguyen, Khanh Nghiem, Jin Guo, and Nghi D. Q. Bui. 2023. The vault: A comprehensive multilingual dataset for advancing code understanding and generation.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.

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*.

Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. 2021. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 427–435.

Fazle Rabbi and Md Saeed Siddik. 2020. Detecting code comment inconsistency using siamese recurrent network. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 371–375. Association for Computing Machinery.

Inderjot Kaur Ratol and Martin P Robillard. 2017. Detecting fragile comments. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 112–122. IEEE.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 107–119.

Theo Steiner and Rui Zhang. 2022. Code comment inconsistency detection with bert and longformer. *arXiv preprint arXiv:2207.14444*.

Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1609–1620.

Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

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.