CodeFuse-13B: A Pretrained Multi-lingual Code Large Language Model

Peng Di[†], Jianguo Li[†], Hang Yu[†], Wei Jiang[†], Wenting Cai, Yang Cao, Chaoyu Chen, Dajun Chen, Hongwei Chen, Liang Chen, Gang Fan, Jie Gong, Zi Gong, Wen Hu, Tingting Guo, Zhichao Lei, Ting Li, Zheng Li, Ming Liang, Cong Liao, Bingchang Liu, Jiachen Liu, Zhiwei Liu, Shaojun Lu, Min Shen, Guangpei Wang, Huan Wang, Zhi Wang, Zhaogui Xu, Jiawei Yang, Qing Ye, Gehao Zhang, Yu Zhang, Zelin Zhao, Xunjin Zheng, Hailian Zhou, Lifu Zhu, Xianying Zhu[*]
Ant Group, China
[†]Corresponding-authors: {dipeng.dp,lijg.zero,hyu.hugo,jonny.jw}@antgroup.com

ABSTRACT

Code Large Language Models (Code LLMs) have gained significant attention in the industry due to their wide applications in the full lifecycle of software engineering. However, the effectiveness of existing models in understanding non-English inputs for multi-lingual code-related tasks is still far from well studied. This paper introduces CodeFuse-13B, an open-sourced pre-trained code LLM [2]. It is specifically designed for code-related tasks with both English and Chinese prompts and supports over 40 programming languages. CodeFuse achieves its effectiveness by utilizing a high-quality pre-training dataset that is carefully filtered by program analyzers and optimized during the training process. Extensive experiments are conducted using real-world usage scenarios, the industry-standard benchmark HumanEval-x, and the specially designed CodefuseEval for Chinese prompts. To assess the effectiveness of CodeFuse, we actively collected valuable human feedback from the AntGroup's software development process where CodeFuse has been successfully deployed. The results demonstrate that CodeFuse-13B achieves a HumanEval pass@1 score of 37.10%, positioning it as one of the top multi-lingual code LLMs with similar parameter sizes. In practical scenarios, such as code generation, code translation, code comments, and testcase generation, CodeFuse performs better than other models when confronted with Chinese prompts.

KEYWORDS

code large language models, multi-lingual, Chinese prompts

1 INTRODUCTION

Code Large Language Models (Code LLMs) have attracted substantial attention in the industry owing to their vast applications throughout the entire software engineering lifecycle. The release of Copilot, empowered by Codex [7], served as a significant testament to the imminent arrival of the era of intelligent code. One astonishing application, ChatGPT [6, 27], has captivated an incredible user base of over 100 million in two months since its launch. In recent code models such as AlphaCode[21], InCoder[13], SantaCoder[1], StarCoder[20], and Code Llama[30], the incorporation of fill-in-the-middle capabilities has proven to be particularly valuable for practical code completion scenarios.

[*]Non-corresponding authors are listed in alphabetical order.

Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Table 1: CodeFuse project roadmap.

Release date	Model	HumanEval Pass@1
Mar 2023	CodeFuse-1.3B-2K Seq-Length	11.58%
Apr 2023	CodeFuse-6.5B-4K Seq-Length	20.46%
May 2023	CodeFuse-13B-Base-4K Seq-Length	32.93%
Jun 2023	CodeFuse-13B (opened in Sep)	37.10%
Sep 2023	CodeFuse-CodeLlama-34B (opened)	74.40%

While these models have practical applications to software development processes, their effectiveness in comprehending non-English inputs for code-related tasks remains unsatisfactory[51]. To bridge this gap, CodeGeeX [52] attempted to establish a connection between code and non-English languages, incorporating vocabulary tokens from various natural languages. Indeed, by leveraging large, domain-specific datasets, LLMs can significantly enhance their effectiveness in applications that necessitate a combination of natural language understanding and domain-specific knowledge, including specialized terminology.

The paper introduces CodeFuse, a Language Model for coding, which is open-sourced on GitHub [1] and Huggingface [2], an open-source code Language Model (LLM). The CodeFuse project, a collaborative effort within AntGroup, has witnessed monthly model iterations resulting in consistent performance improvements, as depicted in Table 1. As of this September, we open-sourced two versions of CodeFuse, CodeFuse-13B and CodeFuse-CodeLlama-34B. CodeFuse-13B underwent fine-tuning by LoRA/QLoRA on multiple code tasks using the self-pretrained base model, while CodeFuse-CodeLlama-34B was fine-tuned using CodeLlama-34b-Python. Excitingly, *CodeFuse-13B surpasses other code LLMs of similar size, and CodeFuse-CodeLlama-34B outperforms GPT4 and ChatGPT-3.5 on the HumanEval benchmark.*

This paper centers around the pre-trained model CodeFuse-13B, providing a comprehensive overview of the development process and evaluating its performance in real industrial scenarios. The production of CodeFuse-13B encompasses several crucial steps, including:

[1]https://github.com/codefuse-ai
[2]https://huggingface.co/codefuse-ai

- Data collection: We collected about 200+ TB of code-related data, and finally refined it to around 1.6TB (1T Token) of clean data suitable for pre-training.
- Program feature analysis: We extracted a set of program features from the collected code, including syntax correctness, cleanliness score, etc. This analysis serves three purposes: 1) ensuring high-quality code for pre-training data, 2) providing AST/CFG/DFG/IR program feature data and extracting code semantics to facilitate program understanding, and 3) profiling the code dataset to guide constraint-based instruction fine-tuning. The analyzer employed a datalog-based program analysis method, translating analysis into a query system.
- Pre-training: Using the AntGroup's common technology stack, we developed CodeFuse to pre-train a large-scale model with 13 billion parameters in a stable manner.
- Instructional fine-tuning: This stage involved various fine-tuning techniques, such as supervised instruction fine-tuning (SFT), and multi-task instruction fine-tuning (MFT), among others.
- Model evaluation: We provide a comprehensive evaluation kit that supports both online and offline inference evaluation methods. This kit drives model training across different downstream tasks based on performance indicators and visualizes feedback on evaluation results, enabling continuous iterations and optimization of corresponding data, algorithms, and engineering challenges.
- Model operations: In the context of hundreds or even thousands of GPU training instances, numerous challenges emerge, such as timely automatic detection of faulty nodes, automatic initiation of training tasks, and monitoring the convergence of training progress. We have made significant advancements in addressing these challenges through Model Operations.

We conducted an industry-based evaluation of CodeFuse by integrating it into the software development process at AntGroup. Additionally, we developed extensions for several popular Integrated Development Environments (IDEs), namely VSCode, JetBrains, and Ant CloudIDE (a Web IDE). Moreover, we developed and open-sourced a more comprehensive benchmark, named CodefuseEval[3], to support for a broader range of programming scenarios involving Chinese inputs.

The results demonstrate that CodeFuse-13B achieves a HumanEval Pass@1 score of 37.10%, making it one of the top models with similar sizes. In practical multi-lingual scenarios, CodeFuse outperforms other models with Chinese prompts, such as code translation, code comments, testcase generation and others.

We summarize the key contributions as follows:

- We introduce CodeFuse-13B, an open-sourced pre-trained code LLM with 13B parameters and its training procedure. It is specifically designed for code-related tasks with Chinese prompts and supports over 40 programming languages.
- We have developed CodeFuse extensions for various IDEs. These extensions enable developers to seamlessly integrate CodeFuse into their coding workflows, enhancing productivity and code generation capabilities.

[3] https://github.com/codefuse-ai/codefuse-evaluation.

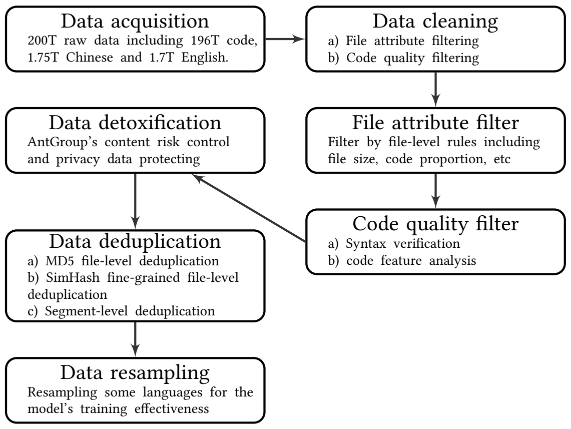

Data acquisition
200T raw data including 196T code, 1.75T Chinese and 1.7T English.
Data cleaning
a) File attribute filtering
b) Code quality filtering
Data detoxification
AntGroup's content risk control and privacy data protecting
File attribute filter
Filter by file-level rules including file size, code proportion, etc
Code quality filter
a) Syntax verification
b) code feature analysis
Data deduplication
a) MD5 file-level deduplication
b) SimHash fine-grained file-level deduplication
c) Segment-level deduplication
Data resampling
Resampling some languages for the model's training effectiveness

Figure 1: The diagram of data processing.

- We evaluate the effectiveness of CodeFuse in various application scenarios, including code generation, code translation, code comments, and testcase generation. The results demonstrate that CodeFuse-13B achieves a HumanEval Pass@1 score of 37.10%, outperforming other multi-lingual models with similar sizes. Moreover, CodeFuse-13B is better than other models in practical scenarios involving Chinese prompts.

2 DATA PREPARATION

2.1 Overview of data processing

Since code LLMs focus more on code-related tasks, there are significant differences compared to the methods used in general LLMs, in terms of data acquisition, data cleaning, detoxification, deduplication, and data resampling, as shown in Figure 1. This paper will primarily focus on how CodeFuse constructs pre-training data for the code domain large model.

Data acquisition. The pre-training data for CodeFuse consists of 196TB of code, 1.75TB of Chinese raw data, and 1.7TB of English raw data, totaling 200TB, that are tokenized into 800 billion tokens of code, 100 billion tokens of Chinese corpus, and 100 billion tokens of English corpus (see Section 3.1). The original code data were a combination of self-crawled GitHub dataset and opensource dataset `Stack` [19]. The combined dataset is deduplicated and filtered to include over 40 programming languages with the distribution shown in Figure 2, in which 13 programming languages account for more than 1% each. Java, Python, C++, and JavaScript surpass 10%, while the remaining nearly 30 programming languages comprise 9% of the total. The Chinese corpus is sourced from `CommonCrawl`, computer-related websites, documentation of programming languages and their third-party libraries, etc. The English corpus is sampled from various categories in `Pile` including `StackExchange`, `Arxiv`, `Wikipedia`, `OpenWebText2`, `Pile-CC`, etc.

Data cleaning. The cleaning strategy for code data is divided into two levels. The first-level filtering strategy involves the aspect of file attributes, including strategies such as discarding large files (e.g., files with more than 10,000 lines or single files larger than 1MB) and discarding abnormal text (e.g., lines with an average length greater than 100 or a proportion of alphanumeric characters less

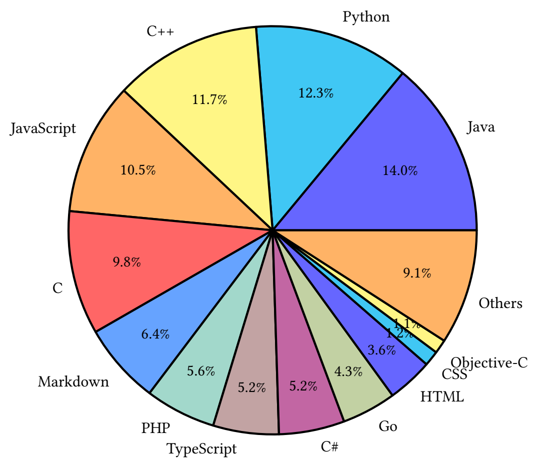


Figure 2: Distribution of programming languages.

than 40%). The second-level filtering strategy relies on AntGroup's program analyses [15, 22, 33, 53, 54] to filter out code data that does not meet the requirements of syntax correctness and code quality. The open-source release of our program feature analyzer is planned, and the details will be introduced in Section 2.2.

Data detoxification. We utilize AntGroup's content risk control and privacy data protecting capabilities to identify and filter out risky data from the training dataset, ensuring data safety.

Data deduplication. Multiple granularities of deduplication are performed to sensuality of training data. First is the global deduplication with file-level `MD5`. The second is fine-grained file-level deduplication with `SimHash` score (e.g. $\geq$0.95). Third is segment-level deduplication based on code analysis to separate codes and comments, and deduplicate code and text segments when document-level `SimHash` score is larger than a threshold (e.g. $\geq$0.90).

Data resampling. Language distribution-based resampling removes data for niche programming languages (with a data proportion below 0.1%) and downsamples HTML, CSS, JSON, and other similar languages that can negatively impact the model's training effectiveness.

Through these data processing stages, the training data for the CodeFuse large model is prepared to ensure high-quality data for training and enable the model to exhibit excellent performance. The filtered and resampled data are tokenized into a format directly usable for the pre-training phase.

2.2 Program feature analysis

To improve the quality of training code data, we propose two approaches to analyze code features: static analysis-based and model-based methods. Static analysis-based method offers cost-effectiveness, interpretability, and iterative refinement but may struggle with quantifying complex code features. The model-based method provides better handling of unquantifiable features but comes with higher costs and weaker interpretability. For massive code data, static analysis is preferred first, yet a hybrid approach combining static analysis with model-based classification can be used for fine-grained cleansing. This approach leverages the strengths of both methodologies to ensure efficient and effective handling of code features.

High-quality code evaluation model. To effectively evaluate the quality of code, we have proposed several metrics integrated into our model. These metrics serve as indicators of code quality, and they are designed to provide comprehensive insights into potential areas of improvement.

- **Correctness measurement:** After performing syntax verification and bug detection, it has been observed that the number of bugs is inversely proportional to the quality of the code. A higher bug frequency indicates lower code quality. To provide a more detailed analysis, bugs have been categorized into three levels: Fatal, Error, and Warning.
- **Readability:** Larger methods, classes, and more method parameters tend to be associated with lower code quality. This is due to the increased complexity and reduced readability that often accompanies large methods, classes, and more parameters.
- **Redundancy:** The presence of redundant classes is a sign of poor code quality. Redundancy in classes often leads to increased complexity and decreased maintainability.
- **Naming style:** Identifier names that are either too long or too short can compromise code quality. Short names may not adequately describe the purpose of a variable or function, while excessively long names can hinder readability.
- **Cyclomatic complexity:** The cyclomatic complexity is a metric used to measure the complexity of the control flow of a module. It quantifies the number of independent paths through the code, which can also be understood as the minimum number of test cases required to cover all possible scenarios.
- **Coupling:** It is a measure of the degree of association between modules. The strength of coupling depends on the complexity of the interfaces between modules, the way modules are invoked, and the amount of data transmitted through interfaces. The coupling between modules refers to the dependency relationship between them, including control relationships, invocation relationships, and data transmission relationships.

Implementation. We utilized AntGroup's static analyzer, named `Sparrow`, to filter the code. Similar to `CodeQL` [25] developed by `GitHub`, `Sparrow` is a datalog-based program analysis tool that translates analysis into a query system. To handle the analysis of a substantial volume of code, we stored the resulting information offline in a database along with a datalog solver engine. This setup allows us to efficiently query and retrieve program analysis results.

2.3 Code semantic extraction

One of the most crucial tasks for code LLMs is to comprehend the semantics of the code. To support that, it is needed to extract codes with natural language annotations, which serve as interpretations of the code's semantics. Therefore, we employ static analysis to extract code snippets and their corresponding comments from high-quality code. This extracted data is then utilized for the Supervised Fine-tune (SFT) of CodeFuse. The goal of this approach is to ensure that the model not only understands the syntax of the programming languages but also gains a deep comprehension of the underlying logic and functionality of the code, thereby enhancing its code understanding capabilities.

Strategy for selecting code-comment pairs. To ensure the quality of code-comment pairs, we focus on extracting functions and their corresponding comments from the code. We utilize rule-based approaches to filter code-comment pairs:

- **Meaningless comments:** Meaningless comments are detected by `Sparrow` using a set of rules, which include comment keyword detection, identification of auto-generated setter/getter methods, and others. These comments are considered irrelevant and are subsequently discarded.
- **Code length limits:** Code containing fewer than 3 lines is discarded to ensure sufficient context in the code.
- **Effective code limits:** Methods with less than 60% of effective code lines are discarded to ensure a significant proportion of meaningful code.
- **Comment length limits:** To maintain readability and manageability, comments longer than 512 characters are discarded.

Implementation. We utilize tree-sitter, a fast and robust parsing tool, to support multiple programming languages and handle large code volumes. By leveraging the tree-sitter's query functionality, we extract function-comment pairs across different languages, enabling a unified data structure. The tool efficiently distributes and aggregates code parsing results, enhancing parsing speed and compatibility with various programming languages.

Application. We have accumulated a dataset of 3.2 billion function-level code-comment pairs. This dataset spans various sources, including GitHub, GeeksforGeeks, StarCoder, and AntGroup's internal code repositories, providing a broad and diverse base for training and evaluation. The dataset is used on the following code-related tasks (see Section 4):

- **Code generation:** This task involves the translation of natural language into code snippets. The large dataset aids in understanding the semantic meaning behind the natural language and generating the corresponding code.
- **Code comment:** This task focuses on the generation of comments for given code. Leveraging the code-comment pairs in our dataset, CodeFuse can effectively generate meaningful and context-specific comments for any given piece of code.
- **Code explanation:** This task deals with providing comprehensive explanations for given code snippets. The vast amount of code-comment pairs within our dataset aids in drawing parallels and generating detailed, understandable explanations of code functionality.

2.4 Code dataset portrait

Code dataset portrait refers to an automated approach that involves annotating, categorizing, and analyzing code in order to capture various dimensions of large-scale codebases. By performing code profiling, a more thorough and detailed understanding of the code data used for training can be obtained. This understanding facilitates more efficient implementation of techniques such as Supervised Self-Training (SST) and Self-Supervised Fine-Tuning (SFT). The applications of code portrait can be categorized as follows:

- **Deep insight of training data:** For each portrait dimension, the portrait analysis can provide measurements like validation loss, perplexity, and even feedback errors. These could show us which portion of data are more difficult to learn.
- **Augmenting training data:** The portrait feedback urges us to augment training data by improving the quantity or quality of the corresponding weak performed dimension. On the one hand, since the distribution of training data could significantly affect a model's behavior [55], code portrait allows us to improve the quantity or portion of weak performed dimension according to evaluation feedback. On the other hand, it may urge us to check the quality of the weak performed portion of data and find potential ways to further improve the data quality. Both ways may optimize the model in subsequent experiments.

The implementation of code portrait is based on mentioned `Sparrow`. Based on the code portrait analysis of the code added to AntGroup's repositories in the past three months, the following key insights have been derived:

- Java is the dominant language, accounting for approximately 40.7% of the newly added code. Within the Java codebase, the majority (over 90%) is attributed to the SOFA (Scalable Open Financial Architecture) projects. Therefore, the primary emphasis for code generation should be on Java applications related to the SOFA stack.
- Certain modules within the Java code, namely 'test', 'model', 'facade', and 'dal', have a higher representation. These sections should be regarded as high-priority areas for code generation, given their significance within the codebase. Focusing on these modules will help ensure that the generated code adequately addresses the needs and requirements of the system.
- JavaScript constitutes 9.23% of the newly added code, making it the second most widely used language. Within the JavaScript codebase, the Bigfish framework stands out as the most predominant. Therefore, generating code specifically for the Bigfish framework should be given priority to cater to the significant usage of this framework within the JavaScript codebase.

3 TRAINING

In this section, we introduce the pre-training procedure of CodeFuse including tokenization, model architecture and training procedure.

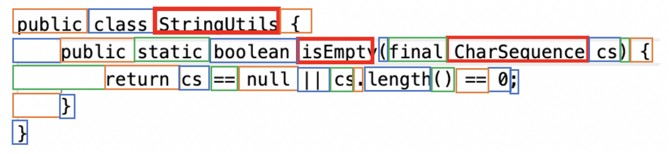
3.1 Tokenization

Our tokenizer is BPE-based [34] and designs to avoid Out of Vocabulary (OOV) problems. It has a vocabulary size of 100,864, covering keywords and common words in programming languages as well as information techniques related corpus. The tokenizer also supports effective segmentation of natural languages like Chinese and English. It ensures accurate code tokenization. Figure 3 provides example cases to demonstrate tokenization by several well-known code LLMs, highlighting the effectiveness of CodeFuse's tokenizer.

To verify the tokenization performance of the CodeFuse tokenizer on code, Chinese, and English, we randomly sampled 100,000 examples from the training dataset. The token count and compression ratio after tokenization are shown in Table 2. The tokenization performance of the CodeFuse tokenizer is significantly better than that of CodeLlama and CodeGen for code, Chinese, and English. This is attributed to its larger vocabulary size and dedicated code-specific vocabulary.

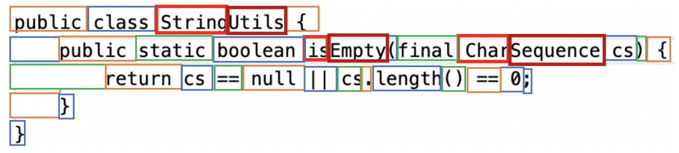
Table 2: Comparison of compression rate (C-Rate) of tokenization. C-Rate = #Tokens / #Characters, the lower the better.

Type	#Characters	CodeFuse		CodeLlama		CodeGen	
		#Tokens	C-Rate	#Tokens	C-Rate	#Tokens	C-Rate
Code	338,758,753	86,787,734	0.25	99,180,237	0.29	96,289,455	0.28
Chinese	85,998,939	98,491,170	1.14	121,180,842	1.41	161,977,211	1.88
English	283,983,202	69,951,060	0.24	78,472,584	0.27	71,393,619	0.25



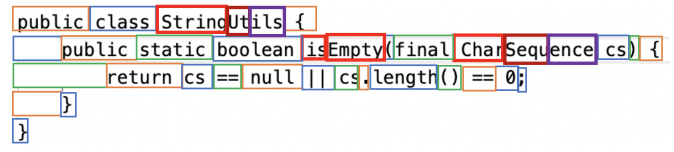
```
public class StringUtils {
    public static boolean isEmpty(final CharSequence cs) {
        return cs == null || cs.length() == 0;
    }
}
```

(a) CodeFuse has 40 tokens



```
public class StringUtils {
    public static boolean isEmpty(final CharSequence cs) {
        return cs == null || cs.length() == 0;
    }
}
```

(b) CodeLlama has 46 tokens



```
public class StringUtils {
    public static boolean isEmpty(final CharSequence cs) {
        return cs == null || cs.length() == 0;
    }
}
```

(c) CodeGen has 47 tokens

Figure 3: Tokenization examples of different models applied to the same code snippet.

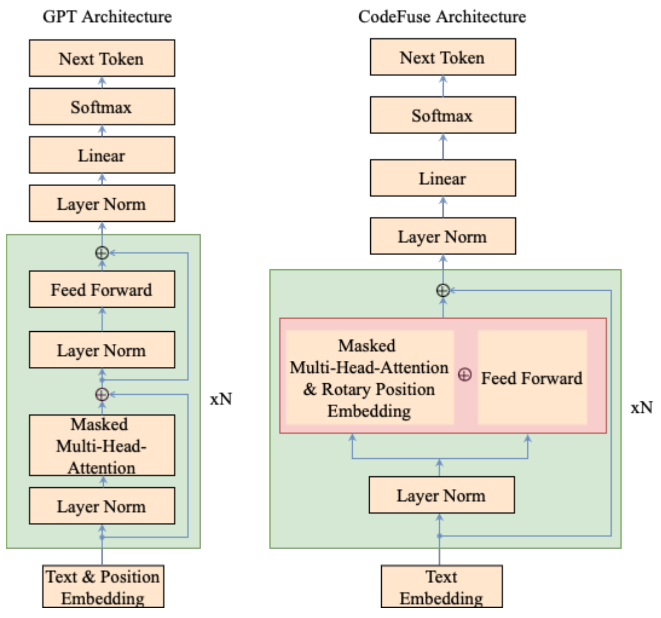


Figure 4: Comparison CodeFuse architecture with GPT.

3.2 Model architecture

The model architecture of CodeFuse is an auto-regressive Transformer, similar to GPT-3 [6], with regular next-token prediction as the learning objective. We made two key changes similar to GPT-J [42]: (1) using Rotary positional embeddings instead of learned positional embeddings, and (2) employing parallel attention and feed-forward layers (FFN) instead of serial layers as in GPT3. The details are shown in Figure 4.

Rotary positional embeddings (RoPE). We adopt RoPE [36] as position embedding in Multi-Head Attention instead of the learned positional embedding. To balance effectiveness and computational efficiency, we apply RoPE only to the first half of embedding vectors [5, 42].

Parallel attention and FFN layers. In GPT-3, Multi-Head Attention is computed first, and then the result is added with the residual connection before being passed into FFN as shown in Equation 1. In our model, Multi-Head Attention and FFN are computed in parallel, and then the results of both are added with the residual and passed to the next layer as shown in Equation 2. This architecture can improve the computing throughput by about 15%.

$$x_{t+1} = x_t + FFN(LN(x_t + Attn(LN(x_t)))) \quad (1)$$

$$x_{t+1} = x_t + Attn(LN(x_t)) + FFN(LN(x_t)) \quad (2)$$

Activations. We use `GeLU` [16] activation function, which can alleviate the issue of gradient vanishing during the training process. Moreover, it introduces a nonlinear transformation similar to the `sigmoid` function, which helps to accelerate the convergence speed of the model.

Layer normalizations. We adopt the pre-layer normalization [4] to the input of the Transformer block which is more robust to input variations and improved gradient flow [48].

3.3 Pre-training

We trained CodeFuse based on the GPT-NeoX [5] framework, which is built on Megatron [35] and DeepSpeed and incorporates deep optimizations for algorithms, communication, and model parallelism. We trained CodeFuse on a 64-node GPU cluster, each with eight NVIDIA A100-SXM-80GB GPUs. We trained CodeFuse in various sizes with 350M, 1.3B, 6B, and 13B parameters, details as shown in Table 3.

Table 3: Family models of CodeFuse.

Model	NumLayers	NumHeads	HiddenSize	SeqLen	BatchSize	LearningRate	Paralells
CodeFuse-350M	24	16	1024	2048	1024	2e-4	DP=64
CodeFuse-1.3B	24	16	2048	2048	1024	2e-4	DP=128
CodeFuse-6B	28	32	4096	4096	2048	1.5e-4	DP=256
CodeFuse-13B	40	40	5120	4096	4096	1.5e-4	DP=256, TP=2

Training optimizations. CodeFuse-13B is trained with 256-way data parallelism, 2-way tensor parallelism, and sequence parallelism, and reducing memory consumption with DeepSpeed ZeRO-1 [29]. The sequence length of CodeFuse-13B is 4096, and accelerate long sequence model training with Flash Attention [10]. The micro batch size is 16, and the global batch size is 4096, we achieved 180 TFLOPS and 56% average utilization rate of tensor cores on 512 GPUs. We use Adam [18] optimizer for training, where the initial learning rate is 1.5e-4, and the min learning rate is 1.5e-5, along with cosine learning rate decay style. We use fp16 mixed precision training mode. To avoid precision underflow or overflow, we set the initial loss scaling to 32768 and the minimum loss scaling to 1.

3.4 Supervised Finetuning

The supervised finetuning (SFT) of CodeFuse containts several dimensions: data collection, instruction augmentation, data formatting, training strategy, and fine-tuning framework. The following sections will elaborate on each dimension.

Data collection for SFT. As a domain-specific LLM for the code field, it needs to have the following functionalities:

- Basic natural language understanding capability: The model should provide a satisfactory understanding of complex natural language questions for text2code tasks.
- Common downstream code tasks: Tasks such as Text2Code (natural language to code snippet), CodeTrans(code translation between programming languages), CodeComment (code commenting), CodeExplain (code explanation), TestCase (generating test cases), and more.
- Multi-turn dialogue capability: It should support multi-turn conversations where user queries may or may not be related to the previous context, requiring the model to make accurate intent judgments.
- Non-toxic output: The model should generate content that is free from toxicity and harm. Besides adhering to basic human values and moral ethics, CodeFuse, being a code LLM, needs to pay special attention to code-related toxicity. For instance, it should not output content that may pose information security issues, such as fishing or Trojan programs.

Self-instruct instruction augmentation. We collected question-answer pairs for code-related tasks from the public domain as well as code semantic extraction as described in Section 2.3. However, the dataset for code tasks from public domain is much smaller than that of language tasks, and the code semantic extraction is restricted for certain code tasks. This requires new ways to augment datasets for poverty tasks. Thanks to the self-instruct techniques introduced in Alpaca [39], CodeFuse leverages those high-quality open or manual writing original dataset as seeds, and generate informative and contextually relevant outputs with the help of off-the-shelf models like ChatGPT. This approach is suitable for divergent data augmentation.

Instruction-following description. The mentioned methods allow us to construct suitable inputs and outputs for different tasks. However, it still has a step gap for an LLM to follow instructions, that is instruction description. Some data may already have instructions included in the input, while others require adding instruction descriptions to familiarize the model with the task-specific instructions. This enables the model to be triggered to follow similar instruction descriptions during usage. Moreover, in-context description and chain-of-thoughts (CoT) information can be added to enhance the model's performance.

Data format. In practice, data for different scenarios, such as multi-turn conversations, few-shot learning, and CoT tasks, comes from diverse channels and has complex formats. To handle this, we standardized the original JSON format for these different task types. Taking into account existing practices in academia and industry, we have ultimately decided to adopt ChatML (Chat Markup Language) [24] and have made optimizations accordingly. We have proactively addressed issues such as tokenization errors, instruction injection attacks, and Multi-Role-Playing scenarios in the format design. LLM model accepts a sequence of text inputs during training and inference and converts the original JSON data into a sequence of text in ChatML format. CodeFuse dialogue model supports three roles by default: System, Human, and Bot. The System role provides initial information and instructions, while the Human and Bot roles represent user inputs and model-generated responses, respectively.

Optimizations. During fine-tuning, the model focuses on learning the Bot's output by calculating the loss only for the Bot role. Additionally, CodeFuse introduces multi-task finetuning (MFT) since downstream code tasks are naturally divided into multi-tasks. MFT introduces well-designed loss functions based on multi-task learning that enables the model to effectively learn from each task even with different sample numbers, difficulties, and convergence speed. MFT can complement different tasks to achieve better results than SFT. Details on MFT will be elaborated in another paper.

3.5 Model operations

CodeFuse-13B was trained using 512 Nvidia A100 GPU cards, with a Hardware FLOPs Utilization (HFU) of approximately 60%. The training process took approximately 40 days to complete. Several key stability-related capabilities were developed to ensure the successful training process. In particular, we developed a cloud-based observability system with two major parts.

```
<| im_start |>system
Provide some context and/or instructions to the model.
<|im_end|>
<| im_start |>user
The user's message goes here
<|im_end|>
<| im_start |> assistant
```

Figure 5: An example of ChatML.

- **Training metrics observability** is essential for monitoring the training process, including important metrics like training/validation loss to assess convergence and computational FLOPs levels. CodeFuse has deployed a TensorBoard instance in the cloud, allowing users to access and analyze these metrics through a web browser.
- **Infrastructure metrics observability** covers various aspects such as GPU and RDMA. CodeFuse uses DCGM/NVML to gather GPU/RDMA performance metrics, ECC errors, and Xid errors. They also employ a node-side detection agent named Walle, to capture hardware anomaly information. The GPU Diagnose component handles fault recovery operations. The GPU cluster utilizes RDMA high-performance networking technology for efficient data transmission and processing.

We developed a GPU diagnosis system for proactive discovery and automatic handling failures of GPU nodes, as well as automatically restarting/recovering terminated training tasks without manual intervention. Within 30 minutes of a failure, the system identifies, isolates, and reschedules faulty GPU cards, allowing training to continue with the latest checkpoint. To maintain the high availability of GPU resources, we define the SLO metric for the schedulability of GPU cards, resulting in an increase of GPU availability from 87% to 94% for the cluster during the training cycle.

4 EVALUATION

In this section, we begin by introducing the models we assessed alongside CodeFuse. Our experiments are conducted by using the NVIDIA A100-SXM-80GB GPUs with a Linux system. We present a comprehensive analysis of the performance of all models on the HumanEval, HumanEval-x [8] and our CodefuseEval benchmarks. We conduct an evaluation of CodeFuse in comparison to CodeGeeX, utilizing a variety of code tasks with Chinese prompts.

4.1 Evaluation benchmarks and protocols

HumanEval, its extension HumanEval-x and MBPP are widely used benchmarks in the field of code LLMs. These benchmarks encompass a vast collection of programming problems, employing test cases to validate the code generated by code LLMs. However, when it comes to Chinese inputs, there is still a need for evaluation methods in certain code scenarios. To handle this issue, we developed and open-sourced CodefuseEval [3] to facilitate evaluation in code completion, code generation, cross-program-language code translation, code commenting, and test case generation scenarios with both English and Chinese prompts. The CodefuseEval benchmark is an extension of the HumanEval and MBPP benchmarks, covering five programming languages: Python, Java, C++, Go, and JavaScript.

For code generation tasks including code completion, text2code, code translation, and test cases generation, we adopt the `pass@k` metric as the evaluation criterion. For code commenting tasks, we use `Bleu` and `BleuRT` as the evaluation metrics. To comprehensively assess the model's code capabilities in the multi-task evaluation, three decoding strategies were employed: temperature sampling, greedy, and beam search. The prompts were formatted using zero-shot, allowing the model to generate responses without task-specific fine-tuning. When employing the temperature sampling strategy, we set the hyperparameters for `pass@1` as follows: `temperature`=0.2, `top_p`=0.95, and `generaten` = 10 samples.

4.2 Compared models

We compare CodeFuse to the following models with similar sizes. The statistic of compared models is from the published reports.

- GPT-NeoX-20B [5] is a 20 billion parameter autoregressive language model trained on the Pile [14].
- StarCoder [20] is a Code LLM with 15B parameters and a context size of 8K, which supports infilling capabilities and fast inference.
- CodeGeeX is a language model that has been trained on a collection of 23 programming languages. It is an open-sourced model with 13 billion parameters. Its training data was selected from the Pile[14], CodeParrot [47], and other datasets. In addition to these datasets, CodeGeeX also includes its own multi-language benchmark suite, HumanEval-x, which we discuss below.
- BaiduErnie [37, 38, 44] (Enhanced Representation through kNowledge IntEgration) is a language representation model enhanced by using knowledge masking strategies. The masking strategy of BaiduErnie inspired by BERT[11] includes phrase-level strategy and entity-level strategy.
- CodeGen [26] has two versions, CodeGen-Mono-16B is a variant of CodeGen-Multi-16B, specifically fine-tuned using additional Python code from GitHub.
- CodeT5+ [45], an encoder-decoder based Code LLM, boasts modular flexibility, accommodating diverse code-related downstream tasks.
- WizardCoder [23] is trained using the Evol-Instruct technique. It has shown remarkable improvement in performance on HumanEval python evaluations compared to previous models.
- PanGu-Coder2 [32] is trained by RRTF (Rank Responses to align Test&Teacher Feedback) framework with the same Evol-Instruct technique as WizardCoder. PanGu-Coder2 has achieved the leading performance on HumanEval among models of similar size. Like WizardCoder, PanGu-Coder2 is also a monolingual model.
- CodeLlama [30] is a family of large language models for code, based on Llama 2. It stands out among other open models with its state-of-the-art performance, infilling capabilities, support for large input contexts, and zero-shot instruction following ability for programming tasks.

4.3 Evaluation on code generation

In our comparison of CodeFuse with existing Code Language Models (LLMs) of similar size, we evaluated their code generation performance. To ensure fairness, we gathered statistics for other models in Table 4 from existing reports. However, the performance of CodeT5+ and CodeLlama on HumanEval-x was not presented in their respective articles or other published related work. As a result, Table 4 lacks data for these models.

We present the performance of multiple versions of CodeFuse because they demonstrate the progress of CodeFuse and achievements in different tasks. Most of these versions are deployed in various scenarios. Additionally, we showcase seven current mainstream multi-lingual models ranging in size from 13 billion to 16 billion parameters. We also present three state-of-the-art code Language Model Models (LLMs), among which PanGu-Coder2, released last month, achieved the highest HumanEval score.

In the HumanEval Python pass@1 evaluation, the open-sourced version of CodeFuse-13B outperforms other multi-lingual models. However, since mono-lingual models like CodeLlama-Python, WizardCoder, and PanGu-Coder2 have made specific optimizations for Python code generation, it is still challenging for CodeFuse to surpass them in Python evaluation.

4.4 Evaluation on multi-lingual code translation

We evaluate CodeFuse on multi-lingual code translation by comparing it to CodeGen-multi-16B and CodeGeeX-13B. Similarly to CodeGeeX-13B, which has a dedicated fine-tuned version for code translation called CodeGeeX-13B-FT, CodeFuse-13B-SFT is a multi-task fine-tuned version that includes code translation.

The code translation evaluation dataset we used is constructed based on MBXP and HumanEval-x. It includes cases that have been reviewed and corrected by experts and has been opened as a part of CodefuseEval.

Table 5 presents the pass@1 results for mutual conversion between Java, Python, and C++ using greedy decoding. The table clearly shows that CodeFuse-13B-SFT outperforms other models when translating Python code to the other two languages. Additionally, it achieves the highest average score across all six translation scenarios.

4.5 Evaluation on code-related tasks with Chinese prompts

We conducted an evaluation of CodeFuse on supporting Chinese prompts by comparing it to CodeGeeX, which is known for its excellent Chinese language support. Figure 6 showcases examples of code generation, code translation, code comments, and test case generation with prompts in Chinese. In addition to the evaluation of code generation and translation mentioned in the previous sections, testcase generation, code comments, and explanation are included in our evaluation.

In the code comment and explanation tasks, We specifically evaluated our CodeFuse, by integrating it into AntGroup's development process. Valuable feedback from developers during daily work was collected to assess performance in real-world scenarios. We did not compare CodeFuse with other models due to the time and effort required to deploy a competing model in our daily development process. After collecting human feedback over several months, CodeFuse-13B-SFT achieved a `Bleu` score of 42.42% and a `BleuRT` score of 36.34% as shown in Table 6. These results indicate that CodeFuse is indeed useful in real development scenarios.

Recently, there have been efforts to automatically generate test cases by providing a code snippet to LLMs [31, 43, 49]. In our evaluation, we used the HumanEval-x dataset and selected CodeGeeX as the baseline model due to its comparable model size, support for Chinese prompts, and relevance to industry applications. We made minor adjustments to the dataset to adhere to the prompt format shown in Figure 6(c). As shown in Table 7, CodeFuse outperforms CodeGeeX in the task of testcase generation with Chinese prompts.

5 RELATED WORK

Large language models. LLMs have exhibited remarkable accomplishments across a wide range of tasks. Leading technology companies have made substantial progress in creating highly capable LLMs. Notable examples include OpenAI's GPT3&4 [6, 27], Google's PaLM [2, 9], DeepMind's Chinchilla [17] and Gopher [28], as well as Anthropic's Claude4. However, these models are closed-source and only accessed through specific APIs.

Several open-source LLMs have been released and made valuable contributions to the public. EleutherAI has contributed GPT-NeoX-20B [5] and GPT-J-6B [42]. Google has released UL2-20B. Tsinghua University has introduced GLM-130B [50, 52]. Meta has released LLaMA [40] and LLaMA2 [41].

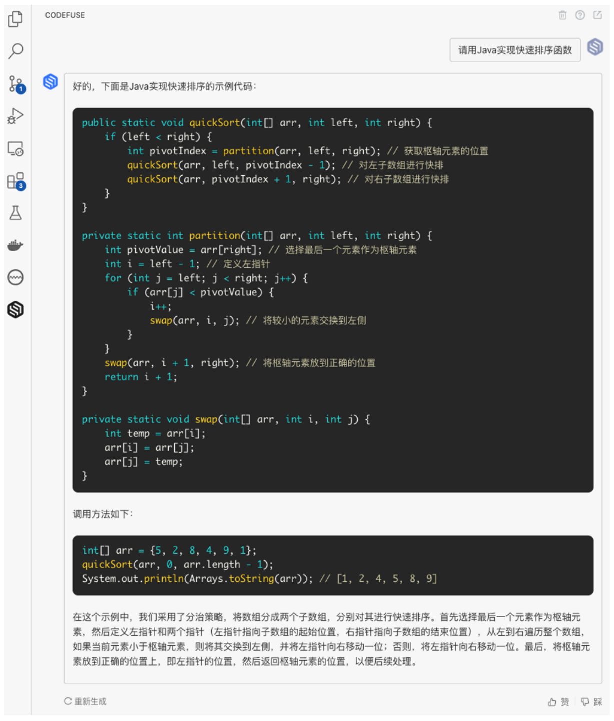
Code large language models. Many works have introduced LLMs to tackle the challenges of code understanding and generation problems. Codex [7] has powered Copilot for code tasks. Google has proposed PaLM-Coder [9]. These models have shown exceptional performance on popular code completion benchmarks such as HumanEval [8] and MBPP [3]. However, it is important to note that these models are closed-source.

There also are several open-source Code LLMs available. Salesforce has developed CodeGen [26], CodeT5 [46], and CodeT5+ [45]. Tsinghua University has contributed CodeGeeX [52], and the BigCode Project has created StarCoder [20]. While recent models like WizardCode [23], PanGu-Coder2 [32], and CodeLLaMA [30] have achieved impressive scores on HumanEval, it is worth noting that they are weak for multi-lingual prompts. In contrast, CodeFuse is designed to be a multi-lingual code Language Model (LLM) that supports various code-related tasks across both English and Chinese prompots. This makes CodeFuse a valuable solution for developers working in diverse linguistic environments.

6 DISCUSSION, CONCLUSION, FUTURE WORK

Extending MFT framework to support opensource models. We develop a multi-task finetuning framework and open-source it as MFTCoder[4]. This framework can be used to fine-tune both our in-house developed CodeFuse-13B and newly emerging open-sourced Code LLMs such as StarCoder [20] and CodeLLaMA [30]. We fine-tuned StarCoder and CodeLLaMA models with the MFT-Coder framework on our collected datasets, and open-sourced the

[4]https://github.com/codefuse-ai/MFTCoder



请用Java实现快速排序函数

好的，下面是Java实现快速排序的示例代码：

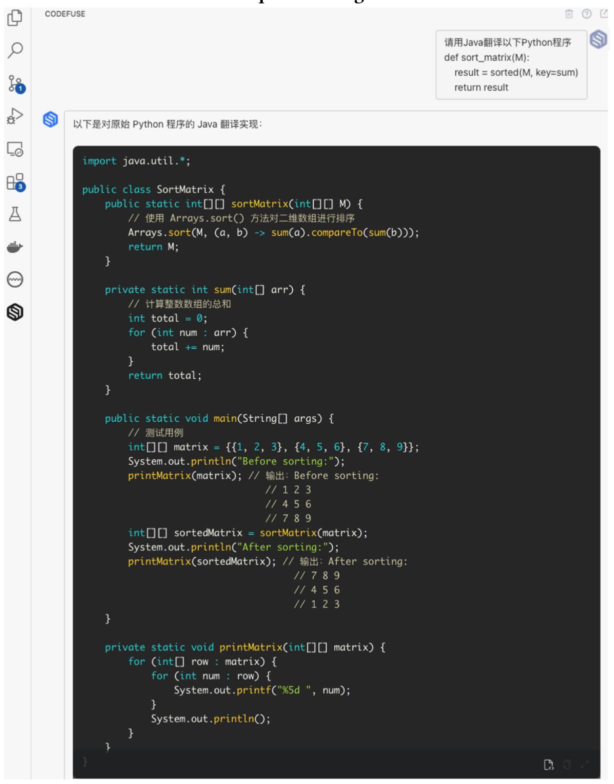
```java
public static void quickSort(int[] arr, int left, int right) {
    if (left < right) {
        int pivotIndex = partition(arr, left, right); // 获取枢轴元素的位置
        quickSort(arr, left, pivotIndex - 1); // 对左子数组进行快排
        quickSort(arr, pivotIndex + 1, right); // 对右子数组进行快排
    }
}

private static int partition(int[] arr, int left, int right) {
    int pivotValue = arr[right]; // 选择最后一个元素作为枢轴元素
    int i = left - 1; // 定义左指针
    for (int j = left; j < right; j++) {
        if (arr[j] < pivotValue) {
            i++;
            swap(arr, i, j); // 将较小的元素交换到左侧
        }
    }
    swap(arr, i + 1, right); // 将枢轴元素放到正确的位置
    return i + 1;
}

private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

调用方法如下：

```java
int[] arr = {5, 2, 8, 4, 9, 1};
quickSort(arr, 0, arr.length - 1);
System.out.println(Arrays.toString(arr)); // [1, 2, 4, 5, 8, 9]
```

在这个示例中，我们采用了分治策略，将数组分成两个子数组，分别对其进行快速排序。首先选择最后一个元素作为枢轴元素，然后定义左指针和两个指针（左指针指向子数组的起始位置，右指针指向子数组的结束位置），从左到右遍历整个数组，如果当前元素小于枢轴元素，则将其交换到左侧，并将左指针向右移动一位；否则，将左指针向右移动一位。最后，将枢轴元素放到正确的位置上，即左指针的位置，然后返回枢轴元素的位置，以便后续处理。

a. An example of code generation.

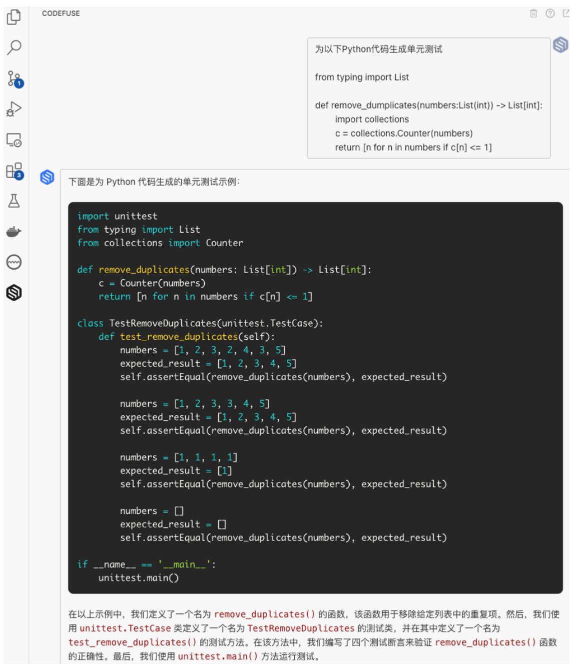


请用Java翻译以下Python程序
```python
def sort_matrix(M):
    result = sorted(M, key=sum)
    return result
```

以下是对原始 Python 程序的 Java 翻译实现：

```java
import java.util.*;

public class SortMatrix {
    public static int[][] sortMatrix(int[][] M) {
        // 使用 Arrays.sort() 方法对二维数组进行排序
        Arrays.sort(M, (a, b) -> sum(a).compareTo(sum(b)));
        return M;
    }

    private static int sum(int[] arr) {
        // 计算整数数组的总和
        int total = 0;
        for (int num : arr) {
            total += num;
        }
        return total;
    }

    public static void main(String[] args) {
        // 测试用例
        int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
        System.out.println("Before sorting:");
        printMatrix(matrix); // 输出: Before sorting:
                                   // 1 2 3
                                   // 4 5 6
                                   // 7 8 9
        int[][] sortedMatrix = sortMatrix(matrix);
        System.out.println("After sorting:");
        printMatrix(sortedMatrix); // 输出: After sorting:
                                       // 7 8 9
                                       // 4 5 6
                                       // 1 2 3
    }

    private static void printMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int num : row) {
                System.out.printf("%5d ", num);
            }
            System.out.println();
        }
    }
}
```

b. An example of code translation.



为以下Python代码生成单元测试

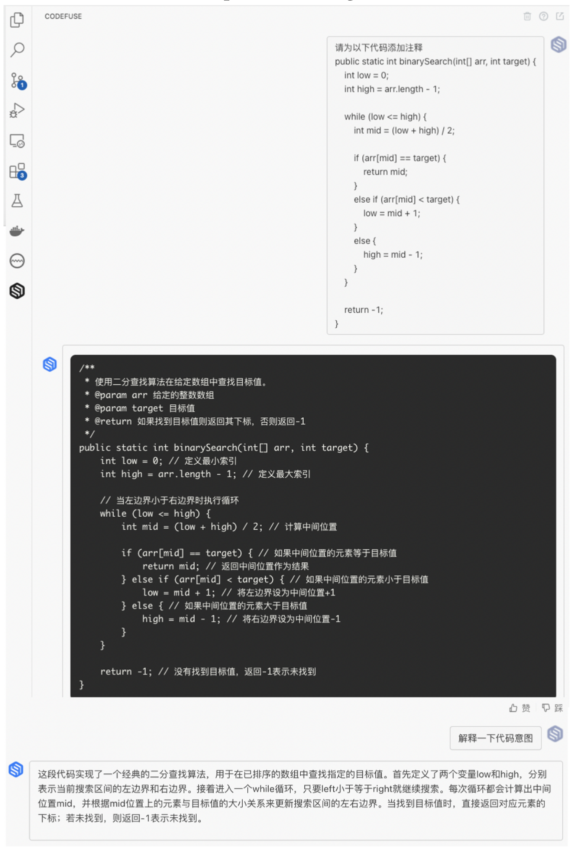
```python
from typing import List

def remove_dumplicates(numbers:List(int)) -> List[int]:
    import collections
    c = collections.Counter(numbers)
    return [n for n in numbers if c[n] <= 1]
```

下面是为 Python 代码生成的单元测试示例：

```python
import unittest
from typing import List
from collections import Counter

def remove_duplicates(numbers: List[int]) -> List[int]:
    c = Counter(numbers)
    return [n for n in numbers if c[n] <= 1]

class TestRemoveDuplicates(unittest.TestCase):
    def test_remove_duplicates(self):
        numbers = [1, 2, 3, 2, 4, 3, 5]
        expected_result = [1, 2, 3, 4, 5]
        self.assertEqual(remove_duplicates(numbers), expected_result)

        numbers = [1, 2, 3, 3, 4, 5]
        expected_result = [1, 2, 3, 4, 5]
        self.assertEqual(remove_duplicates(numbers), expected_result)

        numbers = [1, 1, 1, 1]
        expected_result = [1]
        self.assertEqual(remove_duplicates(numbers), expected_result)

        numbers = []
        expected_result = []
        self.assertEqual(remove_duplicates(numbers), expected_result)

if __name__ == '__main__':
    unittest.main()
```

在以上示例中，我们定义了一个名为 `remove_duplicates()` 的函数，该函数用于移除给定列表中的重复项。然后，我们使用 `unittest.TestCase` 类定义了一个名为 `TestRemoveDuplicates` 的测试类，并在其中定义了一个名为 `test_remove_duplicates()` 的测试方法。在该方法中，我们编写了四个测试断言来验证 `remove_duplicates()` 函数的正确性。最后，我们使用 `unittest.main()` 方法运行测试。

c. An example of testcase generation.



请为以下代码添加注释
```java
public static int binarySearch(int[] arr, int target) {
  int low = 0;
  int high = arr.length - 1;

  while (low <= high) {
    int mid = (low + high) / 2;

    if (arr[mid] == target) {
      return mid;
    }
    else if (arr[mid] < target) {
      low = mid + 1;
    }
    else {
      high = mid - 1;
    }
  }

  return -1;
}
```

```java
/**
 * 使用二分查找算法在给定数组中查找目标值。
 * @param arr 给定的整数数组
 * @param target 目标值
 * @return 如果找到目标值则返回其下标，否则返回-1
 */
public static int binarySearch(int[] arr, int target) {
    int low = 0; // 定义最小索引
    int high = arr.length - 1; // 定义最大索引

    // 当左边界小于右边界时执行循环
    while (low <= high) {
        int mid = (low + high) / 2; // 计算中间位置

        if (arr[mid] == target) { // 如果中间位置的元素等于目标值
            return mid; // 返回中间位置作为结果
        } else if (arr[mid] < target) { // 如果中间位置的元素小于目标值
            low = mid + 1; // 将左边界设为中间位置+1
        } else { // 如果中间位置的元素大于目标值
            high = mid - 1; // 将右边界设为中间位置-1
        }
    }

    return -1; // 没有找到目标值，返回-1表示未找到
}
```

解释一下代码意图

这段代码实现了一个经典的二分查找算法，用于在已排序的数组中查找指定的目标值。首先定义了两个变量low和high，分别表示当前搜索区间的左边界和右边界。接着进入一个while循环，只要left小于等于right就继续搜索。每次循环都会计算出中间位置mid，并根据mid位置上的元素与目标值的大小关系来更新搜索区间的左右边界。当找到目标值时，直接返回对应元素的下标；若未找到，则返回-1表示未找到。

d. An example of code comments and explanation.

Figure 6: Examples of CodeFuse generation with prompts in Chinese. The results are generated by CodeFuse VSCode extension, and indicators such as "#" have been removed for better human readability.

Table 4: Performance comparison of CodeFuse with previous models with similar size on HumanEval-x.

Lingual	Models	HumanEval-x pass@1				
		Python	Java	C++	JavaScript	Go
CodeFuse	CodeFuse-13B-Base	24.83%	23.78%	22.08%	19.62%	18.17%
	CodeFuse-13B-SFT	37.10%	26.22%	19.51%	31.71%	24.39%
Multi-	GPT-NeoX-20B	13.83%	8.87%	9.90%	11.28%	5.00%
	CodeGEEX-13B	22.89%	20.04%	17.06%	17.59%	14.43%
	Baidu-ERNIE-3.5-15.5B	35.37%	26.22%	20.11%	34.76%	27.43%
	StarCoder-15.5B	33.57%	30.22%	31.55%	30.79%	17.61%
	CodeGen-multi-16B	19.22%	14.95%	18.05%	18.40%	13.03%
	CodeT5+-16B	30.90%				
	CodeLlama-13B	36.00%				
Mono-	CodeLlama-Python-13B	43.30%				
	WizardCoder-16B	57.30%				
	PanGu-Coder2-15B	61.64%				

Table 5: Performance(pass@1) comparison of CodeFuse with previous models on code translation using greedy decoding

Models	Java to Py	C++ to Py	C++ to Java	Java to C++	Py to Java	Py to C++	Average
CodeFuse-13B-Base	53.66%	55.49%	41.46%	37.80%	48.10%	50.00%	47.75%
CodeFuse-13B-SFT	66.46%	59.15%	54.27%	47.56%	**56.31%**	**55.40%**	**56.53%**
CodeGen-multi-16B	52.73%	33.83%	43.20%	41.42%	29.27%	35.94%	39.40%
CodeGeeX-13B	43.41%	27.18%	22.56%	39.33%	25.84%	26.54%	30.81%
CodeGeeX-13B-FT	**75.03%**	**62.79%**	**71.68%**	**49.67%**	41.98%	34.16%	55.89%

Table 6: Evaluation of Chinese code comments and explanations on CodefuseEval and real feedback

Models	Bleu	BleuRT
CodeFuse-13B-Base	36.75%	27.76%
CodeFuse-13B-SFT	42.42%	36.34%

Table 7: Evaluation of Chinese testcase generation on CodefuseEval

Models	pass@1 Python	pass@1 Java
CodeFuse-13B-SFT	31.20%	24.32%
CodeGeeX-13B	22.89%	20.04%

fine-tuned version as CodeFuse-CodeLlama-34B and CodeFuse-StarCoder-15B. CodeFuse-CodeLlama-34B achieves 74.4% pass@1 score on HumanEval, which surpasses the score by GPT4 and ChatGPT-3.5, and represents the state-of-the-art results for open-sourced Language Model Models (LLMs). It is also evident that the data preparation strategy in CodeFuse greatly enhances the performance of other code LLMs.

Deployment. CodeFuse is deployed in the production environment within AntGroup in the form of both IDE plugin and web-based chat. See Figure 6 for some examples. To facilitate the model response time and service throughput, we introduce a series of optimizations for the model service, which include (1) quantizing the model to 4bits with negligible accuracy loss using automatic iterative refined GPTQ [12]; (2) leveraging software optimization provided by Nvidia TensorRT-LLM [5]; (3) performing service optimization through semantic cache and streaming output. The product now supports daily software developing of more than 5K engineers in AntGroup.

Conclusion. This paper introduces CodeFuse-13B, an open-sourced pre-trained Language Model (LLM) with 13 billion parameters designed for code-related tasks with multi-lingual (English and Chinese) prompts. It supports over 40 programming languages and utilizes a carefully filtered pre-training dataset. Experiments using real-world scenarios and industry benchmarks demonstrate that CodeFuse-13B achieves a HumanEval Pass@1 score of 37.10%, making it one of the top multi-lingual models with similar parameter sizes. It outperforms other models in code generation, translation, comments, and testcase generation tasks with Chinese inputs. Valuable human feedback from AntGroup's software development process confirms the successful integration of CodeFuse-13B.

Future work. Besides models of CodeFuse and the MFTCoder framework, we plan to further open-source two significant components of CodeFuse: the CodefuseEval benchmark and the `Sparrow` program query system for high-quality code data cleaning. By open-sourcing these components, we aim to contribute to the research community and facilitate further advancements in the full lifecycle of AI native software development.

[5]https://developer.nvidia.com/tensorrt-llm-early-access

REFERENCES

[1] Loubna Ben Allal, Raymond Li, Denis Kocetkov, et al. 2023. SantaCoder: don't reach for the stars! arXiv:2301.03988 [cs.SE]

[2] Rohan Anil, Andrew M. Dai, Orhan Firat, et al. 2023. PaLM 2 Technical Report. arXiv:2305.10403 [cs.CL]

[3] Jacob Austin, Augustus Odena, Maxwell Nye, et al. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer Normalization. arXiv:1607.06450 [stat.ML]

[5] Sid Black, Stella Biderman, Eric Hallahan, et al. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. arXiv:2204.06745 [cs.CL]

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[7] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[8] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, et al. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311 [cs.CL]

[10] Tri Dao, Daniel Y. Fu, Stefano Ermon, et al. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG]

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[12] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323* (2022).

[13] Daniel Fried, Armen Aghajanyan, Jessy Lin, et al. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999 [cs.SE]

[14] Leo Gao, Stella Biderman, Sid Black, et al. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. arXiv:2101.00027 [cs.CL]

[15] Ant Group. 2023. Sparrow. http://sparrow.alipay.com.

[16] Dan Hendrycks and Kevin Gimpel. 2023. Gaussian Error Linear Units (GELUs). arXiv:1606.08415 [cs.LG]

[17] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, et al. 2022. Training Compute-Optimal Large Language Models. arXiv:2203.15556 [cs.CL]

[18] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[19] Denis Kocetkov, Raymond Li, Loubna Ben Allal, et al. 2022. The Stack: 3 TB of permissively licensed source code. arXiv:2211.15533 [cs.CL]

[20] Raymond Li, Loubna Ben Allal, Yangtian Zi, et al. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]

[21] Yujia Li, David Choi, Junyoung Chung, et al. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (dec 2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[22] Jiangchao Liu, Jierui Liu, Peng Di, et al. 2023. Hybrid Inlining: A Framework for Compositional and Context-Sensitive Static Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 114–126.

[23] Ziyang Luo, Can Xu, Pu Zhao, et al. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568 [cs.CL]

[24] MicroSoft. 2023. ChatML. https://github.com/openai/openai-python/blob/main/chatml.md.

[25] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, et al. 2007. Keynote Address: .QL for Source Code Analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. 3–16. https://doi.org/10.1109/SCAM.2007.31

[26] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, et al. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs.LG]

[27] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[28] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, et al. 2022. Scaling Language Models: Methods, Analysis & Insights from Training Gopher. arXiv:2112.11446 [cs.CL]

[29] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. arXiv:1910.02054 [cs.LG]

[30] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, et al. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[31] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. arXiv:2302.06527 [cs.SE]

[32] Bo Shen, Jiaxin Zhang, Taihong Chen, et al. 2023. PanGu-Coder2: Boosting Large Language Models for Code with Ranking Feedback. arXiv:2307.14936 [cs.CL]

[33] Qingkai Shi, Xiao Xiao, Rongxin Wu, et al. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 693–706.

[34] Yusuke Shibata, Takuya Kida, Shuichi Fukamachi, et al. 1999. Byte Pair Encoding: A Text Compression Scheme That Accelerates Pattern Matching. (09 1999).

[35] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, et al. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL]

[36] Jianlin Su, Yu Lu, Shengfeng Pan, et al. 2022. RoFormer: Enhanced Transformer with Rotary Position Embedding. arXiv:2104.09864 [cs.CL]

[37] Yu Sun, Shuohuan Wang, Yukun Li, et al. 2019. ERNIE 2.0: A Continual Pre-training Framework for Language Understanding. arXiv:1907.12412 [cs.CL]

[38] Yu Sun, Shuohuan Wang, Yukun Li, et al. 2019. ERNIE: Enhanced Representation through Knowledge Integration. arXiv:1904.09223 [cs.CL]

[39] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, et al. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

[40] Hugo Touvron, Thibaut Lavril, Gautier Izacard, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[41] Hugo Touvron, Louis Martin, Kevin Stone, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[42] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

[43] Junjie Wang, Yuchao Huang, Chunyang Chen, et al. 2023. Software Testing with Large Language Model: Survey, Landscape, and Vision. arXiv:2307.07221 [cs.SE]

[44] Shuohuan Wang, Yu Sun, Yang Xiang, et al. 2021. ERNIE 3.0 Titan: Exploring Larger-scale Knowledge Enhanced Pre-training for Language Understanding and Generation. arXiv:2112.12731 [cs.CL]

[45] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, et al. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. arXiv:2305.07922 [cs.CL]

[46] 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. arXiv:2109.00859 [cs.CL]

[47] Thomas Wolf, Lysandre Debut, Victor Sanh, et al. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45.

[48] Ruibin Xiong, Yunchang Yang, Di He, et al. 2020. On Layer Normalization in the Transformer Architecture. arXiv:2002.04745 [cs.LG]

[49] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, et al. 2021. Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, 435–450.

[50] Aohan Zeng, Xiao Liu, Zhengxiao Du, et al. 2022. GLM-130B: An Open Bilingual Pre-trained Model. arXiv:2210.02414 [cs.CL]

[51] Wayne Xin Zhao, Kun Zhou, Junyi Li, et al. 2023. A Survey of Large Language Models. arXiv:2303.18223 [cs.CL]

[52] Qinkai Zheng, Xiao Xia, Xu Zou, et al. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. arXiv:2303.17568 [cs.LG]

[53] Zexin Zhong, Jiangchao Liu, Diyu Wu, Peng Di, et al. 2022. Field-Based Static Taint Analysis for Industrial Microservices. In *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 149–150.

[54] Zexin Zhong, Jiangchao Liu, Diyu Wu, Peng Di, et al. 2023. Scalable Compositional Static Taint Analysis for Sensitive Data Tracing on Industrial Micro-Services. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 110–121.

[55] Xin Zhou, Kisub Kim, Bowen Xu, et al. 2023. The Devil is in the Tails: How Long-Tailed Code Distributions Impact Large Language Models. arXiv:2309.03567 [cs.SE]