Comparative Analysis of Transformer Models for Code Summarization

Abstract—**This study investigates the performance of three prominent code summarization models—CodeT5, CodeBERT, and GPT-2—when applied to a custom dataset of Python repositories. The research examines how each model's architectural strengths and limitations impact their effectiveness in generating accurate and efficient code summaries. A key focus is placed on the influence of dataset quality and training resource constraints on model performance. My findings reveal that CodeT5 outperforms the others, demonstrating superior capability in handling structured code summarization tasks due to its sequence-to-sequence, auto-regressive architecture. In contrast, CodeBERT and GPT-2 showed varied success; their performance was notably affected by the richness of the dataset context and the adequacy of training resources. The study also identifies significant threats to validity, including the risk of reduced model generalizability across different coding environments and the potential inaccuracies in ground truth data that could skew efficiency and accuracy assessments. To address these challenges, I suggest enhancing dataset design and optimizing computational resource allocation to improve training outcomes. Future research should explore these dimensions to develop more robust models capable of delivering high-quality automated code documentation across diverse software development frameworks. This work lays the groundwork for advancing automated tools that assist in maintaining and understanding complex codebases, contributing to improved software development practices and efficiencies.**

Index Terms—**Code Summarization, transformer model, repositories, codeT5**

I. Introduction

In the rapidly evolving domain of software engineering, the ability to quickly understand and manipulate code is crucial. Code summarization, an essential task within this field, aims to generate brief yet comprehensive descriptions of code snippets. By automating this process, developers can significantly enhance their efficiency, especially when dealing with large and complex codebases. This task employs sophisticated techniques from natural language processing (NLP) and machine learning (ML) to convert raw code into succinct, human-readable summaries. These summaries not only aid in comprehending the functionality of code segments but also facilitate more efficient code reviews, debugging, and maintenance.[1]

With the increasing complexity and volume of software development, effective code summarization has emerged as a critical tool. It enables developers to save considerable time typically spent on manual code inspection and comprehension, which are often recognized as bottlenecks in the software development lifecycle. Automated summarization tools are particularly valuable for onboarding new developers, who must quickly become acquainted with large existing codebases, and for experienced developers who are tasked with navigating and modifying vast amounts of legacy code.[1]

The strategic implementation of code summarization can also promote better documentation practices and consistency across projects, which are often neglected in fast-paced development environments. By providing a clear and consistent overview of code functionalities, these tools help maintain high standards of code quality and coherence across teams and projects.[2]

Despite its significant potential, code summarization presents numerous challenges. Code differs fundamentally from natural language, possessing its own unique syntax, semantics, and structure, which makes the direct application of traditional NLP techniques less effective. The lack of explicit and comprehensive documentation in many codebases further complicates the summarization process, as does the contextual complexity within which code operates. For example, understanding the purpose and functionality of a piece of code often requires knowledge of its runtime environment, dependencies, and the specific problem it addresses.[3]

Additionally, the high variability in coding styles and the technical debt in many projects means that automated tools must be exceptionally robust and adaptable to diverse coding practices and languages. These challenges necessitate not only the development of advanced ML models that are fine-tuned to the nuances of code but also innovative approaches to training these models effectively on suitable datasets.

Overview of Our Approach

This paper investigates the performance of three state-of-the-art NLP models—CodeT5, CodeBERT, and GPT-2—on a meticulously curated dataset of Python repositories. This dataset has been selected to represent a wide range of coding practices and includes repositories known for their high quality and active maintenance. This research is structured around specific questions that seek to uncover both the strengths and limitations of these models in the context of automated code summarization.

Research Questions

I pose the following research questions to guide our exploration and analysis:

RQ1: How do different code summarization models (CodeT5, CodeBERT, and GPT-2) compare in terms of accuracy and efficiency when applied to a custom dataset of Python repositories?

RQ2: What impact does the quality and structure of the dataset have on the performance of code summarization models?

Through these questions, I aim to advance the understanding of how automated tools can be optimized to improve the efficiency and effectiveness of code summarization processes, ultimately contributing to more agile and robust software development practices.

II. Related work

Overview of Code Summarization Research Code summarization has garnered significant attention in recent years, with numerous studies exploring various approaches to automatically generate summaries for code snippets. This section reviews key contributions in the field, focusing on different methodologies and their effectiveness.

Template-Based Methods Early research in code summarization often relied on template-based methods. Sridhara et al. (2010) proposed a technique that used predefined templates to generate summaries for Java methods. These templates were designed to capture common patterns in method signatures and comments. While effective for simple cases, template-based methods struggled with more complex and diverse codebases.[4]

Statistical Methods Statistical approaches marked a significant advancement over template-based methods. Haiduc et al. (2010) introduced a method that used information retrieval techniques to generate summaries by identifying important terms in the code. McBurney and McMillan (2014) extended this work by incorporating topic modeling to improve the relevance of the generated summaries. These methods leveraged statistical patterns in code and comments but often fell short in capturing the deeper semantics of the code.[5]

Neural Network-Based Methods The advent of deep learning has revolutionized code summarization. Iyer et al. (2016) were among the first to apply neural networks to this task, using a sequence-to-sequence (Seq2Seq) model to generate summaries for code snippets. This approach demonstrated the potential of neural networks to learn complex patterns in code and natural language.[6]

CodeBERT Feng et al. (2020) introduced CodeBERT, a bimodal pre-trained model for programming and natural languages. CodeBERT is trained on a large corpus of code and natural language pairs, enabling it to generate meaningful summaries by capturing the relationships between code and its corresponding documentation. Empirical evaluations showed that CodeBERT outperformed traditional methods and earlier neural models in terms of accuracy and coherence.[7]

CodeT5 Wang et al. (2021) proposed CodeT5, a model built on the T5 (Text-to-Text Transfer Transformer) architecture. CodeT5 treats all NLP tasks as text-to-text tasks, allowing it to generate summaries by converting code into natural language descriptions. It is pre-trained on a diverse set of programming languages, making it highly versatile and effective for code summarization. Experiments demonstrated that CodeT5 achieved state-of-the-art performance on several code summarization benchmarks.[8]

GPT-2 Radford et al. (2019) developed GPT-2, a generative pre-trained transformer model originally designed for natural language generation. While not specifically tailored for code, GPT-2 has shown promise in generating code summaries due to its ability to model complex sequences and generate coherent text. Fine-tuning GPT-2 on code datasets has yielded competitive results, highlighting its potential for code summarization tasks.[9]

Impact of Dataset Quality The quality and structure of the dataset play a crucial role in the performance of code summarization models. Hu et al. (2018) emphasized the importance of high-quality datasets with well-documented code and diverse examples. Their study showed that models trained on datasets with rich documentation and a variety of code examples tend to perform better in generating meaningful and coherent summaries.[10]

The advancements in neural network-based methods, particularly with models like CodeBERT, CodeT5, and GPT-2, have significantly improved the state of code summarization. These models leverage large-scale datasets and sophisticated architectures to generate high-quality summaries, addressing many of the limitations of earlier approaches. The quality of the dataset remains a critical factor in the success of these models, highlighting the importance of careful dataset curation and preprocessing.[11]

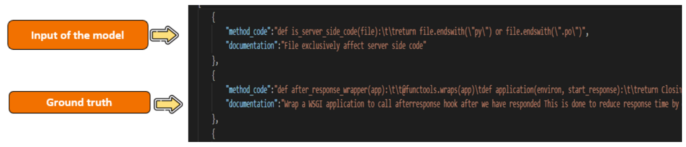
III. Methodology

A. Dataset Preparation

My study utilizes a custom dataset composed of Python repositories chosen for their robust activity levels and quality of documentation. Each repository provides code snippets and corresponding expert-written summaries that serve as the ground truth.

Data Cleaning and Preprocessing: Standardization processes include removing irrelevant elements (comments, extra whitespaces), tokenizing the code using a Python-specific tokenizer to maintain syntactic accuracy, and encoding the summaries.

Data Split: The dataset is segmented into training (70%), validation (15%), and testing (15%) subsets to ensure diverse representation across different model evaluations.



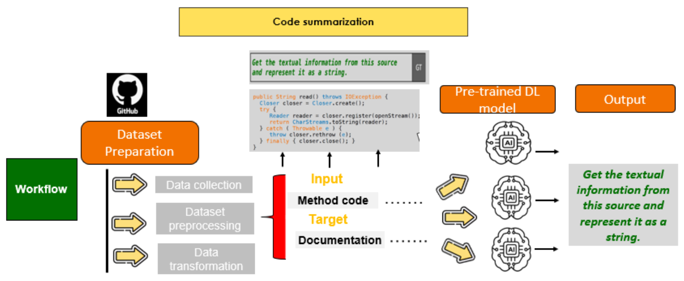
```
{
    "method_code":"def is_server_side_code(file):\t\treturn file.endswith(\"py\") or file.endswith(\".po\")",
    "documentation":"File exclusively affect server side code"
},
{
    "method_code":"def after_response_wrapper(app):\t\t@functools.wraps(app)\tdef application(environ, start_response):\t\treturn Closi
    "documentation":"Wrap a WSGI application to call afterresponse hook after we have responded This is done to reduce response time by
},
{
```

Fig. 1: Model Input Pattern

B. Model Architectures

CodeT5

CodeT5 extends the T5 model, which was designed to convert all NLP tasks into a unified text-to-text format. This means that every task (be it translation, summarization, or classification) is treated as a problem of generating text output from text input. CodeT5 adapts this framework for the programming language domain, where tasks might include



```
Get the textual information from this source
and represent it as a string.
```

```
public String read() throws IOException {
  Closer closer = Closer.create();
  try {
     Reader reader = closer.register(openStream());
     return CharStreams.toString(reader);
  } catch ( Throwable e ) {
     throw closer.rethrow (e);
  } finally { closer.close(); }
}
```

```
Get the textual information from this source and represent it as a string.
```

Fig. 2: Workflow overview

code summarization, code generation, code translation, and more.

Architecture of CodeT5

Unified Text-to-Text Approach: Similar to T5, CodeT5 treats all inputs and outputs as sequences of text. For code-related tasks, this means that inputs (e.g., code snippets) and outputs (e.g., summaries or comments) are strings of text. CodeT5 employs a tokenizer that is aware of programming language syntax. This tokenizer is capable of splitting text into tokens that represent syntactically meaningful units of code, which helps preserve the structure of the programming language in the model's input.[8]

CodeT5 uses a standard transformer architecture with an encoder and a decoder. Both components are composed of multiple layers that include self-attention mechanisms and feed-forward networks.[8]

Encoder: The encoder processes the input sequence into a continuous representation that captures both the content and context of the input. It uses self-attention layers to weigh the importance of different tokens in the input sequence.

Decoder: The decoder generates the output sequence step-by-step from the encoded information. It also uses self-attention layers to focus on different parts of the input sequence and a cross-attention layer that attends to the encoder's output while generating each token.

In both the encoder and decoder, the self-attention mechanism allows the model to consider other tokens in the input or previously generated tokens in the output when processing or generating a token. This mechanism is vital for understanding the context and dependencies within code.

Equations The transformer model relies on self-attention mechanisms and feed-forward neural networks. The key equations for the transformer are:

Self-Attention Mechanism:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

For the feed-forward network (FFN):

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2}$$

Positional Encoding: Unlike recurrent neural networks, transformers do not process sequences in order. To account for the sequence order, positional encodings are added to the input embeddings at the bottoms of the encoder and decoder stacks. This helps the model use the order of tokens in its reasoning.[12] 5. Task-Specific Adaptations

Pre-training Objectives: CodeT5 is pre-trained on a mixture of natural language and programming language tasks. Its pre-training includes objectives specific to code, such as code completion, code correction, and translating comments between programming languages. These tasks help the model learn programming language patterns and nuances.[12]

Equation: The output summary y for a given input code x can be modeled as:

$$P(y|x) = \prod_{t=1}^{n} P(y_t | y_{<t}, x; \theta) \tag{3}$$

[12]

CodeBERT CodeBERT is a groundbreaking model in the field of programming language processing, developed by Microsoft. It extends the capabilities of BERT (Bidirectional Encoder Representations from Transformers), which is originally designed for natural language processing tasks, to the domain of programming languages. CodeBERT is unique in its bimodal nature, designed to understand both programming languages and natural language, making it highly effective for tasks that involve code understanding and documentation, such as code search, code documentation, and code summarization.[13]

CodeBERT is built to process and understand both natural language text and programming code. This dual capability is essential for tasks like code summarization where understanding the code and expressing its functionality in human-readable form are equally important. CodeBERT utilizes the same architecture as BERT, which is based on the transformer model. It consists of multiple layers of transformer blocks that use self-attention mechanisms to process input data.[13]

Pre-trained with Masked Language Modeling (MLM) and Replaced Token Detection (RTD): CodeBERT was initially pre-trained on a large corpus comprising both monolingual (code-only) and bimodal (code paired with natural language) data. MLM involves masking some percentage of the input tokens randomly and then predicting those masked tokens. RTD, a task specific to CodeBERT's training, involves predicting whether a token is original or replaced in a given sequence, enhancing its understanding of syntax and semantics.[14]

Masked Language Modeling (MLM) For the MLM equation, we want to express the loss for predicting masked tokens within an input sequence.

$$L_{\text{MLM}} = -\sum_{i \in \text{masked}} \log P(x_i \mid x_{\text{masked}}) \tag{4}$$

[12] This equation uses the log probability of correctly predicting the masked token x_i, summed over all masked tokens in the sequence. The sequence with some tokens masked is represented by $masked_x$

Replaced Token Detection (RTD) For the RTD equation, which is used to calculate the loss based on whether tokens in the sequence have been replaced or not,

$$L_{\text{RTD}} = -\sum_i \log P(y_i \mid x) \tag{5}$$

[12] Here, y_i, represents a binary label indicating whether the token x_i has been replaced, and x is the input sequence.

Unlike some other transformer models used for translation or text generation, CodeBERT does not include a decoder. It is designed as an encoder-only model, which processes the input and outputs a rich vector representation of the input text. These representations can then be used for various downstream tasks, such as classification or token-level tasks.[14]

CodeBERT uses a tokenizer that is capable of understanding programming syntax, which is crucial for accurately capturing the structure and meaning of code. Although primarily trained on English and programming languages, CodeBERT's architecture and training enable it to adapt to cross-lingual settings, facilitating its use on code written in various programming languages paired with different human languages.[14]

GP2

GPT-2, short for Generative Pre-trained Transformer 2, is an advanced version of the original Generative Pre-trained Transformer (GPT) developed by OpenAI. It is primarily designed for natural language understanding and generation. Still, its flexible architecture allows it to be adapted for various other tasks, including those involving programming languages like code summarization and code generation.[15]

GPT-2 is based on the transformer architecture, specifically leveraging the decoder part of the transformer model. It consists of stacked transformer blocks that use masked self-attention layers, where each layer only allows attention to earlier positions in the sequence. This setup is ideal for language modeling and generation tasks where the sequence's future tokens should not influence the current token. One of the defining features of GPT-2 is its scale. The model comes in various sizes, with the largest having 1.5 billion parameters. This large scale allows GPT-2 to capture a vast range of nuances in data.[9]

$$L_{\text{CLM}} = -\sum_t \log P(x_t \mid x_{<t}) \tag{6}$$

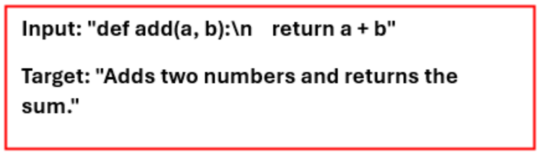
[12]

where x_t is the token at position t, and $x_{<t}$ represents all previous tokens in the sequence. This setup encourages the model to accurately predict each subsequent token based on the preceding context, which is crucial for tasks like text generation and machine translation.

GPT-2 is trained on a diverse internet dataset using unsupervised learning. It learns to predict the next word in a sentence without needing aligned input-output pairs, which differs from supervised learning models that require specific targets for each input during training. Each layer in GPT-2 computes self-attention, where it weighs the significance of all previous tokens for each token in the sequence. This mechanism allows GPT-2 to generate coherent and contextually relevant text based on the input it has processed. GPT-2 uses Byte Pair Encoding, a form of tokenization that splits text into common subwords or sequences of characters. This method allows the model to efficiently handle a large vocabulary without the explosion in size and manages to capture the morphology of complex words.

C. Model Input Preparation

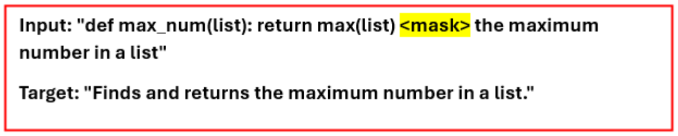
To accommodate the specific needs of each transformer model, I prepared the inputs in three distinct formats: For CodeT5, inputs were straightforward code-summary pairs.



```
Input: "def add(a, b):\n   return a + b"

Target: "Adds two numbers and returns the sum."
```

Fig. 3: CodeT5 input format

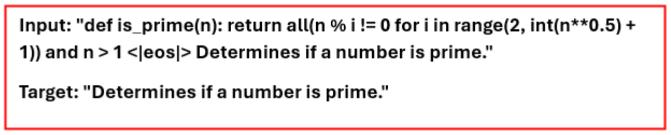
For CodeBERT, we used code concatenated with a partially masked summary to facilitate masked token prediction.



```
Input: "def max_num(list): return max(list) <mask> the maximum number in a list"

Target: "Finds and returns the maximum number in a list."
```

Fig. 4: CodeBert input format

For GPT-2, inputs consisted of code followed by an ”eos” token and the summary, training the model to treat the summary as a continuation of the code.



```
Input: "def is_prime(n): return all(n % i != 0 for i in range(2, int(n**0.5) + 1)) and n > 1 <|eos|> Determines if a number is prime."

Target: "Determines if a number is prime."
```

Fig. 5: GP2 input format

This input configuration is tailored specifically for GPT-2 to utilize its text generation capabilities effectively by providing it with a context (the code) and the expected continuation (the summary) in one input sequence.

D. Model training

In My research, I streamlined the training process to address computational constraints effectively by reducing our dataset from 37,000 to 8,000 data points. This decision was essential to manage memory limitations within our hardware capabilities.

During model training, I employed TrainingArguments with a focus on optimizing the use of our resources. I implemented a learning rate of 5e-5 and a small batch size of 1, supplemented by 4 gradient accumulation steps. This setup mimicked the effects of larger batch sizes, allowing for efficient learning without exceeding memory limits. Additionally, I enabled mixed precision training (fp16=True) to halve the memory usage, which was crucial for accommodating the larger model architecture and longer sequence lengths.

For tasks involving GPT-2 architecture, I opted for DistilGPT-2 due to its efficiency and reduced memory requirements. DistilGPT-2 was fine-tuned in about three hours, significantly faster than what would have been required for the full GPT-2 model under similar constraints. The fine-tuning process was carefully monitored using the 'bleu' metric to ensure linguistic quality in the generated summaries.

E. Evaluation Protocol

I utilized an epoch-wise evaluation strategy to optimize and monitor the training, limiting the number of saved model iterations by the best-performing snapshot. This method helped manage disk space efficiently while allowing for detailed performance tracking.

The adjustments made during the training phase highlight the practical challenges and trade-offs faced when training large models with limited resources. These modifications not only ensured efficient use of computational resources but also maintained satisfactory model performance, demonstrating that effective machine learning outcomes are achievable even under constrained conditions.

IV. Dataset

Dataset Collection

Repository Selection Initially, the SEART GHS tool was employed to curate a diverse set of Python repositories. The selection criteria included repositories with at least 500 stars, 500 commits, 500 issues, and 50 watchers, ensuring a collection of high-quality and active repositories. This process resulted in a collection of 3000 Python repositories, from which 100 repositories were chosen at random for deeper analysis.

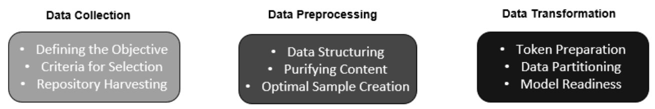


Fig. 6: Dataset Preprocessing overview

Repository Cloning and Data Extraction Process A CSV file containing key details about each project was created to organize this large dataset. Each link was verified to ensure they were direct Git links, critical for the accuracy of the dataset and shallow cloning. Selected repositories were cloned using shallow cloning to manage data volume efficiently. Python files within these repositories were specifically targeted. Data extraction was implemented using an abstract syntax tree function in Update_method_extractor.py. The collected data encompassed various attributes such as project name, method start and end lines, the method code itself, its documentation, and a combined field of method with documentation. This meticulously structured data was then compiled into a CSV file and converted into a JSON file named Dataset_methods.json, containing 37,000 Python method codes.

Before tokenization, Python's tokenizer module (RegexpTokenizer) was used to extract individual tokens from the method code. I have maintained 4 data preparation criteria :

1. Removed examples that codes cannot be parsed into an abstract syntax tree.

2. Removed examples that tokens of documents is

$$< 3 or > 256$$

3. Removed examples that documents contain special tokens (e.g.

$$< img... >$$

or

$$https:$$

...)

4. Removed examples that documents are not English.

This preprocessing step ensured that the data was clean and ready for training the code summarization models.[16]

V. Evaluation

In the evaluation section of this study on code summarization using transformer models, I specifically assess the performance using both Exact Match (EM) and BLEU scores as key metrics. While traditionally Exact Match might not fully encapsulate the strengths of a generative model like GPT-2 (DistilGPT-2), it provides a straightforward, binary indication of whether the generated summary is an exact replica of the reference text. This metric is calculated simply by comparing the generated text to the reference text; a score of 1 is awarded if they match perfectly, and a score of 0 is given otherwise. Its binary nature means it lacks the nuance to recognize semantic equivalences or paraphrases, which are common in natural language generation.

On the other hand, the BLEU (Bilingual Evaluation Understudy) score, originally designed for evaluating machine-translated text against human translations, has been adapted to assess the quality of text generated by models in tasks like summarization. BLEU evaluates the correspondence of n-grams between the machine-generated text and a set of reference texts. It calculates precision scores for n-grams of different lengths and combines them into an overall score using a geometric mean, typically also applying a brevity penalty to discourage overly short responses. This metric provides a more nuanced assessment than Exact Match, as it considers the presence of key phrases and their order in the generated text, thereby reflecting both accuracy and fluency.

$$\text{BLEU} = b(p) \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{7}$$

where:

- p_n is the precision of n-grams,
- w_n are weights for each n-gram size, typically set equally,

- $b(p)$ is the brevity penalty, calculated as:

$$b(p) = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

- c is the length of the candidate translation,
- r is the effective reference corpus length.

However, BLEU also has limitations. It focuses primarily on the presence and frequency of n-grams, which means it can overlook semantic nuances that do not alter n-gram frequencies but change the implied meaning. Moreover, it does not account for the syntactic or grammatical correctness of the generated text unless those errors specifically disrupt the n-grams it measures.

In my evaluation, the use of Exact Match alongside BLEU allows me to combine a strict measure of textual fidelity with an assessment of linguistic quality, offering a more comprehensive overview of model performance. By incorporating these metrics, I aim to balance the evaluation of exact textual reproduction with the ability to generate coherent, contextually appropriate summaries. This approach acknowledges the inherent limitations of each metric while leveraging their combined strengths to provide a detailed analysis of how well each model performs in practical code summarization applications.

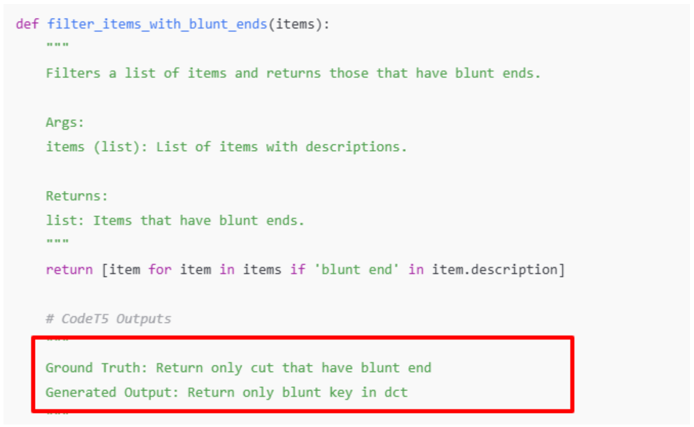
A. Result

In our comparative study of code summarization models, CodeT5 outperformed both CodeBERT and GPT-2 significantly, achieving the highest accuracy and BLEU scores at 14.286% and 27.30%, respectively. This demonstrates its superior capability in precisely capturing and reproducing the complexities of Python code. Meanwhile, CodeBERT, with a BLEU score of 11.17%, showed moderate effectiveness in generating linguistically coherent summaries, though its exact match rate was lower at 7.025%. GPT-2, optimized for generative fluency rather than precision, lagged in both metrics with the lowest scores, emphasizing its creative output over direct accuracy. This variance underscores the critical importance of model selection based on the specific needs of code summarization tasks

TABLE I: Performance of Different Models on Code Summarization

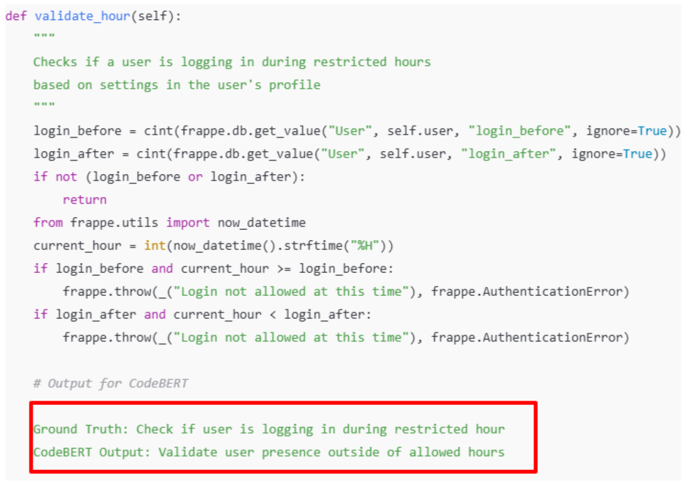
Proposed Model	Accuracy (Exact Match)	BLEU Score
CodeT5	14.286%	27.30%
CodeBERT	7.025%	11.17%
GPT-2	3.009%	6.02%

CodeT5 vs. CodeBERT Performance CodeT5 outperformed CodeBERT in the code summarization task, which can be attributed to its sequential, auto-regressive nature as a sequence-to-sequence model. Unlike CodeBERT, which processes inputs bidirectionally to understand context, CodeT5 generates each token based on the sequence of previously generated tokens. This auto-regressive behavior allows CodeT5 to maintain a focused, coherent narrative in summarization tasks, building each subsequent token in a way that is contextually dependent on the tokens before it. This methodology is particularly beneficial in code summarization, where the logical flow and technical accuracy are paramount. In contrast, CodeBERT's bidirectional model offers a broad contextual understanding, which, while comprehensive, may not align precisely with tasks that require a sequential build-up of logic, such as summarizing code into a cohesive narrative.



```
def filter_items_with_blunt_ends(items):
    """
    Filters a list of items and returns those that have blunt ends.

    Args:
    items (list): List of items with descriptions.

    Returns:
    list: Items that have blunt ends.
    """
    return [item for item in items if 'blunt end' in item.description]

    # CodeT5 Outputs
    """
    Ground Truth: Return only cut that have blunt end
    Generated Output: Return only blunt key in dct
    """
```

Fig. 7: CodeT5 Output Illustration

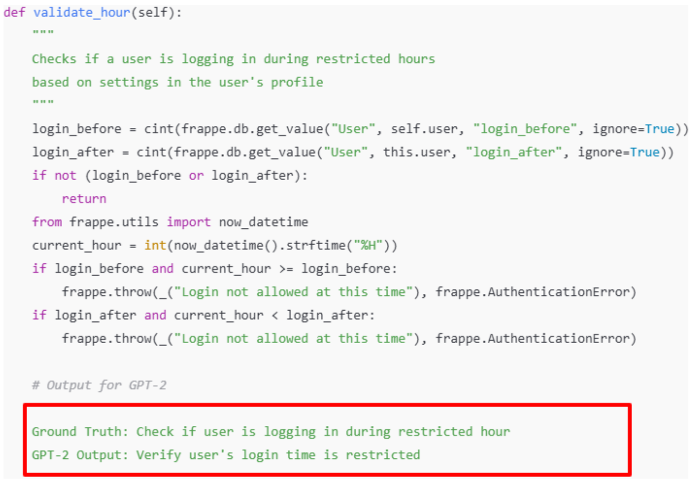


```
def validate_hour(self):
    """
    Checks if a user is logging in during restricted hours
    based on settings in the user's profile
    """
    login_before = cint(frappe.db.get_value("User", self.user, "login_before", ignore=True))
    login_after = cint(frappe.db.get_value("User", self.user, "login_after", ignore=True))
    if not (login_before or login_after):
        return
    from frappe.utils import now_datetime
    current_hour = int(now_datetime().strftime("%H"))
    if login_before and current_hour >= login_before:
        frappe.throw(_("Login not allowed at this time"), frappe.AuthenticationError)
    if login_after and current_hour < login_after:
        frappe.throw(_("Login not allowed at this time"), frappe.AuthenticationError)

    # Output for CodeBERT

Ground Truth: Check if user is logging in during restricted hour
CodeBERT Output: Validate user presence outside of allowed hours
```

Fig. 8: CodeBert Output Illustration

GPT-2's Performance in Code Summarization:

GPT-2, primarily trained on a diverse natural language corpus, is geared towards generating general-purpose text rather than handling the specific structural and logical nuances of programming languages, which differ significantly from everyday language in function and complexity.



```
def validate_hour(self):
    """
    Checks if a user is logging in during restricted hours
    based on settings in the user's profile
    """
    login_before = cint(frappe.db.get_value("User", self.user, "login_before", ignore=True))
    login_after = cint(frappe.db.get_value("User", this.user, "login_after", ignore=True))
    if not (login_before or login_after):
        return
    from frappe.utils import now_datetime
    current_hour = int(now_datetime().strftime("%H"))
    if login_before and current_hour >= login_before:
        frappe.throw(_("Login not allowed at this time"), frappe.AuthenticationError)
    if login_after and current_hour < login_after:
        frappe.throw(_("Login not allowed at this time"), frappe.AuthenticationError)

    # Output for GPT-2

    Ground Truth: Check if user is logging in during restricted hour
    GPT-2 Output: Verify user's login time is restricted
```

Fig. 9: GP2 Output Illustration

When fine-tuned on a dataset of Python code, GPT-2 struggled to adapt due to its pre-training on natural language, which values fluency over the precision required for code summarization. Without initial training on programming constructs, GPT-2 lacks the necessary background to accurately capture and express the detailed semantics of code, resulting in summaries that fail to reflect the precise operations and logic inherent in the source code. This fundamental mismatch in training focus led to its underperformance in generating contextually accurate code summaries.

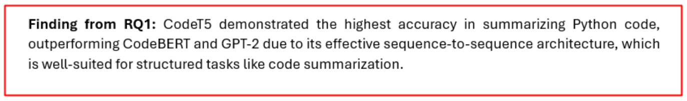
VI. Discussion

RQ1: How do different code summarization models (CodeT5, CodeBERT, and GPT-2) compare in terms of accuracy and efficiency when applied to a custom dataset of Python repositories?

The results indicate a clear hierarchy in performance among the models tested. CodeT5 demonstrated superior accuracy with the highest Exact Match and BLEU scores, reinforcing the efficacy of its sequence-to-sequence, auto-regressive architecture for tasks requiring precise adherence to source text. This model's design, which builds outputs sequentially and relies on context from previously generated tokens, appears particularly suited to the structured nature of code, where each line may depend closely on preceding lines.

CodeBERT, while not matching the textual fidelity of CodeT5, still performed moderately well, especially in terms of BLEU score. Its bidirectional model, which evaluates text in both forward and backward directions, offers a comprehensive understanding of context but may not be as effective in tasks where the sequence of generation impacts the outcome, such as in code summarization where following the logical order strictly is crucial.

GPT-2 showed the lowest scores in both metrics, which could be attributed to its foundational training predominantly on natural language text rather than code. The model's generative capabilities, while impressive in broad linguistic contexts, may not align well with the stringent requirements of summarizing syntax-heavy and logically complex programming languages without extensive retraining or fine-tuning on relevant technical data.

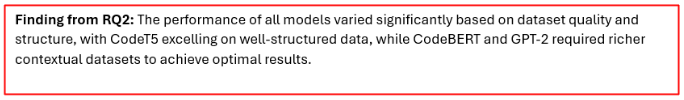


Finding from RQ1: CodeT5 demonstrated the highest accuracy in summarizing Python code, outperforming CodeBERT and GPT-2 due to its effective sequence-to-sequence architecture, which is well-suited for structured tasks like code summarization.

Fig. 10: Findings 1

RQ2: What impact does the quality and structure of the dataset have on the performance of code summarization models?

The differential performance across the models also hints at the sensitivity of these models to the quality and structure of the dataset. CodeT5's robust performance across both metrics suggests it can leverage well-structured and consistent datasets effectively, possibly due to its ability to learn from and adapt to clear patterns and dependencies in data. In contrast, CodeBERT and GPT-2 might require datasets that offer a richer context or more varied examples to truly excel, indicating a potential area for further research in data preparation and enhancement.



Finding from RQ2: The performance of all models varied significantly based on dataset quality and structure, with CodeT5 excelling on well-structured data, while CodeBERT and GPT-2 required richer contextual datasets to achieve optimal results.

Fig. 11: Findings 2

The varying results highlight the critical importance of dataset design in training models for code summarization. A well-curated dataset, tailored specifically to mirror the eventual application environment of the model, could significantly impact the effectiveness of the model. This points to the need for ongoing efforts in dataset optimization and suggests that future advancements in model training might focus as much on improving how data is presented to these models as on the models themselves.

VII. Threats to Validity

The validity of this research is primarily challenged by two major factors: the quality of the dataset and the resource constraints during model training. Firstly, the dependency on dataset quality and the specificity of model training pose significant threats. The models' performance, particularly that of CodeBERT and GPT-2, might not generalize well across different types of code repositories or programming languages if the training data does not adequately represent the target environments. The ground truths in the dataset, which inform accuracy and efficiency measurements, might not always fully capture the complex context or nuances of the codesnippets. Secondly, the experimental setup itself—limited by memory management and computational efficiency—constrained the potential of the models. The use of just 10 epochs and small batch sizes, coupled with a restricted amount of training data, was primarily driven by the need to manage computational resources effectively. This setup especially affected GPT-2, which typically requires more extensive training on larger datasets to fine-tune its performance for specialized tasks like code summarization.

VIII. Conclusion

This study underscores the critical roles of model architecture, dataset quality, and resource management in the task of code summarization. CodeT5's superior performance highlights the importance of selecting a model whose capabilities are closely aligned with the specific requirements of the task. The varied performances of CodeBERT and GPT-2 illuminate the need for not only rich and contextually varied datasets but also sufficient computational resources to train models effectively. These findings suggest that future research should focus on enhancing dataset designs and exploring how different model architectures can be optimized or adapted, with appropriate resource allocations, to improve code summarization outcomes. This dual focus on data quality and resource optimization will pave the way for more robust and universally applicable solutions in the field of automated code documentation.

References

[1] R. D. Darshan, I. Surya, and G. Malarselvi, "English-language abstract text summarization using the t5 model," in *AIP Conference Proceedings*, vol. 3075, no. 1. AIP Publishing, 2024.

[2] R. Haldar and J. Hockenmaier, "Analyzing the performance of large language models on code summarization," *arXiv preprint arXiv:2404.08018*, 2024.

[3] J. Zhu, Y. Miao, T. Xu, J. Zhu, and X. Sun, "On the effectiveness of large language models in statement-level code summarization," in *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2024, pp. 216–227.

[4] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[5] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working conference on reverse engineering*. IEEE, 2010, pp. 35–44.

[6] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *54th Annual Meeting of the Association for Computational Linguistics 2016*. Association for Computational Linguistics, 2016, pp. 2073–2083.

[7] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[8] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[9] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[10] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," 2018.

[11] S. Gao, C. Gao, Y. He, J. Zeng, L. Nie, X. Xia, and M. Lyu, "Code structure–guided transformer for source code summarization," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–32, 2023.

[12] OpenAI, "Chatgpt-4: Advanced language models by openai," https://openai.com/chatgpt-4, 2023, accessed: insert-date-of-access.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[14] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," *Applied Sciences*, vol. 11, no. 11, p. 4793, 2021.

[15] E. T. R. Schneider, J. V. A. de Souza, Y. B. Gumiel, C. Moro, and E. C. Paraiso, "A gpt-2 language model for biomedical texts in portuguese," in *2021 IEEE 34th international symposium on computer-based medical systems (CBMS)*. IEEE, 2021, pp. 474–479.

[16] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.