





# AsmDocGen: Generating Functional Natural Language Descriptions for Assembly Code

Jesia Quader Yuki<sup>1</sup>, Mohammadhossein Amouei<sup>1</sup> <sup>a</sup>, Benjamin C. M. Fung<sup>1</sup> <sup>b</sup>,  
Philippe Charland<sup>2</sup> <sup>c</sup> and Andrew Walenstein<sup>3</sup> <sup>d</sup>

<sup>1</sup>*School of Information Studies, McGill University, Montreal, QC, Canada*

<sup>2</sup>*Mission Critical Cyber Security Section, Defence R&D Canada, Quebec, QC, Canada*

<sup>3</sup>*BlackBerry Limited, Waterloo, ON, Canada*

{jesia.yuki, mohammadhossein.amouei}@mail.mcgill.ca, ben.fung@mcgill.ca, philippe.charland@drcd-rddc.gc.ca,  
awalenstein@blackberry.com

**Keywords:** Assembly Code, Reverse Engineering, CodeBERT, Transformers, Code Summarization.

**Abstract:** This study explores the field of software reverse engineering through the lens of code summarization, which involves generating informative and concise summaries of code functionality. A significant aspect of this research is the application of assembly code summarization in malware analysis, highlighting its critical role in understanding and mitigating potential security threats. Although there have been recent efforts to develop code summarization techniques for high-level programming languages, to the best of our knowledge, this study is the first attempt to generate comments for assembly code. For this purpose, we first built a carefully curated dataset of assembly function-comment pairs. We then focused on automatic assembly code summarization using transfer learning with pre-trained natural language processing (NLP) models, including BERT, DistilBERT, RoBERTa, and CodeBERT. The results of our experiments show a notable advantage of CodeBERT: despite its initial training on high-level programming languages alone, it excels in learning assembly language, outperforming other pre-trained NLP models.


## 1 INTRODUCTION


Reverse engineering refers to the process of analyzing an existing software system to recover its design: to understand its functionality, design, and implementation details. This technique involves examining software code, system behavior, and dependencies to create a representation of the architecture and functionality of the system. It is often used to update or improve existing systems, create documentation, or build new software applications compatible with the original system. Reverse engineering can also help detect malicious software or potential vulnerabilities.


Not all reverse engineering efforts have the luxury of starting with source code, as software source code may not always be available to reverse engineers for several reasons, such as the code being proprietary or protected by copyright laws. In addition, the code


may be lost or corrupted, making it impossible to analyze it directly. However, when starting with just the executable, we have effective techniques to recover an assembly-language representation, so understanding assembly code is a common task in reverse engineering. But understanding assembly code is more complicated than understanding high-level programming languages for several reasons. Assembly code is a low-level language consisting of complex instructions that can be challenging to read and understand. It typically lacks the abstraction and structure found in high-level programming languages, making it harder to identify program flow and comprehend the overall purpose of the code. As a result of the unavailability of source code and the complexity of assembly code, there is a greater need for automated assistance to the reverse engineer trying to understand assembly code.

A valuable tool for reverse engineering is code summarization. It is also known as “code commenting” and concerns generating a concise and informative summary of a software code’s functionality or behavior. Code summarization techniques use natural language processing and machine learning algo-

<sup>a</sup>  <https://orcid.org/0000-0003-4208-4014>

<sup>b</sup>  <https://orcid.org/0000-0001-8423-2906>

<sup>c</sup>  <https://orcid.org/0000-0003-4051-9942>

<sup>d</sup>  <https://orcid.org/0000-0003-1103-2465>

rithms to analyze the syntax, structure, and comments of the code to generate a human-readable and easy-to-understand summary (Steidl et al., 2013). The generated summary can provide a rapidly reviewed proposal of the code’s likely functionality, helping the reverse engineer identify potential flaws and more rapidly understand the functionality and behavior of a large and complex software codebase (Tenny, 1988; Woodfield et al., 1981). Several prior works have indicated effective advances in code summarization, including (Allamanis et al., 2016; LeClair et al., 2019; Ahmad et al., 2020; Kusupati and Ailavarapu, 2022; Wang et al., 2020).

However, little appears to be known about utilizing such summarization techniques on assembly language, specifically: about how well transformer technologies work, whether ML models trained on non-assembly corpora can effectively be leveraged through transfer learning, and what qualities are required of training corpora to yield effective learning performance. We aimed to explore how pre-trained NLP models can be trained for the specific task of summarizing assembly code, leveraging their existing knowledge and experience with natural language processing tasks. By comparing the performance of different pre-trained NLP models, we aimed to identify which models are most effective for transfer learning in this context, providing insight into how best to utilize pre-trained models for this specific task.

The contributions of this paper relating to these research problems are as follows:

- We propose AsmDocGen, a CodeBERT-based solution that generates human-readable comments for assembly functions. AsmDocGen makes assembly code easier to understand by automatically creating clear comments for it. This is an important step forward in making complex code more accessible and easier to work with, especially in areas like software reverse engineering. Our experiments show that the comments generated are accurate and useful. This progress is crucial to improving the way we handle and document lower-level programming languages. To the best of our knowledge, we are the first to report code summarization techniques applied to assembly code, so the performance results serve as a baseline for future work in the area.
- We provide a well-curated dataset of 5,084 assembly function comment pairs for training and validating assembly code summarization solutions. As subject matter experts, we handpicked and manually edited assembly function comment pairs to create our dataset. The resulting dataset provides a valuable resource for future research on

assembly code summarization.

- We provide evidence that NLP models pre-trained on other corpora can be successfully retrained and tuned to be applied to assembly code. Specifically, we show that CodeBERT can be re-trained to understand assembly language and generate concise comments, describing the functionality of a piece of assembly code by leveraging its knowledge of high-level programming languages.

The rest of this paper is organized as follows. Section 2 briefly reviews the literature on NLP and code summarization. In Section 3, we explain our data collection strategy and dataset. Section 4 details our proposed method. In Sections 5 and 6, we outline the experiments and results we achieved. Finally, Section 7 concludes the article.

## 2 RELATIONS TO PRIOR WORK

AsmDocGen adapts and extends previous work in the so-called ‘transformer’ architectures for NLP. Transformer based approaches have been shown to be effective in a wide range of NLP tasks, such as text summarization and chatbots (Vaswani et al., 2017; Brown et al., 2020; Devlin et al., 2019). Recent advances, including novel pretraining techniques (Lewis et al., 2020; Radford et al., 2018), have resulted in a state-of-the-art where transformer-based solutions have shown to match or exceed previous language modeling techniques for many NLP tasks, including but not limited to text summarization, translation, dialogue generation, and code summarization.

### 2.1 Text Summarization

Sarkar (Sarkar, 2013) proposed a method to summarize a document by extracting its main concepts. This approach aims to provide an overall understanding of the content of a document. Similarly, Christian et al. (Christian et al., 2016) created an automatic text summarizer using the TF-IDF algorithm. The TF-IDF algorithm measures the importance of each word in a document, and the words with the highest TF-IDF scores are used to construct the summary. Verma and Om (Verma and Om, 2018) presented a novel technique for extracting multi-document summaries based on Shark Smell Optimization (SSO). The approach is based on graph-based optimization and aims to produce concise and coherent summaries. Liu and Lapata (Liu and Lapata, 2019) applied BERT to the summarization task and showed that it can perform well for both extractive and abstractive summarization tasks. Gupta et al. (Gupta et al., 2022) applied

a few pre-trained models such as BERT, GPT, and RoBERTa for text summarization. Our work is akin to text summarization in the sense that we generate text that purposefully omits details to offer a concise representation of important elements in the original work; however, it differs from text summarization in that the language and even language type of the summarization (natural language, English say) is different from the source language (assembly).

## 2.2 Translation

The use of transformer models has been widely adopted in neural machine translation. Researchers have applied these models for the translation of English into French, as demonstrated by Sutskever et al. and Cho et al. (Sutskever et al., 2014; Cho et al., 2014). Bahdanau et al. (Bahdanau et al., 2014) expanded the basic encoder-decoder for the English-to-French translation task. Furthermore, Gao et al. (Gao et al., 2021) introduced a Scalable Transformers model and demonstrated its effectiveness in translating English to German and English to French. Like such prior work, AsmDocGen translates semantic content from one language to another, but unlike translation, where faithful representation of the content is expected, we purposefully reduce the content of the output compared to the input for the purposes of summarization.

## 2.3 Dialogue Generation and Grammar

Olabiyi and Mueller (Olabiyi et al., 2020) presented DLGNet, a transformer-based model for dialogue modeling. Lee et al. (Lee et al., 2021) used the Transformer with Copying Mechanism that outperformed two commercial grammar checks and other NMT-based models. Cao et al. (Cao et al., 2020) investigated dialogue models with numerous input sources modified from the pretrained language model GPT2. They evaluated several strategies for fusing multiple sources of attention information. Their experimental results reveal that correct fusion procedures outperform simple fusion baselines in terms of relevance with the dialogue history.

## 2.4 Code Summarization

Iyer et al. (Iyer et al., 2016) presented CODE-NN, a novel method that uses LSTM and an attention procedure to generate summaries of C# code snippets and SQL queries. Hu et al. (Hu et al., 2018b) presented DeepCom, a model that examines the structural information of Java methods to generate better comments.

Furthermore, Hu et al. (Hu et al., 2018a) developed TL-CodeSum, an RNN-based model that effectively used API knowledge in conjunction with source code to generate code summarization.

Allamanis et al. (Allamanis et al., 2016) presented a unique convolutional self-attention network to perform extreme summarization based on source code where “extreme” denotes the production of extremely brief messages. LeClair et al. (LeClair et al., 2019) use an attentional GRU encoder-decoder model to produce summaries for code. Ahmad et al. (Ahmad et al., 2020) showed that relative encoding significantly enhances summarization efficiency by using transformers to generate a comprehensible summary that represents the functionality of a program. Similarly, Kusupati and Ailavarapu (Kusupati and Ailavarapu, 2022) used transformers for code summarization. PYMT5, the PYTHON method text-to-text transfer transformer, is presented by Clement et al. (Clement et al., 2020). This model has the capability to predict complete methods based on natural language documentation strings (docstrings), and it can also condense code into docstrings of various conventional styles.

Some advanced pre-trained language models, such as BERT (Devlin et al., 2019), XLNet (Yang et al., 2019), GPT (Brown et al., 2020), RoBERTa (Liu and Lapata, 2019), and CodeBERT (Feng et al., 2020), have seemed promising for pairing comments with code (Liu and Lapata, 2019; Husain et al., ). This has inspired different researchers to outperform those state-of-the-art models by employing pre-trained language models in the task of code summarization (Barone and Sennrich, 2021; Wang et al., 2020) or natural language-based code search (Gu et al., 2018).

Our study is distinctive in its focus on generating comments that summarize the functionality of assembly code, which has received little attention in previous research on code summarization. Assembly code can be particularly challenging to comprehend, due to its high complexity and lack of high-level abstraction.

## 2.5 Reference Code Corpora

One of our contributions is a reference code corpus for the purposes of training or validating software engineering tools, such as the CoNaLa (Yin et al., 2018) and Bellon’s clone detector corpus (Bellon et al., 2007). Our corpus<sup>1</sup>, offered with a license of permissive use, is expected to be a valuable result for further studies in the area.

---

<sup>1</sup><https://github.com/McGill-DMaS/AsmDocGen>

Table 1: Numbers of functions taken from various online platforms.

Platforms	No. of functions
Github	2428
HackerRank	1505
StackOverflow	574
Codeforces	423
Codechef	154

### 3 DATASET

The absence of a comprehensive and well-curated dataset has made it difficult to train machine learning and deep learning models for assembly code summarization. To overcome this obstacle, we have carefully curated a dataset of assembly function comment pairs.

To create our dataset, we curated a selection of assembly code functions and their corresponding comments. Initially, we collected a diverse range of C and C++ source code samples from well-known online sources such as GitHub, Codeforces, StackOverflow, HackerRank, and Codechef.

After collecting the source code samples, we manually examined each file to ensure correct matching between functions and their corresponding comments. Once we finished curating the source code comment pairs, we compiled the source code using the GCC compiler. Then, we used IDA Pro to disassemble the resulting executables. Finally, we manually correlated assembly functions with their corresponding source code functions to identify the original source comments that match the disassembled functions.

During the matching process, we found many unexplained functions within the source code, and among those with comments, we observed that many comments suffer from poor quality. Thus, we manually eliminated low-quality comments. The characteristics of those comments classified as poor quality are:

- Stating the obvious without providing additional insights or details about the functionality.
- Being obsolete and not corresponding to the current version of the code.
- Being overly wordy or containing excessively technical language.
- Having grammatical mistakes.
- Containing irrelevant information.
- Being unclear or ambiguous
- Being misleading or incorrect

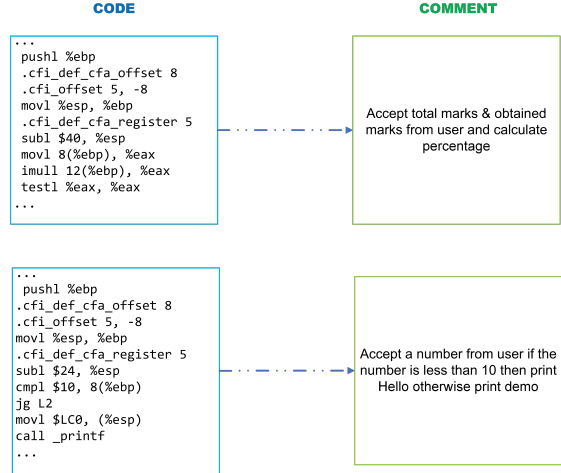


Figure 1: Snippet of the dataset.

Overall, the process of building a high-quality dataset involved four months of work with meticulous attention to detail. As a result of our effort, we collected 5,084 assembly function comment pairs (see Table 1); Figure 1 shows an excerpt from our dataset.

### 4 THE METHOD: AsmDocGen

Given the limited size of our assembly language comment dataset, it is not feasible to train large NLP models from scratch. Further, it is not clear that constructing an extensive-enough dataset of assembly-comment pairs that alone can be used to train a highly-performant comment generator is required. Instead, we hypothesize that transfer learning by using a suitable foundation NLP model and retraining on our targeted data set can yield significantly improved performance than using non-assembly-based foundation models directly or trying to train an NLP model from scratch using such a limited data set as ours.

We chose CodeBERT (Feng et al., 2020) as our foundation to build AsmDocGen. CodeBERT is a state-of-the-art pretrained language model based on the Transformer architecture. It has been trained on a vast corpus of English words and source code, including Python, Java, and C++, making it one of the most versatile pre-trained models available and making it adept at understanding code structure and function. CodeBERT’s flexibility stems from its ability to capture both syntactic and semantic information from natural language and programming language inputs. Therefore, we hypothesize that CodeBERT’s ability to capture semantic representations for natural and programming languages is advantageous for understanding assembly code’s unique grammar for as-

sembly code summarization, making it a more effective choice for training on small datasets than training models from scratch. Furthermore, the research conducted by Zhou et al. (Zhou and Su, 2002) has confirmed the ability of CodeBERT to adapt beyond its pre-trained data. This ability makes CodeBERT a more robust and adaptable model, enabling it to perform well on new tasks and domains.

We expect that the semantic representations learned by CodeBERT are particularly beneficial for understanding assembly code. Assembly code has a unique grammar for constructing instructions and linking operations and operands. CodeBERT’s ability to capture these structures and their relationships makes it an effective tool for generating accurate comment representations.

#### 4.0.1 Model Background

CodeBERT is based on the transformer architecture, similar to the original BERT model. It has a multi-layer transformer encoder, which takes as input code and comments. To tokenize the input, CodeBERT uses the WordPiece tokenization method for both code and comments (Wu et al., 2016). The input to the model consists of a sequence of tokens, which are the individual words and symbols of the code and comments.

CodeBERT’s training objectives include Masked Language Modeling (MLM) and Replaced Token Detection (RTD). MLM involves masking parts of the text at random and requesting the model to predict them, while RTD involves replacing tokens in the text with plausible alternatives and having the model determine which tokens have been replaced. This approach improves the robustness of the model by allowing it to handle variations in input data.

In addition to MLM and RTD, CodeBERT includes a Cross-Lingual Language Model (XLM) objective, which enables it to learn cross-lingual representations by jointly training on monolingual and parallel data. This feature makes CodeBERT particularly useful for natural language processing tasks involving multilingual inputs.

In the pre-training phase, the input is set as the concatenation of two segments with special separator tokens. The input format is  $[CLS], w_1, w_2, \dots, w_n, [SEP], c_1, c_2, \dots, c_m, [EOS]$ , where  $[CLS]$  is added at the beginning of each sentence to capture the sentence representation. The  $[SEP]$  token separates the code and comment tokens, making it easier for BERT to understand that the input is made up of two parts, code and comments. The  $[EOS]$  token is used to indicate the end of a sentence.

The input tokens are passed through an embed-

ding layer, where they are transformed into numerical vectors that capture the meaning of the tokens. These tokens are then combined with three other types of embeddings to form a single input vector for the model. The three types of embedding include segmentation embeddings, position embedding, and token embeddings. Figure 2 demonstrates how the tokens are passed through various embedding layers to form a single input vector. The role of these three types of embedding involves:

- **Segmentation** embeddings are used to differentiate between distinct lines of code.
- **Position** embeddings show the position of each token inside the line of code.
- **Token** embeddings refer to the semantics of each token.



Figure 2: Illustration of the multi-layered process used by the CodeBERT model to transform individual tokens into embeddings via various layers.

CodeBERT is pre-trained on a dataset of code and comment pairs using masked language modelling and replaced token detection objectives. The training process involves two neural networks, a generator G and a discriminator D. The encoder of both networks, usually a transformer network, converts a sequence of embedding tokens  $x = [x_1, \dots, x_n]$  into a sequence of contextualized vector representations  $h(x) = [h_1, \dots, h_n]$ . These embeddings are then passed through the transformer encoder, which consists of multiple layers of self-attention and feed-forward neural networks. The self-attention layers enable the model to focus on distinct segments of the input sequence, while the feed-forward layers help the model acquire a deeper understanding of the relationships between the individual tokens.

The final output of the transformer encoder is a set of embeddings for each token in the input sequence, which captures the meaning of the code and comments in a fixed-length vector representation. These embeddings are then used to train the model for code summarization tasks.

In general, the overall structure of CodeBERT for code summarization is an encoder-decoder structure, where the encoder is the transformer-based neural network, and the decoder is the task-specific network that generates the summary of the code based

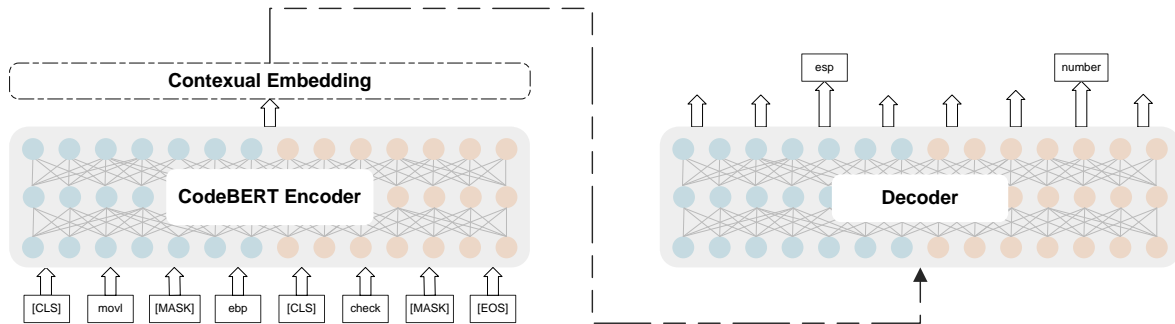


Figure 3: Overview of training AsmDocGen. During the test, the comment is removed from the input and appears on the model’s output.

on the encoded representation. We train this encoder-decoder model using our dataset, which consists of assembly functions and their corresponding ground-truth comments.

#### 4.1 Training AsmDocGen

**Dataset Construction.** We collected a comprehensive dataset of assembly code functions according to the description in Section 3. We then tokenized the dataset using the WordPiece algorithm, which is well suited for tokenizing both natural language and code (Gu et al., 2018). Assembly language has its own unique syntax and vocabulary, and we find that WordPiece is the optimal choice for our problem. The algorithm enables the encoding of any unusual words in the lexicon with suitable subword tokens, without requiring the addition of any “unknown” tokens.

To maintain consistency in sentence length, we utilize padding and truncation techniques to generate sentences of a uniform length of 100 tokens. We determined this length by conducting experiments with varying sentence lengths and evaluating the trade-off between the model’s accuracy and training time.

**Training.** To generate code summaries, a sequence-to-sequence pipeline was utilized, where the encoder was initialized using CodeBERT, which comprises 12 layers, a maximum length of 512, an embedding size of 768, and 12 attention heads of size 64. For the decoder, a randomly initialized Transformer with 6 layers, hidden states of 768 dimensions, and 12 attention heads were used.

To update the model parameters, Adam Optimizer was used with a learning rate of  $5e-5$  and a batch size of 32. The hyperparameters were optimized and early stopping was applied based on the validation dataset. We partitioned the dataset into training, validation, and test sets with proportions of 70%, 15%, and 15%, respectively.

**Inference.** As illustrated in Figure 3, we pass the tokenized words through the CodeBERT encoder to

produce fixed-length embeddings. These embeddings capture the meaning of the code and comments, creating a contextual vector representation for each token in the input sequence. This contextual vector is finally passed to a decoder to generate a code summary.

## 5 EXPERIMENTAL SETUP

To test our transfer learning hypothesis and evaluate AsmDocGen’s performance, we train a sample of NLP models to compare their results with AsmDocGen. The CPU architecture of the code is x86/x64.

### 5.1 Metrics

We use four widely accepted metrics within this field to compare the performance of AsmDocGen against the experimental sample of foundation models that have not been trained for the assembly language commenting task. These metrics provide a quantifiable means of determining the success of the model in achieving its objectives.

#### 5.1.1 BLEU

BLEU (Bilingual Evaluation Understudy) is a metric used to evaluate the quality of machine-generated text, such as machine translation or text summarization. It measures the similarity between the generated text and the reference text (usually human-generated) based on n-gram overlap. The higher the BLEU score, the better the machine-generated text matches the reference text. BLEU score ranges from 0 to 1, where a score of 1 indicates that the machine-generated text is identical to the reference text. BLEU score is commonly used in natural language processing research as a standard metric to evaluate the quality of machine-generated text.



### 5.1.2 ROUGE-1

ROUGE-1 is a metric used to evaluate the quality of text summarization or machine translation. It measures the overlap between word unigrams in the reference summary (or translation) and the generated summary (or translation). The ROUGE-1 score ranges from 0 to 1, where 1 indicates a perfect overlap between the reference summary and the generated summary in terms of unigrams.

### 5.1.3 ROUGE-2

ROUGE-2 is an evaluation metric used for automatic summarization tasks that measures the overlap of word bigrams between the system-generated summary and the reference summary. It is similar to ROUGE-1, but instead of considering individual words, it looks at pairs of words in the summary and reference summary. The score is calculated as the number of overlapping bigrams divided by the total number of bigrams in the reference summary.

### 5.1.4 ROUGE-L

ROUGE-L is a metric to evaluate the quality of text summaries or translations. It stands for Recall-Oriented Understudy for Gisting Evaluation - Longest Common Subsequence, and it measures the longest common subsequence (LCS) of words between the summary and the reference text. The LCS is the longest sequence of words that appear in the same order in both the summary and the reference text. ROUGE-L considers all such LCS sequences and takes their length into account to compute a weighted average of the F1 score. The ROUGE-L score gives more weight to long sequences of words that are similar in the summary and reference text, and it is often used as a more comprehensive evaluation metric than ROUGE-1 and ROUGE-2.

## 5.2 Sample and Procedure

We evaluated the performance of our model by comparing it against a convenience sample of four leading NLP models: RoBERTa, BERT, DistilBERT and Transformer. Each of the models was trained on our assembly-comment pair corpus (see Section 3), and then they and AsmDocGen were passed a validation input set of undocumented assembly language functions to generate English summaries. The metrics of Section 5.1 were collected for each and for AsmDocGen.

### 5.2.1 Transformer

Transformer, introduced by Vaswani et al. (Vaswani et al., 2017), is a neural network architecture based on the concept of self-attention, which allows the model to weigh the importance of different parts of the input sequence when generating output. The Transformer consists of an encoder and a decoder, each containing multiple layers of self-attention and feed-forward neural networks. Its use of self-attention allows the model to capture long-range dependencies more effectively than other NLP models and parallelize computations across the input sequence, making it more computationally efficient. The Transformer has achieved state-of-the-art performance on a wide range of NLP tasks and has inspired the development of other Transformer-based models such as BERT.

### 5.2.2 BERT

BERT, introduced by Devlin et al. (Devlin et al., 2019) in 2019, is an NLP model developed by Google in 2018. It is based on the transformer architecture and is pre-trained on a large corpus of unannotated text using a masked language modeling task and a next-sentence prediction task. BERT can be trained or fine-tuned on a variety of NLP tasks, achieving state-of-the-art performance on many benchmark NLP datasets with relatively small amounts of task-specific data. Its ability to handle a wide range of NLP tasks has made it a popular model for NLP research and applications.

### 5.2.3 RoBERTa

RoBERTa, introduced by Liu et al. (Liu and Lapata, 2019) in 2019, is an NLP model that is based on the same architecture as BERT but with several modifications to its training process and hyperparameters. The model is trained on a much larger corpus of data, with up to 160 GB of text, and uses dynamic masking during pre-training. RoBERTa also changes the hyperparameters used in BERT, including removing the next sentence prediction task, increasing the batch size, and training the model for longer durations. Additionally, RoBERTa uses byte-pair encoding for subword tokenization, which can improve the model's ability to handle rare and out-of-vocabulary words. These modifications allow RoBERTa to achieve state-of-the-art performance on a wide range of NLP tasks.

### 5.2.4 DistilBERT

DistilBERT, introduced by Sanh et al. (Sanh et al., 2019), is a compact and efficient version of the BERT

Table 2: A comparison of the BLEU scores between our proposed approach and the baseline results.

Models	BLEU Score
Transformer	34.54
RoBERTa	50.01
DistilBERT	50.82
BERT	51.85
AsmDocGen	<b>54.10</b>

model, created through a process called distillation. It has fewer parameters (40% less) than the BERT base model. DistilBERT is trained using knowledge distillation, where the knowledge of a larger model, in this case BERT, is distilled into a smaller model. Despite its smaller size, DistilBERT achieves performance similar to that of the larger BERT model for many NLP tasks, while being faster and requiring less memory to train and run. It has become a popular choice for NLP tasks, where computational resources are limited.

## 6 RESULTS

Table 2 shows the BLEU score for Transformer, RoBERTa, DistilBERT, BERT and AsmDocGen. Table 3 shows that, on average, AsmDocGen outperforms the trained sample models BERT, RoBERTa and DistilBERT by 26%, 23%, and 20% in terms of precision, recall, and F1-score, respectively.

### 6.1 Quantitative Analysis

In order to investigate the effectiveness of AsmDocGen, we compared it with the baseline models mentioned above using the metrics BLEU, ROUGE-1, ROUGE-2, and ROUGE-L. The comparison between the Transformer and the pre-trained models shows the effectiveness of transfer learning in training a model for code summarization using a small dataset. The outcome of the comparison between AsmDocGen, BERT, RoBERTa, and DistilBERT shows the high capability of CodeBERT to learn and understand the assembly language.

The results in Table 2 show that the pre-trained models, despite their initial unfamiliarity with assembly code syntax, significantly outperform a model (Transformer) trained from scratch using our smaller dataset. This suggests that patterns and knowledge, even when marginally relevant, captured from extensive datasets during the pretraining phase can be significantly beneficial for learning new tasks with limited data.

Consistent outperformance of AsmDocGen against the other models, shown in Table 3, supports the inference that CodeBERT’s pre-training objectives, which specifically target high-level code and comment pairs, can provide an advantage for understanding low-level languages compared to models that were pre-trained on general language data. Additionally, our results highlight the importance of selecting a model that is well suited to the task at hand rather than relying solely on pretraining size or architecture.

### 6.2 Qualitative Analysis

Our observations found three types of correctly generated comments in terms of their similarity to the ground-truth comments. These three types are:

- **Identical:** Descriptions that include an exact set of words in the same order as the ground truth description (see Table 4).
- **Partially Similar:** This group refers to comments that are semantically similar to the ground truth but only include a subset of original words (see Table 5).
- **Contextually Similar:** These comments have different structures and wording from the ground truth, but convey the relevant context or semantics (see Table 6).

In this section, we present some examples from each group to compare the ground truth with the results generated by the model. The aim of this demonstration is to illustrate the model’s accuracy in predicting results that are consistent with the actual ones. This information is important in evaluating the performance of the model and determining its effectiveness in solving the problem it was designed to address.

Table 5 shows some examples of partially similar generated comments. It provides a noteworthy example that showcases the learning capabilities of the model, going beyond mere pattern memorization. The table demonstrates how the model accurately comprehended the meaning of the words “unopened” and “closed”, and produced correct predictions. This outcome aligns with our expectations for the model and highlights its desired performance.

Sometimes, the comments generated by our model, as presented in Table 6, differ completely from the ground-truth comments. This raises the question of whether these predictions are actually relevant to the code’s functionality.

As shown in Table 6, AsmDocGen generated the comment “returns the size of the queue,” whereas the ground-truth comment is “fuzzy compare operations.”



Table 3: Performance comparison of AsmDocGen and baseline results based on the ROUGE-1, ROUGE-2, and ROUGE-L scores in terms of precision, recall, and F1-score.

	Models	ROUGE-1	ROUGE-2	ROUGE-L	Average
Precision	RoBERTa	0.50	0.16	0.50	0.39
	DistilBERT	0.52	0.45	<b>0.52</b>	0.50
	BERT	0.56	0.45	0.50	0.50
	AsmDocGen	<b>0.70</b>	<b>0.69</b>	0.51	<b>0.63</b>
Recall	RoBERTa	0.36	0.16	0.37	0.30
	DistilBERT	0.47	0.30	0.47	0.41
	BERT	0.55	0.39	0.46	0.47
	AsmDocGen	<b>0.70</b>	<b>0.56</b>	<b>0.48</b>	<b>0.58</b>
F1-score	RoBERTa	0.42	0.16	0.43	0.34
	DistilBERT	0.49	0.36	<b>0.49</b>	0.45
	BERT	0.55	0.43	0.48	0.49
	AsmDocGen	<b>0.68</b>	<b>0.59</b>	<b>0.49</b>	<b>0.59</b>

Table 4: Three examples of AsmDocGen’s description that are an exact match to the true description.

Identical	
Predicted comment	Ground truth
find vertex number and edges out	find vertex number and edges out
perform subtraction then addition	perform subtraction then addition
perform multiple multiplication	perform multiple multiplication

Table 5: Samples of the predicted output which uses similar words when compared to the true description.

Partially Similar	
Predicted comment	Ground truth
find unopened closing brackets	check too many closing brackets
find second thursday of sept	find second thursday of september 2013
find partially paired brackets	find paired and nested brackets

Given that in a fuzzy system, a queue can be utilized to keep track of intermediate results or manage the sequence of various operations, the generated comment seems relevant. Our manual examination of the code confirmed that this assembly code specifically uses a queue in a fuzzy system. Additionally, it is worth noting that while "Find edge destination" and "find last node" are not identical, they still convey similar con-

Table 6: Predicted output for sentences that are completely different and not identical to the truth, but similar to the concept based on code’s functionality.

Contextually Similar	
Predicted comment	Ground truth
returns the size of the queue	fuzzy compare operations
find last node	find edge destination

cepts. "Find edge destination" is a specific term in graph theory, while "find last node" is a broader term that can apply to various structures, such as linked lists, trees, or graphs. These findings suggest that although AsmDocGen’s comments may not be identical to the ground truth, they are still relevant to the functionality of the code.

## 7 CONCLUSIONS

The comparison between the Transformer and the pretrained models shows the effectiveness of transfer learning in training a model for code summarization using a small dataset. The outcome of the comparison between AsmDocGen, BERT, RoBERTa, and DistilBERT shows the high capability of CodeBERT to learn and understand the assembly language, and the overall performance of AsmDocGen as a whole. The evaluation supports the argument that AsmDocGen represents a significant advance in the field of automatic code commenting for low-level programming languages by using a transformer-based model. The reference corpus we created of assembly-comment pairs was shown to be beneficial for retraining Code-

BERT to align it better with this assembly documentation task. This innovative approach sets a new standard in the area.

Future work on this research could include improving the performance of the model by training it on a larger dataset. Another area of improvement could be expanding the system to overcome compiler optimization challenges in generating comments for similar functions that are compiled with different compilers/optimization levels. In addition, the system could be modified to generate multi-sentence summaries, instead of just one-sentence comments.

## ACKNOWLEDGMENT

This research is supported by NSERC Alliance Grants (ALLRP 561035-20), BlackBerry Limited, and Defence Research and Development Canada (DRDC).

## REFERENCES

- Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. (2020). A transformer-based approach for source code summarization. *In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, page 4998–500.
- Allamanis, M., Peng, H., and Sutton, C. (2016). A convolutional attention network for extreme summarization of source code. *In Proceedings of the International Conference on Machine Learning*, page 2091–2100.
- Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *ArXiv*, 1409.
- Barone, A. V. M. and Sennrich, R. (2021). A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *In Proceedings of the Eighth International Joint Conference on Natural Language Processing*, 2:314–319.
- Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007). Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9):577–591.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. *In Advances in Neural Information Processing Systems*, (33):1877–1901.
- Cao, Y., Bi, W., Fang, M., and Tao, D. (2020). Pretrained language models for dialogue generation with multiple input sources. pages 909–917.
- Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation.
- Christian, H., Agus, M. P., and Suhartono, D. (2016). Single document automatic text summarization using term frequency-inverse document frequency (tf-idf). *ComTech: Computer, Mathematics and Engineering Applications*, 7:285.
- Clement, C., Drain, D., Timcheck, J., Svyatkovskiy, A., and Sundaresan, N. (2020). Pymt5: Multi-mode translation of natural language and python code with transformers. *In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*.
- Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, page 4171–4186.
- Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). Code-bert: A pre-trained model for programming and natural languages. *In Findings of the Association for Computational Linguistics: EMNLP 2020*, page 1536–1547.
- Gao, P., Geng, S., Qiao, Y., Wang, X., Dai, J., and Li, H. (2021). Scalable transformers for neural machine translation.
- Gu, X., Zhang, H., and Kim, S. (2018). Deep code search. *In Proceedings of IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, page 933–944.
- Gupta, A., Chugh, D., Anjum, and Katarya, R. (2022). Automated news summarization using transformers. pages 249–259.
- Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. (2018a). Deep code comment generation. *In Proceedings of the 26th Conference on Program Comprehension*, page 200–210.
- Hu, X., Li, G., Xia, X., Lo, D., Lu, S., and Jin, Z. (2018b). Summarizing source code with transferred api knowledge. *In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence Main track*, pages 2269–2275.
- Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search.
- Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. *In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 1:2073–2083.
- Kusupati, U. and Ailavarapu, V. R. T. (2022). Natural language to code using transformers.
- LeClair, A., Jiang, S., and McMillan, C. (2019). A neural model for generating natural language summaries of program subroutines. *In Proceedings of the 41st International Conference on Software Engineering*, page 795–806.

- Lee, M., Shin, H., Lee, D., and Choi, S.-P. (2021). Korean grammatical error correction based on transformer with copying mechanisms and grammatical noise implantation methods. *Sensors*, 21:2658.
- Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. (2020). Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, page 7871–7880.
- Liu, Y. and Lapata, M. (2019). Text summarization with pretrained encoders.
- Olabiyi, O. O., Bhattarai, P., Bruss, C. B., and Kulis, Z. (2020). Dlgnet-task: An end-to-end neural network framework for modeling multi-turn multi-domain task-oriented dialogue. *In Proceedings of the 2nd Workshop on Natural Language Processing for Conversational AI*.
- Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training.
- Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2019). Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter.
- Sarkar, K. (2013). Automatic single document text summarization using key concepts in documents. *J. Inf. Process. Syst.*, 9:602–620.
- Steidl, D., Hummel, B., and Juergens, E. (2013). Quality analysis of source code comments. *In Proceedings of the 2013 21st International Conference on Program Comprehension (ICPC), San Francisco, CA, USA*, page 20–21.
- Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems*, 4:3104–3112.
- Tenny, T. (1988). Program readability: procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *In Advances in Neural Information Processing Systems*, (30):5998–6008.
- Verma, P. and Om, H. (2018). Mcmr: Maximum coverage and relevancy with minimal redundancy based multi-document summarization. *Expert Systems with Applications*, 120.
- Wang, R., Zhang, H., Lu, G., Lyu, L., and Lyu, C. (2020). Fret: Functional reinforced transformer with bert for code summarization. *IEEE Access*.
- Woodfield, S. N., Dunsmore, H. E., and Shen, V. Y. (1981). The effect of modularization and comments on program comprehension. *In Proceedings of the 5th international conference on Software engineering*, page 215–223.
- Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Łukasz Kaiser, Gouws, S., Kato, Y., Kudo,
- T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google’s neural machine translation system: Bridging the gap between human and machine translation.
- Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., and Le, Q. (2019). Xlnet: Generalized autoregressive pre training for language understanding. *In Advances in Neural Information Processing Systems*, 32:5753–5763.
- Yin, P., Deng, B., Chen, E., Vasilescu, B., and Neubig, G. (2018). Learning to mine aligned code and natural language pairs from stack overflow. *In International Conference on Mining Software Repositories, MSR*, pages 476–486. ACM.
- Zhou, G. and Su, J. (2002). Named entity recognition using an hmm-based chunk tagger. *proceedings of the 40th Annual Meeting on Association for Computational Linguistics*.