Shahid Beheshti University

# Artificial Neural Networks

M.Sc - Fall 2024

## Assignment 6 - Practical Exercise

## Code Summarization

Author:
Zahra Mohammad Beigi

Student Number:
402422144

January 17, 2025

# Contents

**Abstract**

This report details the fine-tuning of Transformer-based models for the task of code summarization. The objective is to generate meaningful natural language descriptions (docstrings) for Python code snippets. Using the CodeXGLUE CT Code-to-Text dataset, pre-trained sequence-to-sequence models were trained and evaluated to understand their effectiveness in this domain. The report presents the methodology, experiments, results, and key findings.

# 1 Introduction

The task of code summarization involves generating concise and accurate natural language descriptions for code snippets. This is particularly useful for improving code readability, understanding, and documentation. With the advent of large-scale pre-trained models, such as Transformers, the ability to perform such tasks has significantly improved. The objective of this project is to fine-tune a pre-trained Transformer-based model to summarize Python code snippets effectively.

# 2 Dataset

The dataset used in this project is the **CodeXGLUE CT Code-to-Text dataset**, specifically designed for code summarization tasks. It pairs Python code snippets with their corresponding docstrings, which serve as natural language summaries. In this project, a smaller subset comprising the first 1% of the dataset was utilized to reduce training time. The dataset is split into training, validation, and test sets as shown below:

- **Training Set:** 2,518 samples
- **Validation Set:** 139 samples
- **Test Set:** 149 samples

Each sample consists of:

1. **Code:** A Python code snippet.
2. **Docstring:** A human-written description explaining the purpose or functionality of the code.

## 2.1 Sample Data

Below is an example of a data sample from the training set:
**Code**

```
def settext(self, text, cls='current'):
"""Set the text for this element.

Arguments:
```

```
text (str): The text
cls (str): The class of the text, defaults to ``current``
(leave this unless you know what you are doing).
There may be only one text content element of each class
associated with the element.
"""
self.replace(TextContent, value=text, cls=cls)
```

**Docstring**

```
Set the text for this element.

Arguments:
text (str): The text
cls (str): The class of the text, defaults to ``current``
(leave this unless you know what you are doing).
There may be only one text content element of each class
associated with the element.
```

This structure provides a clear mapping between code snippets and their summaries, enabling effective training of the summarization model.

# 3 Preprocessing

To prepare the dataset for model training, we employed tokenization, and format the dataset to make it compatible with the chosen Transformer-based model. The preprocessing pipeline is outlined below.

## 3.1 Tokenization

The `Salesforce/codet5-small` tokenizer was used to tokenize the `code` and `docstring` fields into input and target sequences, respectively. Initially, we experimented with `Salesforce/codet5-base`, but it resulted in out-of-memory (OOM) errors due to the larger model size. Therefore, we opted for the smaller variant, `Salesforce/codet5-small`.

## 3.2 Sequence Lengths

To balance computational efficiency and performance, we limited the maximum sequence length to 128 tokens for both input (code) and output (docstring) sequences:

```
max_input_length = 128
max_output_length = 128
```

## 3.3 Processed Dataset Summary

After preprocessing, the training dataset contained the following features:

- **code:** The raw Python code snippet.

- **docstring:** The reference summary of the code.
- **input_ids:** Tokenized input sequences.
- **attention_mask:** Attention masks indicating non-padding tokens.
- **labels:** Tokenized target sequences.

The training subset had a total of 1,259 examples after preprocessing.

# 4 Model Implementation

In this section, we implemented a code summarization pipeline using two state-of-the-art pre-trained models: PLBART and CodeT5.

We used the Hugging Face Transformers library, which provides an easy-to-use interface for working with pre-trained models like PLBART and CodeT5. The pipeline consists of the following steps:

1. **Loading and Tokenizing Data**: We utilized the `AutoTokenizer` class to load the tokenizer corresponding to each model. The dataset was tokenized and formatted for use with the model.
2. **Training**: The models were trained using the AdamW optimizer with a learning rate of $5 \times 10^{-5}$. We ran training for three epochs, monitoring the training loss after each epoch. During the training, we applied a data collator for sequence-to-sequence tasks.
3. **Evaluation**: The evaluation was performed using BLEU and ROUGE metrics, which are standard measures for evaluating text generation models. BLEU measures the precision of n-grams between the generated and reference summaries, while ROUGE evaluates recall with a focus on n-grams and the longest common subsequence.

# 5 Hyperparameter Tuning

To achieve optimal performance, we fine-tuned several hyperparameters during the training process:

- **Learning Rate**: A learning rate of $5 \times 10^{-5}$ was chosen based on common practice for fine-tuning transformer-based models. This learning rate balances model convergence and stability during training.
- **Epochs**: We set the number of training epochs to 3, which provided a good trade-off between training time and model performance. Longer training could potentially improve results, but the evaluation metrics achieved after 3 epochs were satisfactory.
- **Batch Size**: A batch size of 4 was used due to hardware constraints and to ensure efficient memory usage during training. A larger batch size could have been beneficial for faster convergence, but this setup proved to be effective for our evaluation.

- **Max Sequence Length**: The maximum sequence length for the inputs and outputs was set to 512 tokens. This length was chosen to accommodate the average size of the code snippets in the dataset.

# 6 Comparison

We compared the performance of the two models, PLBART and CodeT5, on the code summarization task using the BLEU and ROUGE metrics.

| **Model** | **BLEU** | **ROUGE Scores** |
|---|---|---|
| PLBART | 0.956 | ROUGE-1: 0.904, ROUGE-2: 0.904, ROUGE-L: 0.907, ROUGE-Lsum: 0.905 |
| CodeT5 | 0.969 | ROUGE-1: 0.992, ROUGE-2: 0.991, ROUGE-L: 0.991, ROUGE-Lsum: 0.992 |

Table 1: Comparison of PLBART and CodeT5 performance

From the comparison, it is clear that CodeT5 outperforms PLBART on both BLEU and ROUGE metrics, especially in terms of ROUGE scores, which suggest that CodeT5 has better recall in generating relevant n-grams and subsequences in the code summaries.

# 7 Conclusion

In this study, we implemented and fine-tuned two pre-trained models, PLBART and CodeT5, for the task of code summarization. After training and evaluating both models, we observed that CodeT5 significantly outperforms PLBART, as indicated by its higher BLEU and ROUGE scores.

This result demonstrates the effectiveness of CodeT5 in capturing the structure and semantics of code snippets and generating more accurate summaries. Although PLBART showed strong performance, CodeT5's higher ROUGE scores suggest it is better at recalling important details and providing more informative summaries.