Code to Comment Translation: A Comparative Study on Model Effectiveness & Errors

Junayed Mahmud,
Fahim Faisal,
Raihan Islam Arnob,
Antonios Anastasopoulos,
and Kevin Moran

The First Workshop on Natural Language Processing for Programming (NLP4Prog'21)

AI Pair Programmer

GitHub
Copilot

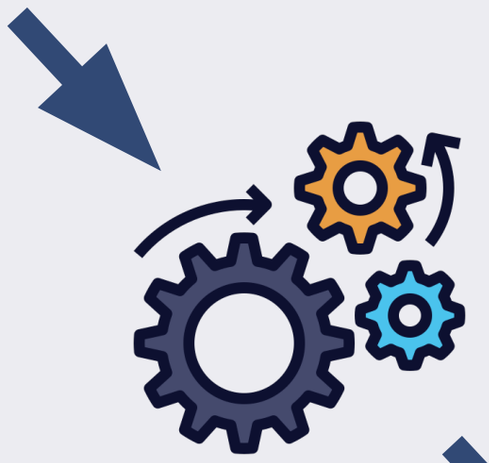
```typescript
interface CommentMarker {
  start: string;
  end: string;
}

const markers: { [language:string]: CommentMarker } = {

```

```python
def strip_suffix(filename):
    """
    Removes the suffix from a filename
    """
    return filename[:filename.rfind('.')]

def te

```

```javascript
import urllib.request, json

```

Automated Software Documentation

```
def isPrime(n):
    prime = True
    if n==1:
        return False
    for i in range (2, n):
        if n % i == 0:
            prime = False
    return prime
```



Automated Software Documentation Tool

Lexical Gap

Return true if a given number is prime, and false otherwise

Related Work

Related Work

Summarizing Source Code using a Neural Attention Model

Srinivasan Iyer **Ioannis Konstas** **Alvin Cheung** **Luke Zettlemoy**
Computer Science & Engineering
University of Washington
Seattle, WA 98195
{sviyer,ikonstas,akcheung,lsz}@cs.washington.edu

Abstract

```
1. Source Code (C#):
public int TextWidth(string text) {
                                  lock();
                                  ctualWidth
                                  g rounded up
                                  h inside a
```

A Neural Model for Generating Natural Language Summaries of Program Subroutines

Alexander LeClair*, Siyuan Jiang†, Collin McMillan*
*Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
Email: {aleclair, cmc}@nd.edu
†Dept. of Computer Science
Eastern Michigan University
Ypsilanti, MI, USA
Email: sjiang1@emich.edu

***Abstract*—Source code summarization – creating natural language descriptions of source code behavior – is a rapidly-growing research topic with applications to automatic documentation generation, program comprehension, and software maintenance. Traditional techniques relied on heuristics and templates built manually by human experts. Recently, data-driven approaches based on neural machine translation have largely overtaken template-based systems. But nearly all of these techniques rely almost entirely on programs having good internal documentation; without clear identifier names, the models fail to create good summaries. In this paper, we present a neural model that combines words from code with code structure from an AST. Unlike previous approaches, our model processes each data source as a separate input, which allows the model to learn code structure independent of the text in code. This process helps our approach provide coherent summaries in many cases even when zero internal documentation is provided. We evaluate our technique with a dataset we created from 2.1m Java methods. We find improvement over two baseline techniques from SE literature and one from NLP literature.**

***Index Terms*—automatic documentation generation, source code summarization, code comment generation**

I. Introduction

A "summary" of source code is a brief natural language description of that section of source code [1]. One of the most common targets for summarization are the subroutines in a

technologies that improve aut
process," and "that documenta
ter extract knowledge from c
code [7]. However, the state-of
since that time for tool suppor
generation. Tools such as Ja
automate the format and pres
still leave programmers with th
writing the text and examples.

Research into generation o
of code has come to be kn
rization" [9], with significant
of summaries of subroutines:
progress was made based on
templates [1], [10]–[14] or eve
tions such as mimicking humar
as in many research areas and
by Allamanis *et al.* [16], these
way to AI based on big data i

The inspiration for a vast m
code summarization originates
(NMT) from the natural langu
munity. An NMT system con
another. It is typically thoug

Deep Code Comment Generation*

Xing Hu[1] , Ge Li[1], Xin Xia[2], David Lo[3], Zhi Jin[1]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, China
[2]Faculty of Information Technology, Monash University, Australia
[3]School of Information Systems, Singapore Management University, Singapore
[1]{huxing0101,lige,zhijin}@pku.edu.cn, [2]xin.xia@monash.edu, [3] davidlo@smu.edu.sg

ABSTRACT

During software maintenance, code comments help developers comprehend programs and reduce additional time spent on reading and navigating source code. Unfortunately, these comments are often mismatched, missing or outdated in the software projects. Developers have to infer the functionality from the source code. This paper proposes a new approach named DeepCom to automatically generate code comments for Java methods. The generated comments aim to help developers understand the functionality of Java methods. DeepCom applies Natural Language Processing (NLP) techniques to learn from a large code corpus and generates comments from learned features. We use a deep neural network

1 INTRODUCTION

In software development and maintenance, developers spend around 59% of their time on program comprehension activities [45]. Previous studies have shown that good comments are important to program comprehension, since developers can understand the meaning of a piece of code by using the natural language description of the comments [35]. Unfortunately, due to tight project schedule and other reasons, code comments are often mismatched, missing or outdated in many projects. Automatic generation of code comments can not only save developers' time in writing comments, but also help in source code understanding.

Many approaches have been proposed to generate comments for ethods [24, 35] and classes [25] of Java, which is the most popu- programming language in the past 10 years[1]. Their techniques ry from the use of manually-crafted [25] to Information Retrieval) [14, 15]. Moreno et al. [25] defined heuristics and stereotypes to nthesize comments for Java classes. These heuristics and stereo- pes are used to select information that will be included in the mment. Haiduc et al. [14, 15] applied IR approaches to generate mmaries for classes and methods. IR approaches such as Vector ace Model (VSM) and Latent Semantic Indexing (LSI) usually arch comments from similar code snippets. Although promising, ese techniques have two main limitations: First, they fail to ex- ct accurate keywords used for identifying similar code snippets en identifiers and methods are poorly named. Second, they rely whether similar code snippets can be retrieved and how similar e snippets are.

Recent years have seen an emerging interest in building proba- istic models for large-scale source code. Hindle et al. [17] have dressed the naturalness of software and demonstrated that code n be modeled by probabilistic models. Several subsequent studies ve developed various probabilistic models for different software sks [12, 23, 40, 41]. When applied to code summarization, different om IR-based approaches, existing probabilistic-model-based ap- oaches usually generate comments directly from code instead of nthesizing them from keywords. One of such probabilistic-model- sed approaches is by Iyer et al. [19] who propose an attention- sed Recurrent Neural Network (RNN) model called CODE-NN. builds a language model for natural language comments and gns the words in comments with individual code tokens directly attention component. CODE-NN recommends code comments ven source code snippets extracted from Stack Overflow. Experi- ental results demonstrate the effectiveness of probabilistic models code summarization. These studies provide principled methods r probabilistically modeling and resolving ambiguities both in tural language descriptions and in the source code

Project-Level Encoding for Neural Source Code Summarization of Subroutines

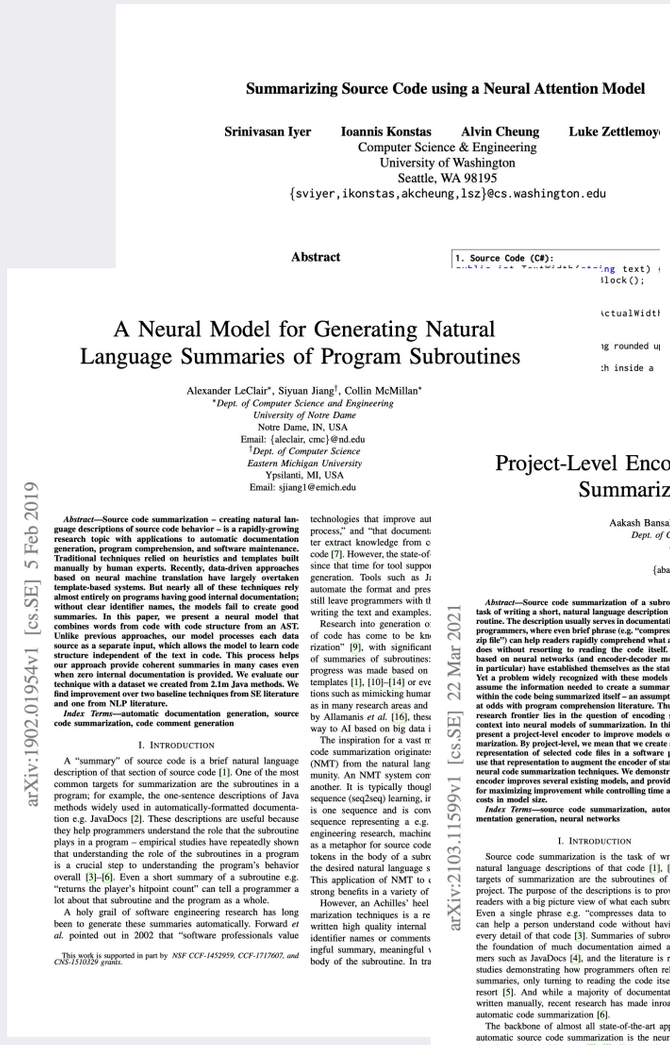
Aakash Bansal, Sakib Haque, and Collin McMillan
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{abansal1, shaque, cmc}@nd.edu

***Abstract*—Source code summarization of a subroutine is the task of writing a short, natural language description of that subroutine. The description usually serves in documentation aimed at programmers, where even brief phrase (e.g. "compresses data to a zip file") can help readers rapidly comprehend what a subroutine does without resorting to reading the code itself. Techniques based on neural networks (and encoder-decoder model designs in particular) have established themselves as the state-of-the-art. Yet a problem widely recognized with these models is that they assume the information needed to create a summary is present within the code being summarized itself – an assumption which is at odds with program comprehension literature. Thus a current research frontier lies in the question of encoding source code context into neural models of summarization. In this paper, we present a project-level encoder to improve models of code summarization. By project-level, we mean that we create a vectorized representation of selected code files in a software project, and use that representation to augment the encoder of state-of-the-art neural code summarization techniques. We demonstrate how our encoder improves several existing models, and provide guidelines for maximizing improvement while controlling time and resource**

roots in machine translation [10], in which the encoder creates a vectorized representation of a sentence in one language (e.g. English), while the decoder creates a representation of that same sentence in a different language (e.g. French). When trained with enough data (on the order of millions of examples [11]), these models can learn to associate patterns in the encoder representation to patterns in the decoder representation. After training, the encoder can be given an input example, and the model can generate a likely decoder representation and therefore a likely output example – and translate French sentences to English. For source code, the encoder's job is to represent the source code, while the decoder represents the source code summary – give the encoder source code, and the decoder generates a summary.

The obvious problem with these approaches is that they can only generate a summary based on whatever source code is passed to the encoder. Thus these approaches make

Related Work



Summarizing Source Code using a Neural Attention Model

Srinivasan Iyer **Ioannis Konstas** **Alvin Cheung** **Luke Zettlemoy**
Computer Science & Engineering
University of Washington
Seattle, WA 98195
{sviyer,ikonstas,akcheung,lsz}@cs.washington.edu

Abstract

A Neural Model for Generating Natural Language Summaries of Program Subroutines

Alexander LeClair*, Siyuan Jiang†, Collin McMillan*
*Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
Email: {aleclair, cmc}@nd.edu
†Dept. of Computer Science
Eastern Michigan University
Ypsilanti, MI, USA
Email: sjiang1@emich.edu

Deep Code Comment Generation*

Xing Hu[1] , Ge Li[1], Xin Xia[2], David Lo[3], Zhi Jin[1]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, China
[2]Faculty of Information Technology, Monash University, Australia
[3]School of Information Systems, Singapore Management University, Singapore
[1]{huxing0101,lige,zhijin}@pku.edu.cn, [2]xin.xia@monash.edu, [3] davidlo@smu.edu.sg

ABSTRACT

1 INTRODUCTION

Project-Level Encoding for Neural Source Code Summarization of Subroutines

Aakash Bansal, Sakib Haque, and Collin McMillan
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{abansal1, shaque, cmc}@nd.edu

BLEU

METEOR

ROUGE-L

PROBLEM

returns the tip text for this property

```
public String numRulesTipText() {
        return "Number of rules to find.";
}
```

Human written docstring

Smoothed BLEU-4: 16.99

Error: Repetition

Code Summarization Tool

returns the text of the text of the current text

Research Questions

RQ_1: *Effectiveness* in predicting natural language summaries

RQ_2: *Errors* made by our studied models

RQ_3: *Differences* in the errors made by different models

DATASET: FUNCOM

Code Summarization

500000 method-comment pairs

Java

Training

Validation

Testing

Funcom	400000	50000	49997

Neural Code Summarization Models

code2seq

- Encoder-decoder model
- Capture the syntactic construction of source code by encoding AST paths
- Applies attention to generate the final sequence

NeuralCodeSum

- Encoder-decoder model with a copy mechanism
- Relative positioning to encode pairwise token

CodeBERT

- Pre-trained on both bimodal and unimodal data
- Similar architecture as RoberTa
- Uses multi-layer bidirectional Transformer

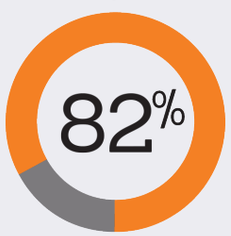
Qualitative Evaluation Methodology

Initial Discussion: Gold set (15 method-comment pairs)

label errors

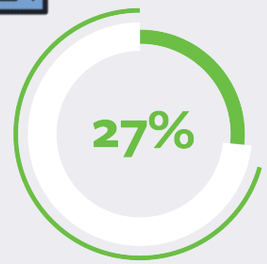
conflict resolve

First round: 148 method-comment pairs,
148 x 3 = 444 predictions



Number of conflicts after first round

Second round: 50 method-comment pairs,
50 x 3 = 150 predictions



Number of conflicts after second round

Taxonomy of the errors

Missing Context (1, 2, 2)

Missing Prog. Language Information (0, 0, 2)
- Missing Attributes that refer to PL specific information.

Missing Database Information (1, 2, 0)
- Missing database attributes that provide needed context to method functionality.

Missing Information (65,56,27)

Missing Critical Information (21, 14, 7)
- Comment is missing critical semantic information.

Missing Task Elaboration (5, 2, 1)
- Did not describe what code was doing properly.

Missing Non-Critical Information (28, 19, 5)
- Useful comment but non-critical info missing.

Missing Web-Related Information (0, 1, 0)
- Comment failed to mention web-related identifier.

Failed to Mention Identifiers (0, 11, 6)
- Does not mention specific variable/attribute names, often using a generic identifier.

Missing Identifier (5, 3, 7)
- No identifier mentioned at all.

Missing Data Structure Information (2, 0, 1)
- Does not capture relevant data structure info

Missing Syntax Information (2, 6, 0)
- Important syntactic information (e.g. code ordering) is missing.

Missing Exception (1, 0, 0)
- Does not mention relevant exception info

Missing Conditional Information (1, 0, 0)
- Misses code branching information

Incorrect Semantic Information (6, 27, 19)

Partial Incorrect Information (6, 11, 3)
- Semantically meaningful, with a few errors.

Semantically Unrelated to Code (0, 11, 13)
- Does not capture code context whatsoever.

Algorithmically Incorrect (0, 5, 3)
- Conveys a different algorithmic meaning as compared to the code.

Incorrect Construction (26, 31, 53)

Incorrect Identifier/Attribute (5, 19, 15)
- Correctly identifies a variable or attribute, but uses it incorrectly.

Incomplete Sentence (1, 1, 10)
- Predicted comment is grammatically incomplete.

Repetition (0, 7, 27)
- Comment contains unnecessary repetition of a word or fragment between 2-3 times.

Extreme Repetition (0, 2, 1)
- Comment contains unnecessary repetition of a word or fragment more than 2-3 times.

Focusing Only on Method Name (20, 1, 0)
- When comment focuses mostly on the method name, which provides an incomplete but partial description of the functionality.

Grammatical Errors (0, 1, 0)
- Grammatical Error is present in predicted caption.

The numbers shown for each category illustrate the number of instances found for (**CodeBERT**, **NeuralCodeSum**, and **code2seq**) respectively

Consistent with Ground Truth (88, 57, 17)

Consistent with Specific Info (30, 15, 5)
- Comment matches ground truth well.

Consistent but Missing Specific Info (56, 35, 12)
- Comment matches ground truth mostly, but misses some important specific information.

Improves upon Semantic Meaning (2, 6, 0)
- The predicted comment matches the ground and improves capturing method meaning.

Consistent but with Unnecessary Info (0, 1, 0)
- Accurate but has some unnecessary info.

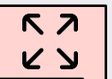
Extraneous/Unnessecary Information Included (2, 3, 4)

Unnecessary Data Structure Info (1, 0, 0)
- Adds unnecessary data structure info to comment.

Unnecessary File Information (0, 1, 1)
- Adds unnecessary file information to comment.

Unnecessary Incorrect Information (1, 2, 3)
- Adds information to comment that is both incorrect and unnecessary.

Over-Generalization (7, 21, 21)

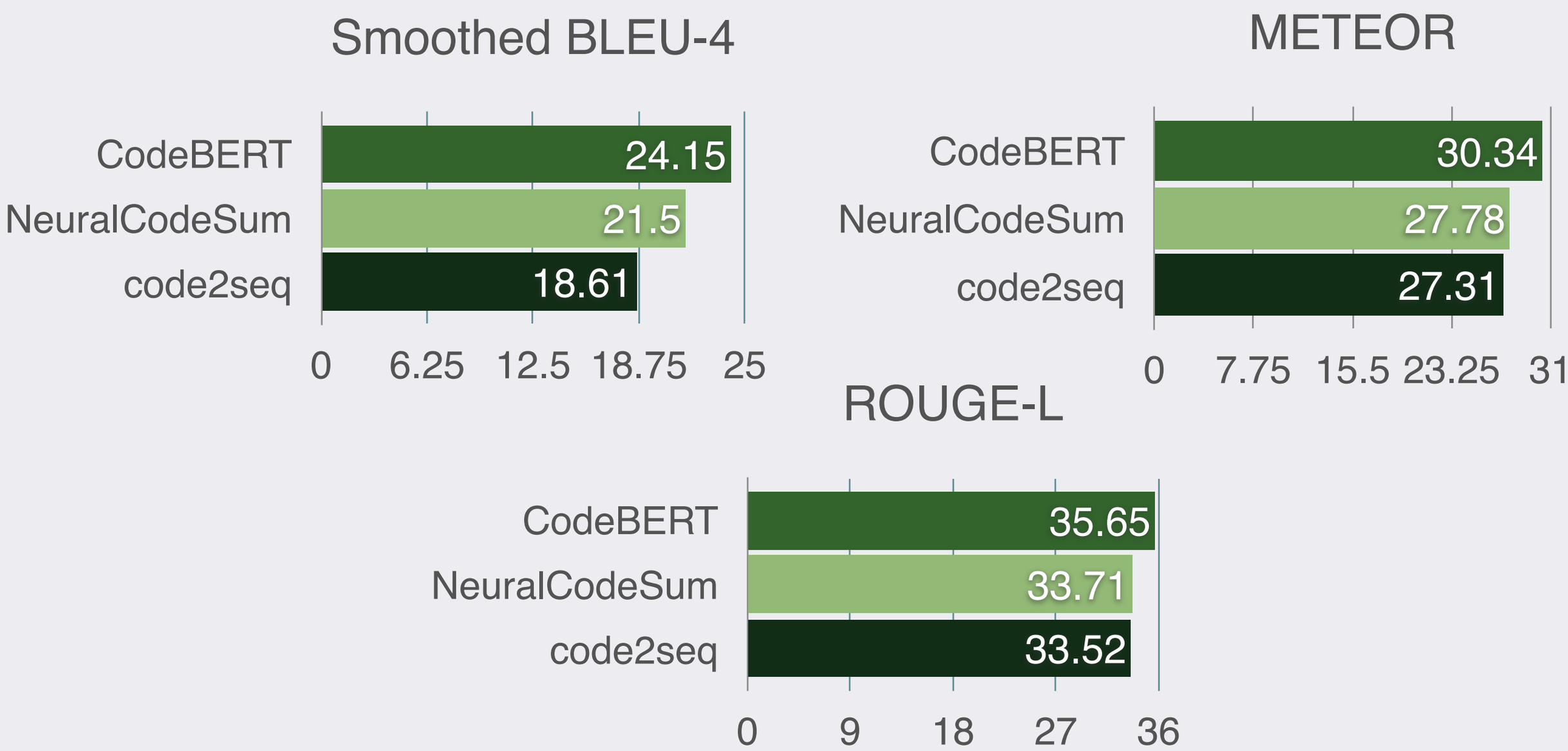


Different Meaning (2, 3, 3)
- Comment over-generalizes on the meaning of the code functionality.

Algorithmically Incorrect (1, 6, 3)
- Overgeneralizes to the point of incorrectness

Missing Attribute Specification (4, 12, 15)
- Uses generic names such as **var.**

RQ1 Results: Prediction of Natural Language Summaries



CodeBERT is the best performing model among the three neural models

RQ2 Results: Types of Errors Compared to Ground Truth

Consistent to the ground truth (30.28%)

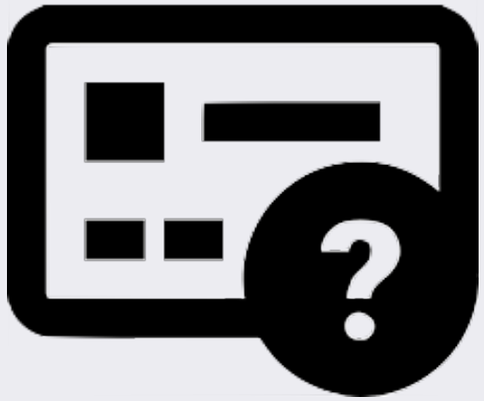
```
public float getDashPhase() {
        return dashPhase;
}
```

Ground truth: gets the dash phase of the basicstroke

Prediction: gets the dashphase

RQ2 Results: Types of Errors Compared to Ground Truth

Consistent to the ground truth (30.28%)



Missing Information (27.66%)

public String getSchema() {

return fSchema;

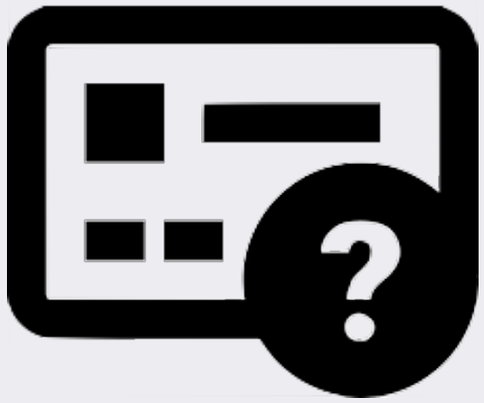
}

Ground truth: returns a path to the xml schema of a extension point

Prediction: returns the name of the xml schema

RQ2 Results: Types of Errors Compared to Ground Truth

Consistent to the ground truth (30.28%)



Missing Information (27.66%)

Incorrect Construction (20.56%)

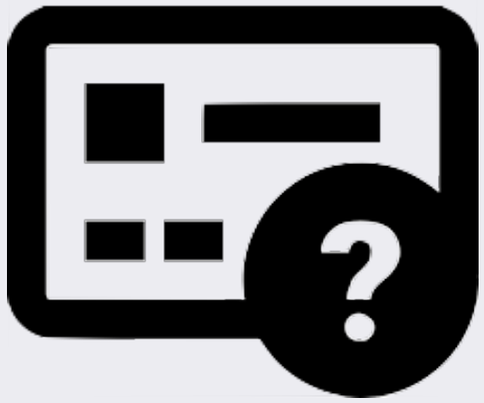
```
public String numRulesTipText() {
    return "Number of rules to find.";
}
```

Ground truth: returns the tip text for this property

Prediction: returns the text of the text of the current text

RQ2 Results: Types of Errors Compared to Ground Truth

Consistent to the ground truth (30.28%)



Missing Information (27.66%)

Incorrect Construction (20.56%)

Incorrect or failed to recognize identifiers (13.20%)

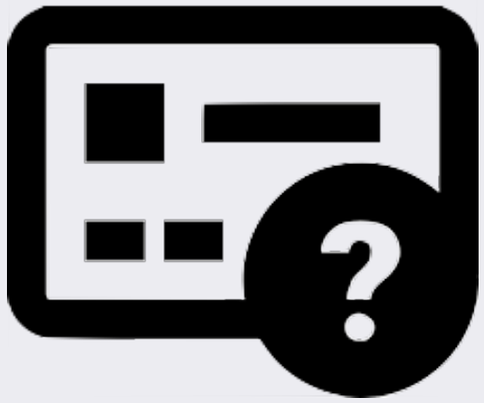
```
public void setHeight (int height) {
containerHeight = height;
}
```

Ground truth: this method sets the minimum height of the table in pixels

Prediction: sets the height of the image

RQ2 Results: Types of Errors Compared to Ground Truth

Consistent to the ground truth (30.28%)



Missing Information (27.66%)

Incorrect Construction (20.56%)

Incorrect or failed to recognize identifiers (13.20%)

Over-generalization (9.15%)

```
public boolean isCreateIds() {
        return createIds;
}
```

Ground truth: returns true if the model automatically creates ids and resolves id collisions

Prediction: returns whether the is the default id

RQ3 Results: Differences Between the Errors Made by Different Models

Most frequent categories

CodeBERT & NeuralCodeSum → Consistent but missing specific information

code2seq → Repetition

CodeBERT → ***Focusing only on method name***

```
public void setReadTimeout (int timeout) {
        if (0 > timeout) {
                this.readTimeout = timeout;
        }
}
```

Ground truth: sets the timeout value in milliseconds for reading from the input stream

Prediction: sets the read timeout

RQ3 Results: Differences Between the Errors Made by Different Models

Most frequent categories

CodeBERT & NeuralCodeSum → Consistent but missing specific information

code2seq → Repetition

CodeBERT → ***Focusing only on method name***

NeuralCodeSum & code2seq → ***Semantically unrelated to code***

```
public float getDashPhase() {
        return dashPhase;
}
```

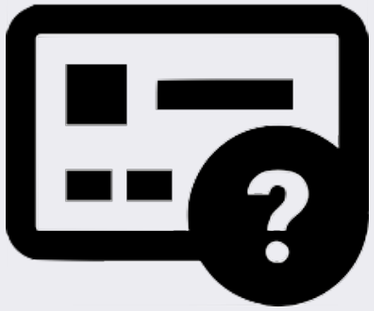
Ground truth: gets the dash phase of the basicstroke

Prediction: get the current velocity of the current value

Conclusion

CodeBERT performs the best compared to other models based on reference-based metrics

Over-generalization

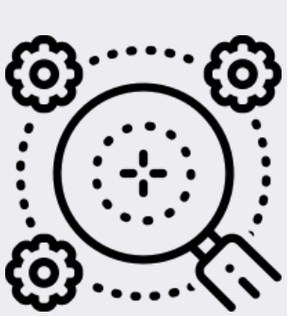


Missing Information

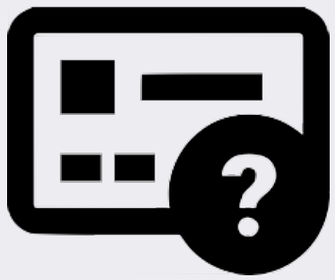
Incorrect Construction

Conclusion

CodeBERT performs the best compared to other models based on reference-based metrics



Over-

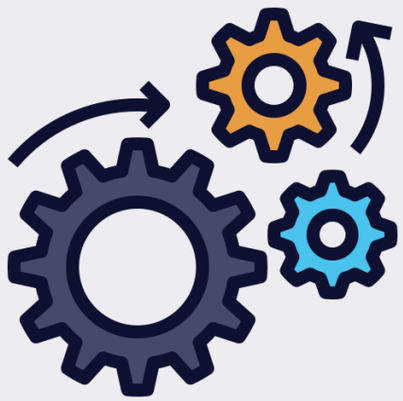


Missing Information

Incorrect Construction

Future studies should also do similar kind of qualitative study

Future work

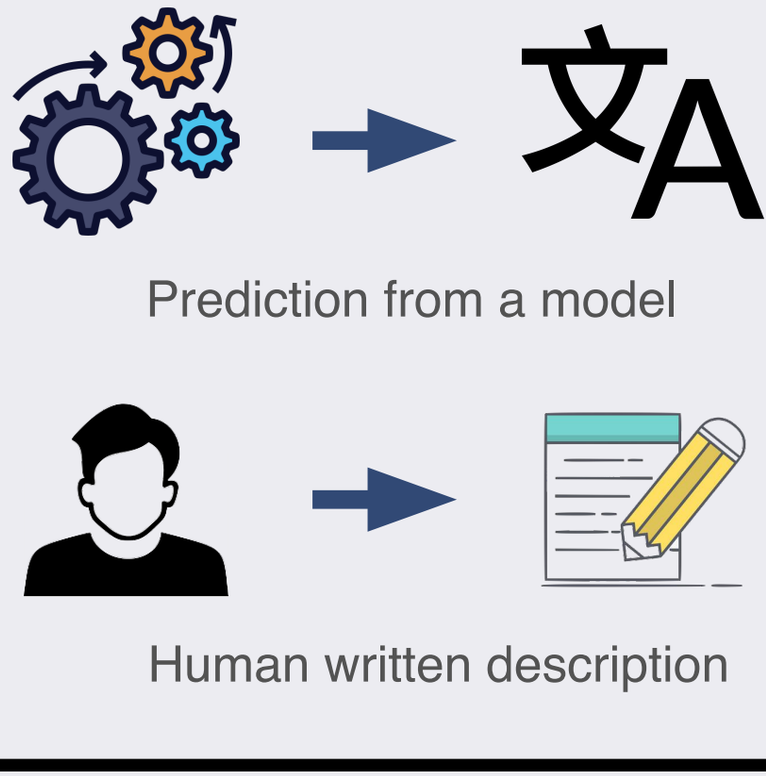


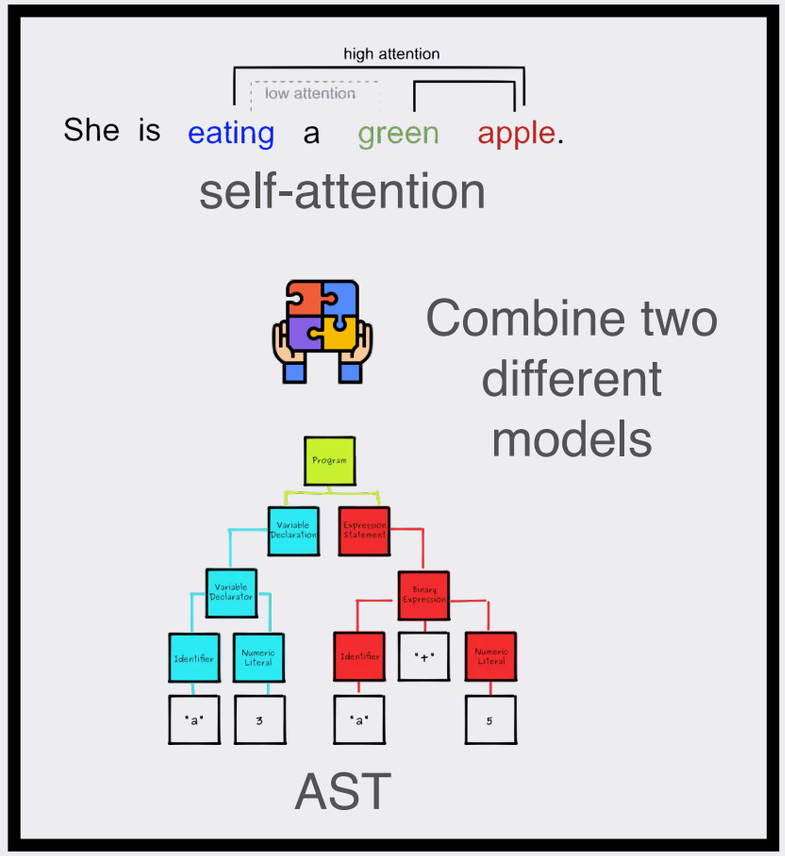
Prediction from a model

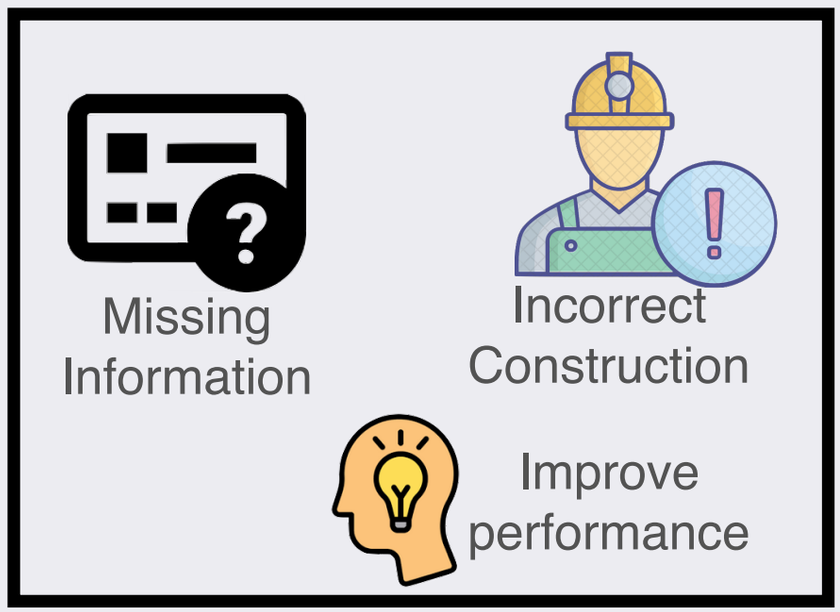
Human written description

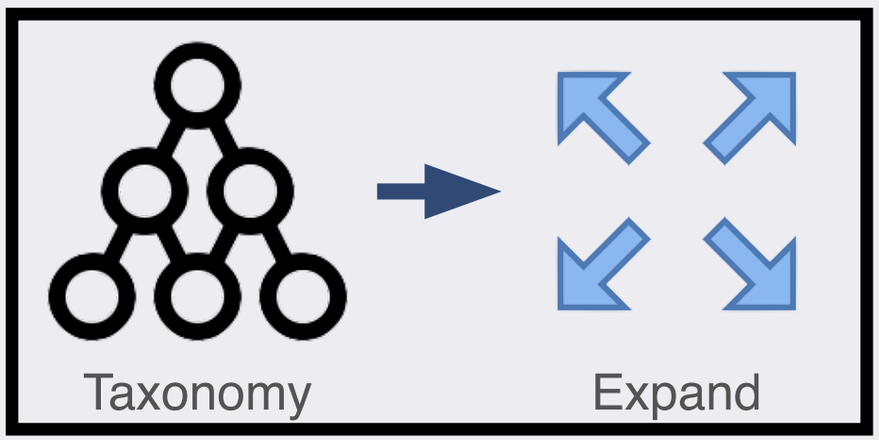
Use newer machine translation metrics

Future work


Prediction from a model
Human written description
Use newer machine translation metrics


high attention
low attention
She is eating a green apple.
self-attention
Combine two different models
Program
Variable Declaration
Expression Statement
Variable Declarator
Binary Expression
Identifier
Numeric Literal
Identifier
"+"
Numeric Literal
"a"
3
"a"
5
AST


Missing Information
Incorrect Construction
Improve performance


Taxonomy
Expand

Relevant Links

Zenodo

https://zenodo.org/record/4904024#.YNRJoy9h3AY

Github

https://github.com/SageSELab/CodeSumStudy

Junayed Mahmud (jmahmud@gmu.edu)

Fahim Faisal (ffaisal@gmu.edu)

Raihan Islam Arnob (rarnob@gmu.edu)

Antonios Anastasopoulos (antonis@gmu.edu)

Kevin Moran (kpmoran@gmu.edu)

Problem

```
public String numRulesTipText() {
        return "Number of rules to find.";
}
```

Error: Repetition

Human written docstring

returns the tip text for this property

Smoothed BLEU-4: 16.99

Code Summarization Tool

returns the text of the text of the current text

Qualitative Evaluation Methodology

Initial Discussion: Gold set (15 method-comment pairs)

label errors

conflict resolve

148

First round: 148 method-comment pairs,
148 x 3 = 444 predictions

82%

Number of conflicts after first round

50

Second round: 50 method-comment pairs,
50 x 3 = 150 predictions

27%

Number of conflicts after second round

Taxonomy of the errors

Missing Context (1, 2, 2)
- **Missing Prog. Language Information (0, 0, 2)**
  - Missing Attributes that refer to PL specific information.
- **Missing Database Information (1, 2, 0)**
  - Missing database attributes that provide needed context to method functionality.

Missing Information (65,56,27)
- **Missing Critical Information (21, 14, 7)**
  - Comment is missing critical semantic information.
- **Missing Task Elaboration (5, 2, 1)**
  - Did not describe what code was doing properly.
- **Missing Non-Critical Information (28, 19, 5)**
  - Useful comment but non-critical info missing.
- **Missing Web-Related Information (0, 1, 0)**
  - Comment failed to mention web-related identifier.
- **Failed to Mention Identifiers (0, 11, 6)**
  - Does not mention specific variable/attribute names, often using a generic identifier.
- **Missing Identifier (5, 3, 7)**
  - No identifier mentioned at all.
- **Missing Data Structure Information (2, 0, 1)**
  - Does not capture relevant data structure info
- **Missing Syntax Information (2, 6, 0)**
  - Important syntactic information (e.g. code ordering) is missing.
- **Missing Exception (1, 0, 0)**
  - Does not mention relevant exception info
- **Missing Conditional Information (1, 0, 0)**
  - Misses code branching information

Incorrect Semantic Information (6, 27, 19)
- **Partial Incorrect Information (6, 11, 3)**
  - Semantically meaningful, with a few errors.
- **Semantically Unrelated to Code (0, 11, 13)**
  - Does not capture code context whatsoever.
- **Algorithmically Incorrect (0, 5, 3)**
  - Conveys a different algorithmic meaning as compared to the code.

Incorrect Construction (26, 31, 53)
- **Incorrect Identifier/Attribute (5, 19, 15)**
  - Correctly identifies a variable or attribute, but uses it incorrectly.
- **Incomplete Sentence (1, 1, 10)**
  - Predicted comment is grammatically incomplete.
- **Repetition (0, 7, 27)**
  - Comment contains unnecessary repetition of a word or fragment between 2-3 times.
- **Extreme Repetition (0, 2, 1)**
  - Comment contains unnecessary repetition of a word or fragment more than 2-3 times.
- **Focusing Only on Method Name (20, 1, 0)**
  - When comment focuses mostly on the method name, which provides an incomplete but partial description of the functionality.
- **Grammatical Errors (0, 1, 0)**
  - Grammatical Error is present in predicted caption.

The numbers shown for each category illustrate the number of instances found for (**CodeBERT, NeuralCodeSum,** and **code2seq**) respectively

Consistent with Ground Truth (88, 57, 17)
- **Consistent with Specific Info (30, 15, 5)**
  - Comment matches ground truth well.
- **Consistent but Missing Specific Info (56, 35, 12)**
  - Comment matches ground truth mostly, but misses some important specific information.
- **Improves upon Semantic Meaning (2, 6, 0)**
  - The predicted comment matches the ground and improves capturing method meaning.
- **Consistent but with Unnecessary Info (0, 1, 0)**
  - Accurate but has some unnecessary info.

Extraneous/Unnessecary Information Included (2, 3, 4)
- **Unnecessary Data Structure Info (1, 0, 0)**
  - Adds unnecessary data structure info to comment.
- **Unnecessary File Information (0, 1, 1)**
  - Adds unnecessary file information to comment.
- **Unnecessary Incorrect Information (1, 2, 3)**
  - Adds information to comment that is both incorrect and unnecessary.

Over-Generalization (7, 21, 21)
- **Different Meaning (2, 3, 3)**
  - Comment over-generalizes on the meaning of the code functionality.
- **Algorithmically Incorrect (1, 6, 3)**
  - Overgeneralizes to the point of incorrectness
- **Missing Attribute Specification (4, 12, 15)**
  - Uses generic names such as **var.**

Conclusion

CodeBERT performs the best compared to other models based on reference-based metrics

Over-

Missing Information

Incorrect Construction

Future studies should also do similar kind of qualitative study