Avaliação de Modelos Neurais para Sumarização de Código-fonte

Leandro Baêta Lustosa Pontes[1], **Hilário Tomaz Alves de Oliveira**[1], **Francisco de Assis Boldt**[1]

[1]Programa de Pós-graduação em Computação Aplicada (PPComp), Instituto Federal do Espírito Santo (IFES), Serra, Brasil

leandroblpontes@gmail.com, {hilario.oliveira, franciscoa}@ifes.edu.br

Abstract. *Source code summarization is the task of automatically creating a natural language description from a snippet of source code. In recent years, several models based on Deep Learning algorithms have been proposed in the literature for this task. In this work, we performed a comparative analysis among four state-of-the-art neural models (CodeBERT, CodeT5, CodeTrans, and PLBART) using two databases commonly adopted for the Java programming language. The experimental results demonstrate that the CodeTrans model obtained the best performance based on different evaluation measures and that there is great variability in the descriptions generated by the evaluated models.*

Resumo. *Sumarização de código-fonte é a tarefa de criar automaticamente uma descrição em linguagem natural a partir de um trecho de código-fonte. Nos últimos anos, diversos modelos baseados em algoritmos de Aprendizado Profundo têm sido propostos na literatura para essa tarefa. Neste trabalho, realizamos uma análise comparativa entre quatro modelos neurais (CodeBERT, CodeT5, CodeTrans e PLBART) do estado da arte utilizando duas bases de dados comumente usadas para a linguagem de programação Java. Os resultados experimentais demonstram que o modelo CodeTrans obteve o melhor desempenho com base em diferentes medidas de avaliação e que existe uma grande variabilidade nas descrições geradas pelos modelos avaliados.*

1. Introdução

Segundo Sommerville [2011], um *software* pode ser definido como um programa de computador associado à sua documentação. Essa definição ressalta que a documentação gerada para descrever diferentes aspectos do *software* é um importante artefato criado ao longo do seu desenvolvimento. Em geral, os desenvolvedores passam aproximadamente 58% do seu tempo em tarefas relacionadas a compreensão das funcionalidades dos sistemas [Xia et al. 2018] e cerca de 90% do tempo é gasto em sua manutenção e evolução durante o seu ciclo de vida [Wan et al. 2018].

Apesar de a tarefa de documentação de *software* ser uma boa prática de programação e ser foco de diversos estudos que demonstram como uma documentação de qualidade pode facilitar o seu entendimento e manutenção, o processo de documentar o código-fonte de uma aplicação é muitas vezes negligenciado pelos programadores [Liu et al. 2020]. Em geral, a criação dessa documentação é feita manualmente, o que demanda tempo e esforço, fazendo com que essa atividade se torne um fardo para os desenvolvedores diante das constantes pressões por prazos de entregas cada vez mais curtos [Zhang et al. 2020].

Com o intuito de mitigar o problema da necessidade de esforço e tempo gastos para gerar uma documentação de *software* de qualidade, diversas pesquisas têm focado no desenvolvimento de sistemas capazes de gerar automaticamente descrições textuais a partir da codificação. Nesse cenário que surge a tarefa de sumarização automática de código-fonte, do inglês *Automatic Source Code Summarization*, que consiste na criação automática de um resumo em linguagem natural a partir da codificação ou de elementos que o compõem, como classes, métodos, funções, entre outros [Zhu and Pan 2019]. As descrições geradas por esses sistemas de sumarização podem auxiliar na compreensão e manutenção dos programas, pois os desenvolvedores podem entender rapidamente uma parte do código lendo sua descrição em linguagem natural.

Na Figura 1 é apresentado um exemplo de uma função escrita na linguagem de programação Java, uma descrição de referência criada manualmente por um ser humano e resumos gerados automaticamente pelos modelos de sumarização de código-fonte avaliados neste trabalho e apresentados na Seção 3.1.

Figura 1. Exemplos de descrições geradas pelos modelos de sumarização de código-fonte (CodeBERT, CodeT5, CodeTrans e PLBART).

```java
public static String cutToIndexOf(String string, final String substring) {
    int i = string.indexOf(substring);
    if (i != -1) {
        string = string.substring(0, i);
    }
    return string;
}
```

Modelo	Descrição
CodeBERT	Cut the given substring to the given string .
CodeT5	Cut the string to the first occurrence of the given substring.
PLBART	Cut to index of string.
CodeTrans Large	Cuts the string from the given index of the given substring.
Descrição humana	Cuts the string from beginning to the first index of provided substring.

Recentemente, com o advento dos algoritmos de Aprendizado Profundo (*Deep Learning*), diversos trabalhos têm explorado a aplicação de diferentes arquiteturas baseadas em redes neurais profundas para a tarefa de sumarização de código-fonte [Wan et al. 2018, Liu et al. 2020, Zhang et al. 2020]. Em geral, esses modelos neurais buscam aprender o alinhamento entre o código-fonte e a sua descrição em linguagem natural, de maneira similar aos sistemas de tradução automática de texto. Dada a grande diversidade de trabalhos existentes na literatura, é difícil realizar uma comparação direta entre os sistemas propostos visto que eles adotam ambientes experimentais diferentes. Aspectos como a configuração da base de dados utilizada nos experimentos, as medidas de avaliação adotadas e as configurações dos modelos podem afetar uma comparação justa entre as diferentes abordagens.

Neste trabalho, realizamos uma análise comparativa entre quatro modelos do es-

tado da arte identificados na literatura para a tarefa de sumarização de código-fonte, sendo eles: CodeBERT [Feng et al. 2020], CodeTrans [Elnaggar et al. 2021], CodeT5 [Wang et al. 2021] e PLBART [Ahmad et al. 2021]. Os modelos avaliados geram uma breve descrição (resumo) em linguagem natural a partir do código-fonte de uma função. Este trabalho foca em códigos escritos na linguagem de programação Java por ser uma das mais utilizadas nos trabalhos identificados na literatura. Diversos experimentos foram realizados em um mesmo ambiente experimental usando as bases de dados propostas por Hu et al. [2018] e Lu et al. [2021], com base nas medidas de avaliação do ROUGE-L [Lin 2004], METEOR [Banerjee and Lavie 2005] e BLEU-4 suavizado [Lin and Och 2004]. Duas configurações de pré-processamento do código-fonte foram avaliadas e o impacto que essas variações geraram nas descrições foram analisados. Além disso, foi realizada uma análise da similaridade entre as descrições produzidas pelos modelos utilizando o índice de similaridade de Jaccard. A implementação e os dados usados em nossos experimentos estão disponíveis em[1].

2. Trabalhos Relacionados

Os primeiros trabalhos na área de sumarização de código-fonte utilizavam técnicas de Processamento de Linguagem Natural (PLN) e heurísticas, por exemplo, explorando a frequência com que palavras eram citadas no código ou o tradicional método de *Term Frequency–Inverse Document Frequency* (TF-IDF). O trabalho de Haiduc et al. [2010] foi um dos percursores na utilização de técnicas de PLN para sumarização de código-fonte. Rodeghero et al. [2014] evoluíram o trabalho desenvolvido por Haiduc et al. [2010], sendo incorporada à análise do movimento dos olhos dos programadores, onde eles primeiro analisavam as assinaturas dos métodos para depois explorar a sua implementação. Sridhara et al. [2010] empregaram técnicas de PLN para obter informações semânticas sobre os métodos analisados, por exemplo, comentários feitos no bloco de código eram utilizados para melhorar a sumarização final.

Com o surgimento de diversas arquiteturas baseadas em modelos de Aprendizado Profundo, especialmente aquelas capazes de geração de linguagem, a maioria dos trabalhos da área de sumarização de código-fonte tem explorado diferentes tipos de redes neurais profundas devido aos resultados do estado da arte obtidos [Zhu and Pan 2019]. Iyer et al. [2016] foram pioneiros no uso de redes neurais para sumarização de código-fonte, sendo utilizada uma arquitetura do tipo *Long Short-Term Memory* (LSTM) integrada com um modelo de atenção para melhorar o processo de geração dos resumos.

Hu et al. [2018] adotaram um modelo do tipo sequência para sequência, do inglês *Sequence-to-Sequence* (Seq2Seq), composto por dois módulos: um codificador e um decodificador. Os autores propõem o uso da Árvore Sintática Abstrata, do inglês *Abstract Syntax Tree (AST)* na etapa de treinamento do codificador e apresentam um novo método de travessia, chamado de *Structure-Based Traversal* (SBT). A hipótese dos autores é que a estrutura da AST pode agregar informações estruturais do código e auxiliar na geração de uma melhor descrição.

Segundo Zhu and Pan [2019], as Redes Neurais Recorrentes, do inglês *Recurrent Neural Network* (RNN), são as mais utilizadas para a tarefa de sumarização de código-fonte pelo bom desempenho obtido pelos sistemas que a adotam. Outro destaque é o uso

[1]https://github.com/laicsiifes/code_summarization

dos modelos do tipo *Sequence-to-sequence* (Seq2seq) que são comumente adotados em abordagens de tradução automática de texto e logo foram adaptados por diversos autores para a tarefa de sumarização de código-fonte.

3. Materiais e Métodos

Nesta seção são apresentados os modelos de sumarização de código-fonte que foram analisados neste trabalho (Seção 3.1) e as bases de dados adotadas nos experimentos (Seção 3.2).

3.1. Modelos Avaliados

Os seguintes modelos de sumarização de código-fonte do estado da arte foram avaliados.

CodeBERT [Feng et al. 2020] é um modelo que aprende representações de uso geral, dando suporte as tarefas que envolvem linguagens de programação e linguagem natural, por exemplo, sumarização de código-fonte, pesquisa de código em linguagem natural, entre outras. O CodeBERT é baseado na arquitetura *Transformer* [Vaswani et al. 2017] de múltiplas camadas sendo treinado com uma função objetivo híbrida que combina as tarefas de modelagem de linguagem mascarada, do inglês *Masked Language Modeling* (MLM), e de detecção de token substituído, do inglês *Replaced Token Detection* (RTD). Quando aplicado a tarefa de sumarização de código o modelo CodeBERT requer um decodificador, que precisa ser treinado especificamente para essa tarefa. Para isso, o CodeBERT recebe como entrada a sequência de tokens do código-fonte e quando treinado conjuntamente com um decodificador gera como saída uma descrição em linguagem natural. O código e as bases de dados usadas pelo trabalho original do CodeBERT estão disponíveis em[2].

CodeT5 [Wang et al. 2021] é um modelo do tipo de codificador e decodificador que incorpora informações do tipo dos tokens do código-fonte. O CodeT5 foi construído com base na arquitetura T5 [Raffel et al. 2020] que adota um pré-treinamento do tipo Sequência para Sequência (*Seq2Seq*), que tem demonstrado bons resultados na literatura tanto nas tarefas de compreensão de texto quanto nas de geração de linguagem natural. O CodeT5 busca capturar a semântica do código-fonte a partir dos identificadores utilizados pelos desenvolvedores para nomear os diferentes elementos que compõe a prática de codificação, por exemplo, variáveis, métodos, classes, interfaces, entre outros. Para isso, os autores propõem uma nova tarefa de pré-treinamento para o reconhecimento de identificadores contidos no código que permite o modelo recuperá-los quando esses são mascarados. Na tarefa de sumarização de código-fonte, o CodeT5 recebe os tokens do código-fonte de entrada e gera uma descrição em linguagem natural de saída. O CodeT5 está disponível em[3].

CodeTrans [Elnaggar et al. 2021] é um modelo do tipo codificador e decodificador proposto para diferentes tarefas envolvendo linguagem natural e linguagem de programação, dentre elas a sumarização de código-fonte, a geração de documentação, entre outras. O CodeTrans é baseado na arquitetura T5 [Raffel et al. 2020] e diversos modelos foram gerados explorando as seguintes estratégias de treinamento: aprendizado de única

[2]https://github.com/microsoft/CodeBERT
[3]https://github.com/salesforce/CodeT5

tarefa (*single-task*), aprendizado por transferência (*transfer-learning*), multitarefa (*multi-task*) e multitarefa com ajuste fino (*multi-task with fine-tuning*). Os diversos modelos pré-treinados e informações sobre as bases de dados usadas podem ser encontradas em[4].

PLBART [Ahmad et al. 2021] é um modelo do tipo sequência para sequência (*Seq2Seq*) capaz de realizar diversas tarefas de geração e compreensão envolvendo linguagem natural e linguagens de programação. O PLBART é baseado no modelo BART [Lewis et al. 2020] que é um codificador automático de redução de ruído que usa um codificador bidirecional e um decodificador autorregressivo. O PLBART é pré-treinado em uma extensa coleção de funções desenvolvidas nas linguagens de programação Java e Python, e associadas com as descrições dessas funções em linguagem natural. Três estratégias para a redução de ruído são usadas no treinamento do PLBART, sendo elas: mascaramento de tokens, exclusão de tokens e preenchimento de tokens. O código-fonte do PLBART e as bases de dados usadas nos seus experimentos podem ser encontradas em[5].

3.2. Bases de Dados

Os modelos avaliados e apresentados na Seção 3.1 possuem suporte a diferentes linguagens de programação, sendo Java e Python as linguagens em comum entre todos os modelos. Neste trabalho focaremos na linguagem Java pela diversidade de bases de dados disponíveis na literatura. Duas bases propostas por Hu et al. [2018] e Lu et al. [2021] foram adotadas nos experimentos realizados. Essas bases de dados apresentam exemplos de códigos-fontes de funções em conjunto com uma descrição em linguagem natural.

A base de dados proposta por Hu et al. [2018] foi coletada a partir de 9.714 projetos de código aberto disponíveis na plataforma do Github[6]. Essa base é composta por códigos-fontes de funções escritas na linguagem Java e cada uma possui uma descrição em linguagem natural gerada usando a primeira frase na documentação do Javadoc. Diversos trabalhos na literatura [Elnaggar et al. 2021, Yang et al. 2021] têm usado essa base de dados e sua versão original pode ser encontrada em[7].

O CodeXGLUE [Lu et al. 2021] é uma base de dados de referência criada para impulsionar pesquisas envolvendo a área de Aprendizado de Máquina para tarefas de compreensão e geração de linguagens de programação e linguagem natural. O CodeXGLUE fornece suporte a 10 tarefas (sumarização de código-fonte, detecção de falhas, identificação de cópias, entre outros) a partir de 14 conjuntos de dados e uma plataforma para a avaliação e a comparação de modelos. Para a tarefa de sumarização de código-fonte, a base de dados foi construída usando o CodeSearchNet [Husain et al. 2019] que possui dados disponíveis para seis linguagens de programação (Python, Java, PHP, JavaScript, Ruby e Go) para diversas tarefas.

Após uma análise manual das duas bases de dados, foi identificada a necessidade de realizar alguns ajustes em ambas as bases. Inicialmente, identificou-se a presença de exemplos (código-fonte e descrição) duplicados entre os conjuntos de treinamento, validação e teste na base proposta por Hu et al. [2018]. Por isso, nas duas bases foi

[4]https://github.com/agemagician/CodeTrans
[5]https://github.com/wasiahmad/PLBART
[6]https://github.com/
[7]https://github.com/xing-hu/DeepCom

realizada uma filtragem para a remoção dos exemplos duplicados. Em ambas as bases de dados foram identificados códigos-fontes e/ou descrições muito pequenas, em alguns casos com apenas uma palavra, ou muito grandes, por exemplo, códigos com mais de mil tokens. Por isso, foram removidos exemplos contendo: **(i)** códigos-fontes com menos do que cinco ou mais de trezentos tokens; e **(ii)** descrições com menos do que quatro ou mais do que cem palavras. Além disso, foram removidos exemplos com descrições contendo caracteres escritos em outro idioma que não a língua inglesa.

A Tabela 1 apresenta estatísticas descritivas das duas bases de dados após as filtragens realizadas neste trabalho. As estatísticas geradas são: total de exemplos nos conjuntos de treinamento, validação e teste, e para cada conjunto foi verificada a quantidade média de *tokens* presentes nos códigos-fontes e de palavras nas descrições. Para a geração das estatísticas os códigos-fontes foram fragmentados em tokens utilizando a ferramenta javalang[8] e as descrições foram divididas em palavras usando a ferramenta spaCy[9].

Tabela 1. Estatísticas descritivas das bases de dados utilizas.

Base de dados	Divisão	Exemplos	Código-fonte	Descrição
Hu et al. [2018]	**Treinamento**	51.759	80,04	12,24
	Validação	7.366	79,84	12,17
	Teste	7.769	78,63	12,13
CodeXGLUE	**Treinamento**	158.155	68,24	11,92
	Validação	4.892	66,92	11,64
	Teste	10.517	66,57	11,26

4. Experimentos

Experimentos foram realizados para analisar as seguintes questões: **(i)** uma comparação entre os modelos CodeBERT, CodeT5, CodeTrans e PLBART usando duas configurações de pré-processamento do código-fonte (Seção 4.2); e **(ii)** uma análise da similaridade das descrições geradas pelos modelos avaliados (Seção 4.3). Antes de apresentar e discutir os resultados obtidos, uma breve apresentação do ambiente experimental e alguns detalhes de implementação são apresentados na próxima seção.

4.1. Ambiente Experimental

Implementação dos Modelos. Neste trabalho não foi realizada nenhuma etapa de treinamento ou ajuste fino dos modelos avaliados. Por isso, utilizamos somente o conjunto de testes das bases de dados. Foram utilizadas as implementações dos modelos CodeBERT, CodeT5, CodeTrans e PLBART pré-treinados e disponibilizadas na plataforma HuggingFace[10]. Como comentado na Seção 3.1, o modelo CodeBERT possui apenas o módulo codificador pré-treinado, sendo necessário incluir o decodificador e realizar o treinamento para a tarefa de sumarização de código-fonte. Para isso, utilizamos um decodificador baseado na arquitetura (*Transformer*) que foi pré-treinado e disponibilizado em um *AWS S3 Bucket*[11]. O CodeTrans possui diversos modelos pré-treinados disponíveis, avaliamos as

[8]https://github.com/c2nes/javalang
[9]https://spacy.io/
[10]https://huggingface.co/
[11]https://code-summary.s3.amazonaws.com/pytorch_model.bin

versões pequena (*small*), base e grande (*large*) treinados usando a estratégia de aprendizado multitarefa com ajuste fino para a tarefa de geração de documentação de código.

Medidas de avaliação. Para avaliar os resumos gerados pelos modelos investigados foram utilizadas as medidas do ROUGE-L [Lin 2004], METEOR [Banerjee and Lavie 2005] e BLEU-4 suavizada [Lin and Och 2004]. Essas medidas comparam a descrição gerada automaticamente com resumos de referência disponíveis nas bases de dados adotadas. Os valores gerados são normalizados e quanto mais próximos de $1,0$ (100%) mais similar é o resumo criado em relação ao de referência. A medida *Metric for Evaluation of Translation with Explicit ORdering* (METEOR) [Banerjee and Lavie 2005] computa a sobreposição de palavras (unigramas) entre a descrição de referência e a gerada automaticamente. Foi utilizada a implementação da medida disponibilizada na ferramenta *Natural Language Toolkit* (NLTK)[12]. A medida *Recall-Oriented Understudy for Gisting Evaluation Longest Common Subsequence* (ROUGE-L) [Lin 2004] verifica a maior cadeia de n-gramas em comum entre a descrição de referência e a gerada automaticamente. Utilizamos a implementação da medida Rouge-L disponibilizada em[13]. Por fim, a medida *Bilingual Evaluation Understudy* (BLEU) [Lin and Och 2004] computa a média geométrica entre as medidas de precisão das sobreposições de n-gramas (1-grama, 2-grama, 3-grama e 4-grama) entre a referência e a descrição candidata. Quando os textos de referência e os candidatos são muito curtos, é possível que não ocorra nenhuma sobreposição de 4-gramas, resultando em um valor zero na média geométrica. Por isso, é comum a utilização de métodos de suavização em conjunto com a medida do BLEU-4. Para computar essa medida foi utilizada a implementação disponível em[14].

4.2. Avaliação dos Modelos

Neste primeiro experimento, foi realizada uma análise do desempenho dos modelos CodeBERT, CodeT5, CodeTrans (Small, Base e Large) e PLBART. Todos os modelos recebem como entrada a sequência de tokens do código-fonte extraída usando a ferramenta $javalang$. Uma etapa de normalização realizada durante a geração dos tokens dos códigos foi a substituição dos literais (*String*, *Integer*, *Float* e *Bool*) por nomes genéricos (*STR*, *NUM* e *BOOL*) para diminuir o tamanho do vocabulário [Hu et al. 2018].

Duas configurações de pré-processamento nos nomes dos identificadores presentes no código-fonte foram aplicadas. A primeira configuração mantém os nomes da maneira original como eles aparecem, enquanto na segunda configuração foi realizada uma fragmentação dos identificadores formados por palavras compostas com base nas seguintes regras: **(i)** identificadores cujas primeiras letras das palavras compostas começam com letra maiúsculas (*Camel Case*), por exemplo, $getFile$ se transforma em dois tokens get e $file$; e **(ii)** identificadores cujas palavras compostas são separadas pelo caractere de $underscore$ (*Snake Case*), por exemplo, $remove\_element$ se transforma em dois tokens $remove$ e $element$. Essa divisão no nome dos identificadores é uma prática comum observada em alguns trabalhos [Hu et al. 2018] e tem por objetivo diminuir o problema de *Out-Of-Vocabulary* (OOV) que acontece quando surgem novos tokens que não estavam presentes nos exemplos usados durante o treinamento dos modelos.

[12]https://www.nltk.org/

[13]https://github.com/Diego999/py-rouge

[14]https://github.com/microsoft/CodeBERT/blob/master/CodeBERT/code2nl/bleu.py

Na Tabela 2 são apresentados os resultados (%) deste experimento com base nas medidas do ROUGE-L (RL), METEOR (M) e BLEU-4 suavizada (B4). É possível observar que a estratégia de fragmentação dos nomes dos identificadores usando as regras do *Camel* e *Snake Case* (CSC) melhorou os resultados dos modelos em quase todos os cenários comparados. A única exceção foi na base de dados de [Hu et al. 2018] usando o modelo $CodeTrans_{Large}$. Analisando os cenários de melhoria no desempenho nota-se diferenças mínimas de menos de 0,04% até maiores do que 3,50% dependendo da medida de avaliação utilizada. Isso demonstra que adotar a estratégia de CSC é recomendada para melhorar o desempenho dos modelos na linguagem de programação Java.

Tabela 2. Resultados dos experimentos (%) para avaliação dos modelos. O melhor resultado em cada medida e base de dados é destacado em negrito.

Modelos	Pré-proc.	Hu et al. [2018]			CodexGlue		
		RL	M	B4	RL	M	B4
CodeBERT	-	37,25	22,63	16,16	37,80	19,52	16,46
	CSC	39,23	24,01	16,86	40,63	21,03	17,68
CodeT5	-	38,07	24,31	15,09	39,09	26,50	16,46
	CSC	39,88	25,38	15,89	41,29	**27,93**	17,40
PLBART	-	36,16	19,52	15,83	36,16	19,52	15,83
	CSC	39,80	21,58	17,16	39,80	21,58	17,16
$CodeTrans_{Small}$	-	40,71	20,80	18,06	41,53	23,58	19,47
	CSC	43,00	21,99	18,90	**44,73**	25,50	**20,51**
$CodeTrans_{Base}$	-	43,42	23,86	20,38	40,73	23,81	19,34
	CSC	44,37	23,66	20,51	43,47	24,59	20,25
$CodeTrans_{Large}$	-	**48,47**	**30,41**	**25,62**	39,38	24,23	17,85
	CSC	48,43	30,04	25,50	40,50	24,80	18,47

De maneira geral, existe uma grande variabilidade nos resultados obtidos pelos modelos avaliados. Na base de dados de Hu et al. [2018], o modelo $CodeTrans_{Large}$ obteve o melhor desempenho nas três medidas de avaliação, enquanto na base CodexGlue [Lu et al. 2021] o $CodeTrans_{Small}$ alcançou melhores resultados nas medidas do ROUGE-L e BLEU-4, e o modelo CodeT5 apresentou o melhor desempenho na medida do METEOR.

4.3. Avaliação da Similaridade das Descrições

No experimento apresentado na Seção 4.2 observou-se uma grande diversidade no desempenho dos modelos (CodeBERT, CodeT5, CodeTrans (*Small*, *Base* e *Large*) e PLBART) com base nas medidas de avaliação comumente adotadas na tarefa de sumarização de código-fonte. Contudo, não é possível verificar o quão distintos são os resumos gerados apenas analisando essas medidas. Por isso, neste segundo experimento, analisamos a similaridade das descrições geradas pelos modelos avaliados visando verificar de forma quantitativa quais deles geram resumos mais semelhantes e quais produzem descrições mais distintas. Essa análise permite verificar modelos que geram resumos complementares a outros.

O cálculo da similaridade entre as descrições geradas foi realizado utilizando o coeficiente de similaridade de $Jaccard$. Dada duas descrições geradas pelos modelos A_C e B_C para o mesmo código-fonte C, então o coeficiente de $Jaccard$ é definido pela interseção dos conjuntos $|A_C \cap B_C|$ dividido pelo tamanho da união dos dois conjuntos $|A_C \cup B_C|$, ou seja:

$$J(A_C, B_C) = \frac{|A_C \cap B_C|}{|A_C \cup B_C|}$$

As descrições A_C e B_C foram pré-processadas utilizando as técnicas de lematização das palavras e remoção de *stopwords* usando a ferramenta *spaCy* para diminuir o tamanho do vocabulário e aumentar a cobertura da verificação de similaridade.

Na Tabela 3 e na Tabela 4 são apresentadas as médias das similaridades obtidas, com três casas decimais e arredondamento, das descrições geradas pelos modelos avaliados nas bases de dados do Codexglue [Lu et al. 2021] e de Hu et al. [2018] respectivamente. De maneira geral, os cenários com maiores índices de similaridade são observados comparando as descrições geradas pelos modelos sem o pré-processamento $(-)$ nos nomes dos identificadores e os resultados com a fragmentação usando *Camel* e *Snake Case* (CSC). Esse comportamento já era esperado por se tratar do mesmo modelo. No entanto, nota-se em sua maioria um índice inferior a 60%, o que demonstra que utilizar a estratégia CSC faz com que os modelos gerem, em geral, descrições que são 40% diferentes. Os maiores índices de similaridade foram obtidos na comparação das descrições geradas pelas diferentes versões (*Small*, *Base* e *Large*) do modelo $CodeTrans$.

Tabela 3. Índices de similaridade entre os modelos no Codexglue [Lu et al. 2021].

Modelos	Pré	CodeBERT	CodeT5		PLBART		$CodeTrans_{Small}$		$CodeTrans_{Base}$		$CodeTrans_{Large}$	
	Proc.	CSC	-	CSC	-	CSC	-	CSC	-	CSC	-	CSC
CodeBERT	-	0,5684	0,3457	0,3587	0,3099	0,3516	0,3845	0,4099	0,3568	0,3749	0,3188	0,3254
	CSC		0,3526	0,4303	0,3154	0,4247	0,3955	0,5084	0,3702	0,4648	0,3424	0,4013
CodeT5	-			0,5516	0,3421	0,3709	0,4033	0,4080	0,3978	0,3865	0,3621	0,3499
	CSC				0,3397	0,4542	0,3994	0,5112	0,3955	0,4813	0,3724	0,4304
PLBART	-					0,4376	0,3556	0,3613	0,3328	0,3373	0,3000	0,3010
	CSC						0,3932	0,5060	0,3752	0,4677	0,3494	0,4072
$CodeTrans_{Small}$	-							0,6015	0,4934	0,4535	0,3875	0,3631
	CSC								0,4638	0,5886	0,3977	0,4710
$CodeTrans_{Base}$	-									0,5531	0,4093	0,3793
	CSC										0,4063	0,4767
$CodeTrans_{Large}$	-											0,5309

Foi observado também que os valores dos desvios padrões para ambas as bases de dados foram altos ficando entre $0,224$ e $0,343$. Esses valores de desvios padrões indicam que os modelos estão gerando descrições com uma alta diversidade. Comparando as descrições geradas entre os diferentes modelos, observa-se que os modelos CodeBERT e PLBART apresentam uma similaridade mais alta com o modelo $CodeTrans_{Small}$ para ambas as bases de dados avaliadas considerando o pré-processamento CSC. Já o menor coeficiente de similaridade foi observado entre os modelos PLBART e $CodeTrans_{Large}$ com ou sem o pré-processamento CSC em ambas as bases de dados.

5. Considerações Finais e Trabalhos Futuros

Neste trabalho foi realizada uma análise comparativa entre quatro modelos neurais do estado da arte para a tarefa de sumarização de código-fonte. Os modelos CodeBERT,

Tabela 4. Índices de similaridade entre os modelos na base de dados de Hu et al. [2018].

Modelos	Pré	CodeBERT	CodeT5		PLBART		$CodeTrans_{Small}$		$CodeTrans_{Base}$		$CodeTrans_{Large}$	
	Proc.	CSC	-	CSC	-	CSC	-	CSC	-	CSC	-	CSC
CodeBERT	-	0,5736	0,3419	0,3592	0,3109	0,3528	0,3923	0,4115	0,3513	0,3699	0,3174	0,3250
	CSC		0,3531	0,4268	0,3205	0,4162	0,4100	0,5000	0,3751	0,4464	0,3489	0,3910
CodeT5	-			0,5450	0,3376	0,3673	0,4068	0,4105	0,3867	0,3837	0,3485	0,3445
	CSC				0,3345	0,4402	0,4096	0,5027	0,3945	0,4698	0,3697	0,4176
PLBART	-					0,4390	0,3606	0,3636	0,3324	0,3354	0,3004	0,3022
	CSC						0,3982	0,4893	0,3757	0,4472	0,3525	0,3969
$CodeTrans_{Small}$	-							0,6057	0,4959	0,4552	0,3921	0,3735
	CSC								0,4655	0,5685	0,4075	0,4634
$CodeTrans_{Base}$	-									0,5627	0,4325	0,4027
	CSC										0,4245	0,4827
$CodeTrans_{Large}$	-											0,5223

CodeT5, PLBART e três versões (*Small*, *Base* e *Large*) do CodeTrans foram avaliados usando as bases de dados apresentadas em Hu et al. [2018] e Lu et al. [2021] para a linguagem de programação Java. Dois experimentos foram realizados buscando analisar: **(i)** as descrições geradas pelos modelos aplicando duas configurações de pré-processamento com base nas medidas do ROUGE-L, METEOR e BLEU-4 suavizado que são tradicionalmente usadas na literatura; e **(ii)** as similaridades entre as descrições geradas pelos modelos.

Com base nos resultados experimentais obtidos pode-se concluir que: **(i)** fragmentar os nomes dos identificadores usando os padrões *Camel case* e *Snake case* (CSC) melhorou o desempenho dos modelos avaliados em quase todos os cenários avaliados; **(ii)** de maneira geral, em ambas as bases de dados, o modelo $CodeTrans$ apresentou desempenho superior aos demais modelos analisados; e **(iii)** existe uma grande diversidade nas descrições geradas.

Como trabalhos futuros, pretendemos: **(i)** expandir os experimentos para outras bases de dados em diferentes linguagens de programação; **(ii)** realizar o processo de ajuste fino (*fine-tuning*) dos modelos; e **(iii)** investigar estratégias para a combinação dos resumos gerados pelos modelos, visto que eles produziram conteúdos diversos, o que sugere que descrições com informações complementares podem ser criadas quando diferentes modelos forem combinados.

Agradecimentos

Os autores agradecem ao Ifes, apoio da FAPES e CAPES (processo 2021-2S6CD, nº FAPES 132/2021) por meio do PDPG (Programa de Desenvolvimento da Pós-Graduação, Parcerias Estratégicas nos Estados).

Referências

Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. (2021). Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Banerjee, S. and Lavie, A. (2005). METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Elnaggar, A., Ding, W., Jones, L., Gibbs, T., Feher, T., Angerer, C., Severini, S., Matthes, F., and Rost, B. (2021). Codetrans: Towards cracking the language of silicone's code through self-supervised deep learning and high performance computing. *CoRR*, abs/2104.02443.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Haiduc, S., Aponte, J., Moreno, L., and Marcus, A. (2010). On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44. IEEE.

Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. (2018). Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 200–210, New York, NY, USA. Association for Computing Machinery.

Husain, H., Wu, H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

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 (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

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*, pages 7871–7880, Online. Association for Computational Linguistics.

Lin, C.-Y. (2004). Rouge: A package for automatic evaluation of summaries. In Marie-Francine Moens, S. S., editor, *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Lin, C.-Y. and Och, F. J. (2004). ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING.

Liu, S., Chen, Y., Xie, X., Siow, J. K., and Liu, Y. (2020). Automatic code summarization via multi-dimensional semantic fusing in gnn. *arXiv preprint arXiv:2006.05405*.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., GONG, M.,

Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and LIU, S. (2021). CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Rodeghero, P., McMillan, C., McBurney, P. W., Bosch, N., and D'Mello, S. (2014). Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 390–401, New York, NY, USA. Association for Computing Machinery.

Sommerville, I. (2011). *Engenharia de software*. Pearson Prentice Hall.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., and Vijay-Shanker, K. (2010). Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 43–52, New York, NY, USA. Association for Computing Machinery.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., and Yu, P. S. (2018). Improving automatic source code summarization via deep reinforcement learning. *CoRR*, abs/1811.07234.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., and Li, S. (2018). Measuring program comprehension: A large-scale field study with professionals. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 584, New York, NY, USA. Association for Computing Machinery.

Yang, G., Chen, X., Cao, J., Xu, S., Cui, Z., Yu, C., and Liu, K. (2021). Comformer: Code comment generation via transformer and fusion method-based hybrid code representation. In *8th International Conference on Dependable Systems and Their Applications, DSA 2021, Yinchuan, China, August 5-6, 2021*, pages 30–41. IEEE.

Zhang, J., Wang, X., Zhang, H., Sun, H., and Liu, X. (2020). Retrieval-based neural source code summarization. In *Proceedings of the 42nd International Conference on Software Engineering. IEEE*.

Zhu, Y. and Pan, M. (2019). Automatic code summarization: A systematic literature review. *ArXiv*, abs/1909.04352.