Análise Experimental de Abordagens para Sumarização de Código Fonte

Gustavo Pereira Flores
Escola Politécnica
Pontifícia Universidade Católica (PUCRS)
Porto Alegre, Brasil
gustavo.pereira87@edu.pucrs.br

***Resumo*—A geração automática de documentação de código-fonte é uma tarefa desafiadora dentro do campo de Processamento de Linguagem. Os modelos de linguagem baseados em Neural Machine Translation (NMT) não apresentam bons resultados quando aplicados sumarização, e a literatura da área tem dificuldade em criar um corpus de treino amplo e padronizado para comparação de resultados. Uma das contribuições deste trabalho é realizar uma análise experimental das abordagens produzidas na literatura de sumarização de código-fonte utilizando o corpus de treino CoDesc, uma combinação de diversos conjuntos de dados da área, que resulta 4 milhões de pares de descrição e código-fonte em Java. Avaliaremos duas abordagens recentes que constituem o estado da arte para a tarefa de sumarização de código, *CodeBERT* e *NeuralCodeSum*, ambas baseadas em arquiteturas do tipo Transformer. Os resultados mostram que a rede *CodeBERT* supera a *NeuralCodeSum*, e que ambas performam melhor que a base comparativa, uma rede RNN com camadas de atenção.**

***Palavras-Chave*—Sumarização, Geração Automática, Código-Fonte, NL-PL**

I. Introdução

A maior parte do tempo de trabalho de um desenvolvedor de software se resume a fazer manutenção em códigos de programação já escritos, sendo indispensável que os métodos contidos dentro de um código-fonte sejam bem documentados. Entretanto, criar documentação é uma tarefa onerosa, que precisa ser realizada manualmente pelo desenvolvedor.

A evolução do campo de linguagem natural ofereceu oportunidade para que os modelos de linguagem fossem utilizados para processar também códigos de programação. O processamento conjunto de linguagens naturais e de programação é uma área de pesquisa interessada em tarefas como sumarização de código-fonte, geração automática ou busca de código-fonte a partir de descrições de linguagem natural.

```
// Source-Code
public int compare To(Object other) {
   @ Suppress Warnings(STRING) Ranking < V > other
       Ranking = (Ranking < V > ) other;
   return Double.compare(other Ranking.rank Score,
       rank Score);
}

// Description
Compares two ranking based on the rank score.
```

Exemplo 1. Código-fonte e descrição de um método

Abordaremos a evolução das arquiteturas propostas para a tarefa de sumarização de código-fonte, bem como dos conjuntos de dados disponibilizados publicamente para treino desses modelos. Atualmente não há um conjunto de dados de referência para esta tarefa, como ocorre em outras áreas do processamento da linguagem natural. Os modelos são treinados em diferentes conjuntos de dados, não permitindo a existência de uma análise comparativa unificada para validação de hipóteses de pesquisa.

O primeiro passo do nosso trabalho é estabelecer um conjunto de dados abrangente, capaz de oferecer dados limpos e livres de ruídos. Posteriormente, buscamos implementar duas abordagens recentes do estado da arte que foram treinadas inicialmente em conjuntos de dados diferentes, para extrair resultados destas sob as mesmas premissas.

Avaliando as diferentes redes no conjunto de dados de teste, obtivemos um resultado da métrica BLUE-4 de 15,29 para a rede *CodeBERT*, enquanto que a rede *NeuralCodeSum* registrou 14,69. Ainda, uma rede RNN com camadas de atenção foi estabelecida como nossa base comparativa, atingindo o score de 13,04 sob a mesma métrica e submetida aos mesmos dados [1].

II. Trabalhos Relacionados

Nosso enfoque é a geração automática de sumarização de código-fonte, tarefa que ganhou bastante importância na literatura a partir de 2015. LeClair [1] argumenta sobre as limitações dos modelos de sumarização inspirados em NMT. Diferentemente das linguagens naturais, códigos-fonte costumam ser mais longos e contêm muito menos palavras que possam ser mapeadas diretamente para suas descrições. Ainda, o código-fonte não é uma sequência de palavras, mas sim um conjunto de componentes que se relacionam de forma complexa.

A. Code-NN

No artigo pioneiro da geração de documentação de código-fonte, Iyer at al. [2] treinaram o *Code-NN*, um modelo LSTM combinado com camadas de atenção, utilizando *tokens* como representações de texto. Esse modelo foi treinado em 66.015 pares de código-fonte e descrição para a linguagem C#, e 32.337 pares para SQL retirados do *StackOverflow*.

[1]Todos os códigos estão disponíveis em https://github.com/gpfl/tc-codesum

B. Code2Seq

Adaptando a abordagem *seq2seq* dos modelos de Neural Machine Translation (NMT), o modelo *code2seq* [3] atingiu o estado da arte na tarefa de documentação de código-fonte. O *code2seq* segue a arquitetura *encoder-decoder* dos NMT, mas seu *encoder* cria uma representação vetorial do código que caracteriza um caminho dentro de uma estrutura de árvore sintática abstrata — *abstract syntax tree* (AST), em inglês. O *decoder* da rede, através do uso de camadas de atenção, seleciona os caminhos relevantes da AST enquando gera a sequência de saída.

Os autores treinaram o *code2seq* em três conjuntos de dados de tamanhos diferentes: *java-small, java-med e java-large*. O primeiro destes, *java-small*, é um conjunto de onze projetos em Java, com 700 mil métodos descritos, onde nove destes projetos são utilizados para treinar o modelo, enquanto que os outros dois são divididos entre validação e teste.

O conjunto *java-med* compreende os mil projetos Java mais bem avaliados na plataforma *GitHub*. Estes projetos totalizam quatro milhões de métodos e suas descrições. A divisão entre treino, validação e teste segue a proporção 8:1:1.

O maior de todos os conjuntos de dados, o *java-large* é composto pelos 9500 projetos mais bem avaliados no *GitHub*, totalizando 16 milhões de exemplos. Para treinar o modelo, são utilizados 9000 projetos. Se dividem igualmente os dados de validação e treino, sendo 250 para cada um.

C. AST e Redes Neurais para Grafos

Para LeClair [1], a abordagem utilizada por autores anteriores [3] é vista como uma limitação das árvores AST. Ao utilizar representações sequenciais para caracterizar os caminhos dentro da estrutura AST, os modelos não são efetivos em utilizar toda a informação fornecida pela árvore sintética.

O autor então propõe, dentro do contexto de sumarização de código-fonte, uma arquitetura combinando estruturas AST com Redes Neurais para Grafos (GNN). Para modelar as AST, a abordagem utiliza o *encoder* baseado em GNN do modelo *graph2seq*, apresentado por [4] para a tarefa de geração de linguagem natural. Ainda, é combinado a este um segundo *encoder* baseado em Redes Neurais Recorrentes (RNN) para modelar as AST como representação vetorial sequencial. Esses dois resultados são então passados para um mecanismo de atenção antes de serem reconstruídos pelo *decoder*. Esse modelo foi treinado com 2,1 milhões de pares de documentação e código-fonte do conjunto de dados *FunCom*.

D. CodeBERT

O modelo *CodeBERT* [5] se diferencia dos trabalhos anteriores por adotar uma arquitetura *Transformer*, até então explorada apenas em modelos de linguagem. Outra característica diferenciadora deste trabalho é o uso de modelos pré-treinados em linguagem natural. O *CodeBERT* se baseia em modelos *BERT* [6] e *RoBERTa* [7], e usa um *Transformer* bidirecional multicamadas. Além disso, o modelo é treinado com uma função objetivo híbrida, incluindo as técnicas de *Masked Language Modeling* (MLM) e *Replaced Token Detection* (RTD).

Os autores treinaram o *CodeBERT* para as tarefas de busca de código-fonte e documentação de código-fonte. Para treino do modelo foram utilizados 2,1 milhões de dados do CodeSearchNet [8], conjunto de dados composto por 6 linguagens de programação (Python, Java, JavaScript, PHP, Ruby, e Go).

E. NeuralCodeSum

Um segundo modelo utilizando *Transformers* foi treinado por Ahmad [9] utilizando um mecanismo de cópia por camadas de atenção e representações posicionais relativas dos *tokens*. Os autores treinaram o *NeuralCodeSum* em um conjunto de dados contendo 69 mil métodos em Java, utilizando outras 8 mil observações para validação e teste do modelo. Comparado com outras abordagens, os autores reportaram resultados de estado da arte.

Conduziremos com a implementação, treino e avaliação destes dois últimos modelos abordados, apresentados em [5] e [9]. Ambos possuem uma arquitetura similar baseada em *Transformers*, e reportam resultados de estado da arte em sumarização de código-fonte.

III. Metodologia Experimental

A. Conjuntos de Dados

Comparado aos modelos de linguagem da mesma época, o volume de dados utilizado para treinar o *Code-NN* é bastante pequeno.

```
// Source-Code
public void remove Swipe Listener(Swipe Listener
    listener)
  if (m Listeners == null) {
   return;
  }
  m Listeners.remove(listener);
}

// True Description
Removes a listener from the set of listeners

// Generated Description
Removes a listener from the list of listeners
```

```
// Source-Code
private static void use Missile(Player player) {
  Stackable Item projec. tiles Item = null;
  if (player.get Range Weapon() != null) {
    projec tiles Item = player.get Am munition();
  }
  if (projec tiles Item == null) {
    projec tiles Item = player.get Missile If Not
        Holding Other Weapon();
  }
  if (projec tiles Item!= null) {
    projec tiles Item.remove One();}
}

// True Description
Remove an used up missile from an attacking player

// Generated Description
Use this method to use a weapon
```

Exemplo 2. CodeBERT: Exemplos de sumarização

Deste então esforços têm sido feitos para formar conjuntos maiores de dados, contendo métodos e suas descrições.

Em 2018, Iyer lançou o *CONCODE* [2] [10], um conjunto de 100 mil pares de observações em Java e suas descrições. No mesmo ano, foi disponibilizado o *DeepCom* [3] [11] com 588 mil pares também em Java. Em 2019, surge o *FunCom* [4] [12] com 2,1 milhões de entradas para código-fonte em Java e descrição. Um ano depois, o conjunto de dados *CodeSearchNet* [5] [8] foi publicado, com um número total de 2,3 milhões de métodos documentados de código-fonte em seis linguagens de programação diferentes. Apesar de ser um conjunto amplo de dados na totalidade, este fica limitado quando olhamos individualmente para cada linguagem: 700 mil em PHP, 500 mil em Java, 500 mil em Python, e o resto dividido entre Go, Javascript e Ruby.

1) CoDesc: O *CoDesc* [6] [13] foi criado a partir da combinação dos conjuntos de dados descritos anteriormente — *CONCODE*, *DeepCom*, *FunCom* e *CodeSearchNet*. Para as observações do *CodeSearchNet*, métodos em linguagem Python foram traduzidos automaticamente para Java.

Em cima dos dados brutos foi realizado um trabalho de limpeza, detecção e remoção de ruídos. Esse tratamento é aplicado tanto para os códigos-fonte quanto para as descrições.

a) Limpeza de Dados e Tratamento de Ruídos: Para o tratamento das descrições dos métodos em linguagem natural, foram removidos símbolos, caracteres especiais, códigos HTML e XML presentes nos textos. Já dos códigos-fonte foram retirados comentários e caracteres não ASCII. É removida só a parte do texto que contém ruídos, sem descartar toda a entrada, para reduzir a perda de dados durante o pré-processamento. Foram também removidos os código-fonte com menos de três *tokens*, e todos os caracteres das descrições foram transformados em caracteres minúsculos.

Ao final do tratamento, o *CoDesc* acabou composto de 4,2 milhões de códigos-fonte e suas descrições. Este conjunto final de dados foi então dividido em três partes — treino, validação e teste.

Na Tabela I podemos ver o tamanho dos conjuntos de dados antes e depois da aplicação do tratamento descrito acima, e o particionamento dos dados de treino, validação e teste.

B. Implementação dos Modelos

Utilizamos os dados do *CoDesc* disponíveis para a tarefa de sumarização [7] para treinar a rede *CodeBERT* descrita em [5] para esta tarefa [8] e também duas implementações feitas em [9] disponíveis em repositório no *GitHub* [9]: a rede *NeuralCodeSum* implementada no artigo e uma rede RNN com camadas de atenção usada como base comparativa para os outros modelos.

[2]https://github.com/sriniiyer/concode
[3]https://github.com/xing-hu/DeepCom
[4]http://leclair.tech/data/funcom/
[5]https://github.com/github/CodeSearchNet
[6]https://github.com/code-desc/CoDesc
[7]https://github.com/code-desc/CoDesc/tree/master/CodeSummarization
[8]https://github.com/guoday/CodeBERT/tree/master/CodeBERT/code2nl
[9]https://github.com/wasiahmad/NeuralCodeSum

Tabela I
NÚMERO DE EXEMPLOS DO *CoDesc*

Fonte	Dados Brutos	Dados Limpos
DeepCom	588.108	424.028
FunCom	2.149.121	2.130.247
CodeSearchNet	998.991	924.201
CoDesc	5.920.530	4.211.516
treino	—	3.369.218
validação	—	421.149
teste	—	421.149

Fonte: Hassan, 2021 [13]

```
// Source-Code
public void add Restricted Domain(String domain
    Name) {
  if (String Utils.is Empty(domain Name)) {
    return;
  }
  if (restricted Domains = = null) {
    restricted Domains = new Array List < > ();
  }
  restricted Domains.add(domain Name);
}

// True Description
Adds specified domain name to the list of
    restricted domains

// Generated Description
Add a restricted domain
```

```
// Source-Code
private static void use Missile(Player player) {
  Stackable Item projec tiles Item = null;
  if (player.get Range Weapon() != null) {
    projec tiles Item = player.get Am munition();
  }
  if (projec tiles Item == null) {
    projec tiles Item = player.get Missile If Not
        Holding Other Weapon();
  }
  if (projec tiles Item!= null) {
    projec tiles Item.remove One();
  }
}

// True Description
Remove an used up missile from an attacking player

// Generated Description
Sets the projec munition tiles
```

Exemplo 3. NeuralCodeSum: Exemplos de sumarização

Para rodar esses modelos utilizamos o ambiente em nuvem da *Azure Machine Learning* com uma máquina Standard NC6, com seis núcleos de processamento, 56GB de memória RAM, e uma placa de vídeo NVIDIA Tesla K80. Devido às limitações de poder computacional dessa configuração para realizar treino e refinamento de redes *Transformers*, tivemos que alterar os parâmetros das redes definidos nas implementações dos repositórios.

```
// Source-Code
private static boolean is Double Equal(double
    value, double value To Compare) {
  return (Math.abs(value - value To Compare) < NUM);
}

// True Description
Checks if is double values are equal

// Generated Description
Checks if two double values are equal
```

```
// Source-Code
private static void use Missile(Player player) {
  Stackable Item projec tiles Item = null;
  if (player.get Range Weapon() != null) {
    projec tiles Item = player.get Am munition();
  }
  if (projec tiles Item == null) {
    projectiles Item = player.get Missile If Not
        Holding Other Weapon();
  }
  if (projec tiles Item!= null) {
    projec tiles Item.remove One();
  }
}

// True Description
Remove an used up missile from an attacking player

// Generated Description
Use this api to use ns simple item
```

Exemplo 4. RNN: Exemplos de sumarização

Todas as abordagens foram treinadas com um conjunto de dados reduzidos de 500 mil exemplos, sendo utilizados outros 8.714 exemplos para validação, e outro conjunto de 8.714 exemplos para teste. Todos os modelos foram treinados por 10 épocas, e tiveram o parâmetro *batch_size* reduzido para 16, tendo em vista a limitação de memória do nosso ambiente.

A Tabela II mostra os valores dos parâmetros que foram ajustados em relação às implementações descritas nos repositórios.

Tabela II
MODIFICAÇÕES NOS PARÂMETROS DOS MODELOS

Parâmetro	CodeBERT	NeuralCodeSum	RNN
data_workers	—	2	2
batch_size	16	16	16
epochs	10	10	10

C. Medida de Avaliação

Para avaliar o desempenho das abordagens adotadas, utilizamos a métrica BLEU-4. Essa métrica mede a qualidade de uma tradução de texto entre duas linguagens.

Para o nosso problema, cada descrição de sumariação recebe um BLEU score entre 0 e 1 indicando o quão similar a descrição gerada é em relação à descrição verdadeira.

O número 4 em BLEU-4 indica que a métrica computa os scores de unigramas, bigramas até 4-gramas. Esses scores são então ponderados para todo o corpus de treino para se chegar em um desempenho geral do modelo. O cálculo do BLEU pode ser visto na equação abaixo:

$$\text{BLEU-N} = \min\left(1, \exp\left(1 - \frac{r}{c}\right)\right) \prod_{n=1}^{N} p_n^{w_n} \qquad (1)$$

onde c é o número total de unigramas na descrição candidata, r é o número de unigramas candidatos encontrados na descrição de referência, p_n é a precisão para cada n-grama e finalmente w_n é um peso entre 0 e 1 atribuído a p_n, onde $\sum_{n=1}^{N} w_n = 1$.

IV. RESULTADOS

As três abordagens foram treinadas no conjunto de treino, e a métrica BLUE-4 foi computada nos dados de validação após cada época de treino. Após o fim do treino e validação, foi utilizado o conjunto de dados de teste para computar o BLUE-4 de cada abordagem. Na Tabela III abaixo apresentamos os resultados da nossa métrica obtidos nessas etapas.

Tabela III
RESULTADOS AVALIADOS EM
CONJUNTO DE VALIDAÇÃO E TESTE

Dataset	CodeBERT	NeuralCodeSum	RNN
Validação	15,38	14,50	12,87
Teste	15,29	14,62	13,04

Dentre os três modelos avaliados, o *CodeBERT* obteve o melhor resultado, com um BLUE score de 15,29 — diferença de quase um ponto na métrica em relação à segunda abordagem, e mais de dois pontos em relação à nossa base comparativa. A proximidade entre os valores de validação e teste também afasta a hipótese de sobreajuste dos modelos.

V. CONCLUSÕES

Interessante notar que as abordagens que fazem uso de *Transformers* tem um desempenho visivelmente superior à abordagem RNN. Este resultado vai de encontro com a literatura recente da área.

Também cabe salientar que as diferenças observadas nos resultados podem conter viés advindo dos comprometimentos feitos por limitação da nossa capacidade computacional. Os resultados deste trabalho não esgotam a discussão sobre o estado da arte na tarefa de sumarização de código-fonte, conquanto trazem luz à discussão sobre conjuntos de dados de referência e a comparabilidade entre diferentes abordagens.

A. Próximos Passos

Apontamos aqui alguns caminhos como próximos passos do presente trabalho. O primeiro deles seria implementar modelos que utilizem estruturas de dados complexas como as árvores sintéticas abstratas (AST), e fazer estudos de ablação em abordagens que combinem representações vetoriais e árvores sintáticas abstratas.

O tópico das redes neurais por grafos também é um caminho ser explorado com profundidade. Analisamos aqui um trabalho

de LeClair [1] que utilizou redes neurais convolucionais por grafos (*ConvGNN*), mas outras arquiteturas de rede por grafos poderiam ser testadas.

A abordagem do *CodeBERT* referida neste trabalho possui uma implementação utilizando grafos, chamada *GraphCodeBERT*. Parece ser um caminho interessante aliar o poder das redes *Transformers* com estruturas de dados capazes de capturar relações complexas como os grafos.

BIBLIOGRAFIA

[1] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *2020 IEEE/ACM International Conference on Program Comprehension*, Oct. 2020.

[2] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "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)*, (Berlin, Germany), pp. 2073–2083, Association for Computational Linguistics, Aug. 2016.

[3] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019.

[4] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin, "Graph2seq: Graph to sequence learning with attention-based neural networks," *arXiv preprint arXiv:1804.00823*, 2018.

[5] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *CoRR*, vol. abs/2002.08155, 2020.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[7] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.

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

[9] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

[10] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," *CoRR*, vol. abs/1808.09588, 2018.

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

[12] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," *Proceedings of the 2019 Conference of the North*, 2019.

[13] M. Hasan, T. Muttaqueen, A. A. Ishtiaq, K. S. Mehrab, M. M. A. Haque, T. Hasan, W. Ahmad, A. Iqbal, and R. Shahriyar, "CoDesc: A large code–description parallel dataset," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, (Online), pp. 210–218, Association for Computational Linguistics, Aug. 2021.

[14] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *CoRR*, vol. abs/1707.02275, 2017.

[15] J. Moore, B. Gelman, and D. Slater, "A convolutional neural network for language-agnostic source code summarization," *CoRR*, vol. abs/1904.00805, 2019.