OTIMIZAÇÃO DE MODELOS LLM PARA AUXILIAR NA REVISÃO DE CÓDIGO EM CLOJURE.

Danielly Rayanne Macedo Lima[1], João Arthur Brunet Monteiro [2]

RESUMO

A revisão de código é uma parte essencial do ciclo de desenvolvimento de *software*, pois garante qualidade e minimiza o surgimento de *bugs.* Contudo, ainda é realizada majoritariamente de forma manual por outros desenvolvedores. Embora muito relevante, ela ainda é considerada demorada, custosa e suscetível a erros. O surgimento de Grandes Modelos de Linguagem possibilitou a automação de várias tarefas, incluindo a assistência nessa atividade. Assim, o propósito deste trabalho é explorar a utilização desses modelos no processo de revisão de código, utilizando a linguagem de programação Clojure, que, apesar de ser relativamente emergente, tem ganhado notoriedade no mercado. O modelo escolhido para a condução desta pesquisa foi o *Mistral-7B-Instruct-v0.2,* combinado com a técnica de aprimoramento *Retrieval Augmented Generation (RAG),* que em português é traduzida para Geração Aumentada por Recuperação. Os dados foram obtidos de projetos *open source* disponíveis na plataforma do *GitHub,* e os resultados de ambos os modelos foram avaliados por meio de similaridade de cosseno. Por fim, os resultados mostraram que o modelo consegue revisar código de forma comparável a revisores humanos, e que a técnica de aprimoramento consegue fornecer revisões mais específicas quando comparado com o modelo sem técnica de aprimoramento.

Palavras-chave: Grandes Modelos de Linguagem, Geração Aumentada via Recuperação, Processamento de Linguagem Natural.

[1]Aluno de Ciência da Computação, Departamento de <Nome do Departamento>, UFCG, Campina Grande, PB, e-mail: danielly.rayanne.macedo.lima@ccc.ufcg.edu.br
[2]<Titulação>, <Função>, <Departamento>, UFCG, Campina Grande, PB, e-mail: joao.arthur@computacao.ufcg.edu.br

USING OPTIMIZATION TECHNIQUES TO ENHANCE CODE REVIEWS WITH LARGE LANGUAGE MODELS IN CLOJURE

ABSTRACT

Code review is an essential part of the software development cycle as it ensures quality and minimizes the emergence of bugs. However, it is still predominantly performed manually by other developers. Despite its relevance, the process is considered time-consuming, costly, and a failure-susceptible task. The advent of Large Language Models has enabled the automation of various tasks, including assistance in this activity. Thus, the purpose of this work is to explore the use of these models in the code review process, using Clojure programming language, which, although relatively new, has gained prominence in the market. The chosen model for conducting this research was Mistral-7B-Instruct-v0.2, combined with the Retrieval Augmented Generation (RAG) optimization technique. Furthermore, the data was obtained from open source projects available on the GitHub platform, and the results of both models were evaluated using cosine similarity. The results showed that the model can review code comparably to human reviewers, but the RAG optimization technique did not manage to improve the reviews in this case.

Keywords: Large Language Models, Retrieval Augmented Generation, Natural Language Processing.

1. INTRODUÇÃO

Grandes Modelos de Linguagem (referenciados comumente como LLM, acrônimo em inglês para *Large Language Models*) são modelos que foram desenvolvidos com o propósito de executar tarefas diversas em linguagem natural. Após a publicação do artigo *"Attention is All You Need",* em 2017, os modelos LLM tiveram uma evolução significativa, tornando-os cada vez mais importantes. Assim, com o contínuo esforço e investimento de pesquisadores e empresas, a escala e o alcance desses modelos nos últimos anos têm estendido nosso entendimento sobre o que pode ser alcançado [1].

Esses modelos LLM utilizam arquiteturas de aprendizado profundo, como a arquitetura de Transformer introduzida no artigo previamente mencionado. Essas estruturas possibilitam o treinamento de uma rede neural a partir de uma fonte extensa de dados. Como resultado, inúmeros são os benefícios de se utilizar um modelo LLM, dado a sua flexibilidade a diversos contextos e aplicações. Além disso, os modelos apresentam um bom desempenho em atividades relacionadas a tarefas de Processamento de Linguagem Natural (PLN), sendo tipicamente usados para atividades que envolvam geração de texto, perguntas e respostas, tradução automática, análise de sentimentos, entre outras. Dentro da lista de modelos notáveis e amplamente utilizados atualmente estão o LLaMa, desenvolvido pela Meta, e o renomado GPT, desenvolvido pela OpenAI.

Adicionalmente, observa-se um crescente interesse na exploração dos modelos em atividades relacionadas a Engenharia de Software, dentre elas classificação de severidade de bugs, sumarização de bug reports, geração automática de testes e revisão de código. Nesse contexto, o processo de revisão de código, inicialmente discutido por Michael E. Fagan [5], aprimora a produtividade e a qualidade de um projeto, por meio de operações formais bem definidas durante várias fases do desenvolvimento a fim de mitigar possíveis erros. A revisão de código é uma parte essencial do ciclo de vida do desenvolvimento de software, pois tem como objetivo garantir a qualidade do código e minimizar o aparecimento de bugs [2]. Em particular, há avanços significativos no uso de LLM no contexto de revisão de código [2, 3, 4].

No sentido de explorar a revisão de código com os modelos, no trabalho de Li *et al. (2022)* foi desenvolvido o *CodeReviewer*, um modelo pré-treinado capaz de realizar tarefas de estimação de alterações de código, geração de comentários e refinamento de código. O modelo foi treinado utilizando as nove linguagens de programação mais populares do ano, e os resultados obtidos mostraram-se relevantes. Além disso, nos trabalhos de Tufano et. al [3] e Li et. al [4]também foi explorada a construção de modelos LLM no contexto de revisão de código e, dessa vez, ambos focando na linguagem de programação Java.

Como podemos observar, há diversos estudos que focam em linguagens de programação populares e amplamente utilizadas em pesquisas e no mercado. No entanto, a exploração de modelos para a linguagem Clojure ainda é limitada. A linguagem de programação Clojure é um dialeto LISP que vem ganhando cada vez mais espaço na comunidade de desenvolvedores por inúmeros motivos, como sua simplicidade e legibilidade de código, suporte a projetos de alta concorrência, entre outras características. Recentemente, vem recebendo notoriedade e suporte de empresas como o Nubank, que mantém atualmente o suporte da linguagem. Além disso, gigantes como Amazon, Globo e Walmart também a utilizam em suas soluções.

Esse déficit nos estudos de LLMs nesta linguagem se deve, em parte, ao seu status emergente e à falta de uma vasta fonte de dados para um bom treinamento do modelo. Sendo assim, esta pesquisa tem o propósito de explorar e avaliar o uso de LLM na revisão de código Clojure. Utilizando técnicas de aprimoramento, busca-se analisar a possibilidade de obter resultados melhores, uma vez que os modelos são generalistas e, quando otimizados, podem gerar resultados mais satisfatórios.

O treinamento e o ajuste fino de modelos são atividades custosas e desafiadoras. Diante desse fato, surgiram outras abordagens e técnicas que visam aprimorar as respostas de modelos, como o *Retrieval Augmented Generation* (RAG)[7], que vem sendo bastante explorado. Proposta por pesquisadores da *Meta,* essa abordagem consiste na adição de uma fonte externa e específica de conhecimento ao já presente nos modelos, permitindo que eles gerem respostas mais precisas. O RAG é uma alternativa considerada simples, eficaz e bem menos custosa que o ajuste fino tradicional. Em termos gerais, ele recupera informações relevantes ao *prompt* inicial as injeta em um novo *prompt* reformulado, orientando o processo de geração do modelo dentro do contexto específico da aplicação.

Portanto, as principais contribuições deste trabalho são:

- Experimentação do uso da técnica de aprimoramento *Retrieval Augmented Generation (RAG)* para melhorar a qualidade das respostas geradas pelo modelo;
- Análise manual das respostas do modelo para fornecer uma avaliação mais detalhada dos resultados.

2. MATERIAIS E MÉTODOS (OU METODOLOGIA)

Nesta seção, é descrita detalhadamente a condução desta pesquisa, desde a construção da base de dados até o *design* do plano de experimento e a avaliação da técnica de otimização de modelos em relação ao modelo não otimizado. Na imagem a seguir, temos uma visão geral e norteadora dessas etapas, que detalharemos ao longo da seção:

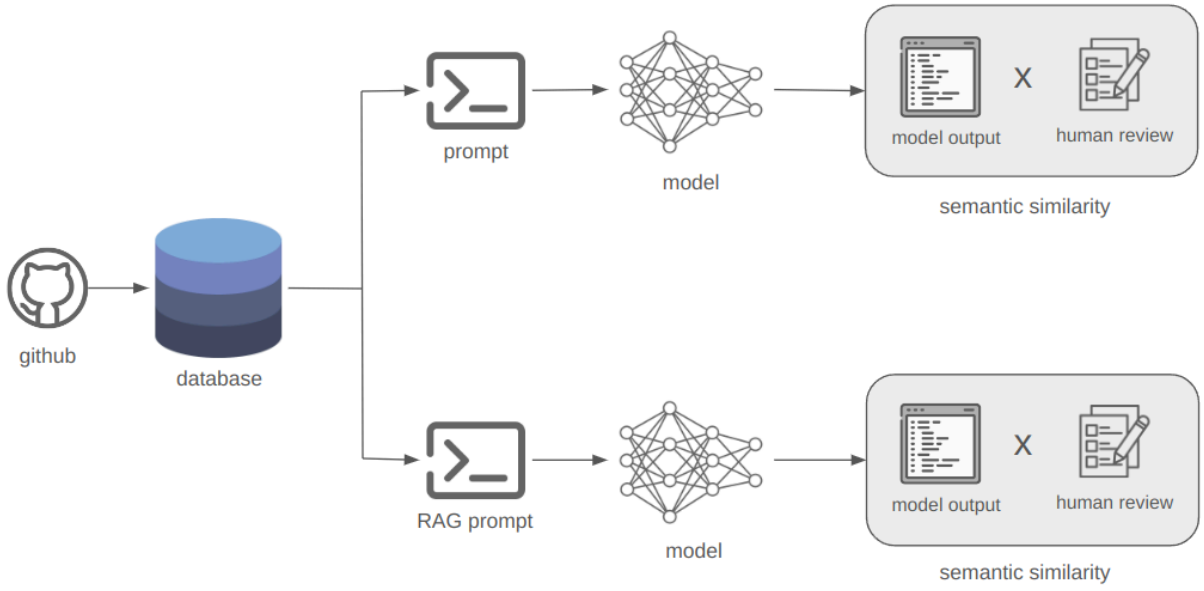

Imagem 1. Norteador da Pesquisa.

2.1. QUESTÕES DE PESQUISA

Para alcançar os objetivos propostos, é necessário delimitar as questões que guiam esta pesquisa. Assim, temos a finalidade de esclarecer duas perguntas essenciais:

- Qual a natureza das revisões de código efetuadas pelos modelos LLM?
- Qual a diferença entre as revisões automáticas das revisões geradas por seres humanos?
- Qual o desempenho do uso do RAG no contexto de revisões de código?

2.2. COLETA E TRATAMENTO DE DADOS

Nessa primeira etapa, foram selecionados *pull requests* de projetos *open-source* em Clojure, disponíveis no *Github*, seguindo a lógica de extrair esses dados de projetos do Top 100 Clojure. Esse ranking sumariza os repositórios mais populares da linguagem que estão hospedados na plataforma, levando em consideração a quantidade de estrelas que cada um possui. Tal escolha se deu pelo fato de que essa amostra de códigos é mais suscetível a possuir alguma discussão entre desenvolvedores por meio de comentários nos *pull requests.* Assim, teremos revisão e avaliação de código feita por seres humanos para servir de parâmetro no momento de análise dos resultados obtidos nos experimentos, além do trecho de código modificado.

Como feito no estudo "*Automating Code Review Activities by Large-Scale Pre-Training*" [2], escolhemos coletar e levar em consideração apenas o *diff hunk* (trecho de código modificado). O *dataset* então possui, para cada trecho de código, um comentário atrelado referente a revisão humana. Assim, para extração desses dados foi desenvolvido um script na linguagem de programação *Go* que, juntamente com a *API* da plataforma, foram responsáveis pelos mais de 12 mil *pull requests* retirados ao longo dos 100 repositórios catalogados previamente.

A etapa subsequente, de pré-processamento e limpeza dos dados, se deu com o intuito de mitigar possíveis ruídos nos dados que pudessem atrapalhar o desempenho das revisões do modelo. Sendo assim, foram removidos trechos de código em outras linguagens de programação, todas as tags referentes a diff e possíveis linhas que não continham código, como documentação da linguagem, importações de bibliotecas, comentários, entre outros. Além disso, foram selecionados apenas os trechos que tivessem 5 ou mais linhas de código, com o intuito de tentar evitar que trechos que tivessem apenas importações ou documentações fossem considerados nas revisões. Ao fim dessa etapa, o *dataset* foi reduzido para um pouco mais de 5.000 linhas.

2.3. FERRAMENTAS

O modelo escolhido foi o Mistral-7B, desenvolvido pela Mistral AI e *open-source,* promete resultados semelhantes ou superiores ao LLaMA 2 13B. Ainda mais, vem ganhando notoriedade por utilizar as escolhas arquiteturais de *Sliding Window Attention (SWA)* e *Grouped Query Attention (GQA),* que permitem eficiência e escalabilidade para longos textos e a inferência mais rápida utilizando menos memória. Além disso, o ambiente de execução escolhido para a condução dos experimentos foi a plataforma do *Google Colab*, utilizando a assinatura do *Colab Pro+,* que nos permite acesso a uma NVIDIA A100 40 GB PCIe GPU Accelerator, com 40 GB de RAM e 200 GB de disco.

2.4. PROMPT

O prompt utilizado no experimento foi definido por meio algumas estratégias descritas na documentação, com a utilização dos *tokens* especiais de delimitação: *Beginning of string (BOS)* <s> e *End of string (EOS)* </s> e das tags de marcação de instruções: [INST] e [/INST]. Assim, o prompt segue a seguinte formatação:

```
<s>[INST] Instruction [/INST] Model answer</s>[INST] Follow-up instruction [/INST]
```

Imagem 2. Formatação do prompt.

Os *tokens* de delimitação são responsáveis por ajudar o modelo a separar a instrução em si de qualquer outro texto ao redor, o que permite que ele concentre melhor seu poder de processamento na tarefa específica fornecida e, sem ele, o modelo pode precisar de etapas extras para determinar em qual lugar a instrução começa. Já em relação às tags de instrução, essas são responsáveis por informar ao modelo qual tipo de tarefa deve ser executada.

```
<s>[INST] You are an experienced programmer in Clojure. Review the following code snippet and provide feedback
on its readability, efficiency, and potencial bugs.

** Code Snippet: **
{diff_hunk}

</s> [INST] Format your review as text with itemized concrete instructions to the author of the code and do not add
this prompt to the answer. [/INST]
```

Imagem 3. Prompt sem RAG.

<s>[ISNT] You are an experienced programmer in **Clojure**. Review the following code snippet and provide feedback on: readability, efficiency, maintainability, and potencial bugs.

{code_snippet}

Conduct your review based on the following review examples:

{context}

</s> [INST] Format your review as text with itemized concrete instructions to the author of the code. Do not add this prompt to the output. [/ISNT]

Imagem 4. Prompt com RAG.

2.5. EXPERIMENTO

O experimento foi dividido em duas partes. Na primeira parte, o modelo foi testado sem considerar nenhuma técnica de aprimoramento, enquanto na segunda parte foi utilizada a técnica de aprimoramento RAG. Além disso, em ambas as partes do experimento foi utilizado a versão do Mistral-7B-Instruct-v0.2, que é um ajuste fino do Mistral-7B com a capacidade de dar respostas diretas em forma instruções.

2.6. SUMARIZAÇÃO E ANÁLISE DOS RESULTADOS

Após a realização dos experimentos em ambos os modelos (com e sem otimização) vem a última etapa, a de comparação dos resultados obtidos a fim de analisarmos o desempenho de ambos os modelos. Em avaliações feitas na área de *Machine Learning* tradicional é realizada a análise de performance com métricas como *F1-Score, precision* e *recall.* No entanto, no contexto de avaliação da performance de um modelo LLM, essas métricas não são as mais adequadas, necessitando de uma outra forma de análise.

Existem algumas abordagens que estão sendo utilizadas para fazer a análise de respostas de modelos LLM, como *BLEU Score,* similaridade de cosseno, entre outras. Neste trabalho foi utilizado a similaridade de cosseno para avaliar as respostas do modelo. Sendo assim, a análise foi dividida em 3 etapas (Saída do modelo sem aplicação de nenhuma técnica de otimização em relação aos comentários extraídos dos *Pull Requests*; Saída do modelo com aplicação da técnica de otimização RAG em relação aos comentários; Comparativo entre os resultados das análises anteriores).

3. DESENVOLVIMENTO

O desenvolvimento desta pesquisa se deu em quatro grandes etapas, seguindo a lógica apresentada anteriormente na figura norteadora da pesquisa e detalhada nas próximas seções.

3.1. ETAPA 1: COLETA E LIMPEZA DOS DADOS

A primeira etapa foi dedicada à coleta e limpeza de todos os dados que serviram de base ao longo desta pesquisa. Utilizando um script de coleta que acessou a API do GitHub, foram extraídos um pouco mais que 12 mil *pull requests*. Esses dados foram a priori filtrados para incluir apenas aqueles que possuíssem comentários, uma vez que o foco principal do estudo é o da interação entre desenvolvedores por meio de revisões de código. A existência desses comentários foi fundamental ao desenvolvimento da pesquisa, pois eles servem como referência para a comparação das revisões feitas pelo modelo.

Para a escolha de quais projetos *open-source* seriam selecionados para extração dos dados, foi utilizado o Top 100 Clojure do GitHub, como descrito na seção de materiais e métodos, que facilita achar a interação entre desenvolvedores por meio de comentários. Além disso, consideramos apenas os *diff hunks* de cada *pull request* extraído, visto que é uma abordagem utilizada em outros estudos da área e que não compromete os resultados.

O *diff hunk* representa exclusivamente a parte modificada e enviada para o *pull request*, desconsiderando o restante do arquivo, que pode ser muito extenso e, no contexto da pesquisa, inviável de ser utilizado. Essa estratégia ajudou a eliminar possíveis ruídos nos dados com partes irrelevantes do código, contribuindo para uma análise mais precisa do modelo. A imagem 5 mostra uma visão geral desse processo.

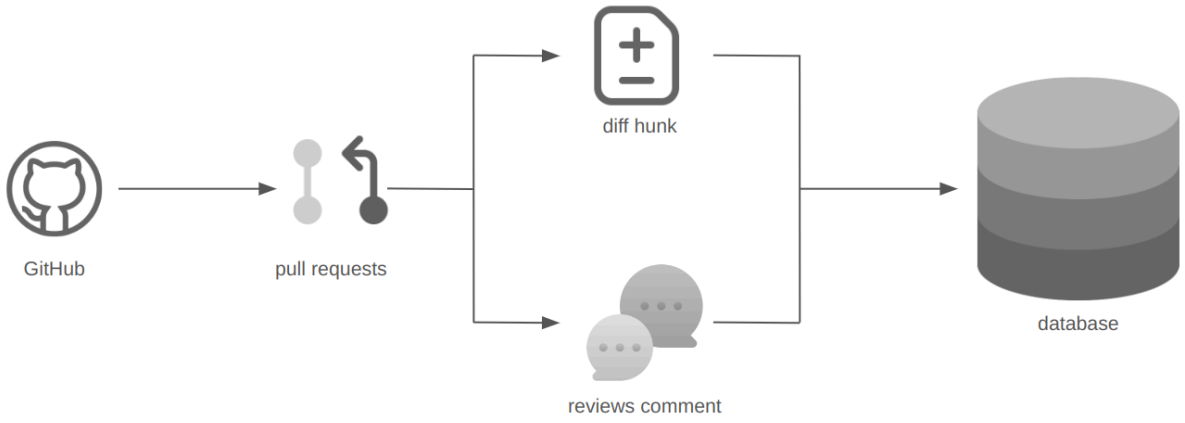

Imagem 5. Processo de extração de dados.

Posteriormente a coleta de dados, foi feita uma limpeza para tentar mitigar ainda mais possíveis ruídos. Nessa fase, a abordagem utilizada removeu trechos de código menores que 5 linhas, dado ao fato de que geralmente esses trechos eram apenas de *pull requests* que continham elementos como importações, documentação, comentários, entre outros. Também foram removidas todas as tags referentes a adições (“+”) e deleções (“-”) associadas aos *diff hunks*. Ao final desse processo, a base de dados foi reduzida a 5105 *pull requests.*

3.2. ETAPA 2: EXPERIMENTO NO MODELO SEM OTIMIZAÇÃO

Na segunda etapa, foi realizado o experimento com o modelo para analisar o seu comportamento quando nenhuma técnica de aprimoramento dos seus resultados havia sido aplicada. Como descrito anteriormente na seção da metodologia, o modelo escolhido para a condução dos experimentos foi o Mistral-7B,

mais especificamente o Mistral-Instruct-7B-v0.2, que é uma versão do modelo que traz suas respostas no formato de instruções.

A escolha por esse modelo se deu pelo fato dele ser de código aberto e estar sendo amplamente utilizado por acadêmicos e empresas, sendo conhecido pela sua alta capacidade de geração de textos (comparáveis ou superiores ao modelo *Llama2* de 13 bilhões de parâmetros, desenvolvido pela *Meta*).

Uma das principais limitações para a utilização de grandes modelos de linguagem ainda é a demanda significativa por recursos de hardware, mais especificamente pelas GPUs. Com isso, foi necessário recorrer a uma solução de ambiente de desenvolvimento baseado na nuvem que nos fornecesse os recursos necessários para conseguir conduzir os experimentos com o modelo.

3.3. ETAPA 3: EXPERIMENTO NO MODELO COM APRIMORAMENTO

A terceira etapa do estudo foi realizada com o experimento do modelo com a técnica de aprimoramento RAG. Dessa forma, foram utilizados os próprios *pull requests* coletados na primeira etapa para servir de conhecimento externo e específico que essa abordagem requer. Para a implementação desse experimento, foi utilizado também o *Chroma DB,* um banco de dados *open-source* que serve para armazenar e gerenciar vetores para serem utilizados pelos modelos. No momento da criação do prompt para ser enviado ao modelo são recuperados dados importantes e relevantes ao contexto, para dar mais contexto ao modelo e auxiliá-lo na geração de uma resposta mais específica. O experimento anterior foi adaptado para que ele utilizasse RAG, o processo como um todo é ilustrado na imagem abaixo.

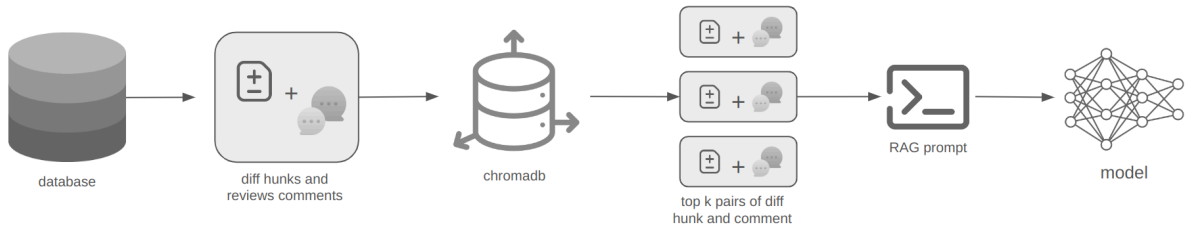

Imagem 6. Experimento com RAG.

3.4. ETAPA 4: SUMARIZAÇÃO E ANÁLISE DOS RESULTADOS

A etapa final desta pesquisa se deu pela a análise quantitativa e qualitativa dos resultados obtidos da realização dos dois experimentos. A análise quantitativa foi conduzida com a métrica da similaridade de cosseno, que é uma medida de similaridade entre dois vetores num espaço vetorial e que avalia o valor do cosseno do ângulo compreendido entre eles. Essa métrica possui valores entre o intervalo de -1 a 1, sendo -1 o resultado obtido para vetores totalmente opostos (ou seja, não similares) e 1 para vetores idênticos. Para transformar as saídas dos modelos e os comentários dos *pull requests* coletados em vetores, foi necessário utilizar o modelo de *embeddings* '*bert-base-uncased'* e para fazer o cálculo da similaridade foi utilizada a função *cosine_similarity* da biblioteca *scikit-learn.* A imagem 6 a seguir exemplifica o processo de análise dos resultados.

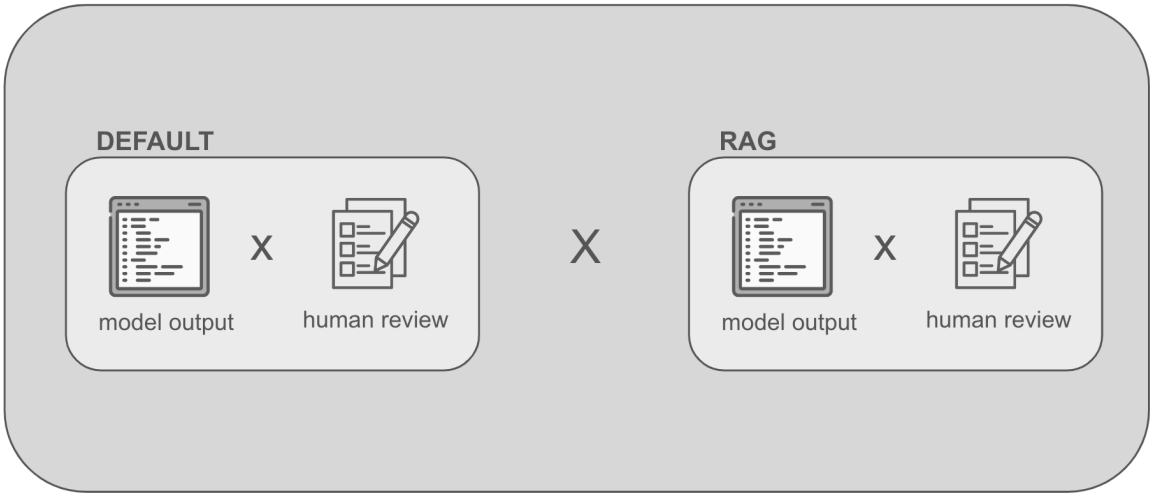

Imagem 7. Processo de análise quantitativa dos resultados.

Para a condução dessa pesquisa foram gastos aproximadamente 1.000 créditos do *Google Colab,* em cerca de 96 horas totais, para realizar ambos os experimentos (com e sem a técnica de aprimoramento), totalizando um valor de R$516,00 reais gastos para os mais de 5 mil revisões de cada parte do experimento. Ainda mais, a relação com todos os artefatos necessários à execução desta pesquisa, incluindo a relação dos projetos que estavam no ranking no momento da coleta dos dados, os scripts de coleta e limpeza, os experimentos e seus resultados, podem ser encontrados no repositório destinado à pesquisa [6].

4. RESULTADOS E DISCUSSÕES

Ao longo deste estudo, foi analisada a utilização da técnica de aprimoramento RAG como uma abordagem alternativa ao ajuste fino de modelos que, como visto nas seções acima, é uma atividade custosa e muitas vezes inviável para ser realizada. Nesta seção, são apresentados os resultados obtidos dos experimentos realizados que tiveram como foco revisões de código em Clojure.

4.1. MODELO SEM NENHUMA TÉCNICA DE APRIMORAMENTO

O objetivo deste experimento foi estabelecer uma base para a comparação e avaliação do desempenho das revisões feitas pelo modelo com a aplicação da técnica de aprimoramento. Neste caso, o modelo somente utiliza para fazer a inferência os dados que foram disponibilizados durante o treinamento, ou seja, dados gerais sem nenhuma especificidade do contexto de revisões de código em Clojure.

Após a execução do experimento, todas as respostas geradas e todos os comentários das revisões humanas presentes na base de dados foram transformados em *embeddings*. Essa conversão possibilita uma comparação quantitativa direta entre os textos. Os resultados estão sumarizados e representados na imagem 8.

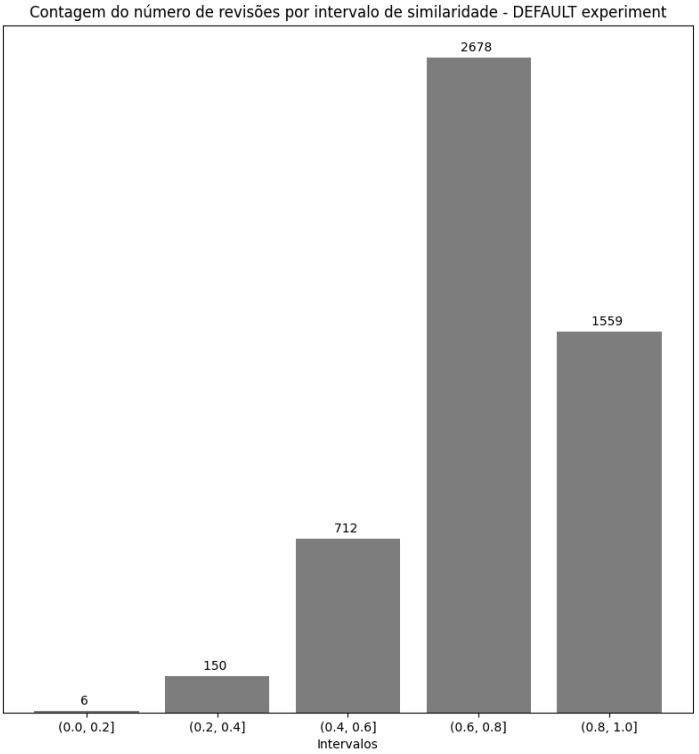

Imagem 8. Gráfico da similaridade semântica *DEFAULT*.

É possível notar, com auxílio do gráfico, que a maioria dos valores de similaridade de cosseno concentram-se no intervalo de 0,6 a 1,0. Isso indica que o modelo já consegue ser bastante próximo das revisões humanas em sua versão padrão.

4.2. MODELO COM A TÉCNICA DE APRIMORAMENTO RAG

Com a técnica de aprimoramento aplicada para a condução desta etapa, as respostas do modelo também foram transformadas em *embeddings* para serem comparadas juntamente com os *embedding*s das revisões humanas. O resultado dessa comparação está sumarizado na imagem abaixo.

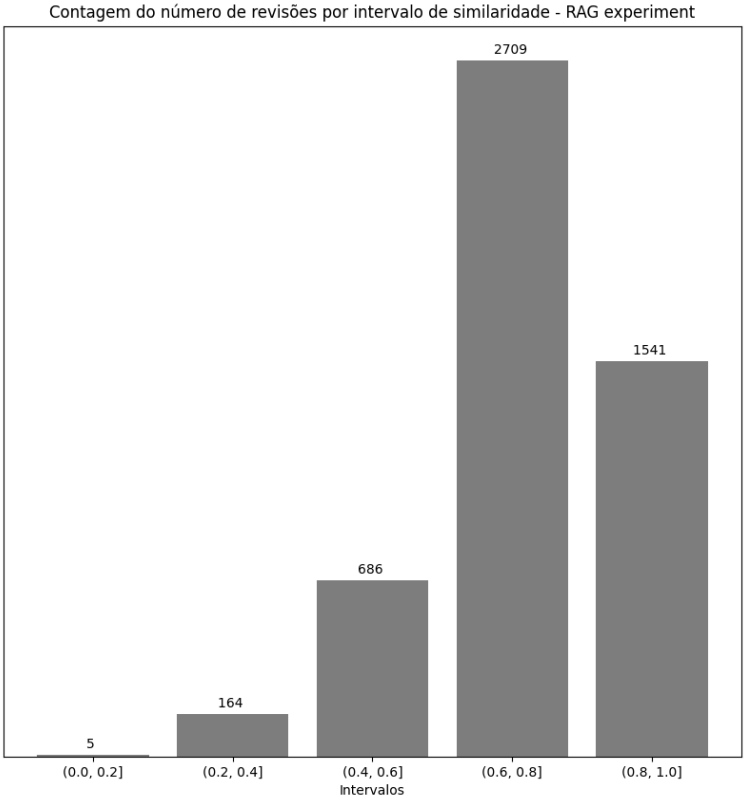

Imagem 9. Gráfico da similaridade semântica *RAG*.

No entanto, foi percebido que as similaridades de ambos os experimentos resultaram em valores muito parecidos. Em um primeiro momento, isso sugere que a técnica de aprimoramento não conseguiu melhorar significativamente as respostas, do ponto de vista quantitativo. Diante desse fato, tem-se necessária a análise qualitativa de uma amostra desses resultados para obter uma visão geral de como as revisões se comportam, como está descrita no próximo tópico.

4.3. COMPARAÇÃO ENTRE AS ABORDAGENS

Esta seção examina com mais detalhes e de forma qualitativa os dados obtidos dos dois experimentos deste trabalho. A análise qualitativa, diferentemente da quantitativa, consegue proporcionar uma visão melhor da qualidade dos resultados e pode revelar nuances importantes para a pesquisa. Sendo assim, dos mais de 5 mil trechos de código que foram revisados por ambas as abordagens, 5% deles (mais especificamente 262) foram analisados manualmente. A imagem 10 abaixo traz o comparativo entre ambas as similaridades de cosseno.

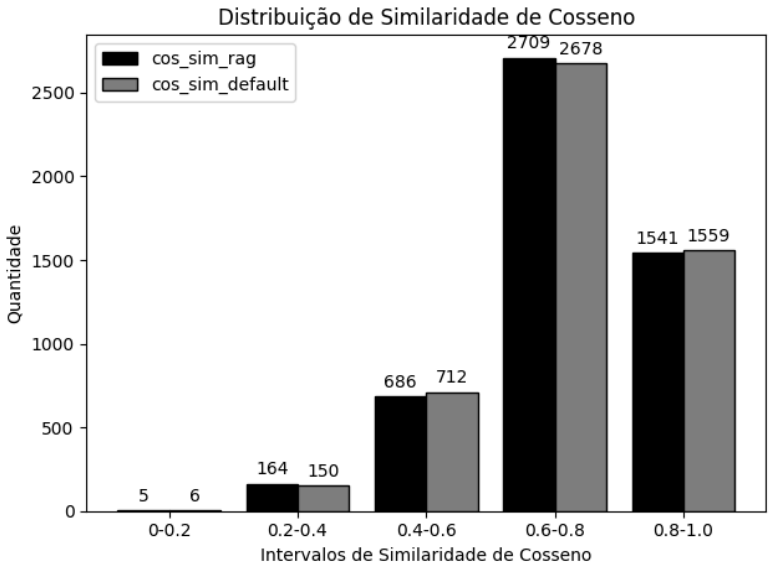

Imagem 10. Comparação entre as similaridades de ambos modelos.

Do total analisado, em 161 casos (ou 61,45%) foi considerado que o modelo com a aplicação da técnica de aprimoramento conseguiu ser mais específico. Essa especificidade se dá pelo fato de que o *RAG* cita trechos do código enviado ao longo da revisão muito mais que o *DEFAULT*, que trouxe revisões mais genéricas. Em 73 casos, ou 27,86% deles, o *DEFAULT* consegue ser mais específico que o *RAG*. Por fim, em apenas 10,68% ou 28 casos nenhum dos dois conseguiram trazer revisões mais específicas. Dessa forma, somente por meio dessa análise conseguiu-se ter uma resposta mais concreta do comportamento dessas revisões, mostrando que o *RAG* de fato conseguiu melhorá-las. A imagem 11 mostra uma nuvem de palavras criadas a partir das revisões dos modelos onde podemos ver as palavras com mais frequência nesses textos.

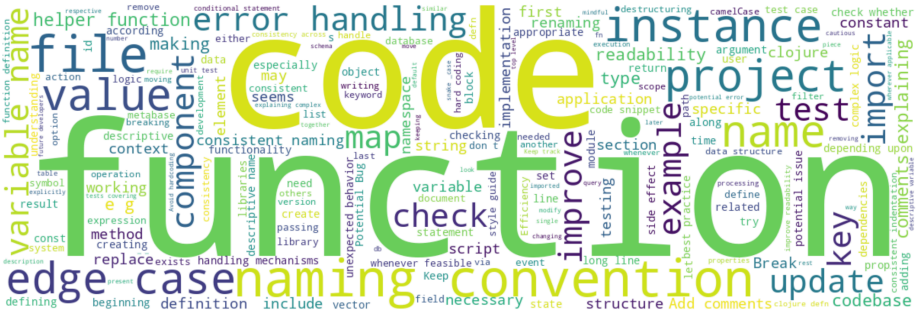

Imagem 11. Nuvem de palavras.

5. CONCLUSÃO

Diante dos resultados apresentados, podemos concluir a importância da análise qualitativa nas pesquisas que envolvem a utilização de métricas de similaridade textual, uma vez que essas métricas são limitadas no que diz a respeito ao entendimento da semântica dos textos. As similaridades de cosseno aplicadas a ambos os experimentos retornam valores semelhantes, pois ambos os modelos abordam os mesmos tópicos ao fazer uma revisão, como legibilidade, organização de código, tratamento de erros, criação de testes, eficiência do código, entre outros.

No entanto, foi possível observar, por meio de análise manual, que a aplicação da técnica de aprimoramento (RAG) consegue fornecer uma revisão mais específica, uma vez que cita trechos do código analisado para guiar a revisão, no lugar de apenas oferecer uma revisão geral sobre boas práticas. Portanto, conclui-se que o RAG aprimora os resultados, proporcionando respostas mais específicas e direcionadas, ao fornecer mais contexto ao modelo.

AGRADECIMENTOS

O presente trabalho foi realizado com apoio do CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico – Brasil, através do programa de Iniciação Científica PIBIC/CNPq-UFCG.

REFERÊNCIAS

[1] BOMMASANI, Rishi et al. **On the opportunities and risks of foundation models**. arXiv preprint arXiv:2108.07258, 2021.

[2] LI, Zhiyu et al. Automating code review activities by large-scale pre-training. In: **Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. 2022. p. 1035-1047.

[3] TUFANO, Rosalia et al. Using pre-trained models to boost code review automation. In: **Proceedings of the 44th international conference on software engineering**. 2022. p. 2291-2302.

[4] LI, Lingwei et al. Auger: Automatically generating review comments with pre-training models. In: **Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. 2022. p. 1009-1021.

[5] FAGAN, Michael E. Design and code inspections to reduce errors in program development. **IBM Systems Journal**, v. 38, n. 2.3, p. 258-287, 1999.

[6] LIMA, D.R.M**.** llm-code-review-clj. 2024.
Disponível em: <https://github.com/raiaiaia/llm-code-review-clj/>. Acesso em: 24 set. 2024.

[7] LEWIS, Patrick et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP tasks. **In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20)**. Curran Associates Inc., Red Hook, NY, USA, Article 793, 9459–9474.