LUCAS RAFAEL RODRIGUES PEREIRA

UMA ABORDAGEM DE DEEP-LEARNING PARA REALIZAR A PREDIÇÃO DE REFATORAÇÕES

LAVRAS – MG

2024

LUCAS RAFAEL RODRIGUES PEREIRA

UMA ABORDAGEM DE DEEP-LEARNING PARA REALIZAR A PREDIÇÃO DE REFATORAÇÕES

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós Graduação em Ciência da Computação para obtenção do título de Mestre.

Dr. Rafael Serapilha Durelli
Orientador

Dr. Dilson Pereira
Coorientador

LAVRAS – MG

2024

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Pereira, Lucas Rafael Rodrigues.
Uma abordagem em Deep Learning para realizar a predição de refatorações / Lucas Rafael Rodrigues Pereira. - 2024.
42 p.

Orientador(a): Rafael Serapilha Durelli.

Dissertação (mestrado) - Universidade Federal de Lavras, 2024.
Bibliografia.

1. Deep Learning. 2. Machine Learning. 3. Refatoração. I. Durelli, Rafael Serapilha. II. Título.

LUCAS RAFAEL RODRIGUES PEREIRA

UMA ABORDAGEM DE DEEP-LEARNING PARA REALIZAR A PREDIÇÃO DE REFATORAÇÕES

REALIZING REFACTORING PREDICTION THROUGH DEEP-LEARNING

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós Graduação em Ciência da Computação para obtenção do título de Mestre.

APROVADA em 17 de Abril de 2024

Prof. Dr. Diego Roberto Colombo Dias	UFSJ
Prof. Dr. Paulo Afonso Parreira Junior	UFLA
Prof. Dr. Johnatan Alves de Oliveira	ICTIN

Dr. Rafael Serapilha Durelli
Orientador

Dr. Dilson Pereira
Coorientador

LAVRAS – MG
2024

AGRADECIMENTOS

Gostaria de dizer que estou muito agradecido ao meu orientador Professor e Dr. Rafael Durelli por ter desempenhado tal encargo com dedicação e amizade, apoio contínuo e paciência durante o desenvolvimento deste trabalho, enriquecendo o meu processo de aprendizado, pois, o imenso conhecimento e experiência me encorajaram a estudar cada vez mais. Igualmente gostaria de agradecer a minha família que, como eu acredita que as pessoas são transformadas pelo conhecimento, em especial, a meus pais pelo incondicional apoio para a conclusão de mais essa etapa. Por fim, agradeço a UFLA e todos que de alguma foram auxiliaram para a realização deste trabalho. Muito obrigado.

RESUMO

A pesquisa mostrou que a refatoração geralmente leva a uma maior capacidade de manutenção, resultando em um código mais legível e compreensível para os desenvolvedores. No entanto, ao aplicar métodos de refatoração para aumentar a qualidade do software, os desenvolvedores enfrentam desafios na identificação de métodos de refatoração eficazes. Acontece que encontrar oportunidades de refatoração é uma tarefa desafiadora. Um problema notável é a ausência de diretrizes específicas e práticas para determinar o método de refatoração apropriado para um determinado trecho de código. Consequentemente, já que as decisões sobre quando refatorar geralmente são baseadas em conceitos subjetivos, como codesmells, desenvolvedores menos experientes frequentemente contam com a orientação de desenvolvedores seniores para determinar quando o software precisa passar por refatoração. Pesquisas anteriores mostraram que algoritmos de aprendizado de máquina podem ser usados para ajudar os desenvolvedores a identificar oportunidades de refatoração. Com os recentes avanços em hardware, algoritmos de aprendizado profundo têm atraído cada vez mais atenção. Nessa pesquisa, pretendemos avaliar a eficácia de alguns modelos de *Deep Learning* (CNN, RNN, LSTM e DenseLayer) na previsão de oportunidades de refatoração, em comparação com modelos tradicionais de aprendizado de máquina. Especificamente, avaliamos esses modelos usando métricas padrão, como precisão, recall e exatidão. Nossas descobertas parecem sugerir que, embora os modelos de aprendizado de máquina geralmente superem os modelos de aprendizado profundo, os últimos apresentam desempenho superior aos primeiros quando treinados em conjuntos de dados não balanceados.

Palavras-chave: deep-learning; inteligência artificial; engenharia de software; refatoração; qualidade de software.

ABSTRACT

Research has shown that refactoring often leads to greater maintainability, resulting in more readable and understandable code for developers. However, when applying refactoring methods to increase software quality, Developers face challenges in identifying effective refactoring methods. It turns out that finding refactoring opportunities is a challenging task. One notable problem is the absence of specific, practical guidelines for determining the appropriate refactoring method for a given piece of code. Consequently, since decisions about when to refactor are often based on subjective concepts such as codesmells, Less experienced developers often rely on guidance from senior developers to determine when software needs to undergo refactoring. Previous research has shown that machine learning algorithms can be used to help developers identify refactoring opportunities. With recent advances in hardware, Deep learning algorithms have attracted more and more attention. In this research, we intend to evaluate the effectiveness of some *Deep Learning* models (CNN, RNN, LSTM and DenseLayer) in predicting refactoring opportunities, compared to traditional machine learning models. Specifically, We evaluate these models using standard metrics such as precision, recall, and accuracy. Our findings seem to suggest that although machine learning models generally outperform deep learning models, the latter perform better than the former when trained on unbalanced datasets.

Keywords: deep-learning; artificial intelligence; software engineering; refactoring; software quality.

INDICADORES DE IMPACTO

Destaca-se como um impacto tecnológico positivo gerado por esse trabalho a evolução do estado da arte no que se diz respeito à refatorações e uso de *deep-learning* em qualidade de software. A geração de um *dataset* contendo mais de 10.000 refatorações de código JAVA classificadas, com dois milhões de linhas de códigos classificadas e um repositório contendo os códigos utilizados na pesquisa para gerar a predição dos resultados, permitem qualquer pessoa com acesso à internet a disponibilização do material para replicar e aprimorar o experimento. O trabalho ainda fomenta a expansão da pesquisa para adicionar refatorações que não foram utilizadas no trabalho, como refatorações de classe e variáveis, e para linguagens de programação diferentes da utilizada no trabalho. Espera-se com este trabalho reafirmar a importância da qualidade de software, da refatoração de software e o uso da inteligência artificial como uma aliada ao desenvolvimento inteligente de software, com códigos mais limpos e melhor estruturados.

IMPACT INDICATORS

The evolution of the state of the art with regard to refactorings and the use of deep-learning in software quality stands out as a positive technological impact generated by this work. The generation of a dataset containing more than 10,000 classified JAVA code refactorings, with two million lines of classified code and a repository containing the codes used in the research to generate the prediction of results, allows anyone with internet access to make the material available to replicate and improve the experiment. The work also encourages the expansion of research to add refactorings that were not used in the work, such as class and variable refactorings, and for programming languages other than the one used in the work. This work is expected to reaffirm the importance of software quality, software refactoring and the use of artificial intelligence as an ally to intelligent software development, with cleaner and better structured codes.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Segundo Fowler (FOWLER, 2018) a refatoração de um software pode ser definida como um processo de alteração da estrutura interna, a fim de melhorar sua qualidade, sem alterar o comportamento externo do mesmo. Ao longo dos anos estudos têm estabelecido uma correlação entre operações de refatoração e qualidade de software (ALOMAR et al., 2019; KATAOKA et al., 2002; LEITCH; STROULIA, 2004). Entretanto, ainda assim, a refatoração não é tratada como uma prioridade para os times de desenvolvimento (LOPES; HORA, 2022).

Decidir quando e o que refatorar é o maior desafio dos desenvolvedores, vez que se trata de iniciar o processo de refatoração (LOPES; HORA, 2022). Times de desenvolvimento de software tendem a encarar a refatoração como dívida técnica e como qualquer outra alteração, a refatoração exige custos. Ainda mais quando se desconhece o tamanho ou impacto da refatoração, já que quanto mais postergada, mais impactos o código "defeituoso" poderá gerar (LOPES; HORA, 2022).

Para auxiliar no processo de refatoração e reduzir custos, desenvolvedores têm utilizado ferramentas de análise estática, tais como Sonarqube (S.A, 2008–2022), PMD (DANGEL BBG, 2022) e ESLint (ZAKAS BRANDON MILLS, 2022)(BELLER et al., 2016). Entretanto, essas ferramentas apresentam estratégias baseadas em *code smells* comuns e catalogados, tais como *God Class* e *Long Method*. Os *code smells* por si só, são motivos suficientes para refatorar um código, mas não são a única razão, o que torna a estratégia de utilizar ferramentas de análise estática e baseadas em heurística, não tão eficaz (JOHNSON et al., 2013).

1.1 Motivação

As ferramentas utilizadas para a análise estática frequentemente apresentam uma grande quantidade de falsos positivos ao indicar recomendações de refatoração, resultando na perda de confiança por parte dos desenvolvedores (DO; WRIGHT; ALI, 2022). Essas ferramentas geralmente possuem algoritmos baseados em um único princípio, como o PMD, que considera um método com mais de cem linhas como 'Long Method' sem realizar outras análises (DANGEL BBG, 2022). Ainda que essas ferramentas ofereçam um certo grau de customização, ainda assim é provável que haja falsos positivos tendo em vista a complexidade dos softwares atuais, que envolvem uma variedade muito grande de classes, *frameworks* e pacotes (DO; WRIGHT; ALI, 2022).

O uso de *Deep Learning* no contexto de engenharia de software vem conquistando relevância nos últimos anos, com a criação de novos modelos para solucionar problemas de qualidade de software, definição de arquitetura, modularização e resolução de problemas específicos de software (ARPTEG et al., 2018).

Pesquisadores vêm tentando a abordagem de Inteligência Artificial para recomendação de pontos de refatoração, utilizando técnicas como algoritmos de busca (MARIANI; VERGILIO, 2017; O'KEEFFE; CINNÉIDE, 2008), *pattern minner* (BAVOTA et al., 2014) e *Machine Learning* (ANICHE et al., 2020). Entretanto, ao utilizar *Deep Learning* como técnica de *Machine Learning*, acredita-se que é possível obter um resultado favorável em relação à predizer refatoração de código de software, ainda mais quando aplicado em diferentes áreas de engenharia de software, como predição de defeitos (D'AMBROS; LANZA; ROBBES, 2012), compreensão de código(LIU et al., 2019) e *code smells* (AZEEM et al., 2019).

1.2 Problemática

Atualmente, o processo de refatoração é predominantemente realizado por meio de ferramentas estáticas. No entanto, nos últimos anos, têm surgido diversas pesquisas que exploram o potencial do *Machine Learning* e *Deep Learning* para aprimorar e auxiliar o processo de refatoração. Diante de tal contexto, o principal objetivo desta dissertação de mestrado é responder se é viável prever a necessidade de refatoração por meio de um modelo de *Deep Learning*.

Para responder essa pergunta, o trabalho foi dividido em três menores perguntas:

RQ_1 - Como transformar os códigos de software em uma entrada válida para um modelo de *Deep Learning*?

Nem sempre é ideal utilizar dados em linguagem natural para o treinamento de modelos de inteligência artificial. Além de tornar o modelo mais complexo, torna o pré-processamento e a avaliação dos resultamos mais difícil.

Para resolver este problema, foram analisados dois modelos de representação de código-fonte foram analisados CODEBERT(FENG et al., 2020) e Code2vec(ALON et al., 2019).

O CODEBERT(FENG et al., 2020) foi escolhido por sua capacidade de transformar códigos de software em vetores, aplicar pesos nos vetores em relação à linguagem escolhida, e por disponibilizar modelos pré-treinados em JAVA. Deste modo, invés de passarmos diretamente um código de software para o modelo, o CODEBERT(FENG et al., 2020) proporciona uma entrada mais viável e de treinamento mais fácil, vez que é possível aplicar algoritmos de *oversampling*, *undersampling*, *CNN*, dentre outros algoritmos que funcionam apenas com vetores numéricos.

RQ_2 - Qual modelo de *Deep Learning* irá performar melhor predizendo refatoração em código de software?

Utilizando técnicas de pré-processamento para lidar com o desbalanceamento das classes, combinadas com redes neurais, LSTM, normalização de Batch e Pooling foi possível alcançar uma acurácia

de 71.94% e uma precisão de 77.48%, o que pode-se considerar otimista em um cenário onde os dados de teste são desbalanceados.

RQ_3 - Como o modelo de *Deep Learning* performa comparado aos modelos de aprendizado de máquina

Para avaliação dos resultados do modelo utilizando *Deep Learning* em relação aos modelos de aprendizado de máquina, foi escolhido um trabalho(ANICHE et al., 2020) que utiliza a mesma base de dados e apresenta resultados para vários algoritmos de *Machine Learning* diferentes.

1.3 Organização da dissertação

Além deste capítulo, o trabalho segue a seguinte organização:

Capítulo 2: Fundamentação Teórica: apresenta conceitos utilizados para criação deste trabalho. Tópicos discutidos incluem refatoração de código, *code smells* e refatorações de software baseadas em *Deep Learning*.

Capítulo 3: Metodologia: apresenta o processo de desenvolvimento do projeto, desde a estruturação, até a escolha dos repositórios, mineração dos repositórios, pré-processamento dos dados, treinamento dos modelos de *Deep Learning* e avaliação da performance dos modelos escolhidos.

Capítulo 4: Resultados: resultados do processo de execução dos modelos de *Deep Learning*. Os resultados gerados pelos modelos são gerados e comparados com modelos de *Machine Learning* e predições de desenvolvedores para melhor avaliar o desempenho dos modelos. Também são discutidos as ameaças à validação do projeto e pontos para serem estudados em trabalhos futuros.

Capítulo 5: Trabalhos relacionados: trabalhos que ajudaram na formação do conhecimento para esta pesquisa. Os resultados obtidos por trabalhos relacionados foram analisados e comparados para que o resultado desta pesquisa fosse o melhor possível.

Capítulo 7: Replicação: disponibilização dos recursos utilizados para implantação dos modelos no Google Colab.

Capítulo 7: Conclusão: conclusão do projeto, com resumo das etapas, compilado dos resultados, contribuição do projeto e trabalhos futuros descobertos ao decorrer do projeto.

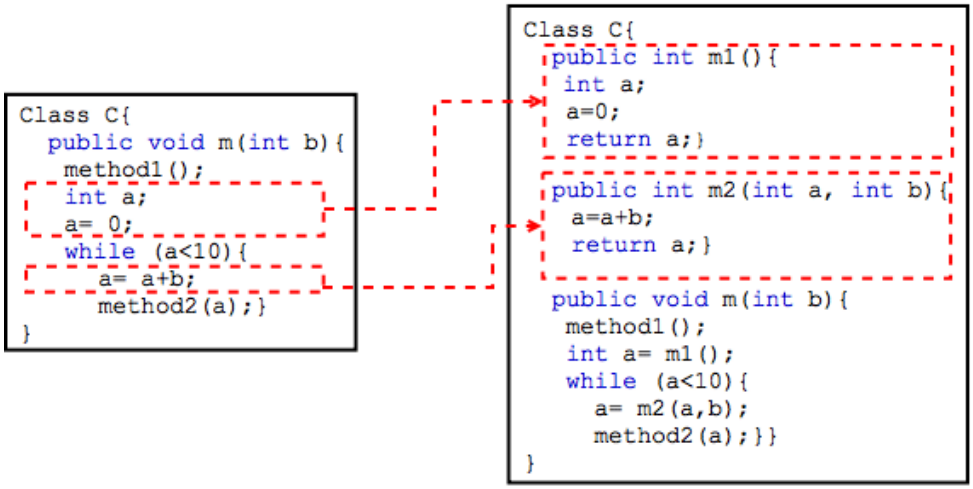
2 FUNDAMENTAÇÃO TEÓRICA

Os temas: Refatoração de código, *Code Smells*, Refatoração baseada em *Deep Learning*, Processamento de Linguagem Natural e Mineração de *Datasets* foram escolhidos baseados em trabalhos similares e na revisão da literatura.

2.1 Refatoração de código

No livro *Refactoring: Improving the Design of Existing Code*, de Fowler (FOWLER, 2018) a Refatoração é descrita como o processo de modificação do software para melhorar a sua estrutura interna, sem prejudicar o comportamento externo. A principal razão pela qual devemos nos preocupar em realizar uma refatoração no código, principalmente no ciclo de desenvolvimento, é aumentar a qualidade do código, evitar a presença de defeitos, aumentar a escalabilidade e facilidade de manutenção futura no código. Na Figura 1 conseguimos ver um exemplo de uma refatoração do tipo EXTRACT METHOD, onde partes de um método são removidos do método original e inseridos em métodos próprios ao contexto, porém não afetando o comportamento atual do código de software.

Figura 1 – Refatoração EXTRACT METHOD



Fonte: SELMADJI (2019).

Entretanto, decidir quando e o que ser refatorado, é um desafio para grande parte dos desenvolvedores. Toda atividade de refatoração vem acrescida de custos, pois envolve um desenvolvedor experiente para analisar o código e o tempo necessário para analisar todo o contexto da aplicação, porém esse esforço raramente é remunerado como divida técnica (KIM; ZIMMERMANN; NAGAPPAN, 2012; KRUCHTEN; NORD; OZKAYA, 2012).

Sem se preocupar com a escalabilidade e com a qualidade do código, os desenvolvedores podem se deparar no futuro com um código tão complexo, que é praticamente impossível realizar uma alteração sem criar um impacto em outra parte do software (KIM; ZIMMERMANN; NAGAPPAN, 2012).

A importância da refatoração é percebida, pelo tempo economizado ao realizar uma alteração no código. O desenvolvedor gasta menos tempo tentando entender o código, o que faz com que o investimento gasto na refatoração ou em processos de organização durante o desenvolvimento, sejam recuperados a curto e longo prazo (KIM; ZIMMERMANN; NAGAPPAN, 2012).

A literatura atual discute diferentes formas de abordar a refatoração de código de software. Algumas dessas abordagens são orientadas a ferramentas customizáveis e outras abordagens mais recentes, focam em automação do processo de refatoração. A remoção de *code smells* presentes no código por exemplo, pode ser indicada de forma automática por uma ferramenta, porém, estudos mais recentes abordam o uso de *Machine Learning* e algoritmos baseados em busca ou heurísticas (BAQAIS; ALSHAYEB, 2020). Com a popularização do uso de algoritmos de *Machine Learning* para descobrir pontos de refatoração no software, este estudo foca em aplicar a técnica de *Deep Learning*, que ainda é pouco explorada na literatura, para verificar sua eficácia na detecção de pontos de refatoração em código de software.

2.2 Code Smells

Smell Effect é o termo utilizado por (SANTOS et al., 2018) para nomear o impacto de um code smell em código de software no processo de desenvolvimento. Em seu estudo, (SANTOS et al., 2018) analisou 64 estudos publicados entre 2000 e 2017 e uma das principais descobertas desta análise segundo os autores, é que a análise de code smells por desenvolvedores não é algo completamente confiável. O estudo indica que os desenvolvedores tem pouco conhecimento do que são os *code smells* e apesar de conhecer um ou outro tópico, não possuem completo entendimento do assunto. Um estudo demográfico realizado pelo mesmo, indicou que a experiência do desenvolvedor impacta diretamente na detecção de *code smells*.

Para suprir a falta de experiência ou conhecimento do tema, desenvolvedores vêm procurando ajuda de ferramentas para detecção de *code smells* e ajudar na análise do código de software. (FERNANDES et al., 2016) realizou um estudo focado nas ferramentas de análise heurísticas para detecção de *code smells*. Em seu estudo, é analisado quais as principais causas e os tipos de *code smells* que as ferramentas de detecção de *code smells* mais utilizadas são capazes de detectar. Entretanto, seu estudo não aborda o uso de inteligência artificial, pois o tema é muito recente.

2.3 Refatoração baseada em Deep Learning

O uso de inteligência artificial para predição de refatoração em código de software é um tema relativamente novo. A interpretação de códigos por modelos de aprendizado de máquina foi possível apenas recentemente com abordagens como a do CODEBERT(FENG et al., 2020) que transformam código de software em vetores numéricos para serem pesados e avaliados pelos modelos de inteligência artificial.

Em uma revisão da literatura feita por (NAIK et al., 2023), revela uma análise de 17 estudos sobre o uso de técnicas de *Machine Learning* e *Deep Learning* para o uso de refatoração de código de software. O estudo aborda quais as técnicas foram utilizadas por cada estudo, qual linguagem de programação foi utilizada para treinamento, qual a avaliação do modelo e técnicas de pré-processamento. Neste estudo foi concluido que o uso de *recurrent neural netword*(RNN), *convolutional neural networks*(CNN), *multilayer perceptrons*(MLP) e *graph neural networks*(GNN) foram os modelos de *Deep Learning mais utilizados*, sendo o MLP o mais performático.

O estudo realizado por (NAIK et al., 2023) também revela que os estudo são em sua maioria baseados em JAVA, refatorações a nível de método e utilizam apenas uma linguagem de programação. Características estas, que são apresentadas no modelo desenvolvido por esse trabalho.

2.4 Processamento de linguagem natural

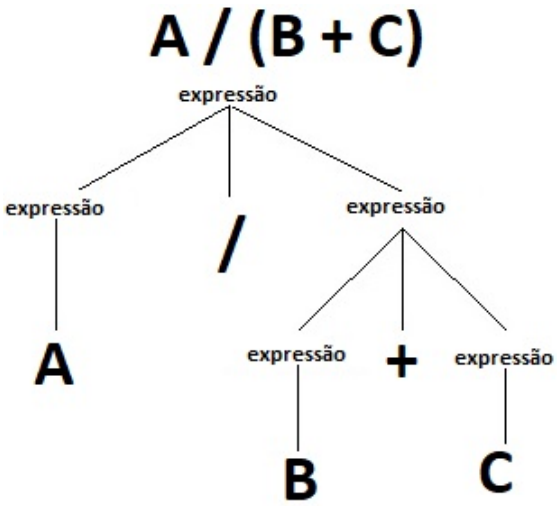
O Processamento de Linguagem Natural ou NLP(*Natural Language Processing*) é uma técnica utilizada no ramo da inteligência artificial que permite os computadores ler, interpretar e até mesmo gerar linguagem humana. Um exemplo da crescente utilização de NLP é a utilização de Chatbots no mercado, que vem cada vez mais ganhando espaço na área de atendimento ao consumidor de praticamente todas as áreas(ADAMOPOULOU; MOUSSIADES, 2020).

Entretanto o código-fonte em si não é uma linguagem natural como um idioma como o português. A análise da semântica e sintaxe devem ser feitas de forma diferentes e considerando os elementos específicos de cada linguagem de programação. Os algoritmos de NLP tendem a tratar texto como uma sequência linear de *tokens*, e alguns modelos de NLP utilizam essa mesma abordagem para código de software. Entretanto uma abordagem levando a sintaxe e a estrutura adequada para uma linguagem de programação podem obter resultados significantes comparados aos métodos mais simples(ALON et al., 2019).

Para fazer o arranjo de *tokens*, algoritmos de PLN como code2Vec(ALON et al., 2019) utilizam uma técnica chamada AST ou *Abstract Syntax Tree*, na qual o segmento de código é organizado em uma estrutura sintáxica de árvore de dados. Dessa forma, ao realizar a tokenização dos elementos presentes no código, o algoritmo de NLP consegue ponderar e organizar o código de acordo com a identação da

linguagem de software ou de acordo com a estrutura lógica presente no código. Na Figura2 temos a representação na forma de AST de uma expressão regular em linguagem natural.

Figura 2 – Representação de linguagem natural utilizando Árvore Sintática Abstrata



Fonte: Do Autor (2024)

2.5 Mineração de dataset

Data Minning é um assunto que tem chamado a atenção de vários pesquisadores nos últimos anos. Extrair informações de repositórios on-line permitiu a coleta de informações úteis e em grande quantidade. Com uma amostragem maior de dados, melhores decisões podem ser tomadas e estudos mais precisos podem ser realizados (TOMASEVIC; GVOZDENOVIC; VRANES, 2020; PUJARI, 2001).

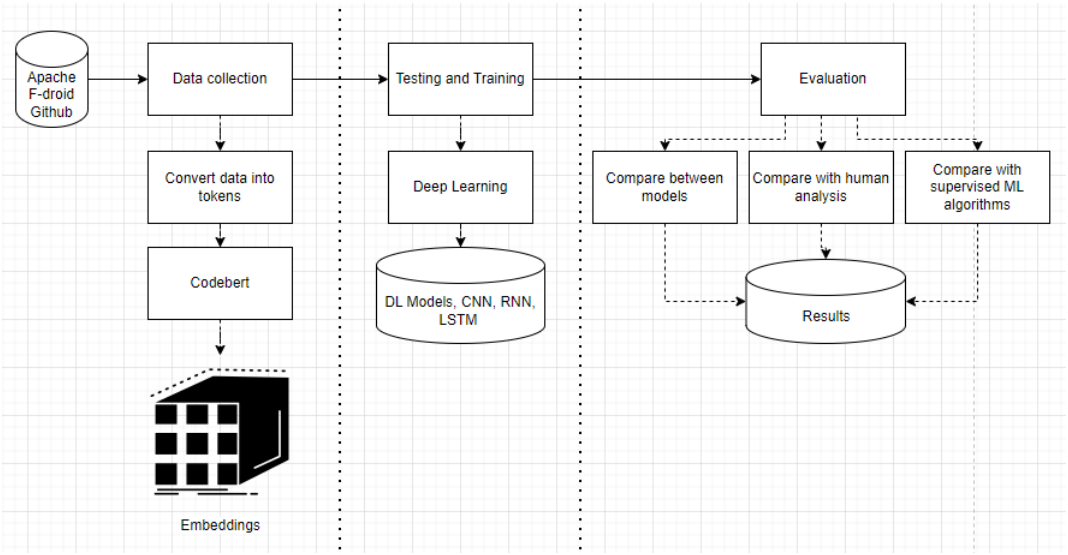
Apesar do enorme volume de dados contidos em repositórios on-line, a extração bem sucedida de informações ainda é um desafio para os pesquisadores, devido a necessidade de combinar as informações para gerar um objeto de análise (PONCIN; SEREBRENIK; BRAND, 2011). Percorrer *branchs*, analisar
o domínio, verificar a quantidade de alterações em cada arquivo, são tarefas que são de interesse para pesquisa, entretanto é um desafio conseguir tais informações sem recorrer a ferramentas externas ou heurísticas de mineração de dados.

Além dos desafios para extrair a informação, filtrar códigos funcionais e sem *"bugs"* requer uma análise prévia do repositório de dados. É necessário conferir se os dados extraídos pela mineração são confiáveis e se contém informações relevantes para a análise.

3 METODOLOGIA

A Figura 3 indica o modelo utilizado para responder as perguntas levantadas pela questão de pesquisa. O modelo é estruturado em três partes: Coleta de dados, Treinamento & testes e Avaliação do modelo. Cada uma dessas partes será explicada a seguir, e depois com mais detalhes em uma subseção dedicada.

Figura 3 – Metodologia de pesquisa



Fonte: Do Autor (2024).

A etapa de coleta de dados é focada na escolha do *dataset*, mineração dos repositórios escolhidos e pré-processamento dos dados. Uma vez que detectado uma refatoração dos tipos *extract method, rename method, move method e pull up method*, seriam coletados os métodos presentes no *commit* e uma rotulação para cada um desses métodos indicando a presença ou ausência de refatoração. Em seguida, esses métodos rotulados serão convertidos em tokens binários(0,1) e atribuídos um peso pré-treinado pelo CodeBERT(FENG et al., 2020).

Após a preparação dos dados, tem-se início a etapa de treinamento, onde os dados pré-processados são treinado e testados em diferentes algoritmos de *Deep Learning*. O tamanho e a quantidade das camadas da *Deep Learning* foram alterados de acordo com o algoritmo utilizado, como RNN, CNN e LSTM, a fim de obter os melhores resultados para cada modelo.

Por fim, os resultados dos modelos são coletados e a acurácia de cada modelo é avaliada. Para cada um dos modelos, foram avaliados a precisão, acurácia e *recall*. Para avaliar a performance geral do modelo, também foram utilizados os resultados de algoritmos de *Machine Learning*(ANICHE et al., 2020) e resultados coletados de desenvolvedores.

3.1 Amostragem Experimental

Tabela 1 – Amostragem utilizada no pesquisa

	Número de projetos	Quantidade de Commits
Apache	844	1.1471.203
F-droid	1.233	814.418
Github	9.072	6.517.597
Total	**11.149**	**8.803218**

Fonte: Do Autor (2024).

A fim de estabelecer uma comparação com os algoritmos de *Machine Learning*, foi utilizado a mesma amostragem estabelecida por (ANICHE et al., 2020). A amostragem contra com uma grande quantidade de projetos feitos em JAVA de três diferentes fontes:

a) o *Apache Software Foundation*(ASF) é uma organização que armazena e suporta a todos projetos Apache. O repositório conta com 860 projetos baseados em Java;

b) o F-Droid que é um repositório para código Android para aplicativos móveis. O repositório contém 1352 projetos de código de software livre;

c) o GitHub é o repositório gratuito mais popular do mundo contendo milhões de usuários registrados e diversos projetos em várias linguagens diferentes. Para o GitHub foi planejado utilizar os 10000 primeiros repositórios contendo unicamente código JAVA, porém as ferramentas conseguiram coletar com sucesso apenas 9072.

Com os três repositórios juntos, propõe-se uma grande variação em relação a quantidade e tamanho dos códigos, assim como complexidade, domínios, tecnologias utilizadas e equipes envolvidas. O resultado da amostragem é exibido na Tabela 1.

3.2 Extração de refatorações

O processo de coleta dos dados foi realizado pela ferramenta Refactoring Minner(TSANTALIS; KETKAR; DIG, 2020). A ferramenta clona os repositórios da lista de repositórios selecionados, verifica todos os tipos de refatoração no histórico da *branch master* e em seguida retorna as informações desejadas de acordo com a programação da ferramenta. Para cada repositório clonado, o Refactoring Minner(TSANTALIS; KETKAR; DIG, 2020) acessa a *branch master* e analisa todos *commits*, do mais antigo até o mais recente.

Esses *commits* são analisados em par, e a ferramenta consegue com 98% de acurácia determinar quais tipos de refatoração ocorreram entre esses dois *commits*. Após detectado uma refatoração, o código antes da refatoração é armazenado com um rótulo determinando o tipo da refatoração. Desta

forma o produto final da extração é uma tabela contendo em uma coluna diversos métodos presentes nos repositórios, e na outra uma marcação com zeros e uns indicando a presença ou ausência de refatoração.

Após a coleta dos dados, uma limpeza torna-se necessária uma vez que existem códigos duplicados, e métodos com rótulos contrários. Isso ocorre pois no processo de coleta, um método que foi selecionado por conter uma refatoração em um *commit* será marcado como sem refatoração quando examinado em outro *commit*. Para evitar redundâncias no processo de treinamento e confusão no processo de aprendizado, todos os códigos duplicados foram excluídos da base e todos os códigos duplicados porém com rótulos diferentes foram excluídos com exceção de uma única cópia contendo uma rotulação de refatoração.

3.3 Tokenização de código-fonte

Uma vez que os métodos foram coletados e devidamente classificados, é necessário ainda transformar os dados em uma entrada válida para os modelos de *Deep Learning*.

Código fonte e arquivos em linguagem natural exigem um tratamento para que possam ser utilizados por algoritmos de *Deep Learning* que apenas utilizam números reais.

Após uma revisão das ferramentas disponíveis para transformar código em vetor numérico, foram eleitas as duas melhores opções: CODEBERT(FENG et al., 2020) e Code2VEC(ALON et al., 2019).

O Code2VEC(ALON et al., 2019) separa o código fonte em palavras individuais utilizando identificadores para separar variáveis, nomes de funções, palavras reservadas e símbolos especiais. Uma vez que separadas, essas palavras passam a ser vetor de *tokens*. Em seguida, os *tokens* são mapeados em uma camada de *embeddings* onde são extraídos as informações semânticas e sintáticas em cima de cada palavra.

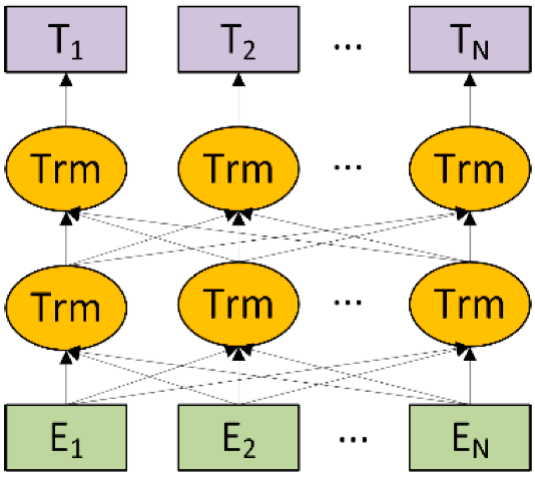
Após extrair os *tokens* o Code2VEC utiliza uma camada LSTM(HOCHREITER; SCHMIDHUBER, 1997) para gerar um vetor de contexto. Este vetor de contexto será utilizado em conjunto com o vetor de *tokens* para tentar prever os próximos *tokens*. Para prever qual a probabilidade de um *token* completar o contexto, o Code2VEC incorpora uma camada SOFTMAX para gerar a possível distribuição de probabilidade dos *tokens* de saída.

Entretanto, para efetivamente utilizar o Code2vec, é necessário treinar o vocabulário com os códigos fontes a serem utilizados no treinamento da *Deep Learning*.

Pela possibilidade de utilizar dados pré-treinados para o treinamento, poupando tempo e gastos com o trabalho, foi decidido que a ferramenta CODEBERT(FENG et al., 2020) ficaria responsável em transformar o código-fonte em um vetor numérico que servisse de entrada para os algoritmos de *Deep Learning*.

O CODEBERT(FENG et al., 2020) utiliza a arquitetura BERT(DEVLIN et al., 2018) para converter linguagem natural em vetores numéricos. O modelo BERT ajuda no pré-treinamento de linguagem natural ao utilizar uma arquitetura baseada em *transformers*, que utilizam os *transformers* em uma arquitetura *encoder-decoder* baseadas em um mecanismo de atenção. Vale ressaltar que os *transformers* utilizados na arquitetura BERT(DEVLIN et al., 2018) são bi-direcionais, que permitem extrair contexto da palavra tanto da direita pra esquerda, quanto da esquerda para a direita 4.

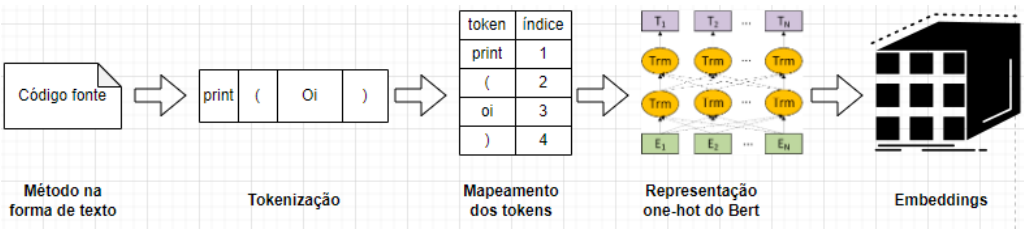
Figura 4 – Arquitetura BERT



Fonte: Feng (2020).

O CODEBERT vem com a linguagem JAVA pré-treinada pela arquitetura BERT, e o resultado desse mapeamento é um vetor *one-hot*, no qual cada *token* é representado por um ID com dimensão equivalente ao vocabulário. Os vetores *one-hot* são multiplicados por uma matriz de pesos treinada durante o processo de pré-treinamento e ajuste fino do CodeBERT. Essa matriz de pesos ou *embeddings* atribui valores reais aos elementos dos vetores *one-hot* permitindo a representação densa e contínua dos tokens, ou seja, ideal para algoritmos matemáticos de aprendizado de máquina. A representação do modelo pode ser observada na Figura 5.

Figura 5 – Processo de embedding utilizando CODEBERT



Fonte: Do Autor (2024).

3.4 Treinamento do modelo

Para o treinar a *Deep Learning* para predizer se um determinado método deve ou não sofrer refatoração do tipo EXTRACT METHOD, MOVE METHOD, RENAME METHOD ou MOVE UP METHOD, foram utilizados vários exemplos reais contendo métodos que sofreram refatoração e exemplos sem refatoração.

Os quatro algoritmos a seguir foram utilizados para treinar e avaliar a performance dos modelos:

a) CNN: Redes neurais convolucionais são amplamente utilizado no treinamento com imagens e visão computacional, mas por lidar bem com dados de classificação binária, acredita-se que tenha relevância para o treinamento (O'SHEA; NASH, 2015);
b) RNN: Redes neurais recorrentes, projetadas para lidar com dados sequenciais como texto e código de software. Possuem laços em sua estrutura que permitem que informações relevantes sejam passadas a diante (HOCHREITER; SCHMIDHUBER, 1997);
c) Dense layers: Camada densa é a camada básica em redes neurais profundas. Nela todos neurônios são conectados a todos os neurônios da camada anterios, e da camada seguinte e geram uma saída por meio de uma função de ativação;
d) LSTM: Variação das redes neurais recorrentes(RNN) com adaptação para lidar com sequências muito longas e retem informação por mais tempo que uma rede recorrente simples (HOCHREITER; SCHMIDHUBER, 1997).

Antes de utilizar qualquer um dos algoritmos, primeiro foi necessário lidar com o desbalanceamento natural dos dados, uma vez que a proporção entre métodos classificados como "com refatoração"foi dez vezes menor do que os classificados como "sem refatoração". Para lidar com esse desbalanceamento foi utilizado o algoritmo de *undersample* simples, que seleciona uma quantidade de métodos classificados como "sem refatoração"equivalente a quantidade dos classificados como "com refatoração"pareando o balanceamento em 50% para cada lado.

Em seguida os hiperparâmetros de cada um dos algoritmos foi ajustado e a quantidade camadas foi decidida pensando na complexidade do algoritmo e no melhor desempenho, a fim de gerar um resultado satisfatório.

Uma vez que o modelo foi treinado, o resultado para um determinado método, será a probabilidade de que este método deverá ou não sofrer a refatoração.

3.5 Avaliação

Para responder as questões elaboradas na introdução, foram comparados a média de precisão, acurácia e *recall* de cada um dos quatro algoritmos utilizados em diversas conFigurações e hiperparametrização. Cada um dos modelos foi comparado entre os outros modelos de *Deep Learning* a fim de

avaliar qual dos modelos performou melhor, e depois comparados com modelos de *Machine Learning* presentes na literatura (ANICHE et al., 2020).

Em adição, os resultados gerados pelos modelos, são comparados com uma pesquisa com a participação de desenvolvedores reais, a fim de levantar um novo olhar em cima da performance dos modelos. Esta comparação foi realizada com o entuito de avaliar a capacidade de prever as refatorações dos algoritmos de *Deep Lerning*

Destaca-se que o trabalho de (ANICHE et al., 2020) disponibiliza a precisão, acurácia e recall dos modelos, então não é necessário recalcular essas medidas para fins de comparação. Ao avaliar os resultados obtidos, com os realizados por esse estudo, é possível avaliar a performance dos modelos de *Deep Learning* utilizados, além de obter informações importantes sobre a efetividade dos modelos.

A acurácia pode ser vista como a taxa de sucesso do modelo avaliado. A acurácia é determinada ao calcular quantos resultados o modelo previu corretamente e dividí-los sobre a quantidade total de previsões. A fórmula da acurácia pode ser visto na Fórmula seguinte:

$$Acurácia = \frac{Previsões\ corretas}{Total\ de\ previsões}$$

A precisão mede a confiança do modelo em relação aos resultados positivos. A precisão é calculada ao medir quantos resultados da classe positiva o modelo previu, dividido pela quantidade de acertos. Com isso, podemos ter um valor de quantos métodos o modelo previu que havia uma refatoração, sobre a taxa real de métodos refatorados. A fórmula da precisão é dada por:

$$Precisão = \frac{Positivos\ Verdadeiros}{Positivos\ Verdadeiros\ +\ Positivos\ Falsos}$$

Por fim, o recall mede quantas instâncias da classe positivas, de fato eram positivas. A fórmula é obtida ao dividir a quantidade de positivos verdadeiros pela quantidade positivos(positivos verdadeiros ou positivos marcados como negativos). A fórmula do recall é dada por:

$$Recall = \frac{Positivos\ Verdadeiros}{Positivos\ Verdadeiros\ +\ Negativos\ Falsos}$$

3.6 Implementação e Execução

A implantação do RefactoringMinner (TSANTALIS; KETKAR; DIG, 2020) foi realizada em JAVA e todos dados coletados armazenados localmente em um banco de dados MySQL. Para extração dos métodos dos *commits*, foi elaborado um *script* em Python para detecção e separação dos métodos.

Por conta da integração entre JAVA, MySQL e Python, o tempo necessário para a execução do algoritmo de coleta utilizando uma máquina com 16gb de Ram, CPU Quadcore com 3.2GHz de processamento e SSD dedicados, levou em média 16 minutos por repositório para coletar todos repositórios estabelecidos. Aproximadamente um décimo dos repositórios demoraram mais de uma hora para execução, o que implicava em erros de conexão entre o MySQL e o JAVA ou Python.

Para o treinamento, foi utilizado o Google Collab Pro, com uma máquina virtual de 50gb de memória ram, CPU ajustável e 200gb de disco. A computação de cada um dos algoritmos em 200 épocas levou em torno de 24 horas.

4 RESULTADOS

Podemos responder as questões propostas na introdução analisando os resultados gerados pelos modelos de *Deep Learning*.

RQ_1 - Como transformar os códigos de software em uma entrada válida para um modelo de *Deep Learning*?

Após implementar e analisar a performance dos modelos de *Deep Learning* com os algoritmos de interpretação de código-fonte Code2Vec e CODEBERT obteve-se os resultados observados nas Tabelas 2 e 3

Tabela 2 – Precisão, Recall e Acurácia dos modelos de *Deep Learning* treinados utilizando Code2Vec

Code2Vec	Precisão	Recall	Acurácia
CNN	22.00	13.47	89.59
RNN	21.32	11.02	89.95
LSTM	71.72	00.96	52.93
Dense Layers	16.96	14.56	89.95

Fonte: Do Autor (2024).

Tabela 3 – Precisão, Recall e Acurácia dos modelos de *Deep Learning* treinados utilizando CODEBERT

CODEBERT	Precisão	Recall	Acurácia
CNN	70.90	64.83	67.68
RNN	63.55	68.65	64.64
LSTM	64.94	66.87	64.73
Dense Layers	65.61	64.76	65.45

Fonte: Do Autor (2024).

Observação 1: CODEBERT obteve uma performance melhor do que o Cod2Vec nos quatro modelos de *Deep Learning*. Os resultados observados indicam que o CODEBERT(FENG et al., 2020) obteve um melhor desempenho comparado ao Code2Vec. Por mais que os valores de acurácia do Code2Vec superem os valores os gerados pelo CODEBERT, a precisão e o *recall* indicam uma taxa de predição melhor do CODEBERT.

Observação 2: CODEBERT foi o modelo escolhido para interpretar os códigos-fontes utilizados neste trabalho. Além da performance superior ao Code2Vec, o CODEBERT não requer um treinamento local como o Code2Vec, isso torna a replicação do experimento mais confiável, tais como o reduzir o custo para o processamento do projeto.

RQ_2 - Qual modelo de *Deep Learning* irá performar melhor predizendo refatoração em código de software?

Nas Tabelas 4, 5, 6, e 7, são exibidos os valores de precisão, recall e acurácia obtidos utilizando cada um dos algoritmos para cada tipo específico de refatoração utilizado.

Tabela 4 – Precisão, Recall, e Acurácia dos modelos de Deep Learning para refatorações do tipo EXTRACT METHOD

	Precisão	Recall	Acurácia
CNN	67.30	79.740	70.48
RNN	65.31	74.240	67.36
LSTM	61.640	80.83	65.27
Dense Layer	65.86	77.56	68.62
Média	**65.03**	**78.10**	**67.93**

Fonte: Do Autor (2024).

Tabela 5 – Precisão, Recall, e Acurácia dos modelos de Deep Learning para refatorações do tipo MOVE METHOD

	Precisão	Recall	Acurácia
CNN	64.32	62.37	63.90
RNN	61.42	59.83	60.99
LSTM	61.23	56.43	60.36
Dense Layer	64.78	54.87	62.52
Média	**62.94**	**58.38**	**61.94**

Fonte: Do Autor (2024).

Tabela 6 – Precisão, Recall, e Acurácia dos modelos de Deep Learning para refatorações do tipo RENAME METHOD

	Precisão	Recall	Acurácia
CNN	74.50	55.36	64.38
RNN	63.55	68.36	64.50
LSTM	65.85	65.28	65.71
Dense Layer	63.01	60.57	62.61
Média	**66.73**	**62.34**	**64.30**

Fonte: Do Autor (2024).

Observação 3: LSTM foi o modelo com melhor resultado entre os modelos propostos. LSTM conseguiu uma média de acurácia de 64.73 %, precisão de 64.93% e recall de 66.87% durante o treinamento dos *datasets*. O segundo modelo que melhor performou foi o *Dense Layers* com resultados de 65.45% na acurácia, 64.94% na precisão e 66.87% no recall quando treinado com os mesmos dados.

Tabela 7 – Precisão, Recall, e Acurácia dos modelos de Deep Learning para refatorações do tipo PULL UP METHOD

	Precisão	Recall	Acurácia
CNN	77.48	61.86	71.94
RNN	62.93	72.15	65.71
LSTM	68.80	66.04	67.56
Dense Layer	71.05	61.95	68.04
média	**70.32**	**66.25**	**68.31**

Fonte: Do Autor (2024).

Observação 4: Todos os modelos conseguiram performar bem em relação ao desbalanceamento natural dos dados. Embora a proporção de métodos não refatorados fosse quase 90% em relação a quantidade de métodos refatorador com extração do método, a precisão de todos os modelos se mantiveram acima de 63.5%, com um recall mínimo de 64.76%.

RQ_3 - "Como o modelo de *Deep Learning* performa comparado aos modelos de *Machine Learning*?"

Para comparar a performance dos modelos de *Deep Learning* com os modelos de *Machine Learning*, iremos fazer uma comparação com os resultados obtidos com o trabalho de Aniche(ANICHE et al., 2020), que analisou e mapeou o resultado de 6 diferentes algoritmos de *Machine Learning*. Os resultados podem ser conferidos na Tabela 8

Tabela 8 – Precisão, Recall e Acurácia dos modelos de *Machine Learning*

	Precisão	Recall	Acurácia
Logistic Regression	0.80	0.87	0.82
SVM	0.77	0.88	0.80
Naive Bayes(gaussian)	0.65	0.95	0.70
Decision Tree	0.81	0.86	0.82
Random Forest	0.80	0.92	0.84
Neural Network	0.84	0.84	0.84
Média	**0.77**	**0.88**	**0.80**

Fonte: Do Autor (2024).

Na Tabela 9 podemos ver a comparação entre a média da acurácia, precisão e recall dos modelos de *Deep Learning* e *Machine Learning*. Todos os modelos utilizam a mesma base de dados do GitHub, Apache e F-droid, o que torna a comparação possível.

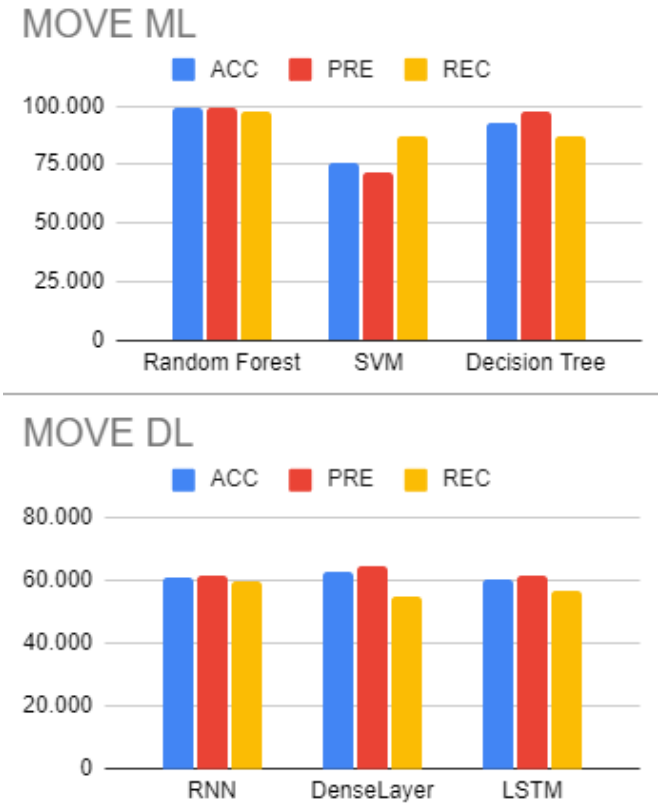
Nas figuras 6,7,8,9, exibimos uma comparação entre a média, precisão e acurácia dos modelos de *Deep Learning* e *Machine Learning*. Ambos modelos utilizam a mesma base de dados com refatorações extraídas do GitHub, Apache e F-droid.

Tabela 9 – Comparativo entre os modelos de *Deep Learning* e *Machine Learning*

	Precisão	Recall	Acurácia
Média Deep Learning Models	0.59	0.72	0.60
Média Machine Learning Models	0.77	0.88	0.80
Diferença	**-0.18**	**-0.16**	**-0.20**

Fonte: Do Autor (2024).

Figura 6 – Comparação entre ML e DL para refatorações MOVE



Fonte: Do Autor (2024).

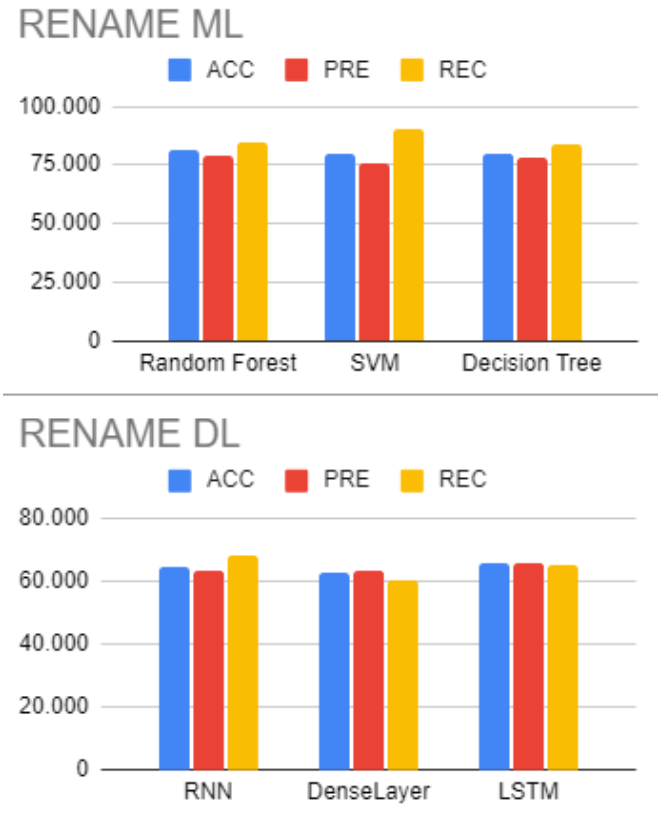
Na figura 6 mostramos um comparativo da performance entre os algoritmos de *Machine Learning* e *Deep Learning* para predizer uma refatoração do tipo MOVE. A comparação destaca que enquanto o modelo de *Deep Learning* oferece resultados promissores, os resultados apresentados pelos algoritmos de *Machine Learning* performam melhor que os algoritmos de *Deep Learning* para refatorações do tipo MOVE. Esses resultados sugerem que técnicas tradicionais de *Machine Learning* ainda levam vantagem em cima de algoritmos computacionalmente mais complexos para predizer alguns cenários de refatoração, particularmente refatorações do tipo MOVE, devido a sua capacidade de identificar padrões independente da complexidade e variância dos códigos de software.

Na figura 7 a comparação entre algoritmos de *Machine Learning* e *Deep Learning* demonstram uma precisão, recall e acurácia maior atribuida ao uso dos algoritmos de *Machine Learning*. Essa comparação sugere que o uso de *Machine Learning* pode ser mais adequada para tarefas que exigem identificação de nuâncias nos códigos de software como semântica, estrutura e lógica computacional.

A compração entre os algoritmos de *Machine Learning* e *Deep Learning* exibido na figura 8 exibe um intervalo menor entre os valores de precisão, recall e acurácia. Esta figura revela uma performance similar entre ambos algoritmos para refatorações do tipo EXTRACT. A análise sugere que a capacidade dos algoritmos de *Deep Learning* de identificar padrões complexos e reter uma quantidade maior de informação, pode ser uma abordagem interessante para tipos específicos de refatoração de código de software. Desafiando assim o uso dos algoritmos de *Machine Learning* nesta área.

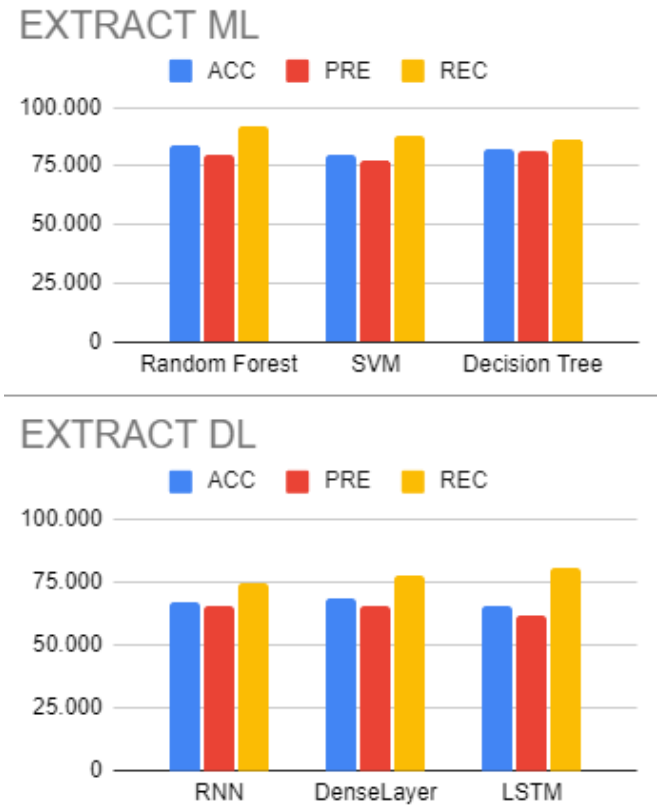
A figura 9 ilustra um cenário similar, onde os algoritmos de *Machine Learning* demonstram um resultado superior, entretando demonstra um potencial dos algoritmos de *Deep Learning* para desafiar o uso dos algoritmos de *Machine Learning*. Acredita-se que com refinamento dos modelos e treinamento em bases de dados diferentes, os algoritmos de *Deep Learning* podem performar ainda melhor, podendo desafiar os resultados dos algoritmos de *Machine Learning*.

Figura 7 – Comparação entre ML e DL para refatorações RENAME



Fonte: Do Autor (2024).

Figura 8 – Comparação entre ML e DL para EXTRACT



Fonte: Do Autor (2024).

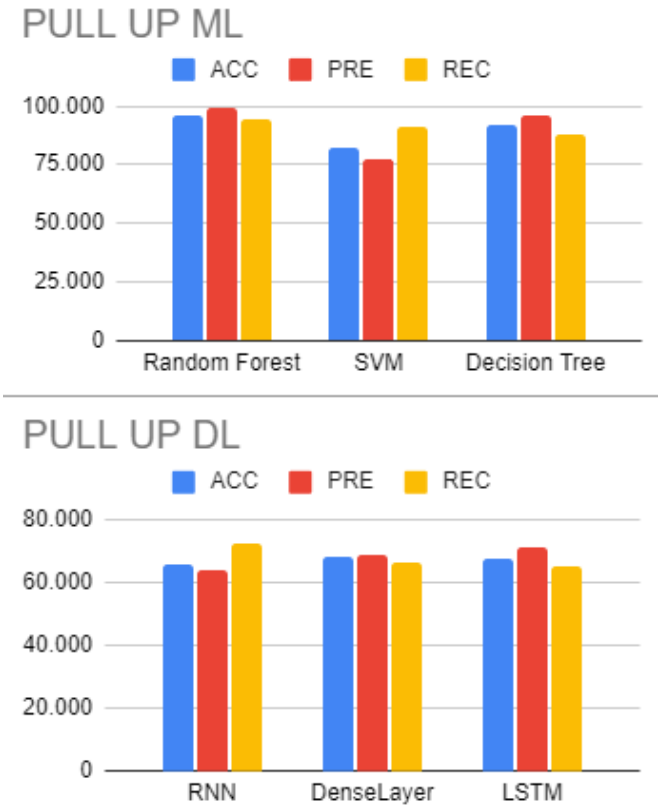
Observação 5: Modelos de *Machine Learning* demonstraram um desempenho melhor comparado aos modelos de *Deep Learning*. Como todos modelos compartilham a mesma base de dados, é possível fazer a comparação dos resultados, entretanto, acredita-se que com uma base de dados maior e mais diversa os algoritmos de *Deep Learning* podem apresentar uma melhora nos resultados.

A fim de trazer mais uma comparação dos resultados encontrados em relação a performance dos modelos, foram consultados 10 desenvolvedores JAVA e perguntado se determinado método deveria sofrer refatoração. Os métodos em questão foram extraídos da base de dados utilizada para o treinamento dos modelos. A resposta dos desenvolvedores pode ser analisada na seguinte Tabela 10

O nível de experiência dos desenvolvedores pode ser observado na Figura 10 e o formulário de perguntas utilizado pode ser encontrado no domínio: (FORMULáRIO. . . ,).

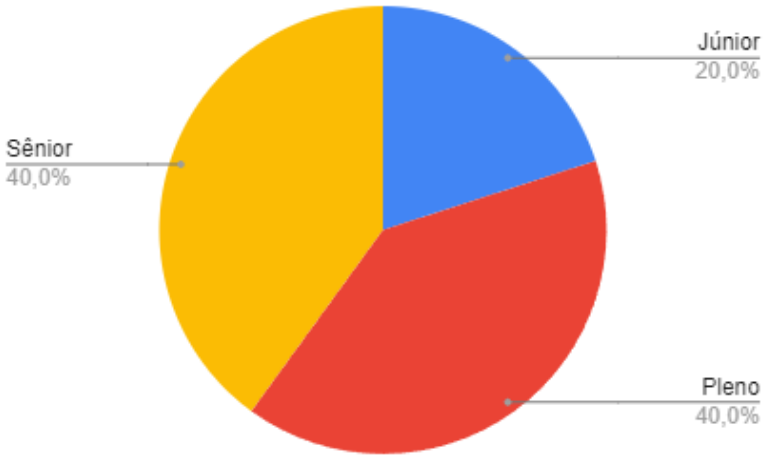
Observação 6: Desenvolvedores possuem uma taxa de acerto semelhante à dos modelos de *Deep Learning*. A taxa média de acerto dos desenvolvedores com 5 métodos retirados do conjunto de dados utilizado para o treinamento dos modelos de *Deep Learning* obteve uma diferença de aproximadamente 8%, como demonstrado na Tabela 11

Figura 9 – Comparação entre ML e DL para PULL UP



Fonte: Do Autor (2024).

Figura 10 – Nível de expertise dos desenvolvedores envolvidos na pesquisa



Fonte: Do Autor (2024).

Tabela 10 – Taxa de acerto dos desenvolvedores ao predizer uma refatoração analisada pelos modelos de *Deep Learning*

	Dev 1	Dev 2	Dev 3	Dev 4	Dev 5	Dev 6	Dev 7	Dev 8	Dev 9	Dev 10	Precisão
Método 1	1	1	1	1	1	1	1	1	1	1	1.0
Método 2	0	1	0	0	0	0	0	1	0	0	0.2
Método 3	1	1	0	1	0	1	0	0	1	1	0.6
Método 4	0	1	1	1	1	1	0	1	1	1	0.8
Método 5	1	1	1	1	1	1	0	0	1	1	0.8
Precisão	**0.6**	**1.0**	**0.6**	**0.8**	**0.6**	**0.8**	**0.2**	**0.6**	**0.8**	**0.8**	

Fonte: Do Autor (2024).

Tabela 11 – Comparação entre a precisão dos modelos de *Deep Learning* com a precisão média dos desenvolvedores

	Precisão
Média Deep Learning Models	0.5997
Média Desenvolvedores	0.6800
Diferença	**-0.0803**

Fonte: Do Autor (2024).

5 AMEAÇAS À VALIDADE

Nessa seção elencamos pontos de destaque na construção da validade do projeto.

5.1 Validade de construção

Ameaças relacionadas aos fatores teóricos e observados:

a) durante o processo de coleta dos repositórios, foi mencionado o uso da ferramenta Refactoring Miner(TSANTALIS; KETKAR; DIG, 2020). Essa ferramenta apresenta uma precisão de 98% e um recall de 87%, para o uso de EXTRACT METHODS, portanto ao coletar os repositórios não re-calculamos esses valores com base nos dados coletados;
b) a fim de comparar resultados com os algoritmos de *Machine Learning*, optamos em utilizar os mesmos repositórios utilizados por (ANICHE et al., 2020). Entretanto o processo de extração é custoso e pode acarretar em dificuldades para quem for minerar as mesmas instâncias de reposi-tório. Aproximadamente 8% dos repositórios falharam em ser coletados, mas este número pode variar de acordo com os recursos alocados para extração, tal como as ferramentas utilizados para mineração dos *datasets*.

5.2 Validade Interna

Ameaças à validade interna aborda fatores não considerados, mas que podem afetar as variáveis e os objetos analisados:

a) o desbalanceamento natural da classes de refatoração utilizadas apresenta muito mais instâncias de classes não refatoradas do que de classes refatoradas em uma proporção de 9 para 1. Para lidar com o desbalanceamento foram utilizados algoritmos de balanceamento dos dados sem a criação de valores genéricos, como *Random Undersampler*. Sabendo que pré-processamento dos dados podem levar à modelos menos precisos, a performance do modelo pode variar quando comparada a um cenário com base na realidade;
b) a ordem das refatorações não foi considerada para esse estudo. Acredita-se que o estado atual do método deva ser suficiente para a avaliação sobre a necessidade de refatoração dos tipos analisados;
c) por levar em consideração apenas o corpo do método para fazer a análise dos modelos de *Deep Le-arning*, pode-se perder informações relevantes externas ao método na hora de fazer a refatoração. Trabalhos futuros devem levar esse aspecto em consideração.

5.3 Validade Externa

Ameaças externas sobre fatores que influenciam na generalização dos resultados:

a) os resultados obtidos são baseados em projetos *Open Source*, o que pode afetar a generalização no contexto industrial. É necessário, entretanto, replicar esse estudo em um *dataset* contendo projetos de fato em uso pela indústria e com uma variação maior de domínios;
b) por escolher JAVA como objeto de estudo, os resultados encontrados podem não ser replicáveis para outras linguagens de programação. Acredita-se que os resultados podem ser similares para quaisquer linguagem baseada em orientação à objetos como o JAVA, entretanto trabalhos futuros devem levar esse fato em consideração.

6 TRABALHOS RELACIONADOS

Este estudo é focado na utilização de uma abordagem para detecção de pontos de refatoração, entretanto, outras pesquisas relacionadas utilizaram metodologias semelhantes que inspiraram o desenvolvimento deste projeto.

Em ANICHE et.al(2020) a pesquisa utiliza técnicas de mineração de repositórios on-line para selecionar códigos de software que sofreram algum tipo de refatoração e, após modelar e classificar os dados, utilizaram técnicas diferentes de *Machine Learning* para predizer quais locais nos códigos coletados, que seria interessante uma refatoração. O estudo faz uma comparação entre seis técnicas diferentes de *Machine Learning(Logistic Regression, Naive Bayes, Support Vector Machine, Decision Trees, Random Forest e Neural Network)* com um dataset com mais de duas milhões de refatorações. Na Tabela 12 é possível ver o quão performático os algoritmos de *Machine Learning* foram ao prever refatorações no repositório escolhido, ainda quando treinado em um repositório e testado em outro para validar a generalização do algoritmo. Essa pesquisa disponibiliza o *dataset* utilizado para que, ao treinar a *Deep Learning* desenvolvida para este projeto, possa ser comparado o resultado com o treinamento da *Machine Learning*.

Tabela 12 – Precisão(Pr) e Recall(Re) média dos modelos de Random Forest, quando treinados em um dataset e testado em outro

	Apache		GitHub		F-Droid	
	Pr	Re	Pr	Re	Pr	Re
Apache	-	-	0.84	0.79	0.77	0.70
GitHub	0.87	0.84	-	-	0.84	0.80
F-droid	0.77	0.73	0.81	0.76	-	-

Fonte: Aniche (2020)

Outros artigos também focam no uso de inteligência artificial para detectar um ponto específico de refatoração, como é o caso da pesquisa (JIANG; LIU; JIANG, 2019) que foca em utilizar técnicas de *Machine Learning* para sugerir qual o melhor nome para um método, evitando assim uma refatoração do tipo *"Rename Method"*. A pesquisa utiliza a técnica code2Vec (ALON et al., 2019) para transformar o código de software em um vetor que alimenta a inteligência artificial. Essa pesquisa foi importante para entender a necessidade de trabalhar com o código-fonte tokenizado invés de linguagem natural, pois com os pesos utilizados no treinamento do code2Vec o algoritmo foi capaz de performar melhor e entender o contexto, para sugerir uma refatoração e um novo nome para método. O estudo também destaca a performance do algoritmo com *tokens* pouco utilizados comparado aos *tokens* mais utilizados, no qual o algoritmo treinado, performa melhor com *tokens* mais comuns, pois ajudam na formação de contexto.

Autores como (BAVOTA et al., 2014) utilizam técnicas como *pattern minning* para predição de refatorações, dentre outras técnicas, para aprimorar o estado da arte. Estudos recentes focam em aplicar cada vez mais métricas, técnicas e heurísticas para comparar e melhorar os resultados de estudos anteriores. Uma das informações exibidas no estudo de (BAVOTA et al., 2014) é a relação entre componentes do código-fonte e a relação que estes possuem com determinado tipo de refatoração. Outra contribuição para literatura, é o levantamento de quais algorítmos estão sendo utilizados para predizer cada tipo de refatoração, como visto no quadro 1

Quadro 1 – Algoritmos abordados por tipo de refatoração

Tipo de Refatoração	Abordagem
Extract class	Clusterização, Grafos
Move method	Heurístico, Busca
Extract method	"Slicing"
Extract package	Grafo
Move class	Heurístico, Busca
Múltiplas operações	Busca

Fonte: Bavota (2014)

7 REPLICAÇÃO

Todos os modelos utilizados podem ser acessados e executados pelo Google Colab. Os links disponíveis abaixo levam aos respectivos modelos:

Rename Method: <https://colab.research.google.com/drive/1S_w-8uJd2K9pfYNnb9S7H7KEibxNvDOg?usp=sharing>;

Pull Up Method: <https://colab.research.google.com/drive/1VKm8wYHkH5ojm1zU2kLY2gAG0giLuriV?usp=sharing>;

Extract Method: <https://colab.research.google.com/drive/1Hq_LKe17gv3GoZ2YNOuBmYsxOv465Jxt?usp=sharing>;

Move Method: <https://colab.research.google.com/drive/1HBB-vuKfS51p5GymKDjnEmPFmXJaZKUl?usp=sharing>.

Por mais que os modelos foram disponibilizados no Google Colab, destaca-se que a execução dos códigos podem exigir a alocação de uma assinatura do Google Colab para adiquirir recursos de processamento mais fortes. O tempo de execução dos modelos também dependem dos recursos alocados e to tamanho das camadas.

8 CONCLUSÃO

Neste capítulo abordam-se os resultados deste trabalho, tais como seus resultados, contribuições, limitações e trabalhos futuros.

8.1 Síntese

Refatoração é uma tarefa comum, porém muito negligenciada durante o processo de desenvolvimento de Softwares. Mesmo com ferramentas de análise estática, desenvolvedores tendem a ter dificuldades em encontrar janelas para realizar refatorações do código de *software*. Com métodos e ferramentas cada vez mais sofisticados, e com a inteligência artificial sendo um assunto cada vez mais discutido, torna-se essencial o estudo de uma abordagem utilizando tais recursos.

Neste trabalho, foi proposto um modelo de *Deep Learning* capaz de predizer quando um método deve ser refatorado por EXTRACT METHOD. Para atingir tal objetivo, foram realizados estudos em torno da viabilidade do projeto e suas necessidades. Uma vez que o estudo foi realizado e as necessidades foram levantadas, o modelo foi elaborado, implementado e avaliado.

Durante a elaboração do modelo, três perguntas nortearam o processo de desenvolvimento. A primeira pergunta **"Como transformar os códigos de software em uma entrada válida para um modelo de *Deep Learning*."** levantou pontos abordados na metodologia e nos resultados. O estudo de modelos capazes de interpretar código-fonte tornou-se essencial e após deliberar entre os modelos mais utilizados na literatura Code2Vec e CODEBERT, optamos pelo CODEBERT pela facilidade da implantação e por possuir um modelo pré-treinado. Além disso, o CODEBERT obteve resultados favoráveis quando comparado ao modelo Code2Vec que demonstraram um *recall* bem abaixo do CODEBERT.

Uma vez que temos os códigos-fonte transformados em vetores numéricos pelo CODEBERT, avançamos para a segunda pergunta: **"Qual modelo de *Deep Learning* irá performar melhor predizendo refatoração em código de software?"**, para tal, foram comparados os modelos RNN, CNN, LSTM e *DenseLayers* propostos na metodologia e os resultados podem ser encontrados no Capítulo 4. Em suma, os modelos LSTM e *DenseLayers* obtiveram um melhor resultado comparados aos modelos RNN e CNN. Acreditamos que a capacidade dos algoritmos LSTM e *DenseLayers* de lidar com entradas maiores, e reter mais informação durante o processo de treinamento influenciou nos resultados.

Após coletar os resultados individuais dos modelos de *Deep Learning* podemos avançar para a última pergunta: **"Como o modelo de *Deep Learning* performa comparado aos modelos de aprendizado de máquina?"**. Após analisar um trabalho(ANICHE et al., 2020) que analisou e comparou diversos modelos de *Machine Learning*, comparamos os resultados obtidos pelos algoritmos de *Deep Learning* com os resultados descritos por esse trabalho. A comparação revelou que os algoritmos de *Machine Learning* tiveram maior sucesso na predição das refatorações do tipo EXTRACT METHOD ao

utilizar a mesma base de dados, embora as abordagens necessitem de uma quantidade diferente de dados para performar ao máximo.

Portanto, acredita-se que os resultados exibidos por essa pesquisa tenham relevância ao predizer refatoração em código de software, e deva influenciar novas pesquisas na área de *Deep Learning* e qualidade de software.

8.2 Contribuições

Aprincipal contribuição deste trabalho é a metodologia para a predição das refatorações dos tipos EXTRACT METHOD, RENAME METHOD, MOVE METHOD E PULL UP METHOD. Os modelos estudados podem e devem ser ajustados de acordo com a necessidade de cada usuário, entretanto acredita-se que ao estudar qual melhor interpretador de linguagem JAVA, quais os melhores modelos de *Deep Learning* e comparar os resultados, pode gerar uma base para notos trabalhos futuros em relação a outros tipos de refatoração ou outro tipo de linguagem de software.

8.3 Limitações

O trabalho descrito é limitado ao treinamento dos repositórios escolhidos. É necessário uma investigação em relação a abordagem de softwares industriais e ambientes mais diversos para validar os resultados, entretanto acredita-se que a generalização dos repositórios escolhidas é suficiente para sustentar a pesquisa.

Por utilizar apenas o escopo do método para ditar a predição do software, o modelo não leva em conta outros métodos e elementos visíveis apenas a nível de classe. Isso pode levar a uma falta de informação na hora de predizer a refatoração, porém acredita-se que ao decidir se o método contém informação suficiente para ser extraída, toda informação deva estar retida no escopo do próprio método.

Apenas 10 indivíduos responderam o questionário, o que pode gerar problemas na amostragem da pesquisa. Todos indivíduos trabalham na mesma empresa, o que pode tornar a amostra variável, embora tenham níveis de experiência distintos.

8.4 Trabalhos Futuros

A metodologia adotada pelo trabalho gera uma classificação binária em cima de um determinado método em JAVA. Acredita-se que a solução utilizada possa funcionar para qualquer tipo de refatoração binária no qual a informação esteja inteiramente contida no escopo do método. Um estudo é necessário para o ajuste dos hiper-parâmetros utilizados pelos modelos, entretanto a coleta dos dados e o pré-processamento descritos na metodologia desta pesquisa podem ser reutilizados para a geração de novos trabalhos.

Por utilizar apenas o corpo do método na abordagem de classificação binária, perde-se a informação não visível ao método, como variáveis, elementos da classe e outros métodos vizinhos. Um estudo dessa abordagem pode gerar uma nova perspectiva em relação à abordagem de *Deep Learning* na predição da refatoração em código de software.

Por fim, um estudo dos demais tipos de refatoração torna-se necessário, uma vez que este estudo apenas analisa refatorações do tipo EXTRACT METHOD, RENAME METHOD, PULL UP METHOD E MOVE METHOD. Analisar os demais tipos de refatoração, pode levar a resultados que superem os modelos de *Machine Learning* e gerem mais estudos em relação ao uso da *Deep Learning* na predição de refatorações e demais assuntos relacionados à qualidade de software.

REFERÊNCIAS

ADAMOPOULOU, E.; MOUSSIADES, L. An overview of chatbot technology. In: SPRINGER. **IFIP international conference on artificial intelligence applications and innovations**. [S.l.], 2020. p. 373–383.

ALOMAR, E. A. et al. Do design metrics capture developers perception of quality? an empirical study on self-affirmed refactoring activities. **arXiv preprint arXiv:1907.04797**, 2019.

ALON, U. et al. code2vec: Learning distributed representations of code. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 3, n. POPL, p. 1–29, 2019.

ANICHE, M. et al. The effectiveness of supervised machine learning algorithms in predicting software refactoring. **IEEE Transactions on Software Engineering**, IEEE, 2020.

ARPTEG, A. et al. Software engineering challenges of deep learning. In: IEEE. **2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [S.l.], 2018. p. 50–59.

AZEEM, M. I. et al. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. **Information and Software Technology**, Elsevier, v. 108, p. 115–138, 2019.

BAQAIS, A. A. B.; ALSHAYEB, M. Automatic software refactoring: a systematic literature review. **Software Quality Journal**, Springer, v. 28, n. 2, p. 459–502, 2020.

BAVOTA, G. et al. Recommending refactoring operations in large software systems. In: **Recommendation Systems in Software Engineering**. [S.l.]: Springer, 2014. p. 387–419.

BELLER, M. et al. Analyzing the state of static analysis: A large-scale evaluation in open source software. In: IEEE. **2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.], 2016. v. 1, p. 470–481.

DANGEL BBG, J. M. S. D. C. F. P. R. R. S. A. 2022. <https://github.com/pmd/pmd>.

DEVLIN, J. et al. Bert: Pre-training of deep bidirectional transformers for language understanding. **arXiv preprint arXiv:1810.04805**, 2018.

DO, L. N. Q.; WRIGHT, J. R.; ALI, K. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. **IEEE Transactions on Software Engineering**, v. 48, n. 3, p. 835–847, 2022.

D'AMBROS, M.; LANZA, M.; ROBBES, R. Evaluating defect prediction approaches: a benchmark and an extensive comparison. **Empirical Software Engineering**, Springer, v. 17, n. 4, p. 531–577, 2012.

FENG, Z. et al. Codebert: A pre-trained model for programming and natural languages. **arXiv preprint arXiv:2002.08155**, 2020.

FERNANDES, E. et al. A review-based comparative study of bad smell detection tools. In: **Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering**. [S.l.: s.n.], 2016. p. 1–12.

FORMULáRIO de validação de modelos DL. <https://forms.gle/pL9HRfZnJmocQwgt6>. Acessado: 2021-06-11.

FOWLER, M. **Refactoring: improving the design of existing code**. [S.l.]: Addison-Wesley Professional, 2018.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural computation**, MIT press, v. 9, n. 8, p. 1735–1780, 1997.

JIANG, L.; LIU, H.; JIANG, H. Machine learning based recommendation of method names: how far are we. In: IEEE. **2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2019. p. 602–614.

JOHNSON, B. et al. Why don't software developers use static analysis tools to find bugs? In: IEEE. **2013 35th International Conference on Software Engineering (ICSE)**. [S.l.], 2013. p. 672–681.

KATAOKA, Y. et al. A quantitative evaluation of maintainability enhancement by refactoring. In: IEEE. **International Conference on Software Maintenance, 2002. Proceedings.** [S.l.], 2002. p. 576–585.

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. A field study of refactoring challenges and benefits. In: **Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2012. p. 1–11.

KRUCHTEN, P.; NORD, R. L.; OZKAYA, I. Technical debt: From metaphor to theory and practice. **Ieee software**, IEEE, v. 29, n. 6, p. 18–21, 2012.

LEITCH, R.; STROULIA, E. Assessing the maintainability benefits of design restructuring using dependency analysis. In: IEEE. **Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717)**. [S.l.], 2004. p. 309–322.

LIU, K. et al. Learning to spot and refactor inconsistent method names. In: IEEE. **2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)**. [S.l.], 2019. p. 1–12.

LOPES, M.; HORA, A. How and why we end up with complex methods: a multi-language study. **Empirical Software Engineering**, Springer, v. 27, n. 5, p. 1–42, 2022.

MARIANI, T.; VERGILIO, S. R. A systematic review on search-based refactoring. **Information and Software Technology**, Elsevier, v. 83, p. 14–34, 2017.

NAIK, P. et al. Deep learning-based code refactoring: A review of current knowledge. **Journal of Computer Information Systems**, Taylor & Francis, p. 1–15, 2023.

O'SHEA, K.; NASH, R. An introduction to convolutional neural networks. **arXiv preprint arXiv:1511.08458**, 2015.

O'KEEFFE, M.; CINNÉIDE, M. O. Search-based refactoring for software maintenance. **Journal of Systems and Software**, Elsevier, v. 81, n. 4, p. 502–516, 2008.

PONCIN, W.; SEREBRENIK, A.; BRAND, M. V. D. Process mining software repositories. In: IEEE. **2011 15th European conference on software maintenance and reengineering**. [S.l.], 2011. p. 5–14.

PUJARI, A. K. **Data mining techniques**. [S.l.]: Universities press, 2001.

S.A, S. 2008–2022. <https://sonarqube.org>.

SANTOS, J. A. M. et al. A systematic review on the code smell effect. **Journal of Systems and Software**, Elsevier, v. 144, p. 450–477, 2018.

TOMASEVIC, N.; GVOZDENOVIC, N.; VRANES, S. An overview and comparison of supervised data mining techniques for student exam performance prediction. **Computers & education**, Elsevier, v. 143, p. 103676, 2020.

TSANTALIS, N.; KETKAR, A.; DIG, D. Refactoringminer 2.0. **IEEE Transactions on Software Engineering**, 2020.

ZAKAS BRANDON MILLS, M. D. N. C. 2022. <https://eslint.org/>.