UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALTINO DANTAS BASÍLIO NETO

Aplicação de CNN e LLM na Localização de Defeitos de Software

Goiânia
2024

Processo: 23070.043391/2024-12 Documento: 4983127

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO (TECA) PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei 9.610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou download, a título de divulgação da produção científica brasileira, a partir desta data.

O conteúdo das Teses e Dissertações disponibilizado na BDTD/UFG é de responsabilidade exclusiva do autor. Ao encaminhar o produto final, o autor(a) e o(a) orientador(a) firmam o compromisso de que o trabalho não contém nenhuma violação de quaisquer direitos autorais ou outro direito de terceiros.

1. Identificação do material bibliográfico

[ ] Dissertação [X] Tese [ ] Outro*:_____

*No caso de mestrado/doutorado profissional, indique o formato do Trabalho de Conclusão de Curso, permitido no documento de área, correspondente ao programa de pós-graduação, orientado pela legislação vigente da CAPES.

Exemplos: Estudo de caso ou Revisão sistemática ou outros formatos.

2. Nome completo do autor

Altino Dantas Basílio Neto

3. Título do trabalho

Aplicação de CNN e LLM na Localização de Defeitos de Software

4. Informações de acesso ao documento (este campo deve ser preenchido pelo orientador)

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. Após esse período, a possível disponibilização ocorrerá apenas mediante:
a) consulta ao(à) autor(a) e ao(à) orientador(a);
b) novo Termo de Ciência e de Autorização (TECA) assinado e inserido no arquivo da tese ou dissertação.
O documento não será disponibilizado durante o período de embargo.
Casos de embargo:
- Solicitação de registro de patente;
- Submissão de artigo em revista científica;
- Publicação como capítulo de livro;
- Publicação da dissertação/tese em livro.

Obs. Este termo deverá ser assinado no SEI pelo orientador e pelo autor.

Documento assinado eletronicamente por **Altino Dantas Basilio Neto**, **Discente**, em 21/11/2024, às 15:28, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Celso Goncalves Camilo Junior**, **Professor do Magistério Superior**, em 22/11/2024, às 13:06, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **4983127** e o código CRC **33C49095**.

Referência: Processo nº 23070.043391/2024-12 SEI nº 4983127

ALTINO DANTAS BASÍLIO NETO

Aplicação de CNN e LLM na Localização de Defeitos de Software

Tese apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Doutor em Programa de Pós-graduação em Ciência da Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Celso Gonçalves Camilo Junior

Goiânia
2024

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Basílio Neto, Altino Dantas
Aplicação de CNN e LLM na Localização de Defeitos de Software [manuscrito] / Altino Dantas Basílio Neto. - 2024.
CLXXIX, 179 f.

Orientador: Prof. Dr. Celso Gonçalves Camilo Junior .
Tese (Doutorado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2024.
Bibliografia. Apêndice.
Inclui siglas, gráfico, tabelas, algoritmos, lista de figuras, lista de tabelas.

1. Localização de defeitos. 2. Redes Neurais Artificiais. 3. Redes Neurais Convolucionais. 4. Modelos de Linguagem de Grande Porte. I. Camilo Junior , Celso Gonçalves , orient. II. Título.

CDU 004

Processo: 23070.043391/2024-12 Documento: 4983122

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

ATA DE DEFESA DE TESE

Ata nº **32** da sessão de Defesa de Tese de **Altino Dantas Basílio Neto**, que confere o título de Doutor em Ciência da Computação, na área de concentração em Ciência da Computação.

Aos dezesseis dias do mês de outubro de dois mil e vinte e quatro, a partir das catorze horas, via webconferência, realizou-se a sessão pública de Defesa de Tese intitulada **"Aplicação de CNN e LLM na Exploração dos Aspectos-Chave da Localização de Defeitos de Software"**. Os trabalhos foram instalados pelo Orientador, Professor Doutor Celso Gonçalves Camilo Junior (INF/UFG) com a participação dos demais membros da Banca Examinadora: Professor Doutor Auri Marcelo Rizzo Vincenzi (DC/UFSCAR), membro titular externo; Professor Doutor Jerffeson Teixeira de Souza (UECE), membro titular externo; Professor Doutor Plínio de Sá Leitão Júnior (INF/UFG), membro titular interno; e Professor Doutor Sávio Salvarino Teles de Oliveira (INF/UFG), membro titular externo. A realização da banca ocorreu por meio de videoconferência. Durante a arguição os membros da banca não fizeram sugestão de alteração do título do trabalho. A Banca Examinadora reuniu-se em sessão secreta a fim de concluir o julgamento da Tese, tendo sido o candidato **aprovado** pelos seus membros. Proclamados os resultados pelo Professor Doutor Celso Gonçalves Camilo Junior, Presidente da Banca Examinadora, foram encerrados os trabalhos e, para constar, lavrou-se a presente ata que é assinada pelos Membros da Banca Examinadora, aos dezesseis dias do mês de outubro de dois mil e vinte e quatro.

TÍTULO SUGERIDO PELA BANCA

"Aplicação de CNN e LLM na Localização de Defeitos de Software"

Documento assinado eletronicamente por **Altino Dantas Basilio Neto**, **Discente**, em 21/11/2024, às 15:28, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Auri Marcelo Rizzo Vincenzi**, **Usuário Externo**, em 21/11/2024, às 15:48, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Savio Salvarino Teles De Oliveira**, **Professor do Magistério Superior**, em 21/11/2024, às 19:05, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Plinio De Sa Leitao Junior**, **Professor do Magistério Superior**, em 21/11/2024, às 20:50, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Jerffeson Teixeira de Souza**, **Usuário Externo**, em 21/11/2024, às 21:10, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Celso Goncalves Camilo Junior**, **Professor do Magistério Superior**, em 22/11/2024, às 13:06, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **4983122** e o código CRC **37D7A5E5**.

Referência: Processo nº 23070.043391/2024-12 SEI nº 4983122

Dedico este trabalho a Emanoel Dantas Feitosa, o sobrinho que foi um filho e nos deixou de forma abreviada, redefinindo para sempre o conceito de saudade e a Domingos Dantas de Souza, o tio que foi um pai, no amor, nos ensinamentos, nas longas e variadas conversas, que foi embora, mas nos deixou um legado de companheirismo, humildade e espírito de liberdade.

Agradecimentos

Aos familiares e amigos que sempre entenderam minha escolha de vida e apoiaram da forma que os competia, que me proporcionaram momentos de descontração, mas que também conciliaram minha condição de tempo restrita.

Aos colegas da fotografia aeronáutica e do vôlei, atividades que foram fundamentais para suportar o período pandêmico; em especial o sempre parceiro João Victor.

Aos ilustres professores Jerffeson Teixeira e Sávio Teles por aceitarem o convite para composição da banca avaliadora, assim como aos professores Auri Vincenzi e Plínio de Sá que já contribuíram enormemente no exame de qualificação.

Ao professor Celso Camilo, não apenas pela extraordinária compacidade técnica da qual pude me beneficiar, mas também e fundamentalmente pelas muitas oportunidades de atividades para além das fronteiras da universidade e, principalmente, pela humanidade demonstrada nas situações mais difíceis.

À FAPEG e ao CEIA pelas bolsas que viabilizaram, ainda que parcialmente, o financiamento da pesquisa.

Aos colegas do Lab 254 e do grupo I4Soft, que proporcionaram um ambiente de conhecimento e amizade, fundamentais para o desenvolvimento e consolidação de ideias técnicas e concepções sobre a vida.

Ao primeiro e grande amigo goiano, Diogo Freitas, pelo acolhimento, companheirismo e parceria em diferentes fases dessa jornada.

Às Marianas do INF: à Mariana Crisóstomo (prefiro Martins) pela companhia nos almoços do REUNI, amizade sincera e reflexões profundas, à Mariana Ramada por emprestar toda inteligência para conversas leves ou complexas e à Mariana Rodrigues pelo excelente serviço na secretaria.

Aos amigos Diogo Stelle e Vilson Soares, pelas incontáveis conversas, jantares depois de longos dias de laboratório e provas de companheirismo em diversos momentos.

Ao amigo Lucas Roque por ter iniciado essa caminhada comigo e ter sido o companheiro de todas as horas.

Ao amigo Marcos Vinícius pelas horas de laboratório, lanches da tarde e, principalmente, pelas muitas conversas repletas de desabafos, lamentações e compartilhamento de expectativas.

Ao amigo Eduardo Souza que, para além da amizade inexorável, foi o ouvido mais atento e o coração mais sensível para todos os percalços dessa batalha, não sem provocar e desenvolver pensamentos e reflexões sofisticados, que muitas vezes deram sentido ao percurso.

À Juliana Machado pelo apoio na mudança para Goiânia e às comunidades evangélicas Igreja da Paz e Luz Para os Povos que me acolheram de forma carinhosa.

À Aline Lemos pelo companheirismo em muitos momentos, especialmente, nos mais difíceis.

Às minhas companheiras dos momentos mais divertidos, Thayná Lucas e Carol Maceno, pelos incontáveis encontros de descontração, aventuras mais radicais e sofrimento nas provas de corrida, experiências fundamentais como válvula de escape.

Aos amigos Deuslírio Junior, Sávio Sampaio e Thiago Fernandes, não apenas pela parceria acadêmica frutífera, conversas sobre esta tese e suporte técnico, mas principalmente pelo fundamental apoio motivacional na reta final.

À Letícia Linhares, que mesmo de longe, assumiu a missão de motivadora implacável, fez valer os 12 anos de amizade e foi determinante para chegarmos até aqui.

À minha mãe, Rita Dantas, pela vida, dela.

Resumo

Basílio Neto, Altino Dantas. **Aplicação de CNN e LLM na Localização de Defeitos de Software**. Goiânia, 2024. 178p. Tese de Doutorado Relatório de Graduação. Programa de Pós-Graduação em Ciência da Computação, Instituto de Informática, Universidade Federal de Goiás.

O aumento na quantidade ou complexidade dos sistemas computacionais leva a um crescimento na ocorrência de defeitos em *software*. A indústria investe altas quantias na depuração de código e parte considerável do custo está associada à tarefa de localizar o elemento responsável pelo defeito. Técnicas automatizadas para a localização de defeitos são amplamente exploradas, com avanços recentes impulsionados pelo uso de modelos de aprendizado profundo que combinam diferentes informações sobre o código-fonte defeituoso. Contudo, a acurácia dessas técnicas ainda apresenta espaço para melhorias, sugerindo a existência de questões em aberto nesse campo. Este trabalho tem como objetivo formalizar e investigar os aspectos mais impactantes para técnicas de localização de defeitos, propondo um *framework* de caracterização de abordagens do problema e duas metodologias de solução: a) baseada em redes neurais convolucionais (CNNs) e b) baseada em modelos de linguagem de grande porte (LLMs). A partir de uma experimentação envolvendo conjuntos de dados públicos em linguagem Java e Python, demonstrou-se que as CNNs são capazes de se comparar com métodos tradicionais, porém se mostraram inferiores a outros métodos da literatura. Já a proposta baseada em LLM superou amplamente heurísticas como Ochiai e Tarantula e se mostrou competitiva com a literatura mais recente. Uma experimentação em cenário livre do problema de vazamento de dados mostrou que as abordagens baseadas em LLM podem ser aprimoradas pela combinação com a heurística Ochiai.

Palavras–chave

Localização de defeitos, Redes Neurais Artificiais, Redes Neurais Convolucionais, Modelos de Linguagem de Grande Porte.

Abstract

Basílio Neto, Altino Dantas. **Deep neural networks applied to software fault localization**. Goiânia, 2024. 178p. PhD. Thesis Relatório de Graduação. Programa de Pós-Graduação em Ciência da Computação, Instituto de Informática, Universidade Federal de Goiás.

The increase in the quantity or complexity of computational systems has led to a growth in the occurrence of software defects. The industry invests significant amounts in code debugging, and a considerable portion of the cost is associated with the task of locating the element responsible for the defect. Automated techniques for fault localization have been widely explored, with recent advances driven by the use of deep learning models that combine different types of information about defective source code. However, the accuracy of these techniques still has room for improvement, suggesting open challenges in the field. This work aims to formalize and investigate the most impactful aspects of fault localization techniques, proposing a framework for characterizing approaches to the problem and two solution methodologies: a) based on convolutional neural networks (CNNs) and b) based on large language models (LLMs). From experimentation involving public datasets in Java and Python, it was demonstrated that CNNs are comparable to traditional methods but were found to be inferior to other methods in the literature. The LLM-based approach, on the other hand, greatly outperformed heuristics like Ochiai and Tarantula and proved competitive with more recent literature. An experiment in a scenario free from the data leakage problem showed that LLM-based approaches can be improved by combining them with the Ochiai heuristic.

Keywords

Fault Localization, Artificial Neural Network, Convolutional Neural Networks, Large Language Model.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas e Figuras

ADD – *Abordagem Dependente de Descrição*
ANDD – *Abordagem Não Dependente de Descrição*
API – *Application Programming Interface*
AST – *Árvore Sintática Abstrata*
CNN – *Convolutional Neural Network*
IA – *Artificial Intelligence*
JSON – *JavaScript Object Notation*
LLM – *Large Language Model*
MLP – *Multilayer Perceptron*
PLN – *Processamento de Linguagem Natural*
PMC – *Perceptron Multicamadas*
ReLU – *Rectified Linear Unit*
RNA – *Redes Neurais Artificiais*
RNN – *Recurrent Neural Network*

CAPÍTULO 1

Introdução

1.1 Contextualização

Um software é um conjunto de artefatos que incluem os programas propriamente ditos, ou seja, sequência de instrução para realização de uma tarefa específica, mas também documentação de implementação e manual de utilização, dados de operação e configurações, tudo isso para viabilizar o funcionamento de computadores, celulares e outros dispositivos digitais (SOMMERVILLE, 2015)

O software desempenha um papel crucial em diversas esferas do cotidiano, com aplicações que variam de tarefas simples, como compras online, até operações sofisticadas, como diagnósticos médicos e previsões de desastres. Apesar da ausência de dados globalmente unificados que mensurem o volume de softwares no mundo, a crescente complexidade dos sistemas computacionais é acompanhada por um aumento na demanda por profissionais de software. Nos Estados Unidos, por exemplo, segundo o levantamento do U.S. Bureau of Labor Statistics (2023), o número de desenvolvedores aumentou de 677.900 em 2003 para 1.656.880 em 2023, uma tendência que reflete o cenário global.

Dentre as muitas atividades que os profissionais do desenvolvimento e sustentação de software desempenham, uma das mais relevantes e desafiadoras é denominada depuração do código. Segundo Agans (2002), a depuração de código é o processo de identificar, localizar e corrigir erros ou defeitos em um software, que podem ser erros de sintaxe, erros lógicos ou problemas que causam comportamentos inesperados.

Depurar o código é uma parte essencial do ciclo de desenvolvimento de software, e envolve técnicas como o uso de ferramentas de depuração, inserção de pontos de interrupção, e análise detalhada de logs e execuções passo a passo. Com o aumento da complexidade dos softwares e dos processos de desenvolvimento e manutenção, a tarefa de depuração representa um desafio importante, sendo responsável por uma parcela significativa do tempo e dos recursos gastos (HAILPERN; SANTHANAM, 2002).

A localização de defeitos por requerer a identificação precisa de elementos defeituosos como linhas de código, variáveis, instruções, função ou arquivos, consome

uma porção significativa do tempo dedicado à depuração, podendo representar até 95% desse tempo (MYERS; SANDLER; BADGETT, 2011). Um estudo realizado por Böhme et al. (2017) revela que desenvolvedores gastam cerca de 60% do tempo de depuração com a análise e localização de defeitos. Esses números evidenciam a importância de estratégias eficientes para automação dessa tarefa.

Como uma tarefa dispendiosa, inúmeras estratégias são desenvolvidas para automatizar esse processo com o objetivo de auxiliar especialistas em engenharia de software a encontrar e, posteriormente, corrigir as falhas. As abordagens para localização de defeitos de forma automática incluem técnicas baseadas em dados de cobertura de código (ABREU; ZOETEWEIJ; GEMUND, 2007; JANSSEN; ABREU; GEMUND, 2009; XIE et al., 2013), métodos que usam análise de mutantes (PAPADAKIS; TRAON, 2015; DE-FREITAS et al., 2018), fatiamento dinâmico do programa (AGRAWAL et al., 1995; MAO et al., 2014), rastreamento da pilha de erro gerada por IDE (WONG et al., 2014; WU et al., 2014) e análise de informação de relatório do defeito e histórico de desenvolvimento (KIM et al., 2007; RAHMAN et al., 2011).

Recentemente, trabalhos mais sofisticados exploram a combinação de informações de forma eficiente, utilizando técnicas de inteligência computacional para localização de defeitos. Entre essas técnicas, destacam-se a computação evolucionária (SILVA-JUNIOR et al., 2020), redes neurais profundas (ZHANG et al., 2021) e modelos de linguagem de grande porte (KANG; AN; YOO, 2024; YANG et al., 2024), que oferecem novas possibilidades ao combinar diferentes fontes de informação sobre o código defeituoso.

1.2 Justificativa

Ao ser analisado o cenário das soluções para localização de defeitos de software de forma automática, percebe-se que tipicamente as diversas soluções propostas se distinguem com relação ao ambiente ou contexto de aplicação, o tipo de informações que pode ser usada para a localização e como essas informações são representadas e manipuladas, por métodos específicos de combinação de dados. No entanto, não há um esforço para formalização desses aspectos de modo a se classificar ou categorizar as soluções, por exemplo, com relação a questões sensíveis como escalabilidade, custo computacional, disponibilidade de informações fundamentais.

Além disso, embora existam métodos capazes de alcançar uma precisão considerável, muitas vezes, essas técnicas falham ao lidar com sistemas complexos ou não conseguem ser aplicadas de forma eficiente em grandes bases de código. O custo computacional elevado, a dificuldade de integração com sistemas reais e a falta de aceitação das ferramentas automatizadas pelos desenvolvedores de software são obstáculos adicionais

Nesse cenário, torna-se fundamental explorar novas técnicas que, além de aprimorar a acurácia na localização de defeitos, sejam escaláveis e economicamente viáveis para aplicação prática. Este trabalho busca contribuir para essa área ao propor e avaliar metodologias que integram redes neurais convolucionais e modelos de linguagem de grande porte, com o objetivo de oferecer uma solução flexível para a localização automatizada de defeitos.

1.3 Objetivos

O objetivo geral deste trabalho é desenvolver e avaliar técnicas que melhorem a acurácia na identificação automática de elementos defeituosos no código de software, pela aplicação de Redes Neurais Convolucionais (CNN) e Modelos de Linguagem de Grande Porte (LLM). Espera-se que a utilização dessas duas abordagens e a formalização dos aspectos fundamentais para a localização de defeitos ofereça parâmetros que orientem o desenvolvimento de novas pesquisas na área.

Para alcançar o objetivo geral, foram definidos os seguintes objetivos específicos:

- Elencar e formalizar aspectos-chaves para a construção de soluções de localização de defeitos de forma automática;
- Adaptar e implementar arquiteturas de redes neurais profundas para aprimorar a identificação de defeitos em software;
- Propor e avaliar estratégias de geração de rankings de elementos suspeitos com base em LLMs;
- Integrar múltiplas fontes de informação para enriquecer o processo de localização de defeitos, explorando como essas fontes podem contribuir para aumentar a precisão das técnicas;
- Validar as propostas em projetos de software reais para avaliar a eficácia e aplicabilidade das técnicas desenvolvidas, verificando sua aplicabilidade em contextos práticos;
- Avaliar as propostas considerando conjuntos de dados para diferentes linguagens de programação.

1.4 Organização do Texto

Além desta introdução, o texto que descreve esta pesquisa está organizado da seguinte forma:

O Capítulo 2 apresenta os conceitos fundamentais que sustentam este trabalho, incluindo Localização de Defeitos de Software, Redes Neurais Artificiais, Redes Neurais Convolucionais e Modelos de Linguagem de Grande Porte (LLMs), que fornecem a base teórica para as abordagens propostas.

O Capítulo 3 examina os principais trabalhos relacionados à localização de defeitos, com foco nas metodologias mais próximas deste estudo, especialmente aquelas que utilizam aprendizado de máquina e LLMs.

O Capítulo 4 propõe uma sistematização e formalização dos aspectos essenciais para a localização automática de defeitos, estruturando a base teórica para o desenvolvimento das propostas subsequentes.

O Capítulo 5 apresenta uma abordagem baseada em Redes Neurais Convolucionais (CNNs) para a localização de defeitos, utilizando dados de cobertura de código e detalhando a modelagem do problema como uma tarefa de aprendizado supervisionado.

O Capítulo 6 avalia experimentalmente as abordagens baseadas em CNNs, comparando os resultados obtidos com as técnicas tradicionais, utilizando conjuntos de dados amplamente aceitos na área de depuração de software.

O Capítulo 7 introduz uma abordagem baseada em Modelos de Linguagem de Larga Escala (LLMs) para a localização de defeitos, explorando estratégias de combinação de múltiplas fontes de informação para gerar rankings de suspeita mais precisos.

O Capítulo 8 apresenta um estudo empírico das abordagens baseadas em LLMs, avaliando seu desempenho em um extenso conjunto de dados contendo mais de mil programas escritos em Python.

O Capítulo 9 explora experimentações em cenários livres de vazamento de dados, avaliando a eficácia das abordagens de LLMs em contextos em que os dados de treinamento não influenciam a validação.

Por fim, o Capítulo 10 traz as conclusões, resumindo os principais achados da pesquisa e propondo direções futuras para a continuidade do trabalho, incluindo melhorias nas técnicas e novas áreas de exploração.

CAPÍTULO 2

Fundamentação Teórica

Este capítulo tem como objetivo fornecer uma base teórica sólida para os principais conceitos abordados neste trabalho, organizando o conhecimento em tópicos principais: Localização de Defeitos de Software, Redes Neurais Artificiais e Modelos de Linguagem. A partir de uma análise abrangente de cada um desses temas, busca-se explorar as abordagens mais relevantes e as tecnologias subjacentes que suportam o desenvolvimento da presente pesquisa.

Na Seção 2.1, aborda-se a Localização de Defeitos de Software, pela discussão das técnicas baseadas em espectro de cobertura, método baseado em dados de execução dos testes, e abordagens de aprendizado de máquina, que se destaca nos últimos anos pela capacidade de detectar padrões mais complexos. Além disso, serão apresentadas métricas usadas para avaliar a eficiência dessas técnicas.

A Seção 2.2 explora as Redes Neurais Artificiais (RNAs), enfatizando a importância desses modelos para tarefas de classificação. Nesta Seção, será dado destaque às Redes Neurais Convolucionais (CNNs), um tipo específico de RNA com forte aplicabilidade em tarefas de processamento de imagens, além de discutir as métricas de avaliação mais usadas para mensurar o desempenho dessas redes em tarefas de classificação.

Por fim, a Seção 2.3 trata dos Modelos de Linguagem, que desempenham um papel cada vez mais central em diversas aplicações de Processamento de Linguagem Natural (PLN). Serão abordados conceitos fundamentais de Modelos de Linguagem de Grande Porte (LLMs), discutindo seu impacto no campo da inteligência artificial, bem como os desafios e direções futuras que guiam a evolução dessa tecnologia.

2.1 Localização de Defeitos de Software

Com o aumento da complexidade dos softwares e os altos custos associados à depuração de código, houve um esforço significativo para automatizar, total ou parcialmente, o processo de localização de defeitos. Wong et al. (2016) realizaram um extenso levantamento de trabalhos publicados entre 1977 e 2014, identificando mais de 331 artigos que abordam o problema a partir de diferentes estratégias.

Ainda de acordo com Wong et al. (2016), as técnicas tradicionais de localização de defeitos incluem: 1) monitoramento de registros de variáveis e estados do programa; 2) inserção de afirmações no código que verificam condições durante a execução; 3) uso de pontos de parada para permitir que o desenvolvedor inspecione o estado do programa durante a execução; e 4) criação de perfis de execução que indicam comportamentos anômalos compatíveis com defeitos.

Os autores também identificaram várias técnicas avançadas desenvolvidas para melhorar a eficácia da localização de defeitos. Essas incluem: 1) técnicas baseadas em fatias de código; 2) técnicas baseadas em espectro de cobertura; 3) abordagens estatísticas; 4) análise do estado do programa; 5) métodos de aprendizado de máquina; 6) mineração de dados; 7) modelagem; e 8) outras técnicas diversas (WONG et al., 2016).

Embora o uso de aprendizado de máquina para localização de defeitos estivesse ainda em fase inicial no período do levantamento citado, técnicas como as baseadas em espectro de cobertura já apresentavam resultados promissores (ABREU et al., 2009). Essas técnicas serão detalhadas a seguir, enquanto o uso de aprendizado de máquina será explorado em um capítulo dedicado aos trabalhos relacionados a esta pesquisa.

2.1.1 Localização de Defeitos Baseada em Espectro de Cobertura

O termo “espectro” refere-se à ideia de que é possível analisar as propriedades de um programa observando a execução de componentes específicos (HARROLD et al., 2000). As técnicas de localização de defeitos baseadas em espectro de cobertura utilizam as informações geradas durante a execução de uma suíte de testes para identificar os componentes de software potencialmente responsáveis por um defeito. A suposição é que os componentes do software que foram exercitados pelos casos de teste que falharam estão envolvidos na causa do defeito, e, portanto, devem ser priorizados para inspeção.

Inicialmente, alguns estudos sugeriram o uso exclusivo dos dados de cobertura dos casos de teste que falharam. No entanto, pesquisas subsequentes mostraram que a combinação de dados de casos de teste bem-sucedidos e malsucedidos produz melhores resultados na identificação de defeitos (WONG et al., 2016). A Figura 2.1 ilustra uma matriz de cobertura, onde cada célula indica se um determinado “*statement*” foi exercitado durante a execução de um caso de teste.

Na Figura 2.1, cada posição a_{ij} indica se o statement j foi “tocado” pelo caso de teste i. O vetor falhas armazena os resultados de cada execução, enquanto o valor $S(ej)$ representa a propensão de um *statement* ser o local do defeito, calculada por uma técnica específica. Em geral, as técnicas baseadas em espectro de cobertura aplicam uma equação (heurística) formulada por variáveis cujos valores são obtidos a partir da matriz de cobertura, gerando um *ranking* de *statements* em ordem de suspeita.

Figura 2.1: Representação de dados de cobertura e suspeita por *statement*.

$$
\begin{array}{c}
\qquad\qquad n \text{ statements} \qquad\qquad\qquad \text{falhas} \\
m \begin{array}{c}\text{execuções} \\ \text{casos de teste}\end{array}
\begin{bmatrix}
a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\
a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m,1} & a_{m,2} & \cdots & a_{m,n}
\end{bmatrix}
\quad
\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix} \\
S(e_1) \quad S(e_2) \quad \cdots \quad S(e_n)
\end{array}
$$

Fonte: Adaptada de Abreu et al. (2009).

Uma das primeiras heurísticas empregadas na localização de defeitos em software é denominada Ochiai, originalmente desenvolvida para inferência de similaridade genética e adaptada para a localização de defeitos (ABREU; ZOETEWEIJ; GEMUND, 2006). A heurística Ochiai é calculada da seguinte forma:

$$S(e) = \frac{C_{ef}}{\sqrt{(C_{ef} + C_{nf}) \times (C_{ef} + C_{ep})}} \tag{2-1}$$

onde C_{ef} é o número de casos de teste malsucedidos que tocaram o *statement e*, C_{ep} é o número de casos de teste bem-sucedidos que tocaram o *statement e*, e C_{nf} é o número de casos de teste malsucedidos que não tocaram o *statement e*. A Figura 2.2 ilustra um exemplo de aplicação da heurística Ochiai.

Figura 2.2: Exemplo de programa com dados de cobertura e suspeita calculada pela heurística *Ochiai*.

ID	Código	CT = 1	CT = 2	CT = 3	CT= 4	Ochiai
1	if (n % 2 == 0):	●	●	●	●	0,7
2	x = n + 1 # bug		●		●	1
3	else:	●		●		0
4	x = n - 1	●		●		0
	Resultado da execução	✓	✗	✓	✗	

Fonte: Elaborada pelo autor.

Na Figura 2.2, os círculos preenchidos indicam que o *statement* foi tocado pelo caso de teste, a seta (✓) indica que a execução foi bem-sucedida e o símbolo (**x**) indica uma falha. A aplicação da heurística Ochiai produz resultados que permitem ranquear os *statements* por sua propensão a conter defeitos, priorizando aqueles com maior probabilidade de erro.

A Tarantula, introduzida por Jones, Harrold e Stasko (2002), é outra técnica baseada em dados do espectro de cobertura para assinalar a propensão de um elemento de código ser o responsável pelo defeito. Essa heurística baseia-se na ideia de que os

elementos executados pelos casos de testes bem-sucedidos são menos suspeitos do que aqueles exercitados pelos casos de testes que malsucedidos. Para isso, a Tarantula é dada pela equação:

$$S(e) = \frac{\%C_{ep}}{\%C_{ep} + \%C_{ef}} \tag{2-2}$$

onde, $\%C_{ep}$ é o percentual de casos de testes que exercitam o elemento e e foram bem-sucedidos e $\%C_{ef}$ é o percentual de elementos que exercitam o elemento e e foram malsucedidos.

2.1.2 Localização de defeitos baseada em Aprendizado de Máquina

A localização de defeitos baseada em aprendizado de máquina (AM) é uma abordagem que ganhou destaque nos últimos anos devido à sua capacidade de lidar com a crescente complexidade dos sistemas de software. Diferente das técnicas tradicionais, que dependem de heurísticas ou da inspeção manual, o aprendizado de máquina utiliza modelos estatísticos e algoritmos para identificar padrões nos dados que indicam a presença de defeitos (TANTITHAMTHAVORN et al., 2016; CETINER; SAHINGOZ, 2020; MUMTAZ et al., 2021).

O uso de aprendizado de máquina na localização de defeitos contribui para aumento na acurácia desse processo em sistemas de software cada vez mais complexos. Por acurácia, entenda-se a capacidade das técnicas de localização de defeitos em classificar corretamente elementos de código como sendo ou não os locais de defeito. Paralelamente, a eficiência das técnicas, diz respeito à condição de se realizar a localização de defeitos com um custo computacional razoável, que permita a efetiva utilização da técnica em ambiente real de desenvolvimento e manutenção de software.

À medida que os sistemas se tornam maiores e mais interconectados, a tarefa de localizar manualmente os defeitos se torna impraticável e propensa a erros. O aprendizado de máquina oferece uma solução para esses desafios, permitindo a análise automatizada de grandes volumes de dados de execução, históricos de falhas, métricas de código e outros artefatos gerados durante o desenvolvimento e manutenção de software.

Diversas técnicas de aprendizado de máquina são aplicadas à localização de defeitos, cada uma com seus próprios métodos e áreas de aplicação. As principais abordagens incluem:

Classificação Supervisionada - Nesta abordagem, modelos de aprendizado supervisionado, como árvores de decisão, máquinas de vetores de suporte (SVM), como o trabalho de Jiang e Su (2007) e redes neurais, são treinados com dados rotulados, onde os defeitos conhecidos são mapeados para os respectivos elementos de código. Após o treinamento, o modelo pode prever a presença de defeitos em novos trechos de código com base nas características aprendidas.

Redes Neurais Profundas (Deep Learning) - Com o aumento da disponibilidade de dados e poder computacional, redes neurais profundas, como as Redes Neurais Convolucionais (CNNs), como o trabalho de Zhang et al. (2019) e Redes Neurais Recorrentes (RNNs), como o trabalho de Li et al. (2019), são exploradas para a localização de defeitos. Essas redes são capazes de capturar relações complexas entre os dados de entrada e os defeitos, aprendendo representações em múltiplos níveis de abstração. A vantagem dessas técnicas é sua capacidade de processar dados não estruturados, como código-fonte e logs de execução, sem a necessidade de uma engenharia de características intensiva.

Aprendizado Semi-Supervisionado e Não Supervisionado - Em muitos casos, os dados rotulados (ou seja, onde se conhece a presença de defeitos) são escassos ou caros de se obter. Abordagens semi-supervisionadas e não supervisionadas, como *clustering* e detecção de anomalias, permitem que modelos aprendam a identificar padrões suspeitos sem a necessidade de grandes quantidades de dados rotulados. Essas técnicas são úteis para identificar trechos de código que apresentam comportamentos fora do comum, que podem indicar a presença de defeitos. Um exemplo de uso de método não supervisionado para localização de defeitos é o trabalho de Cai e Xu (2012).

Técnicas Híbridas - Em alguns casos, combinações de diferentes técnicas de aprendizado de máquina são usadas para melhorar a precisão da localização de defeitos (YOO, 2012). Por exemplo, uma abordagem híbrida pode combinar técnicas de *clustering* para agrupar elementos de código similares e, em seguida, aplicar um classificador supervisionado para identificar os grupos mais propensos a conter defeitos, como proposto por Cai e Xu (2012).

Estudos empíricos demonstram que as técnicas de aprendizado de máquina podem melhorar significativamente a acurácia e a eficiência da localização de defeitos. Por exemplo, as redes neurais profundas mostram grande potencial em tarefas complexas de detecção de defeitos, especialmente em sistemas grandes e com código altamente modularizado. Além disso, modelos supervisionados, como SVMs e árvores de decisão, são eficazes na identificação de padrões em métricas de código que indicam a presença de defeitos.

Um estudo relevante é o de Tantithamthavorn et al. (2016), que aplicou técnicas de aprendizado de máquina para a localização de defeitos em um grande repositório de código aberto. Os resultados mostraram que os modelos baseados em aprendizado de máquina superaram técnicas tradicionais, especialmente em projetos com grande volume de código e histórico de falhas.

Embora o aprendizado de máquina ofereça muitas vantagens na localização de defeitos, existem desafios e limitações a serem considerados:

Qualidade e Disponibilidade dos Dados - A acurácia dos modelos de aprendizado de máquina depende da qualidade e da quantidade de dados disponíveis. Dados

incompletos ou ruidosos podem levar a previsões imprecisas.

Complexidade Computacional - Modelos avançados, como redes neurais profundas, podem exigir recursos computacionais significativos para treinamento e inferência, o que pode ser uma barreira em ambientes com restrições de recursos.

Generalização - Um desafio importante é garantir que os modelos de aprendizado de máquina generalizem bem para diferentes projetos e contextos de software. Um modelo treinado em um projeto específico pode não ser diretamente aplicável a outro, devido às diferenças nas práticas de codificação, arquiteturas de software e ambientes de execução.

A pesquisa em localização de defeitos baseada em aprendizado de máquina continua a evoluir. Algumas das direções promissoras incluem:

- Integração de LLMs (Large Language Models): A aplicação de modelos de linguagem de grande porte, como GPT, na análise de código e identificação de defeitos é uma área emergente que promete trazer avanços significativos. Esses modelos podem entender o contexto do código e sugerir correções de forma mais intuitiva.
- Aprendizado por Transferência: O uso de aprendizado por transferência para aplicar modelos treinados em um contexto para outro, reduzindo a necessidade de grandes quantidades de dados rotulados para cada novo projeto.
- *Explicabilidade*: Desenvolver métodos que tornem os modelos de aprendizado de máquina mais interpretáveis, permitindo que desenvolvedores compreendam por que um modelo identificou determinado trecho de código como defeituoso.

2.1.3 Métricas de Avaliação para a Localização de Defeitos

Para mensurar o desempenho das técnicas que se propõem a resolver o problema da localização de defeitos de *software* de forma automática, algumas métricas foram incorporadas de outras áreas e métricas específicas do contexto de localização de defeitos foram desenvolvidas.

Uma dessas métricas adotadas para medir a eficácia de um ranking produzido para a localização de defeitos é a acurácia, referida em muitos trabalhos como *ACC@N* (LE et al., 2016a). Essa métrica trabalha com valores absolutos e mensura a quantidade de *statements* com defeito posicionados até a posição N do ranking de suspeitas. Por exemplo, o *ACC@N*, para $N = 1$, é a quantidade de *statements* que possuem defeito(s) e a técnica de localização colocou tal *statement* na primeira posição do ranking. É comum se utilizar pelo menos 3 valores para N durante uma investigação empírica utilizando essa métrica. De maneira mais formal, a métrica *ACC@N* pode ser dada por:

$$\text{ACC@N} = \sum_{d \in D} 1 \cdot (\text{rank}(d) \leq N) \tag{2-3}$$

onde, D é o conjunto defeitos a serem localizados, $rank(d)$ é a posição do defeito d na lista de elementos ordenados pela suspeita de conter o defeito, N é o número de elementos mais suspeitos a serem considerados e $(\text{rank}(d) \leq N)$ indica se o defeito deve ser contabilizado até N ou não.

A métrica Exam (WONG et al., 2012) ou *Expense* é uma medida de avaliação que possui um valor relativo ao tamanho do programa sob investigação. Dado um *ranking* de suspeitas por *statements*, o valor de Exam é definido pela porcentagem de *statements* inspecionados até que se encontre o defeito, conforme:

$$\text{Exam} = \frac{\text{rank}(d)}{|E|} \tag{2-4}$$

onde, $rank(d)$ é a posição do primeiro defeito encontrado na lista ordenada de elementos de código, classificada pelo modelo com base em suspeitas e $|E|$ é o número total de elementos de código considerados.

Em cenários de experimentação de técnicas de localização de defeitos, é comum que se tenha um conjunto de programas para nos quais se pretende coletar a métrica Exam, logo, pode ser adotada a variação de Exam médio, da seguinte forma:

$$\text{Exam médio} = \frac{1}{|P|}\left(\frac{\text{rank}(d)}{|E|}\right) \tag{2-5}$$

onde, P é a quantidade a quantidade de programas com defeitos.

Apesar da medida relativa ser justa, por considerar a magnitude do programa para avaliar uma técnica que se propõe a localizar um defeito nele, essa métrica pode não ser tão prática, uma vez que os desenvolvedores tendem a sentir fadiga ao inspecionar um número alto de *statements*.

A métrica wef@n (LE et al., 2016b) ou *Wasted Effort* de certa forma é uma combinação das duas anteriores, pois representa o esforço desperdiçado por um desenvolvedor, tomando as n primeiras posições do ranking de suspeitas, até encontrar o primeiro *statement* defeituoso. Ou seja, computa quantos *statements* não defeituosos precisam ser inspecionados até que um defeito seja encontrado, mas inspecionando até um limite n, conforme:

$$\text{WEF@N} = \frac{\min(\text{rank}(d), N) - 1}{N} \tag{2-6}$$

onde, $rank(d)$ é a posição do primeiro defeito encontrado na lista ordenada de elementos de código, classificada pelo modelo com base em suspeitas e $|N|$ é o número de elementos de código mais suspeitos considerados.

2.1.4 Contextos de Avaliação de Métodos de Localização de Defeitos

A complexidade de investigação de um defeito em software pode variar consideravelmente a depender do contexto do ambiente em que o código sob investigação se encontra. Trabalhos anteriores de Silva-Junior et al. (2023) e Hirsch e Hofer (2022), a despeito de terminologias diversas, levantam que, do pontos de vista dos conjuntos de dados largamente utilizados para avaliação de métodos de localização de defeitos, o contexto do software, consequentemente dos dados utilizados para a localização podem ser pelo menos três, softwares de ambiente de produção, software real de ambiente controlado e software sintético, conforme apresentado abaixo:

- **Software de ambiente de produção**: diz respeito ao software que efetivamente é utilizado recorrentemente para solução um problema, em linhas gerais, os conjuntos de dados que representam essa categoria são oriundos de projetos de software de código aberto;
- **Software real de ambiente controlado**: são programas de computador reais, no sentido de terem sido desenvolvidos por programadores humanos, que resolvem um problema concretamente especificado, porém, que não necessariamente entrarão em produção. Por produção, entenda-se uma operação real com usuários comuns que usam o software para uma aplicação de negócio e não apenas com fins acadêmicos. No geral, esse contexto se caracteriza por programas provenientes de cursos de programação e competições de programação;
- **Software sintético**: Software produzido de forma artificial com o intuito de estressar atributos específicos que possam ser usados para construção de cenários hipotéticos e, consequentemente, avaliar capacidades peculiares dos métodos de localização de defeitos.

Em geral, as abordagens de localização de defeitos em si não impõe restrições ou não são desenvolvidas especificamente para software de algum contexto. Todavia, o ambiente de no qual o software está inserido por representar algum nível de limitação em relação à disponibilidade de informações úteis para a localização do defeito.

2.2 Redes Neurais Artificiais

As redes neurais artificiais são uma família de estratégias de aprendizagem de máquina inspiradas nas redes neurais reais. Assim como as redes neurais reais, as redes neurais artificiais trabalham com a ideia de interconexão entre neurônios para controle de sistemas e/ou realização de tarefas. Todavia, de forma prática, as redes neurais artificiais podem ser utilizadas para a resolução de problemas de inteligência artificial sem

necessariamente replicar, em detalhes, um sistema biológico real (SUMATHI; PANEERSELVAM, 2010).

Assim como outros métodos de aprendizagem de máquina, comumente referido pelo termo em língua inglesa *Machine Learning*, as redes neurais artificiais tipicamente constituem um modelo preditivo a partir da auto adaptação por experiências. Ou seja, a partir da sucessiva apresentação de exemplos, a rede neural é capaz de inferir as relações entre as diferentes variáveis que caracterizam o problema (SILVA; SPATTI; FLAUZINO, 2010). De acordo com Russell e Norvig (2010), uma tarefa de aprendizagem supervisionada pode ser definida como:

Considerando um conjunto, tido como conjunto de treinamento, de N pares contendo entradas e saídas (x_1, y_1), (x_2, y_2), ... (x_n, y_n), onde y_i tenha sido gerado por uma função desconhecida $y = f(x)$, pretende-se descobrir uma função h que se aproxime da função f verdadeira. A função h se aproxima da função f à medida que dado o mesmo domínio, a diferença numérica entre cada valor imagem de h e o correspondente valor imagem de f tende a 0.

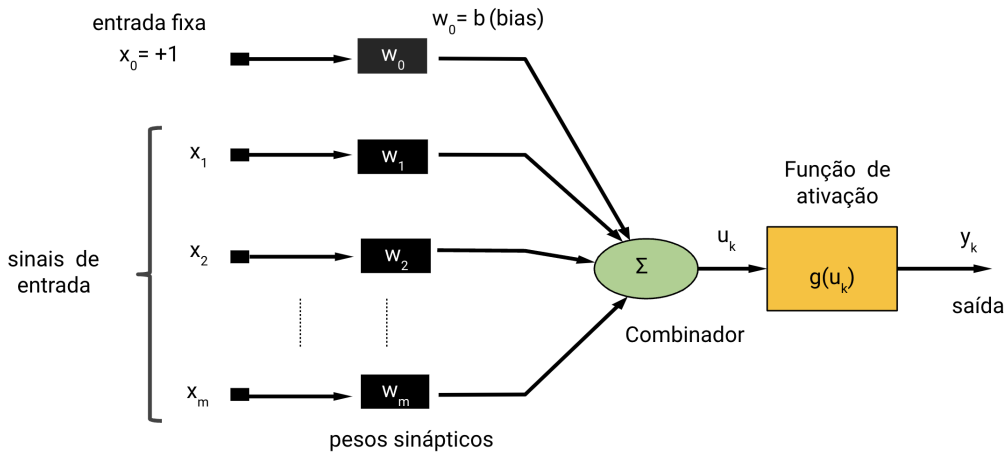
Considerando h como uma hipótese, a aprendizagem é o processo de exploração do espaço de hipóteses que melhor expliquem o comportamento dos dados de treinamento, mas que também seja adequada para explicar novos exemplos que não foram utilizados durante o treinamento. Essa é uma característica conhecida como a capacidade de generalização de um modelo de aprendizagem. De acordo com Silva, Spatti e Flauzino (2010), a capacidade de generalização é uma das principais características das redes neurais artificiais.

Quando a técnica de aprendizagem de máquina supervisionado é utilizada para resolver um problema cuja cardinalidade do domínio de saída é finito, tal como o conjunto $\{$"*gato*","*cachorro*"$\}$, onde cada elemento do conjunto representa uma classe, dizemos que se trata de uma tarefa de classificação. Por outro lado, se as saídas do problema forem valores numéricos do espaço contínuo, trata-se de uma tarefa de regressão. Para o caso das tarefas de classificação, algumas técnicas de aprendizagem de máquina consistem na construção de uma distribuição de probabilidade condicional $P(Y|x)$. Dessa forma, dada uma amostra x, a resposta do modelo treinado será a probabilidade de aquela amostra pertencer à classe Y.

Para a construção de um modelo de predição supervisionado, para tarefas de classificação, uma rede neural artificial usa cada neurônio como uma unidade de processamento dos dados do problema. A Figura 2.3 apresenta um esquema do funcionamento dos principais componentes de um neurônio artificial.

Durante o processo de treinamento, fase em que a rede precisa achar os parâmetros que gerem a melhor aproximação para a verdadeira função que explica as saídas, os neurônios recebem os valores de cada variável x_i que compõe uma amostra k. Os valores

Figura 2.3: Esquema dos componentes de um neurônio artificial



Fonte: Elaborada pelo autor.

de entrada, também conhecidos como sinais sinápticos, são ponderados pelos pesos sinápticos e em seguida combinados por uma função combinadora, tipicamente um somatório. Então, obtém-se um valor chamado de potencial de ativação u_k, que é um valor numérico representando o quão impactantes são os valores das características da amostra k para a ativação do neurônio. O potencial de ativação é submetido a uma função g, conhecida como função de ativação. Finalmente, a saída da função de ativação é uma previsão do valor $y_k = g(u_k)$ para a amostra k. Como se tem acesso ao valor esperado de y_k é possível comparar com a saída obtida e calcular o erro da estimação.

O processo de treinamento de um neurônio ou de uma rede consiste, portanto, na apresentação iterativa das amostras de treinamento, o cálculo do erro e o ajuste dos pesos sinápticos na direção que minimize o erro global do modelo na próxima iteração. O cálculo de erros é feito por uma função denominada função de perda, que indica o quão distante o modelo está do ideal. Os ajustes nos pesos podem ser realizados a cada nova amostra observada ou a cada bloco de amostras observados e, quando todas as amostras forem apresentadas à rede, pode ser realizada uma nova iteração, conhecida como época.

Como pode ser visto na Figura 2.3, cada peso w está associado a uma entrada x, porém, existe um peso w_0, ou bias, que é associado a uma entrada fixa de 1 (ou -1). Esse peso é o responsável por regular o impacto dos sinais de entrada na ativação do neurônio, uma vez que se ele não existisse, apenas os valores de x_i determinariam o valor do potencial de ativação u. Matematicamente, a consequência da introdução desse termo em uma combinação linear, é a capacidade de se afastar a reta produzida pela combinação da origem do espaço formado pelas dimensões x.

A função de ativação é um dos componentes preponderantes para a construção de um bom modelo de uma rede neural artificial. Como este é o componente responsável por regular a saída do neurônio, ele tem papel decisivo na forma numérica da predição

que será realizada, caso seja um neurônio de saída, ou na propagação do sinal de entrada, caso seja um neurônio de uma camada intermediária. Por exemplo, para uma tarefa de classificação binária, para função de ativação pode ser utilizada uma função degrau:

$$g(u) = \begin{cases} 1, & \text{se u } \geq 0 \\ -1, & \text{se u } < 0 \end{cases} \tag{2-7}$$

onde, a saída será 1 se o potencial de ativação for maior ou igual a 0 e -1, caso contrário. Apesar de funcionar para problemas triviais, esse tipo de função não pode ser usado por estratégias de otimização dos pesos que se baseiem em gradiente descendente, pois uma função degrau não é diferenciável em todo seu domínio (SILVA; SPATTI; FLAUZINO, 2010). Por esse motivo, uma função de ativação frequentemente usada para saída binárias é a função logística, dada por:

$$g(u) = \frac{1}{1 + e^{-\beta u}} \tag{2-8}$$

onde, β é uma constante real definindo o nível de inclinação da curva diante do ponto de inflexão da função. Uma das vantagens do uso dessa função é a possibilidade de simular uma probabilidade, pois ela mapeia qualquer valor de entrada para o intervalo [0,1].

Existem diferentes tipo de redes neurais, aplicadas a problemas de diferentes naturezas, por exemplo, arquiteturas recorrentes para tratamento de séries temporais, redes auto-organizadas para lidar com problemas de agrupamentos ou redes de camada única, como *Hopfield* para otimização de sistemas. Uma das arquiteturas mais difundidas é conhecida como *Multilayer Perceptron* (MLP), ou *Perceptron* Multicamadas (PMC). Essa arquitetura é largamente aplicada tanto para tarefas de regressão quando tarefas de classificação e possui desempenho elevado para problemas cujas características sejam predominantemente numéricas.

Um PMC é uma RNA composta por pelo menos uma camada de neurônios escondidos e uma camada com neurônios de saída; utiliza a regra delta generalizada no processo de treinamento; tem uma fase de propagação, em que se projetam os dados da camada de entrada à camada de saída; e, há uma fase de retro propagação, na qual se atualizam os pesos com base na regra delta (RUMELHART et al., 1995).

O treinamento de um modelo baseado em apresentação de exemplos pode ter como consequência o comportamento de "superajuste" nos dados que foram apresentados e, ao invés de ter um modelo generalizável, tem-se um modelo que decorou os dados de treinamento. Esse comportamento é conhecido como *overfitting*. Uma das causas para esse comportamento é uma quantidade alta de neurônios em camadas iniciais ou intermediárias da arquitetura da rede (SILVA; SPATTI; FLAUZINO, 2010). Uma das formas de se verificar uma condição de *overfitting* é o monitoramento dos valores da função de perda no conjunto de validação, durante o treinamento.

Um treinamento com *overfitting* será evidenciado quando a perda deixar de cair e passar a subir, considerando o conjunto de validação, enquanto a perda no conjunto de treinamento continuar a descer, com o passar das épocas. Como o nome sugere, conjunto de validação é uma fatia dos dados de treinamento que é usada para validar alguns aspectos de aprendizagem do modelo. Dentre as opções de tratamento de *overfitting* estão a diminuição da complexidade da arquitetura, a aplicação de regularização, que significa uma penalização no cálculo da função de perda, e a utilização de camadas do tipo *dropout*, que fazem a desativação aleatória de um percentual de neurônios.

2.2.1 Redes Neurais Convolucionais

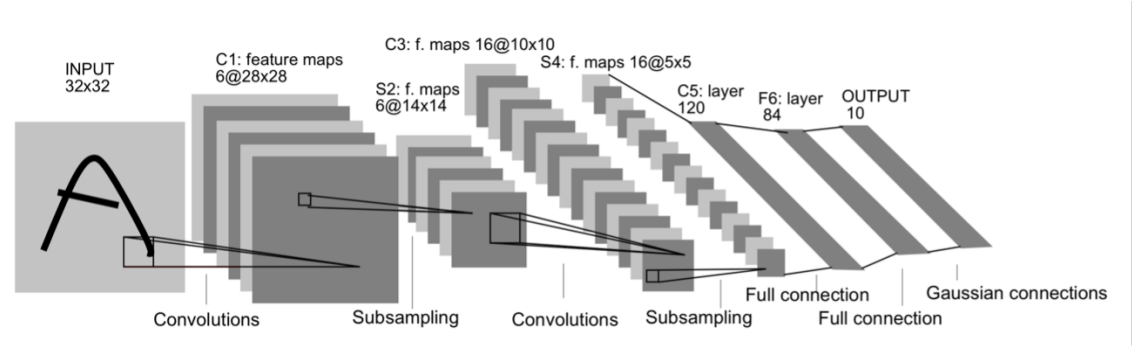
Apesar de terem gerado um impacto inicial maior no campo da visão computacional, principalmente na classificação de imagens, as Redes Neurais Convolucionais, do inglês *Convolutional Neural Networks* (CNNs) são empregadas com sucesso nas mais variadas áreas, tais como: detecção e rastreamento de objetivos, detecção e reconhecimento de textos, reconhecimento de gestos e ações em vídeos ou imagens, anotação de cenas, processamento de fala, e outras aplicações relacionadas a processamento de linguagem natural (GU et al., 2018). Essa variedade de aplicações ilustra a flexibilidade e potencial que esse tipo de arquitetura de rede neural possui.

As CNNs seguem fundamentalmente os mesmo princípios de uma rede neural convencional como um PMC, por exemplo. Uma rede convolucional também aprende por meio da exposição de várias amostras do problema, e faz um ajuste dos pesos sinápticos de acordo com uma função de perda que mede a distância do modelo para a real função que explica os dados (O'SHEA; NASH, 2015). Todavia, as redes neurais convolucionais combinam três princípios arquiteturais que as diferenciam da redes neurais tradicionais, sendo eles: os campos receptivos locais, pesos compartilhados e sub-amostragem espacial ou temporal (LECUN et al., 1998).

Com a ideia dos campos receptivos locais, a rede é capaz de extrair características visuais elementares, como pontos, setas ou curvas. Então, essas características podem ser combinadas em camadas subsequentes para gerar informação de um grau mais elevado. Por exemplo, o reconhecimento de uma bola de futebol, pode se dar pela combinação de várias informações elementares como semicírculos, bordas, restas etc. Pela aplicação de filtros convolucionais, são gerados diversos extratores de características, com seus próprios pesos. Os filtros convolucionais também são matrizes de valores que ao serem deslocados sobre a matriz de entrada, que é a imagem original, destacam informações da imagem (MATHER, 2004). Esses mapas de características possuem, portanto, pesos que modelam estruturas visuais comuns a figuras mais complexas que pretendem ser identificadas.

Uma vez detectadas as características mais relevantes, a posição exata não é tão relevante quanto saber a posição relativa a outras características úteis. Por exemplo, um traço seguido de um triângulo pode ser uma seta de direção. Tem-se então, a sub-amostragem espacial ou temporal, pela qual a rede aplica a redução do espaço de características pela concentração nos valores mais expressivos. A Figura 2.4 mostra um exemplo de uma CNN típica aplicada ao contexto de classificação de imagens, nesse caso, reconhecimento de letras escritas à mão.

Figura 2.4: Exemplo de arquitetura típica de CNN aplicada ao contexto de classificação de imagens



Fonte: Lecun et al. (1998).

Inicialmente, a matriz de características, com todos os *pixels* que compõem uma imagem de tamanho 32x32, é submetida à uma camada com filtros convolucionais que geram 6 mapas de características de tamanho 28x28. Após a aplicação de uma sub-amostragem, que pode ser valor máximo, valor médio, e outros, o tamanho dos mapas diminui para 14x14, ou seja, a metade. Aplica-se mais uma camada de convoluções, seguida de sub-amostragem, até que se tenham 16 filtros de tamanho 5x5, o que significa um total de 400 características. Nesse ponto, todas as características são submetidas a um classificador convencional, ou seja, um PMC, contendo 120 neurônios na camada de entrada, 84 na intermediária e 10 na camada de saída, representado as 10 classes possíveis.

O exemplo anterior detalha a utilização de filtros convolucionais do tipo 2D, ou seja, o deslocamento dos filtros ocorre em duas dimensões. Entretanto, também existem redes convolucionais 1D e 3D. Para compreensão intuitiva, enquanto uma convolução 2D ocorre sobre uma área retangular que tende a um quadrado, uma convolucional 1D estaria sobre um retângulo cuja altura tende a ser pequena e largura grande, de modo que o deslocamento seja apenas no sentido horizontal e, por fim, uma convolucional 3D atua em uma figura similar a um cubo, com deslocamentos em três dimensões.

As CNNs 2D são aplicadas para uma grande variedade de problemas nos quais é possível uma modelagem para um problema de sinais em duas dimensões. Porém, seja por escassez de dados para treinamento, seja por uma necessidade específica de domínio, muitas vezes não é possível aplicar-se filtros 2D. Por isso, as redes convolucionais 1D

foram recentemente desenvolvidas e atingiram desempenho elevado em aplicações como classificação personalizada de dados biomédicos e diagnóstico precoce, monitoramento de integridade estrutural, detecção e identificação de anomalias em eletrônica de potência e elétrica detecção de falhas do motor (KIRANYAZ et al., 2019).

2.2.2 Métricas de Avaliação para Tarefas de Classificação

Para se aferir a qualidade de um modelo produzido por uma rede neural, são utilizadas métricas que dependem do tipo de tarefa, ou seja, existem métricas para tarefas de regressão e métricas para tarefas de classificação. Relativo às métricas para tarefas de classificação, frequentemente usa-se: matriz de confusão, acurácia, precisão, *recall* e F1.

A matriz de confusão indica quantos exemplos existem em cada grupo: falso positivo (FP), falso negativo (FN), verdadeiro positivo (VP) e verdadeiro negativo (VN). Por FP entenda-se a quantidade de amostras classificadas como sendo de uma determinada classe, mas que na verdade pertencem a outra; FN é quantidade de amostras classificadas como não sendo de uma determinada classe, mas que de fato são; VP é a quantidade de amostras classificadas como sendo de uma classe e de fato são; e, finalmente, VN é a quantidade de amostras classificadas como não sendo de uma determinada classe e verdadeiramente não são.

A Tabela 2.1 mostra uma representação de quais grupos estariam em cada posição para o caso binário. Para análise da tabela, deve-se considerar que os valores corretos estão nas linhas e as previsões estão nas colunas. Como os valores apresentados em uma matriz de confusão são absolutos, é importante considerar a proporção entre as classes para se obter conclusões.

Tabela 2.1: Matriz de confusão para o caso binário

	Positivo	Negativo
Positivo	VP	FN
Negativo	FP	VN

Fonte: Elaborada pelo autor.

A acurácia, ou simplesmente acurácia, é dada pelo número de acertos dividido pelo total de exemplos preditos. Essa medida é sensível à distribuição dos dados no conjunto, portanto, é mais susceptível a falsas conclusões quando as classes são desbalanceadas em número de elementos. A acurácia é calculada conforme fórmula abaixo:

$$\text{Acurácia} = \frac{VP+VN}{VP+VN+FP+FN}, \tag{2-9}$$

onde, VP é a quantidade de verdadeiros positivos, VN a quantidade de verdadeiros negativos, FP a quantidade de falso positivos e FN é a quantidade de falsos negativos.

A precisão é o número de exemplos classificados como pertencentes a uma classe, que realmente são daquela classe (Verdadeiros Positivos - VP), dividido pela soma entre este número e o número de exemplos classificados nesta classe, mas que pertencem a outras (falsos positivos - FP), conforme a fórmula abaixo:

$$\text{Precisão} = \frac{VP}{VP+FP}, \tag{2-10}$$

onde, VP é a quantidade de verdadeiros positivos e FP a quantidade de falso positivos.

O *Recall* ou sensibilidade é número de exemplos classificados como pertencentes a uma classe, que realmente são daquela classe, dividido pela quantidade total de exemplos que pertencem a esta classe, mesmo que sejam classificados em outra. No caso binário, Verdadeiros Positivos divididos por total de positivos, conforme:

$$\text{Recall} = \frac{VP}{VP+FN}, \tag{2-11}$$

onde, VP é a quantidade de verdadeiros positivos e FN é a quantidade de falsos negativos.

Finalmente, o F1 é uma medida que harmoniza os cálculos de precisão e sensibilidade, e é dado por duas vezes o produto da precisão pela sensibilidade, dividido pela soma dos dois, conforme Equação 2-12. Essa métrica ajuda a lidar com o problema de classes desbalanceadas no conjunto de dados.

$$F1 = 2 \cdot \frac{\text{Precisão} \cdot \text{Recall}}{\text{Precisão} + \text{Recall}} \tag{2-12}$$

2.3 Modelos de Linguagem

Modelos de linguagem representam um avanço significativo no processamento de linguagem natural (PLN) e são aplicados com sucesso para gerar código com mínima assistência humana. Com o surgimento dos Modelos de Linguagem de Grande Porte (LLMs), essa capacidade foi aprimorada, estabelecendo novos padrões na geração de código baseada em linguagem natural. Esses modelos oferecem uma abordagem mais intuitiva para a criação de código, reduzindo obstáculos na interação entre humanos e máquinas. Eles podem gerar trechos de código complexos dadas descrições em linguagem natural, auxiliando desenvolvedores em tarefas repetitivas do desenvolvimento e manutenção de software. No geral, a integração de LLMs no desenvolvimento de software representa um avanço, oferecendo capacidades aprimoradas para a geração de código e transformando a forma como o software é desenvolvido e mantido (FAN et al., 2023).

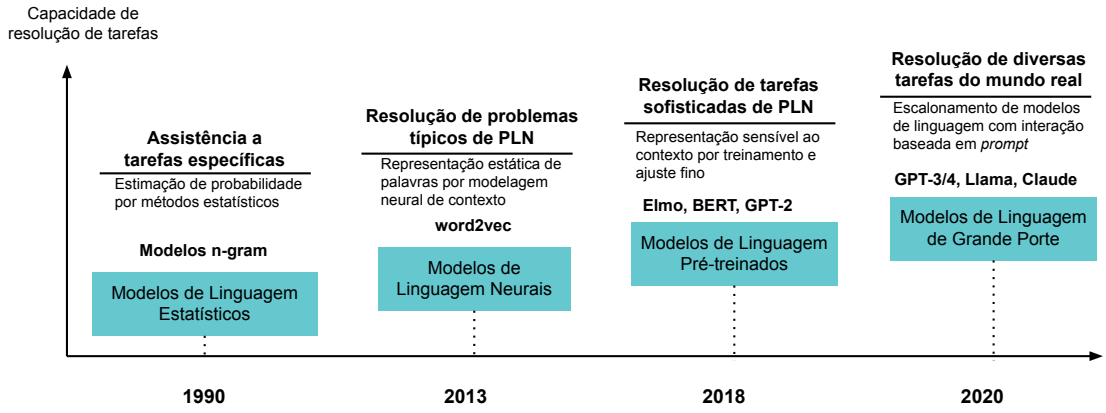
2.3.1 Processamento de Linguagem Natural

O Processamento de Linguagem Natural (PLN) é uma subárea da inteligência artificial que se concentra na interação entre computadores e a linguagem humana. O objetivo principal do PLN é permitir que máquinas compreendam, interpretem e gerem linguagem de uma forma que seja natural para os seres humanos (MANNING, 1999). Nas últimas décadas, o PLN evoluiu significativamente, impulsionado pelo desenvolvimento de algoritmos avançados e pelo aumento da capacidade computacional. Técnicas como análise sintática, semântica e pragmática, juntamente com modelos estatísticos e redes neurais, têm permitido avanços substanciais na compreensão de textos, tradução automática, resumo de documentos, e outras aplicações (JURAFSKY; MARTIN, 2024).

2.3.2 Modelos de Linguagem

Modelos de linguagem são centrais para o PLN, pois são responsáveis por prever a probabilidade de uma sequência de palavras em um dado contexto. Tradicionalmente, esses modelos eram baseados em métodos estatísticos, como os modelos de n-gramas, que calculam a probabilidade de uma palavra ou sequência de palavras com base em sua frequência em grandes corpora de texto (ROSENFELD, 2000). Embora eficazes para certas tarefas, esses métodos eram limitados pela necessidade de grandes quantidades de dados rotulados e pela dificuldade de capturar contextos mais amplos. A Figura 2.5 resume como os Modelos de Linguagem evoluíram ao longo dos anos no que diz respeito à capacidade de resolver problemas e das abordagens mais proeminentes.

Figura 2.5: Evolução das capacidades dos modelos de linguagem ao longo dos anos.



Fonte: Adaptada de Zhao et al. (2023).

Com o advento das redes neurais, os modelos de linguagem evoluíram para capturar padrões mais complexos e contextos de longo alcance em textos. Dessa forma, os modelos passaram a resolver um conjunto mais amplo de problemas.

Como a mostra a Figura 2.5, 2013 é um marco para os modelos de linguagem neurais, a partir dos quais diversos problemas típicos de NLP, como análise de sentimento, reconhecimento de entidade, preenchimento de texto, foram mais efetivamente tratados com o princípio da representação distribuída de palavras, por exemplo, através de Redes Neurais Recorrentes (HOCHREITER, 1997) ou word2vec (MIKOLOV, 2013).

Posteriormente, tem-se a introdução dos chamados modelos de linguagem pré-treinados. Nesse período, os modelos de linguagem ganham a capacidade de aprender a representação de palavras sensíveis ao contexto, ao invés de uma representação estática, por exemplo através do método ELMo introduzido por (PETERS et al., 2018). Com a introdução da arquitetura de *Transformers* por (VASWANI, 2017) e a apresentação do BERT (DEVLIN, 2018), ficou consolidada a ideia de que seria possível aprender a representação semântica de textos a partir de grande conjuntos de dados e em seguida realizar um ajuste fino (fine-tuning) para tarefas específicas.

Finalmente, a partir de 2020 dar-se o surgimento efetivo dos Modelos de Linguagem de Grande Porte, LLMs do inglês *Large Language Models*. A partir do treinamento de modelos de linguagem pré-treinados escalando-se os dados de treinamento e/ou a arquitetura de pesos, foi percebida uma habilidade crescente nos modelos para resolver problemas cada vez mais sofisticados (WEI et al., 2022).

Essa nova realidade muda o patamar da aplicação dos modelos de linguagem pré-treinados. No caso dos LLMs, ganhou-se a capacidade de resolver problemas do mundo real até então intratáveis ou tratados com pouco sucesso pela área de NLP. Diferentemente dos modelos da geração anterior, que supunham um processo de *fine-tuning* para tarefas específicas, os LLMs supõem que o conhecimento paramétrico do modelo pode ser acessado eficientemente a partir de um *prompt*, que seria a ligação entre o tarefa específica e o modelo (JURAFSKY; MARTIN, 2024).

2.3.3 Modelos de Linguagem de Grande Porte (LLMs)

Os Modelos de Linguagem de Grande Porte (LLMs), como GPT-3, Llama e T5, representam um salto significativo na capacidade das máquinas de entender e gerar linguagem natural. Esses modelos são treinados em enormes volumes de dados textuais e consistem em bilhões de parâmetros, permitindo-lhes capturar nuances linguísticas e contextuais com alta precisão (ZHAO et al., 2023).

Tipicamente, LLMs utilizam a arquitetura de *transformers*, que é particularmente eficaz em capturar dependências de longo alcance em textos, permitindo a geração de respostas coerentes e contextualmente relevantes. Essa habilidade é explorada com sucesso em uma larga gama de aplicações, como geração de textos (GOYAL; LI; DURRETT,

2022), resposta a questionários (LI et al., 2024), traduções (BROWN, 2020), resumo de textos (DUBEY et al., 2024) e análise de sentimentos (XING, 2024).

Uma aplicação notável dos LLMs é a geração automática de código a partir de descrições em linguagem natural. Ferramentas como o GitHub Copilot, baseado no modelo Codex da OpenAI, demonstram como esses modelos podem ajudar desenvolvedores a escrever código de forma mais rápida e com menos erros. Ao receber uma descrição em linguagem natural, tais modelos geram automaticamente trechos de código funcional, facilitando tarefas repetitivas e auxiliando na resolução de problemas complexos.

Além disso, LLMs estão transformando a maneira como o software é desenvolvido e mantido. Eles permitem uma interação mais intuitiva entre humanos e máquinas, em que os desenvolvedores podem simplesmente descrever a funcionalidade esperada, e o modelo gera uma solução apropriada. Isso não só aumenta a produtividade, mas também reduz a barreira de entrada para o desenvolvimento de software, permitindo que indivíduos com menos experiência em programação possam contribuir de forma significativa.

2.3.4 Conceitos Fundamentais em LLMs

Como ferramenta originária no Processamento de Linguagem Natural, os Modelos de Linguagem de Grande Porte estão intimamente ligados a conceitos amplamente difundidos na área de PLN, mas também introduzem alguma variações na abordagem de questões clássicas. Uma das distinções importantes no contexto das técnicas são as categorias. Geralmente, os LLM podem ser classificados como: modelo apenas de codificação (encoder-only), modelo de codificação e decodificação (encoder-decorder) e modelo apenas de decodificação.

Os modelos ***encoder-only***, como BERT proposto por Devlin et al. (2019) e RoBERTa introduzido por Liu et al. (2019) caracterizam-se pelo uso da parte codificadora dos *transformers* para, a partir da entrada de sequências textuais, gerar uma representação codificada profunda das relações entre as partes do texto. Esse tipo de modelo possui uma performance alta para tarefas que exigem alta compreensão de texto como classificação textual, análise de sentimentos e detecção de similaridade textual.

Paralelamente, os modelos do tipo ***encoder-decoder*** além de uma rede capaz de gerar a codificação das sequências textuais de entrada, também possui uma rede decodificadora que é capaz de gerar uma sequência textual como saída, a partir da geração iterativa de tokens ou símbolos. Para tal, o modelo gera o próximo elemento com base no contexto codificado no *enconder* e no token anterior. Modelos como T5 proposto em Raffel et al. (2023) e BART de Lewis et al. (2019) são exemplos de encoder-decoder amplamente aplicados para diversas tarefas modeladas como texto-para-texto, como tradução, resumo e geração de texto de uma forma geral.

Por fim, os modelos ***decoder-only***, como a família GPT (BROWN et al., 2020), LLaMA (TOUVRON et al., 2023) e Claude (ANTHROPIC, 2024), diferentemente dos anteriores, não apresentam um componente de codificação. Tais modelos possuem apenas a fase de decodificação, que gera uma sequência textual diretamente a partir de um contexto dado como entrada. Os modelos desse tipo também são tipicamente autoregressores, ou seja, geram a sequência de tokens ou palavras um a um, usando sempre as saídas anteriores como contexto para a geração do próximo token. Na prática, esse modelos são unidirecionais, gerando sequências da esquerda para direita e são amplamente utilizados para construção de *chatbots* e assistentes virtuais.

Como apontado anteriormente, uma vez treinado, um modelo de linguagem de grande porte tem sua utilização, majoritariamente, dependente do componente denominado ***prompting***. Na prática, esse componente consiste nas entradas e instruções que o LLM recebe como entrada para resolver uma tarefa específica. O processo de definição e ajuste do *prompt*, seja manual ou automático, é conhecido como **engenharia de *prompt***. A despeito de existirem trabalhos avaliando diferentes estratégias de definição de *prompt* (WHITE et al., 2023; SANTU; FENG, 2023), para variadas tarefas, existem algumas recomendações comumente aceitas e aplicáveis à maioria dos LLM mais comuns.

De acordo com o trabalho de Zhao et al. (2023), os principais ingredientes para construção de um prompt envolvem: a) uma descrição da tarefa a ser executada, b) os dados de entrada, tipicamente com uma descrição em linguagem natural, c) informações contextuais que possam se unir ao conhecimento paramétrico do modelo para lidar com uma tarefa específica e d) estilo do *prompt* de modo que fique explícito qual o comportamento esperado, por exemplo, indicando que o modelo deva se comportar como especialista em determinado contexto.

Além disso, é importante que a engenharia de *prompt* siga princípios como: expressar com clareza, sem dubiedade, os objetivos da tarefa, para problemas mais complexos, decompor em subtarefas e apontar um passo a passo de como a tarefa deve ser abordada, prover exemplos de resolução do problema e aplicar formatação amigável ao tipo de LLM utilizado, adotando caracteres especiais para separar instruções de contexto.

2.3.5 Arquitetura *Transformers*

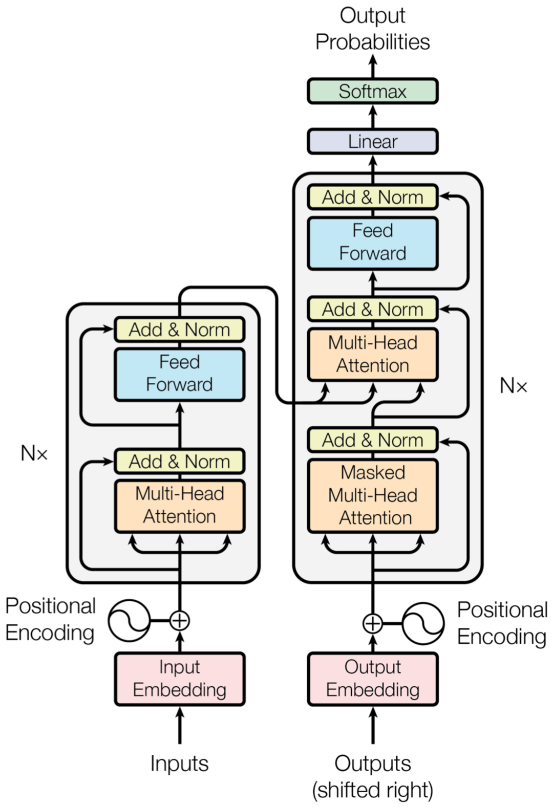
Redes neurais recorrentes (RNNs) são amplamente utilizadas para problemas de modelagem de sequência, como tradução e modelos de linguagem de forma geral. No entanto, a natureza sequencial dessas técnicas limita a paralelização, dificultando o treinamento em sequências longas devido a restrições de memória.

Mecanismos de atenção surgiram como componentes-chave para modelagem de sequência, permitindo capturar dependências entre elementos sem considerar a distância

na sequência. Até a introdução da arquitetura Transformer, a maioria das aplicações de mecanismos de atenção era pela combinação com redes recorrentes. A arquitetura Transformer, proposta por Vaswani (2017), elimina a recorrência e se baseia inteiramente em mecanismos de atenção para capturar dependências globais entre entrada e saída. Isso permite maior capacidade de paralelização, e passando a ser o estado da arte para tarefas como tradução e geração de texto.

A Figura 2.6 apresenta uma visão geral da arquitetura *Transformer*. Os principais componentes desse tipo de rede neural são: codificador (encoder), decodificador (decoder), mecanismo de atenção, rede de alimentação direta (feed-forward), camadas de normalização e codificação posicional.

Figura 2.6: Arquitetura do modelo *transformer*.



Fonte: Vaswani (2017).

O modelo *Transformer* segue uma estrutura de codificador-decodificador, no qual um vetor de entrada x é inserido em um codificador, resultando em uma representação mapeada z. Com z, o decodificador gera uma saída y, um elemento de cada vez, ou seja, primeiro gerando y_1, depois y_2, até y_m. Em cada uma dessas etapas, o modelo é autorregressivo, o que significa que as saídas previamente geradas sempre influenciam as saídas futuras.

O codificador do *Transformer* canônico é composto por $N = 6$ camadas idênticas empilhadas. Cada camada é dividida em dois subníveis: o primeiro é uma camada

de atenção com múltiplas cabeças e o segundo é uma camada de rede *feed-forward* totalmente conectada, aplicada ponto a ponto. O codificador ainda emprega conexões residuais em torno de cada subnível, seguidas por uma camda de normalização.

O decodificador também é composto por $N = 6$ camadas idênticas, mas introduz um terceiro subnível projetado para realizar atenção com múltiplas cabeças sobre as saídas do empilhamento do codificador. Assim como no codificador, são implementadas conexões residuais e normalização de camada.

Para garantir que o modelo seja autorregressivo, isto é, que apenas as posições anteriores influenciem a posição atual, é necessário modificar o primeiro subnível. A presença de um mecanismo de mascaramento e o deslocamento dos *embeddings* de saída em uma posição asseguram que a previsão da posição *i* dependa exclusivamente das saídas conhecidas que precedem *i*.

O componente principal da arquitetura *transformer* é o mecanismo de atenção. Uma função de atenção pode ser descrita como o mapeamento de uma consulta (*query*) e um conjunto de pares chave-valor para uma saída, onde a consulta, as chaves, os valores e a saída são todos vetores. A saída é calculada como uma soma ponderada dos valores, em que o peso atribuído a cada valor é determinado por uma função de compatibilidade entre a consulta e a chave correspondente.

Além dos subníveis de atenção, cada camada em nosso codificador e decodificador contém uma rede *feed-forward* totalmente conectada, aplicada de forma separada e idêntica a cada posição. Esta rede consiste em duas transformações lineares com uma ativação ReLU entre elas:

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2}$$

Embora as transformações lineares sejam as mesmas em diferentes posições, elas utilizam parâmetros diferentes em cada camada. Outra forma de descrever essa estrutura é como duas convoluções com tamanho de kernel igual a 1. A dimensionalidade da entrada e da saída é $d_{model} = 512$, e a camada intermediária possui dimensionalidade $d_{ff} = 2048$.

Como pode ser notado, o *transformer* não possui recorrência nem convolução, então, para que ele possa utilizar a ordem da sequência, existe uma informação sobre a posição relativa ou absoluta dos tokens na sequência. Para isso, adiciona-se conceito de “codificações posicionais” (Positional Encoding) aos *embeddings* de entrada tanto do codificador quanto do decodificador. As codificações posicionais têm a mesma dimensão d_{model} dos *embeddings*, de modo que possam ser somadas. Existem várias opções de codificações posicionais, tanto aprendidas quanto fixas.

2.3.6 Desafios e Futuras Direções

Apesar dos avanços, a integração de LLMs no desenvolvimento de software apresenta desafios significativos. Dois desses desafios são a interpretabilidade e a confiabilidade. Os LLMs, assim como diversos métodos baseados em Aprendizado Profundo, são frequentemente descritos como "caixas-pretas". Entender o raciocínio por trás do código gerado por um LLM pode ser complexo, o que levanta questões sobre a segurança e a robustez do software produzido. Além disso, esses modelos requerem vastos recursos computacionais para treinamento e inferência, o que pode limitar sua aplicabilidade em contextos nos quais o acesso a esses recursos é restrito (FAN et al., 2023).

Ainda de acordo com Fan et al. (2023), um dos caminhos promissores para o sucesso dos LLMs na Engenharia de Software é a adoção de métodos híbridos, de modo que os métodos e ferramentas já consolidados possam atenuar problemas como a não *explicabilidade* e a alucinação dos modelos.

No geral, os Modelos de Linguagem de Grande Porte estão redefinindo o desenvolvimento de software, oferecendo novas ferramentas e abordagens que têm o potencial de transformar a indústria. A pesquisa contínua e o desenvolvimento desses modelos serão cruciais para enfrentar os desafios e maximizar seu impacto positivo.

2.4 Considerações Finais

Neste capítulo, foram explorados conceitos essenciais para a localização de defeitos de software, abordando técnicas tradicionais e avanços recentes baseados em aprendizado de máquina e inteligência artificial. Foram discutidos métodos clássicos, como as técnicas de espectro de cobertura, bem como métricas de avaliação (ACC@N, Exam, Wef@n), que oferecem bases para avaliação de rankings de inspeções de código. Ademais, foi destacado o papel das Redes Neurais Artificiais e, especialmente, das Redes Neurais Convolucionais, cuja capacidade de identificar padrões complexos mostra-se promissora na classificação de imagens e em outras tarefas aplicáveis à análise de código.

A incorporação dos LLMs surge como uma inovação promissora na resolução de problemas clássicos da Engenharia de Software (ES), permitindo, por exemplo, a geração automática e contextualizada de código, elevando o potencial das soluções de ES. LLMs, como GPT e Llama, oferecem uma interação mais intuitiva entre desenvolvedores e máquinas, contribuindo tanto para a detecção como para a correção de defeitos em ambientes de software dinâmicos. Contudo, ainda são observados desafios em relação à explicabilidade, confiabilidade e custo computacional desses modelos, os quais requerem abordagens híbridas para mitigar suas limitações.

Esse embasamento teórico provê os fundamentos para o desenvolvimento das abordagens investigadas nesta pesquisa, e orienta futuras aplicações e melhorias na localização de defeitos, com foco na precisão, eficiência e integração de novas tecnologias.

CAPÍTULO 3

Trabalhos Relacionados

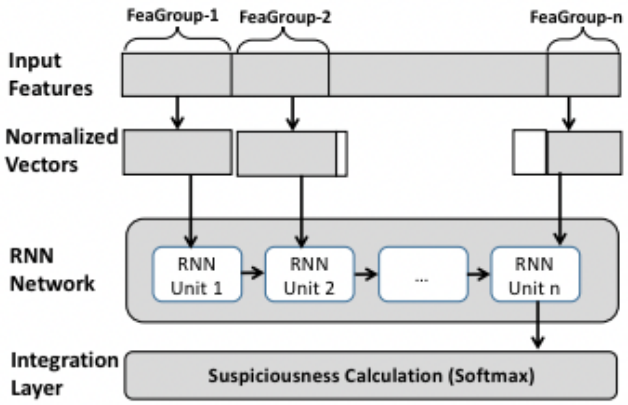
Neste capítulo, são abordagens inovadoras aplicadas à localização de defeitos de software, divididas em duas seções. Na Seção 3.1, são abordados métodos baseados em Redes Neurais, como DeepFL, que utiliza redes neurais recorrentes para integrar múltiplas fontes de informação e otimizar a localização de defeitos em código. Em seguida, exploram-se redes neurais convolucionais, com foco em arquiteturas que combinam dados de cobertura e mutação para identificar padrões de defeitos. A Seção 3.2 examina abordagens baseadas em LLM, que avançam significativamente a área, como na técnica AutoFL que empregam LLMs para localizar defeitos em nível de método e o LLMAO que via processo de fine-tune de LLM, realiza a localização de defeitos em nível de linhas de código sem a necessidade de execução de casos de teste.

3.1 Localização de Defeitos Baseada em Redes Neurais

Em 2019, Zou et al. (2019) apresentaram uma técnica baseada na ideia de *learning to rank*, muito utilizada na área de recuperação de informação, que se apresentou de forma muito competitiva frente às abordagens de localização de defeitos baseados em espectro de cobertura e espectro de mutação (PAPADAKIS; TRAON, 2012; MOON et al., 2014). Na ocasião, os autores demonstraram que a tarefa de localização de defeitos era mais efetivamente solucionada quando se consideravam diferentes famílias de técnicas em conjunto.

Contudo, no mesmo ano, Li et al. (2019) argumentaram que a técnica *learning to ranking* não seria a mais adequada para o problema pois ela teria dificuldade em se manter eficaz para cenários em que a quantidade de características fosse elevada. E então, propuseram a técnica denominada *DeepFL*. A técnica emprega a ideia de combinar diferentes fontes de informação: suspeita baseada em mutante, suspeita baseada em cobertura, tendência a falha com base na complexidade computacional e a similaridade do texto presente do elemento do programa e o texto apresentado na manifestação da falha. Para tanto, propuseram o uso de uma arquitetura baseada em redes neurais recorrentes do tipo LSTM (HOCHREITER; SCHMIDHUBER, 1997), conforme visto na Figura 3.1.

Figura 3.1: Abordagem do *DeepFL* para localização de defeitos baseado em rede recorrente



Fonte: Li et al. (2019).

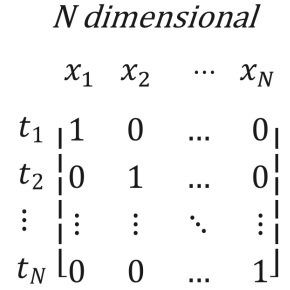
A camada de entrada da arquitetura proposta recebe grupos de características inferidas para cada elemento de código do conjunto de treino, mas antes de seguir para a camada da rede recorrente, contendo as unidade LSTM, os grupos de características são normalizados com a estratégia de preenchimento *0-padding*, ou seja, aqueles grupos que possuem menos características do que a quantidade máxima observada, têm o tamanho normalizado pela adição de características com valor 0.

Como a Figura 3.1 sugere, a saída do processamento realizado pela camada recorrente vai para uma camada responsável pela geração da suspeita. Nesse ponto, em vez de usar um PMC convencional, isto é, camadas sequenciais totalmente interconectadas, fazendo todas as características culminarem diretamente para dois neurônios (um responsável pela resposta positiva e outro pela resposta negativa, quanto ao elemento investigado ser defeituoso), os autores adaptaram camadas totalmente conectadas para grupos de características e subsequentemente combinaram as saídas dessas camadas até os dois neurônios de saída. Sendo avaliado no conjunto de dados do *Defects4J*, os resultados apresentados colocaram a proposta como o estado da arte na ocasião.

O trabalho desenvolvido por Zhang et al. (2021) se propõe a investigar a aplicação de redes neurais profundas ao problema da localização de defeitos de *software*. Os autores introduzem uma modelagem para o problema, em que, ao invés de se criar um modelo para aprender e estimar a associação entre um *statement* e o defeito, usa-se a própria associação como a suspeita. A arquitetura CNN proposta pelo trabalho tem como entrada uma partição da matriz original de cobertura, compreendendo uma quantidade de *h statements*, e usa como saída o vetor de resultados dos casos de teste. Dessa forma, o modelo acaba por aprender os padrões de cobertura que levam um caso de teste a ter um resultado malsucedido. Como saída, o modelo tem uma camada *softmax* correspondente à quantidade de casos de teste. Sendo assim, para que as predições possam ser feitas para os *statements*, é utilizada uma matriz V_{MxN}, onde M são os casos de teste virtuais e N são

os *statements* para os quais se pretende fazer a predição, conforme a Figura 3.2.

Figura 3.2: Abordagem do *DeepFL* para localização de defeitos baseada em RNN



Fonte: Li et al. (2019).

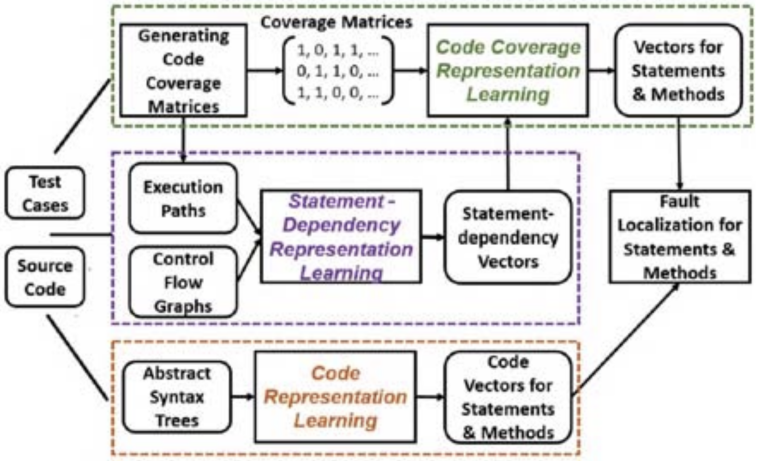
Pela figura, percebe-se que a matriz é preenchida com 1 apenas na diagonal principal, então, cada uma das saídas produzidas pode ser associada a um dos *statements* x_i, e esse é o valor assumido como a suspeita do *statement i*.

Com as adequações pertinentes, o trabalho avalia ainda uma arquitetura baseada em redes neurais recorrentes, com células do tipo LSTM, e uma rede neural com múltiplas camadas, como o PMC. Os resultados reportados indicam que a arquitetura convolucional foi mais adequada para tratar o problema, considerando a modelagem proposta. Apesar de apresentar uma nova modelagem para o problema, o trabalho contém algumas limitações, uma vez pela proposta não há como serem incorporadas outras fontes de informações; a arquitetura avaliada para a rede convolucional foi baseada apenas em filtros do tipo 2D, e, pelo menos em tese, a arquitetura recorrente seria comparável a uma convolucional 1D; e, a rigor, o que os modelos estão aprendendo é muito simular à construção de uma das várias heurísticas que já existem para o contexto em questão.

Outro trabalho utilizando redes neurais convolucionais para localização de defeitos foi proposto por Li, Wang e Nguyen (2021) e superou deficiências de muitas técnicas heurísticas baseadas em espectro de cobertura e espectro de mutação. O trabalho aborda o problema da localização de defeitos em nível de *statement* e em nível de métodos, isto é, em vez de localizar o defeito em um *statement*, localiza o método em que o *statement* defeituoso se encontra. Fundado na ideia de que apenas a matriz de cobertura ou *score* de mutação não carregam todas as informações necessárias para a localização de defeitos em *softwares* reais, o artigo propõe uma abordagem que se utiliza de todas as informações apresentadas na Figura 3.3.

Como pode ser visto, o trabalho propõe o uso de três tipos de informação: cobertura de código, dependência entre *statements* e representação do código. As matrizes de cobertura descritas, são, na verdade, tanto as informações de cobertura de fluxo de controle quanto as informações do espectro de mutação. Os dados de dependência são obtidos por intermédio da combinação dos dados de informações dos caminhos de

Figura 3.3: Visão das informações utilizadas no DeepLR4FL



Fonte: Li, Wang e Nguyen (2021).

execução dos casos de teste e dados de fluxo de controle. Finalmente, os dados que representam o código são obtidos a partir da Árvore Sintática Abstrata.

Sabendo que as três fontes de informação produzem dados representados de maneira diferente, utiliza-se uma série de estratégias para encontrar uma representação vetorial para cada uma dessas informações. Por representação vetorial, entenda-se uma estrutura com uma quantidade fixa de posições (características) que podem ser suficientes para codificar uma informação originalmente conhecida em outro espaço dimensional. Essa estratégia tem sido amplamente utilizada em Processamento de Linguagem Natural e é conhecida como *embeddings*.

Com os vetores que representam as diferentes informações, para cada *statement* ou para cada método, dependendo da estratégia, estes são combinados através de uma operação conhecida como Produto *Hadamard* (STYAN, 1973). O resultado dessa operação é uma matriz tridimensional, em que cada dimensão possui o mesmo tamanho do vetor que a gerou. Em outras palavras, tem-se um cubo de informações que descrevem o *statement*, ou método. A partir desse ponto, os dados são introduzidos em uma arquitetura convolucional convencional, com camadas 3D, com sub-amostragem e saída dada por uma *softmax*, classificando em amostra com defeito e amostra sem defeito.

Em suma, além do custo de obtenção dos dados de todas as fontes de informação utilizadas, um dos fatores chave da abordagem é a representação de dependência entre *statements*. Essa informação é obtida por meio dos dados gerados pela manifestação da falha, no caso dos programas Java, a própria *stack trace*. Sendo assim, a abordagem é fortemente dependente de uma informação que pode não estar disponível, pois, nem sempre o defeito causa uma colisão no local em que a falha se origina.

3.2 Localização de Defeitos Baseada em LLM

Recentemente, com o avanço no desenvolvimento dos Modelos de Linguagem de Grande Porte, muitas áreas da Engenharia de Software estão fazendo uso dos potenciais que tais modelos demonstram em diversos campos além do processamento de linguagem natural, a área que os originou.

Na área de depuração de código, especificamente na tarefa de localização de defeitos, um exemplo de exploração do potencial dos LLMs é o trabalho de Kang, An e Yoo (2024). Os autores apresentam uma técnica, denominada AutoFL, para localização de defeitos de forma automática, considerando métodos como a granularidade da localização, isto é, aponta-se um método e não uma linha ou *statement* como local provável do defeito. Como principais contribuições, esse trabalho conseguiu: a) utilizar LLMs, GPT 3.5 e GPT 4, contornando, em certa medida, a limitação de tamanho de entrada; b) superar outras técnicas de localização de defeitos baseadas em Redes Neurais; e, c) realizar uma validação da abordagem proposta utilizando desenvolvedores humanos.

A abordagem é fundamentada em cinco fases ao longo de dois estágios: No primeiro estágio, o de geração da explicação do defeito, a informação produzida pela execução do casos de teste negativo é passada para um LLM fazer um resumo do erro e decidir se faz chamadas a métodos envolvidos nas informações do erro e repetir esse processo por N vezes, sendo N um parâmetro da abordagem, e, por fim, o LLM é mais uma vez utilizado para gerar uma explicação do defeito, resumindo o resultados de todas as interações realizadas. Para as interações, o LMM pode receber informações de cobertura de classes e métodos, bem como os fragmentos de código propriamente. No segundo estágio, o estádio de predição da localização do defeito, o LLM é requerido para apontar qual seria o método mais propenso a ser a causa do defeito e, consequentemente, retorna o método como sendo o local do defeito. Essa metodologia é repetida m vezes e, no final, há uma normalização dos scores para criação do ranking de suspeita dos métodos. Na prática, todos os métodos cobertos pelos casos de teste, entram no ranking, mas com privilégio para aqueles que foram preditos pela abordagem como suspeitos.

O trabalho utilizou os modelos GPT 3.5 e GPT 4 em um experimento que envolveu problemas reais em linguagem de programação Python e Java. Considerando a métrica de avaliação ACC@1, a proposta superou outras técnicas de localização de defeitos em nível de método. Além disso, os autores conduziram um experimento com desenvolvedores para avaliar a qualidade e relevância das explicações dos *defeitos* aprontadas pela abordagem. De acordo como os resultados reportados, mais da metade dos participantes aprovaram os textos em linguagem natural gerados automaticamente pelo LLM para explicar o defeito.

Apesar de apresentar resultados interessantes, o trabalho traz importantes limi-

tações, tais como a dependência de execução de casos de teste ao longo das interações com o LLM e a predição do defeito ocorrer apenas no nível de método, o que, em alguns cenários pode ainda significar um largo esforço de inspeção por parte do desenvolvedor.

Outro trabalho recente com forte impacto na área de localização de defeitos de forma automática, foi apresentado por Yang et al. (2024). O artigo propõe a criação de um novo modelo baseado no *fine-tuning* do CodeGen que, proposto por Nijkamp et al. (2023), é um LLM específico para a geração de código. A técnica resultante denominada LLMAO é capaz de realizar a localização de defeitos em código-fonte, em nível de *statement* considerando as linguagens Python, Java e C. A inovação central deste trabalho é a introdução de uma técnica de localização de defeitos que não depende da execução de casos de teste, diferentemente de outras abordagens existentes. Por outro lado, supõe a necessidade de adaptação e treino de um novo modelo, sobre o modelo base. Além disso, o modelo se destaca pela sua capacidade de detectar vulnerabilidades de segurança em códigos, superando técnicas tradicionais de localização de defeitos baseadas em aprendizado de máquina.

A abordagem proposta funciona da seguinte forma: a entrada do sistema é um trecho de código defeituoso e a saída é uma lista de probabilidades indicando quais linhas de código têm maior propensão de serem a causa dos defeitos. O processo segue os seguintes passos: **Tokenização e Extração de Representação Vetorial**. Tanto no treinamento quanto na inferência, o código-fonte original é *tokenizado* e, em seguida, enviado para um LLM pré-treinado, que extrai a representação vetorial de cada linha de código. Importante destacar que o LLM base não passa por um novo treinamento, sendo utilizado apenas para gerar essas representações; **Modelagem das Relações entre Linhas de Código**. A partir das representações vetoriais obtidas, um modelo bidirecional é treinado para modelar as relações entre as diferentes linhas de código e suas respectivas propensões a apresentar defeitos. Este modelo bidirecional é composto por um adaptador do tipo *transformer* com duas camadas; **Entropia Binária Cruzada e Ajuste de Pesos**. Para ajustar os pesos do modelo bidirecional, utiliza-se a entropia binária cruzada, tendo como base uma anotação de diff (diferenciação entre a versão incorreta e a versão corrigida de código) que indica a resposta correta para cada linha, ou seja, uma marcação binária sobre a presença ou não de defeito na respectiva linha.

O modelo proposto oferece uma abordagem inovadora para a localização de defeitos sem a necessidade de execução de casos de teste, e também é pioneiro em apresentar resultados positivos na detecção de vulnerabilidades de segurança no nível de código. Comparado com outras técnicas baseadas em aprendizado, o modelo mostrou-se superior, posicionando-se como o estado da arte em termos de localização de defeitos.

Apesar dos avanços, o estudo apresenta alguns pontos de discussão e limitações. Por exemplo, considerando a limitação do tamanho de contexto suportado pelo modelo

base utilizado, um limite de 128 linhas de código foi imposto, forçando-se assim a elaboração de uma heurística para que um arquivo com mais linhas de código possa ser tratado. Além disso, como se supõe um processo de *fine tune*, corre-se o risco de haver uma forte dependência de um volume alto de dados para que o ajuste do modelo bidirecional seja efetivo.

3.3 Considerações Finais

Neste capítulo abordagens de localização de defeitos foram descritas sob duas principais vertentes: técnicas baseadas em aprendizado de máquina e aquelas que usam Modelos de Linguagem de Grande Porte. Na primeira vertente, as técnicas baseadas em aprendizado de máquina apresentadas diferem-se principalmente no tipo de modelo utilizado (Redes Neurais Convolucionais, Recorrentes etc.) e na forma como elas representam e combinam as diferentes fontes de informação (cobertura de código, dependência entre *statements*, representação do código). De maneira geral, essas técnicas obtiveram resultados superiores às abordagens tradicionais de localização de defeitos baseadas em espectro de cobertura e mutação, porém, com níveis de precisão ainda modestos em conjuntos de dados tipicamente utilizados, como o Defects4J.

Na segunda vertente, foram discutidos trabalhos recentes que utilizaram LLMs como base para a definição de abordagens de localização de defeitos. As técnicas dessa vertente apresentam resultados superiores, posicionando-se como o novo estado da arte. O AutoFL aplica um LLM para sintetizar informações contextuais e oferecer uma explicação detalhada dos defeitos. Enquanto o AutoFL ainda depende de casos de teste para gerar predições, a técnica proposta denominada LLMAO inova ao localizar defeitos sem essa necessidade, destacando-se também pela capacidade de detectar vulnerabilidades de segurança no código.

Ambas as vertentes de técnicas mostraram-se promissores, com os LLMs alcançando maior precisão, mas enfrentando desafios de escalabilidade e dependência de grandes volumes de dados para treinamento. Ademais, essas técnicas recentes, baseada em LLM, ainda possuem limitações, como dependência de execução de casos de teste, restrições no tamanho do código-fonte analisado ou dependência de alto volume de dados para realização de *fine-tuning*.

CAPÍTULO 4

Sistematização e Formalização de Aspectos-Chaves na Localização Automática de Defeitos de Software

Este capítulo apresenta uma das contribuições centrais desta tese: a sistematização e formalização dos aspectos fundamentais para o desenvolvimento de métodos de localização automática de defeitos de software. Ao longo dos anos, a pesquisa na área tem progredido com o uso de técnicas que incorporam informações de múltiplas fontes, cada uma trazendo diferentes perspectivas sobre o comportamento do *software*. No entanto, essa diversidade de abordagens tem resultado em uma falta de padronização e clareza metodológica, dificultando a replicação dos estudos e a comparação direta entre as técnicas propostas.

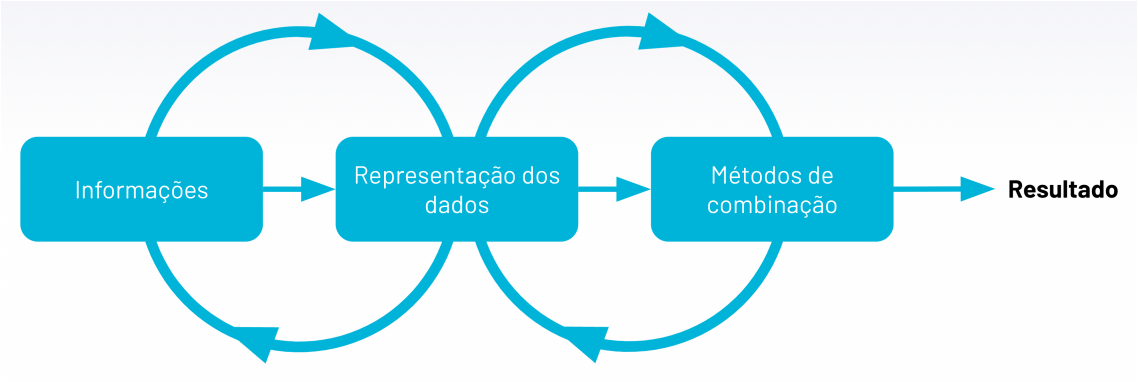
Um dos objetivos deste trabalho é propor um *framework* estruturado que organiza aspectos fundamentais para ferramentas de localização de defeitos, proporcionando uma base metodológica que pode ser utilizada tanto por pesquisadores quanto por profissionais da área. Esta sistematização facilita a replicação e a comparação dos métodos, contribuindo para o avanço contínuo da área ao fornecer uma estrutura clara para a categorização e entendimento das técnicas de localização de defeitos.

Conforme pôde ser notado tanto no Capítulo 2, dos conceitos fundamentais, quanto no Capítulo 3, dos trabalhos relacionados, as metodologias de resolução do problema da localização de defeitos, utilizam informações provenientes de diferentes fontes para a análise dos programas defeituosos. Contudo, observou-se uma lacuna na sistematização desses aspectos comuns, que são fundamentais para o sucesso das técnicas de localização de defeitos. A falta de uma abordagem estruturada limita o potencial de comparação e evolução dos métodos existentes, criando a necessidade de uma formalização que permita maior clareza e replicabilidade.

Neste capítulo, portanto, é proposta uma formalização e discussão desses aspectos essenciais para o desenvolvimento de métodos eficazes de localização automática de defeitos de software. Dessa forma, tanto os trabalhos anteriores, quanto as propostas

apresentadas nos capítulos seguintes, podem ser categorizadas quanto aos aspectos estabelecidos neste capítulo. A proposta de formalização dos aspectos-chaves para a localização de defeitos é dividida em três componentes principais: Caracterização da Informação, Representação dos Dados e Método de Combinação. Naturalmente, esses componentes influenciam o resultado final da localização de defeitos, isto é, têm impacto na acurácia da identificação dos elementos defeituosos, uma vez que se inter-relacionam de forma a criar uma estrutura integrada, como ilustrado na Figura 4.1.

Figura 4.1: Aspectos fundamentais à construção de métodos para localização de defeitos de software



Fonte: Elaborada pelo autor.

4.1 Aspectos-chaves para Localização Automática de Defeitos de Software

A sistematização apresentada neste capítulo facilita a exploração de diferentes combinações de componentes, proporcionando clareza quanto ao custo e impacto de cada um na precisão das técnicas de localização de defeitos. O termo “componente” refere-se à uma estratégia computacional específica que implementa um dos aspectos-chaves discutidos, levando em consideração as características associadas a esses aspectos. A seguir, cada um desses aspectos será detalhado, discutindo-se como, eventualmente, influenciam o desempenho e a eficácia dos métodos de localização automática de defeitos.

Os aspectos-chaves possuem uma natureza de categorização, de modo que cada um deles deve ser preenchido com a descrição equivalente ao que se percebe na solução de localização de defeitos. Tais aspectos-chaves possuem também atributos que podem ser utilizados para categorizar ou avaliar as soluções de localização de defeitos mais detalhadamente em relação ao aspecto-chave correspondente.

4.1.1 Caracterização da Informação

A eficácia de um método de localização de defeitos está diretamente ligada à qualidade da informação disponível sobre o software em análise. Esta seção propõe uma análise das características que definem a qualidade dessa informação, destacando cinco atributos essenciais: Disponibilidade, Custo de Aquisição, Origem, Natureza e Tipo.

Disponibilidade

Refere-se à existência de informações relevantes para a localização de defeitos em um *software* específico ou em um conjunto de *softwares* que compartilham certas características, como linguagem de programação ou plataforma. A relevância desta característica é evidente, pois informações que melhoram a acurácia em um cenário podem não estar disponíveis em outros, limitando a aplicabilidade e a generalização do método. Por exemplo, uma abordagem que depende de dados gerados por relatórios de erro de um compilador pode ser eficaz apenas em ambientes onde esses relatórios estão disponíveis, restringindo seu uso a certos tipos de defeitos e contextos. De forma prática, propõem-se que esse atributo seja avaliado pela anotação de um dos níveis de adequação abaixo:

- ***Não se aplica:*** caso em que não existe possibilidade de obtenção da informação. Por exemplo, informação textual do *log* de erro em ambiente que não gere esse tipo de dado;
- ***Baixa***: caso em que existe a possibilidade de obtenção da informação, mas esta raramente está disponível ou o custo de aquisição é alto. Por exemplo, informação de uso de memória durante a execução de cada elemento de código;
- ***Média:*** quando a informação pode ser obtida, mas está totalmente condicionada a alguma ferramenta, documentação ou procedimento adicional à execução do código. Por exemplo, informação do espectro de cobertura;
- ***Alta:*** quando a informação está sempre disponível a um custo computacional igual ou inferior à execução do próprio código defeituoso. Por exemplo, o próprio código-fonte defeituoso.

Custo de Aquisição

Este atributo representa o esforço necessário para tornar a informação disponível para a técnica de localização de defeitos, abrangendo tanto o esforço computacional quanto o esforço intelectual. O esforço computacional inclui medidas como tempo de processamento e uso de armazenamento, enquanto o esforço intelectual envolve a criação de algoritmos personalizados e o entendimento de lógicas complexas para extrair a informação necessária. O Custo de Aquisição deve ser cuidadosamente avaliado, pois pode

determinar a Disponibilidade da informação e, consequentemente, impactar a acurácia do método. Por exemplo, uma informação que só pode ser obtida durante a execução do *software* pode ser valiosa, mas seu alto custo de aquisição pode limitar sua viabilidade em cenários que demandem maior eficiência computacional. Novamente, propõem-se que esse atributo seja avaliado de acordo com níveis de adequação, conforme detalhado a seguir:

- ***Baixo***: casos em que o esforço computacional, de processamento e armazenamento, é igual ou inferior àquele requerido para execução do programa sob análise. Por exemplo, informação da saída produzida pela execução do programa aplicando-se um caso de teste negativo;
- ***Médio:*** quando o esforço computacional é maior do que aquele caracterizado como baixo ou a obtenção da informação requer uso de ferramentas complementares ao código-fonte sob teste. Por exemplo, informação de espectro de cobertura, que para obtenção exige-se o emprego de ferramentas capazes de coletar o dado de exercitação de cada elemento do código para cada caso de teste;
- ***Alto:*** quando para se obter a informação desejada, são aplicadas múltiplas estratégias (ou ferramentas) classificadas como de custo médio ou ainda quando se exige a elaboração de algoritmos específicos que combinem múltiplas ferramentas ou instrumentalize o código-fonte. Por exemplo, informação de heurísticas baseadas na cobertura do espectro de mutação e uso de memória baseada em ponto de checagem durante a execução do código.

Origem

Este atributo diz respeito à fonte de onde a informação é extraída dentro do ecossistema do *software*. A origem da informação pode variar de componentes diretamente relacionados ao código-fonte, como uma Árvore Sintática Abstrata (AST), a elementos externos, *logs* de execução e bibliotecas de terceiros. A escolha da origem da informação influencia diretamente outras características, como Disponibilidade e Custo de Aquisição. Por exemplo, informações derivadas de componentes externos ao código podem ser custosas computacionalmente, tornando-as menos viáveis para determinados métodos de localização de defeitos. Formalmente, esse atributo deve ser avaliado a partir do uso de uma das seguintes descrições categóricas:

- ***O próprio código***: quando o código-fonte é diretamente usado como informação relevante para a localização do defeito;
- ***Derivada do (ou relacionada ao) código***: informação que represente o código-fonte, como AST, ou que pode ser extraída diretamente do código sem necessidade de execução, como estilo de escrita.

- ***Externa ao código***: qualquer informações relativa à documentação do programa ou obtida pela execução do programa, como histórias de usuários ou dados de cobertura de fluxo de controle.
- ***Híbrida*** quando resulta da combinação de qualquer das origens anteriores.

Natureza

A natureza da informação refere-se ao momento em que a informação pode ser obtida, em relação à execução do método de localização de defeitos: durante a execução (informação dinâmica) e antes (informação estática). A natureza da informação é um fator crítico tanto para a acurácia final da localização de defeitos quanto para o Custo de Aquisição. Informações dinâmicas, por exemplo, podem oferecer uma visão mais detalhada do comportamento do *software*, mas exigem monitoramento contínuo e podem aumentar significativamente o custo computacional envolvido para a obtenção dos dados. Assim, este atributo suporta duas descrições categóricas:

- ***Estática***: quando a informações pode ser obtida antes do início da execução do método de localização de defeitos. Ex.: dados do espectro de cobertura ou documentação do software;
- ***Dinâmica***: quando a informação deve ser obtida durante a execução do método de localização de defeitos. Ex.: dados de saída de uma execução do código incorreto durante a busca pelo local defeituoso;

Importante ressaltar que esse atributo refere-se à existência prévia da informação, ou seja, antes de que se execute a técnica de localização de defeitos já se tenha acesso aos dados. Esse fator é relevante pois ainda que o custo de aquisição da informação possa ser elevado, a possibilidade de processamento *off-line*, em relação ao método de localização de defeitos, pode viabilizar o uso da informação.

Tipo

Este atributo diz respeito à semântica que a informação carrega, em outras palavras, representa sobre o que ou sobre qual aspecto do software a informação diz respeito. Essa caracterização tem fortes implicações na definição da estrutura de dados capaz de representar a informação e, consequentemente, na adequação com a técnica que efetivamente produzirá a localização de defeitos. Por exemplo, alguns trabalhos propõem a utilização de espectro de cobertura de casos de teste para inferir um local no código mais provável de ser o defeito. Para se utilizar dessa informação, um método poderia processar a matriz de cobertura diretamente como uma estrutura de dados do tipo matriz, ou seja, não seriam necessárias transformações computacionalmente complexas para lidar

com esse tipo de informação. Por outro lado, considerando a utilização de um texto em linguagem natural para corroborar a localização de defeitos, seria muito provável que tal informação não pudesse ser usada antes de uma transformação para uma representação de dados mais compatível com as técnicas de localização de defeitos baseadas em redes neurais convolucionais ou recorrentes. Assim, podem ser assumidos, não exaustivamente, os seguintes as seguinte descrições categóricas para este atributo:

- Árvore Sintática Abstrata (AST);
- Cobertura de Fluxo de Controle;
- Cobertura de Fluxo de Dados;
- Cobertura de Mutantes;
- Estatísticas sobre o código-fonte;
- Documentação em linguagem natural;
- Similaridade de código/texto;
- Complexidade de código;
- *Logs* de execução;
- Gráficos de fluxo de controle;
- Caminhos de execução;
- Histórico de edições do código;

4.1.2 Representação dos dados

A representação dos dados tem forte relação com a estrutura computacional utilizada para armazenar as informações sobre o *software* em que se pretende localizar defeitos. Essa representação é fundamental, pois pode influenciar a eficiência e a eficácia das técnicas de localização. Exemplos de representação incluem o uso de matrizes para modelar a cobertura de casos de teste, árvores para representar uma Árvore Sintática Abstrata (AST), grafos para dependências em fluxo de controle, e até a incorporação de dados textuais, como uma *stack-trace* ou documentação em linguagem natural.

Ao definir a representação dos dados, é essencial considerar as características discutidas a seguir.

Consistência

Refere-se à capacidade de uma estrutura de dados, ao ser usada para representar computacionalmente uma informação, preservar a semântica original dos dados envolvidos. Ao transformar dados de uma representação para outra, pode ocorrer perda de precisão ou expressividade, comprometendo a qualidade da informação. Por exemplo, ao codificar informações de *statements* para uso em técnicas de localização de defeitos, diferentes métodos de codificação podem preservar a semântica com variados graus de

precisão. A escolha da representação deve, portanto, balancear a necessidade de precisão com a viabilidade de uso da informação em diferentes contextos. Idealmente, a consistência mais privilegiada nos cenários em que a forma como a informação é originalmente representada pode ser utilizada diretamente na metodologia de combinação da informação, sem necessidade de uma representação intermediária. De forma objetiva, propõem-se que esse atributo seja avaliada na perspectiva de níveis de adequação abaixo:

- ***Baixa***: caso em que uma determinada representação de informação não é capaz de preservar a semântica da informação original. Em outras palavras, qualquer função de transformação aplicada aos dados na representação original, resultará em um conjunto vazio ou com elementos sem utilidade para a localização de defeitos na representação alvo. Por exemplo, tratar uma documentação em linguagem natural com vetores de frequência de termos resultará em pouca ou nenhuma preservação da semântica expressada originalmente;
- ***Média***: quando a representação da informação é capaz de modelar a informação original com algum nível de perda de detalhes dos dados. Por exemplo, uso de uma representação de vetor de inteiros para representar o código-fonte a partir da frequência de ocorrência dos termos presentes no código. Apesar de se possível se extrair a característica de distribuição de termos no código, a dependência semântica entre tais termos seria pedida, o que resultaria em uma representação menos expressiva do que a original, em texto plano;
- ***Alta***: quando a representação da solução preserva completamente os detalhes da informação original. Por exemplo, uso de matrizes para representar dados de cobertura de fluxo de controle, onde as linhas podem representar os elementos de código, as colunas os casos de testes e os valores 1 ou -1, quando presente na célula c_{ij}, indicar que o elemento i é exercitado pelo caso de teste j. Nessa representação, toda informação relativa à cobertura de casos de teste mantém-se preservada.

Compatibilidade

Trata da adequação da representação adotada aos métodos que serão utilizados para processar a informação. Se a representação não for compatível com o algoritmo que manipula os dados, pode ser necessária a aplicação de transformações adicionais, o que pode comprometer a consistência e a integridade dos dados, ou, simplesmente diminuir a expressividade da informação e, consequentemente, a precisão da abordagem como um todo no tocante à localização do defeito. Por exemplo, técnicas baseadas em redes neurais artificiais exigem que os dados sejam representados numericamente. Informações que já possuem uma representação numérica são mais facilmente compatíveis e tendem

a maximizar o desempenho dessas técnicas. Assim, para avaliação desse atributo da representação de informação, deve ser utilizado um dos níveis de adequação a seguir:

- ***Nenhuma***: quando a representação da informação não pode ser utilizada diretamente pelo algoritmo destinado a manipular os dados e não existe uma representação intermediária que viabilize esse uso preservando o mínimo de informação útil à localização do defeito;
- ***Parcial***: quando a representação da informação não pode utilizada diretamente pelo algoritmo destinado a manipular os dados na tentativa de vinculá-los, mas existe uma representação intermediária que possibilita o uso, ainda que com alguma perda. Por exemplo, histórico de alteração do programa pode envolver alterações em diversos artefatos de código e a representação mais precisa das alterações significaria os textos alterados, porém, para uso em uma metodologia de localização de defeitos baseada em redes neurais convolucionais, dada a impossibilidade de uso direto dos textos, poderia ser adotada uma representação vetorial reunindo estatísticas do histórico de alterações.
- ***Total***: quando a representação da informação pode ser utilizada diretamente no algoritmo de manipulação dos dados sem necessidade de transformação ou ajustes. Por exemplo, documentação em linguagem natural utilizada como informação para localização de defeitos baseada em Modelos de Linguagem.

Escalabilidade

Refere-se à capacidade da representação de permanecer eficiente e viável em cenários de grande volume de dados. A viabilidade de uma representação de dados deve ser avaliada em termos de sua capacidade de lidar com grandes quantidades de informação sem perda significativa de desempenho. Por exemplo, uma técnica de localização de defeitos que utiliza um algoritmo genético pode se tornar inviável se a representação dos *statements* em um cromossomo resultar em estruturas de dados muito grandes, dado que algoritmos genéticos possuem limitações conhecidas ao lidar com cromossomos extensos. Objetivamente, esse atributo pode ser avaliado a partir dos níveis de adequação abaixo, sempre levando-se em consideração o algoritmo adotado para manipulação dos dados:

- ***Baixa***: em casos em que a representação da informação não suporta aumento no volume de dados ou que esse aumento resulta em perda de informação. Por exemplo, utilizar uma representação vetorial inteira para modelar a descrição em linguagem natural resultaria em uma solução pouco escalável para descrições com vocabulário grande;
- ***Média***: quando a representação da informação suporta aumento no volume de dados, mas com perda no desempenho do algoritmo de combinação. Por exemplo,

usar texto plano para representar o código-fonte é completamente compatível com aumento no tamanho do código, porém, em um contexto de aplicação de LLMs, esse aumento pode significar uma perda de desempenho na localização de defeitos pois o modelo pode não ser capaz de se atentar a todos os detalhes do texto;
- ***Alta***: quando a representação da informação suporta aumentos no volume de dados sem impacto no desempenho do algoritmo utilizado para manipulação dos dados. Por exemplo, utilizar matrizes para representar os dados de cobertura de fluxo de controle em um cenário de utilização de heurísticas como Ochiai seria robusto para aumentos no volume de elementos de código e casos de teste sem impacto direto no desempenho da técnica de localização de defeitos.

4.1.3 Métodos de Combinação

Os métodos de combinação referem-se aos algoritmos e/ou abordagens que, utilizando as informações representadas, analisam as relações entre os dados para realizar a localização de defeitos no *software*. A escolha adequada desses métodos é crucial para a eficácia de qualquer técnica que integra múltiplas fontes de informação na localização de defeitos. Nesta pesquisa, a escolha do método de combinação deve levar em consideração os atributos que passam a ser detalhados a seguir.

Custo

Refere-se às exigências computacionais associadas à execução do método de combinação de informação, como a complexidade de tempo e o espaço de armazenamento requeridos. Com o crescente uso de técnicas de Aprendizado Profundo (*Deep Learning*), que muitas vezes requerem recursos de *hardware* especializado, o custo computacional pode se tornar proibitivo para alguns usuários. Portanto, é essencial realizar um balanço entre o aumento da acurácia e o esforço computacional necessário para uso de um determinado método. Redes neurais profundas podem oferecer alta acurácia, mas o custo associado ao treinamento e execução pode ser inviável para aplicações de grande escala ou em ambientes com recursos de *hardware* limitados. Para avaliação deste atributo, propõe-se que sejam utilizados os níveis de adequação abaixo:

- ***Baixo***: quando os recursos computacionais (memória e processamento) requeridos para a execução do método de combinação não são superiores aos recursos exigidos para a execução do programa original. Por exemplo, para execução uma heurística como Tarantula, os recursos computacionais mínimos são os mesmo utilizados para execução do programa original.
- ***Médio*** quando os recursos computacionais exigidos para execução do método de combinação podem ser os mesmos da execução do programa original, mas

o consumo de memória ou processamento é superior. Por exemplo, a utilização de Algoritmos Genéticos para a construção de heurísticas customizadas a partir de dados de cobertura pode ocorrer nos mesmos recursos computacionais da execução do programa com defeito, mas com uma demanda superior em consumo de memória e processamento.

- ***Alto*** quando os recursos computacionais necessários para a execução do método de combinação são especializados, distintos daqueles utilizados para execução do programa original por imposição das características das técnicas algorítmicas empregadas. Por exemplo, é comum que abordagens de localização de defeitos baseadas em redes neurais profundas exijam a utilização de hardware específico (GPU) seja para o processo de treinamento dos modelos, seja para a inferência de respostas. Em um cenário em que o software original, com defeito, não requeira uso de GPU, a aplicação de redes neurais profundas como método de combinação para se ter a localização de defeitos representa um custo alto.

Flexibilidade

Refere-se à capacidade do método em suportar diferentes formas de representação de dados e em ser adaptável a novas fontes de informação. Na prática, a eficácia na localização de defeitos muitas vezes depende da incorporação de múltiplos tipos de dados, desde o código-fonte, informações de cobertura de testes, até documentação em linguagem natural. Métodos que não conseguem se adaptar a essa diversidade de dados tendem a ter uma aplicabilidade limitada. Portanto, abordagens que permitem alto grau de personalização e integração com diferentes fontes de dados tendem a maximizar o potencial da técnica de localização de defeitos. De forma prática, para uso deste atributo pode-se utilizar os níveis de adequação a seguir:

- ***Baixa***: quando a abordagem ou o método de combinação é específico para um tipo único de informação. Por exemplo, a heurística Ochiai, projetada exclusivamente para uso de variáveis extraídas de dados de cobertura, não tem flexibilidade para uso sobre outro tipo de informação;
- ***Média***: quando o método de combinação (algoritmo de manipulação de dados) suporta utilização de diferentes tipos de dados desde que modelados para uma representação de dados específica requerido por cada variação de método de combinação. Por exemplo, uso de redes neurais convolucionais suportam diferentes fontes de informação desde que representadas como matrizes n-dimensionais.
- ***Alta***: quando o método de combinação suporta diferentes tipos de informação sem a necessidade de conversão de representação de informação para uma representação

única. Por exemplo, uma abordagem baseada em LLM pode incorporar diferentes tipos de informações sem necessidade de representações de informações especiais.

Escalabilidade

Refere-se à capacidade de um método de manter sua eficácia quando aplicado a grandes volumes de dados ou a sistemas de *software* complexos. No entanto, ao considerar a escalabilidade, é fundamental equilibrar o custo computacional com o tamanho do código em análise. Embora técnicas que funcionam bem em pequena escala possam enfrentar desafios à medida que o volume de dados aumenta, é igualmente relevante desenvolver e aplicar métodos eficientes em cenários de menor escala, onde o custo computacional é uma preocupação significativa.

Em contextos de pesquisa e desenvolvimento com recursos limitados, a escalabilidade deve ser considerada dentro dos limites práticos e econômicos. A capacidade de localizar defeitos com precisão em *software* de menor porte, a um custo computacional reduzido, é uma contribuição valiosa. Esse enfoque permite que técnicas eficazes e acessíveis sejam desenvolvidas e aplicadas, oferecendo soluções viáveis em ambientes onde recursos para experimentos em larga escala não estão disponíveis. Portanto, a escalabilidade deve ser balanceada com o custo, valorizando abordagens que proporcionem bons resultados mesmo em um escopo reduzido. Assim, para avaliação desse atributo, deve ser utilizado um dos níveis de adequação a seguir:

- ***Baixa***: quando a abordagem não suporta aumento no volume de dados, no sentido que a própria técnica impõe um limite de crescimento na quantidade de dados. Por exemplo, uma abordagem baseada em LLM pode ser limitada ao tamanho de contexto suportado pelo modelo;
- ***Média***: quando a abordagem suporta aumento no volume de dados de entrada, mas com degeneração do desempenho. Por exemplo, uma abordagem baseada em algoritmos genéticos para combinar diferentes variáveis de cobertura de fluxo de controle, poderia suportar um aumento da quantidade de variáveis disponíveis, mas à medida que o número aumenta a tendência é que o desempenho da abordagem se degenere;
- ***Alta***: quando a abordagem suporta aumento no volume de dados de entrada sem grande impacto no desempenho na acurácia da localização. Por exemplo, heurísticas baseadas em variáveis extraídas de dados de cobertura, que não sofrem degeneração de desempenho com aumento na quantidade de elementos de código.

A Tabela 4.1 apresenta um resumo de como essa proposta formaliza os aspectos-chaves para construção de soluções para o problema da localização de defeitos. Na coluna Aspecto-chave tem-se a descrição de cada um dos três elementos apontados; na coluna

definição, há a forma como cada um dos aspectos-chaves pode ser descrito quando utilizado para categorizar uma abordagem de localização de defeitos. Essa definição é uma descrição aberta, no sentido de que não existe uma lista exaustiva para tal; A coluna Atributos apresenta a lista de atributos que cada aspecto-chave possui e, por fim, a coluna Avaliação/Classificação indica, para cada um dos atributos, qual a forma de preenchimento.

Tabela 4.1: Resumo da formalização dos aspectos-chaves e os respectivos atributos relevantes para soluções de localização de defeitos de software.

Aspecto-chave	**Definição**	**Atributos**	**Avaliação/Classificação**
Caracterização da Informação	Descrição	**Disponibilidade**	Não se aplica, Baixa, Média, Alta
		Custo de aquisição	Baixo, Médio, Alto
		Origem	O Próprio Código
			Derivada do Código
			Externa ao Código
		Natureza	Estática, Dinâmica
		Tipo	Árvore Sintática Abstrata (AST)
			Cobertura de Fluxo de Controle
			Cobertura de Fluxo de Dados
			Cobertura de Mutantes
			Estatísticas sobre o código-fonte
			Documentação em linguagem natural
			Similaridade de código
			Complexidade de código
			Logs de execução
			Histórico de edições do código
Representação dos Dados	Descrição	**Consistência**	Baixa, Média, Alta
		Compatibilidade	Nenhuma, Parcial, Total
		Escalabilidade	Baixa, Média, Alta
Método de Combinação	Descrição	**Custo**	Baixo, Médio, Alto
		Flexibilidade	Baixa, Média, Alta
		Escalabilidade	Baixa, Média, Alta

A partir da formalização dos aspectos-chaves, pela definição dos atributos e domínio dos valores de cada atributo, foi realizada uma aplicação desse *framework* sobre os trabalhos relacionados a esta pesquisa que foram listados no Capítulo 3. Os detalhes dessa aplicação encontram-se no Apêndice I.

4.2 Considerações Finais

Em resumo, a sistematização e formalização dos aspectos fundamentais para a localização automática de defeitos de *software*, conforme apresentado neste capítulo, oferecem uma contribuição significativa para a área. Este *framework* não apenas facilita o entendimento, construção e avaliação de métodos atuais, mas também estabelece uma base para a adaptação e expansão de novas abordagens. Ao fornecer uma estrutura clara e adaptável, este trabalho apresenta um caminho para novas abordagens que podem melhorar a precisão e eficiência das técnicas de localização de defeitos em *software*, mesmo em contextos de recursos limitados.

A proposta de sistematização aqui delineada visa a preencher lacunas existentes na literatura, ao mesmo tempo em que oferece um arcabouço flexível que pode ser utilizado por futuros pesquisadores para enfrentar novos desafios. Com isso, espera-se que este *framework* contribua para orientar e fomentar avanços contínuos na área de localização de defeitos em *software*.

CAPÍTULO 5

Proposta de Localização de Defeitos em Software Utilizando Redes Neurais Convolucionais: Um Enfoque em Cobertura de Código

Neste capítulo, é apresentada uma proposta para a localização de defeitos baseada no uso de Redes Neurais Convolucionais e fundamentada na análise de informações de cobertura de fluxos de controle da execução de casos de teste. Esta abordagem busca aproveitar a capacidade das CNNs de identificar padrões em grandes volumes de dados, aplicando-as à tarefa de localizar defeitos em código-fonte. A proposta está intrinsecamente conectada aos aspectos sistematizados no capítulo anterior, especialmente na forma como os dados de cobertura são representados e combinados para aprimorar a acurácia do processo de localização de defeitos.

O foco desta abordagem está na exploração de dados gerados a partir da execução de casos de teste, uma informação que pode ser obtida com custo computacional relativamente baixo, a depender das ferramentas disponíveis para tal e da amplitude da suíte de teste, e que, quando bem utilizada, pode fornecer *insights* importantes sobre a localização de defeitos. Ao mesmo tempo, a proposta leva em conta os desafios de escalabilidade, reconhecendo a necessidade de um equilíbrio entre o tamanho do código e o custo computacional, conforme discutido na sistematização dos aspectos-chaves.

5.1 Abordagens

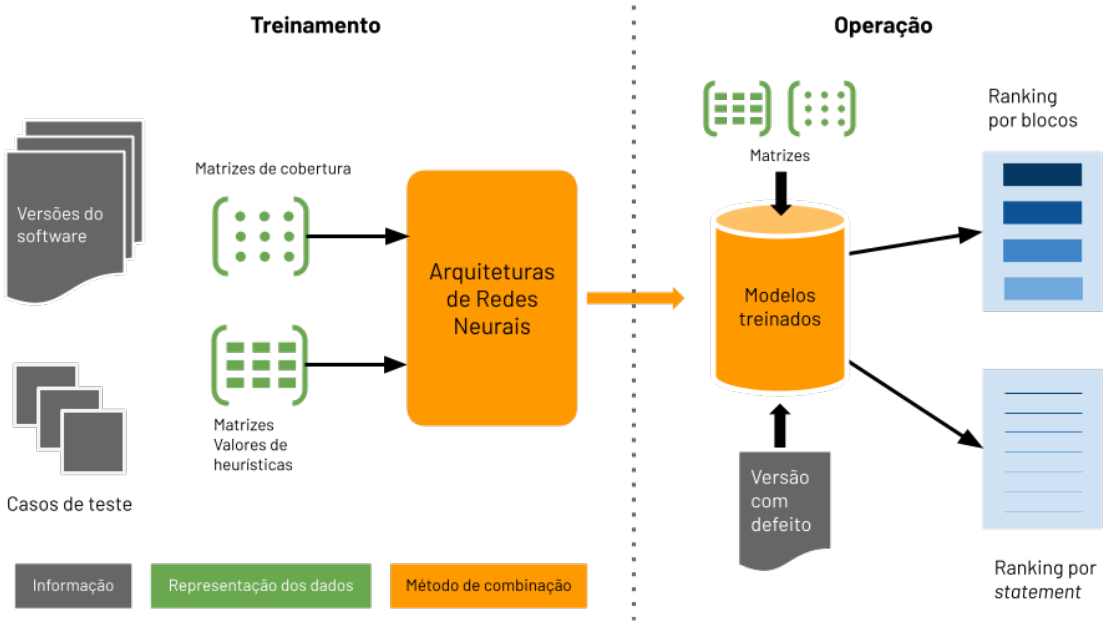
A proposta deste trabalho baseia-se na aplicação de informações de cobertura de código, obtidas por meio da execução de casos de teste, para a localização automática de defeitos em *software*. Dado que a presença de defeitos é confirmada por testes falhos, a coleta dessas informações de cobertura é uma tarefa de baixo custo computacional e,

como discutido anteriormente, essas informações podem ser representadas de maneira consistente para serem processadas por modelos de aprendizado de máquina.

Um dos pontos centrais da proposta é a utilização de CNNs, que foram escolhidas pela sua capacidade de lidar eficientemente com grandes volumes de dados e pela flexibilidade na adaptação a diferentes formas de representação de dados. As CNNs são particularmente adequadas para processar a matriz de cobertura, que cresce linearmente com o número de casos de teste e *statements*, e para gerar um *ranking* das regiões de código mais propensas a conter defeitos.

Conforme ilustrado na Figura 5.1, no contexto de um projeto de *software* com histórico de versões, é possível treinar um modelo de aprendizado de máquina usando as informações de cobertura de testes. Durante a fase de operação, o modelo retorna um *ranking* dos *statements* ou blocos de código mais prováveis de conter defeitos.

Figura 5.1: Visão geral da posposta para localização de defeitos de software



Fonte: Elaborada pelo autor.

Ao desenvolver essa abordagem, alguns desafios significativos foram identificados. Primeiramente, a matriz de cobertura pode se tornar esparsa, o que pode afetar o desempenho da rede neural durante o treinamento. Além disso, a evolução do *software* pode resultar em mudanças na suíte de testes e nos próprios *statements*, gerando uma matriz de cobertura com tamanhos variáveis em ambas as dimensões. Outro desafio é o desbalanceamento dos dados, pois a quantidade de *statements* que apresentam defeitos tende a ser significativamente menor em comparação aos *statements* que nunca foram identificados como defeituosos.

Esses desafios são abordados na proposta por meio de técnicas específicas de pré-processamento dos dados e adaptação das CNNs, que incluem estratégias de pre-

enchimento de dados e mecanismos para lidar com o desbalanceamento. Essas soluções estão alinhadas com os aspectos-chaves do *framework* discutido no capítulo anterior, assegurando que a abordagem proposta não só seja robusta, mas também escalável e aplicável a diferentes cenários de *software*.

5.1.1 Abordagem por *statements*

A abordagem por *statements* visa produzir um *ranking* dos *statements* de um programa, ordenados de acordo com a probabilidade estimada de cada *statement* conter um defeito. Para atingir esse objetivo, são utilizados modelos de redes neurais que fornecem saídas normalizadas entre 0 e 1, onde cada *statement* é avaliado individualmente na fase de operação, resultando em uma estimativa de sua propensão a conter um defeito. Em seguida, os *statements* são ordenados com base nesses valores, permitindo uma priorização dos locais mais prováveis de defeitos no código.

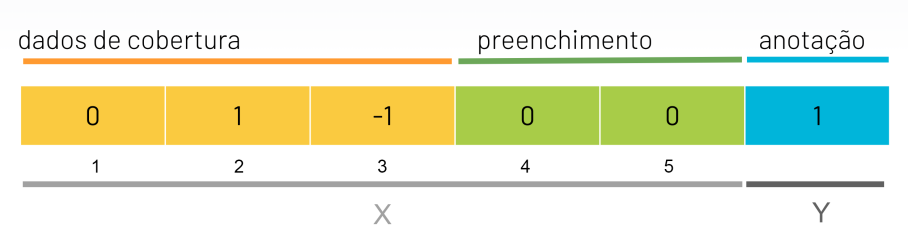
Considera-se a matriz $A_{M \times N}$ como a matriz de cobertura gerada pela execução da suíte de testes, onde M representa a quantidade de *statements* e N a quantidade de casos de teste executados. As entradas da matriz são definidas como $A_{ij} = 0$ quando o caso de teste j não executa o *statement* i, $A_{ij} = 1$ quando o caso de teste j executa o *statement* i com sucesso, e $A_{ij} = -1$ quando o caso de teste j executa o *statement* i, mas falha.

Para o treinamento dos modelos, cada amostra S corresponde a uma linha da matriz, representando os dados de cobertura associados a um *statement*, juntamente com a anotação que indica se o *statement* em questão foi identificado como local de um defeito.

Como dito anteriormente, no caso prático, a tendência é que a quantidade de casos de teste e de *statements* varie entre as versões. Sendo assim, para se construir um conjunto de dados para treinamento, que seja composto por amostras de diferentes versões, é necessário um procedimento para garantir que todas as amostras possuam o mesmo tamanho. Do ponto de vista da rede neural, é necessário que as amostras possuam a mesma quantidade de características. Portanto, nesse caso, utiliza-se o método de preenchimento que consiste em: iterar sobre todas as versões do software e descobrir qual a quantidade máxima de casos de teste em todas as suítes de teste vinculadas às versões. Em seguida, cada uma das amostras que tenham menos características do que a quantidade máxima de casos de teste é preenchida com zeros inseridos imediatamente antes da anotação, até que a amostra possua tantas características quanto a quantidade máxima de casos de teste. Finalmente, uma amostra de dados de treinamento para esta abordagem se parece como o exposto na Figura 5.2.

Como pode ser visto, as três primeiras posições do vetor representado na Figura 5.2, na cor amarelo, contém os dados de cobertura de três casos de testes que foram executados sobre a versão à qual pertence o referido *statement*. Na primeira posição X_1,

Figura 5.2: Representação de uma amostra para treinamento da abordagem por *statement*



Fonte: Elaborada pelo autor.

o valor 0 indica que o caso de teste CT_1, ao ser executado, não exercitou o *statement e*. Por outro lado, o $X_2 = 1$ indica que o caso de teste CT_2 exercitou o *statement e* e passou, enquanto $X_3 = -1$ implica que o caso de teste CT_3 exercitou o *statement e* e falhou. A porção verde da figura, com $X_4 = X_5 = 0$, indica um preenchimento realizado para que todas as amostras tenham o mesmo número de variáveis, nesse caso, 5. Por fim, o valor em azul, $Y = 1$, é a anotação binária do *statement* quanto a ser o local do defeito. Nesse, o *statement* seria o local do defeito, dada a anotação $Y = 1$.

Para avaliar o desempenho dessa abordagem, foram implementadas duas arquiteturas de redes neurais. A primeira, baseada no *Multilayer Perceptron* (MLP), é uma escolha clássica, adaptada para incorporar conceitos de redes neurais profundas, amplamente utilizada em problemas com representações semelhantes. A segunda arquitetura utiliza redes neurais convolucionais (CNNs) com filtros 1D, escolhidas por sua capacidade de capturar relações espaciais ao longo da matriz de cobertura, especialmente úteis para identificar padrões em agrupamentos de casos de teste que são relevantes para a inferência de defeitos.

Essas escolhas metodológicas estão alinhadas com os aspectos discutidos no *framework* proposto, garantindo que as soluções sejam robustas e adaptáveis a diferentes cenários de *software*. Os detalhes de implementação e a avaliação comparativa das duas arquiteturas serão discutidos no capítulo de experimentação (Capítulo 6), onde serão analisados os resultados obtidos em termos de eficácia na localização de defeitos.

5.1.2 Abordagem por Blocos

Nesta abordagem, ao invés de estimar a propensão ao defeito de cada *statement* individualmente, propõe-se a inferência da probabilidade de um bloco de *p statements* conter o *statement* responsável pelo defeito. Dessa forma, as amostras de treinamento consistem em blocos de *p statements*, permitindo que a análise seja feita em uma granularidade maior.

Tal como na abordagem por *statement*, a abordagem por blocos requer a equalização dos tamanhos das linhas da matriz de cobertura, que representam a quantidade de

casos de testes da versão do *software* sob investigação. O procedimento de preenchimento (*padding*) é aplicado para garantir que todas as amostras tenham o mesmo tamanho, com zeros adicionados para normalizar as dimensões entre diferentes blocos.

Há duas estratégias principais para a construção dos blocos: a divisão baseada em conteúdo de métodos ou funções e a divisão por fatiamento.

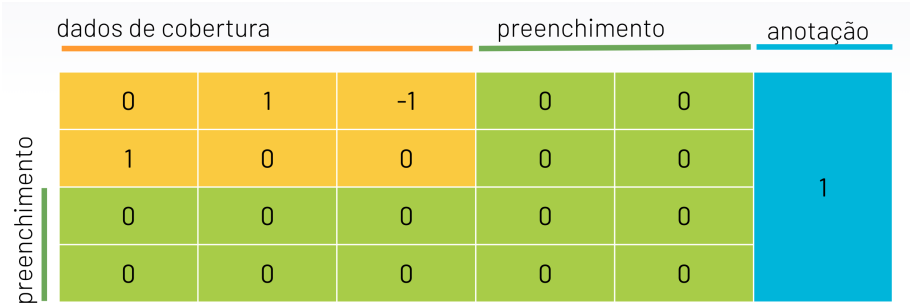
- **Divisão por Métodos/Funções**: Nesta estratégia, os blocos são formados por todos os *statements* de um método ou função. Essa abordagem é particularmente útil em códigos orientados a objetos, onde a estrutura de métodos/funções é bem definida. Para garantir a consistência entre os blocos, é necessário um preenchimento adicional, já que a quantidade de *statements* dentro de métodos pode variar significativamente, tanto dentro de uma versão quanto entre diferentes versões do *software*. O preenchimento garante que todas as amostras tenham o mesmo número de linhas, padronizando a representação dos blocos.
- **Divisão por Fatiamento**: Nessa estratégia, o tamanho do bloco é fixo e definido como um parâmetro ajustável. A ideia principal é expandir a noção de localização de defeitos de um único *statement* para uma região que possa conter o *statement* defeituoso. Essa flexibilidade permite a experimentação com diferentes tamanhos de bloco, incluindo variações nas quais a janela de corte é deslocada ao longo do código, oferecendo uma análise mais robusta de regiões do código potencialmente defeituosas.

A Figura 5.3 exemplifica uma amostra de treinamento para esta abordagem, onde blocos de *statements* são representados em relação à matriz de cobertura. A área verde na figura destaca a extensão do preenchimento necessário para garantir que todas as amostras sejam consistentes em termos de linhas e colunas. Essa estratégia de preenchimento é comum em aplicações de CNNs, como na visão computacional, em que a integridade da representação matricial é crucial.

É importante notar que, ao aplicar o preenchimento, uma parte considerável dos dados da amostra pode ser composta apenas por zeros, o que pode impactar a eficiência do modelo. No capítulo de experimentação, são discutidos em detalhes os desafios decorrentes dessa questão e os impactos observados.

Para avaliar a capacidade dessa abordagem em comportar diferentes informações, alternativamente, para cada um dos blocos, além da informação proveniente da matriz de cobertura produzida pela suíte de testes, podem ser adicionados valores de heurísticas que também usam os dados de cobertura para calcular um valor de propensão a falha para cada *statement*. Os dados das heurísticas são dispostos em uma matriz em que as linhas representam os *statements* e as colunas contém cada uma das heurísticas computadas. Dessa forma, tal matriz guarda coerência com a matriz de cobertura no que diz respeito às linhas.

Figura 5.3: Representação de uma amostra para treinamento da abordagem de blocos, para um caso hipotético em que o tamanho do bloco fosse 4 e 5 fosse a quantidade máxima de casos de teste considerando cada uma das versões do *software*



Fonte: Elaborada pelo autor.

Foi desenvolvida uma rede neural artificial profunda do tipo convolucional, com filtros 2D, para explorar as relações espaciais presentes nas matrizes de blocos. Essa escolha é motivada pela capacidade das CNNs de lidar com dados estruturados em matrizes, explorando padrões espaciais que possam indicar a presença de defeitos em blocos específicos de código. A arquitetura e os mecanismos utilizados serão detalhados no capítulo de experimentação, onde se avalia o desempenho dessa abordagem em diferentes cenários.

5.2 Caracterização da Proposta em Relação aos Aspectos-chaves da Localização de Defeitos de Forma Automática

Nesta seção, examina-se como a proposta de localização de defeitos em software, utilizando Redes Neurais Convolucionais, pode ser caracterizada do ponto de vista dos aspectos-chaves identificados para a localização automática de defeitos. A caracterização detalha a forma como os dados de entrada, as representações e os métodos de combinação utilizados na abordagem proposta interagem com esses aspectos.

Essa análise permite uma avaliação abrangente da aplicabilidade e escalabilidade da técnica proposta, destacando as vantagens de se utilizar CNNs no contexto de localização de defeitos com base em dados de cobertura de código.

A Tabela 5.1 resume essa caracterização, permitindo uma comparação estruturada entre os elementos teóricos discutidos e a implementação prática da proposta.

Tabela 5.1: Caracterização da proposta de localização de defeitos baseada em Redes Neurais Convolucionais segundo os aspectos-chaves.

Aspecto-chave	Classificação dos Atributos
Caracterização da informação	***Dados de espectro de cobertura***
Disponibilidade:	Médio (Dependente da ferramenta de geração)
Custo de aquisição:	Médio (Requer ferramentas específicas para coleta de dados)
Origem:	Externa ao código
Natureza:	Estática
Tipo:	Cobertura de fluxos de controle
Representação dos dados	***Matrizes***
Consistência:	Alta (Preserva totalmente as características)
Compatibilidade:	Total (Redes Neurais Convolucionais usam matrizes nativamente)
Escalabilidade:	Alta (CNNs não perdem desempenho com aumento das matrizes)
Métodos de combinação	***Redes Neurais Convolucionais***
Custo:	Alto (Pode vaiar em função da entrada e da arquitetura definida)
Flexibilidade:	Média (Admite combinação de informações, desde que matrizes)
Escalabilidade:	Alta (Capaz de lidar com grandes volumes de dados)

5.3 Considerações Finais

A proposta apresentada neste capítulo visa aprimorar a localização de defeitos pela introdução de uma abordagem que utiliza CNN para processar dados de cobertura de código de forma eficiente e escalável. As CNNs, que têm a capacidade de manipular grandes volumes de dados e identificar padrões complexos, tendem-se a ser adequadas para lidar com as características específicas das matrizes de cobertura, oferecendo um meio robusto de priorizar possíveis regiões defeituosas no código.

Apesar das limitações, como a esparsidade e o desbalanceamento de dados na matriz de cobertura, as técnicas de pré-processamento e adaptação propostas ajudam a mitigar esses desafios, permitindo que o modelo se adapte a diferentes cenários e versões de software, conferindo boa aplicabilidade da abordagem proposta.

A análise dos aspectos-chaves demonstra que a proposta é viável do ponto de vista da informação requerida para a localização dos defeitos, da representação da informação e do método proposto para combinação dos dados. Apesar de as CNNs caracterizarem-se por um alto custo computacional, a flexibilidade e escalabilidade desses

métodos conferem à proposta uma boa relação custo-benefício.

A proposta representa uma ideia promissora para a localização de defeitos em software, e a experimentação subsequente busca demonstrar a eficácia desta abordagem em diferentes em um contexto de softwares reais. Espera-se que o uso dessa metodologia possa contribuir para tornar o processo de depuração de software mais eficiente, auxiliando na rápida identificação de defeitos e na melhoria da qualidade do código entregue.

CAPÍTULO 6

Avaliação Experimental das Abordagens de Localização de Defeitos Usando Redes Neurais Convolucionais e Cobertura de Código

Este capítulo descreve os experimentos preliminares realizados para avaliar as abordagens propostas para a localização de defeitos em código-fonte, baseadas em redes neurais artificiais. Utilizando a base de dados de defeitos Defects4j, foram conduzidos dois experimentos, cada um com objetivos específicos, como será detalhado a seguir.

6.1 Descrição da Base de Dados e Preparação

Para testar as abordagens propostas, foi escolhida a base de dados Defects4j. Esta base, proposta por Just, Jalali e Ernst (2014), é composta por defeitos reais provenientes de seis projetos de software desenvolvidos em Java. Além dos dados de defeitos, o Defects4j oferece uma infraestrutura abrangente, incluindo ferramentas de software e integrações que suportam a realização de pesquisas na área de Engenharia de Software.

Os dados de cobertura dos casos de teste, essenciais para os experimentos, foram obtidos do repositório fornecido por Pearson et al. (2017). Estes dados, amplamente utilizados em pesquisas correlatas, foram pré-processados para se obter as amostras de treinamento para as redes neurais. Para a abordagem de localização de defeitos por *statements* individuais, cada *statement* foi vinculado ao rótulo indicando se aquele é ou não um local defeituoso; no caso da abordagem de localização de defeitos por blocos de código, cada amostra foi montada com o conjunto de *statements* do bloco, vinculado ao rótulo indicando se o defeito está contido no bloco.

Os projetos contidos na base Defects4j e no repositório de cobertura são organizados em versões específicas, denominadas *bugs*, que identificam defeitos catalogados. Para cada *bug*, é possível acessar os dados do projeto antes e depois da correção do defeito, permitindo a análise detalhada das falhas e suas respectivas correções.

Neste estudo, foram selecionados dois projetos para os testes preliminares: Lang, que possui 65 *bugs* catalogados, e Mockito, com 38 *bugs*. É importante destacar que, para cada versão com defeito, o defeito pode estar presente em múltiplos *statements*.

6.2 Experimento 1 – Localização de Defeitos por Statements

Este experimento foi projetado para verificar a eficácia das arquiteturas propostas na identificação de padrões em *statements* defeituosos e não defeituosos com base nos dados de cobertura.

6.2.1 Seleção do Conjunto de Dados

Para o Experimento 1, foram selecionadas 33 versões do projeto Mockito e 53 do Lang. Durante a mineração dos dados, foi constatada a impossibilidade de capturar automaticamente a linha exata de código onde o defeito foi corrigido para todas as versões, o que resultou na seleção dessas 86 versões.

Os conjuntos de dados de treinamento foram construídos a partir de todos os *statements* presentes nas versões selecionadas, totalizando 46.099 *statements* para o projeto Lang e 62.696 para o Mockito. Para mitigar o desbalanceamento entre *statements* com e sem defeito, adotou-se uma técnica de amostragem que mantém uma proporção de duas amostras sem falha para cada amostra com falha.

6.2.2 Implementação das Redes Neurais

Para a implementação das redes neurais, foi escolhida a ferramenta TensorFlow Keras (CHOLLET et al., 2015), devido à sua ampla adoção e eficácia na construção de redes neurais profundas, além da flexibilidade que oferece para configurar arquiteturas complexas. O ambiente Google Colaboratory (RESEARCH, 2023) foi utilizado para a execução dos experimentos, proporcionando acesso a recursos de computação em nuvem e suporte a GPU, o que facilitou a experimentação iterativa.

Arquitetura MLP_like

Para a implementação da arquitetura *MLP_like* (*Perceptron* Multicamadas), foi necessário conduzir experimentos preliminares para definir os principais parâmetros da rede neural. Essas variações incluíram alterações no número de camadas escondidas, na quantidade de neurônios por camada, nas funções de ativação aplicadas, além da implementação de técnicas de regularização como o *Dropout*.

O processo de teste envolveu:

- **Camadas Escondidas**: Foram testadas diferentes quantidades de camadas escondidas, variando de uma a três, para avaliar o impacto da profundidade na capacidade da rede de capturar padrões complexos.
- **Neurônios por Camada**: O número de neurônios por camada foi ajustado entre 10 e 100, considerando a necessidade de equilibrar a capacidade de aprendizado da rede com a complexidade computacional.
- **Funções de Ativação**: As funções de ativação experimentadas incluíram *ReLU* (Rectified Linear Unit) e *tanh*, sendo que a função *ReLU* apresentou um desempenho superior na maioria dos testes, especialmente em camadas mais profundas, devido à sua capacidade de mitigar o problema de gradientes desaparecendo.
- **Dropout**: O uso de *Dropout* foi avaliado como uma técnica de regularização para prevenir o sobreajuste. Variações nos percentuais de *Dropout* (de 20% a 50%) foram testadas, com 40% e 30% sendo os valores mais eficazes para as primeiras e segundas camadas, respectivamente.
- **Batch Size e Algoritmo de Otimização**: O *batch size* foi definido como 16 após testes com valores que variaram entre 8 e 64, sendo que 16 ofereceu o melhor compromisso entre estabilidade de aprendizado e uso eficiente de memória. O algoritmo *Adam* foi escolhido pela sua eficácia em ajustar rapidamente os pesos da rede, com uma taxa de aprendizado de 0,001, conforme foi determinado por uma série de testes preliminares que compararam este com outros algoritmos, como *SGD* e *RMSProp*.

Com base nesses experimentos, a arquitetura final adotada foi composta por um modelo sequencial com:

- **Primeira Camada**: Totalmente conectada com 10 neurônios e função de ativação *ReLU*.
- **Primeira Camada de Dropout**: *Dropout* de 40%.
- **Segunda Camada**: Totalmente conectada com 20 neurônios e função de ativação *ReLU*.
- **Segunda Camada de Dropout**: *Dropout* de 30%.
- **Camada de Saída**: Um único neurônio com ativação sigmoide para realizar a classificação binária.

Essa configuração foi selecionada após considerar o equilíbrio entre desempenho e simplicidade, buscando minimizar o risco de sobreajuste enquanto maximiza a capacidade da rede em generalizar padrões de defeitos nos *statements* de código. O código que define esta arquitetura está disponível no Apêndice E.

Arquitetura Conv_1D

Para a definição da arquitetura *Conv_1D*, foram realizadas várias experimentações com redes neurais convolucionais unidimensionais, explorando diferentes configurações para maximizar a capacidade de aprendizado e a generalização da rede. As variações incluíram ajustes nos seguintes aspectos:

- **Camadas Convolucionais**: O número de camadas convolucionais foi variado entre uma e três camadas para investigar a profundidade ideal da rede.
- **Filtros e Tamanho dos Núcleos**: Foram testadas diferentes quantidades de filtros (de 8 a 64) e tamanhos de núcleos (variando de 3 a 7) para cada camada convolucional, buscando o equilíbrio entre a capacidade de extração de características e a eficiência computacional.
- **Funções de Ativação**: As funções de ativação testadas incluíram *ReLU*, *tanh* e *sigmoid*, com *ReLU* se destacando devido à sua eficiência em redes convolucionais, especialmente na mitigação de problemas relacionados à saturação dos gradientes.
- **Dropout e Average Pooling**: A implementação de *Dropout* foi crucial para prevenir o sobreajuste, com percentuais variando de 20% a 50%. Adicionalmente, o uso de *Average Pooling* foi testado em diferentes tamanhos de janela, com a configuração final utilizando uma janela de tamanho 2 para reduzir a dimensionalidade após a convolução.
- **Camadas do Classificador**: A quantidade de neurônios na camada totalmente conectada do classificador foi ajustada entre 10 e 50, considerando o trade-off entre a complexidade do modelo e o desempenho na tarefa de classificação.
- **Algoritmo de Otimização**: Após comparar vários algoritmos de otimização, como *Adam* e *RMSProp*, o algoritmo *Stochastic Gradient Descent* (SGD) foi selecionado para a configuração final, utilizando uma taxa de aprendizado de 0,001, *momentum* de 0,9 e taxa de decaimento de 1e-6. Essa escolha se deu pela estabilidade e desempenho do *SGD* em redes convolucionais, especialmente com os parâmetros citados.
- **Batch Size e Ajuste dos Pesos**: Foi adotado um *batch size* de 16 amostras por iteração, o que proporcionou um bom equilíbrio entre a convergência do modelo e a eficiência computacional.

A arquitetura final definida após esses experimentos consiste em:

- **Camada Convolucional 1D**: Com 8 filtros, núcleos de tamanho 3 e função de ativação *ReLU*.
- **Primeira Camada de Dropout**: *Dropout* de 40%.
- **Camada de Pooling**: *Average Pooling* com janelas de tamanho 2.

- **Camada de Achatamento (Flattening)**: Para preparação dos dados para o classificador.
- **Camada Totalmente Conectada**: 20 neurônios com função de ativação *ReLU*.
- **Segunda Camada de Dropout**: *Dropout* de 30%.
- **Camada de Saída**: Um único neurônio com função de ativação *sigmoide* para a tarefa de classificação binária.

O código que implementa essa arquitetura está disponível no Apêndice B, proporcionando detalhes sobre a configuração e execução do modelo.

6.2.3 Avaliação dos Modelos

Para avaliar o desempenho das redes, foi utilizada a métrica de acurácia, além de análises de matriz de confusão para lidar com o desbalanceamento dos dados. Também foi aplicada a métrica *ACC@N*, que mede a quantidade de acertos até a posição *N* no ranking dos *statements*. As arquiteturas foram treinadas utilizando validação cruzada com k-fold (k=5), e o critério de parada antecipada (early stopping) foi aplicado para evitar o sobreajuste.

6.3 Experimento 2 – Localização de Defeitos por Blocos de Código

O segundo experimento foi projetado para testar a abordagem de estimativa de propensão a defeitos em blocos de código, ao invés de *statements* individuais. A seguir, detalham-se os procedimentos utilizados para a construção dos conjuntos de dados e a implementação das arquiteturas de redes neurais convolucionais.

6.3.1 Construção do Conjunto de Dados

Para este experimento, foram selecionadas 53 versões do projeto *Lang*, conforme descrito na seção anterior. Os conjuntos de dados de treino foram construídos com base em dois critérios principais:

1. Blocos que englobam todos os *statements* pertencentes a um único método ou função.
2. Blocos de tamanho fixo, com *p statements* subsequentes, variando entre 8, 16, 32, 64 e 128 *statements*. Nos casos em que o bloco final possuía uma quantidade de *statements* inferior a *p*, completou-se o bloco com zeros.

Dessa forma, foram criados diferentes conjuntos de dados, permitindo a análise do desempenho da rede neural com blocos de variados tamanhos e diferentes níveis de granularidade.

6.3.2 Implementação das Redes Neurais Convolucionais 2D

A implementação das redes neurais convolucionais para este experimento foi realizada utilizando o framework *PyTorch* [1]. O *PyTorch* foi escolhido devido à sua flexibilidade em permitir personalizações avançadas nas arquiteturas de redes neurais, facilitando a integração de dados em diferentes formatos, como matrizes de cobertura e heurísticas.

Durante o desenvolvimento, diversos hiper parâmetros foram ajustados, incluindo o número e o tipo de camadas convolucionais, as funções de ativação, a quantidade de filtros convolucionais, bem como os métodos de sub-amostragem e algoritmos de otimização dos pesos sinápticos. Esse processo foi crucial para maximizar a acurácia e minimizar a perda do modelo, garantindo uma melhor generalização nos dados de teste.

6.3.3 Conv_cov – Arquitetura convolucional com filtros 2D para dados de cobertura

A arquitetura *Conv_cov* foi desenhada para processar uma matriz de cobertura como entrada, onde cada linha representa um *statement* e cada coluna corresponde a um caso de teste. A rede é composta por duas camadas convolucionais:

- A primeira camada possui 4 filtros com tamanho de núcleo 8.
- A segunda camada possui 16 filtros com tamanho de núcleo 5.

Ambas as camadas são seguidas por camadas de *Average Pooling* com filtros de tamanho 2x2 e utilizam a função de ativação *ReLU*. Após a propagação das características através das camadas convolucionais, os dados são linearizados e passam por duas camadas totalmente conectadas:

- A primeira camada possui 120 neurônios com função de ativação *ReLU*.
- A segunda camada possui 84 neurônios, também com função *ReLU*.

Finalmente, a camada de saída contém dois neurônios, utilizando a função *softmax* para classificação. O código que implementa essa arquitetura encontra-se no Apêndice D.

[1] <https://pytorch.org>

6.3.4 Conv_mul – Arquitetura convolucional com filtros 2D para dados de cobertura e heurísticas

A arquitetura *Conv_mul* é uma extensão da *Conv_cov*, projetada para incorporar, além da matriz de cobertura, uma matriz adicional contendo os valores de heurísticas calculados para cada *statement*.

Devido à diferença na quantidade de características extraídas das duas matrizes, foi necessário adicionar um passo intermediário para regularizar as saídas antes da combinação das características. Esse processo consiste em:

- Passar os dados da matriz de heurísticas por uma camada linear, ajustando a quantidade de características para igualar as extraídas nas convoluções.
- Concatenar os vetores de características resultantes das duas matrizes.

O classificador que se segue é similar ao da arquitetura *Conv_cov*, porém ajustado para lidar com a maior quantidade de características, o que exigiu modificações na quantidade de neurônios nas camadas e nas taxas de *dropout*. O código que define essa arquitetura está descrito no Apêndice C.

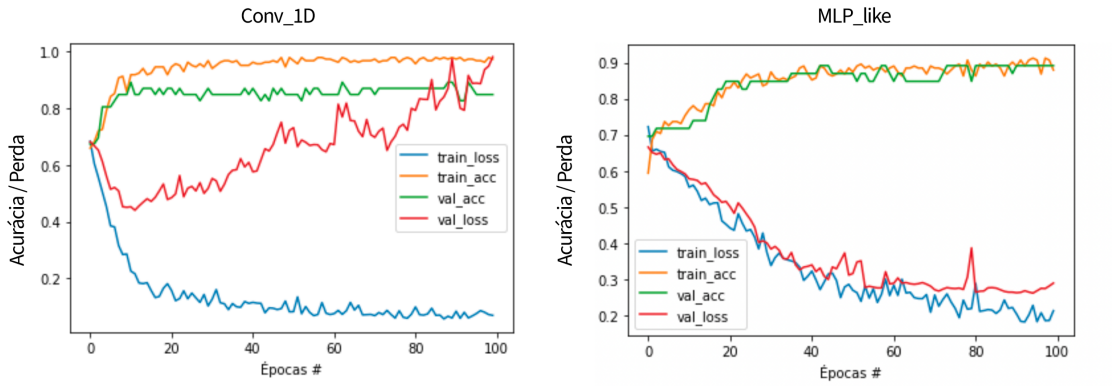
6.4 Resultados e Análises

6.4.1 Resultados do Experimento 1: Desempenho das Arquiteturas MLP e Conv1D

O primeiro aspecto a ser analisado é a comparação entre as duas arquiteturas de redes neurais propostas para o Experimento 1: a rede neural *Perceptron* Multicamadas (MLP) e a rede neural convolucional 1D (Conv1D). A Figura 6.1 apresenta as curvas de acurácia e perda para os dados de treinamento e validação, obtidas durante o treinamento dos modelos utilizando o conjunto de dados do projeto Lang.

Na análise dos resultados obtidos com a rede Conv1D, observa-se que os valores de perda (linha vermelha) diminuem durante as primeiras épocas de treinamento, mas, posteriormente, começam a aumentar, culminando em valores mais altos ao final das 100 épocas em comparação ao início. Paralelamente, a linha verde, representando a acurácia do modelo no conjunto de validação, inicialmente se eleva, indicando que o modelo está capturando padrões nos dados. No entanto, conforme a perda aumenta, a acurácia se estabiliza, sugerindo que o modelo não está conseguindo generalizar adequadamente para os dados de validação. Enquanto isso, os valores de perda nos dados de treinamento continuam a diminuir e a acurácia aumenta, estabilizando-se em um patamar mais elevado do que aquele observado na validação. Esse comportamento é indicativo de superajuste

Figura 6.1: Curvas de acurácia e perda das arquiteturas Conv_1D e MLP_like durante treinamento nos dados do projeto Lang



Fonte: Elaborada pelo autor.

(overfitting), situação em que a rede Conv1D demonstra dificuldades em generalizar o conhecimento adquirido nos dados de treinamento para os dados de validação.

Por outro lado, o comportamento da rede MLP_like, à direita na Figura 6.1, apresenta curvas de perda para treino e validação que diminuem de forma mais sincronizada, enquanto as curvas de acurácia mostram progresso constante. Esse padrão sugere que a arquitetura MLP_like está se comportando de maneira mais robusta e adequada ao cenário de dados utilizado. Embora o valor máximo de acurácia no conjunto de validação seja similar entre as duas arquiteturas, a rede MLP_like exibe maior estabilidade ao longo do treinamento.

Uma possível explicação para a diferença de desempenho entre as arquiteturas reside na característica da matriz de cobertura do projeto Lang, que apresenta alta esparsidade, ou seja, uma grande quantidade de *statements* que não são executados pelos casos de teste. Essa esparsidade, combinada com o fato de que as amostras de dados são representadas por vetores unidimensionais, pode dificultar a extração de características relevantes pela rede Conv1D, que é tradicionalmente aplicada a matrizes multidimensionais. Consequentemente, a entrada no classificador pode se tornar próxima a uma seleção aleatória de características, comprometendo o desempenho do modelo.

Em situações nas quais o desempenho do modelo se degrada ao longo das épocas, pode-se aplicar a técnica de parada antecipada para preservar os pesos antes dessa piora no conjunto de validação. Dessa forma, nos testes subsequentes deste experimento, não foi utilizada a última versão do modelo, mas sim o melhor *checkpoint* registrado durante o treinamento.

Ainda no contexto do Experimento 1, a Tabela 6.1 apresenta os resultados da métrica $ACC@N$, com valores de N definidos como $\{1, 3, 5, 15, 30, 50\}$, ao comparar os rankings gerados pela heurística Ochiai e pelo modelo de rede neural Conv_1D.

Como ilustrado na Tabela 6.1, a heurística Ochiai coloca um número maior de

Tabela 6.1: *ACC@N* da heurística Ochiai e do Conv_1D para os dados do projeto Lang

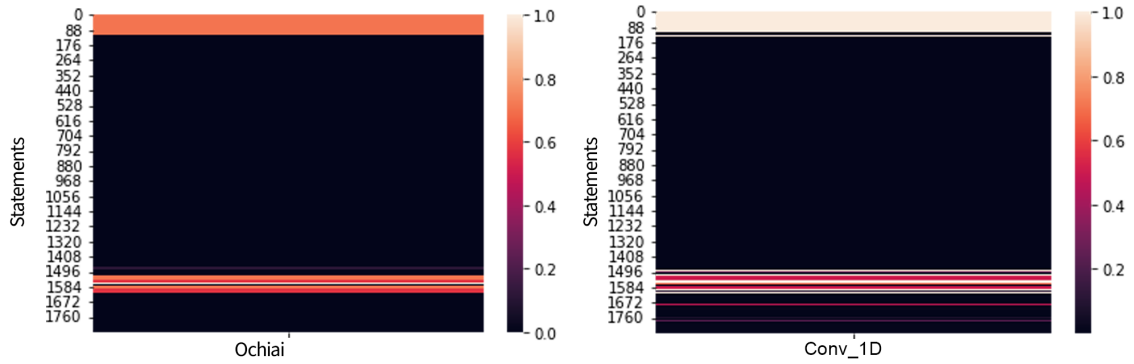
	ACC@1	ACC@3	ACC@5	ACC@15	ACC@30	ACC@50
Ochiai	**7**	**18**	**24**	**35**	**38**	40
Conv_1D	5	9	13	20	35	**41**

Fonte: Elaborada pelo autor.

statements nas primeiras posições quando analisadas as métricas *ACC@N* para todos $N < 50$. Por outro lado, o modelo Conv_1D apresenta desempenho superior apenas em ACC@50. O fato de o Conv_1D superar a heurística Ochiai em ACC@50 é de importância limitada, pois esse resultado decorre, em parte, da menor alocação de *statements* nas primeiras posições pelo modelo.

Para investigar as razões do desempenho inferior do Conv_1D no projeto Lang, foram realizadas análises dos mapas de calor gerados por ambas as técnicas, considerando os valores de propensão atribuídos a cada *statement* em versões defeituosas do código. A Figura 6.2 apresenta os mapas de calor de ambas as técnicas para a sétima versão do projeto Lang, que contém três *statements* defeituosos em um total de 1.846.

Figura 6.2: Mapas de calor dos valores de suspeita produzidos pela métrica Ochiai e pelo modelo Conv_1D para os *statements* da versão Lang-7



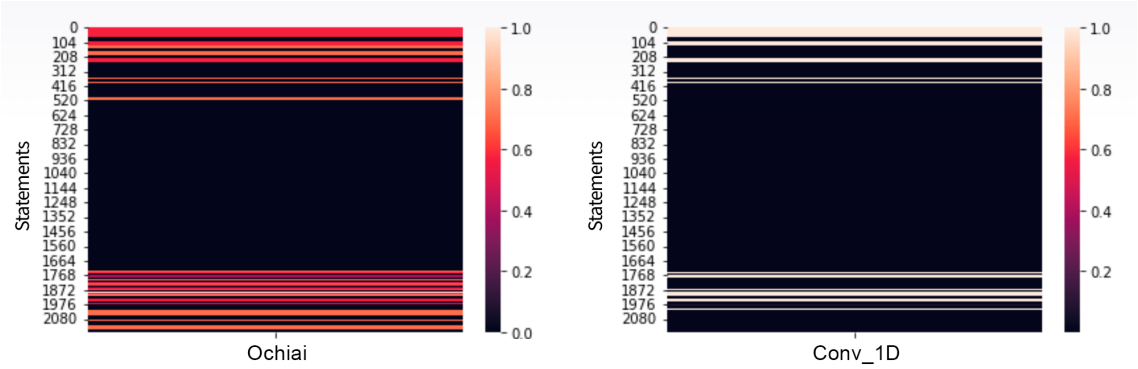
Fonte: Elaborada pelo autor.

Conforme observado, ambas as técnicas apresentam um comportamento semelhante ao atribuir valores de propensão próximos de zero para a maioria dos *statements*, conforme evidenciado pelas áreas mais escuras do mapa de calor. Os *statements* 1528, 1529 e 1624, que são identificados como defeituosos, receberam altos valores de suspeita tanto pela métrica Ochiai quanto pelo modelo, o que sugere que o modelo conseguiu capturar um padrão significativo na relação entre os dados de cobertura e os *statements* defeituosos. No entanto, ao contrário da métrica Ochiai, que atribui valores intermediários para os *statements* iniciais, o modelo destacou-os com valores próximos a 1. Como o modelo opera como um classificador binário de *statements* defeituosos e não defeituosos, esses altos valores indicam uma classificação incorreta de *statements* sem defeito como

defeituosos, resultando em falsos positivos.

Embora a classificação de *statements* sem defeito como defeituosos não seja necessariamente um problema — pois os *statements* defeituosos ainda podem ser corretamente identificados e posicionados nas primeiras colocações do ranking —, a Figura 6.3, que apresenta os mapas de calor para a versão 15, revela que o modelo atribuiu valores máximos para um número elevado de *statements* em alguns casos. Mesmo que os *statements* defeituosos estejam entre os destacados, a ocorrência de muitos empates pode dificultar sua posição elevada no ranking. A função de ativação sigmoide logística utilizada na arquitetura Conv_1D tende a saturar em valores próximos ao máximo da função, o que pode ser resultado da baixa quantidade de amostras ou da esparsidade dos dados. No projeto Lang, em média, 82% dos *statements* não são executados pelos casos de teste, resultando em uma matriz de cobertura com uma quantidade substancialmente maior de zeros do que de 1 ou -1. A escassez de valores não nulos (diferentes de zero) pode ser mais prejudicial para o modelo, que depende da aprendizagem a partir dos exemplos apresentados, enquanto a heurística Ochiai simplesmente realiza um cálculo sobre a matriz de cobertura da versão sob investigação.

Figura 6.3: Mapas de calor dos valores de suspeitas produzidas pela métrica Ochiai e pelo modelo Conv_1D para os *statements* da versão Lang-15.

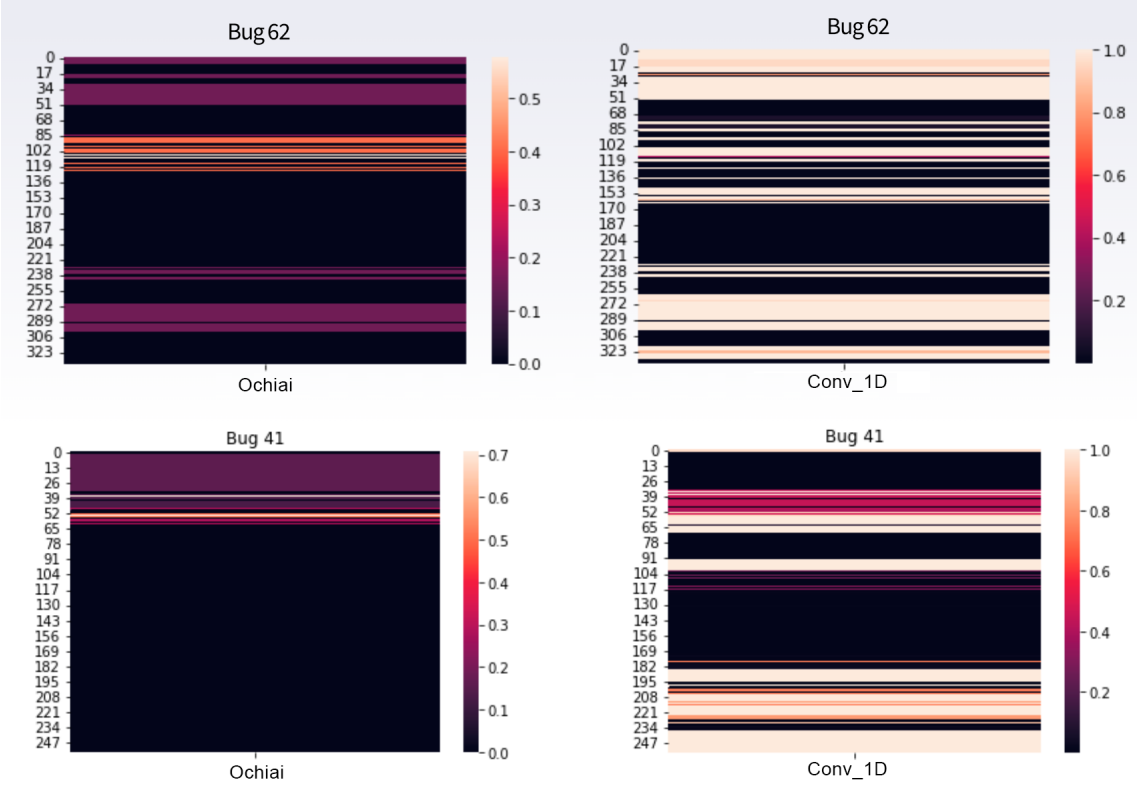


Fonte: Elaborada pelo autor.

Apesar de, na maioria das vezes, o padrão visual produzido pelas duas técnicas ser similar, foram observados cenários nos quais o modelo destaca aproximadamente 50% dos *statements* como sendo os defeituosos. Como trata-se de uma tarefa de classificação binária, tal comportamento assemelha-se a uma busca aleatória. A Figura 6.4 apresenta dois desses casos. Na figura, é possível notar que, no caso do *bug* 62, a heurística Ochiai concentra-se na região do *statement* 100, enquanto o modelo atribui um valor elevado para quase a metade dos *statements*. Ainda na mesma figura, comportamento semelhante ocorre para o *bug* 41, em que a métrica Ochiai concentra-se próximo ao *statement* 50, enquanto o modelo espalha valores altos em diversas regiões do mapa de calor.

Investigando-se as versões nas quais houve esse padrão visual discrepante entre

Figura 6.4: Mapas de calor dos valores de suspeitas produzidas pela métrica Ochiai e pelo modelo Conv_1D para os *statements* da versão Lang-62 e Lang-41, as quais possuem defeitos por omissão.



Fonte: Elaborada pelo autor.

as duas técnicas, percebeu-se que o fator comum entre elas é a presença dos chamados defeitos por omissão. Nesses casos, o *statement* responsável pelo defeito foi deletado (ou movido) durante a correção. Os dados utilizados nesta pesquisa foram coletados a partir do trabalho de Pearson et al. (2017), no qual são assumidas algumas estratégias para a marcação desse tipo de defeito. A regra geral é que o local do defeito passa a ser um local candidato que é representado por um *statement* executável antes ou depois do *statement* deletado. A consequência disso é que um *statement* contendo um padrão de cobertura que antes seria anotado como não defeituoso pode passar a representar um defeito. Isso é particularmente impactante para uma técnica baseada em aprendizado por meio de exemplos, pois uma amostra com características muito similares passa a ter uma classificação diversa do padrão recorrente.

Passando aos resultados obtidos para o projeto Mockito, a Tabela 6.2 apresenta os valores para a métrica $ACC@N$, considerando $n = \{1, 3, 5, 15, 30, 50\}$, quando comparados os rankings produzidos pela heurística Ochiai e pelo modelo da rede neural Conv_1D, considerando as 33 versões selecionadas do referido projeto.

Os dados apresentados na Tabela 6.2 indicam um desempenho ligeiramente superior do modelo Conv_1D em comparação à heurística Ochiai em algumas configurações

Tabela 6.2: *ACC@N* da heurística Ochiai e do Conv_1D para os dados do projeto Mockito

	ACC@1	**ACC@3**	**ACC@5**	**ACC@15**	**ACC@30**	**ACC@50**
Ochiai	**6**	10	13	17	19	20
Conv_1D	5	10	**14**	17	**20**	**23**

Fonte: Elaborada pelo autor.

de *ACC@N*. Especificamente, o modelo obteve vantagem em ACC@5 e ACC@30, com uma diferença de 1 acerto e uma vantagem de 3 no ACC@50. Todavia, como pode ser visto, a heurística Ochiai supera a Conv_1D para o caso do ACC@1.

O aspecto mais significativo desses resultados é a evidência de que a arquitetura Conv_1D se comportou de maneira mais eficaz com os dados do projeto Mockito em comparação ao projeto Lang. Para entender as possíveis razões por trás dessa diferença de desempenho, foram realizadas análises de estatística descritiva sobre três características principais: o número total de casos de teste, a quantidade de casos de teste que falharam, e a esparsidade dos dados, definida como o complemento da cobertura de código.

Tabela 6.3: Medidas de estatística descritiva sobre os dados de cobertura dos projetos Lang e Mockito, onde CT = Casos de Testes e CTN = Casos de Testes Negativos

	Lang (65 versões)				**Mockito (38 versões)**		
	CT	CTN	Esparsidade		CT	CTN	Esparsidade
média	93,77	1,90	0,82	média	671,05	3,16	0,72
desvio padrão	105,79	3,57	0,19	desvio padrão	433,89	4,51	0,06
mínimo.	5,0	1,0	0,08	mínimo.	17,0	1,0	0,61
25%	21,0	1,0	0,71	25%	152,25	1,0	0,68
50%	54,0	1,0	0,91	50%	764,0	1,5	0,72
75%	113,0	2,0	0,97	75%	1040,75	3,0	0,75
máximo.	423,0	27,0	0,99	máximo.	1311,0	26,0	0,86

Fonte: Elaborada pelo autor.

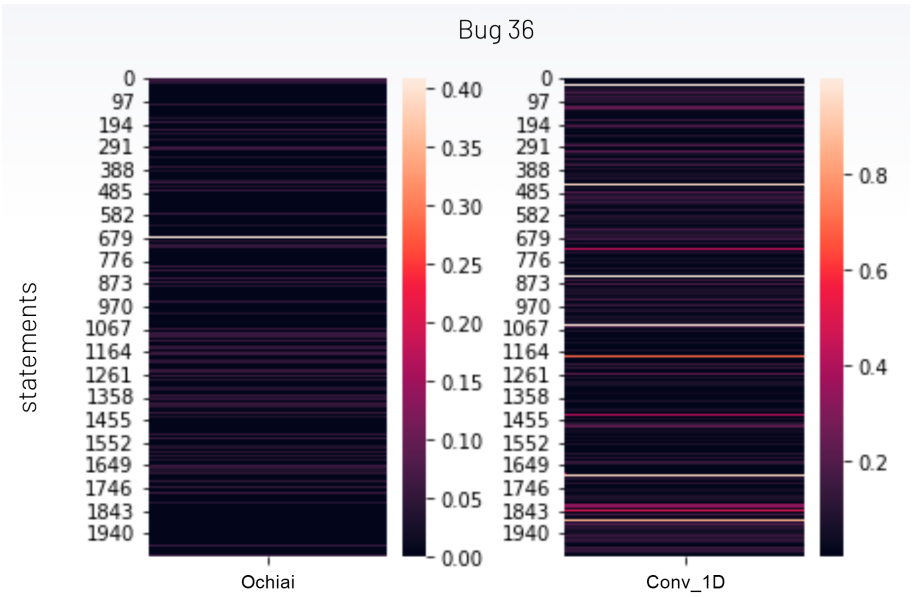
A Tabela 6.3 revela que, ao analisar as matrizes de cobertura de todas as versões do projeto Lang, a média de casos de teste é de 93,77 por versão, enquanto para o projeto Mockito, essa média é significativamente maior, chegando a 671,05 casos de teste por versão. Ao observar os valores de mediana (50% percentil na tabela), verifica-se que a média no caso do Lang está mais distante da mediana em comparação com o projeto Mockito, indicando uma distribuição mais consistente de valores no Mockito. Conforme discutido na Seção 5.1.1, um maior número de casos de teste implica em mais características a serem consideradas pelo modelo. No entanto, um aumento no número de características não necessariamente implica em uma melhor qualidade das informações para a extração de padrões precisos dos dados. Nesse contexto, é importante destacar que o projeto Lang apresenta uma esparsidade média de 82%, enquanto no Mockito a

esparsidade é de 72%, com valores máximos de 99% e 86%, respectivamente. Além disso, a média de esparsidade do Lang é impactada por uma ocorrência isolada de esparsidade mínima (8%), enquanto as medianas de 91% e 72%, respectivamente para Lang e Mockito, confirmam que a esparsidade é mais prevalente no projeto Lang, no qual o modelo obteve pior desempenho.

Outro aspecto analisado foi a quantidade de *statements* marcados como defeituosos por omissão em cada um dos projetos, considerando apenas as versões selecionadas. No projeto Lang, 103 *statements* de um total de 132 foram marcados como defeituosos por omissão, representando aproximadamente 78% dos defeitos identificados. No projeto Mockito, 58 de um total de 76 *statements* foram marcados como defeituosos por omissão, correspondendo a cerca de 76% do total de defeitos. A diferença de apenas dois pontos percentuais entre os projetos indica que ambos possuem uma alta incidência de defeitos por omissão, sugerindo que essa característica, por si só, não explica a diferença no desempenho do modelo entre os dois projetos.

Ao examinar a Figura 6.5, que apresenta os mapas de calor para a versão Mockito-36, onde o *statement* 1847 é marcado como defeituoso por omissão, nota-se que as linhas do mapa de calor gerado pelo modelo para esse projeto exibem maior variação de cores, sem a saturação de aproximadamente metade dos *statements* com a mesma cor, como observado no projeto Lang. Isso sugere que a maior quantidade de informações no projeto Mockito favorece a capacidade do modelo de diferenciar entre *statements*, mesmo em casos de defeitos por omissão.

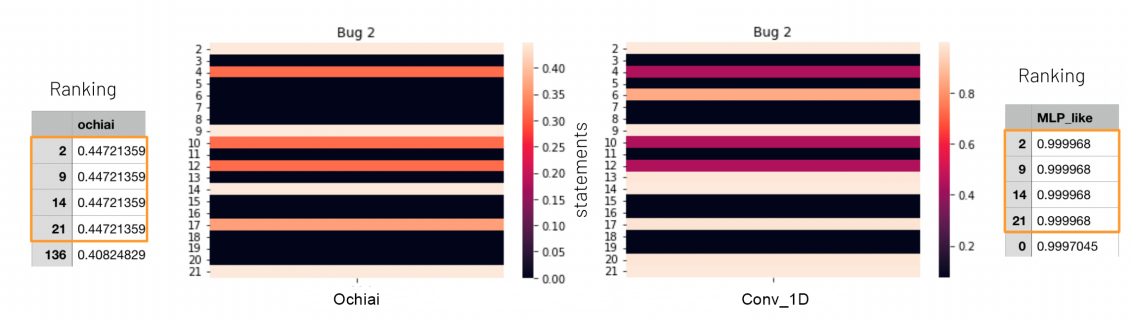
Figura 6.5: Mapas de calor dos valores de suspeitas da heurística Ochiai e do modelo Conv_1D para os *statements* da versão Mockito-36, que possui statement marcado como defeito por omissão



Fonte: Elaborada pelo autor.

Com relação aos resultados de *ACC@N*, em que o modelo da arquitetura Conv_1D superou a heurística Ochiai, uma análise mais detalhada dos valores de propensão atribuídos por ambas as técnicas para as versões que contribuíram para esse desempenho superior revela que as duas abordagens apresentam um comportamento semelhante nas primeiras posições do ranking. A Figura 6.6 ilustra esse ponto, apresentando os mapas de calor gerados pelas propensões das duas técnicas para a versão Mockito-2.

Figura 6.6: Mapas de calor dos valores de suspeitas produzidas pela métrica Ochiai e pelo modelo Conv_1D para os *statements* da versão Mockito-36, que possui statement marcado como defeito por omissão



Fonte: Elaborada pelo autor.

Como mostra a Figura 6.6, ambas as técnicas assinalaram os mesmos *statements* — 2, 9, 14 e 21 — como os primeiros no ranking. Dado que, em casos de posições empatadas, foi adotado um critério de desempate aleatório, é possível que a vantagem do modelo, que obteve duas unidades a mais no ACC@1 e uma unidade a mais no ACC@5, seja atribuída ao acaso. Com base nos dados disponíveis, não foram encontradas outras evidências que indiquem uma superioridade clara do modelo em relação à heurística.

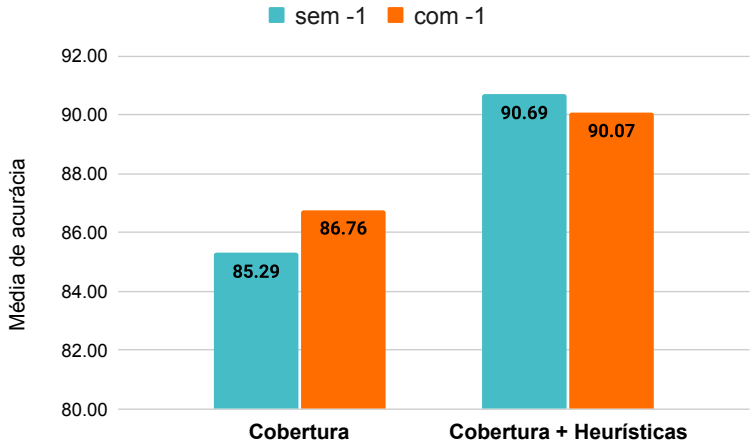
Embora os resultados não demonstrem uma superioridade significativa do modelo em comparação com a heurística, eles corroboram a ideia de que a arquitetura empregada é capaz de extrair padrões dos dados e produzir um desempenho comparável ao de uma boa heurística. Além disso, a abordagem utilizada oferece a vantagem de permitir a incorporação de outras informações que podem contribuir para o aumento da acurácia, como o tipo de *statements*, a similaridade entre *statements*, e a sequência de execução dos *statements* pelos casos de teste.

6.4.2 Resultados do Experimento 2: Análise de Propensão a Falhas por Blocos de Código

No Experimento 2a, focou-se na análise da propensão a falhas em blocos de *statements*, em contraste com a abordagem que considera *statements* individualmente. A Figura 6.7 apresenta a média dos valores de acurácia, obtida ao longo de três execuções, para diferentes configurações dos dados do projeto Lang. A configuração denominada

“Cobertura” inclui apenas os dados da matriz de cobertura e suas respectivas anotações, conforme discutido na seção 5.1.1. Já na configuração "Cobertura + Heurística", além das informações de cobertura, foram adicionadas matrizes com as suspeitas geradas pelas heurísticas Ochiai, Zoltar, Rogers & Tanimoto, e Sokal para cada *statement*. Ambas as configurações foram testadas com e sem a inclusão dos resultados dos casos de teste que exercitam os *statements*.

Figura 6.7: Média de acurácia para diferentes execuções, da abordagem blocos organizados com dados de métodos, com variação das informações nos conjuntos de dados, considerando o projeto Lang

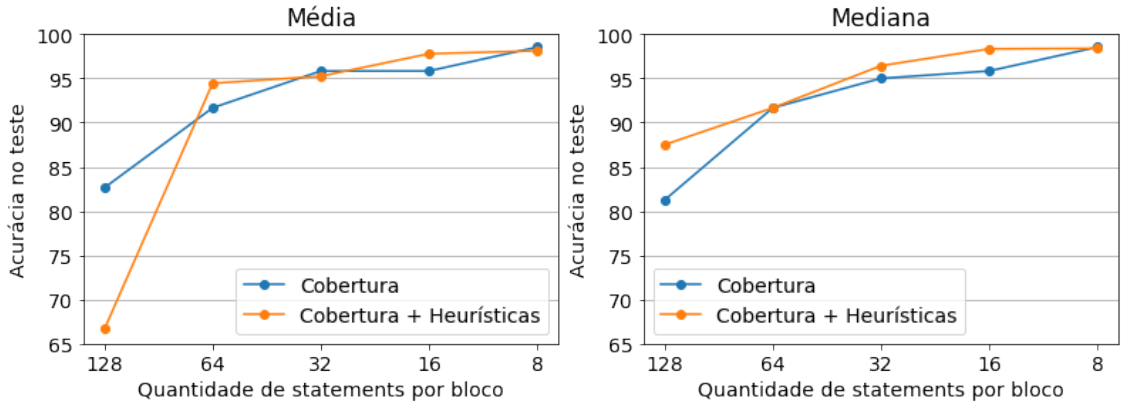


Fonte: Elaborada pelo autor.

Nos testes realizados com as amostras da configuração "Cobertura", a acurácia média aumentou de 85,29% para 86,76% com a inclusão da informação “-1” na matriz de cobertura, sugerindo que os resultados dos casos de teste ajudaram o modelo a identificar padrões entre a cobertura de blocos de *statements* e a presença de defeitos. Contudo, na configuração "Cobertura + Heurísticas", a acurácia média foi de 90,69% sem a informação do resultado do caso de teste, ligeiramente superior aos 90,07% obtidos quando essa informação foi incluída. Neste caso, a acurácia já elevada indica que a adição dos resultados dos casos de teste não trouxe melhorias significativas ao modelo. Isso ocorre porque as heurísticas, como Ochiai, já integram informações sumarizadas da matriz de cobertura, incluindo os resultados dos testes. Portanto, o principal fator que influenciou o desempenho do modelo na abordagem por blocos baseados em métodos foi a inclusão dos valores das heurísticas Ochiai, Zoltar, Rogers & Tanimoto, e Sokal.

No Experimento 2b, a Figura 6.8 apresenta as médias e medianas dos valores de acurácia no conjunto de teste, resultantes de três execuções de treinamento das arquiteturas CNN Conv_cov e Conv_mul. Esses resultados foram obtidos utilizando os *statements* do projeto Lang, considerando todas as versões selecionadas e diferentes tamanhos de blocos.

Figura 6.8: Gráficos média e mediana dos valores de acurácia para execuções da abordagem por blocos, considerando apenas matriz de cobertura ou matriz de cobertura mais valores de heurísticas, para variações no tamanho dos blocos extraídos do projeto Lang



Fonte: Elaborada pelo autor.

Os resultados indicam que o modelo treinado com dados de cobertura de fluxo de controle combinados com heurísticas é superior quando os blocos possuem 64 e 16 *statements*, enquanto se aproxima do modelo treinado apenas com cobertura nos tamanhos de bloco 32 e 8. No entanto, seu desempenho é significativamente inferior quando os blocos possuem 128 *statements*. Analisando a mediana da acurácia, o modelo com heurísticas é superior nos blocos de 128, 32, e 16 *statements*, equipara-se ao modelo com 64 *statements* e permanece próximo quando há 8 *statements* por bloco. Considerando que a mediana é menos sensível a valores extremos do que a média, esses resultados sugerem que a inclusão de heurísticas contribui positivamente na maioria dos testes, com desempenho da arquitetura que utiliza essas informações sendo igual ou superior àquela que utiliza apenas a matriz de cobertura.

Independentemente da inclusão dos dados das heurísticas, observa-se que os resultados de média e mediana para blocos contendo 64, 32, 16 e 8 *statements* superaram 90% de acurácia, sugerindo que a estratégia de dividir os blocos em tamanhos fixos foi mais eficaz para o projeto Lang do que o agrupamento dos blocos com base nos métodos. Em contraste, os resultados foram significativamente inferiores ao utilizar blocos de 128 *statements*. Adicionalmente, a tendência indica que o desempenho do modelo melhora conforme o tamanho dos blocos diminui, a ponto de, em uma das execuções, a rede neural atingir 100% de acurácia no conjunto de teste. Embora os dados discutidos se concentrem na métrica de acurácia, é relevante mencionar que no Apêndice A estão disponíveis os valores das métricas de acurácia, F1 e perda, tanto para os conjuntos de treino quanto de teste, em todas as configurações de conjuntos avaliadas.

6.5 Considerações Finais

Este capítulo apresentou a avaliação empírica de caráter preliminar e exploratório, das abordagens propostas no Capítulo 5, como forma de resolução do problema da localização de defeitos de software baseada na exploração dos dados de cobertura de fluxo de controle, pela aplicação de arquiteturas 1D e 2D de redes neurais convolucionais.

Via experimentação sobre dois dos projetos disponíveis no conjunto de dados Defects4J, foram avaliadas as abordagens de localização de defeitos em nível de *statement* e em nível de blocos de código. No primeiro caso, localização em nível de linhas de *statements*, os testes tanto com uma rede neural do tipo Perceptron Multicamadas, quanto uma Rede Convolucional 1D, demostraram que a modelagem adotada é suficiente para localizar defeitos de software no contexto da linguagem Java. Já para a localização de defeitos no nível de bloco de código, a exploração de variações no tamanho dos blocos de códigos, pela aplicação de CNN com arquitetura 2D foi capaz de relacionar dados de cobertura de fluxo de controle com a localização do defeito em blocos de código.

Do ponto de vista de aprendizado de máquina, os experimentos demonstraram que ambas as modelagens foram, em certa medida, capazes de relacionar as informações de cobertura com o local do defeito. Todavia, referente a métricas próprias de localização de defeitos, o desempenho das redes neurais foi inferior ou pouco competitivo em relação à heurística Ochiai, um método com custo computacional consideravelmente baixo. Assim, demonstrou-se que, apesar de explorar os princípios levantados no Capítulo 4, as propostas de uso de redes neurais convolucionais não representaram um avanço na resolução do problema em voga.

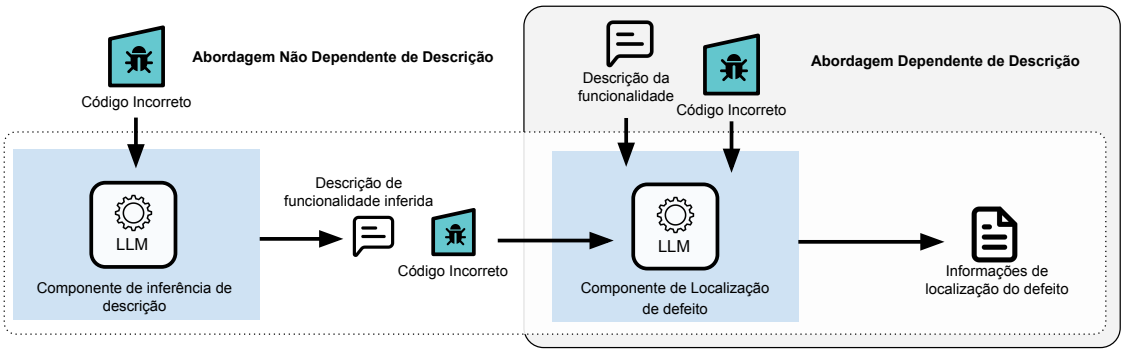
CAPÍTULO 7

Proposta de Localização de Defeitos de Software Baseada em LLM e Descrição de Funcionalidade

Em sintonia com a sistematização dos aspectos fundamentais para a localização automática de defeitos de software, apresentada no Capítulo 4, este capítulo introduz uma nova contribuição desta tese: uma técnica para a localização de defeitos em software baseada em Modelos de Linguagem de Grande Porte (LLMs). Esta proposta se alinha diretamente com os elementos sistematizados anteriormente, especialmente no que diz respeito à Caracterização da Informação, Representação dos Dados e Métodos de Combinação.

A abordagem proposta utiliza um LLM para enfrentar o desafio de identificar a provável localização de um defeito em uma código-fonte sabidamente incorreto. Como ilustrado na Figura 7.1, a proposta permite a implementação de duas abordagens distintas: a Abordagem Dependente de Descrição (ADD), delimitado pelas linhas contínuas, e a Abordagem Não Dependente de Descrição (ANDD), delineada pelas linhas pontilhadas.

Figura 7.1: Visão geral das abordagens dependente e não dependente de descrição.



Fonte: Elaborada pelo autor.

A abordagem ADD utiliza a descrição em linguagem natural da funcionalidade que um código-fonte defeituoso deveria implementar junto com o código incorreto como entrada, enquanto a NDDA é projetada para cenários em que essa descrição em linguagem

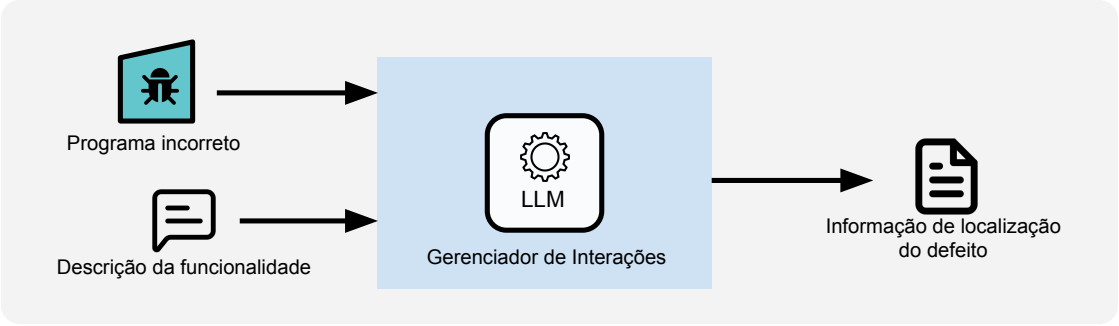
natural não esteja disponível. Nesse caso, um LLM é empregado para gerar uma descrição inferida da funcionalidade esperada, que é então utilizada para localizar o defeito.

Esta proposta não apenas exemplifica a aplicabilidade do *framework* sistematizado, mas também destaca a flexibilidade e a adaptabilidade da estrutura ao incorporar técnicas avançadas como os LLMs, reforçando a relevância e a contribuição desta tese para a área de localização automática de defeitos em software. A seguir, cada uma das abordagens é explicada em detalhes.

7.1 Abordagem Dependente de Descrição - ADD

A Abordagem Dependente de Descrição (ADD), resumida na Figura 7.2, alinha-se a sistematização proposta no Capítulo 4 ao utilizar um LLM para combinar diferentes informações a fim de realizar a localização do defeito em um código defeituoso. A execução desta abordagem começa com a submissão de um código incorreto, acompanhado de uma descrição em linguagem natural que detalha a funcionalidade esperada do código, ao componente baseado em LLM. Esse componente é responsável por gerenciar as entradas e coordenar as interações com o LLM instanciado. Naturalmente, a aplicabilidade dessa abordagem se dá em cenários nos quais exista a disponibilidade de uma descrição textual para a funcionalidade sob processo de depuração.

Figura 7.2: Abordagem Dependente de Descrição



Fonte: Elaborada pelo autor.

O Componente de Localização de Defeito, baseado em LLM, combina técnicas de engenharia de *prompt* com um *parser* de saída para produzir uma resposta em formato JSON, que inclui três variáveis principais:

- **buggyline**: o número da linha, indexado a partir de 1, onde o LLM identifica o defeito.
- **buggycode**: o trecho de código considerado como a causa do defeito.
- **explanation**: uma descrição do motivo pelo qual o trecho de código apontado em *buggycode* é identificado como o local do defeito.

Código 1: Exemplo de saídas produzidas pela abordagem ADD para duas versões incorretas da funcionalidade remoção de itens repetidos de uma lista.

```
[
    {
        "buggycode":"for number in list:",
        "buggyline":"2",
        "explanation":"The line is faulty because the variable
        ↪  'list' is not defined or passed as a parameter to the
        ↪  function. It should be 'for number in lst:' to iterate
        ↪  over the input list 'lst'.",
        "path":"question_3/code/wrong/wrong_3_006.py"
    },
    {
        "buggycode":"if lst not in new_lst:",
        "buggyline":"3",
        "explanation":"The line is faulty because it is checking if
        ↪  the entire list 'lst' is not in 'new_lst' instead of
        ↪  checking if the element 'i' is not in 'new_lst'. This
        ↪  will not effectively remove duplicates from the list.",
        "path":"question_3/code/wrong/wrong_3_057.py"
    },
]
```

No Código 1, pode ser visto um exemplo, recortado, de saída produzida pela abordagem ADD, para uma versão incorreta de um código Python. A adoção da captura das informações *buggyline* e *buggycode* tem por finalidade produzir um nível a mais de detalhamento para um uso prático durante o processo de depuração do código, uma vez que o usuário desenvolvedor poderá comparar as informações, ou mesmo facilitar a produção de *plug-ins* para IDEs. Além disso, como esta proposta está fundamentada no uso de LLMs de proposito geral, e não considerando a técnica de *fine-tuning*, o principal mecanismo de enviesamento de comportamento do modelo passa a ser o processo de engenharia de *prompt*. Sendo assim, a discriminação das informações de trecho de código considerado incorreto da informação de índice da linha onde o código se encontra pode contribuir para o racional do modelo. O conceito de engenharia de prompt tem como produto a construção do conjunto de instruções e dados de entrada que são submetidos ao LLM. O Capítulo 8 traz os detalhes de como esse conceito se implementa na prática dentro desta proposta.

Ainda no Código 1, considerando a linha 5, observa-se que a variável *explanation* apresenta, de fato, uma explicação sobre o motivo do “for number in list:” ter sido considerado, pela abordagem, uma linha defeituosa. De acordo com a saída, o comando *for* tenta realizar uma iteração sobre uma variável não definida. Na verdade, *list* em lin-

Código 2: Exemplo de código Python incorreto para a funcionalidade de remoção de itens duplicados em uma lista

```
1 def remove_extras(lst):
2     new_list = []
3     for number in list:
4        if number not in new_list:
5            new_list.append(number)
6     return new_list
```

guagem de programação Python é uma palavra reservada, que representa o construtor para um objeto do tipo lista. Olhando para o Código 2, a versão incorreta que fora analisada, vê-se que, de fato, a variável definida havia sido *new_list*. Ou seja, objetivamente, o código em questão acabaria por gerar um "TypeError" em tempo de interpretação.

Olhando para as informações de localização de defeito para a versão *wrong_3_057.py*, presentes entre as linhas 9 e 12, a saída da abordagem ADD, ao contrário do exemplo anterior, está apontando um defeito mais fortemente relacionado à lógica, erro que não seria de forma alguma capturado pelo interpretador Python.

A título de clareza, a variável *path* não agrega qualquer valor do ponto de vista da qualidade da proposta, motivo pelo qual não está listada como uma das informações fundamentais para a localização. A presença dessa informação no arquivo JSON é apenas por utilidade, como meio de viabilização de análises de logs e processamentos.

Em suma, mais do que a indicação apenas da suspeita de uma linha ou comando, como as técnicas tradicionais de localização de defeitos costumam abordar o problema, nesta proposta, o processo de localização de defeitos acaba por produzir conjuntos de informações que, na prática, tornam-se essenciais para diagnosticar a falha no código de forma precisa e eficiente.

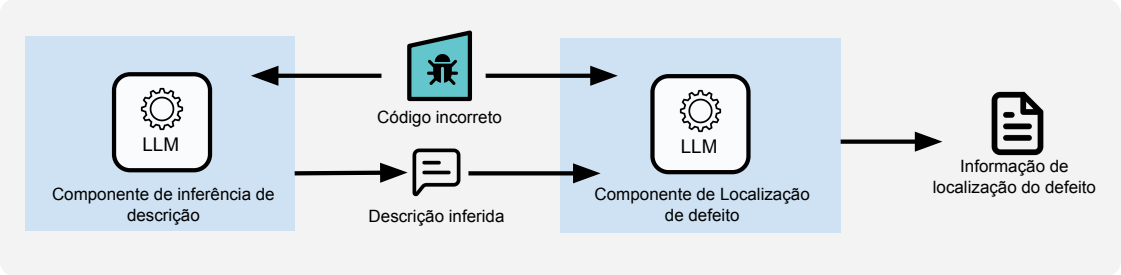
Ao integrar a ADD ao *framework* sistematizado, esta abordagem demonstra a flexibilidade e a capacidade adaptativa da estrutura ao incorporar tecnologias avançadas, como os LLMs, para resolver problemas práticos na localização de defeitos em *software*. A abordagem não só exemplifica a aplicação dos conceitos apresentados anteriormente, mas também reforça a importância de metodologias inovadoras e acessíveis em contextos de desenvolvimento de *software*.

7.2 Abordagem Não Dependente de Descrição - ANDD

A Abordagem Não Dependente de Descrição (ANDD) oferece uma solução para a localização de defeitos em cenários onde não há uma descrição em linguagem natural disponível para auxiliar no processo de localização do defeito. Neste contexto,

um LLM é utilizado inicialmente para a inferência da suposta funcionalidade, ou seja, a funcionalidade que o código deveria implementar se estivesse correto e, posteriormente, outro LLM é utilizado para a localização do defeito, superando assim a ausência de uma descrição explícita, a priori.

Figura 7.3: Abordagem Não Dependente de Descrição



Fonte: Elaborada pelo autor.

Conforme ilustrado na Figura 7.3, dado um código incorreto, a ANDD utiliza dois componentes baseados em LLM:

- **Componente de inferência de descrição:** infere a funcionalidade pretendida pelo código defeituoso e gera uma descrição correspondente. Este componente possui uma instância de LLM e um *prompt* especializado para garantir que a saída seja uma descrição mais próxima possível da funcionalidade esperada.
- **Componente de Localização de defeito:** a descrição gerada e o código defeituoso são passados para o segundo componente, que é idêntico ao da Abordagem Dependente de Descrição (ADD) e responsável por realizar a localização de defeitos.

No Código 3, tem-se exemplo de saída produzida por esta abordagem. Como pode ser visto nas linhas 3 e 10, existe uma variável "*description*", que é precisamente o resultado obtido pelo Componente de inferência de descrição. Nota-se que, para a mesma funcionalidade - ordenação de tuplas em uma lista - para duas versões distintas, um mesmo LLM pode produzir diferentes formas de descrição em linguagem natural. No caso das demais variáveis, são resultado do Componente de Localização de defeito, assim como demonstrado na Seção 7.1.

Desafios e Solução Proposta: A principal preocupação na ANDD é evitar que o LLM descreva a funcionalidade incorreta já presente no código, em vez de inferir a funcionalidade desejada. Para mitigar esse risco, foram realizados testes preliminares e empregadas técnicas de engenharia de *prompt* que guiam o LLM a produzir uma descrição da funcionalidade pretendida, ao invés de apenas reproduzir a lógica falha já presente no código. Essa solução demonstra a capacidade do LLM de extrapolar a partir do código fornecido e deduzir a funcionalidade correta que deveria estar implementada.

Código 3: Exemplo de saídas produzidas pela abordagem ANDD para duas versões incorretas da funcionalidade ordenação de tuplas em uma lista.

```
[
    {
      "description":"Sort a list of tuples based on the second
      ↪  element of each tuple in descending order.",
      "buggycode":"newlst.append(i)",
      "buggyline":"8",
      "explanation":"The line is faulty because it appends the
      ↪  variable 'i' outside the loop where 'i' is defined,
      ↪  leading to incorrect results. It should append 'maximum'
      ↪  instead to maintain the correct sorting order.",
      "path":"question_4/code/wrong/wrong_4_356.py"
   },
   {
      "description":"Sort a list of ages in descending order.",
      "buggycode":"if i > eldest:",
      "buggyline":"5",
      "explanation":"The fault is in line 5 where the comparison
      ↪  operator should be '<=' instead of '>' to correctly sort
      ↪  the list in descending order.",
      "path":"question_4/code/wrong/wrong_4_187.py"
   },
]
```

Modularidade e Flexibilidade: A abordagem ANDD foi projetada para ser adaptável, permitindo a substituição e avaliação de diferentes componentes quase independentes. Por exemplo, diferentes LLMs podem ser utilizados para a geração da descrição e para a localização, aproveitando as especializações de modelos distintos. Essa flexibilidade é particularmente valiosa, considerando a diversidade de LLMs disponíveis e suas diferentes capacidades, o que permite otimizar cada etapa do processo de acordo com as necessidades específicas da tarefa.

Ao integrar a ANDD ao *framework* sistematizado anteriormente, esta abordagem demonstra como a combinação de técnicas de inteligência artificial atuais com estratégias de modularidade pode oferecer soluções robustas e adaptáveis para a localização de defeitos em *software*. Essa proposta não apenas exemplifica a aplicação dos conceitos apresentados anteriormente, mas também contribui para a evolução das práticas de depuração em cenários com informações limitadas.

7.3 Estratégias de Geração de Rankings

O problema da localização de defeitos é tradicionalmente abordado pela definição de um valor de suspeição para cada elemento de código suspeito, a partir do qual é possível construir um ranking que ordene os elementos relevantes para a inspeção do desenvolvedor ou de um método automático de correção.

As abordagens ADD e ANDD, por padrão, possuem uma estratégia para identificar o elemento suspeito de causar o defeito, utilizando um LLM, porém limitando-se ao retorno de um único elemento. Essa abordagem pode dificultar a comparação com métodos que utilizam rankings parciais ou completos dos elementos de código suspeitos. Para superar essa limitação, foram propostas três estratégias para a construção de rankings de suspeita de linhas de código utilizando LLM. Assim, as abordagens de localização de defeitos baseadas em LLM, propostas neste trabalho, podem ser instanciadas para retornarem um único elemento de suspeito, ou implementarem uma das estratégias abaixo de geração de rankings de elementos a serem inspecionados:

7.3.1 Geração de Ranking por Resposta Única (fl-top-k)

Nesta estratégia, o LLM é instruído a responder uma lista de variáveis contendo as linhas de código consideradas suspeitas. A quantidade de variáveis na resposta corresponde ao parâmetro k, equivalente ao tamanho do ranking que se deseja obter. Sendo assim, no limite, tem-se que $0 < k < P$, sendo P a quantidade de elementos de código que podem ser inspecionados. Essa estratégia apresenta pouco impacto em relação ao tráfego de tokens de entrada no LLM, uma vez que o código-fonte e a descrição da funcionalidade são apresentados uma única vez para o modelo. Entretanto, na prática, para que o modelo tenha a capacidade de preencher e responder as k variáveis que representam o ranking de elementos suspeitos, é necessário um processo de engenharia de *prompt*. A partir de testes preliminares, percebeu-se que o LLM utilizado possuía um comportamento mais estável quando se explicitavam cada uma das k variáveis que deveriam ser preenchidas ao invés simplesmente solicitar a lista ordenada de elementos suspeitos. Dessa forma, optou por uma abordagem em que cada posição do ranking é explicitamente solicitada ao modelo, da seguinte forma:

a) Definem-se k variáveis de saída, cujos nomes incluem um sufixo correspondente à posição no ranking, de 1 a k;
b) Indica-se que a k-ésima variável deve ser preenchida com o código da linha considerada a mais suspeita, conforme a ordem correspondente a k. Por exemplo, para a variável buggyline_2, o LLM é instruído a indicar a segunda linha mais suspeita;

c) Apresenta-se o código com defeito e orienta-se o LLM sobre os passos necessários para desenvolver o raciocínio.

Como se percebe, apesar de ter um custo baixo em relação à informação de entrada, essa estratégia possui um limitação de escala, uma vez que para se aumentar o tamanho do ranking é necessário que se explicite cada nova posição. Outra característica a ser destacada é que, como tanto a entrada quanto as variáveis de saída são conhecidas uma única vez, o modelo tem uma única chance de construir o racional para montar o ranking.

7.3.2 Geração de Ranking por Sucessivas Iterações (fl-it-k)

Nessa estratégia, quando solicitado, o LLM devolve um único elemento como a causa do defeito. Assim, para a construção do ranking, são realizadas k sucessivas chamadas ao modelo até que se tenha um ranking de tamanho k. Novamente, $0 < k < P$, sendo P a quantidade de elementos de código que devem ser inspecionados. Para cada uma das iterações a partir da segunda, o LLM recebe como entrada não apenas o código-fonte e eventualmente a descrição da funcionalidade, mas também recebe a lista de linhas que já foram apontadas como suspeitas nas iterações anteriores. Assim, o procedimento prático para essa estratégia de geração de ranking pode ser resumido como:

a) Define-se uma variável de saída contendo o elemento de código considerado responsável pelo defeito;
b) Apresenta-se o código com defeito e a lista de elementos já considerados suspeitos nas iterações anteriores — na primeira iteração, essa lista é vazia — e instrui-se o modelo sobre as respostas anteriores;
c) O procedimento é repetido até que se obtenha um ranking de tamanho k.

Essa estratégia, em tese, é flexível em relação ao aumento do ranking, uma vez que para aumentar a quantidade de elementos apontados como suspeitos, basta escalar para cima a quantidade de iterações realizadas. Todavia, diferentemente da estratégia *fl-top-k*, a *fl-it-k* é sensível em relação à entrada, isto é, o consumo de *tokens* de entrada do modelo tende a crescer linearmente em relação a k.

7.3.3 Geração de Ranking por Comitê de Múltiplos Pontos de Partida (fl-ms)

A última estratégia para a geração de rankings de suspeitas utilizando LLM combina dois conceitos amplamente utilizados em diferentes áreas da computação: comitês

de modelos e busca por múltiplos pontos de partida. No contexto do aprendizado de máquina, a técnica de comitês é comumente utilizada para reduzir a variância nas respostas dos modelos. Em vez de considerar a resposta de um único modelo, combinam-se diferentes respostas para que se produza uma resposta única, presumidamente, mais robusta.

Além disso, em problemas de otimização, a busca a partir de diferentes regiões do espaço de soluções pode ajudar a evitar mínimos locais, permitindo uma exploração mais eficiente. Dado que os LLMs são suscetíveis a fornecer respostas variadas para a mesma entrada, essa estratégia se apresenta como uma abordagem híbrida que combina tarefas de IA generativa e busca combinatória.

Assim, a estratégia *fl-ms* envolve a geração de um ranking a partir da combinação das respostas de dois ou mais LLMs. Isso pode ser feito por meio de diferentes pontos de partida, utilizando a mesma estratégia de geração de ranking (por exemplo, várias execuções da estratégia *fl-top-k*) ou combinando respostas de diferentes estratégias (como *fl-top-k* e *fl-it-k*). O procedimento para obter o ranking *fl-ms* é o seguinte:

a) Dado um código *c* com defeito, obtém-se o ranking de elementos suspeitos utilizando a estratégia A;
b) Para o mesmo código *c*, obtém-se o ranking de suspeitas gerado pela estratégia B;
c) Repete-se o processo até que todos os rankings das estratégias participantes sejam obtidos;
d) Combina-se as respostas das estratégias utilizando uma votação simples ponderada pela posição no ranking.

Diferentemente das estratégias anteriores, que possuem um tamanho fixo de ranking definido pelo parâmetro *k*, nesta abordagem o tamanho do ranking pode variar entre o maior *k* das estratégias participantes e a soma de todos os *k* das estratégias envolvidas, subtraída a dimensão a intersecção entre os rankings originais.

7.4 Caracterização da Proposta em Relação aos Aspectos-Chaves da Localização de Defeitos de Forma Automática

Nesta seção, a proposta de localização de defeitos de software baseada em Modelos de Linguagem de Grande Porte (LLM) é posicionada dentro dos aspectos-chaves identificados para a localização automática de defeitos, conforme discutido no Capítulo 4. Para tanto, avalia-se como a abordagem proposta se relaciona com os aspectos de caracterização da informação, representação dos dados e métodos de combinação.

Essa análise é crucial para entender a aplicabilidade e a eficácia da técnica em diferentes cenários e para destacar as vantagens e limitações do uso de LLMs na tarefa de localização de defeitos em código de *software*.

A Tabela 7.1 resume essa caracterização, facilitando a comparação entre os elementos teóricos discutidos e a implementação prática da proposta deste capítulo.

Tabela 7.1: Caracterização da proposta de localização de defeitos baseada em LLM segundo os aspectos-chaves

Aspecto-chave	**Classificação dos Atributos**	
Caracterização da informação	***Descrição da funcionalidade***	***Código-fonte***
Disponibilidade:	Média (Dependente da documentação)	Alta
Custo de aquisição:	Baixo	Baixo
Origem:	Externa ao código	O próprio código
Natureza:	Estática	Estática
Tipo:	Documentação em linguagem natural	Código-fonte
Representação dos dados	***Descrição em texto plano***	***Código-fonte em texto plano***
Consistência:	Alta (Preserva os detalhes)	Alta (Preserva os detalhes)
Compatibilidade:	Total (Uso direto no LLM)	Total (Uso direto no LLM)
Escalabilidade:	Médio (Limitada à capacidade do LLM)	Médio (Limitada à capacidade do LLM)
Métodos de combinação	***Modelos de Linguagem de Larga Escala - LLM***	
Custo:	Alto (Custo superior ao do programa original)	
Flexibilidade:	Alto (Usa as entradas sem transformações)	
Escalabilidade:	Baixa (Limitado pelo tamanho da entrada)	

Fonte: Elaborada pelo autor.

7.5 Considerações Finais

Neste capítulo, foi introduzida uma proposta para a localização de defeitos em software, baseada na utilização de Modelos de Linguagem de Grande Porte (LLMs). A abordagem proposta foi detalhada em duas abordagens: a Abordagem Dependente de Descrição (ADD) e a Abordagem Não Dependente de Descrição (ANDD). Ambas as abordagens demonstram a aplicabilidade dos LLMs na identificação de defeitos em código-fonte, seja a partir de uma descrição explícita da funcionalidade esperada (ADD) ou mediante a inferência dessa descrição diretamente do código defeituoso (ANDD). Enquanto a ADD se beneficia de informações contextuais sobre a funcionalidade, a ANDD oferece maior flexibilidade em cenários nos quais tais informações não estão disponíveis, ampliando, assim, o escopo de aplicação da proposta.

Além da definição padrão, que retorna uma única resposta sobre o local mais propenso ao defeito, a proposta introduz diferentes estratégias de interação com LLMs, para a geração de rankings de suspeitas. Dessa forma, as abordagens propostas aumentam a aplicabilidade em cenários reais bem como podem ser mais facilmente comparadas com outras técnicas da literatura.

Conforme indicado no Capítulo 3, dos trabalhos relacionados, as pesquisas de Kang, An e Yoo (2024), denominada AutoFL, e Yang et al. (2024), nomeada LLMAO, são exemplos de abordagens recentes que aplicam com sucesso LLM para a localização de defeitos e, consequentemente, são as principais bases de inspiração e comparação para as abordagens deste trabalho.

A proposta de utilização de LLM para localização de defeitos introduzida neste trabalho se assemelha à abordagem AutoFL no que diz respeito ao emprego de modelos de propósito geral, amplamente utilizados para variadas tarefas que extrapolam os limites da área de PLN e na possibilidade de uso LLM para explicar o motivo do defeito. Contudo, a proposta deste trabalho intenta realizar a localização de defeitos em nível de linhas de código, enquanto o AutoFL localiza defeitos em nível de método, ou função.

Por outro lado, assim como na proposta deste trabalho, a abordagem LLMAO realiza a localização de defeitos, com emprego de LLM, em nível de linhas de código. No entanto, este trabalho propõe o uso de LLMs gerais, e aplica *prompt engineering* para refinar o comportamento do modelo, o LLMAO utiliza um LLM base e acrescenta camadas transformes bidirecionais no topo do modelo base para que seja realizado um processo de *fine-tuning*, utilizando-se dados do histórico de programas e defeitos.

CAPÍTULO 8

Avaliação Experimental das Abordagens Baseadas em LLM para Localização de Defeitos

Neste capítulo, descreve-se o projeto experimental desenvolvido para avaliar as abordagens de localização de defeitos em software baseadas em Modelos de Linguagem de Grande Porte (LLMs). A estrutura do capítulo inclui os detalhes das ferramentas utilizadas para possibilitar a implementação dos LLMs testados, a descrição dos componentes propostos por cada abordagem, os LLMs utilizados, a apresentação e caracterização do conjunto de dados (*dataset*) empregado no experimento, e a explicação da estratégia de avaliação da localização de defeitos, com considerações sobre as questões de implementação. São apresentados os *prompt templates* utilizados, seguidos dos resultados obtidos e suas respectivas análises detalhadas para cada abordagem investigada. Por fim, é realizada uma comparação entre a abordagem baseada em CNN e uma das estratégias baseadas em LLM, leva à uma breve investigação sobre a problemática de vazamento de dados que pode contaminar a avaliação de técnicas compostas em LLMs.

8.1 Metodologia e Projeto Experimental

As subseções a seguir descrevem a metodologia aplicada para a avaliação das abordagens baseadas em LLM, apresentadas no Capítulo 7. O experimento foi projetado para responder às seguintes questões de pesquisa:

RQ1 Quão eficazes são os LLMs na localização de defeitos em código-fonte?

RQ2 Os LLMs são capazes de localizar defeitos mesmo sem uma descrição prévia da funcionalidade?

8.1.1 Ferramentas Utilizadas e Modelos de Linguagem de Grande Porte (LLMs)

Para a implementação das abordagens Dependente de Descrição (ADD) e Não Dependente de Descrição (ANDD), foi utilizado o framework LangChain (LANG-

CHAIN, 2024). LangChain é um framework de código aberto amplamente utilizado para o desenvolvimento de aplicações baseadas em Modelos de Linguagem de Grande Porte (LLMs). Ele oferece uma série de blocos de construção e integrações com serviços externos, incluindo diferentes provedores de LLMs, permitindo a construção flexível de fluxos de processamento de dados complexos.

Os modelos de linguagem utilizados neste experimento foram o GPT-3.5-turbo e o Llama 3 70B. O GPT-3.5-turbo, desenvolvido pela OpenAI[1], é um modelo proprietário, acessível via API, com custo por token (OPENAI, 2023a). Já o Llama 3 70B, disponibilizado pela Meta[2], é um LLM de código aberto que pode ser encontrado em diversas plataformas de nuvem, como Hugging Face e Groq (TOUVRON et al., 2024). Em ambos os casos, a interação com os LLMs foi realizada por chamadas de API, utilizando-se *scripts* Python e aproveitando-se os recursos fornecidos pelo *LangChain* para construir todos os fluxos de processamento necessários à experimentação.

8.1.2 Conjunto de Dados Utilizado

O conjunto de dados utilizado neste estudo é composto por 1.783 programas incorretos, implementados por estudantes de graduação para resolver cinco problemas propostos em um curso de programação em Python. Este conjunto de dados foi inicialmente introduzido por Hu et al. (2019) e, desde então, é utilizado pela comunidade de reparo automatizado de programas (REPAIR, 2024). A Tabela 8.1 resume as principais informações relevantes sobre o conjunto de dados utilizado neste experimento.

A coluna “Descrição” na Tabela 8.1 apresenta as descrições exatas fornecidas aos estudantes ao implementarem as funcionalidades. Nota-se que há uma variação na forma como os problemas são descritos; enquanto o primeiro problema possui uma descrição sucinta, os Problemas 3 e 4 incluem até mesmo exemplos detalhados. Na coluna “Programas”, é indicado o número de versões que os estudantes submeteram e que falharam em pelo menos um caso de teste, sendo essas as versões de programas utilizadas neste estudo. Finalmente, a última coluna apresenta o número de casos de teste disponíveis para avaliar a correção de qualquer versão para cada problema. Cabe destacar que este estudo não inclui os programas relacionados ao Problema 2, pois eles envolvem implementações com múltiplas funções Python em um único arquivo .py. Esse cenário está fora do escopo da proposta sob avaliação, mas pode vir a ser considerado em versões futuras, possivelmente mediante pré-processamento dos arquivos para separar as funções e uma adequação da suíte de testes para contemplar as funcionalidades individualmente.

[1] <https://openai.com>
[2] <https://meta.com/>

Tabela 8.1: Informações do conjunto de dados utilizados na avaliação empírica

Problema	Descrição	Programas	Casos de Teste
Question 1	Task: Sequential Search	575	11
Question 3	Task: Duplicate elimination Write a function remove_extras(lst) that takes in a list and returns a new list with all repeated occurrences of any element removed. For example, remove_extras([5, 2, 1, 2, 3]) returns the list [5, 2, 1, 3].	308	6
Question 4	Task: Sorting Tuples Can we sort items other than integers? For this question, you will be sorting tuples! We represent a person using a tuple (<gender>, <age>). Given a list of people, write a function sort_age that sorts the people and return a list in an order such that the older people are at the front of the list. An example of the list of people is [("M", 23), ("F", 19), ("M", 30)]. The sorted list would look like [("M", 30), ("M", 23), ("F", 19)]. You may assume that no two members in the list of people are of the same age.	357	6
Question 5	Task: Top-K Write a function top_k that accepts a list of integers as the input and returns the greatest k number of values as a list, with its elements sorted in descending order. You may use any sorting algorithm you wish, but you are not allowed to use sort and sorted.	108	5

Fonte: Elaborada pelo autor.

8.1.3 Estratégia de Avaliação da Localização de Defeitos

Para avaliar a qualidade de um sistema automatizado de localização de defeitos, é fundamental identificar precisamente onde o defeito está localizado no código. No entanto, o conjunto de dados utilizado neste estudo não fornece essa informação diretamente; os programas apresentam defeitos, ou seja, não exibem o comportamento esperado, mas as linhas de código incorretas não estão identificadas. Em situações como essa, uma abordagem comum para simular o local do defeito é comparar a versão defeituosa com uma implementação corrigida, identificando as alterações realizadas na versão corrigida e atribuindo as linhas modificadas como o local do defeito, conforme sugerido por Just, Jalali e Ernst (2014).

Embora o conjunto de dados original contenha uma implementação de referência, nem todas as versões defeituosas possuem uma correspondente versão corrigida. Diante disso, o seguinte procedimento foi adotado:

a) **Verificação das Correções Candidatas:** Para cada programa defeituoso, verificar se a estratégia de geração de correções candidatas conseguiu produzir uma versão corrigida do programa (essa estratégia será detalhada na próxima seção);
b) **Geração de *Diffs*:** Gerar *diffs* entre cada versão defeituosa e sua respectiva versão corrigida;
c) **Avaliação da Localização:** Verificar se o trecho de código identificado pelo LLM como sendo responsável pelo defeito corresponde a uma alteração no *diff*. Se houver correspondência, considera-se que o LLM localizou o defeito corretamente.

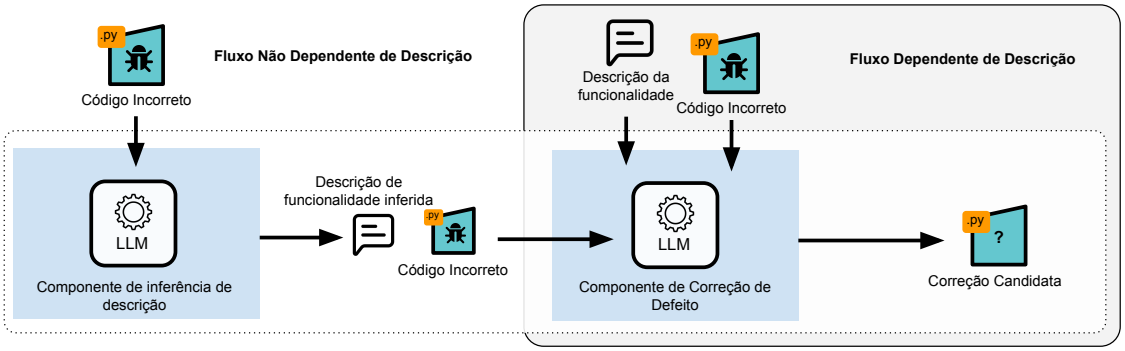
Uma consequência desse procedimento é que o número de programas com

defeitos elegíveis para avaliação da localização de defeitos pode ser inferior ao número total de programas com defeitos, pois alguns deles podem não ter uma versão corrigida disponível.

8.1.4 Processo de Geração e Avaliação das Correções Candidatas

Como mencionado anteriormente, uma das formas de se criar as anotações dos locais de defeito em códigos incorretos é pela comparação da versão defeituosa com a versão que corrigiu o defeito. Assim, neste experimento, foi utilizada uma adaptação da proposta de localização de defeitos para produção de sugestões de correções para cada um dos códigos incorretos presentes no conjunto de dados. Essa adaptação pode ser vista na Figura 8.1

Figura 8.1: Estratégia de geração de correções candidatas para anotação de local de defeito



Fonte: Elaborada pelo autor.

A estratégia de geração de correções candidatas para códigos defeituosos baseia-se na mesma estrutura de fluxo dependente de descrição (linhas contínuas com fundo cinza) e não dependente de descrição (linhas pontilhadas), conforme apresentado anteriormente na tarefa de localização de defeitos. No entanto, o componente denominado anteriormente de Componente de Localização de Defeitos é substituído, nesta adaptação, pelo Componente de Correção de Defeito. A principal mudança nessa adaptação reside no *prompt*, que pode ser visto no Código 4, o qual determina o comportamento do LLM instanciado em cada execução.

Com a estratégia de geração de correções candidatas definida, foi aplicado um fluxo de correção a cada versão de código incorreto presente no conjunto de dados original, variando-se o LLM instanciado — GPT ou Llama 3 — e a presença ou ausência de uma descrição prévia da funcionalidade. Dessa forma, para cada versão defeituosa, no melhor dos cenários, foram geradas até quatro opções de correção. Para que uma versão fosse considerada corrigida e, consequentemente, incluída no repositório de versões elegíveis para o processo de anotação de locais de defeito, conforme explicado na seção anterior, o seguinte procedimento foi adotado:

a) **Obtenção da Saída:** Capturar a saída produzida pelo fluxo de geração selecionado;
b) **Transformação em Código Executável:** Converter a saída em um programa Python executável;
c) **Execução dos Casos de Teste:** Executar o programa gerado em todos os casos de teste relacionados ao problema específico;
d) **Validação da Correção:** Se nenhum teste falhar, a correção é considerada válida, e a versão é adicionada ao repositório de correções alternativas para o respectivo defeito. Caso contrário, a correção candidata é descartada.

Para fins de análise, as versões anotadas, ou seja, as versões incorretas para as

Código 4: *Prompt template* utilizado para a obtenção da descrição de funcionalidade de um código defeituoso.

```
template_description_task = """
You are a Software Engineering Specialist with extensive experience in understanding and describing code functionality. Your expertise allows you to infer the intended functionality of code snippets based on their structure and logic.
Task:
You are given a faulty but valid Python code. Your task is to infer and describe the intended functionality that the code was supposed to implement.

Output Format:
Output will be used in a function call so do not add break lines between the fields.
All following fields are text fields.
Try to limit description to one sentence.
Add a Task Title in the beginning of the description.
description: "Concise explanation of the intended functionality of the code."

Instructions:
Be careful with break line commands and just use it when necessary.
Be aware of the output format.
Only describe what the correct code should do.
Do not explain why the code is faulty.
Description will be used to other developers to write new code.
It is not necessary ask for any usage example.

Code:
{input}
"""
```

quais existe pelo menos uma correção disponível, foram divididas em dois grupos: 1) versões corrigidas utilizando ou não a descrição da funcionalidade, separadas pelo fluxo de geração, independentemente do LLM instanciado; e 2) versões corrigidas independentemente do fluxo de geração de correções e do LLM utilizado.

Como a geração de correções candidatas neste experimento influencia diretamente a avaliação da localização de defeitos, foi realizada uma análise detalhada do desempenho do método de geração de correções candidatas, disponível no Apêndice G.

8.1.5 Desafios de Implementação e Estratégias de Mitigação

Uma parte crucial da proposta apresentada no Capítulo 7 e da experimentação subsequente foi a integração dos componentes baseados em LLM, que variam conforme a abordagem — dependente ou não de descrição — e os processos necessários para avaliar tanto os fluxos de geração de correções candidatas quanto a localização de defeitos. Como mencionado anteriormente, o framework LangChain foi utilizado para construir diversas partes da proposta, com todas as funcionalidades auxiliares implementadas integralmente em scripts Python.

Um aspecto importante da implementação dos componentes baseados em LLM, em qualquer uma das abordagens, foi o tratamento das respostas produzidas pelas chamadas à API do LLM instanciado. Frequentemente, o texto gerado pelo modelo causava falhas nos mecanismos padrão de análise de retornos disponíveis no LangChain ou resultava em erros internos do serviço de API. Para mitigar esses problemas, as seguintes estratégias foram implementadas:

- Definição de um parâmetro de tentativas de invocação da API em caso de falha. Nesse experimento, o valor adotado foi cinco;
- Definição de um *timeout* de 5 segundos para cada chamada de API;
- Refinamento do *prompt* utilizado em cada componente para que explicitamente o LLM instanciado lide com casos particulares, como por exemplo, evitar o retorno de “\n” dentro do código Python.

No fluxo de reparo, durante a geração de correções candidatas, tanto na abordagem ADD quanto na ANDD, após o componente baseado em LLM gerar o candidato à correção, é necessário executar esse candidato (código Python) contra todos os casos de teste disponíveis para o problema. A implementação adotada considera incorreta qualquer versão que atingir uma das seguintes condições para qualquer dos casos de testes: a) não executáveis devido a erros de sintaxe; b) condições de *loop* infinito; e, c) problemas de importação de pacotes e/ou bibliotecas.

8.1.6 Estruturação e Utilização dos Prompts nos Componentes Baseados em LLM

Conforme estabelecido no Capítulo 7, tanto a Abordagem Dependente de Descrição (ADD) quanto a Abordagem Não Dependente de Descrição (ANDD) foram projetadas com componentes que se baseiam em Modelos de Linguagem de Grande Porte (LLMs). Um dos elementos fundamentais para interagir eficientemente com um LLM é o *prompt template*. A seguir, são apresentados os dois *prompts* utilizados neste experimento, independentemente do LLM instanciado no momento do teste. Esses *templates* podem ser compreendidos por meio das variáveis de entrada, que são carregadas dinamicamente conforme a funcionalidade em investigação, e das seções de refinamento, que permanecem constantes ao longo de todo o experimento.

Conforme discutido na revisão apresentada por Zhao et al. (2023), mesmo não havendo uma fórmula única para construção de *prompts*, que seja a ideal para qualquer problema ou LLM, existem alguns ingredientes chave que, no geral, corroboram para um melhor desempenho do LLM na elucidação e resolução da tarefa a ele submetida. Assim, as seções de refinamento do comportamento do LLM nos *prompts* deste experimento foram estruturadas em quatro partes:

- **Contexto Inicial**: Define o papel que o LLM deve adotar, como o de um especialista em engenharia de software com ampla compreensão de códigos. Essa persona auxilia o LLM a assumir uma mentalidade que favorece análises detalhadas e precisas, proporcionando o contexto necessário para a tarefa e orientando a geração da resposta.
- **Tarefa**: Estabelece de forma clara a tarefa que o LLM deve executar, como inferir e descrever a funcionalidade pretendida do código fornecido ou identificar o trecho incorreto.
- **Formato de Saída**: Especifica o formato da resposta para garantir que seja utilizável no contexto em que será aplicada. Ao definir que a descrição deve ser concisa, o *prompt* assegura que a resposta seja eficiente e diretamente aplicável, especialmente em funções automatizadas que podem integrar o retorno do LLM.
- **Instruções**: Fornece diretrizes específicas que ajudam a moldar a resposta do LLM, assegurando que ela seja precisa e útil, evitando distrações.

O Código 5 exemplifica o *prompt* utilizado em ambas as abordagens, ADD e ANDD, no componente denominado Componente de Localização de Defeito.

Conforme demonstrado nas linhas 11 e 22 do código, há duas variáveis principais no template. A primeira, *question_description*, é dinamicamente preenchida com a descrição da funcionalidade que deveria ter sido implementada no código incorreto em análise.

Código 5: *Prompt template* utilizado no componente Localização de Defeito.

```
1  template_fl = """
2  You are a Software Engineer Specialist analyzing programming
    ↪ exercises.
3  Task: Analyze the provided incorrect code, Provide the ID of the
    ↪ line most likely to cause the fault.
4
5  Output Format:
6  Output will be used in a function call so do not add break lines
    ↪ between the fields.
7  All following fields are text fields.
8  buggycode: "The code text found in the faulty line.", buggyline:
    ↪ "The ID of the line with the buggycode (use only numbers,
    ↪ provide only one ID).", explanation: "The summary of explanation
    ↪ and reasons for why the line is faulty."
9
10 Exercise Description:
11 {question_description}
12
13 Instructions:
14 Each line in the code has an unique ID.
15 Line IDs start in 0.
16 Stay aware on the fault and NOT about the possible fix.
17 Maintain the given format and avoid use escape character.
18 Be aware of the output format.
19 Be consistent about buggyline and buggycode, they must represent the
    ↪ same line.
20
21 Code:
22 {input}
23 """
```

A segunda variável, *input*, recebe o código-fonte que está sendo avaliado. Seguindo as melhores práticas descritas na literatura sobre LLMs, o *prompt* inclui instruções claras e específicas sobre o comportamento esperado do modelo.

Nas seções de refinamento, a linha 2 define o contexto inicial, orientando o modelo sobre a "personalidade" que deve assumir. Na linha 3, é especificada a tarefa, que, neste caso, envolve identificar a linha de código mais provável de conter o defeito. Entre as linhas 5 e 8, são descritas as regras para a formatação da saída, incluindo a recomendação de evitar quebras de linha desnecessárias e o detalhamento das variáveis de saída esperadas. Nas linhas 14 a 19, são incluídas instruções específicas para refinar ainda mais o comportamento do modelo. Por exemplo, testes preliminares revelaram que, em alguns casos, o modelo selecionava uma linha (variável *buggyline*) que não correspondia

ao código apontado como defeituoso (variável *buggycode*); para mitigar esse problema, foi adicionada uma instrução específica.

O Código 6 apresenta o *prompt* utilizado na abordagem ANDD, especificamente no componente denominado Componente de Inferência de Descrição.

Código 6: *Prompt template* utilizado no Componente de Inferência de Descrição.

```
1  template_description_task = """
2  You are a Software Engineering Specialist with extensive experience
    ↪  in understanding and describing code functionality. Your
    ↪  expertise allows you to infer the intended functionality of code
    ↪  snippets based on their structure and logic.
3  Task:
4  You are given a faulty but valid Python code. Your task is to infer
    ↪  and describe the intended functionality that the code was
    ↪  supposed to implement.
5
6  Output Format:
7  Output will be used in a function call so do not add break lines
    ↪  between the fields.
8  All following fields are text fields.
9  Try to limit description to one sentence.
10 Add a Task Title in the beginning of the description.
11 description: "Concise explanation of the intended functionality of
    ↪  the code."
12
13 Instructions:
14 Be careful with break line commands and just use it when necessary.
15 Be aware of the output format.
16 Only describe what the correct code should do.
17 Do not explain why the code is faulty.
18 Description will be used to other developers to write new code.
19 It is not necessary ask for any usage example.
20
21 Code:
22 {input}
23 """
```

Diferente do *prompt* utilizado no componente de localização de defeito, o *prompt* destinado à descrição da funcionalidade pretendida para o código com defeito possui apenas uma variável de entrada, a variável *input*, localizada na linha 22. Essa variável é preenchida dinamicamente com o código incorreto, para o qual se deseja inferir e descrever a funcionalidade pretendida.

Nas seções de refinamento, a linha 2 define o contexto inicial, com uma especialização da "personalidade" do modelo para torná-lo mais adequado à tarefa de inferir a

funcionalidade pretendida. Nas linhas 3 e 4, é especificada a tarefa de inferir e descrever a funcionalidade correta do código dado, ignorando eventuais erros presentes. A ênfase aqui é garantir que o modelo não apenas descreva o código, mas sim se concentre na intenção funcional subjacente. Entre as linhas 6 e 11, é definido o formato da saída, destacando a necessidade de evitar quebras de linha e a importância de uma descrição concisa, limitada a uma sentença. Finalmente, nas linhas 13 a 19, são fornecidas instruções para que o modelo ignore os erros no código e foque exclusivamente na funcionalidade que o código deveria implementar, ajudando a manter a relevância e utilidade da resposta na fase de localização do defeito.

Conforme observado, todos os *prompts* utilizados neste experimento foram desenvolvidos em inglês. Essa escolha é justificada pelo fato de que a maioria dos LLMs possui um conhecimento paramétrico, ou seja, o conhecimento presente nos pesos da rede, adquirido predominantemente a partir de bases de dados em inglês, que são mais extensas do que em outros idiomas.

8.2 Resultados e Análises

Nesta seção, são apresentados e discutidos os resultados obtidos em cada uma das abordagens investigadas, com foco em responder às Questões de Pesquisa ***RQ1 Quão eficazes são os LLMs na localização de defeitos em código-fonte?*** e ***RQ2 Os LLMs são capazes de localizar defeitos mesmo sem uma descrição prévia da funcionalidade?***

A análise foca na eficácia das abordagens para a localização de defeitos em códigos escritos em linguagem Python, considerando o conjunto de dados descrito na seção anterior e cada um dos LLMs utilizados. Importante destacar que, para ambas as abordagens, foi adotada a configuração padrão de geração de resposta, ou seja, uma única indicação de local do defeito. Portanto, todas as análises concentram-se exclusivamente na métrica ACC@1.

8.2.1 Abordagem Dependente de Descrição (ADD)

Primeiramente, analisam-se os resultados da Abordagem Dependente de Descrição (ADD) em relação à capacidade de localizar corretamente defeitos nos códigos do conjunto de dados utilizado. A Tabela 8.2 apresenta o número absoluto e a porcentagem de programas nos quais a localização do defeito foi corretamente identificada por cada LLM, considerando os diferentes problemas analisados. Esta porcentagem é calculada com base no número total de versões defeituosas de cada problema, conforme mostrado na Tabela 8.1. A coluna “Falha do LLM” indica a frequência e a porcentagem de vezes

em que cada modelo falhou em produzir uma saída compatível com o segmento de código defeituoso.

Tabela 8.2: Resultados da Abordagem Dependente de Descrição considerando todos os problemas selecionados, ambos os LLMs e a heurística Ochiai

	GPT 3.5		Llama 3		Ochiai
Problema	**ACC@1**	**Falha do LLM**	**ACC@1**	**Falha do LLM**	**ACC@1**
Question 1	220 (38,26%)	-	161 (28,00%)	2 (0,35%)	31 (5%)
Question 3	286 (92,86%)	-	231 (75,00%)	4 (1,3%)	125 (41%)
Question 4	347 (97,20%)	-	292 (81,79%)	-	137 (38%)
Question 5	79 (73,15%)	-	55 (50,93%)	-	78 (72%)

Fonte: Elaborada pelo autor.

Nota-se que, nas rodadas de teste utilizando o GPT-3.5, não houve falha, ou seja, todas as saídas produzidas pelo modelo puderam ser processadas e foram compatíveis com o código presente na versão defeituosa. Já no caso do Llama 3, houve falhas para os problemas Question 1 e Question 3, ainda que em percentuais baixos: 0,35% e 1,3% do total de versões com defeito, respectivamente. Por meio de uma inspeção dos casos de falha, verificou-se que elas ocorreram em situações em que o retorno do LLM provocava erro no *parser* de saída do LangChain ou, mesmo após sucessivas tentativas, a API do serviço utilizado não conseguia fornecer uma resposta. Para a heurística Ochiai, não se pode falar em falha, pois esse método não envolve a geração de código.

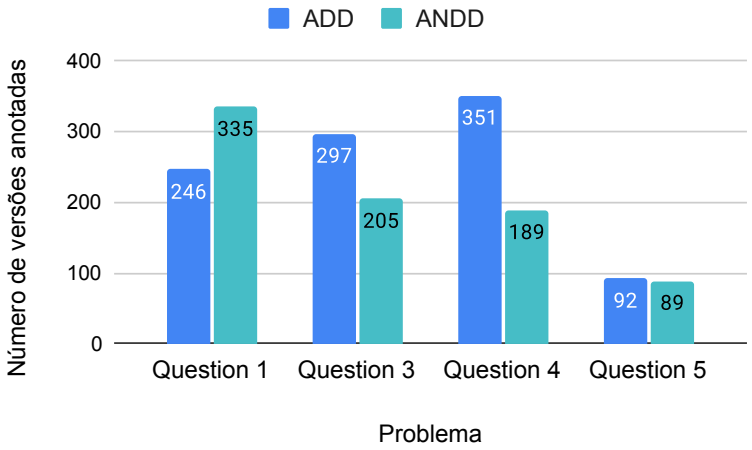
Com relação ao desempenho na métrica ACC@N, nota-se que, para todos os quatro problemas considerados, a abordagem ADD, quando instanciada com GPT-3.5, superou as outras duas técnicas de localização de defeitos. Além disso, ao observar o desempenho do Llama 3, percebe-se que, apesar de inferior ao GPT-3.5, ele ainda supera a heurística Ochiai em todos os problemas, com exceção do Question 5. Especificamente em Question 5, é o único cenário em que Ochiai se aproxima (ou supera) os LLMs. Ao inspecionar esse conjunto, nota-se que os programas apresentam uma lógica relativamente mais complexa em comparação com os demais, o que impõe um desafio maior para a abordagem baseada em LLM, que precisa raciocinar sobre o código. Por outro lado, essa complexidade não afeta uma heurística baseada em dados de cobertura, como Ochiai, que não é impactada por essa característica.

Observando os resultados para o problema Question 4, nota-se que o modelo GPT-3.5 indicou corretamente a localização do defeito em 347 das 357 versões defeituosas, representando 97% dos casos. No mesmo problema, o Llama 3 obteve sucesso em 81% das vezes. Ambos os modelos alcançaram seus melhores desempenhos nesse

problema. O desempenho satisfatório de ambos os modelos também está relacionado ao método de avaliação adotado neste experimento.

Como explicado na Seção 8.1.4, o conjunto de dados original não possui anotações sobre a localização dos defeitos nas versões defeituosas, por isso foi utilizado um procedimento que considera a diferença entre a versão defeituosa e sua correção. Assim, o número máximo de versões que podem ser avaliadas para a localização de defeitos é igual ao número de versões com pelo menos uma versão correta conhecida.

Figura 8.2: Número de versões com anotação de defeito por problema e abordagem. O número é igual ao total de versões corrigidas por qualquer LLM em cada abordagem (Dependente de Descrição ou Não Dependente de Descrição)



Fonte: Elaborada pelo autor.

Como pode ser visto em detalhes no Anexo G, os modelos GPT e Llama produziram cerca de 85% e 95%, respectivamente, de correções para as versões defeituosas do problema Question 4. Usando todas as versões corrigidas por pelo menos um dos modelos para gerar as anotações da localização do defeito, o problema Question 4 teve 351 programas anotados (Consultar a Figura 8.2 para referência). Esse valor representa 98% do total de versões defeituosas, conforme mostrado na Tabela 8.1. Em outras palavras, há um número elevado de versões elegíveis para a localização de defeitos para o conjunto Question 4. Por outro lado, o número de versões anotadas disponíveis para o problema Question 1 é de 246, o que representa 43% do total de 575 versões. Portanto, proporcionalmente ao número total de versões do problema Question 1, poucas versões estão disponíveis para a localização de defeitos, o que corrobora os valores baixos nesta tarefa para este problema.

Em suma, respondendo à Questão de Pesquisa **RQ1**, os resultados sugerem que, mesmo ao se utilizar uma métrica conservadora (comparando o número total de versões defeituosas e não apenas a quantidade de versões anotadas com local de defeito), tanto o GPT-3.5 com uma média de 75,36% de acurácia, quanto o Llama 3, com 58,93%, superaram o desempenho médio de 39% da heurística Ochiai. Isso indica o potencial

promissor dos LLMs, mesmo os de propósito geral, para identificar defeitos em programas de pequeno porte.

8.2.2 Abordagem Não Dependente de Descrição (ANDD)

Nesta subseção, são apresentados os resultados da Abordagem Não Dependente de Descrição (ANDD), que avalia a capacidade dos LLMs de localizar defeitos em código Python sem uma descrição prévia da funcionalidade, respondendo assim à Questão de Pesquisa RQ2. A Tabela 8.3 mostra os resultados de ACC@1 obtidos para cada um dos LLMs utilizados — GPT e Llama —, além da heurística Ochiai, em relação aos problemas selecionados, considerando todas as versões defeituosas disponíveis. Diferentemente da análise relativa à ADD, para a ANDD não são exibidas as colunas de falha do LLM, pois não houve ocorrência de falhas em nenhum dos LLMs.

Tabela 8.3: Resultados de ACC@1 da ANDD considerando todos os problemas selecionados, ambos os LLMs e a Ochiai

Problema	GPT 3.5	Llama 3	Ochiai
Question 1	278 (48.35%)	216 (37.57%)	31 (5%)
Question 3	193 (62.66%)	184 (59.74%)	125 (41%)
Question 4	176 (49.30%)	131 (36.69%)	137 (38%)
Question 5	84 (77.78%)	81 (75.00%)	78 (72%)

Fonte: Elaborada pelo autor.

Assim como na análise da Abordagem Dependente de Descrição (ADD), na ANDD a configuração com o GPT-3.5 foi superior em todos os problemas. Além disso, os resultados mostram que, das oito combinações entre modelos e problemas, seis alcançaram aproximadamente 50% ou mais de sucesso na localização de defeitos. Isso sugere que a abordagem ANDD pode inferir a funcionalidade pretendida e, em seguida, utilizar essa inferência para identificar o local do defeito.

Apesar de ambos os modelos continuarem superiores à heurística Ochiai em todos os casos, com exceção do Llama 3, no problema Question 5, ao se comparar esses resultados com a abordagem ADD, observa-se um certo declínio de desempenho da ANDD nos problemas 3 e 4. Para uma análise mais detalhada da comparação entre as abordagens, a Tabela 8.4 resume os resultados de ACC@1 alcançados por ADD e ANDD, considerando todos os problemas.

Analisando especificamente os resultados do GPT-3.5, percebe-se uma queda no desempenho da ADD para a ANDD nos problemas 3 e 4, com uma redução de 286 para 193 no Question 3 e de 347 para 176 no Question 4. Quanto aos resultados do Llama

Tabela 8.4: Comparativo de ACC@1 entre ADD e ANDD considerando todos os problemas

	GPT 3.5		Llama 3	
Problema	**ADD**	**ANDD**	**ADD**	**ANDD**
Question 1	220	**278**	161	**216**
Question 3	**286**	193	**231**	184
Question 4	**347**	176	**292**	131
Question 5	79	**84**	55	**81**

Fonte: Elaborada pelo autor.

3, observa-se o mesmo comportamento, ou seja, o desempenho da ANDD é inferior ao da ADD nos problemas Question 3 e Question 4. A explicação para esse fenômeno pode ser encontrada na caracterização do conjunto de dados apresentada na Tabela 8.1. As descrições de funcionalidade dos problemas 3 e 4 incluem exemplos de entradas e saídas esperadas. Como na abordagem ANDD a descrição não é conhecida e o processo de inferência não consegue gerar esse nível de detalhe, tanto a geração de correções candidatas quanto a posterior localização do defeito acabam sendo afetadas.

Em relação aos problemas Question 1 e Question 5, o desempenho do GPT e do Llama foi superior na abordagem que não recebe a descrição como entrada (ANDD). No caso específico de Question 1, a descrição original é praticamente inexistente, consistindo apenas no nome da tarefa, sem a especificação da funcionalidade esperada. Nesse cenário, o mecanismo desenvolvido para que o LLM, antes de tentar localizar o defeito, busque inferir a funcionalidade pretendida do código se torna muito útil. Isso faz com que, na prática, a ANDD tenha a chance de gerar uma descrição mais precisa do que a fornecida para a abordagem ADD. Por óbvio, o benefício depende da capacidade do LLM de inferir a funcionalidade e não apenas de descrever o código recebido.

Esta comparação destaca a importância de considerar o contexto específico de aplicação ao escolher entre abordagens dependentes ou não de descrição. Em cenários onde a descrição é detalhada e precisa, a ADD tende a ser mais eficaz, enquanto a ANDD pode oferecer vantagens em casos de descrições limitadas ou inexistentes.

Em resumo, respondendo à Questão de Pesquisa **RQ2**, os resultados corroboram a hipótese de que os LLMs são capazes de localizar defeitos mesmo sem uma descrição prévia da funcionalidade, com a abordagem ANDD atingindo uma taxa média de sucesso de 59,52% para o GPT-3.5 e 52,25% para o Llama 3. Ademais, foi demonstrado que essa abordagem é especialmente útil nos casos em que a descrição da funcionalidade for deficiente.

8.3 Comparação com Abordagem Baseada em CNN

Esta seção apresenta uma comparação entre a abordagem de localização de defeitos avaliada neste capítulo, baseada em LLM, e a proposta apresentada no Capítulo 5, baseada em CNN. Como as propostas que envolvem LLM lidam apenas com a localização de defeitos em nível de linhas, a comparação foi realizada com a abordagem Conv_1D. Para tanto, foi selecionado um dos projetos do conjunto de dados Defects4J, que já havia sido utilizado na avaliação experimental do Capítulo 6.

A Tabela 8.5 apresenta os valores de ACC@N para 50 das 65 versões com defeitos no projeto Lang, considerando os rankings da heurística Ochiai, da abordagem Conv_1D e da estratégia *fl-top-k*, instanciada com um modelo GPT-3.5 turbo e $k = 5$. A quantidade de versões ter sido reduzida para 50 é resultado de um processo de seleção manual que considerou apenas as versões cujas alterações que corrigiram o defeito foram implementadas em um único método.

Tabela 8.5: Resultados de ACC@N para o projeto Lang do conjunto de dados Defects4J alcançados pela heurística Ochiai, a proposta baseada em CNN, Conv_1D, e a proposta baseada em LLM, fl-top-5

	ACC@1	%	**ACC@3**	%	**ACC@5**	%
Ochiai	5	10%	12	24%	15	30%
Conv_1D	4	8%	7	14%	10	20%
fl-top-5	**14**	28%	**21**	42%	**23**	46%

Fonte: Elaborada pelo autor.

Como já havia sido constatado anteriormente, a abordagem Conv_1D não é competitiva nem mesmo em comparação com a heurística Ochiai, visto que, para esse recorte de versões, os resultados de ACC@N para todos os valores de N são inferiores. Por outro lado, como pode ser observado, para todos os valores de ACC@N, a estratégia *fl-top-5*, com GPT-3.5 turbo, obteve os melhores resultados, superando a heurística Ochiai em quase três vezes no caso do ACC@1.

O trabalho de Yang et al. (2024), que introduz o LLMAO, uma abordagem baseada em *fine-tuning* de um modelo de larga escala específico para código, também utiliza o conjunto de dados Defects4J para avaliar a técnica proposta. No trabalho, não são reportados os resultados estratificados por projeto, portanto, não é possível fazer uma comparação direta dos resultados. Todavia, são reportados os valores percentuais relativos ao total de 395 programas disponíveis entre os cinco projetos da versão do conjunto de dados igual a utilizada nesta pesquisa.

De acordo com o artigo, a técnica atingiu 22,3% de acertos no ACC@1, 37,7% no ACC@3 e 46% para ACC@N. Embora esta não seja a comparação ideal, os valores alcançados pela estratégia *fl-top-5*, em termos proporcionais, dentro do subconjunto das

versões do Lang, se comparam aos resultados relatados no LLMAO, considerando todo o conjunto de versões, sendo, inclusive, superiores para @1 e @3.

Apesar desses resultados, é importante pontuar que, devido à limitação do tamanho da janela de contexto suportada pelo GPT-3.5 (4096 tokens), não seria possível, em alguns casos, submeter o arquivo Java inteiro para o fluxo da estratégia *fl-top-k*. Portanto, foi realizado um processo manual de separação dos métodos nos quais houve alteração na versão de correção do defeito. Embora esse procedimento introduza um forte viés, em última instância ele poderia ser substituído por estratégias de localização de defeitos em nível de métodos, cuja precisão é substancialmente mais alta do que a localização em nível de linha.

Naturalmente, um cenário comum de alteração de código para a correção de um defeito consiste na adição de novas linhas, sem alteração no conteúdo das linhas existentes no código. Para efeito de anotação do local do defeito, costuma-se aceitar a linha anterior ou posterior ao local da inserção como sendo o local, na versão com defeito, que deve ser marcado como defeituoso.

Após uma inspeção dos *patches* de correção dos defeitos relativos às 50 versões utilizadas neste experimento, utilizando-se da catalogação realizada por Sobreira et al. (2018), constatou-se que 23 delas foram corrigidas por adição de novas linhas e, em diversos desses casos, o LLM foi capaz de acertar o local da falha na primeira posição do ranking. Diante disso, pode-se conjecturar que, em tese, seria mais fácil para o LLM identificar o local como defeituoso, caso ele já conhecesse o código correto, isto é, o código com a linha adicionada. Essa situação pode estar relacionada a um problema denominado vazamento de dados na avaliação de LLMs, que será discutido em mais detalhes na seção a seguir.

8.4 Problema do Vazamento de Dados em LLMs

Os modelos de linguagem de grande porte, apesar de estarem intimamente ligados, por definição, à área de processamento de linguagem natural, vêm sendo amplamente utilizados nas mais diversas áreas de conhecimento, com adesão inquestionável tanto na indústria quanto na academia (ZHAO et al., 2023). A despeito dos inquestionáveis resultados da aplicação dos LLMs, dos patentes avanços que esses modelos têm apresentado quanto à capacidade de resolução de tarefas complexas, há um esforço da comunidade científica no sentido se desenvolverem mecanismos de avaliação robustos para esse tipo de tecnologia (CHANG et al., 2024).

Por definição, os LLMs podem aumentar suas capacidades à medida que são expostos a volumes de dados cada vez maiores e, na prática, tanto modelos públicos quanto fechados apresentam comumente bases de dados de treinamento na ordem de

trilhões de tokens (ZHAO et al., 2023). A exposição do modelo a um volume tão grande de dados no processo de pré-treinamento aumenta o risco do problema de vazamento de dados, isto é, o ameaça de que ao se avaliar uma técnica baseada em LLM, os dados sob investigação tenham, na verdade, integrado os *corpora* utilizados no pré-treino do modelo.

O problema de vazamento de dados no contexto da avaliação de métodos baseados em redes neurais profundas não é exclusividade da área de processamento de linguagem natural e já é apontado como uma preocupação de em diversos contextos (SAMALA et al., 2020; HU et al., 2022). Quando se fala especificamente dos modelos GPT da Open IA, o trabalho de Aiyappa et al. (2023) aponta que além do problema fundamental de vazamento, ou seja, o risco de os dados sob teste terem sido conhecidos do modelo durante o treinamento, a simples exposição sucessiva do dado sob avaliação ao modelo pode constituir-se em vazamento. Isso aconteceria porque os modelos GPT da Open IA utilizam o conceito de Reinforcement Learning from Human Feed- back (RLHF) (CHRISTIANO et al., 2017). Sendo assim, considerando o volume de usuários da plataforma da Open AI, mais de 100 milhões na data desta pesquisa, maior é a chance de que algum usuário já tenha apresentado ao modelo parcial ou totalmente os dados utilizados para avaliação.

Nesse contexto, o Defects4J e outros conjuntos de dados construídos a partir de repositórios de projetos de softwares abertos, que são vastamente utilizados em trabalhos relacionados ao processo de depuração de código, podem ter feito parte dos dados utilizados no pré-treino de muitos LLMs. Portanto, a fim de se obter alguma indicação de que os códigos-fonte catalogados como defeito dentro do conjunto de dados do Defects4j podem fazer parte do conhecimento paramétrico do GPT-3.5, foi realizado um procedimento definido como:

1. Para cada um dos projetos presentes no benchmark, selecionou-se aleatoriamente uma versão com defeito;
2. Novamente, ao acaso, foi selecionado algum dos métodos onde se encontrava o defeito, ou seja, local onde houve alteração de código para a correção do defeito;
3. Um trecho do código do método selecionado foi então submetido ao GPT-3.5 e perguntado qual seria a próxima linha do código.

Os defeitos selecionados foram: lang-3, chart-19, closure-59, math-69, mockito-5. Após a submissão de trecho de código e requisição de indicação da próxima linha, o LLM apontou exatamente o conteúdo que existe na versão corrigida do respectivo código para os códigos provenientes do Lang, Chart e do Math. Para os trechos de código extraídos de Closure e Mockito, na primeira iteração o LLM completou um código diverso, porém, em ambos os casos, quando realizada uma segunda interação solicitando a fonte do código, a resposta foi satisfatória.

O mesmo procedimento foi repetido para um trecho de código do programa *wrong_5_100* do conjunto de dados da tabela, usado neste experimento. O GPT foi também foi capaz de concluir o código, mas quando perguntado sobre a origem do programa a resposta foi que o trecho não parecia ter sido extraído de nenhum repositório online ou projeto de software aberto, mas que poderia ser um código comum em tutoriais de programação Python.

Esses achados são uma evidência que corrobora a suspeita de que, de fato, o GPT-3.5 turbo tem informações (ou o próprio código-fonte) para parte do seu conhecimento paramétrico, ou seja, os repositórios desses projetos podem ter sido usados como parte dos dados de treinamento do modelo. Nesse sentido, o uso de repositórios públicos de código para a avaliação de métodos e ferramentas baseados em LLM constitui-se em uma importante ameaça à validade dos resultados.

Por todo o exposto, é razoável entender que as pesquisas na área de Engenharia de Software que utilizem Modelos de Linguagem de Grande Porte, pelo menos quando se tratar de tarefas relacionadas a código-fonte, apresentem alguma estratégia para lidar com a ameaça de vazamento ou que ao menos discutam essa questão. Os recentes trabalhos de Yang et al. (2024) e Kang, An e Yoo (2024), por exemplo, relatam bons resultados utilizando o conjunto de dados Defect4J, respectivamente para localização de defeitos em nível de linha de código e em nível de método. Apesar de citarem brevemente essa ameaça, nenhum dos trabalhos apresenta uma discussão mais profunda sobre o tema ou realiza experimentação com dados livres de contaminação.

8.5 Limitações e Ameaças à Validade da Abordagem Baseada em LLM

Apesar dos resultados promissores obtidos com as abordagens baseadas em LLM, existem limitações e ameaças à validade que precisam ser consideradas ao interpretar os resultados deste estudo.

Primeiramente, as abordagens propostas foram testadas em programas relativamente pequenos e em problemas simples de programação, desenvolvidos por estudantes de graduação. Esses programas geralmente possuem uma lógica menos complexa e são compostos por poucas linhas de código. Como consequência, a eficácia dos LLMs na localização de defeitos em sistemas de software mais complexos, com milhares de linhas de código e múltiplas dependências, permanece incerta. A generalização dos resultados para aplicações em cenários reais de desenvolvimento de software ainda precisa ser cuidadosamente avaliada.

Além disso, é importante considerar a possibilidade de que os programas testados

façam parte do conhecimento paramétrico dos LLMs utilizados. Os modelos GPT-3.5 e Llama 3 foram treinados em vastas quantidades de dados, incluindo repositórios públicos de código-fonte. Assim, existe a chance de que os programas avaliados neste estudo, ou programas muito similares, já estivessem presentes no conjunto de dados de treinamento dos modelos, o que poderia influenciar positivamente o desempenho dos LLMs na tarefa de localização de defeitos. Essa possibilidade levanta questões sobre a validade externa dos resultados, uma vez que a performance dos LLMs pode não ser replicada em cenários onde os modelos não têm conhecimento prévio dos programas analisados.

Finalmente, a abordagem atual considera apenas a posição ACC@1 no ranking de suspeitas, o que limita a avaliação da capacidade dos LLMs em fornecer um diagnóstico abrangente de possíveis locais de defeitos. Em situações reais, a localização precisa do defeito pode requerer a análise de múltiplas linhas de código suspeitas, o que não foi explorado neste experimento. Esse foco restrito pode não capturar adequadamente a complexidade da tarefa de depuração em sistemas mais robustos. Por outro lado, ainda que a localização do defeito não seja precisa, a capacidade dos LLMs em gerar uma explicação sobre o defeito pode ser usada para auxiliar o desenvolvedor a pensar sobre o problema e eventualmente na solução.

Essas limitações sugerem a necessidade de estudos futuros que explorem a aplicação das abordagens baseadas em LLM em contextos mais desafiadores, incluindo programas de maior escala e problemas de programação mais complexos, além de investigar estratégias para lidar com a possibilidade de conhecimento prévio dos LLMs sobre os códigos analisados.

8.6 Considerações Finais

Este capítulo apresentou a metodologia utilizada para avaliar a proposta de resolução do problema da localização de defeitos com a aplicação de Modelos de Linguagem de Larga Escala. Para isso, foram instanciados os modelos GPT-3.5 turbo e Llama 3 e realizada uma série de testes sobre o conjunto de dados contendo mais de 1k programas em linguagem Python.

Mesmo com a adoção de uma estratégia mais conservadora, ou seja, obtendo-se apenas uma resposta para o local do defeito, em vez de se ter um ranking de elementos suspeitos, os resultados demostraram a aplicabilidade das abordagens baseadas em LLM. A abordagem que supõe a existência de uma descrição da funcionalidade (ADD) tende a ter melhor desempenho, porém, a abordagem ANDD se mostrou competitiva para cenários em que a descrição seja deficiente.

Entretanto, a análise dos resultados, a natureza do conjunto de dados, possibilidade de vazamentos de dados no processo de treinamentos dos LLMs e outras questões

apontadas na seção de ameaças à validade deixaram evidente a necessidade de expansão da experimentação para asseverar maior generalização aos resultados verificados. Nesse sentido, o capítulo a seguir apresenta um novo experimento sobre um conjunto de dados introduzido recentemente, construído especialmente para lidar com a questão do vazamento de dados na avaliação de LLM, no contexto de localização e reparo de defeitos em software.

CAPÍTULO 9

Experimentação em Cenário Livre de Vazamento

Este capítulo apresenta a metodologia de experimentação projetada para avaliar as diferentes estratégias de geração de ranking de elementos suspeitos baseadas em LLM. O foco desse experimento é evidenciar o comportamento das propostas assegurando a aplicabilidade delas em contexto de dados livre ou pouco susceptível ao problema do vazamento de dados na avaliação de técnicas compostas por LLM.

Inicialmente, a Seção 9.1 descreve a origem e a caracterização do conjunto de dados e da ferramenta que o compõe. Em seguida, são explicadas as decisões tomadas no processo de seleção dos programas, ferramentas, métricas e modelos utilizados na experimentação, assim como introduz-se um algoritmo para combinação de rankings. Na Seção 9.2, são apresentados e discutidos os resultados alcançados e a Seção 9.3 traz um comparativo da abordagem baseada em LLM proposta por este trabalho com o estado da arte da localização de defeitos em nível de linhas de código. Por fim, a Seção 9.4 discute as ameaças à validade dos experimentos apresentados.

9.1 Metodologia e Projeto Experimental

Este experimento responde às seguintes Questões de Pesquisa:

RQ3: Como se comportam as estratégias de localização de defeitos baseadas em LLM para programas com diferentes níveis de complexidade?

RQ4: Qual impacto da inclusão da descrição da funcionalidade na localização de defeitos baseada em LLM?

RQ5: Qual impacto dos dados de cobertura e como o LLM se compara com heurísticas tradicionais de localização de defeitos?

9.1.1 Conjunto de Dados

A avaliação experimental apresentada neste capítulo foi realizada sobre o conjunto de dados ConDefects[1], disponível publicamente. Esse conjunto foi introduzido recentemente por Wu et al. (2024) e tem como objetivo servir como um *benchmark* para métodos de localização de defeitos e/ou reparo automatizado de software, especialmente aqueles baseados em LLMs.

Os LLMs com capacidade para realizar tarefas relacionadas a linguagens de programação são tipicamente treinados sobre grandes volumes de dados, incluindo repositórios de projetos de código aberto. Dessa forma, a utilização de projetos de código aberto para validação de métodos de localização de defeitos ou reparo de software baseados em LLMs pode resultar em uma avaliação enviesada, pois o treinamento do próprio LLM pode ter incluído tais repositórios.

O ConDefects contém 2.045 programas Java com defeitos e 2.864 programas Python com defeitos. Para cada programa, o conjunto de dados apresenta o código com defeito, o código corrigido e a indicação da localização do defeito, isto é, o número da linha na qual a versão com defeito é alterada para se obter a versão corrigida. Todos os programas são originários da plataforma de competição online AtCoder (INC., 2024) e foram produzidos entre outubro de 2021 e junho de 2022.

A Tabela 9.1 apresenta algumas estatísticas do ConDefects, separando-se entre programas Java e Python.

Tabela 9.1: Resumo de caraterísticas do conjunto de dados ConDefects

Característica	**Java**	**Python**
Qtd. Tarefas	810	985
Qtd. Arquivos	2.045	2.864
Qtd. média de linhas de código	259,22	49,03
Qtd. média de funções	22,45	2,91

Fonte: Elaborada pelo autor.

Observa-se que existem mais programas na linguagem de programação Python; por outro lado, os programas em Java possuem, em média, mais linhas de código por arquivo, assim como uma maior quantidade de funções (ou métodos).

Outra característica relevante deste conjunto de dados é que as tarefas possuem um nível de dificuldade associado. Os valores que caracterizam a dificuldade são inteiros que variam de 0 a 3.581. De acordo com o trabalho que introduz o ConDefects, os valores de dificuldade podem ser categorizados em três níveis: *low* (0–400), *medium* (401–1.300) e *high* (1.301–3.581).

[1] <https://github.com/appmlk/ConDefects/tree/main>

Após a coleta de todas as tarefas do conjunto de dados, separando-as por nível de dificuldade, constatou-se uma distribuição de 483 atividades de nível alto (*high*), 310 de nível médio (*medium*) e 356 de nível baixo (*low*).

Além do conjunto de programas propriamente dito, contendo os artefatos de código e a informação do local do defeito, o ConDefects também traz uma suíte de casos de teste e uma ferramenta para suportar o uso do conjunto de dados. Em resumo, a ferramenta facilita:

- a coleta e separação das tarefas a partir de diferentes critérios, como linguagem de programação, competição (ABC, ARC ou AGC), intervalo de datas em que os programas foram produzidos e intervalo de valores de dificuldade;
- a obtenção de metadados sobre as tarefas, tais como a lista de tarefas, a lista de casos de teste para uma dada tarefa, a lista de programas disponíveis para uma determinada tarefa e detalhes sobre o local e conteúdo defeituoso de um programa específico;
- a execução dos casos de teste sobre os programas selecionados e a coleta dos dados de cobertura, que incluem a matriz de cobertura, o resultado da execução de cada teste e a lista de testes executados.

Originalmente, o conjunto de dados do ConDefects não possui uma descrição das tarefas de programação para as quais os programas foram implementados. Entretanto, como os programas foram extraídos da plataforma AtCoder, que está disponível online, foi realizado um procedimento de *web scraping* via Selenium (SELENIUM, 2024) para que o conjunto de dados utilizado neste experimento pudesse, eventualmente, contar com a descrição das tarefas.

9.1.2 Seleção de Programas

Conforme mencionado anteriormente, a ferramenta ConDefects oferece funcionalidades para a seleção de programas que viabilizem o projeto de experimentos específicos. Utilizando a interface de linha de comando da ferramenta, foram executados os comandos apresentados no Código 7 para a construção de três conjuntos de dados: *low*, *medium* e *high*.

O parâmetro `-w` indica o diretório local para o qual se deseja enviar os artefatos relativos aos programas selecionados, enquanto o parâmetro `-d` especifica o intervalo do nível de dificuldade dos programas a serem selecionados. Dessa forma, foram criados três conjuntos de dados, um para cada nível de dificuldade das tarefas de programação.

Para os testes realizados com o intuito de investigar o desempenho do LLM na localização de defeitos, considerando diferentes níveis de complexidade dos programas,

Código 7: Comandos executados na ferramenta do ConDefects para geração de conjuntos de programas separados por nível de dificuldade e linguagem de programação.

```
#low
Python3 ConDefects.py checkout -w difficulty/low -d 0 400

#medium
Python3 ConDefects.py checkout -w difficulty/medium -d 401 1300

#high
Python3 ConDefects.py checkout -w difficulty/high -d 1301 3581
```

foram selecionados aleatoriamente, com distribuição uniforme de probabilidade de escolha, 100 programas para cada um dos três níveis de dificuldade e cada linguagem de programação.

Outra seleção de programas foi realizada para este experimento, desta vez separando-se as tarefas de programação por competição e considerando apenas os programas construídos no ano de 2024. O Código 8 apresenta os comandos executados na ferramenta ConDefects para a geração de dois conjuntos de dados, ABC e ARC, para as linguagens de programação Python e Java.

Código 8: Comandos executados na ferramenta do ConDefects para geração de conjuntos de programas produzidos em 2024 separados por competição e linguagem.

```
#abc
Python3 ConDefects.py checkout -w contest/arc -t 2024-01-01
 ↪  2024-12-31 -l python -c abc
Python3 ConDefects.py checkout -w contest/arc -t 2024-01-01
 ↪  2024-12-31 -l java -c abc

#arc
Python3 ConDefects.py checkout -w contest/arc -t 2024-01-01
 ↪  2024-12-31 -l python -c arc
Python3 ConDefects.py checkout -w contest/arc -t 2024-01-01
 ↪  2024-12-31 -l java -c arc
```

Assim como no caso anterior, o parâmetro `-w` indica o diretório local de destino dos artefatos dos programas selecionados, ou seja, a versão incorreta, a versão corrigida, e o local do defeito; o parâmetro `-t` especifica um intervalo de tempo utilizado para filtrar as tarefas pela data de implementação; o parâmetro `-l` discrimina a linguagem de programação de interesse; e, por fim, o parâmetro `-c` informa a competição de onde foram extraídas as tarefas.

Para os testes com o recorte de programas do ano de 2024, que envolviam a necessidade de obtenção dos dados de cobertura de fluxo de controle, foi realizado um processo de amostragem aleatória simples no conjunto de programas provenientes da competição ABC. Esse procedimento foi necessário em função do custo computacional para a execução de cada versão sobre todos os casos de teste e geração das matrizes de cobertura. Além disso, durante o processo de execução da suíte de testes, alguns programas apresentaram erros que impossibilitaram a coleta dos dados de cobertura ou tornaram o tempo de execução proibitivo. Assim, a Tabela 9.2 apresenta um resumo das quantidades originais de cada conjunto e a quantidade final de versões para as quais foram obtidos os dados de cobertura de fluxo de controle.

Tabela 9.2: Quantidade de programas produzidos em 2024 nas linguagens Python e Java *versus* a quantidade de programas selecionados

	Python		**Java**	
	# original	**# selecionado**	**# original**	**# selecionado**
ABC	439	280	366	184
ARC	127	116	50	41

Fonte: Elaborada pelo autor.

9.1.3 Ferramentas, Métricas e Modelos

A principal ferramenta utilizada para suportar os testes realizados neste estudo experimental foi uma aplicação de código aberto, escrita em Python, disponibilizada junto ao conjunto de dados ConDefects. Esta ferramenta oferece funcionalidades para apoiar a aplicação de técnicas de localização de defeitos sobre os programas do conjunto. Um dos recursos mais relevantes é a geração de dados de cobertura de código a partir da execução dos casos de teste. Para tal, a ferramenta do ConDefects depende das bibliotecas JaCoCo (JaCoCo Contributors, 2024) e Coverage (BATCHELDER, 2024), que geram dados de cobertura para Java e Python, respectivamente.

Quanto às métricas utilizadas para a avaliação do desempenho das abordagens de localização de defeitos, foram aplicadas as métricas ACC@N e Exam, previamente definidas no Capítulo 2. A métrica ACC@N indica a quantidade de elementos realmente responsáveis pelo defeito que aparecem no ranking até a posição N, enquanto a métrica Exam representa o percentual médio de código percorrido até a identificação do defeito.

Adicionalmente, para mensurar o custo associado à aplicação de LLM, foi utilizada uma medida de quantificação de tokens, que avalia o tráfego de informações entrando ou saindo do modelo. Essa métrica é calculada conforme os seguintes passos:

1. Obter o mesmo *tokenizador* utilizado pelo LLM instanciado ou uma alternativa compatível;
2. *Tokenizar* toda a sequência de texto, seja de entrada ou saída do modelo, que se deseja contabilizar;
3. Obter a extensão da sequência após a *tokenização*.

Com esse procedimento, a mensuração da quantidade de tokens é equivalente àquela utilizada pelo próprio modelo de linguagem, permitindo, por exemplo, a estimativa de custo para casos em que se utiliza um modelo servido online com cobrança por tokens trafegados. Neste experimento, foi utilizado o tiktoken (OPENAI, 2024), um tokenizador compatível com o LLM empregado nas abordagens de localização de defeitos.

Considerando a necessidade de mensuração do consumo médio de tokens em um experimento, a seguinte fórmula foi aplicada:

$$CustoTokens(R) = \frac{1}{N} \times \sum_{i=1}^{N} \sum_{j=1}^{P} \sum_{m=1}^{K} qtdTokens(R,N,P,K), \tag{9-1}$$

onde, N é a quantidade de execuções de uma estratégia de geração de ranking para um mesmo conjunto de programas, P é a quantidade de programas com defeito de um conjunto de programas sob investigação, k é a quantidade de respostas devolvidas pelo LLM, ou seja, é o tamanho do ranking gerado; e, por fim, $qtdTokens(R)$ é uma função que dada uma sequência textual R submetida ou recebida do LLM, realiza o procedimento descrito anteriormente e devolve a quantidade de tokens presente na sequência.

Por fim, o modelo GPT-3.5 Turbo foi o LLM predominantemente utilizado nas abordagens deste experimento. Assim como nos experimentos do Capítulo 8, todas as implementações necessárias para viabilizar a interação entre os componentes da abordagem e o LLM foram realizadas em Python 3, utilizando os recursos do *framework* LangChain [2]. Para uma análise pontual, como investigação complementar do comportamento da estratégia *fl-it-k*, também foram utilizados os modelos GPT-4o (OPENAI, 2023b) e Claude 3.5 Sonnet (ANTHROPIC, 2024), conforme detalhamento na Seção 9.2.1.

9.1.4 Combinação de Ranking de Heurística com Rankings de LLM

As propostas de geração de rankings de elementos de código suspeitos de serem a causa do defeito, baseadas em LLMs, consideram um parâmetro k que estabelece o tamanho do ranking a ser gerado. Uma consequência evidente dessa abordagem é que, se $k < P$ (sendo P a quantidade de elementos a serem inspecionados), o ranking gerado pelo LLM não garantirá a inclusão de todos os elementos relevantes. Além disso, devido

[2] <https://www.langchain.com>

à natureza generativa dos modelos, existe o risco de o LLM repetir elementos de código ou até gerar conteúdo que não existe no código original.

Nesse contexto, propõe-se uma abordagem de combinação de rankings baseados em dados de cobertura de fluxo de controle, obtidos por heurísticas tradicionais, como a heurística Ochiai, com os rankings gerados pelas estratégias baseadas em LLM. Espera-se que essa combinação permita: a) superar a limitação de tamanho dos rankings gerados por LLM e b) que as informações de cobertura de fluxo de controle contribuam para a melhoria do desempenho das estratégias baseadas em LLM. O algoritmo utilizado para combinar dois rankings r_1 e r_2 consiste nos seguintes passos:

1. Obtém-se o tamanho máximo entre os rankings;
2. Cria-se uma tabela contendo ambos os rankings r_1 e r_2;
3. Para os elementos de código que não estão presentes em ambos os rankings, atribui-se o valor obtido no passo 1 à coluna de posição no ranking que originalmente não contém o elemento;
4. Para todos os elementos, cria-se uma nova coluna chamada "pontuação final" e preenche-se com a soma das posições do elemento nos dois rankings;
5. Gera-se um novo ranking com base na coluna de pontuação final, utilizando o critério de maior posição de ranking em caso de empates.

9.2 Resultados e Análises

Esta seção apresenta e discute os resultados alcançados, com foco no endereçamento das questões de pesquisa **RQ3**, **RQ4** e **RQ5**. Inicialmente, o desempenho das estratégias de geração de ranking para localização de defeitos é investigado considerando-se conjuntos de programas de acordo com o nível de dificuldade das tarefas de programação associadas. Essa análise leva em consideração tanto o desempenho do ponto de vista da localização de defeitos quanto do custo para utilização de cada uma das estratégias. Depois, analisa-se como e em quais circunstâncias a inclusão da informação de descrição da funcionalidade tem impacto na localização de defeitos. Finalmente, tem-se a avaliação da proposta de utilização de dados de cobertura de fluxo de controle para potencializar os rankings gerados por LLM.

9.2.1 Desempenho das Estratégias de Localização de Defeitos Baseadas em LLM em Função do Nível de Dificuldade

Para investigar o comportamento das estratégias de geração de ranking *fl-top-k* e *fl-it-k*, foi realizada uma série de execuções sobre os conjuntos de dados previamente separados por níveis de dificuldade, conforme descrito na Seção 9.1.2.

Considerando o caráter estocástico dos modelos de linguagem de grande porte (LLMs), ou seja, a possibilidade de uma mesma entrada, com os mesmos parâmetros de execução, resultar em respostas distintas, foram realizadas cinco execuções para cada programa a ser inspecionado. Também foi considerada uma variação no tamanho k do ranking gerado. Todas essas execuções foram realizadas para ambas as linguagens contempladas no conjunto de dados: Python e Java.

Análise da Estratégia *fl-top-k*

A Tabela 9.3 apresenta a quantidade de acertos alcançados por cada configuração de k da estratégia *fl-top-k*, considerando apenas os programas em Python. Além disso, são apresentados os valores de média (M), mediana (MD) e desvio padrão (DP) dos acertos, bem como a quantidade média de tokens consumidos, considerando as cinco execuções. Todos esses dados estão segmentados por nível de dificuldade. O cálculo dos tokens leva em consideração a soma da quantidade de tokens de todas as respostas para os 100 programas selecionados, de um mesmo k, dividida pelo número de execuções, no caso, cinco.

Tabela 9.3: Estatísticas de acertos e tokens gastos por cada configuração *fl-top-k*, considerando cinco execuções por programa Python, para 100 programas de cada nível de dificuldade

				low				**medium**				**high**
k	**M**	**MD**	**DP**	**tokens**	**M**	**MD**	**DP**	**tokens**	**M**	**MD**	**DP**	**tokens**
fl-top-1	37	37	1,22	1.198,8	23,6	23	0,89	1.214,0	9,4	9	0,55	1.432,4
fl-top-2	48,4	49	1,82	2.166,8	36,4	36	2,07	2.213,0	21	21	0,71	2.741,8
fl-top-3	59,4	59	1,14	2.935,4	43,8	44	0,84	3.354,0	23,6	24	1,14	4.106,2
fl-top-4	63	63	0,71	3.682,4	46,2	46	1,30	4.321,4	29,6	30	1,14	5.261,2
fl-top-5	69,4	69	1,67	4.330,6	51,2	51	0,84	5.229,6	33,8	34	2,17	6.282,8

Fonte: Elaborada pelo autor.

Analisando o desempenho para $k = 1$ nos programas de dificuldade *baixa*, observa-se que as cinco execuções produziram, em média, 37 acertos, representando 37% dos programas com defeito. Isso indica que, com um ranking de apenas uma posição, o LLM é capaz de localizar o defeito em mais de um terço dos programas defeituosos, com um desvio padrão de $\pm 1,22$, consumindo em média 1.198,8 tokens, o que equivale a aproximadamente 32,4 tokens gastos por acerto.

Ao se observar os resultados para as variações de k subsequentes, ou seja, o aumento no tamanho do ranking produzido pela estratégia *fl-top-k*, percebe-se que tanto os valores de média quanto de mediana, são consistentemente maiores à medida em que se aumenta o k. Apesar de esperado, esse resultado representa um teste de sanidade, pois,

em tese, não existem garantias prévias de que o LLM gere um ranking com elementos distintos, como acontece com outros métodos de localização de defeitos.

Analisando-se o comportamento da quantidade média de acertos para os conjuntos de programas cujas tarefas são classificadas como *medium* e *high* no tocante ao nível de dificuldade, também verifica-se que para o aumento do tamanho do ranking, tem-se um regular aumento da quantidade acertos. Naturalmente, esse aumento do tamanho do ranking resulta em um aumento na quantidade de tokens trafegados, o que é um atributo relevante no contexto da aplicação de LLMs. Mais adiante, será apresentada uma avaliação mais detalhada sobre o custo-benefício do aumento tamanho do ranking produzido pelo LLM quando leva-se em consideração o custo de tokens.

Outro fator importante a ser destacado nesses resultados é dispersão dos valores de acertos ao longo das cinco execuções, dado pelo valor do Desvio Padrão (DP). Olhando todos os 15 resultados, em apenas dois deles essa dispersão chegou à casa de 2, o que indica um certo nível de estabilidade nas respostas do LLM. Como se sabe, nos LLMs, o parâmetro da temperatura é tipicamente utilizado para controlar o que se conhece como nível de aleatoriedade do modelo. Esse parâmetro, cujo valor varia de 0 a 1, costuma ser ajustado com valor tendendo a 1 quando se pretende deixar o modelo mais "livre" para ser criativo na resposta e próximo a 0 quando se deseja uma resposta mais determinísticas. Na prática, o que se faz com esse parâmetro é controlar o grau de liberdade do modelo em gerar tokens menos prováveis. No casos desses experimentos, todas as execuções foram realizadas com a temperatura igual a 0, o que pode ser um fator que contribua para respostas mais regulares quanto à indicação de elementos suspeito de ser a causa do defeito.

Analisando a estratégia *fl-top-k* para programas em Java, a Tabela 9.4 apresenta as estatísticas de média, mediana e desvio padrão dos acertos das cinco execuções para cada valor de *k*, segmentadas por nível de dificuldade.

Tabela 9.4: Estatísticas de acertos e tokens gastos por cada configuração *fl-top-k*, considerando cinco execuções por programa Java, para 100 programas de cada nível de dificuldade

				low				**medium**				**high**
k	**M**	**MD**	**DP**	**tokens**	**M**	**MD**	**DP**	**tokens**	**M**	**MD**	**DP**	**tokens**
fl-top-1	33,4	33	1,14	1063,6	17,2	17	0,45	1330,4	15,8	15	1,64	1339,4
fl-top-2	50,2	50	1,10	1776,0	24,8	24	1,30	2379,8	21,0	21	1,00	2451,4
fl-top-3	58,8	58	1,64	2454,6	33,2	33	0,45	3239,6	25,0	25	1,22	3529,6
fl-top-4	61,6	62	3,21	2979,2	34,8	35	1,64	4041,2	27,8	28	1,64	4435,2
fl-top-5	66,8	67	1,30	3494,8	40,2	40	0,84	4845,0	31,0	31	1,22	5416,2

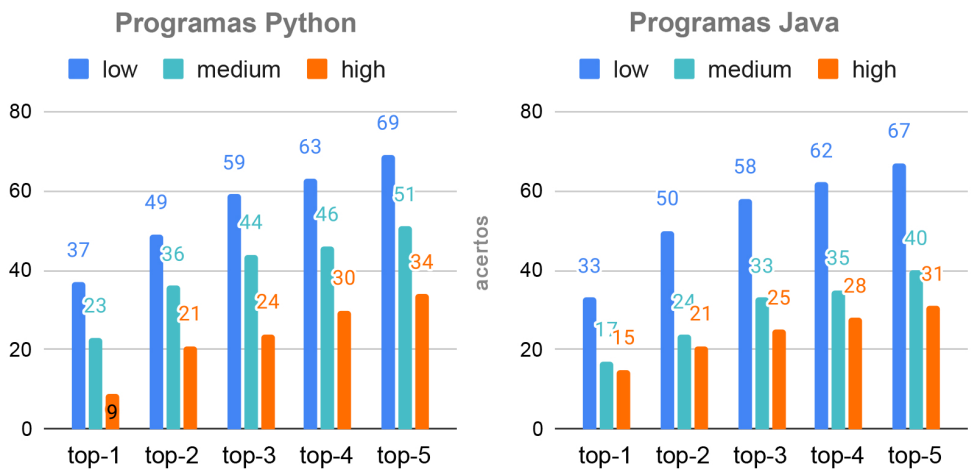
Fonte: Elaborada pelo autor.

Observando os resultados para os programas Java, o comportamento da estraté-

gia *fl-top-k* é similar ao que se verificou em relação aos programas Python. De forma, consistente, ao se aumentar o tamanho do k, tem-se também um aumento na quantidade média de acertos, assim como uma tendência de aumento na média de tokens utilizados, independente do nível de dificuldade. Esses resultados corroboram a noção de que a estratégia de geração de ranking *fl-top-k* possui estabilidade de comportamento independente da linguagem de programação dos programas inspecionados.

Importa notar que para ambas as linguagens, o LLM consegue um desempenho superior dentro do recorte de programas classificados com nível de dificuldade *low*, e esse desempenho vai se deteriorando à medida que o nível de dificuldade aumenta para *medium* e *high*, conforme é ilustrado na Figura 9.1. Essa figura mostra os valores das medianas obtidas para cinco execuções da estratégia *fl-top-k*, variando-se o k de 1 a 5, agrupando-se os acertos por níveis de dificuldade (*low*, *medium* e *high*) e separando-se por linguagem de programação.

Figura 9.1: Desempenho mediano da estratégia *fl-top-k* em cada variação de k, considerando os três níveis de dificuldade.



Fonte: Elaborada pelo autor.

Como nota-se, há uma diferença de desempenho da estratégia de geração de ranking quando se comparam, por exemplo, a mediana dos acertos no conjunto de programas *low* entre Python e Java. Flagrantemente, o LLM consegue produzir rankings mais qualificados quando se trata de programas Python versus programas Java. Sistematicamente, independente do tamanho do ranking, os resultados nos conjuntos *low* e *medium* apresentados para programas Python são sempre superiores àqueles obtidos para os programas Java nas mesmas condições de parâmetros.

Apenas no nível de dificuldade *high*, nos rankings para $k = 1$ e $k = 3$, os resultados para Java superaram os de Python. Contudo, considerando o somatório dos

acertos nos três níveis de dificuldade (*low*, *medium* e *high*), os resultados para programas em Python são superiores em todos os valores de k.

Alguns fatores podem explicar essa diferença de desempenho. Retornando à Tabela 9.1, que apresenta as características do ConDefects, nota-se que, em média, os códigos em Java possuem 259,22 linhas, enquanto os códigos em Python possuem 49,03 linhas. Como a maioria das tarefas possui programas implementados em ambas as linguagens, é razoável afirmar que, dentro desse conjunto de dados, são necessárias mais linhas de código para resolver o mesmo problema em Java comparado a Python, o que é natural dado que Java é uma linguagem mais verbosa. Essa característica pode dificultar o raciocínio do LLM sobre o conteúdo das linhas e suas inter-relações. Além disso, devido à limitação do tamanho do contexto suportado pelo LLM, em diversos casos os programas em Java excederam esse limite, resultando na ausência de geração de ranking para esses programas.

Análise da Estratégia *fl-it-k*

A estratégia *fl-it-k*, detalhada na Seção 7.3, consiste na geração de rankings por meio de sucessivas chamadas ao LLM, de modo que cada chamada subsequente tenha conhecimento das respostas obtidas nas iterações pregressas. Assim como na avaliação da estratégia anterior, foram utilizados os conjuntos de dados separados por nível de dificuldade, dos quais foram selecionados aleatoriamente 100 programas de cada estrato de dificuldade.

A Tabela 9.5 apresenta os resultados de média (M), mediana (MD), desvio padrão (DP) dos acertos e média de tokens consumidos, obtidos a partir de cinco execuções para cada programa, com variações no tamanho do ranking $k = \{1,2,3,4,5\}$, agrupados por nível de dificuldade e linguagem de programação.

Ao analisar os resultados da Tabela 9.5, observa-se que, diferentemente da estratégia *fl-top-k*, na *fl-it-k* o aumento do tamanho do ranking k não implica em um crescimento consistente na quantidade de defeitos localizados. Por exemplo, considerando a mediana (MD) no conjunto *low* para programas em Python, há um aumento de 39 para 44 acertos ao passar de $k = 1$ para $k = 2$. Entretanto, ao comparar $k = 3$ com $k = 4$, não há incremento no número de programas com o defeito identificado, mesmo com uma posição adicional no ranking.

De fato, esse comportamento é ainda mais agudo quando se observa a variação de $k = 4$ para $k = 5$ no conjunto *medium*, para programas Python, onde a mediana dos acertos caiu de 30 para 28, embora nesses casos o Desvio Padrão tenha sido de aproximadamente ± 2, indicando que estatisticamente ambas as distribuições podem ser equivalentes. Esse mesmo comportamento de estagnação (eventualmente, de piora) na quantidade de acertos mesmo quando se aumenta o tamanho do ranking, pode ser notado de $k = 2$ para $k = 3$ do

Tabela 9.5: Estatísticas de acertos e tokens gastos por cada configuração *fl-it-k*, considerando cinco execuções por programa, para 100 programas de cada nível de dificuldade e as duas linguagens Python e Java

												Python
				low				**medium**				**high**
k	**M**	**MD**	**DP**	**tokens**	**M**	**MD**	**DP**	**tokens**	**M**	**MD**	**DP**	**tokens**
top-1	39,0	39,0	1,225	1096,0	20,4	20,0	1,1	1225,8	12,6	12,0	1,5	1369,8
top-2	43,8	44,0	1,095	2160,8	25,2	25,0	1,9	2415,0	12,8	13,0	0,8	2755,8
top-3	49,8	50,0	0,837	3236,0	26,2	27,0	2,3	3622,6	13,2	13,0	1,3	4190,8
top-4	49,8	50,0	1,483	4345,6	29,4	30,0	2,2	4827,4	14,2	14,0	1,5	5597,4
top-5	51,2	51,0	1,304	5450,8	28,8	28,0	2,3	6007,8	14,2	14,0	0,4	6983,6
												Java
				low				**medium**				**high**
k	**M**	**MD**	**DP**	**tokens**	**M**	**MD**	**DP**	**tokens**	**M**	**MD**	**DP**	
top-1	35,2	35,0	0,837	1267,6	14,2	14,0	0,8	1269,6	15,4	15,0	0,5	1434,6
top-2	46,0	46,0	1,225	2449,6	16,0	16,0	1,6	2506,4	18,2	19,0	1,3	2921,6
top-3	48,0	48,0	1,581	3552,4	17,8	18,0	0,4	3645,6	19,8	19,0	1,1	4500,6
top-4	50,2	50,0	0,837	4688,6	19,4	20,0	1,9	5052,4	21,2	21,0	0,8	6031,0
top-5	51,2	51,0	1,095	5828,4	21,4	21,0	0,9	6177,6	22,4	22,0	1,1	7492,8

Fonte: Elaborada pelo autor.

conjunto *high*, para códigos Java, indicando que tal desempenho não seria provocado por peculiaridades das linguagens.

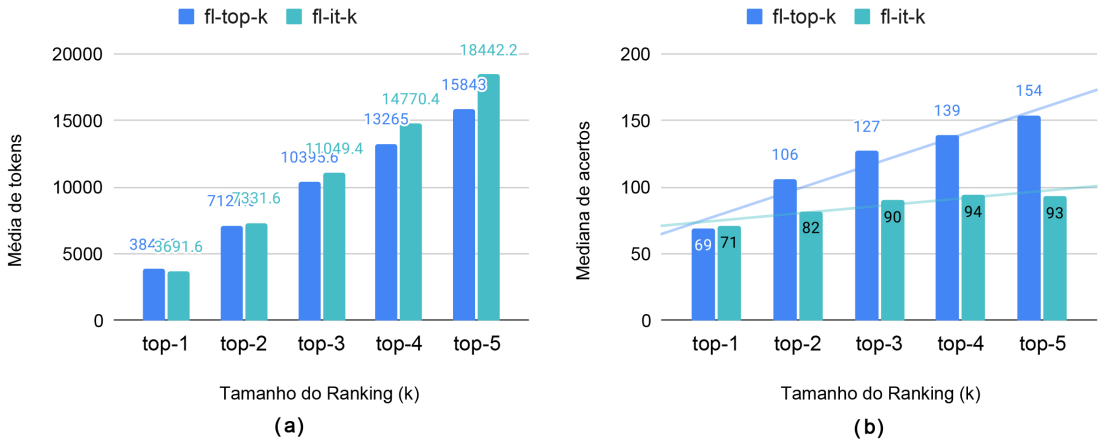
Em relação à variabilidade das respostas, a estratégia *fl-it-k* demonstra um comportamento relativamente estável, com desvios padrão majoritariamente inferiores a 2, independentemente do nível de dificuldade ou da linguagem. Exceções ocorrem nos casos de $k = 3$ a $k = 5$ no conjunto *medium* para programas em Python, onde os desvios padrão atingem valores entre 2,2 e 2,3.

Quanto ao consumo de tokens, observa-se uma regularidade de comportamento, com resultados são compatíveis com o esperado. À medida que se aumenta o tamanho do ranking (k), a quantidade média de tokens também aumenta, independente do nível de dificuldade e da linguagem de programação, o que é natural em função da quantidade de linhas retornadas para cada programa. Quando se comparam os resultados entre as linguagens de programação, tomando-se um mesmo tamanho k, independente do nível de dificuldade, a média de tokens nas respostas para programas Java é sempre superior àquela obtida para os programas Python. Novamente, esse é um comportamento esperado, uma vez que, presumidamente, por ser uma linguagem mais verbosa, as linhas dos programas Java tendem a possuir mais tokens.

Além da avaliação dos resultados da estratégia *fl-it-k* no que se refere às compa-

rações entre as linguagens, os níveis de dificuldade e as variações do tamanho do ranking, é relevante observar como tais resultados se comparam com aqueles obtidos quando utilizada a estratégia *fl-top-k*. Para tanto, a Figura 9.2 apresenta um comparativo entre as estratégias *fl-top-k* e *fl-it-k*, considerando os resultados alcançados para os programas Python sem distinção entre os conjuntos *low*, *medium* e *high*, ou seja, observando-se todos os programas 300 programas selecionados, 100 de cada nível de dificuldade. A Figura 9.2 a) traz os valores das medianas de acertos alcançados nas cinco execuções de cada uma das estratégia à medida em que se varia o tamanho do ranking (*k*). Analogamente, a Figura 9.2 b) apresenta a média de tokens, comparando-se o desempenho de ambas as estratégias quando se varia o tamanho do ranking.

Figura 9.2: Quantidade média de tokens (a) e mediana de acertos (b) obtidos pelas estratégias *fl-top-k* e *fl-it-k* considerando todos os níveis de dificuldade e variando-se o tamanho do ranking, para os programas Python.



Fonte: Elaborada pelo autor.

Conforme ilustrado, ambas as estratégias possuem um comportamento similar no que diz respeito ao crescimento médio da quantidade de tokens consumidos para gerar os rankings, com uma curva de tendência levemente mais acentuada no caso da estratégia *fl-it-k*. A justificativa para essa pequena variação pode estar no fato de que, no casos da estratégia *fl-top-k*, independente do tamanho k, todas as linhas são geradas de uma única vez e existe um mecanismo de validação para evitar posições não preenchidas, o que acaba por inviabilizar todo o ranking. Inspecionando-se os rankings produzidos por ambas as estratégias, foi constatado que existem ocorrências em que a estratégia *fl-top-k* acaba por gerar rankings vazios, diferentemente da estratégia *fl-it-k*, que gera os rankings de forma iterativa e garante que sempre exista conteúdo em todas as posições.

Como já havia sido destacado na análise pormenorizada da estratégia de geração de ranking fl-top-3, essa estratégia possui a capacidade de localizar uma quantidade maior de defeitos à medida que se aumenta o tamanho do ranking, demostrando assim, uma consistência da abordagem proposta. Por outro lado, a Figura 9.2 a) demonstra

que apesar de ser competitiva, até marginalmente superior para $k = 1$, a estratégia *fl-it-k* possui uma curva de tendência com coeficiente de inclinação consideravelmente inferior, demonstrando que o LLM instanciado em conjunto com o algoritmo de iterações não conseguem manter o desempenho quando aumenta-se o k.

A explicação para o desempenho superior de *fl-it-k*, ainda que por pequena margem, para $k = 1$, pode estar na natureza de como cada estratégia implementa a descrição e as instruções do *prompt* submetido ao LLM. No caso da estratégia fl-top-3, como as k linhas suspeitas devem ser retornadas de uma única vez, é utilizada uma descrição que induza o modelo a raciocinar sobre a ordem das linhas mais suspeitas. Já em *fl-it-k*, a descrição sobre a variável retornada é mais direta no sentido de defini-la como o conteúdo responsável pelo defeito.

Esse racional pode explicar o melhor desempenho em $k = 1$, mas não explicaria a degradação (ou pelo menos um aumento menos acentuado) na quantidade de acertos para rankings maiores. Para investigar esse comportamento, foi realizada uma inspeção nas respostas produzidas pela estratégia *fl-it-k* e percebeu-se uma alta taxa de respostas repetidas para diferentes posições no ranking, indicando uma deficiência do modelo em dar a devida atenção às saídas das iterações anteriores. Por exemplo, para os programas Python, do conjunto *low*, a taxa média de conteúdos repetidos dentro dos rankings foi de 63, 137, 222 e 313, respectivamente para $k = [2,3,4,5]$. Obviamente, para quanto maior a taxa de elementos repetido no ranking, menor o benefício de se ter um ranking maior.

Nesse cenário, foi realizado um teste pontual com os modelos GPT-4o e Claude 3.5 para se verificar o comportamento deles comparado ao GPT-3.5 turbo, com respeito à capacidade de levar em consideração informações anteriores e consequentemente diminuir as repetição dentro do ranking. Por restrições de custo financeiro e limitações de políticas de acessos às APIs dos serviços que disponibilizam os referidos modelos, não foi possível a utilização exaustiva desses LLMs. A Tabela 9.6 apresenta os resultados de média, mediana, desvio padrão e taxa de repetição, para o conjunto de dados *low*, considerando a linguagem de programação Python e cinco execução da estratégia *fl-it-3*, ou seja, $k = 3$.

Tabela 9.6: Média, Mediana, Desvio Padrão e Taxa de Repetição atingidos por diferentes modelos, instanciados na estratégia *fl-it-3*, considerando cinco execução dos 100 programas Python classificados como *low*

Estratégia	Média	Mediana	Desvio Padrão	Taxa de Repetição
fl-it-3 (GPT 3.5)	49,8	50	0,837	137
fl-it-3 (GPT 4o)	58,8	59	0,837	62
fl-it-3 (Claude 3.5)	**68,25**	**68**	**0,500**	**25**

Fonte: Elaborada pelo autor.

Como se observa, de uma média de 137 repetições com o modelo GPT 3.5 como

instância de LLM, a estratégia *fl-it-k* passa para uma média 62 quando utilizado o GPT 4o e uma média de 25 para no cenário em que o Claude 3.5 foi a escolha. Em relação ao desempenho nos acertos, saindo de 50 com GPT 3.5, chegando a 59 com GPT 4o e 68 com Claude 3.5, considerando a mediana, obviamente, esses resultados são influenciados pela maior capacidade geral que os dois últimos modelos possuem. De toda forma, a combinação dos dados de taxa de repetição e acertos alcançados pelos modelos superiores é um indício de que a estratégia *fl-it-k* possui um racional coerente, porém mais sensível à capacidade do modelo de seguir as instruções do *prompt*.

Em suma, a comparação entre as estratégias *fl-top-k* e *fl-it-k* evidencia que, embora a *fl-it-k* possa apresentar desempenho competitivo para valores menores de *k*, sua eficácia diminui em rankings maiores, com evidência de que isso tenha relação com a repetição de elementos, quando se utiliza um modelo menos capaz, como é o caso do GPT-3.5. Os dados referentes aos testes com programas em Java corroboram esses achados e estão disponibilizados no Apêndice F, juntamente com as informações utilizadas na elaboração da Figura 9.2.

Análise de Custo-Benefício do Tamanho do Ranking

Nas análises anteriores, demonstrou-se que as estratégias de geração de ranking baseadas em LLM, denominadas *fl-top-k* e *fl-it-k*, são capazes de localizar defeitos em programas implementados nas linguagens Python e Java, considerando diferentes níveis de dificuldade das tarefas de programação. Em ambas as estratégias, como era esperado, à medida que se aumenta o tamanho *k* do ranking, maior tende a ser a quantidade de defeitos localizados.

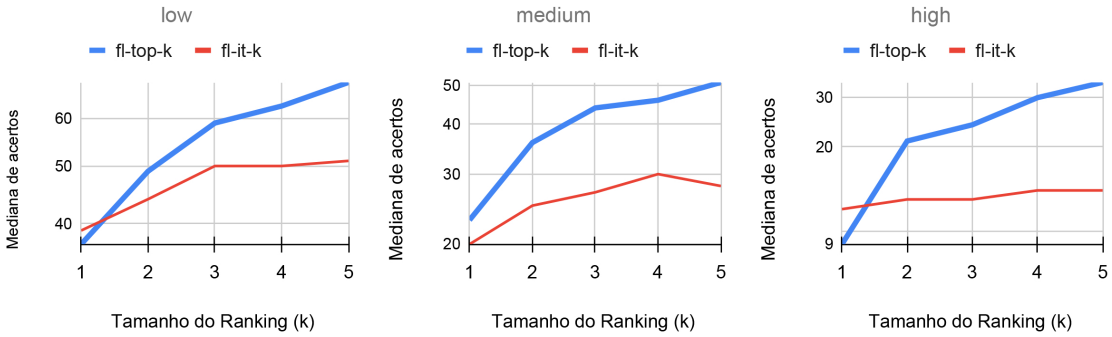
Neste contexto, dois fatores devem ser fortemente considerados:

a) Diferentemente de outros métodos de localização de defeitos que definem o ranking a ser inspecionado com base na suspeição atribuída a cada elemento do código, as estratégias propostas neste trabalho constroem o ranking diretamente a partir do conteúdo dos elementos de código, no caso, as linhas;
b) A criação de rankings mais extensos implica na necessidade de o LLM aplicar um raciocínio de ordenação sobre diversos elementos de código, além de ser obrigado a gerar cadeias mais longas de texto (conteúdo dos elementos de código).

Diante disso, esta seção apresenta uma análise sobre o custo-benefício do aumento no tamanho do ranking, considerando o incremento na capacidade de localização de defeitos e a consequência na quantidade de tokens gerados pelo LLM.

A Figura 9.3 ilustra o desempenho de cada uma das estratégias, considerando a mediana dos acertos obtidos em cinco execuções para 100 programas Python de cada um dos conjuntos *low*, *medium* e *high*, ao longo das variações do tamanho do ranking.

Figura 9.3: Comportamento da mediana dos acertos das estratégias *fl-top-k* e *fl-it-k* para variações de *k* nos diferentes níveis de dificuldade, considerando cinco execuções para cada um dos 300 programas Python.

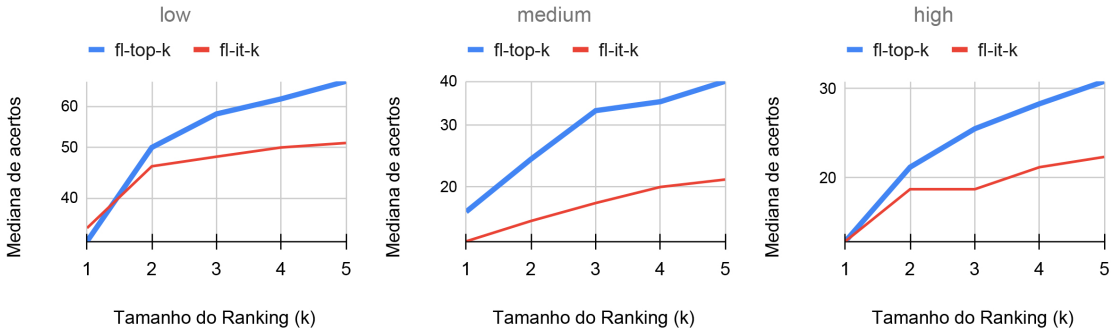


Fonte: Elaborada pelo autor.

Conforme já reportado e discutido anteriormente, a estratégia *fl-it-k*, quando instanciada com o modelo GPT-3.5, o que é o caso deste experimento, de fato, possui baixa capacidade de aumentar a quantidade de acertos na localização de defeitos ao se aumentar o tamanho do ranking. Assim, focando-se no comportamento da estratégia *fl-top-k*, percebe-se que, por uma análise de cotovelo, para os casos dos conjuntos *low* e *medium* o tamanho de ranking 3 representaria uma escolha de bom custo-benefício, supondo um crescimento de tokens próximo a linear. No caso do conjunto *high*, esse ponto crítico seria o ranking de tamanho 2.

Analogamente, a Figura 9.4 apresenta o desempenho das estratégias para programas Java, considerando a mediana dos acertos em cinco execuções sobre 100 programas de cada nível de dificuldade, ao longo das variações do tamanho do ranking.

Figura 9.4: Comportamento de aumento da mediana dos acertos da estratégia *fl-top-k* para variações de k nos diferentes níveis de dificuldade, considerando 5 execuções para cada um dos 300 programas Java.



Fonte: Elaborada pelo autor.

Assim como verificado para os programas Python, a Figura 9.4 mostra que a estratégia *fl-it-k* também não possui uma performance competitiva em nenhum dos con-

juntos após $k = 2$ para tamanho dos rankings. De toda forma, na hipótese da necessidade de escolha de pontos ideais, novamente, por uma análise de cotovelo, ter-se-ia $k = 2$ para os conjuntos *low* e *high*, e 4 para o conjunto *medium*. Já considerando a estratégia *fl-top-k*, o ponto ideal poderia ser $k = 2$ ou $k = 3$ para o conjunto *low*, $k = 3$ para o conjunto *medium* e para o conjunto *high*, apesar de não haver um ponto indubitável, poderia ser $k = 2$ ou $k = 3$.

A análise de cotovelo é útil para a decisão do tamanho do ranking k em casos práticos. Por exemplo, neste trabalho, devido a restrições de recursos, não foi possível testar variações mais amplas em k em investigações que serão apresentadas em seções subsequentes. Portanto, optou-se por utilizar o tamanho $k = 3$, por ter sido a configuração que mais frequentemente apareceu como a melhor escolha nos cenários avaliados.

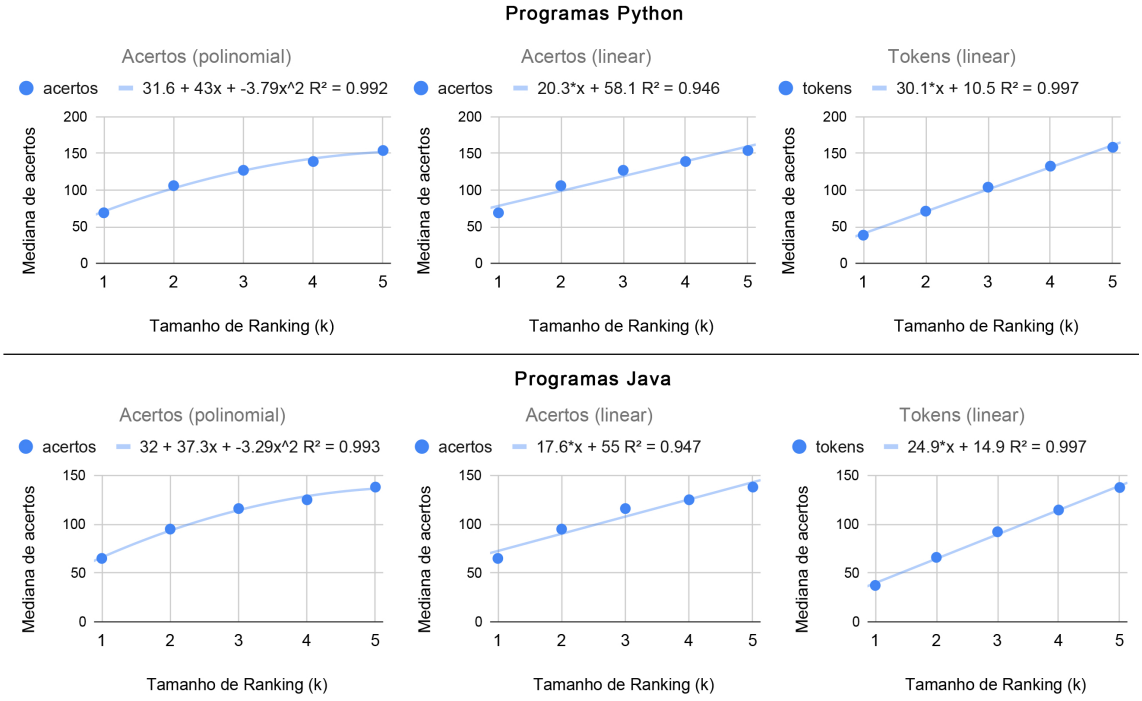
Para complementar e oferecer maior robustez à análise de cotovelo, é importante compreender o comportamento da localização de defeitos e o custo associado em função da variação do tamanho dos rankings. Para isso, aplicou-se uma análise de regressão para verificar como as variáveis dependentes "acertos" e "tokens" se correlacionam com a variável independente k, o tamanho do ranking.

A Figura 9.5 apresenta gráficos de dispersão entre as variáveis "acertos" e "tamanho do ranking", e entre "tokens" e "tamanho do ranking". Nos gráficos das duas primeiras colunas, cada ponto de dados representa o valor mediano obtido em cinco execuções da estratégia *fl-top-k*, referente à quantidade de defeitos localizados em cada valor de k. Na terceira coluna, cada ponto de dados corresponde ao valor médio de tokens utilizados para cada variação de k. A primeira linha de gráficos refere-se aos 300 programas Python selecionados nos três níveis de dificuldade, enquanto a segunda linha apresenta os gráficos para os 300 programas em Java.

Aplicando uma análise de regressão polinomial de grau 2 na avaliação da relação entre k e a mediana dos acertos (primeira coluna), obteve-se um coeficiente de determinação $R^2 = 0,99$, indicando que a variância dos dados é quase completamente explicada pela equação polinomial apresentada. O comportamento é muito similar entre as equações para os programas Python (primeira linha) e Java (segunda linha). Sabe-se que uma função polinomial de grau 2 com concavidade voltada para baixo tenderá a valores menores após atingir seu máximo, o que sugere que os acertos não continuarão a crescer indefinidamente com o aumento de k.

Por outro lado, olhando para os gráficos da última coluna, constata-se uma correlação linear entre o tamanho do ranking e o volume de tokens consumidos, com uma equação de regressão linear explicando 99% da dispersão dos dados. Sendo assim, por óbvio, existe uma expectativa que o custo de aumento do tamanho do ranking tenda a aumentar para aumentos no tamanho do ranking, enquanto o benefício, ou seja, os acertos na localização de defeitos, pode aumentar a uma taxa mais alta, em função do aumento

Figura 9.5: Modelagem por análise regressiva do comportamento de crescimento dos acertos e do custo de tokens da estratégia *fl-top-k*, considerando a separação entre programas Python e Java.



Fonte: Elaborada pelo autor.

no tamanho do ranking, mas se estabilizar e até diminuir a partir de um certo k.

Considerando que o número de pontos de dados (ou seja, o tamanho amostral de cinco) é relativamente pequeno, é possível que a equação polinomial de grau 2 esteja superajustada aos dados, comprometendo a generalização dessa modelagem. Portanto, na segunda coluna, modelou-se a relação entre os acertos e o tamanho k por meio de uma regressão linear. Nesse caso, o coeficiente de determinação caiu para 94% tanto para Python quanto para Java. Ainda assim, ao comparar as equações lineares que explicam os dados de acertos e tokens em função do tamanho do ranking, percebe-se que o coeficiente angular da equação referente aos tokens é maior, indicando uma reta com inclinação mais acentuada.

Em suma, tanto ao modelar o comportamento dos acertos na localização de defeitos como uma regressão quadrática quanto como linear, em função do tamanho do ranking, há indícios de que o benefício adicional de aumentar k diminui em relação ao custo crescente de tokens consumidos.

Diante do exposto, considerando a **RQ3: Como se comportam as estratégias de localização de defeitos baseadas em LLM para programas com diferentes níveis de complexidade?**, fica evidente que ambas as estratégias *fl-top-k* e *fl-it-k*, independentemente da linguagem de programação (Python ou Java), apresentam um desempenho que tende a ser inversamente proporcional ao nível de dificuldade da tarefa. Em termos gerais,

a estratégia *fl-top-k* mostrou-se mais estável na correlação entre o tamanho do ranking k e a capacidade de localizar defeitos do que a *fl-it-k*, quando ambas são instanciadas com o GPT-3.5 Turbo. Além disso, o crescimento esperado no custo de tokens é maior do que o aumento esperado no número de acertos na localização de defeitos à medida que se aumenta o tamanho do ranking.

Esta análise evidencia a importância de equilibrar o tamanho do ranking com o custo associado ao uso de LLMs. Embora rankings maiores possam aumentar a chance de localizar defeitos, o custo em termos de tokens (e, consequentemente, recursos computacionais e financeiros) cresce linearmente, enquanto os ganhos em acurácia tendem a se estabilizar ou até diminuir. Portanto, identificar um ponto de equilíbrio é crucial para a aplicação prática dessas estratégias em diferentes contextos e níveis de complexidade.

9.2.2 Impacto da Inclusão da Descrição da Funcionalidade

Conforme descrito na Seção 9.1.1, o ConDefects não possui a informação de descrição das tarefas de programação para as quais os programas Python e Java foram implementados. Para suprir essa lacuna, as descrições foram obtidas via *web scraping* na plataforma de onde os programas foram extraídos quando da composição do conjunto de dados original do ConDefects. Embora o processo de captura das descrições tenha sido cuidadosamente implementado e tenha havido uma verificação manual de amostras das descrições extraídas, os dados obtidos não foram validados por pares, ao contrário do que ocorre com o conjunto de dados original do ConDefects.

Como mencionado anteriormente, devido a limitações de recursos, o teste envolvendo o acréscimo da descrição como informação adicional para auxiliar no processo de localização de defeitos utilizando LLM foi restrito a uma única estratégia de geração de ranking e a um único tamanho de ranking. Com base em análises anteriores, optou-se pela utilização da estratégia *fl-top-k*, por sua estabilidade de desempenho, escolhendo-se $k = 3$ por ser uma configuração com bom custo-benefício em relação ao consumo de tokens.

A Tabela 9.7 apresenta uma comparação entre duas versões da estratégia *fl-top-3*: uma que inclui a descrição da funcionalidade (*fl-top-3-desc*) e outra que não a inclui. A diferença entre elas reside na presença ou não da descrição da funcionalidade como uma das informações consideradas pelo LLM. Foram coletados os dados de média (M), mediana (MD) e desvio padrão (DP) relativos à quantidade de programas que tiveram a localização do defeito dentro do ranking, equivalente à métrica ACC@3. As avaliações consideraram cinco execuções de cada estratégia, separando-se os programas por nível de dificuldade e linguagem de programação.

Analisando os resultados relativos aos programas em Python, observa-se que, ao considerar a média acumulada (desconsiderando a separação por nível de dificuldade),

Tabela 9.7: Valores de Média (M), Mediana (MD) e Desvio Padrão (DP) para cinco execuções da estratégia *fl-top-3* considerando ou não a informação de descrição, para cada nível de dificuldade e linguagem de programação

											Python
	low			**medium**			**high**				
	M	MD	DP	M	MD	DP	M	MD	DP	Total M	Total MD
fl-top-3	59,4	59	1,14	43,8	44	0,84	23,6	**24**	1,14	126,8	127
fl-top-3-desc	63.2	**63**	1,48	45,2	**45**	1,3	19,2	19	1,48	**127,6**	127
											Java
	low			**medium**			**high**				
	M	MD	DP	M	MD	DP	M	MD	DP	Total M	Total MD
fl-top-3	58,8	58	1,64	33,2	33	0,45	25	25	1.22	117	116
fl-top-3-desc	62.2	**61**	1,79	37,8	**38**	0,84	24,8	25	1.1	**124,8**	**124**

Fonte: Elaborada pelo autor.

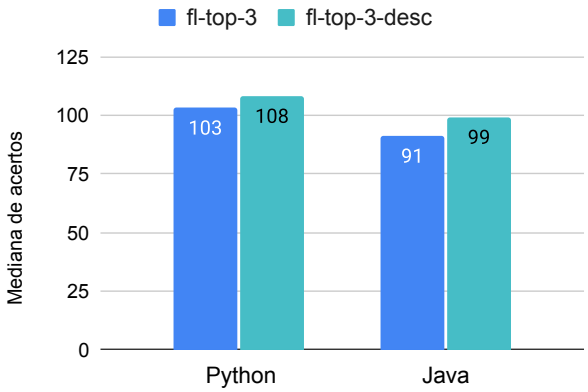
houve um aumento marginal de acertos com a inclusão das descrições das funcionalidades, passando de 126,8 para 127,6. Contudo, ao examinar os valores de mediana, ambas as variações da estratégia *fl-top-3* obtiveram o mesmo valor, 127 acertos. Observando a estratificação por nível de dificuldade, percebe-se que, para os conjuntos *low* e *medium*, a estratégia com descrição obteve desempenho superior tanto em média quanto em mediana. Entretanto, para o conjunto *high*, a estratégia sem descrição foi ligeiramente superior, resultando em um empate na mediana e uma melhoria marginal na média.

Nos dados referentes aos programas Java, considerando os valores acumulados de média e mediana, as estratégia que adota a descrição foi superior, saindo de uma média de 117 para 124,8 e na mediana de 116 para 124. Contudo, novamente, o melhor desempenho é observado para os conjuntos *low* e medium. Para o conjunto *high*, houve um empate em termos medianos e uma piora de 0,2 na média.

Esses resultados corroboram a ideia de que, para problemas cujo nível de dificuldade não seja *high*, a inclusão da descrição na estratégia *fl-top-3* melhora a quantidade de programas para os quais os defeitos foram apontados dentro de um ranking com três elementos. A Figura 9.6 ilustra esse resultado, evidenciando que, nos conjuntos *low* e *medium*, a estratégia com descrição adicionou cinco programas em Python, passando de 103 para 108 acertos (um acréscimo de 4,8%), e oito programas em Java, aumentando de 91 para 99 acertos (um acréscimo de 8,7%).

Em relação ao desempenho da estratégia *fl-top-3* não ter se beneficiado da descrição e, no caso dos programas em Python, até apresentar uma piora quando se refere ao conjunto *low*, a explicação pode estar relacionada à natureza das descrições capturadas. Uma das partes que compõem as instruções para a realização das tarefas na plataforma de origem é a seção *Constraints*. Devido à forma como os símbolos matemáticos são

Figura 9.6: Mediana de acertos nos conjuntos *low* e *medium* considerando cinco execuções da estratégia *fl-top-3* com e sem descrição, para programas Python e Java.



Fonte: Elaborada pelo autor.

renderizados na página web, o processo de *scraping* não foi capaz de extraí-los de maneira legível, optando-se por não incluir essa parte no conteúdo da descrição. É possível que essa informação ausente seja mais relevante para os programas do conjunto *high*, que tendem a ser mais complexos e dependentes de restrições específicas.

Outro aspecto a ser observado nos dados da Tabela 9.7 é o comportamento da variância dos resultados produzidos por cada variação da estratégia *fl-top-3*. Analisando os valores de desvio padrão, dentro de cada conjunto (*low*, *medium* e *high*), a versão contendo a descrição apresentou sempre um valor maior nos programas em Python, e apenas no conjunto *high* dos programas em Java não foi superior. Sabe-se que um dos fatores que podem influenciar na diversidade das respostas de um LLM é o parâmetro de temperatura; entretanto, neste teste, ambas as variantes de *fl-top-3* utilizaram a mesma configuração. Portanto, outras explicações são possíveis. Por exemplo, o tamanho e a complexidade do *prompt* enviado ao LLM podem contribuir para a diversidade de respostas. Embora os *prompts* utilizados possuam instruções claras e diretas sobre como o modelo deve construir o raciocínio, a inclusão da descrição adiciona mais informações às quais o modelo deve prestar atenção, aumentando a chance de variação na composição da resposta.

Diante do exposto, pode-se responder à **RQ4: Qual o impacto da inclusão da descrição da funcionalidade na localização de defeitos baseada em LLM?** Nos cenários avaliados, a incorporação da descrição da funcionalidade teve um impacto positivo, melhorando o desempenho da estratégia de localização de defeitos baseada em LLM para programas em Python (melhoria de aproximadamente 5%) e Java (melhoria de quase 9%) nos conjuntos *low* e *medium*. Além disso, as versões da estratégia que consideraram a descrição apresentaram maior variabilidade nas respostas em comparação com as versões sem descrição, possivelmente devido ao aumento de informações e ao

volume do *prompt* submetido ao LLM.

Este estudo evidencia que a inclusão da descrição da funcionalidade pode melhorar o desempenho de estratégias de localização de defeitos baseadas em LLM, especialmente em programas de menor complexidade. No entanto, a ausência de partes críticas das descrições, como as restrições específicas em problemas mais complexos, pode limitar esse benefício. Além disso, o aumento na variabilidade das respostas indica que a inclusão de mais informações no *prompt* pode influenciar a consistência do modelo, aspecto que merece atenção em aplicações práticas.

9.2.3 Utilização de Dados de Cobertura de Fluxo de Controle

Conforme discutido na Seção 7.3, uma limitação das estratégias de geração de rankings utilizadas neste experimento é o parâmetro k, que estabelece o tamanho máximo do ranking. No pior cenário, a quantidade de elementos a serem inspecionados pode ser maior que k e, dependendo do desempenho da estratégia, o elemento defeituoso pode não estar dentro desse ranking limitado.

Nesta seção, é avaliado como as informações de cobertura de fluxo de controle, obtidas a partir da execução de casos de teste sobre os programas defeituosos, podem melhorar o desempenho das estratégias de localização de defeitos baseadas em LLM. Para essa avaliação, foram utilizados programas em Python e Java, considerando o recorte de programas de 2024, conforme definido na Seção 9.1.2.

A Tabela 9.8 apresenta os resultados para as métricas ACC@N e *Exam* para 280 versões de programas em Python selecionadas aleatoriamente do conjunto ABC. Foram consideradas duas heurísticas tradicionais, Ochiai e Tarantula, e três estratégias de localização de defeitos baseadas em LLM. Para cada uma dessas estratégias, também foi avaliada uma versão combinada com a heurística Ochiai. O tamanho do ranking adotado para as estratégias *fl-top-k* e *fl-it-k* foi $k = 3$, por ter apresentado o melhor custo-benefício em análises anteriores. Em relação à estratégia *fl-ms*, baseada na combinação de modelos, os modelos escolhidos para compor a resposta final foram as estratégias *fl-top-3* e *fl-it-3*.

Observando os resultados da Tabela 9.8, nota-se que as heurísticas Ochiai e Tarantula apresentam desempenhos semelhantes em ACC@N e *Exam*. No entanto, os rankings baseados em Ochiai obtiveram resultados ligeiramente superiores em ACC@3 e *Exam*, razão pela qual Ochiai foi escolhida para as combinações com as estratégias baseadas em LLM.

A estratégia *fl-top-3* demonstra desempenho consistentemente superior às heurísticas tradicionais em ACC@1 até ACC@5. Em ACC@10, porém, ambas as heurísticas superam a *fl-top-3*. Isso ocorre porque o ranking gerado pela *fl-top-3* possui um tamanho máximo de 3, de modo que os valores de ACC@N para $N >= 3$ permanecem constantes.

Tabela 9.8: Resultados de ACC@N e Exam para diferentes estratégias de localização de defeitos sobre 280 versões Python do conjunto ABC.

	ACC@ 1	ACC@ 2	ACC@ 3	ACC@ 5	ACC@ 10	Exam
Ochiai	17	31	52	78	145	0,506
Tarantula	17	31	50	79	144	0,514
fl-top-3	57	94	114	114	114	-
fl-top-3+Ochiai	55	96	117	127	174	0,352
fl-it-3	72	84	96	96	96	-
fl-it-3+Ochiai	**74**	86	99	110	156	0,339
fl-ms	72	99	125	133	134	-
fl-ms+Ochiai	71	**107**	**130**	**142**	**179**	**0,322**

Fonte: Elaborada pelo autor.

Além disso, as heurísticas tradicionais geralmente produzem rankings com empates, o que aumenta a probabilidade de o defeito estar incluído em posições de ranking maiores, favorecendo valores mais altos de N em ACC@N.

Para aprimorar o desempenho da *fl-top-3*, foi adotada a abordagem *fl-top-3+Ochiai*, que é uma combinação dos rankings produzidos pelo LLM e pela Ochiai. Nesse caso, o resultado para ACC@1, caiu de 57 para 55, porém, em todos os outros valores de N, o ranking combinado foi superior à estratégia *fl-top-3*. Vale destacar que, diferentemente do ranking original, o ranking de *fl-top-3+Ochiai* possui a garantia de cobrir todas as linhas do programa, então, o desempenho para os valores de ACC@N com $N > 3$ passa a ser consideravelmente superior. Outra consequência positiva dessa combinação é a possibilidade de cálculo da métrica Exam. Como se observa, com os rankings das heurísticas Ochiai e Tarantula, seria necessário inspecionar-se em média 50% dos elementos de código até se encontrar o defeito. Olhando para a abordagem *fl-top-3+Ochiai*, a inspeção seria em média de 35% dos elementos de código suspeitos.

A estratégia *fl-it-3* apresenta desempenho superior à *fl-top-3* em ACC@1, mas é consistentemente inferior nos demais valores de *N*. Esse comportamento pode ser explicado pelo fato de a *fl-it-3* ser mais eficaz na identificação do defeito na primeira posição do ranking, mas sua eficácia diminui em posições subsequentes devido à repetição de linhas já sugeridas em iterações anteriores, conforme discutido anteriormente. No entanto, a versão combinada *fl-it-3+Ochiai* mantém a melhoria sobre a versão apenas do LLM, tanto em ACC@N quanto em *Exam*, reforçando a vantagem de incorporar dados de cobertura.

Destaca-se que a estratégia *fl-it-3+Ochiai* obteve o melhor resultado em ACC@1 (74 acertos). Isso ocorre porque a *fl-it-3* tende a identificar corretamente o defeito na primeira posição com maior frequência, e a combinação com Ochiai mantém essa característica, beneficiando-se da informação adicional sem prejudicar a eficácia inicial.

A estratégia *fl-ms*, que combina as respostas de múltiplos modelos, supera todas as outras estratégias em ACC@N para $N > 1$, incluindo as versões combinadas com Ochiai. Uma característica distintiva da *fl-ms* é que não se limita a um ranking de tamanho 3, pois agrega as respostas das estratégias *fl-top-3* e *fl-it-3*. Consequentemente, mesmo para valores maiores de N, como ACC@5 e ACC@10, a quantidade de acertos continua aumentando. Por exemplo, para ACC@3 foram 125 acertos, aumentando para 133 em ACC@5 e 134 em ACC@10. Isso ocorre porque, ao combinar as respostas das duas estratégias, o ranking resultante inclui mais linhas distintas, ampliando a cobertura.

A combinação da *fl-ms* com Ochiai (*fl-ms+Ochiai*) também apresentou melhorias significativas. Embora tenha havido uma ligeira redução em ACC@1 (de 72 para 71 acertos), os ganhos nos demais valores de *N* foram substanciais, culminando no melhor desempenho em *Exam* (0,322) dentre todas as estratégias avaliadas. Isso indica que a integração de informações estáticas (código-fonte analisado pelo LLM) com informações dinâmicas (dados de cobertura utilizados pela Ochiai) potencializou a eficiência na localização de defeitos.

Passando à análise da contribuição dos dados de cobertura de fluxo de controle às estratégias baseadas em LLM, no contexto de programas Java, a Tabela 9.9 apresenta as métricas ACC@N e Exam para 184 programas Java considerando a aplicação das heurísticas Ochiai e Tarantula, assim como as três estratégias de geração de ranking baseadas em LLM e para cada uma delas a variação que combina o ranking do LLM com o ranking da Ochiai.

Tabela 9.9: Resultados de ACC@N e Exam para diferentes estratégias de localização de defeitos sobre 184 programas Java do conjunto ABC.

	ACC@1	**ACC@2**	**ACC@3**	**ACC@5**	**ACC@10**	**Exam**
Ochiai	9	13	21	27	65	0,303
Tarantula	9	13	20	27	62	0,305
fl-top-3	49	65	82	82	82	-
fl-top-3+Ochiai	45	63	84	89	99	0,230
fl-it top3	51	58	61	61	61	-
fl-it top3+Ochiai	**56**	61	63	70	86	0,26
fl-ms	51	72	84	89	89	-
fl-ms+Ochiai	51	**74**	**88**	**95**	**103**	**0,225**

Fonte: Elaborada pelo autor.

Os resultados alcançados para o conjunto de programas Java, no que diz respeito ao desempenho das estratégias baseadas em LLM e como os dados de cobertura podem melhorá-las, acabam por replicar os achados identificados na análise sobre os programas Python. Dentre as similaridades, destaque-se que, novamente, os melhores resultados de ACC@N, para $N > 1$, e Exam foram alcançados pela abordagem fl-ms+Ochiai,

reforçando a ideia de que, de fato, os dados podem contribuir para a melhoria do desempenho de estratégias localização de defeitos baseadas em LLM; o comportamento da estratégia *fl-it-3* também se replicou, no sentido de obter o melhor resultado para ACC@1, mas se deteriorar para N maior; e, em resumo, para todas os casos em que a Ochiai foi combinada com a estratégia LLM, houve uma redução no valor de Exam quando se comparado ao Exam das heurísticas, assim, a combinação favorece a obtenção de um ranking que cubra todos os elementos de código, melhora o desempenho na maior parte dos N do ACC@N e diminui a porcentagem média de elementos a serem inspecionados até que se encontre o defeito.

Por outro lado, há algumas nuances. Por exemplo, nos resultados dos programas Python, na Tabela 9.8, quando se comparava o ACC@10 da estratégia *fl-top-3* com o resultado de Ochiai e Tarantula, as heurísticas eram superiores, obtendo 145 e 144 acertos, respectivamente, contra 114 da estratégia baseada em LLM. Para os programas Java, a *fl-top-3* é superior às heurísticas em ACC@10, mesmo só possuindo um ranking de tamanho 3. Esse comportamento diverso ocorre porque, na verdade, a proporção de acertos da Ochiai dentro do ACC@10 em relação à quantidade de versões Python disponíveis (145 de 280 equivale a 51%) é superior a proporção de acertos em ACC@10 da mesma heurística em relação à quantidade de versões Java (65 de 184 equivale a 35%). A explicação para essa proporção diferente está no fato de que, sendo os programas Python, em média, menores do que os programas Java, uma inspeção em 10 linhas tem mais chance de cobrir um percentual maior do programa sob avaliação.

Retomando a **RQ5: Qual o impacto dos dados de cobertura e como o LLM se compara com heurísticas tradicionais de localização de defeitos?** os dados apresentados nesta seção indicam que a combinação de rankings produzidos por heurísticas tradicionais com estratégias baseadas em LLM é benéfica. Essa combinação permite a construção de um ranking completo, superando a limitação imposta pelo parâmetro k. Em termos de desempenho medido pela métrica ACC@N, as versões combinadas apresentaram melhorias em relação às versões originais, especialmente para $N > 1$, tanto em programas Python quanto em Java. No que se refere à métrica *Exam*, independentemente da linguagem de programação, as estratégias que combinam informação estática (código-fonte analisado pelo LLM) com informação dinâmica (dados de cobertura utilizados pelas heurísticas) obtiveram resultados superiores às heurísticas tradicionais isoladas.

Esses resultados evidenciam que os dados de cobertura de fluxo de controle podem contribuir para o aprimoramento das estratégias de localização de defeitos baseadas em LLM, tornando-as mais acuradas em comparação com abordagens tradicionais. A integração de informações de cobertura de fluxo de controle permite que os LLMs superem limitações inerentes às estratégias baseadas apenas em análise estática do código, oferecendo uma abordagem mais robusta para a identificação de defeitos em programas de

diferentes níveis de complexidade e em diversas linguagens de programação.

9.3 Comparação com o Estado da Arte

Conforme apresentado no Capítulo 3, entre os trabalhos relacionados, a abordagem denominada LLMAO, proposta por Yang et al. (2024), é a técnica publicada após revisão por pares que reportou os melhores resultados do uso de LLMs para a localização de defeitos em nível de linhas de código. Portanto, esta seção apresenta uma breve comparação da aplicação da técnica LLMAO com a estratégia de geração de ranking *fl-top-5+Ochiai*, em condições peculiares que serão discutidas a seguir.

Diferentemente dos demais LLMs utilizados neste trabalho, o LLMAO não se encontra, pelo menos até o último esforço de pesquisa desta tese, disponível em qualquer uma das plataformas que tipicamente servem modelos de linguagem de larga escala. Isso implica na necessidade de acesso a uma infraestrutura de processamento baseada em GPU. Além disso, de acordo com o artigo que introduz o modelo, foram realizados processos de *fine-tuning* nas linguagens C, Java e Python. No entanto, no repositório oficial da ferramenta[3] não estão disponíveis quaisquer versões dos pesos do modelo alegadamente treinado para a linguagem Python.

Diante dessas circunstâncias, foi realizada uma comparação limitada a partir da execução da ferramenta LLMAO em uma GPU Nvidia Titan XP de 12 GB. Em função da limitação de espaço, utilizou-se o menor *checkpoint* disponível no repositório (350M), com requisito de 2,5 GB e apenas para a linguagem Java. Devido ao elevado tempo de inferência, foram selecionados, aleatoriamente, cinco programas Java dentro de cada um dos conjuntos *low*, *medium* e *high*. Para esses 15 programas, aplicou-se o LLMAO e uma das combinações utilizadas anteriormente neste experimento, a estratégia *fl-top-5+Ochiai*.

Tabela 9.10: Posições em que os defeitos foram localizados pela estratégia *fl-top-5+Ochiai* (LLM+H) e LLMAO considerando cinco programas Java de cada nível de dificuldade

		low			**medium**			**high**
Id	**LLM+H**	**LLMAO**	**Id**	**LLM+H**	**LLMAO**	**Id**	**LLM+H**	**LLMAO**
39120071	9	2	33790374	**2**	23	40807993	**18**	34
34641363	**39**	97	32678976	*	11	54846225	**19**	33
40606913	**8**	9	38311271	**	**	52374275	**14**	36
41127680	**1**	18	52685505	19	4	48155601	*	144
32809318	**12**	17	35459787	**6**	86	32294543	*	7

Fonte: Elaborada pelo autor.

[3] <https://github.com/squaresLab/LLMAO>

Os valores em negrito representam os casos em que, na comparação com o LLMAO, a abordagem proposta neste estudo foi superior, ou seja, encontrou o defeito em uma posição do ranking inferior à posição em que o LLMAO localizou o defeito. Observa-se que, no caso dos programas do conjunto *low*, em 4 dos 5 cenários, a proposta LLM+H foi superior. Nos programas do conjunto *medium*, dos 3 casos em que foram possíveis as comparações, a LLM+H foi superior em duas ocasiões. Finalmente, para os programas do conjunto *high*, nos três cenários passíveis de comparação, a técnica proposta foi superior em todas as ocorrências.

Os casos marcados com asteriscos são situações nas quais não foi possível obter os dados necessários para a produção do ranking LLM+H. Nos casos marcados com (*), houve problemas no processo de execução dos casos de teste e/ou geração da matriz de cobertura de fluxo de controle. Isso representa uma vantagem para o LLMAO, que dispensa a necessidade de informações de execução dos casos de teste. Já no caso marcado com (**), após uma inspeção mais detalhada nos metadados do programa dentro do ConDefects, percebeu-se que a indicação do local defeituoso estava apontando para a linha 0, circunstância que impossibilita a recuperação do conteúdo defeituoso e inviabiliza o uso da técnica LLMAO.

Inspecionando as respostas produzidas pelo LLMAO, ou seja, os rankings de suspeitas gerados para cada um dos programas testados, foi verificada que muitas das linhas possuem a suspeita como 0. Por exemplo, o Código 9 apresenta o ranking gerado pelo LLMAO para o programa Java de identificador 32809318. Como pode ser visto, das 17 linhas consideradas elegíveis para inspeção, segundo o método do LLMAO, apenas cinco tiveram uma suspeita diferente de zero. Essa característica é muito relevante pois, nesse caso particular, o defeito estava localizado na linha 9 (line-8, devido a forma de indexação adotada). Assim, ao se utilizar o critério de pior posição para os casos de empate, o local seria anotado como localizado apenas ao se inspecionar as 17 linhas.

Naturalmente, a abordagem LLM+H não possui esse problema porque o próprio algoritmo de combinação de rankings já define as posições finais com base em rankings anteriores. No caso específico do ranking produzido apenas por estratégia baseadas em LLM, também não haveria esse cenário, pois os LLMs respondem o conteúdo e não a suspeita da linha. De toda forma, o efeito deletério dos empates no ranking final da estratégia LLM+H ainda por acontecer por influência do componentes baseado na Ochiai.

Para analisar o quanto a combinação com Ochiai poderia ser influenciada por valores zerado no ranking dessa heurística, foi computada a mediana do percentual de linhas para as quais o LLMAO e o ranking Ochiai geraram uma suspeita diferente de zero, considerando os cinco programas de cada conjunto. Os valores coletados relevam que o LLMAO gerou 14% para o conjunto *low*, 36% para o *medium* e 26% para o *high*. Já os rankings Ochiai atingiram 40%, 26%, 51%, respectivamente para *low*, *medium* e

Código 9: Saída produzida pelo LLMAO para o programa Java 32809318.

```
line-16 sus-0.09%:             for(int i=0; i<n+1; i++) {
line-3 sus-0.07%:     public static void main(String[] args) {
line-14 sus-0.06%:             Arrays.sort(a);
line-15 sus-0.04%:             int ans = 0;
line-2 sus-0.03%: class Main {
line-0 sus-0.0%:           System.out.println(ans);
line-1 sus-0.0%: import java.util.*;
line-4 sus-0.0%:         try(Scanner sc = new Scanner(System.in)) {
line-5 sus-0.0%:             int n = sc.nextInt();
line-6 sus-0.0%:             int[] a = new int[n+2];
line-7 sus-0.0%:             a[n] = 0;
line-8 sus-0.0%:             a[n+1] = 0;
line-9 sus-0.0%:             int s = 0;
line-10 sus-0.0%:             for(int i=0; i<n; i++) {
line-11 sus-0.0%:                 int nxt = sc.nextInt();
line-12 sus-0.0%:                 s += nxt;
line-13 sus-0.0%:                 a[i] = s%360;
```

high. Ou seja, apenas no conjunto *medium* os rankings Ochiai produziram mais linhas com suspeita 0.

Importante destacar que a maior proporção de linhas com suspeita 0 dos rankings do LLMAO não necessariamente se relacione com pior desempenho na localização de defeitos, pois uma linha com suspeita 0 significa que, para o modelo, aquela linha não tem chance alguma de ser o potencial local do defeito. Isso é verdade porque o LLMAO utiliza um classificador com função probabilística sigmoide para indicar que quanto mais próximo de 1 maior a chance de a linha se o local do defeito e quanto mais perto de 0 menor a chance de a linha ser o local do defeito.

Nesse sentido, para o caso em questão, nota-se que a linha 17 (anotada como line-16), tida como a linha de maior suspeita, tem uma suspeita de 0,09. Esse valor, apesar de sinalizar a primeira posição do ranking, do ponto de vista do quanto o modelo reconhece o conteúdo da referida linha como um defeito é pouco significativo. Esse comportamento pode ter relação com o quão adequado estaria o modelo aos dados a ele submetidos. Conforme foi explicado na Seção 3, o LLMAO utiliza um LLM e aplica um fine-tune para três linguagens de programação, incluindo Java. Esse cenário pode provocar uma situação em que o modelo resultante consiga ter um bom desempenho no cenário específico no qual foi refinado, mas não possa ser generalizado para programas desconhecidos do modelo durante o processo.

Em suma, embora a comparação tenha sido limitada devido às restrições técnicas e de disponibilidade do modelo LLMAO, os resultados sugerem que a estratégia *fl-top-*

5+Ochiai (LLM+H) apresenta desempenho competitivo, superando o LLMAO na maioria dos casos analisados. Isso indica o potencial das estratégias baseadas em LLMs combinadas com heurísticas tradicionais de localização de defeitos, especialmente quando se considera a facilidade de implementação e a menor dependência de recursos computacionais específicos.

9.4 Limitações e Ameaças à Validade

As estratégias propostas para a geração de rankings de suspeita, introduzidas e avaliadas neste experimento, são fundamentalmente dependentes do desempenho do LLM instanciado. Observa-se que tem ocorrido uma profusão de modelos de linguagem nos últimos anos, tornando praticamente impossível determinar com clareza qual deles é o mais capaz. Notadamente, existem modelos mais ou menos adequados dependendo do contexto da aplicação ou da complexidade da tarefa. Nesse sentido, os experimentos apresentados possuem a limitação do uso predominante de um único modelo. Apesar de amplamente utilizado, o GPT-3.5 Turbo é um modelo proprietário, sobre o qual não se conhecem todos os detalhes, uma vez que seu processo de treinamento não é totalmente divulgado. Embora o foco desta pesquisa não seja a avaliação de LLMs, o emprego de outros modelos, incluindo aqueles treinados especificamente para tarefas de código, poderia conferir maior generalização aos resultados.

Assim como outras técnicas de computação inteligente que possuem componentes baseados em princípios estocásticos, os LLMs apresentam uma complicação no sentido da replicação precisa dos resultados alcançados por abordagens baseadas nesses métodos. Devido ao caráter estocástico do GPT-3.5 Turbo, não há garantias de que as estratégias de geração de rankings avaliadas atinjam o mesmo desempenho em execuções futuras. Para atenuar essa questão, a maioria dos testes realizados foi executada cinco vezes para uma mesma configuração, e o resultado final considerado foi a média ou a mediana do conjunto de execuções. Idealmente, para se obter maior segurança estatística, seriam realizadas ao menos 30 execuções para cada configuração; todavia, os custos envolvendo testes com LLMs podem ser proibitivos. Nesse sentido, a literatura tem acomodado uma quantidade menor de rodadas, balanceando entre rigor estatístico e viabilidade prática.

O conjunto de dados utilizado neste experimento, o ConDefects, reúne características que atenuam o problema do vazamento de dados, o qual pode afetar a avaliação de técnicas baseadas em LLM. Em tese, tanto a disponibilização do conjunto de dados curados pelo ConDefects quanto a construção dos próprios programas foram posteriores à data de corte dos dados utilizados para treinamento dos modelos GPT-3.5 e GPT-4o. No entanto, pontualmente, foi utilizado o LLM Claude 3.5, que possui treinamento com

dados coletados até abril de 2024. Embora exista o risco de que esse modelo tenha tido acesso aos programas originais que posteriormente foram obtidos pelo ConDefects, isso é pouco provável, visto que não existe uma forma trivial de acessar tais programas diretamente da plataforma onde foram produzidos.

Outro ponto que pode impactar a generalização dos resultados alcançados é o fato de que os experimentos não foram exaustivos em relação a todos os programas disponíveis no conjunto de dados. Em função de limitações de recursos e tempo, foram realizadas seleções aleatórias de subconjuntos de programas, buscando-se manter uma representatividade do universo de programas em ambas as linguagens.

Por fim, conforme reportado anteriormente, o ConDefects originalmente não inclui a descrição da funcionalidade pretensamente implementada no código incorreto. Sendo um conjunto de dados construído a partir de uma plataforma de competição de programação, uma forma de inferir a descrição da funcionalidade seria a obtenção da especificação da tarefa de programação que deu origem ao programa. Esse procedimento foi adotado neste experimento para viabilizar a avaliação do impacto dessa informação no desempenho das estratégias de localização de defeitos. Contudo, em função de limitações do procedimento de captura utilizado, símbolos matemáticos podem não ter sido completamente extraídos, e a descrição final pode conter algum nível de ruído.

Algumas limitações identificadas nesta pesquisa são inerentes ao uso de LLMs em tarefas complexas como a localização de defeitos em código-fonte. A dependência de um modelo específico, as restrições impostas pelo caráter estocástico dos LLMs e as limitações dos conjuntos de dados disponíveis representam desafios significativos. No entanto, ao reconhecer e discutir essas limitações, é possível contextualizar os resultados obtidos e fornecer direções para trabalhos futuros. Investigações adicionais que explorem a utilização de diferentes modelos de linguagem, ampliem o conjunto de dados e aprimorem as técnicas de captura e processamento de informações podem contribuir para a evolução e o fortalecimento das estratégias propostas.

CAPÍTULO 10

Conclusões

Esta pesquisa focou na investigação e compreensão dos principais fatores que influenciam a acurácia das técnicas automáticas de localização de defeitos de software. A partir de uma revisão das abordagens mais recentes na área, foram apresentadas três contribuições principais: i) uma sistematização dos fatores-chaves para a construção de metodologias de localização de defeitos de software; ii) uma proposta de localização de defeitos utilizando CNNs; e, iii) uma metodologia baseada em LLMs.

Baseando-se na formalização dos aspectos-chaves e no potencial das CNNs em contextos com grandes volumes de dados, foi proposta uma modelagem de localização de defeitos com duas abordagens: i) predição da propensão ao defeito de um único *statement*; e ii) predição da propensão de um bloco de código conter elementos defeituosos. A experimentação em projetos do *benchmark* defects4j demonstrou que, embora as CNNs sejam competitivas em comparação com heurísticas tradicionais como Ochiai, essas propostas não representaram um avanço na área de localização de defeitos de software.

Em paralelo, explorando o potencial dos LLMs, a pesquisa propôs uma metodologia baseada em modelos de linguagem para a localização de defeitos. Essa metodologia foi avaliada em duas abordagens: i) uma dependente de descrição, que requer documentação prévia em linguagem natural da funcionalidade defeituosa; e ii) uma não dependente de descrição, onde o LLM infere a descrição da funcionalidade a partir do próprio código. Os resultados experimentais em um conjunto de dados com mais de 1000 programas Python mostraram que ambas as abordagens superaram heurísticas tradicionais baseadas em cobertura, revelando-se competitivas em relação a outros métodos de *Deep Learning*.

Diante da realidade do problema de vazamento de dados em diversos contextos de avaliação de métodos baseados em LLM, foi realizada uma avaliação experimental em cenário controlado onde foi possível aplicar mecanismos para atenuar tal problemática. Os resultados demonstraram que as propostas de geração de rankings de suspeita baseados em LLM foram capazes de localizar defeitos mesmo em programas com remota possibilidade terem pertencido ao treinamento do LLM. Ademais, a utilização de dados de cobertura de fluxo de controle, uma informação seguramente alheia ao treinamento do LLM, melhorou o resultado produzido pelas técnicas baseadas em LLM isoladamente.

10.1 Trabalhos Futuros

Um dos caminhos naturais de evolução desta pesquisa reside na exploração da integração entre LLMs e CNNs. Explorar como os LLMs podem ser integrados com Redes Neurais Convolucionais para melhorar a análise de logs e documentação durante o processo de localização de defeitos. Essa integração poderia enriquecer o modelo com informações contextuais que não estão presentes na matriz de cobertura. Paralelamente, a capacidade das CNNs, especialmente em relação à localização de defeitos por blocos de código, poderia ser utilizada como fase anterior da aplicação de LLMs para realizar um corte no espaço de busca do programa, atenuando o problema de limitação do tamanho do contexto de alguns LLMs.

Especificamente com relação às abordagens baseadas em LLM, aproveitando a capacidade de sumarização desses modelos, devem ser desenvolvidas estratégias que incorporem, além da descrição de funcionalidade, outros dados que concorram para a localização de defeitos, como informações sobre os casos de teste ou mesmo dados de cobertura de fluxo de controle. Por exemplo, é possível utilizar as heurísticas como Ochiai e Tarantula para gerar suspeitas que indicam ao LLM áreas do código que sejam mais propensas ao defeito.

O aprimoramento das abordagens de localização de defeitos baseadas em LLM passa também pela aplicação de técnicas mais avançadas e que já são amplamente usadas no contexto em linguagem natural. Por exemplo, pretende-se utilizar o conceito de Geração Aumentada de Informação (RAG) para fornecer ao LLM a capacidade de acessar múltiplas informações de documentação, histórico de edições, *logs* de execução e arquivos de configuração que podem ser relevantes para determinação do local do defeito. Uma abordagem como essa é especialmente útil em cenários de software mais complexos nos quais o artefato executável, ou código-fonte, possui relação ou dependência de outros arquivos do ambiente do ecossistema.

Por fim, como esforço para uma maior adesão da indústria a ferramentas práticas de localização de defeitos, pretende-se trabalhar o desenvolvimento de ferramentas de Visualização de localização de defeitos, que não apenas apresentem o ranking de *statements* ou blocos suspeitos gerados pelo técnica de localização de defeitos, mas que também explorem a capacidade dos LLMs em explicar os motivos de defeito. Ferramentas com essas características têm potencial de aumentar a produtividade dos engenheiros de software, uma vez que haverá uma rápida identificação de áreas de interesse e, possivelmente, melhorar a qualidade da correção aplicada, em função da existência de mais informação, em linguagem natural, sobre as causas do defeito.

10.2 Lista de Publicações

A seguir, é apresentada a lista de publicações científicas resultantes deste processo de doutoramento, que possuem relação direta ou indireta com a presente pesquisa. Todos os artigos listados abordam, em diferentes graus, o uso de inteligência computacional no contexto da Engenharia de Software, embora nem todos tratem diretamente da localização de defeitos de software:

- D. Silva-Junior, P. Leitao-Junior, A. Dantas, C. G. Camilo-Junior and R. Harrison, **Data-Flow-Based Evolutionary Fault Localization.** Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing, 2020.
- Roque, Lucas and Dantas, Altino and Camilo-Junior, Celso G., **Programming Style Analysis with Recurrent Neural Network to Automatic Pull Request Approval.** The International Joint Conference on Neural Networks, 2019.
- L. Afonso Amorim and M. F. Freitas and A. Dantas and C. G. Camilo-Junior and W. Martins and W. Santos Martins, **Accelerating word Embedding Generation with Fine-Grain Parallelism.** Brazilian Conference on Intelligent Systems (BRACIS), 2019.
- Dantas, Altino and de Souza, Eduardo F. and Souza, Jerffeson and Camilo-Junior, Celso G., **Code Naturalness to Assist Search Space Exploration in Search-Based Program Repair Methods.** Symposium on Search-Based Software Engineering (SSBSE), 2019.
- Roque, Lucas and Dantas, Altino and Camilo-Junior, Celso, **Análise de estilo de programação com Rede Neural Recorrente para aprovação automática de pull request.** 9º Workshop de Engenharia de Software Baseada em Busca (WESB), 2018.
- L. A. Amorim and M. F. Freitas and A. Dantas and E. F. de Souza and C. G. Camilo-Junior and W. S. Martins, **A New Word Embedding Approach to Evaluate Potential Fixes for Automated Program Repair.** International Joint Conference on Neural Networks (IJCNN), 2018.
- de Freitas, Diogo M and Leitao-Junior, Plinio and Camilo-Junior, Celso and Dantas, Altino and HARRI-SON, Rachel, **Genetic Programming-based Composition of Fault Localization Heuristics.** 7º Workshop de Engenharia de Software Baseada em Busca (WESB), 2017.
- Olveira, Vinicius and de Souza, Eduardo Faria and Dantas, Altino and Roque, Lucas and Camilo-Junior, Celso and Souza, Jerffeson, **Ternarius: Um Operador de Mutação Para o Reparo de Software Baseado em Busca com Representação Subpatch.** 7º Workshop de Engenharia Baseada em Busca (WESB), 2017.

Referências Bibliográficas

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. V. An evaluation of similarity coefficients for software fault localization. In: *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. [S.l.: s.n.], 2006. p. 39–46.

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. V. On the accuracy of spectrum-based fault localization. In: IEEE. *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. [S.l.], 2007. p. 89–98.

ABREU, R. et al. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, v. 82, n. 11, p. 1780–1792, 2009. ISSN 0164-1212. SI: TAIC PART 2007 and MUTATION 2007. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121209001319>.

AGANS, D. J. *Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems*. [S.l.]: HarperChristian+ ORM, 2002.

AGRAWAL, H. et al. Fault localization using execution slices and dataflow tests. In: IEEE. *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. [S.l.], 1995. p. 143–151.

AIYAPPA, R. et al. Can we trust the evaluation on chatgpt? *arXiv preprint arXiv:2303.12767*, 2023.

ANTHROPIC. *Claude 3.5: Anthropic's Large Language Model*. 2024. <https://www.anthropic.com/index/claude>. Accessed: 2024-09-12.

BATCHELDER, N. *coverage.py: Code coverage testing for Python*. [S.l.], 2024. Versão 7.2.0. Disponível em <https://coverage.readthedocs.io/>. Disponível em: <https://coverage.readthedocs.io/>.

BöHME, M. et al. Where is the bug and how is it fixed? an experiment with practitioners. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 117–128. ISBN 9781450351058. Disponível em: <https://doi.org/10.1145/3106237.3106255>.

BROWN, T. B. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

BROWN, T. B. et al. *Language Models are Few-Shot Learners*. 2020. Disponível em: <https://arxiv.org/abs/2005.14165>.

CAI, H.; XU, X. A new spectrum-based fault localization method by using clustering algorithm. *International Journal of Advancements in Computing Technology*, Advanced Institute of Convergence IT Technology, v. 4, n. 22, 2012.

CETINER, M.; SAHINGOZ, O. K. A comparative analysis for machine learning based software defect prediction systems. In: IEEE. *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. [S.l.], 2020. p. 1–7.

CHANG, Y. et al. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, Association for Computing Machinery, New York, NY, USA, v. 15, n. 3, mar. 2024. ISSN 2157-6904. Disponível em: <https://doi.org/10.1145/3641289>.

CHOLLET, F. et al. *Keras*. 2015. <https://keras.io/>. Accessed: 2023-10-21.

CHRISTIANO, P. F. et al. Deep reinforcement learning from human preferences. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2017. p. 4299–4307.

DE-FREITAS, D. M. et al. Mutation-based evolutionary fault localisation. In: *2018 IEEE Congress on Evolutionary Computation (CEC)*. [S.l.: s.n.], 2018. p. 1–8.

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

DEVLIN, J. et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. Disponível em: <https://arxiv.org/abs/1810.04805>.

DUBEY, A. et al. Auctions with llm summaries. In: *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2024. (KDD '24), p. 713–722. ISBN 9798400704901. Disponível em: <https://doi.org/10.1145/3637528.3672022>.

FAN, A. et al. Large language models for software engineering: Survey and open problems. In: IEEE. *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. [S.l.], 2023. p. 31–53.

GOYAL, T.; LI, J. J.; DURRETT, G. News summarization and evaluation in the era of gpt-3. *arXiv preprint arXiv:2209.12356*, 2022.

GU, J. et al. Recent advances in convolutional neural networks. *Pattern Recognition*, v. 77, p. 354–377, 2018. ISSN 0031-3203. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0031320317304120>.

HAILPERN, B.; SANTHANAM, P. Software debugging, testing, and verification. *IBM Systems Journal*, v. 41, n. 1, p. 4–12, 2002.

HARROLD, M. et al. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, v. 10, 09 2000.

HIRSCH, T.; HOFER, B. A systematic literature review on benchmarks for evaluating debugging approaches. *Journal of Systems and Software*, v. 192, p. 111423, 2022. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121222001303>.

HOCHREITER, S. Long short-term memory. *Neural Computation MIT-Press*, 1997.

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

HU, X. et al. A systematic view of model leakage risks in deep neural network systems. *IEEE Transactions on Computers*, IEEE, v. 71, n. 12, p. 3254–3267, 2022.

HU, Y. et al. Re-factoring based program repair applied to programming assignments. In: IEEE/ACM. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2019. p. 388–398.

INC., A. *AtCoder Programming Contests*. 2024. <https://atcoder.jp>. Accessed: 2024-09-27.

JaCoCo Contributors. *JaCoCo: Java Code Coverage Library*. [S.l.], 2024. Available at <https://www.jacoco.org/jacoco/>. Disponível em: <https://www.jacoco.org/jacoco/>.

JANSSEN, T.; ABREU, R.; GEMUND, A. J. van. Zoltar: a spectrum-based fault localization tool. In: *Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime*. New York, NY, USA: Association for Computing Machinery, 2009. (SINTER '09), p. 23–30. ISBN 9781605586816. Disponível em: <https://doi.org/10.1145/1596495.1596502>.

JIANG, L.; SU, Z. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2007. p. 184–193.

JONES, J.; HARROLD, M.; STASKO, J. Visualization of test information to assist fault localization. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. [S.l.: s.n.], 2002. p. 467–477.

JURAFSKY, D.; MARTIN, J. H. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. ed. [s.n.], 2024. Online manuscript released August 20, 2024. Disponível em: <https://web.stanford.edu/~jurafsky/slp3/>.

JUST, R.; JALALI, D.; ERNST, M. D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2014. (ISSTA 2014), p. 437–440. ISBN 9781450326452. Disponível em: <https://doi.org/10.1145/2610384.2628055>.

KANG, S.; AN, G.; YOO, S. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proceedings of the ACM on Software Engineering*, ACM New York, NY, USA, v. 1, n. FSE, p. 1424–1446, 2024.

KIM, S. et al. Predicting faults from cached history. In: IEEE. *29th International Conference on Software Engineering (ICSE'07)*. [S.l.], 2007. p. 489–498.

KIRANYAZ, S. et al. 1d convolutional neural networks and applications: A survey. *CoRR*, abs/1905.03554, 2019. Disponível em: <http://dblp.uni-trier.de/db/journals/corr/corr1905.html#abs-1905-03554>.

LANGCHAIN. *LangChain 3: A Framework for Developing Applications Powered by Language Models*. 2024. Accessed: 2024-07-21. Disponível em: <https://www.langchain.com/langchain3>.

LE, T.-D. B. et al. A learning-to-rank based fault localization approach using likely invariants. In: . New York, NY, USA: Association for Computing Machinery, 2016. (ISSTA 2016), p. 177–188. ISBN 9781450343909. Disponível em: <https://doi.org/10.1145/2931037.2931049>.

LE, T.-D. B. et al. A learning-to-rank based fault localization approach using likely invariants. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2016. (ISSTA 2016), p. 177–188. ISBN 9781450343909. Disponível em: <https://doi.org/10.1145/2931037.2931049>.

LECUN, Y. et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, v. 86, n. 11, p. 2278–2324, 1998.

LEWIS, M. et al. *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. 2019. Disponível em: <https://arxiv.org/abs/1910.13461>.

LI, X. et al. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In: ____. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2019. p. 169–180. ISBN 9781450362245. Disponível em: <https://doi.org/10.1145/3293882.3330574>.

LI, Y.; WANG, S.; NGUYEN, T. N. Fault localization with code coverage representation learning. In: ____. *Proceedings of the 43rd International Conference on Software Engineering*. IEEE Press, 2021. p. 661–673. ISBN 9781450390859. Disponível em: <https://doi.org/10.1109/ICSE43902.2021.00067>.

LI, Z. et al. Flexkbqa: A flexible llm-powered framework for few-shot knowledge base question answering. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. [S.l.: s.n.], 2024. v. 38, n. 17, p. 18608–18616.

LIU, Y. et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. Disponível em: <https://arxiv.org/abs/1907.11692>.

MANNING, C. D. *Foundations of statistical natural language processing*. [S.l.]: The MIT Press, 1999.

MAO, X. et al. Slice-based statistical fault localization. *Journal of Systems and Software*, Elsevier, v. 89, p. 51–62, 2014.

MATHER, P. M. *Computer Processing of Remotely-Sensed Images: An Introduction*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2004. ISBN 0470849185.

MIKOLOV, T. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

MOON, S. et al. Ask the mutants: Mutating faulty programs for fault localization. In: IEEE. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. [S.l.], 2014. p. 153–162.

MUMTAZ, B. et al. Feature selection using artificial immune network: An approach for software defect prediction. *Intelligent Automation & Soft Computing*, v. 29, n. 3, 2021.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The Art of Software Testing*. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962.

NIJKAMP, E. et al. Codegen2: Lessons for training llms on programming and natural languages. *ICLR*, 2023.

OPENAI. *GPT-3.5-turbo*. 2023. <https://platform.openai.com>. Accessed: 2024-08-20.

OPENAI. *GPT-4: OpenAI's Large Language Model*. 2023. <https://openai.com/research/gpt-4>. Accessed: 2024-09-12.

OPENAI. *tiktoken: Tokenizer for OpenAI models*. [S.l.], 2024. Disponível em <https://github.com/openai/tiktoken>. Disponível em: <https://github.com/openai/tiktoken>.

O'SHEA, K.; NASH, R. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015. Disponível em: <http://dblp.uni-trier.de/db/journals/corr/corr1511.html#OSheaN15>.

PAPADAKIS, M.; TRAON, Y. L. Using mutants to locate"unknown"faults. In: IEEE. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. [S.l.], 2012. p. 691–700.

PAPADAKIS, M.; TRAON, Y. L. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, Wiley Online Library, v. 25, n. 5-7, p. 605–628, 2015.

PEARSON, S. et al. Evaluating and improving fault localization. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2017. p. 609–620.

PETERS, M. E. et al. Deep contextualized word representations. In: WALKER, M.; JI, H.; STENT, A. (Ed.). *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, 2018. p. 2227–2237. Disponível em: <https://aclanthology.org/N18-1202>.

RAFFEL, C. et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2023. Disponível em: <https://arxiv.org/abs/1910.10683>.

RAHMAN, F. et al. Bugcache for inspections: hit or miss? In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. [S.l.: s.n.], 2011. p. 322–331.

REPAIR, P. *Program Repair - Research and Tools*. 2024. Accessed: 2024-07-21. Disponível em: <https://program-repair.org/index.html>.

RESEARCH, G. *Google Colaboratory*. 2023. Accessed: 2023-10-21. Disponível em: <https://colab.research.google.com/>.

ROSENFELD, R. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, IEEE, v. 88, n. 8, p. 1270–1278, 2000.

RUMELHART, D. E. et al. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, Lawrence Erlbaum Hillsdale, NJ, USA, p. 1–34, 1995.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 3. ed. [S.l.]: Prentice Hall, 2010.

SAMALA, R. K. et al. Hazards of data leakage in machine learning: a study on classification of breast cancer using deep neural networks. In: SPIE. *Medical Imaging 2020: Computer-Aided Diagnosis*. [S.l.], 2020. v. 11314, p. 279–284.

SANTU, S. K. K.; FENG, D. *TELeR: A General Taxonomy of LLM Prompts for Benchmarking Complex Tasks*. 2023. Disponível em: <https://arxiv.org/abs/2305.11430>.

SELENIUM. *Selenium with Python*. 2024. <https://www.selenium.dev/documentation/webdriver/>. Accessed: 2024-09-27.

SILVA, I. N. d.; SPATTI, D. H.; FLAUZINO, R. A. *Redes neurais artificiais para engenharia e ciências aplicadas*. [S.l.]: Artliber Editora, 2010.

SILVA-JUNIOR, D. et al. Data-flow-based evolutionary fault localization. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2020. (SAC '20), p. 1963–1970. ISBN 9781450368667. Disponível em: <https://doi.org/10.1145/3341105.3373946>.

SILVA-JUNIOR, D. d. et al. A systematic mapping of the proposition of benchmarks in the software testing and debugging domain. *Software*, v. 2, n. 4, p. 447–475, 2023. ISSN 2674-113X. Disponível em: <https://www.mdpi.com/2674-113X/2/4/21>.

SOBREIRA, V. et al. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In: *Proceedings of SANER*. [S.l.: s.n.], 2018.

SOMMERVILLE, I. *Software Engineering*. 10th. ed. Boston, MA: Pearson, 2015. ISBN 978-0-13-394303-0.

STYAN, G. P. Hadamard products and multivariate statistical analysis. *Linear Algebra and its Applications*, v. 6, p. 217–240, 1973. ISSN 0024-3795. Disponível em: <https://www.sciencedirect.com/science/article/pii/0024379573900232>.

SUMATHI, S.; PANEERSELVAM, S. *Computational Intelligence Paradigms: Theory and Applications Using MATLAB*. 1st. ed. USA: CRC Press, Inc., 2010. ISBN 143980902X.

TANTITHAMTHAVORN, C. et al. Automated parameter optimization of classification techniques for defect prediction models. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2016. p. 321–332.

TOUVRON, H. et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. Disponível em: <https://arxiv.org/abs/2302.13971>.

TOUVRON, H. et al. *LLaMA 3: Open and Efficient Foundation Language Models*. 2024. <https://ai.meta.com/>. Accessed: 2024-08-20.

U.S. Bureau of Labor Statistics. *Occupational Outlook Handbook*. 2023. Accessed: 2024-09-30. Disponível em: <https://www.bls.gov/oes/tables.htm>.

VASWANI, A. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

WEI, J. et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

WHITE, J. et al. *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*. 2023. Disponível em: <https://arxiv.org/abs/2302.11382>.

WONG, C.-P. et al. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: IEEE. *2014 IEEE international conference on software maintenance and evolution*. [S.l.], 2014. p. 181–190.

WONG, W. E. et al. Software fault localization using dstar (d*). In: *2012 IEEE Sixth International Conference on Software Security and Reliability*. [S.l.: s.n.], 2012. p. 21–30.

WONG, W. E. et al. A survey on software fault localization. *IEEE Transactions on Software Engineering*, v. 42, n. 8, p. 707–740, 2016.

WU, R. et al. Crashlocator: Locating crashing faults based on crash stacks. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2014. p. 204–214.

WU, Y. et al. Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair. In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2024. (FSE 2024), p. 642–646. ISBN 9798400706585. Disponível em: <https://doi.org/10.1145/3663529.3663815>.

XIE, X. et al. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on software engineering and methodology (TOSEM)*, ACM New York, NY, USA, v. 22, n. 4, p. 1–40, 2013.

XING, F. Designing heterogeneous llm agents for financial sentiment analysis. *ACM Trans. Manage. Inf. Syst.*, Association for Computing Machinery, New York, NY, USA, ago. 2024. ISSN 2158-656X. Just Accepted. Disponível em: <https://doi.org/10.1145/3688399>.

YANG, A. Z. et al. Large language models for test-free fault localization. In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. [S.l.: s.n.], 2024. p. 1–12.

YOO, S. Evolving human competitive spectra-based fault localisation techniques. In: SPRINGER. *Search Based Software Engineering: 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings 4*. [S.l.], 2012. p. 244–258.

ZHANG, Z. et al. Cnn-fl: An effective approach for localizing faults using convolutional neural networks. In: IEEE. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.], 2019. p. 445–455.

ZHANG, Z. et al. A study of effectiveness of deep learning in locating real faults. *Information and Software Technology*, v. 131, p. 106486, 2021. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584920302287>.

ZHAO, W. X. et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

ZOU, D. et al. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, IEEE, v. 47, n. 2, p. 332–347, 2019.

ZOU, D. et al. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, v. 47, p. 332–347, 2021.

APÊNDICE A

Resultados do Experimento 2

Tabela A.1: Métricas Acurácia, F1 e Perda nos conjuntos treino e teste para diferentes execuções de treinamento e avaliação considerando diferentes configurações de dados

		Conjunto de treino			**Conjunto de teste**		
Configuração	Execução	Acurácia	F1	Perda	Acurácia	F1	Perda
Cobertura sem (-1)	1	98,1061	97,8066	3,1993	89,7059	89,0756	7,5373
	2	95,4545	95,4040	8,9928	77,9412	75,7983	12,6414
	3	95,0758	94,4012	10,0846	88,2353	91,2045	4,5672
Cobertura sem (-1) + Heurísticas	1	89,3939	87,8427	16,5639	88,2353	89,0476	3,6832
	2	91,6667	90,7143	15,5262	89,7059	89,0756	4,7026
	3	90,5303	89,0188	16,8516	94,1176	94,1176	2,4192
Cobertura com (-1) + Heurísticas	1	92,0455	90,5195	15,3560	92,6471	93,2773	2,8982
	2	94,3182	93,1313	12,8068	85,2941	80,6162	5,1028
	3	91,2879	90,5483	16,1828	91,1765	89,3838	3,7334
	4	91,6667	90,5988	14,7315	91,1765	92,2409	4,1862
Cobertura com (-1)	1	97,3485	96,8615	9,8635	83,8235	83,5854	12,0273
	2	94,3182	93,8312	12,4578	94,1176	92,1289	3,2589
	3	96,5909	96,1760	8,5703	82,3529	77,1709	20,7705
Cobertura + Heurísticas Bloco tamanho = 128	1	81,2500	73,2143	2,6838	87,5000	82,1429	1,3256
	2	12,5000	5,0000	2,8737	25,0000	16,6667	1,4120
	3	81,2500	73,2143	2,4427	87,5000	82,1429	1,1886
Cobertura + Heurísticas Bloco tamanho = 64	1	91,6667	88,4921	7,6284	91,6667	88,0952	1,9006
	2	89,5833	85,5159	6,9009	100,0000	100,0000	1,6024
	3	91,6667	88,8889	7,8845	91,6667	88,0952	1,9634
Cobertura + Heurísticas Bloco tamanho = 32	1	95,3704	93,5626	17,9721	96,4286	94,8980	4,6498
	2	94,4444	92,2399	17,1466	96,4286	94,8980	4,3934
	3	96,2963	94,7090	17,2823	92,8571	89,7959	4,5194
Cobertura + Heurísticas Bloco tamanho = 16	1	97,4576	96,3680	30,3941	96,6667	95,2381	7,7509
	2	97,0339	95,7627	28,7540	98,3333	97,6190	7,1620
	3	97,0339	95,8434	29,4686	98,3333	97,6190	7,3683
Cobertura + Heurísticas Bloco tamanho = 8	1	98,7903	98,3103	63,3570	96,7742	95,3917	15,9432
	2	98,3871	97,6959	46,9171	98,3871	97,6959	11,5865
	3	98,1855	97,4078	80,0149	99,1935	98,8479	19,9615
Cobertura Bloco tamanho = 128	1	86,1111	80,5397	8,4628	87,5000	82,1429	2,2687
	2	88,3333	83,3333	7,1921	79,1667	71,6667	2,1206
	3	88,3333	83,3333	7,6739	81,2500	74,4048	2,1989
Cobertura Bloco tamanho = 64	1	92,0455	89,0693	14,2646	87,5000	82,1429	3,9034
	2	90,9091	87,0130	11,9149	91,6667	88,0952	3,1682
	3	89,7727	85,3896	8,6972	95,8333	94,0476	2,0121
Cobertura Bloco tamanho = 32	1	93,2432	90,4762	21,9857	100,0000	100,0000	5,6846
	2	95,2703	93,2432	24,0246	92,5000	89,2857	6,5038
	3	94,5946	92,2780	14,5920	95,0000	92,8571	3,9379
Cobertura Bloco tamanho = 16	1	97,8571	96,9388	35,7377	94,4444	92,0635	9,3602
	2	97,3810	96,2721	40,4967	95,8333	94,0476	10,4254
	3	97,1429	95,9184	35,3687	97,2222	96,0317	9,0223
Cobertura Bloco tamanho = 8	1	98,1481	97,3545	86,9807	98,5294	97,8992	21,8736
	2	97,5926	96,5961	82,2435	100,0000	100,0000	20,5562
	3	98,5185	97,8836	28,8854	97,0588	95,7983	8,0216

Fonte: Elaborada pelo autor.

APÊNDICE B

Arquitetura Conv_1D em TensorFlow keras

Código 10: Definição da Arquitetura Conv_1D em TensorFlow keras.

```
# definindo a aquitetura do modelo
def create_model():

    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Conv1D(filters=8,
                                     kernel_size=3,
                                     activation='relu',
                                     input_shape=(n_timesteps,
                                      ↪  n_features)))
    model.add(tf.keras.layers.Dropout(0.4))

    model.add(tf.keras.layers.AveragePooling1D(pool_size=2))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(20, activation='relu'))
    model.add(tf.keras.layers.Dropout(0.3))
    model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

    optimizer = tf.keras.optimizers.SGD(learning_rate=0.001,
                                        momentum=0.9,
                                        decay=1e-6)
    model.compile(loss='binary_crossentropy', optimizer=optimizer,
                  metrics=['mae', 'acc'])

    return model

```

APÊNDICE C

Arquitetura Conv_mu em PyThorch

Código 11: Definição da Arquitetura Conv_mu em PyThorch.

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        # conv of coverage
        self.conv1 = nn.Conv2d(1, 4, 8)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(4, 16, 5)
        self.fc1 = nn.Linear(54064, 512) # d1
        # encode f_matrix
        self.fc2 = nn.Linear(5428, 512)
        # merge cov and f_matrix
        self.fc3 = nn.Linear(1024, 512)
        # classififer
        self.dr1 = nn.Dropout(0.2)
        self.fc4 = nn.Linear(512, 256)
        self.dr2 = nn.Dropout(0.05)
        self.fc5 = nn.Linear(256, 2)

    def forward(self, cov, f_matrix):
        x = self.pool(F.relu(self.conv1(cov)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x_ = F.relu(self.fc2(f_matrix))
        x = torch.cat((x, x_), 1)
        x = F.relu(self.fc3(x))

        x = self.dr1(x)
        x = F.relu(self.fc4(x))
        x = self.dr2(x)
        x = self.fc5(x)
        return x
```

APÊNDICE D

Arquitetura Conv_cov em PyThorch

Código 12: Definição da Arquitetura Conv_cov em PyThorch.

```
# Definindo a arquitetura
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 4, 8)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(4, 16, 5)
        self.fc1 = nn.Linear(546720, 120)
        self.dr1 = nn.Dropout(0.3)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 2)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except
        ↪ batch
        x = F.relu(self.fc1(x))
        x = self.dr1(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=lr,
 ↪  weight_decay=weight_decay)

```

APÊNDICE E

Arquitetura MLP_like em TensorFlow Keras

Código 13: Definição da Arquitetura Conv_cov em PyThorch.

```
# definindo a aquitetura do modelo
def create_model():

    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Dense(10,
     ↪  input_dim=num_tc,activation='relu'))
    model.add(tf.keras.layers.Dropout(0.4))
    model.add(tf.keras.layers.Dense(20, activation='relu'))
    model.add(tf.keras.layers.Dropout(0.3))
    model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
    model.compile(loss='binary_crossentropy',
                        optimizer=optimizer,
                        metrics=['mae', 'acc'])
    return model

```

APÊNDICE F

Comparativo entre as estratégias fl-top-k e fl-it-k

Tabela F.1: Mediana de acertos e média de tokens das cinco execuções de cada valor de k comparando-se as estratégias fl-top-k e fl-it-k para programas 300 Python

	fl-top-k		fl-it-k	
k	**acertos**	**tokens**	**acertos**	**tokens**
top-1	69	3845.2	71	3691.6
top-2	106	7121.6	82	7331.6
top-3	127	10395.6	90	11049.4
top-4	139	13265	94	14770.4
top-5	154	15843	93	18442.2

Fonte: Elaborada pelo autor.

Tabela F.2: Mediana de acertos e média de tokens das cinco execuções de cada valor de k comparando-se as estratégias fl-top-k e fl-it-k para 300 programas Java

	fl-top-k		fl-it-k	
k	**acertos**	**tokens**	**acertos**	**tokens**
top-1	65	3733,4	64	3971,8
top-2	95	6607,2	81	7877,6
top-3	116	9223,8	85	11698,6
top-4	125	11455,6	91	15772
top-5	138	13756	94	19498,8

Fonte: Elaborada pelo autor.

APÊNDICE G

Análise de Comportamento da Estratégia de Geração de Correções Candidatas

Neste apêndice, é apresentada a análise do comportamento do fluxo de geração de correções candidatas para cada problema selecionado no conjunto de dados utilizado no experimento apresentado no Capítulo 8, separando-se a análise por cada um dos fluxos, dependente ou não dependente de descrição. Vale a nota de que esse procedimento foi adotado como medida para contornar a ausência de anotação sobre local do defeito no conjunto de dados original.

G.1 Fluxo Dependente de Descrição

A Tabela G.1 apresenta as porcentagens de correções alcançadas por cada modelo em relação ao número total de versões defeituosas para cada problema (coluna "Correção"), bem como as porcentagens de ocorrências de falhas durante a geração de respostas (coluna "Falha do LLM"), tudo isso, considerando o fluxo dependente de descrição. Destaque-se que, para esta análise, vale a definição de código corrigido que é utilizada no item d) da seção G.1, a saber: é considerada uma correção, uma versão que altera uma versão incorreta e que sejam positivos todos os casos de teste executados sobre si.

Primeiramente, podemos destacar os resultados relacionados às falhas dos LLMs em produzir a saída esperada ao receber uma versão defeituosa e uma descrição da funcionalidade desejada como entrada. Comparado com as falhas da Abordagem Dependente de Descrição para localização de defeitos (Tabela 8.2), é perceptível que tanto o GPT quanto o Llama tiveram consideravelmente mais dificuldade em gerar saídas adequadas. Essa diferença é esperada porque o modelo gera texto já presente na entrada para indicar a localização do defeito. Por outro lado, para o fluxo de correção, realmente é criado novo conteúdo. Nesse aspecto, o Llama 3 apresentou um desempenho inferior ao GPT 3.5. Ao se investigar os casos, descobriu-se que a maioria das falhas foi causada por caracteres gerados repetidamente, característicos de um comportamento de alucinação do modelo.

Tabela G.1: Desempenho do Fluxo de Geração de Correções Candidatas Dependente de Descrição, considerando todos os problemas e LLMs

	GPT 3.5		Llama 3	
Problema	**Correção**	**Falha do LLM**	**Correção**	**Falha do LLM**
Question 1	**165 (29%)**	12 (2.09%)	118 (20.52%)	20 (3.48%)
Question 3	233 (75.65%)	4 (1.30%)	**251 (81.49%)**	11 (3.57%)
Question 4	307 (85.99%)	2 (0.56%)	**341 (95.52%)**	12 (3.36%)
Question 5	**72 (66.67%)**	7 (6.48%)	62 (57.41%)	12 (11.11%)

Fonte: Elaborada pelo autor.

Em relação às correções, houve um equilíbrio no comportamento de ambos os modelos. Enquanto o GPT foi superior nos problemas Question 1 e Question 5, o Llama se saiu melhor nos programas dos problemas Question 3 e Question 4. Explicitamente discutindo o problema Question 1, ambos os modelos tiveram um desempenho substancialmente inferior em comparação com os outros problemas. Esse comportamento pode ser devido ao fato de que a descrição da funcionalidade fornecida para o problema Question 1 não parece ser suficientemente detalhada. Enquanto os outros problemas possuem uma anotação para a tarefa e uma descrição detalhada da funcionalidade, o problema Question 1 possui apenas o nome da tarefa (Sequential Search).

Por outro lado, em termos relativos, o Llama 3 alcançou o melhor resultado para as versões do problema Question 4 (95,52%). Se voltarmos às descrições de cada problema, o problema Question 4 é exatamente aquele com a descrição de funcionalidade mais detalhada, incluindo um exemplo de entrada e saída. Esse achado apoia a intuição de que uma descrição mais detalhada favorece a correção da funcionalidade implementada.

Ainda em relação à capacidade dos modelos em gerar correções, foi observado que há um certo nível de complementaridade entre eles. Observando o problema Question 3, o GPT corrigiu 233 versões, das quais 46 foram corrigidas exclusivamente por ele. Por outro lado, para o mesmo problema, o Llama corrigiu 251 versões, com 64 sendo exclusivas. Portanto, juntos, os modelos corrigiram 297 das 308 versões. Para nota, ambos os modelos corrigiram a versão “wrong 3 017.py”, apenas o GPT corrigiu a versão "wrong 3 006.py", e apenas o Llama corrigiu a versão “wrong 3 007.py”. A lista completa das versões corrigidas para todos os problemas, incluindo arquivos executáveis .py, está disponível no material suplementar desta pesquisa.

No Código 14, pode ser visto o código da versão nomeada “wrong 3 017.py” do problema Question 3 e duas versões corrigidas, geradas pelo Llama 3 e pelo GPT, respectivamente.

Além do código Python, pedimos ao LLM que explicasse brevemente sua

Código 14: Programa Original 017 do Problema Question 3 e suas correções

```
# buggy version
def remove_extras(lst):
    for i in range(len(lst)):
        for j in range(j+1, len(lst)):
            if lst[j] == lst[i]:
                lst = lst[:j] + lst[j+1:]
    return lst

# correct version generated by llama 3
def remove_extras(lst):
    new_list = []
    for item in lst:
        if item not in new_list:
            new_list.append(item)
    return new_list

# correct version generated by GPT
def remove_extras(lst):
    return list(dict.fromkeys(lst))

```

solução neste experimento. Assim, para a correção da versão "wrong 3 017.py", o GPT respondeu o seguinte: “O código corrigido usa uma abordagem mais concisa ao converter a lista de entrada em um dicionário para remover duplicatas e, em seguida, converter de volta em uma lista. Esse método garante que todas as ocorrências repetidas de qualquer elemento sejam removidas de forma eficiente, sem a necessidade de laços aninhados.” O Llama 3 retornou o seguinte: “O código original não era eficiente e poderia levar a um erro de índice fora do intervalo. O novo código cria uma nova lista e apenas adiciona itens que ainda não estão na nova lista, garantindo que não haja duplicatas e evitando possíveis erros.”

Em resumo, destaque-se que o modelo GPT corrigiu, em média, 64% das versões entre os problemas, atingindo 85,99% no melhor cenário, que foi para o problema Question 4. Ao tempo que o modelo Llama atingiu uma média de 63,73%, com 95,52% no melhor caso.

G.2 Fluxo Não Dependente de Descrição

A Tabela G.2 trás a taxa de correções alcançadas por cada modelo em relação ao número total de versões defeituosas para cada problema, bem como o percentual de ocor-

rência de falhas durante a geração de respostas (coluna "Falha do LLM"), considerando-se o fluxo de geração de correções candidatas não dependente de descrição.

Tabela G.2: Desempenho do Fluxo de Geração de Correções Candidatas Não Dependente de Descrição, considerando todos os problemas e LLMs

	GPT 3.5		**Llama 3**	
Problem	**Fix**	**LLM failure**	**Fix**	**LLM failure**
Question 1	272 (47.30%)	14 (2.43%)	202 (35.13%)	36 (6.26%)
Question 3	156 (50.65%)	47 (15.26%)	176 (57.14%)	16 (5.19%)
Question 4	150 (42.02%)	49 (13.73%)	165 (46.22%)	6 (5.56%)
Question 5	79 (73.15%)	6 (5.56%)	74 (68.52%)	12 (11.11%)

Fonte: Elaborada pelo autor.

Como pode ser visto, o GPT atingiu 47,3% e o Llama 35,13%, em comparação com 29% e 20,52%, respectivamente, na abordagem anterior, considerando o problema Question 1. Essa melhoria pode ser explicada pelo fato de que a descrição do problema original é muito sucinta, o que prejudica a abordagem Dependente de Descrição e favorece a abordagem Não Dependente de Descrição, que infere a funcionalidade a partir do próprio código, em vez de utilizar o texto de descrição do problema original.

Analogamente, este princípio também ajuda a explicar o declínio de desempenho de ambos os modelos ao comparar os resultados de correção de bugs para os problemas Question 3 e Question 4. Como mencionado antes, no conjunto de dados original, as descrições para esses problemas são bastante detalhadas, incluindo casos de teste. Como a abordagem Não Dependente de Descrição “vê” apenas o código defeituoso, a descrição gerada, embora precisa, não inclui exemplos de casos de teste e, consequentemente, pode ser desvantajosa na etapa de geração de correções de bugs. No futuro, podemos investigar como a introdução de casos de teste no prompt do segundo componente baseado em LLM da Abordagem Não Dependente de Descrição pode impactar.

Em síntese, para o processo de geração de correção candidata, considerando um fluxo sem a dependência das descrição da funcionalidade, o GPT teve uma taxa média de sucesso de 53,28%, e o Llama 3 obteve 51,75%, considerando todos os problemas.

APÊNDICE H

Resultados de ACC@N para heurísticas Tarantula e Ochiai

A tabela neste apêndice mostra os resultados de ACC@N das heurísticas Tarantula e Ochiai calculados para o conjunto de dados utilizado no experimento do Capítulo 8. Na Seção 8.2, são incluídos e discutidos os valores de ACC@1 de Ochiai.

Tabela H.1: Resultados na métrica ACC@N obtidos pelas heurísticas Tarantula e Ochiai para todos os problemas

	Tarantula			**Ochiai**		
	ACC@1	**ACC@5**	**ACC@10**	**ACC@1**	**ACC@5**	**ACC@10**
Question 1	26	74	88	31	87	102
Question_3	84	99	103	125	156	166
Question_4	51	112	129	137	219	246
Question_5	52	53	56	78	81	88

Fonte: Elaborada pelo autor.

APÊNDICE I

Caracterização dos Trabalhos Relacionados pelos Aspectos-chaves da Localização de Defeitos

Neste apêndice, é realizada a caracterização dos Trabalhos Relacionados, apresentados no Capítulo 3, por intermédio da formalização dos aspectos-chaves introduzidos no Capítulo 4. Dentre os trabalhos relacionados, apenas a abordagem proposta por Zou et al. (2021), pois essa técnica não combina diretamente as fontes de informações vinculadas ao defeito, e sim combina resultados de outras abordagens. Além dos trabalhos baseados em Redes Neurais Profundas e LLMs, também é apresentada a caracterização das heurísticas Ochiai e Tarantula.

Tabela I.1: Heurísticas Ochiai e Tarantula segundo os aspectos-chaves da Localização de Defeitos

Aspecto-chave	**Classificação dos Atributos**
Caracterização da Informação	***Dados de espectro de cobertura***
Disponibilidade	Média
Custo de aquisição	Médio
Origem	Externa ao código
Natureza	Estática
Tipo	Cobertura de fluxo de controle
Representação da Informação	***Matriz***
Consistência	Alta
Compatibilidade	Total
Escalabilidade	Alta
Método de combinação	***Operações Aritméticas***
Custo	Baixo
Flexibilidade	Baixa
Escalabilidade	Alta

Fonte: Elaborada pelo Autor.

Tabela I.2: AutoFL, de Kang, An e Yoo (2024), segundo os aspectos-chaves da Localização de Defeitos

Aspecto-chave	**Classificação dos Atributos**	
Caracterização da Informação	***Caso de teste negativo***	***Código-fonte***
Disponibilidade	Alta	Alta
Custo de aquisição	Baixo	Baixo
Origem	Externa ao código	O próprio código
Natureza	Dinâmica	Estática
Tipo	***Log* de execução**	***Código-fonte***
Representação da Informação	***Texto plano***	***Texto plano***
Consistência	Alta	Alta
Compatibilidade	Total	Total
Escalabilidade	Média	Média
Método de combinação		**LLM**
Custo		Alto
Flexibilidade		Alta
Escalabilidade		Baixa

Fonte: Elaborada pelo Autor.

Tabela I.3: LLMAO, de Yang et al. (2024), segundo os aspectos-chaves da Localização de Defeitos. *Informação requerida apenas na fase de treinamento

Aspecto-chave	**Classificação dos Atributos**	
Caracterização da Informação	***Histórico de correção*** *	***Código-fonte***
Disponibilidade	Média	Alta
Custo de aquisição	Médio	Baixo
Origem	Externa ao código	O próprio código
Natureza	Estática	Estática
Tipo	Histórico de versões corrigidas	Código-fonte
Representação da Informação	***Texto plano***	***Texto plano***
Consistência	Alta	Alta
Compatibilidade	Total	Total
Escalabilidade	Média	Média
Método de combinação		***LLM***
Custo		Alto
Flexibilidade		Alta
Escalabilidade		Baixa

Fonte: Elaborada pelo Autor.

Tabela I.4: Abordagem de Zhang et al. (2021) segundo os aspectos-chaves da Localização de Defeitos

Aspecto-chave	**Classificação dos Atributos**	
Caracterização da Informação	***Dados de espectro de cobertura***	
Disponibilidade	Média	
Custo de aquisição	Médio	
Origem	Externa ao código	
Natureza	Estática	
Tipo	Cobertura de fluxo de controle	
Representação da Informação	***Matriz***	
Consistência	Alta	Alta
Compatibilidade	Total	Parcial
Escalabilidade	Alta	Média
Método de combinação	***CNN***	***RNN***
Custo	Alto	Alto
Flexibilidade	Média	Média
Escalabilidade	Média	Média

Fonte: Elaborada pelo Autor.

Tabela I.5: DeepFL, Li et al. (2019), segundo os aspectos-chaves da Localização de Defeitos

Aspecto-chave				**Classificação dos Atributos**
Caracterização da Informação	***Dados de espectro de cobertura***		***Variáveis de complexidade***	***Variáveis de similaridade***
Disponibilidade	Média	Baixa	Baixa	Média
Custo de aquisição	Médio	Alto	Alto	Médio
Origem	Externa ao código	Externo ao código	Derivada do código	Híbrida
Natureza	Estática	Estática	Estática	Estática
Tipo	Cobertura de fluxo de controle	Cobertura de mutantes	Complexidade de código	Similaridade de código/texto
Representação da Informação	***Matriz***	***Matriz***	***Vetores***	***Vetores***
Consistência	Alta	Alta	Média	Média
Compatibilidade	Total	Total	Total	Total
Escalabilidade	Média	Média	Média	Média
Método de combinação				***Redes Neurais Recorrentes***
Custo				Alto
Flexibilidade				Média
Escalabilidade				Média

Fonte: Elaborada pelo Autor.

Tabela I.6: DeepLR4FL, Li, Wang e Nguyen (2021), segundo os aspectos-chaves da Localização de Defeitos

Aspecto-chave					**Classificação dos Atributos**
Caracterização da Informação	***Dados de espectro de cobertura***		***Representação abstrata do código***	***Representação de fluxo de controle***	***Representação de execução***
Disponibilidade	Média	Baixa	Média	Média	Média
Custo de aquisição	Médio	Alto	Médio	Médio	Médio
Origem	Externa ao código	Externo ao código	Derivado do código	Externa ao código	Externa ao código
Natureza	Estática	Estática	Estática	Estática	Estática
Tipo	Cobertura de fluxo de controle	Cobertura de mutantes	AST	Gráficos de fluxo de controle	Caminhos de execução
Representação da Informação	***Vetores***	***Vetores***	***Vetores***	***Vetores***	***Vetores***
Consistência	Média	Média	Média	Média	Média
Compatibilidade	Total	Total	Total	Total	Total
Escalabilidade	Alta	Alta	Alta	Alta	Alta
Método de combinação					***Redes Neurais Convolucionais***
Custo					Alto
Flexibilidade					Média
Escalabilidade					Média

Fonte: Elaborada pelo Autor.