UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CAMILA NUNES DE PAULA SOUZA

Análise de Mensagens de Commit com IA: Uma Nova Perspectiva para o Algoritmo SZZ

Recife
2025

CAMILA NUNES DE PAULA SOUZA

Análise de Mensagens de Commit com IA: Uma Nova Perspectiva para o Algoritmo SZZ

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. George Gomes Cabral

Recife
2025

Dados Internacionais de Catalogação na Publicação
Sistema Integrado de Bibliotecas da UFRPE
Bibliotecário(a): Auxiliadora Cunha – CRB-4 1134

S719a Souza, Camila Nunes de Paula.
Análise de mensagens de Commit com IA: uma nova perspectiva para o Algoritmo SZZ / Camila Nunes de Paula Souza. – Recife, 2025.
47 f.; il.

Orientador(a): George Gomes Cabral.

Trabalho de Conclusão de Curso (Graduação) – Universidade Federal Rural de Pernambuco, Bacharelado em Ciência da Computação, Recife, BR-PE, 2025.

Inclui referências.

1. ust-in-time Software Defect Predition. 2. Inteligência Artificial. 3. Commits. 4. Algorítmos computacionais 5. ChatGPT. I. Cabral, George Gomes, orient. II. Título

CDD 004

MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO
UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO (UFRPE)
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
<http://www.bcc.ufrpe.br>

FICHA DE APROVAÇÃO DO TRABALHO DE CONCLUSÃO DE CURSO

Trabalho defendido por Camila Nunes de Paula Souza às 16:00 hs do dia 17/02/2025, de forma remota, como requisito para conclusão do curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, intitulado "*Análise de Mensagens de Commit com IA: Uma Nova Perspectiva para o Algoritmo SZZ*", orientado por prof. George Gomes Cabral e aprovado pela seguinte banca examinadora:

Prof. George Gomes Cabral
ORIENTADOR
DC/UFRPE

Prof. Kellyton dos Santos Brito
AVALIADOR
DC/UFRPE

Agradecimentos

Nada me foi dado, eu conquistei!

Essa frase resume a jornada que vivi até chegar a este momento. Ser mulher na área de tecnologia nunca foi fácil. Enfrentei desafios, dúvidas e, muitas vezes, a sensação de que não pertencia a esse espaço. Houve momentos em que pensei em desistir, e até tentei, mas sou aguerrida e não desisto tão fácil assim. Hoje, olho para trás e vejo que cada obstáculo superado foi um passo em direção a essa conquista. Finalizar este trabalho é a prova de que, tudo aquilo que eu me proponha a fazer será realizado, mesmo quando o caminho parece incerto. E cá estou eu, pronta para o próximo passo!

Aos meus pais, minha imensa gratidão. Vocês sempre fizeram de tudo para que eu tivesse tudo aquilo de que precisei, o apoio de vocês foi a base que me permitiu chegar até aqui. Cada sacrifício que fizeram por mim está refletido nesta conquista. Obrigada por acreditarem em mim, mesmo quando eu mesma duvidei.

Aos meus amigos Thiago, Elivelton, Helton e Luis, quero agradecer por terem partilhado comigo essa vivência (fica aí o trocadilho) e participado de uma forma tão incrível da minha história. Vocês foram mais do que colegas; se tornaram meus amigos, meus ouvintes, meus conselheiros. Obrigada por terem me feito chegar aqui, tenho certeza que sem vocês eu não conseguiria!

Aos meus amigos do busão (Nat, Lauro, Sivi, Edu, Carol, Nick, Miltinho), obrigada por transformarem os trajetos diários em momentos de descontração e alegria. Vocês fizeram com que a jornada até a universidade fosse muito mais do que um simples deslocamento; foi um espaço de amizade e apoio. Aos demais (Tatá, Malu, Talita, Ná, Emerson, Ivson) obrigada por estarem ao meu lado, me apoiando em todos os momentos, vocês são a minha rede de apoio, e sei que posso contar com vocês nos momentos de alegria e também nos desafios que ainda virão.

Aos meus professores da graduação, obrigada pela enorme contribuição no meu crescimento profissional, o conhecimento não é apenas poder, é liberdade. Em especial, agradeço ao meu orientador, Prof. Dr. George Gomes Cabral, pela paciência, dedicação e orientação ao longo deste trabalho.

E, finalmente, agradeço a todos por fazerem parte desta jornada. A minha aprovação no mestrado é um novo capítulo que se abre, e sei que tudo o que vivi até aqui — os desafios, as conquistas e o apoio de cada um de vocês — me preparou para esse momento. Esta conquista é nossa.

Com amor, Camila Nunes.

Lista de ilustrações

Lista de abreviaturas e siglas

API	Application Programming Interface
CNN	Convolutional Neural Network
Commit Guru	Ferramenta de análise de commits que pode precisar de explicação se for usada como sigla
ETL	Extract, Transform and Loading (Extração, Transformação e Carregamento)
GA	Algoritmo Genético
JIT-SDP	Just-in-Time Software Defect Prediction (Predição Just-in-Time de Defeitos de Software)
LLM	Large Language Models
ML	Machine Learning
MSR	Mining Software Repositories (Mineração de Repositórios de Software)
NLP	Natural Language Processing
PLN	Software Defect Prediction
SDP	Previsão de Defeitos de Software
SVC	Support Vector Classifier
SZZ	Algoritmo utilizado para identificar commits que introduzem defeitos

Resumo

Este trabalho propõe uma abordagem inovadora para aprimorar o algoritmo SZZ utilizado na identificação de commits que introduzem defeitos em sistemas de software. A metodologia proposta envolve o uso do ChatGPT, para realizar uma análise semântica das mensagens de commit, classificando-as em duas categorias: ”introduz bug”e ”não introduz bug”.

O objetivo é melhorar a confiabilidade das classificações geradas pelo SZZ, reduzindo falsos positivos e melhorando a qualidade dos dados utilizados para a geração de modelos preditivos de detecção de defeitos. Para validar a abordagem, foram realizados experimentos com duas bases de dados (Neutron e Nova), utilizando os classificadores Random Forest e SVC, além de técnicas de balanceamento como oversampling e undersampling.

Os resultados demonstram que a integração do ChatGPT ao SZZ resultou em uma redução significativa de commits erroneamente classificados como introdução de bugs, além de melhorar o desempenho dos classificadores, especialmente o Random Forest. Conclui-se que a utilização de LLMs pode aprimorar a eficácia do SZZ, contribuindo para a melhoria da qualidade de software e a eficiência na detecção de defeitos.

Palavras-chave: Just-in-time Software Defect Predition, Inteligência Artificial, Commits, SZZ, Machine Learning, ChatGPT.

Abstract

This work proposes an innovative approach to improve the SZZ algorithm used to identify commits that introduce defects in software systems. The proposed methodology involves the use of ChatGPT to perform a semantic analysis of commit messages, classifying them into two categories: ”introduces a bug” and ”does not introduce a bug”.

The objective is to improve the reliability of the classifications generated by SZZ, reducing false positives and improving the quality of the data used to generate predictive defect detection models. To validate the approach, experiments were carried out with two databases (Neutron and Nova), using the Random Forest and SVC classifiers, as well as balancing techniques such as oversampling and undersampling.

The results demonstrate that the integration of ChatGPT with SZZ resulted in a significant reduction in commits erroneously classified as introducing bugs, in addition to improving the performance of the classifiers, especially Random Forest. It is concluded that the use of LLMs can improve the effectiveness of SZZ, contributing to the improvement of software quality and efficiency in detecting defects.

Keywords: Artificial Intelligence, Commits, SZZ Algorithm, Machine Learning, ChatGPT.

Sumário

1 Introdução

Neste capítulo serão apresentados o contexto da pesquisa, bem como sua justificativa e objetivos, além de apresentar a questão de pesquisa que se pretende responder.

1.1 Engenharia de Software

1.1.1 Processo de Desenvolvimento de Software

Um conceito que podemos utilizar para compreender o processo de desenvolvimento de software foi o apresentado por Wazlawick (2013): o processo de desenvolvimento de software é formado por um conjunto de passos parcialmente ordenados, relacionados a artefatos, pessoas, estruturas organizacionais e restrições, tendo como objetivo produzir e manter os produtos de software finais requeridos.

O ciclo de vida do desenvolvimento de software segue uma abordagem iterativa e incremental, possuindo fases como planejamento, análise de requisitos, projeto, implementação, teste, implantação e manutenção. As fases do processo possuem interdependência, porém permitem ajustes a medida que novos requisitos são criados ou problemas são detectados ao longo do processo (Pressman (2019)).

Tudo começa com um plano. A fase de planejamento é o alicerce do projeto. Durante esta fase, a equipe de desenvolvimento define o escopo do projeto, os objetivos, os requisitos e os recursos necessários, como a identificação das partes interessadas, o orçamento e o cronograma. O objetivo final desta fase é criar uma visão clara do que será desenvolvido e como será alcançado.

Na fase de análise de requisitos, a equipe se concentra em compreender todas as necessidades e as especificações do sistema. Há a criação de um documento de especificação de requisitos, através da coleta de informações por meio de entrevistas e análises, que servirá como guia para todo o processo de desenvolvimento.

Com os requisitos definidos, entramos na fase de projeto. Nela, os arquitetos de software criam uma estrutura geral para o sistema, e determinam como os componentes irão interagir entre si. Além da criação de diagramas de fluxo, esquemas de banco de dados e a definição de algoritmos. Esta fase é fundamental para garantir que o software seja eficiente, escalável e capaz de atender aos requisitos estabelecidos na fase de análise de requisitos.

A fase de implementação é quando o código real é escrito. Com boas práticas

de codificação, garantia de legibilidade do código e realização de testes unitários para identificar e corrigir erros o mais cedo possível.

Após a implementação, o software passa por um processo de teste rigoroso. O objetivo é identificar e corrigir defeitos, garantindo que o software funcione conforme o esperado. Existem vários tipos de testes que podem ser realizados, incluindo testes de unidade, testes de integração, testes de aceitação do usuário e testes de desempenho.

Após o software ser testado e aprovado, é hora de implantá-lo para uso pelos usuários finais. Isso envolve a instalação do software em servidores, a configuração de sistemas e a disponibilização para os usuários. A implantação também pode incluir a migração de dados de sistemas antigos, treinamento de usuários e a criação de documentação.

A fase de manutenção e evolução é contínua. Os desenvolvedores continuam monitorando o software em produção, identificando e corrigindo problemas conforme surgem. Além disso, o software pode precisar de atualizações para adicionar novos recursos, atender a novos requisitos ou corrigir vulnerabilidades de segurança.

1.1.2 Funcionamento de Uma Fábrica de Software

Uma fábrica de software trabalha com processos de desenvolvimento de software massivos e possui uma estrutura padronizada e dividida em componentes. Os componentes são reutilizados de versões anteriores, e por esse motivo há uma rapidez no processo de construção, aumentando a eficiência do desenvolvimento de projetos através da diminuição do retrabalho. Apesar disto, os componentes são customizados de acordo com a demanda de cada projeto. Através desta padronização o processo se torna altamente produtivo, reduzindo drasticamente os custos de desenvolvimento, manutenção, operações e complexidade. E para assegurar a qualidade dos entregáveis, essa padronização de cada solução é feita por um time de profissionais responsáveis por entender as necessidades de cada negócio e dar apoio ao cliente durante todo o processo de criação de ferramentas de tecnologia de excelente custo-benefício.

1.1.3 Repositórios de Software e Defeitos de Software

Os repositórios de software funcionam como um banco de dados que armazena todo o histórico e evolução de um projeto, isto inclui: alteração de código, documentação e artefatos relacionados aos projetos. É possível enxergar que esses repositórios não possuem apenas a finalidade de ser uma ferramenta de armazenamento, mas é uma ferramenta para desenvolvimento colaborativo, trazendo a facilidade no controle de versão, a rastreabilidade de mudanças e a análise de métricas relacionadas ao desempenho do projeto e qualidade do software.

Durante o processo de desenvolvimento de software a ocorrência de bugs de programação é inevitável. De acordo com Sá (2024), essa situação ocorre por diversos motivos, como erros de lógica no código, pouca familiaridade com a linguagem de programação, compreensão insuficiente dos requisitos relacionados à tarefa ou projeto, dificuldades com a sintaxe, entre outros. Quando um desenvolvedor começa a trabalhar com uma nova linguagem, é comum que os bugs sejam mais frequentes, já que ele ainda não domina completamente a sintaxe e as especificidades dessa linguagem, isso torna a identificação e compreensão de bugs uma tarefa difícil. A complexidade de localizar e corrigir bugs aumenta conforme a linguagem, e o nível de experiência do desenvolvedor (Zhang et al. (2020)).

A identificação e a correção de bugs em software são desafios centrais no desenvolvimento de sistemas, devido ao impacto direto que falhas podem causar na qualidade, confiabilidade e segurança dos produtos, porém a inteligência artificial aliada a tecnologias de LLMs como o ChatGPT, pode ajudar a lidar com essa crescente complexidade. (Kondo (2021) ; Dantas, Rocha e Maia (2023)).

Existe uma forma de análise de defeitos de forma sistemática, e temos atribuída aos repositórios de software esta capacidade. Através da mineração de repositórios de software (Mining Software Repositories – MSR), pesquisadores e desenvolvedores conseguem identificar padrões em bugs e corrigir falhas de maneira mais eficiente. A mineração de repositórios permite:

1. Identificar commits que introduzem bugs, utilizando ferramentas como o algoritmo SZZ (Śliwerski, Zimmermann e Zeller (2005)), que rastreia alterações no código para identificar potenciais commits indutores de defeitos de software.

2. Analisar métricas de qualidade do código, como complexidade ciclomática, que podem indicar áreas propensas a defeitos.

3. Examinar mensagens de commit e relatórios de problemas para entender as causas raízes de falhas recorrentes (Hassan (2008)).

Em 2018, o Consórcio para a Qualidade da Informação e Software (CISQ) publicou um relatório intitulado ”The Cost of Poor Quality Software in the US: A 2018 Report”[1], estimando que a má qualidade de software resultou em aproximadamente US$ 2,8 trilhões em custos nos Estados Unidos. Esse valor foi distribuído entre falhas operacionais (US$ 1.275 trilhão), projetos de desenvolvimento mal sucedidos (US$ 177,5 bilhões) e vulnerabilidades de segurança (US$ 635 bilhões).

[1] Report 2018: <https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/>

Em 2020, o Consórcio para a Qualidade da Informação e Software (CISQ) publicou o relatório ”The Cost of Poor Software Quality in the US: A 2020 Report”[2], que atualiza os dados de 2018 e indica um aumento nos prejuízos associados a softwares de baixa qualidade. De acordo com o relatório, o custo de má qualidade do software nos Estados Unidos foi estimado em aproximadamente US$ 2,08 trilhões em 2020, distribuídos da seguinte forma:

Projetos de de desenvolvimento mal sucedidos: US$ 260 bilhões (aumento em relação aos US$ 177,5 bilhões em 2018), vulnerabilidade de segurança: US$ 520 bilhões (redução em relação aos US$ 635 bilhões em 2018) e falhas operacionais: US$ 1,56 trilhão (aumento em relação aos US$ 1,275 trilhão em 2018).

Comparando com o relatório de 2020, observa-se que, apesar da redução no custo associado aos sistemas legados, os prejuízos com projetos mal sucedidos e falhas operacionais aumentaram significativamente. Esse aumento evidencia o alto custo da má qualidade do software, sendo que uma grande parte desses prejuízos está relacionada a bugs.(Inforchannel (2021))

1.2 Contextualização

1.2.1 Detecção de Defeitos de Software em Arquivos de Código Fonte

A detecção de defeitos é uma etapa fundamental para reduzir custos associados a falhas e melhorar a confiabilidade do software.

Para realizar a análise de código fonte são utilizadas ferramentas para inspecionar o código-fonte sem necessariamente executar o programa. Essas ferramentas, como FindBugs (Maryland (2006)) e PMD (Team (2025)), são capazes de identificar padrões que normalmente resultam em defeitos, como variáveis não inicializadas, operações inseguras e violações de boas práticas de programação (Zeller (2009)).

Estudos também mostram que a probabilidade de defeitos pode ser prevista em arquivos específicos com base em métricas como tamanho do código, frequência de alterações e complexidade estrutural. Ferramentas como o Commit Guru [3] integram essas métricas para prever commits de risco, auxiliando na priorização de revisões e testes (Shihab (2015)).

1.2.1.1 Algoritmos de IA para Detecção de Defeitos em Software

A detecção de defeitos em software tem se beneficiado dos avanços em técnicas de aprendizado de máquina (machine learning). Esses métodos permitem a iden-

[2] Report 2020: <https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report/>
[3] Commit Guru: <http://commit.guru/repo/pip(main)>

tificação de defeitos por meio da análise de padrões em métricas de código, histórico de mudanças e dados de defeitos anteriores. A aplicação de algoritmos de machine learning nesse contexto visa aumentar a precisão e a eficiência da detecção, reduzindo custos e melhorando a qualidade do software.

De acordo com Lessmann et al. (2008), técnicas como Random Forest, Máquina de Vetores de Suporte (SVM) e Redes Neurais são amplamente utilizadas para prever defeitos em software. Random Forest, por exemplo, é frequentemente empregado devido à sua capacidade de lidar com grandes volumes de dados e capturar relações complexas entre métricas de código, como coesão e acoplamento. Já as Redes Neurais, são aplicadas para analisar padrões em dados estruturados, como históricos de commits e métricas de código-fonte.

Além disso, técnicas de Processamento de Linguagem Natural (NLP) têm sido integradas à detecção de defeitos para analisar comentários de código, mensagens de commit e documentação, identificando possíveis problemas que foram descritos textualmente. Segundo Pan et al. (2021), essa abordagem tem se mostrado eficaz para detectar defeitos que não são facilmente identificáveis por meio de métricas tradicionais.

Outro algoritmo destacado é o Algoritmo Genético (GA), que é utilizado para otimizar a seleção de características (feature selection) em modelos de detecção de defeitos, melhorando a precisão das previsões. Estudos como o de Azhagusundari e Thanamani (2013) demonstram que o GA pode ser combinado com outros métodos de machine learning para aumentar a eficácia da detecção.

Esses algoritmos, quando aplicados em bases de dados históricas de defeitos e métricas de código, permitem a construção de modelos preditivos que identificam áreas críticas do software com maior probabilidade de apresentar defeitos, e contribui para um desenvolvimento mais confiável e eficiente.

1.2.2 Detecção de Defeitos de Software no Momento do Commit

Muitos sistemas preditivos são baseados na análise de arquivos, que podem conter centenas de linhas, dependendo da distribuição do sistema e também do tamanho dos arquivos que foram desenvolvidos. As predições Just-In-Time (JIT) apresentam uma proposta alternativa à detecção de defeitos de software (SDP) a nível de arquivo, operando a nível de commit (Ramos (2020)).

Just-in-time Software Defect Prediction (JIT-SDP) consiste em uma abordagem de SDP baseada na análise instantânea do conjunto de mudanças realizadas no software (commit). Ao invés de analisar arquivos completos, a técnica foca na avaliação do risco de introdução de defeitos diretamente no momento do commit. Para isso, são

utilizadas métricas diferentes em comparação com as abordagens tradicionais de análise de arquivos, como a quantidade de linhas modificadas, quantidade de arquivos alterados, a frequência dos commits, entre outros (Ramos (2020)).

1.2.3 Rotulação de Defeitos de Software

O algoritmo SZZ (Śliwerski, Zimmermann e Zeller (2005)) é amplamente utilizado na detecção de defeitos em software para identificar commits que introduziram bugs. No entanto, conforme demonstrado por Rosa et al. (2021), o SZZ apresenta algumas limitações que impactam a qualidade da rotulação desses defeitos, esta tarefa é essencial para o treinamento de modelos de aprendizado de máquina. Uma das principais limitações é a dependência do SZZ em mensagens de commit que nem sempre são precisos ou bem descritos o suficiente para identificar corretamente a introdução de defeitos. E isso muitas vezes ocorre porque os desenvolvedores podem não descrever explicitamente os bugs corrigidos ou podem corrigir múltiplos problemas em um único commit, dificultando a associação entre commits e defeitos específicos.

Há uma outra limitação do SZZ que é a incapacidade de distinguir entre commits que corrigem defeitos e commits que introduzem novos problemas. POis um commit pode corrigir um bug, mas também adicionar novos defeitos ou alterar alguma funcionalidade já existente, o que vai acabar gerando uma rotulação imprecisa.

O estudo de Rosa et al. (2021) também destaca que a qualidade da rotulação gerada pelo SZZ é extremamente dependente da base de código e das práticas de desenvolvimento da equipe. Pois projetos com mensagens de commit pouco descritivas ou com um histórico de mudanças fragmentado tendem a gerar resultados menos confiáveis. Dessa forma, temos limitações que impactam diretamente as abordagens de Just-in-Time Software Defect Prediction (JIT-SDP), pois se esses modelos de aprendizado de máquina forem treinados com dados rotulados incorretamente podem apresentar baixa precisão e recall.

Para mitigar essas limitações, os autores sugerem a combinação do SZZ com técnicas adicionais, como a análise semântica de mensagens de commit. Além disso, a validação manual de uma amostra dos dados rotulados pode ajudar a melhorar a confiabilidade do conjunto de dados.

1.3 Problema de Pesquisa

A identificação de commits que introduzem ou corrigem falhas no código é uma tarefa essencial no desenvolvimento de modelos preditores de defeito de software. O algoritmo SZZ (Śliwerski, Zimmermann e Zeller (2005)) permite automatizar esse processo ao rastrear mudanças no histórico de commits e identificar possíveis commits

indutores de defeito. A intuição por trás do SZZ é simples: ao detectar o commit responsável pela correção de um bug, é possível rastrear os commits anteriores que introduziram o defeito, estabelecendo assim uma relação entre as mudanças no código e os problemas reportados (Kondo (2021)).

No entanto, estudos como o de Kondo (2021) destacam que a ferramenta apresenta limitações significativas, como a dependência de mensagens de commit frequentemente vagas ou subjetivas, resultando em erros de classificação. Além disso, Shihab (2015) sugere que técnicas preditivas e analíticas podem ser integradas ao processo para melhorar a confiabilidade das classificações, mas ainda há um longo caminho a ser percorrido para alcançar resultados satisfatórios.

Diante dessas limitações, surgem oportunidades para aprimorar a qualidade das classificações feitas pelo SZZ, por meio da integração de técnicas mais avançadas, como os Modelos de Linguagem de Grande Escala (LLMs), como o ChatGPT. Estes modelos possuem a capacidade de analisar e interpretar de maneira mais refinada as mensagens de commit, capturando nuances que podem passar despercebidas por métodos convencionais. Ao incorporar LLMs ao processo de rotulação, seria possível melhorar significativamente a confiabilidade do SZZ, tornando-o mais eficaz e preciso na identificação de commits que introduzem ou corrigem bugs, e, consequentemente, aprimorando a manutenção e evolução dos sistemas de software (Training (2024)).

Diante desse cenário, este estudo busca responder à seguinte questão: ”De que forma modelos de linguagem de larga escala (LLMs), como o ChatGPT, podem ser integrados ao algoritmo SZZ para melhorar a análise semântica de mensagens de commit e, consequentemente, aprimorar a precisão na identificação de commits que introduzem bugs, bem como o desempenho de classificadores preditivos baseados nesses rótulos?”

1.4 Justificativa

A detecção de defeitos em software é crucial para garantir a qualidade e a confiabilidade dos sistemas. O algoritmo SZZ desempenha um papel fundamental nesse processo, identificando commits que introduzem bugs e fornecendo dados para modelos preditivos. No entanto, o SZZ tradicional possui limitações, como a geração de falsos positivos e a dificuldade de lidar com commits emaranhados. Este trabalho propõe uma abordagem aprimorada, integrando técnicas de Processamento de Linguagem Natural (PLN) para analisar mensagens de commit e reduzir essas limitações. A melhoria na precisão do SZZ impacta diretamente a eficácia de modelos preditivos e a manutenção de sistemas, justificando a relevância desta pesquisa.

O algoritmo SZZ possui o papel de identificar commits responsáveis por intro-

duzir bugs em sistemas de software, o processo se dá através de análises das mensagens de commit e uma posterior vinculação entre commits. Cada commit possui uma messagem associada a ele, esta mensagem descreve quais alterações foram realizadas ou quais informações foram incrementadas. Quando um commit possui em sua mensagem os termos ”bug”, ”defect”, ”fix”ou afins, ele é rotulado como um commit do tipo ’corretivo’. Dessa forma, commits imediatamente anteriores a esse que alteraram os arquivos supostamente corrigidos são rotulados como indutores de defeito. Como cada commit pode realizar várias modificações em diferentes commits anteriores, vários candidatos a ’indutores de erro’ podem ser gerados e associados a um único commit, sendo alguns deles reais e outros falsos positivos.

A qualidade do SZZ impacta diretamente a eficácia de modelos preditivos e a manutenção de sistemas. Por exemplo, Herbold et al. (2019) destaca que ”o algoritmo SZZ é o padrão de fato para rotular commits de correção de bugs e encontrar mudanças que induzem defeitos para conjuntos de dados de previsão de defeitos”.

A utilização de LLMs como o ChatGPT permite uma abordagem mais refinada para a identificação de commits emaranhados, que são uma das principais fontes de erro no SZZ. A integração do ChatGPT ao SZZ representa uma oportunidade significativa para aprimorar a qualidade da detecção de defeitos em software. Ao combinar a robustez do SZZ com a capacidade de análise contextual do ChatGPT, é possível reduzir falsos positivos, melhorar a precisão dos modelos preditivos e, consequentemente, contribuir para a melhoria da qualidade do software. Essa abordagem não apenas resolve problemas existentes no SZZ, mas também abre caminho para novas pesquisas e aplicações de LLMs na engenharia de software.

1.5 Objetivos

1.5.1 Objetivo Geral

- Desenvolver uma instância aprimorada do algoritmo SZZ utilizando modelos de linguagem de larga escala (LLMs), como o ChatGPT, para refinar o processo de análise de commits e melhorar a confiabilidade na identificação de introdução de bugs. A abordagem proposta visa também avaliar o impacto desse aprimoramento no desempenho de modelos para detecção de defeitos em software.

1.5.2 Objetivos Específicos

1. Investigar o uso de modelos de linguagem de larga escala (LLMs), como o ChatGPT, para realizar uma análise semântica das mensagens de commit, classificando-as em duas categorias: introduz bug e não introduz bug.

2. Propor abordagens de aprendizado de máquina (ML) para comparar o desempenho de classificadores preditivos frente aos rótulos gerados pelo SZZ original e pelo SZZ proposto.

3. Analisar os resultados experimentais para identificar o desempenho dos métodos de aprendizado de máquina em cada categoria, comparando-os com as classificações realizadas pelo SZZ original.

1.6 Etapas da Pesquisa

Toda pesquisa utiliza-se de métodos, técnicas e procedimentos para alcançar o fim pretendido. Com base nisso, o presente estudo irá transpassar por 5 etapas que serão explicitadas a seguir.

- 1. Estudo da Literatura no Estado da Arte
  - Revisão sistemática de trabalhos sobre detecção de defeitos em software, com foco em aprendizado de máquina e técnicas de rotulação de commits, como o algoritmo SZZ.
- 2. Coleta de dados
  - Coleta de dados através do Commit Guru [4], uma ferramenta de análise e previsão de riscos em commits de software, e em seguida será realizado o carregamento das bases de dados 'Neutron' e 'Nova' e o tratamento dos dados removendo commits classificados como *Merge* para gerar um subconjunto mais relevante.
- 3. Investigação de Prompts para o ChatGPT
  - Definição do *prompt* (pergunta) mais adequada para consultar a API do ChatGPT. O objetivo é classificar as mensagens de commit quanto à possibilidade de indicarem falhas no sistema.
  - As respostas da API serão registradas em uma nova coluna (revised-label) com valores: "1"(sim, caso possua possibilidade de indicar falha) ou "2"(não, caso não possua possibilidade de indicar falaha).
- 4. Análise dos Resultados do ChatGPT
  - Comparação entre a classificação gerada pelo ChatGPT e os resultados do SZZ original. Essa etapa inclui a avaliação da confiabilidade e precisão da nova abordagem.

[4] Commit Guru: <http://commit.guru/repo/pip(main)>

- 5. Geração de Modelos Preditivos
  - Desenvolvimento de modelos preditivos para avaliar o impacto da rotulação proposta na qualidade da detecção de commits indutores de defeitos. Serão utilizadas técnicas de aprendizado de máquina para analisar a relação entre as mensagens de commit e a introdução de falhas no código, gerando insights que possam aprimorar a precisão do SZZ.

Essas etapas serão fundamentais para validar a eficácia da metodologia proposta e compreender o impacto do uso de inteligência artificial no aprimoramento do algoritmo SZZ e na mineração de repositórios de software.

1.7 Cronograma

	2024	2025	
	Dezembro	Janeiro	Fevereiro
1. Compreensão dos Dados	☑		
2. Preparação dos Dados	☑		
3. Classificação de Mensagens de Commit com o ChatGPT	☑		
4. Comparação com a Base de Dados Original		☑	☑
5. Condução de Experimentos Complementares		☑	☑
6. Escrita do Trabalho	☑	☑	☑

Figura 1 – Cronograma de Desenvolvimento do Projeto

2 Fundamentação Teórica

2.1 Detecção de Defeitos em Software

A detecção de defeitos em software utilizando algoritmos de aprendizagem de máquina (ML) é bastante eficaz em identificar módulos que possuem uma predisposição a erros, como arquivos de código-fonte ou até mesmo pacotes de software. Conforme destacado por Malhotra e Khanna (2020), técnicas de ML têm sido aplicadas para prever defeitos com base em métricas de código e histórico de mudanças, demonstrando resultados promissores em termos de precisão e recall. Quando analisadas as características dos módulos como métricas de complexidade, histórico de alterações e interações entre componentes, os modelos de aprendizagem de máquina podem prever a probabilidade de ocorrência de defeitos. Dessa forma, é possível que equipes de desenvolvimento priorizem recursos e esforços em áreas críticas do software, aumentando a eficiência do processo de garantia de qualidade.

Tendo como exemplo a pesquisa de Herbold et al. (2022), ele destaca a importância de dados refinados na identificação de defeitos, sugerindo que a precisão dos modelos pode ser melhorada caso se considere a granularidade adequada dos dados.

Além disso, a integração de modelos de linguagem natural, como o ChatGPT, no processo de análise de código-fonte, tem potencial para aprimorar a detecção de defeitos. Conforme discutido por Dias (2023), a utilização de inteligência artificial no teste de software pode fornecer *insights* sobre padrões de defeitos, permitindo a identificação de áreas problemáticas que poderiam passar despercebidas por análises humanas. A eficácia dessas abordagens depende da qualidade e relevância dos dados utilizados para treinar os modelos de ML, pois de acordo com a pesquisa de Herbold et al. (2022) os commits entrelaçados, aqueles abordam múltiplas tarefas simultaneamente como correções de bugs, melhorias de desempenho, refatorações de código e atualizações de documentação, tudo ao mesmo tempo, podem introduzir ruído nos dados, o que torna extremamente importante adotar práticas rigorosas de preparação e limpeza de dados para garantir a confiabilidade das previsões de defeitos em software.

2.1.1 Previsão de Defeitos de Software Just In Time

A predição de defeitos em software (SDP) trabalha buscando identificar módulos que provavelmente irão conter algum defeito no sistema, normalmente em níveis como pacotes, arquivos ou até classes de códigos. Já a abordagem JIT trabalha buscando prever se uma modificação específica, como por exemplo um commit, introduz

ou não um defeito. Ambas as técnicas se baseiam em melhorar a qualidade do software, mas suas diferenças são marcantes em termos de granularidade, objetivos e aplicação prática (Kamei et al. (2013) ; Tan, Luo e Li (2015)).

Em relação aos dados, as características (*features*) usadas na predição de defeitos para a representação do problema geralmente incluem métricas estáticas de código, como complexidade ciclomática, tamanho do arquivo e número de funções, além de histórico de alterações Herbold et al. (2019). Já no JIT-SDP, as características são extraídas diretamente de modificações, como o número de linhas adicionadas, removidas, ou a dispersão das alterações (entropia), além de informações sobre o autor e o histórico do commit.

Enquanto a abordagem tradicional de predição de defeitos em software (SDP) age identificando áreas que podem conter defeitos para orientar testes e revisões, a abordagem Just-In-Time Software Defect Prediction (JIT-SDP) foca no momento em que os defeitos são introduzidos: as mudanças de código (commits). A principal vantagem do JIT-SDP é a capacidade de alertar os desenvolvedores sobre possíveis defeitos logo após o código ser submetido, quando o contexto da mudança ainda está atual. Isso facilita a correção imediata, reduzindo o custo e o esforço necessários para resolver problemas mais tarde, quando o desenvolvedor pode já não se lembrar dos detalhes da implementação. Dessa forma, o JIT-SDP não apenas identifica defeitos, mas também otimiza o processo de correção e o torna menos propenso a erros adicionais.

Ambas as estratégias têm desafios, que variam de qualidade e consistência das métricas à dependência de informações contextuais que muitas vezes são subjetivas. Em resumo, as abordagens são complementares e podem ser usadas de forma integrada para maximizar a confiabilidade e eficiência no processo de desenvolvimento de software.

2.1.2 Características (*features*) utilizadas em JIT-SDP

Como já visto anteriormente, as *features* utilizadas para representar commits contém informações relevantes das mudanças realizadas nos commits. McIntosh et al. (2017) reportou um conjunto de métricas amplamente utilizadas para modelar commits, categorizadas em quatro dimensões que refletem propriedades da modificação, do histórico, da difusão e da experiência do desenvolvedor.

- Propriedades da Modificação: Incluem métricas como o número de linhas adicionadas (LA), linhas removidas (LD) e linhas totais modificadas (LT). Essas métricas avaliam o impacto direto da mudança no código.
- Propriedades do Histórico: Analisam a frequência e o contexto das alterações

em arquivos previamente modificados. Métricas como o número de subsistemas modificados (NS), número de diretórios alterados (ND) e entropia da modificação são utilizadas para identificar padrões de risco.

- Propriedades de Difusão: Avaliam a abrangência da modificação em termos de propagação. Por exemplo, o número de desenvolvedores que modificaram o arquivo (NDEV) e o intervalo de tempo desde a última alteração (AGE) são indicadores importantes.
- Propriedades de Experiência: Capturam a expertise dos desenvolvedores envolvidos na modificação. Métricas como experiência geral (EXP), experiência recente (REXP) e experiência no subsistema (SEXP) ajudam a prever a probabilidade de introdução de defeitos com base no conhecimento do desenvolvedor.

A Figura 2 apresenta uma tabela com mais detalhes a respeito de cada métrica:

Dimensão	**Nome**	**Definição**	**Fundamentação (KAMEI *et al.*, 2013)**
Difusão	*NS*	Número de subsistemas modificados.	Alterações que modificam muitos subsistemas são mais propensas a serem defeituosas.
Difusão	*ND*	Número de diretórios modificados.	Alterações que modificam muitos diretórios são mais propensas a serem defeituosas.
Difusão	*NF*	Número de arquivos modificados.	As mudanças que afetam muitos arquivos são mais propensas a serem defeituosas.
Difusão	*ENTROPIA*	Distribuição do código modificado dentro dos arquivos.	Mudanças com entropia alta são mais propensas a apresentarem defeitos, porque um desenvolvedor terá que lembrar e rastrear um grande número de mudanças espalhadas em cada arquivo.
Tamanho	*LA*	Linhas adicionadas.	Quanto mais linhas de código se adiciona nos arquivos, mais chances de se introduzir um defeito.
Tamanho	*LD*	Linhas deletadas.	Quanto mais linhas se exclui de um arquivo, mais chances de se introduzir um defeito.
Tamanho	*LT*	Linhas totais.	Quanto maior for o arquivo, mais chances de se introduzir um defeito.
Propósito	*FIX*	Determina se a mudança é ou não para corrigir um defeito.	Corrigir um defeito significa que um erro foi cometido em uma implementação anterior; portanto, pode indicar uma área em que os erros são mais prováveis.
Histórico	*NDEV*	Número de desenvolvedores que modificaram o arquivo.	Quanto maior o NDEV, maior a probabilidade de um defeito ser introduzido, porque os arquivos revisados por muitos desenvolvedores geralmente contêm pensamentos de design e estilos de codificação diferentes.
Histórico	*AGE*	Intervalo de tempo médio entre a última modificação e a atual.	Quanto menor o AGE, ou seja, quanto mais recente a última alteração, maior a probabilidade de um defeito ser introduzido.
Histórico	*NUC*	Número de alterações únicas.	Quanto maior o NUC, maior a probabilidade de um defeito ser introduzido, porque um desenvolvedor terá que recuperar e rastrear muitas alterações anteriores.
Experiência	*EXP*	Experiência do desenvolvedor.	Quanto mais experiência um desenvolvedor tiver no sistema, menos chances de introduzir um defeito.
Experiência	*REXP*	Experiência recente do Desenvolvedor.	Um desenvolvedor que freqüentemente modificou os arquivos nos últimos meses tem menos probabilidade de introduzir um defeito, porque ele estará mais familiarizado com os desenvolvimentos recentes no sistema.
Experiência	*SEXP*	Experiência do desenvolvedor no subsistema.	O desenvolvedor familiarizado com os subsistemas modificados por uma alteração tem menor probabilidade de introduzir um defeito.

Figura 2 – Descrição das métricas

Essas *features*, quando combinadas, permitem que modelos de aprendizado de máquina identifiquem padrões associados a commits indutores de defeito, resultando em previsões de commits indutores e não indutores aceitáveis (McIntosh et al. (2017)).

2.1.3 Problemas Existentes em JIT-SDP

Apesar da grande relevância e contribuição do Just-In-Time Software Defect Prediction (JIT-SDP), esta abordagem enfrenta desafios significativos que impactam a

eficácia dos modelos preditivos. Ao olharmos para os dados, temos um desbalanceamento de classes, e isso ocorre pois a quantidade de commits indutores de defeito é consideravelmente menor que a de não indutores. Dessa forma temos um conjunto de dados com exemplos negativos (commits que não introduzem defeitos) desproporcional, causando um enviesamento ao treinar esses dados em modelos de aprendizado em favor da classe majoritária, resultando em baixa sensibilidade na detecção de commits indutores de defeito. Como consequência, commits potencialmente problemáticos podem ser ignorados, reduzindo a utilidade prática da predição (McIntosh et al. (2017)).

O desenvolvimento de software está em constante mudança, adaptando-se a novos dispositivos, paradigmas e desafios (Blog (2025)). A medida que o tempo vai passando, os modelos que foram treinados com dados históricos vão perdendo a validade das suas predições devido a mudança nas características dos commits. Se os modelos não forem atualizados regularmente para refletir as mudanças dos commits, seu desempenho tende a degradar significativamente. Há também a latência de verificação, que se refere ao tempo necessário para determinar se um commit introduziu ou não um defeito. Esse problema ocorre porque alguns defeitos levam dias para serem detectados, há cenários em que essa detecção ocorre após semanas ou até mesmo meses quando algum teste é executado, ou até mesmo quando o usuário final está utilizando o sistema. Esse tempo entre a inserção de um defeito e a sua descoberta, gera um atraso na obtenção de feedbacks e consequentemente um impacto na qualidade dos rótulos utilizados para treinar os modelos, comprometendo a eficácia e introduzindo ruídos nos conjuntos de dados.

2.2 O algoritmo SZZ

Em 2005, foi apresentado um algoritmo, conhecido atualmente como SZZ, que foi projetado para identificar commits responsáveis por introduzir bugs em sistemas de software (Śliwerski, Zimmermann e Zeller (2005)). A versão inicial desse algoritmo, chamada de B-SZZ, é composta por duas etapas principais. Na primeira, ele busca identificar commits, conhecidos como commits de correção, que corrigem bugs utilizando heurísticas específicas. Essas heurísticas incluem a análise de mensagens de commit em busca de termos como ”fix”e ”bug”, bem como a utilização de identificadores, como IDs de bugs ou títulos de relatórios. Esses commits são então vinculados a bugs registrados em sistemas de rastreamento de problemas.

Na segunda etapa, o B-SZZ utiliza ferramentas de comparação, como diff, para analisar os commits de correção identificados anteriormente e determinar as linhas de código modificadas, conhecidas como hunks. Essas ferramentas destacam as diferenças entre versões do código antes e depois das modificações. A partir daí, o B-SZZ

utiliza funções como *git blame* para identificar os commits em que essas linhas foram alteradas pela última vez. Esses commits são então marcados como possíveis introdutores de bugs.

Embora amplamente reconhecido por sua utilidade em detectar commits que introduzem bugs, o B-SZZ possui limitações. Por exemplo, ele pode gerar falsos positivos devido a commits com mudanças intercaladas ou falhar em identificar commits problemáticos relacionados a alterações não rastreadas (Herzig, Just e Zeller (2016); Rodríguez-Pérez, Nagappan e Robles (2022)).

2.3 Commits Emaranhados

As limitações existentes no algoritmo SZZ são agravadas pela presença de commits emaranhados. Um commit é considerado emaranhado quando aborda múltiplos objetivos ou funções, como implementar novos recursos, corrigir bugs ou refatorar (Kawrykow e Robillard (2011)). Commits emaranhados são comuns em projetos de software, principalmente em equipes que adotam práticas ágeis, onde várias tarefas são realizadas em paralelo e integradas em um único commit.

Vários estudos anteriores destacaram que commits emaranhados representam desafios para algoritmos de classificação e análise de dados, principalmente na identificação de commits de correção de bug e de introdução de bug (Herzig, Just e Zeller (2016)).

Herbold et al. (2022) examinou 2.238 bugs em 28 projetos para medir a prevalência de commits emaranhados: eles descobriram que apenas 38% das linhas modificadas dentro de commits de correção de bug eram necessárias para corrigir um bug. De acordo com Kawrykow e Robillard (2011), a frequência com que commits não essenciais ocorrem em históricos de commits, ou seja, commits que não modificam a lógica do código-fonte é de até 15,5%.

2.4 Commit Guru

O Commit Guru é uma ferramenta que auxilia no processo de análise de repositórios de software, identificando commits potencialmente arriscados que podem introduzir bugs no sistema. Seu funcionamento envolve três etapas principais: ingestão, análise e predição (Shihab (2015)).

Na etapa de ingestão, o Commit Guru processa o repositório solicitado, armazenando localmente as mudanças e extraindo 13 métricas (Figura 2) associadas aos commits, como linhas de código adicionadas e removidas, número de subsistemas alterados e entropia da modificação. Durante esse processo, as mensagens de commit

são analisadas semanticamente para classificar as mudanças em categorias com base em palavras-chave.

Na etapa de análise, o Commit Guru identifica commits que têm maior probabilidade de introduzir bugs, utilizando as métricas e vinculando os commits corretivos (que corrigem bugs) a commits anteriores, identificando padrões que podem indicar a introdução de defeitos.

Também é possível calcular valores medianos das métricas para cada conjunto de commits, oferecendo dados estatísticos sobre os padrões de risco no repositório. Dessa forma, os desenvolvedores podem priorizar a revisão dos commits que têm maior probabilidade de introduzir problemas.

Na etapa de predição, o Commit Guru utiliza dados históricos de commits para construir modelos preditivos baseados em técnicas de aprendizado de máquina, como o Random Forest. Esses modelos são treinados com um conjunto fixo de métricas que prevê a probabilidade de cada commit introduzir defeitos, permitindo que os usuários visualizem e analisem os commits mais propensos a causar problemas.

2.5 ChatGPT

O ChatGPT é um modelo de linguagem baseado em inteligência artificial desenvolvido pela empresa OpenAI [1], projetado para compreender e gerar texto em linguagem natural de forma fluida e coerente. Ele utiliza a arquitetura GPT (Generative Pre-trained Transformer), que é fundamentada em redes neurais com a arquitetura Transformers e treinada com grandes volumes de dados textuais. Essa tecnologia é amplamente empregada em diversas aplicações, como atendimento automatizado, assistência no desenvolvimento de código, suporte à pesquisa acadêmica e criação de conteúdo.

Quando as mensagens são analisadas pelo ChatGPT, é possivel identificar commits que estão atrelados à correção de defeitos mesmo sem conter os termos ”bug”ou ”fix”. Além disso, sua capacidade de processar grandes volumes de texto permite uma análise mais abrangente do contexto das alterações, ajudando a distinguir entre commits que introduzem bugs e aqueles que realizam melhorias ou refatorações.

Por exemplo, considere uma mensagem de commit como: ”fix white spaces in db.py”. Um método tradicional, como é o caso do SZZ, pode identificar esse commit como corretivo devido ao termo fix, e rotular automaticamente algum commit anterior como indutor de defeito. No entanto, o ChatGPT pode analisar o contexto e entender que a alteração trata apenas de ajustes estéticos no código, sem impacto funcional.

[1] OpenAI: <https://openai.com/>

Essa abordagem pode reduzir falsos positivos na identificação de commits realmente responsáveis por introduzir defeitos, especialmente quando combinada com métricas de código que indicam mudanças estruturais relevantes.

Em resumo, a combinação do algoritmo SZZ, técnicas de PLN, ferramentas como o Commit Guru e modelos avançados como o ChatGPT representa uma abordagem promissora para aprimorar a qualidade de dados para a modelagem da detecção de defeitos em software. Enquanto o SZZ fornece a base para identificar commits problemáticos, o PLN e o ChatGPT oferecem a capacidade de analisar mensagens de commit de forma mais refinada, reduzindo falsos positivos e melhorando a qualidade das classificações. Já o Commit Guru atua como um coletor de dados e rotulador prévio (por usar o SZZ original para fornecer uma primeira rotulação dos dados). Essa sinergia entre métodos tradicionais e tecnologias avançadas de IA tem o potencial de aperfeiçoar a forma como defeitos são detectados e corrigidos no desenvolvimento de software, contribuindo para a melhoria da qualidade e confiabilidade dos sistemas.

3 Trabalhos Relacionados

A detecção de defeitos em software, especialmente no nível de commits, tem sido amplamente estudada na literatura, com diversas abordagens propostas para melhorar a precisão e a eficácia dos modelos preditivos. Este capítulo apresenta uma revisão de trabalhos relacionados ao tema, com foco em pesquisas que exploram técnicas avançadas para identificar commits que introduzem defeitos, como o uso de pull requests e Processamento de Linguagem Natural (PLN). A seleção dos estudos foi realizada com base em buscas em bases de dados científicas, como IEEE Xplore, ACM Digital Library e Springer, priorizando trabalhos recentes (publicados nos últimos 5 anos) e altamente citados na área de engenharia de software. Os critérios de seleção incluíram a relevância do tema, a originalidade da abordagem e a aplicabilidade dos resultados ao contexto deste trabalho.

Um dos estudos mais relevantes na área é o trabalho de Bludau e Pretschner (2022), intitulado ”PR-SZZ: How Pull Requests Can Support the Tracing of Defects in Software Repositories”. Os autores abordam um problema central do algoritmo SZZ tradicional: a dificuldade em identificar commits que introduzem defeitos de forma precisa, especialmente devido à dependência de mensagens de commit vagas ou ambíguas. Além disso, a presença de commits emaranhados, que abordam múltiplas tarefas simultaneamente, como correções de bugs e implementação de novas funcionalidades, dificulta a vinculação correta entre commits corretivos e commits indutores de defeitos. Para superar essas limitações, os autores propõem uma versão aprimorada do SZZ, chamada PR-SZZ, que utiliza pull requests para melhorar a precisão na identificação de defeitos. A abordagem baseia-se na análise de pull requests, que são revisões de código mais detalhadas e estruturadas do que commits individuais. O PR-SZZ analisa as alterações propostas nos pull requests e as vincula a commits anteriores que podem ter introduzido os defeitos. Os resultados dos experimentos, conduzidos em seis projetos de código aberto com mais de 50 mil commits e 35 mil pull requests, mostraram que o PR-SZZ conseguiu mapear adicionalmente 18% dos tickets de bugs para commits de correção, alcançando uma pontuação F1 de 0,75. Além disso, a abordagem reduziu significativamente os falsos positivos em comparação com o SZZ tradicional. No entanto, uma limitação importante do PR-SZZ é a dependência de pull requests, o que restringe sua aplicabilidade a projetos que utilizam essa prática. Em cenários onde os pull requests não são bem documentados ou onde as alterações são integradas diretamente no repositório principal sem revisão, a eficácia da abordagem pode ser comprometida.

Outro trabalho relevante é o de Hammad e Shihab (2020), intitulado ”Using Na-

tural Language Processing to Improve Software Defect Prediction". Este estudo aborda a limitação das abordagens tradicionais de predição de defeitos, que dependem principalmente de métricas de código e histórico de alterações, ignorando informações contextuais presentes em mensagens de commit e relatórios de bugs. Para superar essa lacuna, os autores propõem a integração de técnicas de Processamento de Linguagem Natural (PLN) para analisar mensagens de commit e relatórios de bugs. A abordagem combina métricas tradicionais de código, como complexidade ciclomática e histórico de alterações, com análises textuais para identificar padrões que indicam a introdução de defeitos. Foram utilizados modelos de aprendizado de máquina, como Random Forest e SVM, para prever defeitos com base nas métricas de código e nas análises textuais. Os experimentos, conduzidos em projetos de código aberto, mostraram que a integração de PLN melhorou a precisão dos modelos preditivos em comparação com abordagens tradicionais. A análise textual permitiu identificar commits que introduziram defeitos, mesmo quando as mensagens de commit não continham termos explícitos como "bug"ou "fix". No entanto, uma limitação do trabalho é a dependência de técnicas de PLN tradicionais, que podem não capturar nuances contextuais mais complexas, como a distinção entre correções de bugs e alterações cosméticas. Além disso, a abordagem pode ser menos eficaz em cenários onde as mensagens de commit são pouco descritivas ou onde há uma grande variação no estilo de escrita dos desenvolvedores.

4 Metodologia

O objetivo geral desta pesquisa é desenvolver uma instância aprimorada do algoritmo SZZ utilizando modelos de linguagem de larga escala (LLMs), como o ChatGPT, para refinar o processo de análise de commits e melhorar a confiabilidade na identificação de introdução de bugs. A abordagem proposta visa também avaliar o impacto desse aprimoramento no desempenho de modelos para detecção de defeitos em software.

4.1 Método Proposto

Neste trabalho, iremos utilizar dados de projetos de software publicamente disponíveis no Github e processados pela ferramenta CommitGuru. Por sua vez, a ferramenta CommitGuru rotula commits desses projetos como contendo ou não bugs. Essa rotulação é feita utilizando-se o algoritmo SZZ original que, como já mencionado em seções anteriores, carrega diversas limitações. Dessa forma, a ideia principal da abordagem é realizar uma verificação/correção nas rotulações atribuídas pelo SZZ de forma a identificar commits falso-positivos, ou seja, rotulados erroneamente como indutores de bugs e corrigir tais casos. Isso será feito através de uma análise semântica das mensagens de commit de forma a identificar se essas mensagens se referem a uma correção ou não de defeitos.

No algoritmo 1, df é uma base de dados onde as linhas são informações sobre commits e as colunas são as características desses commits, assim como informações complementares necessárias ao algoritmo SZZ. Dentre essas colunas estão:

- **natureza**: indica se o commit está corrigindo um defeito (corretivo); é um commit de merge de código (merge); se está adicionando uma nova feature (add feature); etc.
- $commit\_msg$: a mensagem cadastrada pelo desenvolvedor. Essa mensagem é determinante para a identificação da natureza do commit;
- **id**: identificador único do commit;
- **fixes**: caso o commit seja da classe $indutor\_defeito$, ele contém uma lista de identificadores de commmits que o corrigiram. Essa lista está localizada na coluna fixes.

- **classe**: indica se um commit é $indutor\_defeito$ ou $limpo$. É o rótulo do commit a ser passado para o classificador.

Ainda no algoritmo 1, a função $chat\_gpt(m)$ recebe como parâmetro a mensagem de commit e identifica se ela está corrigindo um defeito de software ou não.

O Algoritmo 1 funciona da seguinte maneira. Para cada commit c, se c é de natureza corretivo segundo o SZZ, a mensagem de c é avaliada semanticamente de acordo com a função $chat\_gpt()$ e caso o agente a identifique como uma mensagem não relacionada à correção de um defeito, o identificador de c ($c[id]$) é removido de todos os commits a quem ele supostamente corrigiu. Essa é a função d trecho de código da linha 1 até a linha 12.

Como mencionado no algoritmo anterior,a funão $chat\_gpt()$ realiza uma avaliação semântica da mensagem de forma a identificar se a mesma se refere à correção de um bug ou não. O ChatGPT foi utilizado por meio de sua API, com acesso à versão 4.0 do modelo. O custo associado ao uso da API foi financiado pelo orientador do trabalho, permitindo a execução das análises em larga escala. O prompt utilizado para a análise semântica das mensagens de commit foi (em inglês):

Consider the following commit message: **whitespace fixes for nova/utils.py** Does this message describes the fix of a defect that may cause a software malfunctioning or software crash, i.e., make it unexpectedly stop (1 to yes, 2 to no)?

Em negrito no *prompt* acima se encontra mensagem a ser analisada.

Uma vez que commits indevidamente classificados como corretivos pelo SZZ tenham sido removidos da lista de fixes de seus respectivos commits indutores de defeitos, todos os commits indutores de defeitos são avaliados para identificar casos onde essa lista de fixes é vazia. Nesses casos, os rótulos dos commits indutores de defeitos são então corrigidos para commits limpos, ou seja, não indutores de defeitos. Linhas 14 até à 20 do Algoritmo.

Algorithm 1: Algoritmo para rotulação de commits baseado no chat gpt

```
   Data: df
 1 for i ← 1 to n do
 2     if df[i][natureza] == corretivo then
 3         msg ← df[i][commit_msg];
 4         cls_gpt ← chat_gpt(msg);
 5         if cls_gpt != corretivo then
 6             for j ← 1 to n do
 7                 if df[i][id] ∈ df[j][fixes] then
 8                     df[j][fixes] ← df[j][fixes] − df[i][id]
 9                 end
10             end
11         end
12     end
13 end
14 for i ← 1 to n do
15     if df[i][classe] == indutor_defeito then
16         if df[i][fixes] == ∅ then
17             df[i][classe] ← limpo
18         end
19     end
20 end
```

5 Experimentos e Análises de Resultados

5.1 Comparação com Trabalhos Relacionados

Nesta seção, comparamos a abordagem proposta neste trabalho com estudos anteriores que também buscaram aprimorar a detecção de defeitos em software, especialmente aqueles que utilizaram o algoritmo SZZ ou técnicas de processamento de linguagem natural (PLN) para análise de mensagens de commit.

O algoritmo SZZ tradicional, proposto por Śliwerski, Zimmermann e Zeller (2005), é amplamente utilizado para identificar commits que introduzem defeitos em sistemas de software. No entanto, como destacado por Kondo (2021), o SZZ apresenta limitações significativas, como a dependência de mensagens de commit vagas ou subjetivas, o que pode levar a falsos positivos. A abordagem proposta neste trabalho supera essa limitação ao integrar o ChatGPT para realizar uma análise semântica mais refinada das mensagens de commit, reduzindo a ocorrência de falsos positivos e melhorando a precisão das classificações.

Estudos recentes, como o de Pan et al. (2021), têm explorado o uso de técnicas de PLN para análise de mensagens de commit e identificação de defeitos. No entanto, a maioria desses trabalhos utiliza modelos de PLN tradicionais, como análise de sentimentos ou extração de palavras-chave, que não capturam o contexto completo das mensagens de commit. A abordagem proposta neste trabalho utiliza o ChatGPT, um modelo de linguagem de grande escala (LLM), que é capaz de compreender nuances contextuais e identificar padrões que métodos tradicionais de PLN não conseguem capturar. Isso resulta em uma análise mais precisa e confiável das mensagens de commit.

5.2 Método Experimental

Para validar a eficácia da abordagem proposta, realizamos alguns experimentos. Esses experimentos foram conduzidos utilizando dois projetos de software (Neutron[1] e Nova[2]) e dois classificadores: Random Forest e Support Vector Classifier (SVC).

O algoritmo Random Forest é reconhecido por seu desempenho eficaz em dados tabulares e por sua capacidade de mitigar o risco de overfitting. Conforme destacado por Breiman (2001), ”as florestas aleatórias são uma combinação de preditores

[1] https://github.com/openstack/neutron
[2] https://github.com/openstack/nova

de árvore, onde cada árvore depende dos valores de um vetor aleatório amostrado independentemente e com a mesma distribuição para todas as árvores da floresta". Além disso, estudos como o de Matsuki, Kuperman e Dyke (2016) demonstram que o método Random Forest supera outros em sua habilidade de lidar com overfitting, explicando uma quantidade comparável ou maior de variância em medidas de leitura.

O classificador Support Vector Classifier (SVC) Vapnik (1995) foi o segundo classificador utilizado nos experimentos. O SVC é um método baseado em máquinas de vetores de suporte, que busca encontrar um hiperplano ótimo para separar as classes no espaço de características (*features*). Embora seja um classificador poderoso, o SVC pode ser sensível a conjuntos de dados desbalanceados e requer um pré-processamento mais cuidadoso, como a normalização das *features*.

O SVC foi incluído nos experimentos para fornecer uma comparação com o Random Forest em termos de desempenho em conjuntos de dados desbalanceados.

Dados desbalanceados demandam um tratamento antes de alimentarem um modelo de *machine learning*. Isso porque modelos treinados com dados assim, não conseguem generalizar tão bem as previsões para a classe minoritária. Nas nossas bases há um desbalanceamento considerável, sendo a classe de commits que contém bugs considerada a minoritária.

Para lidar com esse problema, foram aplicadas duas técnicas de balanceamento de dados:

Oversampling: A técnica de oversampling aumenta o número de exemplos da classe minoritária, replicando ou gerando novos exemplos sintéticos. Isso ajuda a equilibrar a distribuição das classes e melhora a capacidade do modelo de aprender padrões associados a commits defeituosos.

Undersampling: A técnica de *undersampling* reduz o número de exemplos da classe majoritária, removendo aleatoriamente alguns exemplos que podem ser importantes. Nesta técnica há uma probabilidade de perda de informações, o *undersampling* é útil quando o custo computacional é uma preocupação ou quando a base de dados é muito grande.

Dito isso, é importante mencionar que a necessidade de se analisar vários classificadores é de interesse secundário. De maior relevância é a identificação de potenciais impactos no poder de discriminação entre as classes antes e depois da aplicação do método proposto para refinamento na rotulação do SZZ.

5.3 Análise do Impacto do Método Proposto na Rotulação de Commits que Introduzem Defeitos

Na Figura 3 foi feita uma análise referente a quantidade de bugs diagnosticados como False e como True em ambas as bases de dados.

	SZZ Original		SZZ-GPT	
	Bug = False	Bug = True	Bug = False	Bug = True
Neutron	7608	6613	10332	3889
Nova	18134	14651	24712	8073

Figura 3 – Distribuição da Rotulação de Commits

É notório que há uma redução no número de commits classificados como ”bug”após a intervenção do modelo GPT, de cerca de 41% para a base Neutron e cerca de 44% para a base Nova. Isso pode ser visto como um efeito de refinamento do modelo SZZ, que ajudou a reduzir a quantidade de falso-positivos (commits erroneamente classificados como indutores de bug); resultando em uma melhora na qualidade das bases de dados.

A lista a seguir apresenta (em inglês) algumas mensagens que semanticamente foram classificadas como não relacionadas a bugs capazes de causar mau funcionamento ou quebra do sistema:

Neutron

- fix pep8 warnings
- Sorting correctly all imports for the Nexus Driver and Unit Test
- PEP8 fixes for setup.py
- Noticed some pep8 errors, fixed them.
- Fix unit test printing (lp837174)
- clean up code and fix some comments
- clean code and fix some comments.
- pylint and pep8 fixes.
- pylint and pep8 fix
- Fixing typo

Nova

- fix copyrights for new files, etc
- fixed doc string
- fixed typo from auth refactor
- typo fixes and extra print statements removed
- Quick fix to variable names for consistency in documentation...
- Whitespace fix.
- fix a typo.
- fix for extra space in vblade-persist
- whitespace fixes and header changes

5.4 Análise de Desempenho Preditivo

A análise de desempenho preditivo é uma etapa crucial para avaliar a eficácia da metodologia proposta.

5.4.1 Métricas Utilizadas

Para avaliar o desempenho dos modelos, foram utilizadas as seguintes métricas:

- **Precisão (Precision):** Mede a proporção de commits classificados como ”bug”que realmente são ”bug”. É calculada como:

$$\text{Precisão} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

- **Recall (Sensibilidade):** Mede a proporção de commits que são realmente ”bug”e foram corretamente classificados. É calculada como:

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

- **F1-Score:** É a média harmônica entre precisão e recall, equilibrando as duas métricas. É calculada como:

$$\text{F1-Score} = 2 \times \frac{\text{Precisão} \times \text{Recall}}{\text{Precisão} + \text{Recall}}$$

- **Acurácia (Accuracy):** Mede a proporção de predições corretas em relação ao total de predições. É calculada como:

$$\text{Acurácia} = \frac{\text{True Positives (TP)} + \text{True Negatives (TN)}}{\text{Total de Predições}}$$

5.4.2 Resultados dos Experimentos

Os experimentos foram realizados utilizando as bases de dados Neutron e Nova, ambas processadas com o algoritmo SZZ original e com a versão aprimorada pelo ChatGPT. Os resultados foram comparados utilizando as métricas mencionadas acima.

5.4.2.1 Base Neutron

Após realizar os experimentos comparamos a Base Neutron gerada pelo SZZ e a Base Neutron reprocessada pelo GPT, ambas usando no seu experimento o algoritmo Random Forest, e as classes sendo balanceadas através do modelo Oversampling.

Neutron GPT Oversampled

Algoritmo Random Forest
Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.85	0.86	0.86	2067
TRUE	0.86	0.85	0.86	2066
accuracy			0.86	4133
macro avg	0.86	0.86	0.86	4133
weighted avg	0.86	0.86	0.86	4133

Matriz de Confusão

TN - 1780	FP - 287
FN - 307	TP - 1759

Interpretação

True Negatives (TN) : O modelo acertou 1780 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 287 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 307 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 1759 vezes ao prever que contém bug .

Figura 4 – Base Neutron rotulada pelo ChatGPT (Random Forest)

Neutron Original Oversampled

Algoritmo Random Forest
Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.81	0.79	0.80	1559
TRUE	0.79	0.80	0.79	1485
accuracy			0.80	3044
macro avg	0.80	0.80	0.80	3044
weighted avg	0.80	0.80	0.80	3044

Matriz de Confusão

TN - 1239	FP - 320
FN - 297	TP - 1188

Interpretação

True Negatives (TN) : O modelo acertou 1239 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 320 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 297 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 1188 vezes ao prever que contém bug .

Figura 5 – Base Neutron rotulada pelo SZZ (Random Forest)

Verificamos que a base de dados Neutron GPT Oversampling utilizando o algoritmo Random Forest mostra um desempenho superior aos obtidos pelos dados rotulados pelo SZZ original, com alta precisão e recall equilibrados.

Em relação ao classificador SVC, os resultados obtidos se encontram nas Figuras 6 e 7.

SVC

Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.75	0.63	0.68	1559
TRUE	0.67	0.78	0.72	1485
accuracy			0.70	3044
macro avg	0.71	0.70	0.70	3044
weighted avg	0.71	0.70	0.70	3044

Matriz de Confusão

TN - 982	FP - 577
FN - 329	TP - 1156

Interpretação

True Negatives (TN) : O modelo acertou 982 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 577 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 329 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 1156 vezes ao prever que contém bug .

Figura 6 – Base Neutron rotulada pelo SZZ (SVC)

SVC

Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.75	0.63	0.68	2067
TRUE	0.68	0.79	0.73	2066
accuracy			0.71	4133
macro avg	0.71	0.71	0.71	4133
weighted avg	0.71	0.71	0.71	4133

Matriz de Confusão

TN - 1292	FP - 775
FN - 433	TP - 1633

Interpretação

True Negatives (TN) : O modelo acertou 1292 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 775 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 433 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 1633 vezes ao prever que contém bug .

Figura 7 – Base Neutron gerada pelo ChatGPT (SVC)

Quando comparamos com o SVC, notamos que foi apresentado um desempenho inferior em relação ao Random Forest nas duas versões da base, especialmente no que diz respeito ao recall e à precisão para a classe ”False”, o que pode ser visto como um sinal de que o modelo não está capturando tão bem os commits sem bugs. A melhoria apresentada pela rotulação com o modelo proposto associada ao uso do Random Forest não foi confirmada através do uso do SVC. Porém, para uma conclusão estatisticamente relevante mais experimentos devem ser conduzidos.

É possível verificar que a introdução do modelo ChatGPT para análise semântica das mensagens de commit trouxe uma melhoria no desempenho dos classificadores, especialmente o Random Forest. A redução do número de commits classificados como ”bug”após a intervenção do GPT indica um aprimoramento na precisão da base de dados. Isso também reflete na eficácia do modelo GPT em refinar as análises e ajudar os classificadores a tomar decisões mais precisas.

As Figura 8 e 9 apresentam os resultados dos experimentos com o classificador Random Forest quando aplicado o método *undersampling* para tratamento do desbalanceamento entre as classes.

Neutron GPT Undersampled

Algoritmo Random Forest
Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.76	0.77	0.76	776
TRUE	0.76	0.75	0.76	780
accuracy			0.76	1556
macro avg	0.76	0.76	0.76	1556
weighted avg	0.76	0.76	0.76	1556

Matriz de Confusão

TN - 594	FP - 182
FN - 192	TP - 588

Interpretação

True Negatives (TN) : O modelo acertou 594 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 182 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 192 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 588 vezes ao prever que contém bug

Figura 8 – Base Neutron rotulada pelo GPT (Random Forest)

Neutron Original Undersampled

Algoritmo Random Forest
Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.78	0.79	0.78	1313
TRUE	0.79	0.78	0.78	1333
accuracy			0.78	2646
macro avg	0.78	0.78	0.78	2646
weighted avg	0.78	0.78	0.78	2646

Matriz de Confusão

TN - 1036	FP - 277
FN - 299	TP - 1034

Interpretação

True Negatives (TN) : O modelo acertou 1036 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 277 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 299 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 1034 vezes ao prever que contém buç

Figura 9 – Base Neutron rotulada pelo SZZ (Random Forest)

E podemos observar que a base Neutron SZZ Undersampling teve um desempenho ligeiramente melhor no Random Forest, com métricas (Precision, Recall e F1-Score) de 0.78, em comparação aos 0.76 da base Neutron GPT Undersampling.

Agora iremos analisar os resultados obtidos utilizando o algoritmo de machine learning SVC na mesma base de dados:

SVC
Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.74	0.63	0.68	1313
TRUE	0.68	0.78	0.73	1333
accuracy			0.70	2646
macro avg	0.71	0.70	0.70	2646
weighted avg	0.71	0.70	0.70	2646

Matriz de Confusão

TN - 824	FP - 489
FN - 295	TP - 1038

Interpretação

True Negatives (TN) : O modelo acertou 824 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 489 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 295 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 1038 vezes ao prever que contém bug .

Figura 10 – Base Neutron gerada pelo SZZ (SVC)

SVC
Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.72	0.64	0.67	776
TRUE	0.68	0.75	0.71	780
accuracy			0.69	1556
macro avg	0.70	0.69	0.69	1556
weighted avg	0.70	0.69	0.69	1556

Matriz de Confusão

TN - 494	FP - 282
FN - 194	TP - 586

Interpretação

True Negatives (TN) : O modelo acertou 494 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 282 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 194 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 586 vezes ao prever que contém bug .

Figura 11 – Base Neutron gerada pelo GPT (SVC)

O desempenho foi similar entre as duas bases, mas a base Neutron GPT teve métricas um pouco mais equilibradas (por exemplo, F1-Score = 0.71 para a classe TRUE contra 0.73 para SZZ). Apesar disso, a base SZZ resultou em mais erros absolutos (Falsos Positivos e Falsos Negativos), indicando maior dificuldade para o SVC generalizar os padrões dessa base.

Agora iremos avaliar os dados obtidos pelas matrizes de confusão geradas para cada algoritmo de ML e comparar as técnicas de balanceamento:

Random Forest (Neutron GPT - Oversampled vs. Undersampled)			
Tipo de Erro	Oversampled	Undersampled	
False Positives	118	182	
False Negatives	124	192	
SVC (Neutron GPT - Oversampled vs. Undersampled)			
Tipo de Erro	Oversampled	Undersampled	
False Positives	198	282	
False Negatives	139	194	

Figura 12 – Matriz de Confusão Comparativa

O oversampling é claramente superior ao undersampling tanto para os modelos Random Forest como para o modelo SVC, onde ambos possuem uma menor taxa de erros (FP e FN). Diante dos resultados, é conclusivo que o undersampling é mais recomendado em casos onde o custo computacional é crítico ou a base é muito grande e precisa ser simplificada.

5.4.2.2 Base Nova

Foi realizado o mesmo experimento na Base Nova e comparamos a base gerada pelo SZZ com a base reprocessada pelo GPT. Neste primeiro cenário iremos analisar as métricas geradas a partir do uso do algoritmo Random Forest e o balanceamento sendo feito com o oversampling.

Nova GPT Oversampled

Algoritmo Random Forest

Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.84	0.89	0.86	4857
TRUE	0.88	0.84	0.86	5028
accuracy			0.86	9885
macro avg	0.86	0.86	0.86	9885
weighted avg	0.86	0.86	0.86	9885

Matriz de Confusão

TN - 4310	FP - 547
FN - 824	TP - 4204

Interpretação

True Negatives (TN) : O modelo acertou 4310 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 547 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 824 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 4204 vezes ao prever que contém bug .

Figura 13 – Base Nova gerada pelo GPT (Random Forest)

Nova Original Oversampled

Algoritmo Random Forest

Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.77	0.79	0.78	3606
TRUE	0.79	0.77	0.78	3648
accuracy			0.78	7254
macro avg	0.78	0.78	0.78	7254
weighted avg	0.78	0.78	0.78	7254

Matriz de Confusão

TN - 2861	FP - 745
FN - 831	TP - 2817

Interpretação

True Negatives (TN) : O modelo acertou 2861 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 745 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 831 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 2817 vezes ao prever que contém bug .

Figura 14 – Base Nova gerada pelo SZZ (Random Forest)

Para a base reclassificada pelo GPT, o modelo conseguiu um bom equilíbrio entre as classes, com alta precisão e recall. O número total de erros (falso positivo + falso negativo) foi de 1371, que representa 4,1% da base, o que é relativamente baixo.

SVC

Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.66	0.70	0.68	4857
TRUE	0.70	0.65	0.67	5028
accuracy			0.68	9885
macro avg	0.68	0.68	0.68	9885
weighted avg	0.68	0.68	0.68	9885

Matriz de Confusão

TN - 3421	FP - 1436
FN - 1742	TP - 3286

Interpretação

True Negatives (TN) : O modelo acertou 3421 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 1436 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 1742 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 3286 vezes ao prever que contém bug .

Figura 15 – Base Nova gerada pelo GPT (SVC)

SVC

Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.69	0.61	0.65	3606
TRUE	0.65	0.72	0.69	3648
accuracy			0.67	7254
macro avg	0.67	0.67	0.67	7254
weighted avg	0.67	0.67	0.67	7254

Matriz de Confusão

TN - 2197	FP - 1409
FN - 1004	TP - 2644

Interpretação

True Negatives (TN) : O modelo acertou 2197 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 1409 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 1004 vezes ao prever que não contém bug quando, na verdade, contém.

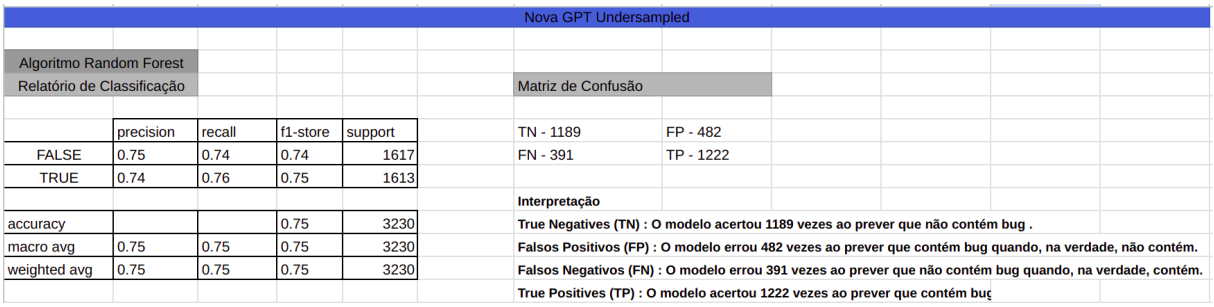
True Positives (TP) : O modelo acertou 2644 vezes ao prever que contém bug .

Figura 16 – Base Nova gerada pelo SZZ (SVC)

Quando analisamos os dados do uso do algoritmo SVC, notamos que o desempenho do mesmo foi significativamente inferior ao Random Forest, com o número total

de erros (falso positivo + falso negativo) sendo 3178, mais que o dobro do Random Forest.

Como já mencionado anteriormente, duas técnicas foram utilizadas para balanceamento de carga com objetivo de ter mais insumos para análise. A seguir iremos analisar os dados estatísticos do experimento com a técnica undersampling:



Nova GPT Undersampled

Algoritmo Random Forest

Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.75	0.74	0.74	1617
TRUE	0.74	0.76	0.75	1613
accuracy			0.75	3230
macro avg	0.75	0.75	0.75	3230
weighted avg	0.75	0.75	0.75	3230

Matriz de Confusão

TN - 1189	FP - 482
FN - 391	TP - 1222

Interpretação

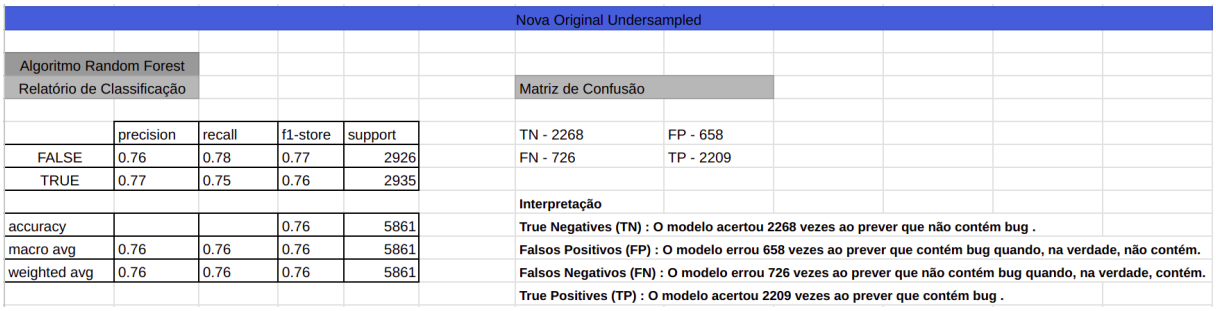
True Negatives (TN) : O modelo acertou 1189 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 482 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 391 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 1222 vezes ao prever que contém bug

Figura 17 – Base Nova gerada pelo GPT (Random Forest)



Nova Original Undersampled

Algoritmo Random Forest

Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.76	0.78	0.77	2926
TRUE	0.77	0.75	0.76	2935
accuracy			0.76	5861
macro avg	0.76	0.76	0.76	5861
weighted avg	0.76	0.76	0.76	5861

Matriz de Confusão

TN - 2268	FP - 658
FN - 726	TP - 2209

Interpretação

True Negatives (TN) : O modelo acertou 2268 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 658 vezes ao prever que contém bug quando, na verdade, não contém.

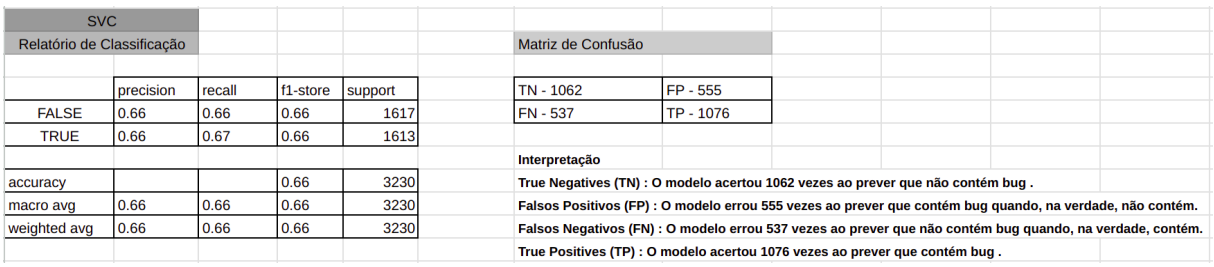
Falsos Negativos (FN) : O modelo errou 726 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 2209 vezes ao prever que contém bug .

Figura 18 – Base Nova gerada pelo SZZ (Random Forest)

A precisão, recall e F1-score da base reprocessa pelo GPT estão com valores equilibrados para ambas as classes (FALSE e TRUE), próximos de 0.74 a 0.75. A acurácia resultou em um valor de 75%, o que podemos indicar com uma boa performance sólida considerando o cenário de dados balanceados. Houve 658 falsos positivos e 726 falsos negativos, valores que são relativamente aceitáveis considerando o contexto e a técnica de undersampling.

Em seguida analisaremos o algoritmo SVC:



SVC

Relatório de Classificação

	precision	recall	f1-store	support
FALSE	0.66	0.66	0.66	1617
TRUE	0.66	0.67	0.66	1613
accuracy			0.66	3230
macro avg	0.66	0.66	0.66	3230
weighted avg	0.66	0.66	0.66	3230

Matriz de Confusão

TN - 1062	FP - 555
FN - 537	TP - 1076

Interpretação

True Negatives (TN) : O modelo acertou 1062 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 555 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 537 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 1076 vezes ao prever que contém bug .

Figura 19 – Base Nova gerada pelo GPT (SVC)

SVC				
Relatório de Classificação				
	precision	recall	f1-store	support
FALSE	0.69	0.62	0.66	2926
TRUE	0.66	0.73	0.69	2935
accuracy			0.67	5861
macro avg	0.68	0.67	0.67	5861
weighted avg	0.68	0.67	0.67	5861

Matriz de Confusão	
TN - 1822	FP - 1104
FN - 803	TP - 2132

Interpretação

True Negatives (TN) : O modelo acertou 1822 vezes ao prever que não contém bug .

Falsos Positivos (FP) : O modelo errou 1104 vezes ao prever que contém bug quando, na verdade, não contém.

Falsos Negativos (FN) : O modelo errou 803 vezes ao prever que não contém bug quando, na verdade, contém.

True Positives (TP) : O modelo acertou 2132 vezes ao prever que contém bug .

Figura 20 – Base Nova gerada pelo SZZ (SVC)

O F1-score foi de 0.66 para FALSE e 0.69 para TRUE, com uma acurácia global de 67%. O SVC novamente apresentou desempenho inferior ao Random Forest. A quantidade de falsos positivos (1104) e falsos negativos (803) é 27,4% maior do que no Random Forest , reforçando a dificuldade do SVC em lidar com dados balanceados via undersampling.

6 Conclusão

O presente trabalho propôs uma abordagem inovadora para aprimorar o algoritmo SZZ, utilizado na identificação de commits que introduzem defeitos em sistemas de software, por meio da integração de modelos de linguagem de grande escala (LLMs), como o ChatGPT. A metodologia proposta envolveu a análise semântica das mensagens de commit, classificando-as em duas categorias: ”introduz bug e ”não introduz bug”. O objetivo foi melhorar a confiabilidade das classificações geradas pelo SZZ, reduzindo falsos positivos e melhorando a qualidade dos dados utilizados para a geração de modelos preditivos de detecção de defeitos.

Os resultados demonstram que a integração do ChatGPT ao algoritmo SZZ trouxe uma melhoria significativa no desempenho preditivo, principalmente na redução de falsos positivos. Isso ocorre porque o ChatGPT é capaz de analisar as mensagens de commit levando em consideração um contexto mais amplo, identificando padrões que métodos tradicionais não conseguem capturar. Além disso, a técnica de oversampling mostrou-se mais eficaz que o undersampling para lidar com o desequilíbrio de classes, proporcionando métricas mais robustas e uma menor taxa de erros.

Porém, algumas limitações foram observadas. Podemos notar que o SVC, apesar de ser um classificador popular, apresentou um desempenho inferior ao Random Forest, em especial nos conjuntos de dados desbalanceados. Isso sugere que a escolha do classificador é um fator importante para o sucesso da abordagem proposta.

Em conclusão, a utilização de LLMs pode aprimorar a eficácia do SZZ, contribuindo para a melhoria da qualidade de software e a eficiência na detecção de defeitos. A abordagem proposta mostrou-se promissora, especialmente quando combinada com técnicas de balanceamento e classificadores robustos como o Random Forest.

6.1 Trabalhos Futuros

6.1.1 Experimentos com mais Bases de Dados

Para validar a generalização e robustez da abordagem proposta, é improtante realizar experimentos com um maior número de bases de dados, envolvendo diferentes tamanhos de projetos e práticas de desenvolvimento. Atualmente, os experimentos foram conduzidos com as bases Neutron e Nova, que pertencem ao ecossistema OpenStack. No entanto, a inclusão de projetos de outras áreas, como sistemas embarcados, aplicações web, ou até mesmo projetos de código aberto de diferentes comunidades, permitirá avaliar se a metodologia proposta é eficaz em contextos variados.

Além disso, a diversificação das bases de dados ajudará a identificar possíveis vieses ou limitações do método em cenários específicos. A expansão para mais bases de dados não apenas validará a abordagem, mas também fornecerá insights valiosos para ajustes e melhorias futuras.

6.1.2 Experimentos com mais Classificadores

Embora o Random Forest e o SVC tenham sido utilizados neste trabalho, a exploração de outros algoritmos de aprendizado de máquina pode revelar classificadores mais adequados para a tarefa de detecção de defeitos em software. A inclusão de técnicas de deep learning, como Transformers, também pode ser explorada. A comparação do desempenho desses classificadores com os resultados obtidos pelo Random Forest e SVC permitirá identificar o método mais eficaz para a tarefa, além de fornecer uma visão mais abrangente das possibilidades de aplicação de técnicas de machine learning na detecção de defeitos em software.

6.1.3 Investigação de Falsos Negativos

Enquanto este trabalho focou na redução de falsos positivos (commits erroneamente classificados como indutores de bugs), a investigação de falsos negativos (commits que introduzem bugs mas não são corretamente identificados) é igualmente crucial para melhorar a eficácia geral do modelo. Falsos negativos podem ocorrer devido a mensagens de commit ambíguas, alterações de código complexas ou até mesmo pela falta de termos explícitos que indiquem a correção de um bug. Para mitigar esse problema, sugere-se a realização de uma análise mais detalhada dos commits que foram classificados como ”não introduz bug”pelo modelo, mas que posteriormente foram associados a defeitos reportados. Essa análise pode envolver a revisão manual de uma amostra desses commits, a fim de identificar padrões ou características que o modelo atual não consegue capturar. Além disso, a integração de técnicas de análise de código estático, como a inspeção de métricas de complexidade ou a detecção de padrões de código propensos a erros, pode complementar a análise semântica das mensagens de commit, reduzindo a ocorrência de falsos negativos. A investigação de falsos negativos não apenas melhorará a precisão do modelo, mas também contribuirá para uma compreensão mais profunda das limitações e oportunidades de aprimoramento da abordagem proposta.

Referências

AZHAGUSUNDARI, B.; THANAMANI, A. S. Feature selection using genetic algorithm for software defect prediction. *International Journal of Advanced Research in Computer Science and Software Engineering*, v. 3, n. 5, p. 1–6, 2013.

BLOG, C. *Desenvolvimento de Software: As Tendências do Momento e do Futuro*. 2025. Acesso em: 10 jan. 2025. Disponível em: <https://blog.cronapp.io/desenvolvimento-de-software-as-tendencias-do-momento-e-do-futuro/#:~:text=pelo%20seu%20neg%C3%B3cio-,O%20atual%20momento%20dos%20recursos%20para%20desenvolvimento%20de%20softwares,das%20novas%20tend%C3%AAncias%20do%20mercado>.

BLUDAU, P.; PRETSCHNER, A. Pr-szz: How pull requests can support the tracing of defects in software repositories. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2022. p. 1–12.

BREIMAN, L. Random forests. *Machine learning*, Springer, v. 45, n. 1, p. 5–32, 2001.

DANTAS, C.; ROCHA, A.; MAIA, M. Assessing the readability of chatgpt code snippet recommendations: A comparative study. In: *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. Association for Computing Machinery, 2023. p. 283–292. ISBN 9798400707872. Disponível em: <https://doi.org/10.1145/3613372.3613413>.

DIAS, J. C. *Testes automatizados e inteligência artificial: Explorando novas fronteiras para qualidade de software*. 2023. Trabalho de Graduação, Universidade Federal de Pernambuco, Brasil. Disponível em: <https://www.cin.ufpe.br/~tg/2022-2/TG_CC/tg_jcd2.pdf>.

HAMMAD, M.; SHIHAB, E. Using natural language processing to improve software defect prediction. *IEEE Transactions on Software Engineering*, IEEE, v. 48, n. 4, p. 1400–1416, 2020.

HASSAN, A. E. The road ahead for mining software repositories. In: *Frontiers of Software Maintenance (FoSM)*. [S.l.]: IEEE, 2008. p. 48–57.

HERBOLD, S. et al. Empirical evaluation of bug prediction models using software repository data. *arXiv preprint arXiv:1911.08938*, 2019. Disponível em: <https://arxiv.org/abs/1911.08938>.

HERBOLD, S. et al. A refined dataset and analysis of tangling in bug-fixing commits. *Empirical Software Engineering*, v. 27, n. 6, p. 125, 2022.

HERZIG, K.; JUST, S.; ZELLER, A. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, v. 21, n. 2, p. 303–336, 2016.

INFORCHANNEL. *Prejuízo com softwares de baixa qualidade ultrapassa US$2 trilhões*. 2021. Accessed: 2025-01-05. Disponível em: <https://inforchannel.com.br/2021/01/06/prejuizo-com-softwares-de-baixa-qualidade-ultrapassa-us-2-trilhoes/>.

KAMEI, Y. et al. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, IEEE, v. 39, n. 6, p. 757–773, 2013.

KAWRYKOW, D.; ROBILLARD, M. P. Non-essential changes in version histories. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. [S.l.]: IEEE, 2011. p. 351–360.

KONDO. Why is szz incomplete? an investigation of factors that impact the accuracy of szz. *Empirical Software Engineering*, v. 26, n. 36, 2021. Disponível em: <https://link.springer.com/article/10.1007/s10664-021-10083-5>.

LESSMANN, S. et al. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, IEEE, v. 34, n. 4, p. 485–496, 2008.

MALHOTRA, R.; KHANNA, M. Machine learning techniques for software defect prediction: A systematic literature review. *Journal of Systems and Software*, Elsevier, v. 162, p. 110567, 2020.

MARYLAND, C. P. University of. *FindBugs*. 2006. Acessado em: 09 fev. 2025. Disponível em: <http://findbugs.sourceforge.net/>.

MATSUKI, K.; KUPERMAN, V.; DYKE, J. A. V. The random forests statistical technique: An examination of its value for the study of reading. *Scientific Studies of Reading*, Routledge, v. 20, n. 1, p. 20–33, 2016.

MCINTOSH, S. et al. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, Springer, v. 22, n. 2, p. 772–817, 2017. Disponível em: <https://link.springer.com/article/10.1007/s10664-016-9458-9>.

PAN, C. et al. Natural language processing for software engineering: A systematic literature review. *Journal of Systems and Software*, Elsevier, v. 182, p. 111073, 2021.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 9th. ed. New York, NY: McGraw-Hill Education, 2019. ISBN 978-1260548006.

RAMOS, I. A. *Predição de defeitos just-in-time em software utilizando inteligência artificial*. 2020.

RODRíGUEZ-PéREZ, G.; NAGAPPAN, M.; ROBLES, G. Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. *IEEE Transactions on Software Engineering*, v. 48, n. 4, p. 1400–1416, 2022. Disponível em: <https://doi.org/10.1109/TSE.2020.3021380>.

ROSA, G. et al. Evaluating szz implementations through a developer-informed oracle. In: IEEE. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. [S.l.], 2021. p. 436–447.

SHIHAB, R. . Commit guru: Analytics and risk prediction of software commits. In: *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*. [s.n.], 2015. Disponível em: <https://users.encs.concordia.ca/~eshihab/pubs/Rosen_FSE2015Tool.pdf>.

Sá, A. M. B. d. Uso de large language models (llms) para auxílio na correção de bugs. Universidade Federal de Uberlândia, 2024.

TAN, M.; LUO, L.; LI, X. Improved defect prediction using just-in-time quality assurance. *IEEE Transactions on Software Engineering*, IEEE, v. 41, n. 12, p. 1219–1238, 2015.

TEAM, P. D. *PMD*. 2025. Acessado em: 09 fev. 2025. Disponível em: <https://pmd.github.io/>.

TRAINING, S. A. *The Only Guide on AI Training Data You Will Need*. 2024. Acesso em: 10 jan. 2025. Disponível em: <https://pt.shaip.com/blog/the-only-guide-on-ai-training-data-you-will-need-in/>.

VAPNIK, V. *The Nature of Statistical Learning Theory*. New York: Springer-Verlag, 1995.

WAZLAWICK, R. S. *Engenharia de Software: conceitos e práticas*. Rio de Janeiro: Elsevier, 2013. Livro sobre conceitos e práticas em engenharia de software.

ZELLER, A. *Why Programs Fail: A Guide to Systematic Debugging*. 2nd. ed. Burlington, MA: Morgan Kaufmann, 2009. ISBN 978-0123745156.

ZHANG, J. M. et al. A study of bug resolution characteristics in popular programming languages. *IEEE Transactions on Software Engineering*, IEEE, v. 46, n. 12, p. 1390–1407, 2020. Disponível em: <https://doi.org/10.1109/TSE.2019.2961897>.

ŚLIWERSKI, J.; ZIMMERMANN, T.; ZELLER, A. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, ACM, v. 30, n. 4, p. 1–5, 2005.