UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALYSSON BISPO PEREIRA

FILTRAGEM ROBUSTA DE RUÍDO DE RÓTULO PARA PREVISÃO DE DEFEITOS DE SOFTWARE

Recife

2022

ALYSSON BISPO PEREIRA

FILTRAGEM ROBUSTA DE RUÍDO DE RÓTULO PARA PREVISÃO DE DEFEITOS DE SOFTWARE

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Área de Concentração: Inteligência Computacional

Orientador (a): Ricardo Bastos Cavalcante Prudêncio

Recife

2022

Catalogação na fonte
Bibliotecária Nataly Soares Leite Moro, CRB4-1722

P436f Pereira, Alysson Bispo
Filtragem robusta de ruído de rótulo para previsão de defeitos de software / Alysson Bispo Pereira. – 2022.
133 f.: il., fig., tab.

Orientador: Ricardo Bastos Cavalcante Prudêncio
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2022.
Inclui referências e apêndices.

1. Inteligência computacional. 2. Ruído de rótulo. 3. Predição de defeito em software. 4. Garantia de qualidade de software. I. Prudêncio, Ricardo Bastos Cavalcante (orientador). II. Título

006.31 CDD (23. ed.) UFPE - CCEN 2022 – 79

Alysson Bispo Pereira

“FILTRAGEM ROBUSTA DE RUÍDO DE RÓTULO PARA PREVISÃO DE DEFEITOS DE SOFTWARE”

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação. Área de Concentração: Inteligência Computacional

Aprovado em: 23/02/2022.

_____

Orientador: Prof. Dr. Ricardo Bastos Cavalcante Prudêncio

BANCA EXAMINADORA

_____

Prof. Dr. Cleber Zanchettin
Centro de Informática /UFPE

_____

Prof. Dr. Luciano de Andrade Barbosa
Centro de Informática /UFPE

_____

Prof. Dr. Alexandre Cabral Mota
Centro de Informática /UFPE

_____

Profa. Dra. Anne Magaly de Paula Canuto
Departamento de Informática e Matemática Aplicada / UFRN

_____

Profa. Dra. Ana Carolina Lorena
Instituto Tecnológico de Aeronáutica

AGRADECIMENTOS

Em primeiro lugar a minha família, em especial à minha mãe, meu pai e meu irmão, pela força, sabedoria, estrutura e por sempre estar ao meu lado durante toda minha jornada. À minha esposa Gabrielly Faustino, pela compreensão, apoio, carinho e paciência. À meu orientador Ricardo Prudêncio por toda paciência, prontidão e presteza durante a elaboração deste trabalho.

Enfim, a todos que contribuíram direta ou indiretamente para o desenvolvimento desse trabalho. A todos, meu muito obrigado!

RESUMO

Várias métricas de software e métodos estatísticos foram desenvolvidos para ajudar as empresas a prever defeitos de software. Tradicionalmente, para garantir a qualidade do software, o código-fonte pode ser inspecionado estaticamente por processos de revisão de código ou mesmo avaliado por meio da execução de testes por meio da execução do software. Por outro lado, os métodos de aprendizado de máquina foram treinados usando conjuntos de dados rotulados com base nas alterações de código e bugs relatados. Estudos anteriores demonstraram que esses conjuntos de dados são geralmente ruidosos devido a bugs não relatados ou devido a inconsistências nos relatórios de bug. Nesses casos, muitas instâncias de treinamento são atribuídas ao rótulo de classe errado. Como em qualquer outro contexto de aprendizado de máquina, o desempenho de um preditor de defeito depende de um conjunto de dados de treinamento confiável. Assim, evitar o uso de instâncias ruidosas na fase de treinamento pode ser crucial. As abordagens usadas até agora para detectar ruídos não consideraram métodos tradicionais de garantia de qualidade de software, como revisão de código. Neste artigo, propomos Robust Label Noise Filtering (RLNF) para aplicar técnicas de detecção de ruído de rótulo para identificar defeitos de software não relatados, identificando artefatos de software rotulados como livres de defeitos quando na verdade possuem defeitos ainda não encontrados pelos usuários finais. Para isso, estamos utilizando diferentes estratégias de detecção de ruído de rótulo para reproduzir os mecanismos usados no processo de revisão de código. Os experimentos foram realizados em um conjunto de benchmarking de projetos de software, alcançando resultados promissores.

Palavras-chaves: ruído de rótulo; predição de defeito em software; garantia de qualidade de software.

ABSTRACT

Various software metrics and statistical methods have been developed to help companies to predict defects in software. Traditionally, for software quality assurance, the source code could be statically inspected by code review processes or even evaluated by running tests through software execution. On the other hand, machine learning methods have been trained using datasets labeled based on code changes and reported bugs. Previous studies demonstrated that these datasets are usually noisy due to non-reported bugs or due to inconsistencies in the bug reports. In such cases, many training instances are assigned to the wrong class label. As in any other machine learning context, the performance of a defect predictor depends on a reliable training dataset. Thus, avoiding the usage of noisy instances in the training phase can be crucial. The approaches used so far to detect noises did not consider traditional methods for software quality assurance like code review. In this document, we propose Robust Label Noise Filtering (RLNF) to apply label noise detection techniques to identify unreported software defects by identifying software artifacts labeled as being defect-free when they actually have defects not yet found by end-users. For that, we are using different label noise detection strategies to reproduce the mechanisms used in the code review process. Experiments were performed on a benchmarking set software projects, achieving promising results.

Keywords: label noise; software defect predicition; software quality assurance.

LISTA DE FIGURAS

Figura 1 – Processo tradicional de predição de defeitos em software. 26
Figura 2 – Processo de criação de uma base de dados para PDS. 28
Figura 3 – Processo de Rotulação de Artefatos de Software 28
Figura 4 – Processo de limpeza de conjunto de dados de treinamento com ruído em rótulo. 38
Figura 5 – Ilustração do F3R 53
Figura 6 – Distribuição dos dados 61
Figura 7 – Resultados da avaliação de precisão 62
Figura 8 – Resultados da avaliação da cobertura 63
Figura 9 – Resultados do F1 score 64
Figura 10 – Distribuição dos dados 69
Figura 11 – Resultados da Precisão do estudo de caso 2 69
Figura 12 – Resultados da Cobertura do estudo de caso 2 70
Figura 13 – Resultados do F1 score do estudo de caso 2 71
Figura 14 – Distribuição de dados do Conjunto de dados AVRO 122
Figura 15 – Distribuição de dados do Conjunto de dados GIRAPH 123
Figura 16 – Distribuição de dados do Conjunto de dados IVY 123
Figura 17 – Distribuição de dados do Conjunto de dados OPENJPA 124
Figura 18 – Distribuição de dados do Conjunto de dados PROTON 124
Figura 19 – Distribuição de dados do Conjunto de dados SSHD 125
Figura 20 – Distribuição de dados do Conjunto de dados STORM 125
Figura 21 – Distribuição de dados do Conjunto de dados WHIRR 126
Figura 22 – Distribuição de dados do Conjunto de dados ZEPPELIN 126
Figura 23 – Distribuição de dados do Conjunto de dados ZOOKEEPER 127
Figura 24 – Distribuição de dados do Conjunto de dados breast-c-w 128
Figura 25 – Distribuição de dados do Conjunto de dados column2C 129
Figura 26 – Distribuição de dados do Conjunto de dados credit 129
Figura 27 – Distribuição de dados do Conjunto de dados glass1 130
Figura 28 – Distribuição de dados do Conjunto de dados heart-c 130
Figura 29 – Distribuição de dados do Conjunto de dados hill-valley 131

Figura 30 – Distribuição de dados do Conjunto de dados mushroom 131
Figura 31 – Distribuição de dados do Conjunto de dados pima 132
Figura 32 – Distribuição de dados do Conjunto de dados sonar 132
Figura 33 – Distribuição de dados do Conjunto de dados tic-tac-toe 133

LISTA DE TABELAS

Tabela 1 – Defect prediction Metrics 58
Tabela 2 – Distribuição de ruído de rótulo por projeto 60
Tabela 3 – Resultados estatísticos do F1 score 64
Tabela 4 – Algoritmos de aprendizagem para filtragem de ruídos de rótulo 66
Tabela 5 – Processo de Limpeza de ruídos, onde I.D. representa o índice desbalanceamento de cada conjunto de dados, Att. a quantidade de atributos e % Rem refere-se a porcentagem de ruídos removida. 67
Tabela 6 – Resultado do processo de desbalanceamento e inserção de ruídos para cada conjunto de dados utilizado. 67
Tabela 7 – Resultados dos testes estatísticos do F1 score do estudo de caso 2 72
Tabela 8 – Resultados do KNN utilizando K=5 88
Tabela 9 – Resultados do KNN utilizando K=10 89
Tabela 10 – Resultados do Boost utilizando N=5 e D=5 89
Tabela 11 – Resultados do Boost utilizando N=5 e D=10 90
Tabela 12 – Resultados do Boost utilizando N=10 e D=5 90
Tabela 13 – Resultados do Boost utilizando N=10 e D=10 91
Tabela 14 – Resultados do Boost utilizando N=15 e D=5 91
Tabela 15 – Resultados do Boost utilizando N=15 e D=10 92
Tabela 16 – Resultados do SEQ utilizando N=5, K=5 e D=5 92
Tabela 17 – Resultados do SEQ utilizando N=5, K=10 e D=5 93
Tabela 18 – Resultados do SEQ utilizando N=5, K=5 e D=10 93
Tabela 19 – Resultados do SEQ utilizando N=5, K=10 e D=10 94
Tabela 20 – Resultados do SEQ utilizando N=10, K=5 e D=5 94
Tabela 21 – Resultados do SEQ utilizando N=10, K=10 e D=5 95
Tabela 22 – Resultados do SEQ utilizando N=10, K=5 e D=10 95
Tabela 23 – Resultados do SEQ utilizando N=10, K=10 e D=10 96
Tabela 24 – Resultados do SEQ utilizando N=15, K=5 e D=5 96
Tabela 25 – Resultados do SEQ utilizando N=15, K=10 e D=5 97
Tabela 26 – Resultados do SEQ utilizando N=15, K=5 e D=10 97
Tabela 27 – Resultados do SEQ utilizando N=15, K=10 e D=10 98

Tabela 28 – Resultados do F3R utilizando N=5, K=5 e D=5 98
Tabela 29 – Resultados do F3R utilizando N=5, K=10 e D=5 99
Tabela 30 – Resultados do F3R utilizando N=5, K=5 e D=10 99
Tabela 31 – Resultados do F3R utilizando N=5, K=10 e D=10 100
Tabela 32 – Resultados do F3R utilizando N=10, K=5 e D=5 100
Tabela 33 – Resultados do F3R utilizando N=10, K=10 e D=5 101
Tabela 34 – Resultados do F3R utilizando N=10, K=5 e D=10 101
Tabela 35 – Resultados do F3R utilizando N=10, K=10 e D=10 102
Tabela 36 – Resultados do F3R utilizando N=15, K=5 e D=5 102
Tabela 37 – Resultados do F3R utilizando N=15, K=10 e D=5 103
Tabela 38 – Resultados do F3R utilizando N=15, K=5 e D=10 103
Tabela 39 – Resultados do F3R utilizando N=15, K=10 e D=10 104
Tabela 40 – Resultados do KNN utilizando K=5 105
Tabela 41 – Resultados do KNN utilizando K=10 106
Tabela 42 – Resultados do Boost utilizando N=5 e D=5 106
Tabela 43 – Resultados do Boost utilizando N=5 e D=10 107
Tabela 44 – Resultados do Boost utilizando N=10 e D=5 107
Tabela 45 – Resultados do Boost utilizando N=10 e D=10 108
Tabela 46 – Resultados do Boost utilizando N=15 e D=5 108
Tabela 47 – Resultados do Boost utilizando N=15 e D=10 109
Tabela 48 – Resultados do SEQ utilizando N=5, K=5 e D=5 109
Tabela 49 – Resultados do SEQ utilizando N=5, K=10 e D=5 110
Tabela 50 – Resultados do SEQ utilizando N=5, K=5 e D=10 110
Tabela 51 – Resultados do SEQ utilizando N=5, K=10 e D=10 111
Tabela 52 – Resultados do SEQ utilizando N=10, K=5 e D=5 111
Tabela 53 – Resultados do SEQ utilizando N=10, K=10 e D=5 112
Tabela 54 – Resultados do SEQ utilizando N=10, K=5 e D=10 112
Tabela 55 – Resultados do SEQ utilizando N=10, K=10 e D=10 113
Tabela 56 – Resultados do SEQ utilizando N=15, K=5 e D=5 113
Tabela 57 – Resultados do SEQ utilizando N=15, K=10 e D=5 114
Tabela 58 – Resultados do SEQ utilizando N=15, K=5 e D=10 114
Tabela 59 – Resultados do SEQ utilizando N=15, K=10 e D=10 115
Tabela 60 – Resultados do F3R utilizando N=5, K=5 e D=5 115

Tabela 61 – Resultados do F3R utilizando N=5, K=10 e D=5 116
Tabela 62 – Resultados do F3R utilizando N=5, K=5 e D=10 116
Tabela 63 – Resultados do F3R utilizando N=5, K=10 e D=10 117
Tabela 64 – Resultados do F3R utilizando N=10, K=5 e D=5 117
Tabela 65 – Resultados do F3R utilizando N=10, K=10 e D=5 118
Tabela 66 – Resultados do F3R utilizando N=10, K=5 e D=10 118
Tabela 67 – Resultados do F3R utilizando N=10, K=10 e D=10 119
Tabela 68 – Resultados do F3R utilizando N=15, K=5 e D=5 119
Tabela 69 – Resultados do F3R utilizando N=15, K=10 e D=5 120
Tabela 70 – Resultados do F3R utilizando N=15, K=5 e D=10 120
Tabela 71 – Resultados do F3R utilizando N=15, K=10 e D=10 121

LISTA DE ABREVIATURAS E SIGLAS

F3R	Filtragem Robusta de Ruído de Rótulo
KNN	*K-Nearest Neighbors*
PDS	Predição de Defeito em Software
SCV	Sistema de Controle de Versão
SRP	Sistema de Rastreamento de Problemas

SUMÁRIO

1 INTRODUÇÃO 15
1.1 CONTEXTUALIZAÇÃO 15
1.2 MOTIVAÇÃO 16
1.3 OBJETIVOS 17
1.4 ORGANIZAÇÃO 19
2 PREDIÇÃO DE DEFEITOS EM SOFTWARE 21
2.1 ESTRATÉGIAS TRADICIONAIS DA ENGENHARIA DE SOFTWARE . . . 21
2.1.1 Revisão de Código 22
2.1.2 Teste de Software 25
2.2 ESTRATÉGIAS BASEADAS EM APRENDIZAGEM DE MÁQUINA 27
2.2.1 Rotulação de Dados 28
2.2.2 Métricas de Software 30
2.2.3 Modelos de Aprendizagem de Máquina para Sistemas de PDS . . . 31
2.2.4 Desafios 32
2.3 CONSIDERAÇÕES FINAIS 34
3 DETECÇÃO DE RUÍDOS EM RÓTULOS 36
3.1 TAXONOMIA DO RUÍDO DE RÓTULO 36
3.2 ESTRATÉGIAS DE DETECÇÃO DE RUÍDOS DE RÓTULO 37
3.2.1 Modelos robustos 37
3.2.2 Métodos de Limpeza 38
3.2.3 Métodos tolerantes a ruídos de classe 40
3.3 RUÍDO DE RÓTULO NA PREDIÇÃO DE DEFEITOS EM SOFTWARE . . 40
3.4 CONSIDERAÇÕES FINAIS 43
4 FILTRAGEM ROBUSTA DE RUÍDO DE RÓTULO 44
4.1 ALGORITMO 47
4.1.1 Filtragem KNN (KNNFilter) 47
4.1.2 Filtragem Robusta de Ruído de Rótulo (F3R) 50
4.2 CONSIDERAÇÕES FINAIS 53
5 EXPERIMENTOS 55
5.1 MÉTODOS AVALIADOS 55

5.2 MEDIDA DE DESEMPENHO 56
5.3 ESTUDO DE CASO 1: RUÍDOS DE RÓTULO EM CONJUNTOS DE DADOS PARA PDS 58
5.3.1 Conjunto de Dados 58
5.3.2 Resultados 60
5.4 ESTUDO DE CASO 2: RUÍDOS DE RÓTULO EM CONJUNTOS DE DADOS DIVERSOS 65
5.4.1 Conjuntos de Dados 65
5.4.1.1 Limpeza 66
5.4.1.2 Desbalanceamento 66
5.4.1.3 Inserção de ruídos 68
5.4.2 Resultados 68
5.5 CONSIDERAÇÕES FINAIS 72
6 CONCLUSÕES 73
6.1 CONTRIBUIÇÕES 73
6.2 LIMITAÇÕES E TRABALHOS FUTUROS 75
REFERÊNCIAS 77
APÊNDICE A – RESULTADOS DO ESTUDO DE CASO 1 88
APÊNDICE B – RESULTADOS DO ESTUDO DE CASO 2 105
APÊNDICE C – CONJUNTO DE DADOS ESTUDO DE CASO 1 . 122
APÊNDICE D – CONJUNTO DE DADOS ESTUDO DE CASO 2 . 128

1 INTRODUÇÃO

Diversos métodos, técnicas, processos e ferramentas foram criados ao longo do tempo para facilitar o desenvolvimento de softwares de qualidade e de baixo custo (SMITH, 2020) (PRESSMAN; MAXIM, 2016) (LENARDUZZI et al., 2021). A execução de testes durante o desenvolvimento de software é uma das tarefas mais importantes nesse processo e demanda uma gestão de esforços e recursos para que defeitos sejam encontrados desde as primeiras fases do desenvolvimento (PIETRANTUONO et al., 2018) (DALAL; SHAH, 2017). Para isso, gestores exploram ferramentas que apoiam a tomada de decisão de alocação de recursos para a execução de testes. Uma forma de otimizar esta alocação é identificar áreas de um software que possuem mais chances de ter defeitos (WAHONO, 2015). Para tal, a Predição de Defeito em Software (PDS) vem sendo utilizada para auxiliar na identificação de partes do software com maiores chances de ter defeito e que precisam ser bem testadas (MALHOTRA, 2015).

A PDS utiliza métricas de software, informações de alterações de código e dados históricos de defeitos para treinar classificadores que terão como missão definir se um artefato de software possui defeito ou não (YANG; TANG; YAO, 2015). Um sistema de PDS geralmente é treinado a partir de artefatos de software já lançados (e.g., classes em Java), onde atributos preditores são métricas descritivas do software (e.g., número de métodos da classe) e o atributo alvo corresponde a existência ou não de defeito no artefato de software. Através de preditores de defeito, é possível identificar falhas a partir das métricas extraídas, antes que os defeitos se manifestem durante a execução do software após seu lançamento. Com a predição é possível direcionar esforços do time de desenvolvimento e teste para áreas críticas do software, reduzir o tempo gasto em testes (manuais e automáticos) e aumentar a percepção de qualidade no software (HALL et al., 2012) (THOTA et al., 2020).

1.1 CONTEXTUALIZAÇÃO

Muitas áreas se tornaram dependentes de softwares nos últimos anos, o que torna a sua qualidade um elemento crucial. Antecipar defeitos em programas deve ser um dos objetivos essenciais do processo de desenvolvimento. Falhas em software já causaram, por exemplo, um prejuízo de 125 milhões para a NASA (MICHAELS, 2008) e de 4 bilhões para ao Departamento de Defesa dos Estados Unidos (DICK et al., 2004). Apesar disso, apenas 30% das empresas

alocam um orçamento específico para realizar testes de software nos seus projetos (ARAR; AYAN, 2015).

Desta forma, existe uma busca constante por métodos que melhoram a qualidade de software (WAHONO, 2015) (THOTA et al., 2020). Estratégias convencionais de garantia de qualidade de software, como testes de software e revisão de código, são métodos convencionais utilizados para aumentar a qualidade antes do lançamento de uma versão oficial e podem consumir cerca de 23% do orçamento total de um software (ARAR; AYAN, 2015).

Técnicas de aprendizado de máquina têm sido usadas ao longo do tempo para prever erros em softwares através da construção de modelos preditivos (ÖZAKINCI; TARHAN, 2018). Nos trabalhos da área, modelos de classificação podem ser usados para prever a presença de defeitos em um artefato de software ou modelos de regressão podem ser usados para prever a quantidade de defeitos em um software (NAM, 2014). Um artefato de software pode representar uma classe, arquivo, módulo ou qualquer parte do software. Ambos os problemas (classificação e regressão) podem ser solucionados usando no aprendizado de máquina.

1.2 MOTIVAÇÃO

Estudos recentes demonstraram que dados utilizados para treinamento de sistemas de PDS podem possuir ruídos de rótulo (KIM et al., 2011) (LIU et al., 2015) (KRAWCZYK, 2016) (RIAZ; ARSHAD; JIAO, 2018). No processo de rotulagem de dados de treinamento de um sistema de PDS existe uma forte dependência de recursos humanos como testadores e desenvolvedores de software, relatórios de defeitos, relatórios de lançamento de novas versões de software e alterações de código (KIM et al., 2011) (COSTA et al., 2016) (NETO; COSTA; KULESZA, 2018). Devido à natureza do processo de rotulagem neste domínio, as instâncias defeituosas representam apenas problemas já relatados por testadores ou usuários finais, o que pode introduzir ruído: o software é rotulado como livre de defeitos, embora contenha defeitos não reportados ainda (NETO; COSTA; KULESZA, 2018). Nesse contexto, esta tese tem como foco investigar e propor soluções para o problema de ruído de rótulo em conjuntos de dados utilizados para PDS.

Devido aos ruídos, defeitos podem permanecer em um software mesmo após o lançamento de novas versões até serem reportados e consertados em versões seguintes (CHEN et al., 2014) (VANDEHEI; COSTA; FALESSI, 2021). Estudos passados estimaram que 33% dos defeitos introduzidos em uma versão de software podem não ser descobertos mesmo depois do lançamento

de várias novas versões (CHEN et al., 2014). Além disso, 18,9% dos defeitos relatados em uma versão de software podem não ter sido introduzidos na última versão de software lançada, ou seja, são defeitos inseridos em versões anteriores do software (CHEN et al., 2014).

Além do ruído de rótulo, outros problemas podem ser encontrados em conjuntos de dados utilizados para PDS. Ao longo dos anos, diversos trabalhos foram desenvolvidos com o objetivo de minimizar alguns problemas comuns. Entre eles é possível destacar trabalhos relacionados com seleção de características relevantes (WANG; KHOSHGOFTAAR; NAPOLITANO, 2010) (LIU et al., 2014) (XU et al., 2019), desbalanceamento de classes (SAIFUDIN; HERYADI et al., 2019) (CHEN et al., 2018) (PELAYO; DICK, 2007), ruído de rótulo (CHEN et al., 2021) (BASHIR et al., 2020) (TANTITHAMTHAVORN et al., 2015) e sobreposição de classes (CHEN et al., 2018) (GONG et al., 2019) (GUPTA; GUPTA, 2017) (FENG et al., 2021). Desbalanceamento, sobreposição e características irrelevantes, além de serem problemas em bases de dados para PDS, são problemas comumente encontrados em bases de dados utilizadas para aprendizagem de máquina em diversos outros domínios (JAPKOWICZ; STEPHEN, 2002) (KUMAR; MINZ, 2014). Entretanto, grande parte das soluções utilizadas para minimizar esses problemas utilizam o rótulo das instâncias como referência. Ou seja, dependem da qualidade do processo de rotulação utilizado. Desta forma, detectar ruídos de rótulo pode ser considerada uma das principais estratégias para se obter um bom desempenho em um modelo de predição de defeitos.

1.3 OBJETIVOS

O objetivo principal desse trabalho é minimizar o problema de ruídos em rótulos na PDS. É importante lembrar que a abordagem que está sendo proposta no presente documento refere-se a ruídos de rótulo e não a ruídos de atributos (DENOEUX, 2000). O problema aqui abordado é o de falhas na rotulação de instâncias em bases de dados de PDS que causam desalinhamento entre características de artefatos de softwares com suas respectivas classes. Além do objetivo principal, alguns objetivos específicos podem ser elencados:

- Desenvolvimento de uma estratégia de detecção de ruídos de rótulo para identificar defeitos ainda não reportados em softwares. Ou seja, identificar artefatos de softwares rotulados como sem defeito quando na verdade possuem defeitos ainda não encontrado por usuários finais;
- Investigação de diferentes estratégias de detecção de ruído de rótulo para reprodução dos

mecanismos utilizados em processos tradicionais de garantia de qualidade de software. Mais especificamente, a técnica de revisão de código;

- Validação da estratégia desenvolvida utilizando conjuntos de dados extraídos de softwares reais com ruídos reais;
- Validação da estratégia desenvolvida utilizando conjuntos de dados de diversos domínios com ruídos inseridos sinteticamente.

Visando alcançar os objetivos elencados, nessa tese propomos o Filtragem Robusta de Ruído de Rótulo (F3R). Trata-se de uma nova abordagem para detecção de ruído de rótulo em conjuntos de dados utilizados para PDS inspirada no processo de revisão de código estático em que desenvolvedores inspecionam o código e compartilham sugestões de alterações a fim de remover defeitos e garantir a qualidade do produto de software. A partir disso, o time de desenvolvedores pode concordar ou discordar com as sugestões de alterações e após algumas rodadas de revisão o código é aprovado e integrado ao código principal.

O Filtragem Robusta de Ruído de Rótulo (F3R) combina algoritmos baseados em vizinhança com algoritmos baseados em conjuntos de classificadores. Os algoritmos baseados em vizinhança são utilizados para simular o processo de revisão individual de cada desenvolvedor. Ou seja, algoritmos de vizinhança refletem a capacidade de um desenvolvedor realizar analogias entre os códigos desenvolvidos e revisados até o momento e os códigos que precisam de revisão a fim de encontrar defeitos. Os algoritmos baseados em comitês representam o processo de revisão de código feito por diversos desenvolvedores que podem compartilhar opiniões e concluir que um artefato de software tem defeito ou não. No processo de revisão de código cada desenvolvedor pode ter atuado na codificação e revisão de diferentes partes do software, portanto possuem níveis de experiência diferentes que são necessários para uma boa revisão de código.

Para simular diversos desenvolvedores com níveis de experiência diferentes, foi utilizado o algoritmo de *Adaboost*. O *Adaboost* aprende novos classificadores a partir de algumas iterações, concentrando-se em instâncias consideradas a priori como de difícil classificação, ou seja, que foram classificadas incorretamente em iterações anteriores. Através de um esquema de pesos, instâncias de difícil classificação são replicadas dentro do conjunto de dados fazendo com que classificadores tenham mais exemplos de cada instância problemática e reduza a quantidade de erros de classificação. Em paralelo ao processo de revisão de código, cada desenvolvedor aqui

é representado por um classificador que é treinado a cada iteração com diferentes amostragens do mesmo conjunto de dados de treinamento. Consequentemente, desenvolvedores com mais experiência possuem mais chances de terem revisado ou desenvolvido artefatos de software similares aos de difícil classificação que podem representar ruídos de rótulo, podendo assim rotular com mais propriedade como instância ruidosa ou não.

Além disso, problemas similares aos encontrados em bases de dados de sistemas de PDS, tais como desbalanceamento e sobreposição, também são encontrados pelos desenvolvedores durante o processo de revisão de código. As chances de erro são reduzidas quando desenvolvedores compartilham experiências e principalmente conseguem realizar analogias entre os códigos em revisão e os já revisados. Outra característica interessante na revisão de código é que desenvolvedores mais experientes geralmente são os responsáveis pela revisão final e integração do código. Ou seja, depois que diversos desenvolvedores com níveis de experiência diferentes realizaram inspeções no código e já evidenciaram problemas mais evidentes. Desta forma, desenvolvedores mais experientes podem visualizar todos os comentários adicionados e dar destaque aos artefatos de software ainda não apontados como problemáticos.

Diversos aspectos do processo de revisão de código foram reproduzidos na estratégia aqui apresentada. A revisão de código é uma prática bastante popular para garantia da qualidade do software e de fácil integração com diversas metodologias de desenvolvimento de software. Apesar disso, é um processo que exige atenção e experiência dos desenvolvedores. Desta forma, o uso de técnicas de aprendizagem de máquina que podem representar os mecanismos hoje utilizados no processo de revisão de código pode aumentar as chances de detecção de ruídos, ou seja, defeitos presentes em um software que ainda não foram reportados por usuários finais.

1.4 ORGANIZAÇÃO

Este documento está organizado da seguinte forma: no Capítulo 2 são apresentadas as estratégias tradicionais da engenharia de software para garantia de qualidade, as estratégias baseadas em aprendizagem de máquina para PDS e os desafios encontrados na PDS; no Capítulo 3 é abordado o problema de detecção de ruído em rótulo e algumas estratégias que podem ser utilizadas para amenizar esse problema; no Capítulo 4 é apresentado o F3R e detalhes de sua implementação da solução; no Capítulo 5 é apresentado os métodos avaliados, as medidas de desempenho utilizadas e os resultados obtidos de dois estudos de caso realizado; e por fim, o Capítulo 6 traz as principais observações sobre os resultados obtidos e alguns

trabalhos futuros.

2 PREDIÇÃO DE DEFEITOS EM SOFTWARE

O código-fonte de um software passa por diversas mudanças durante o processo de desenvolvimento e manutenção que visam corrigir defeitos e implementar novos requisitos. Além de trazer benefícios, alterações no código de um software podem introduzir novos problemas, gerando retrabalho e aumento de custos. A fim de reduzir a quantidade de defeitos nos softwares, tarefas de garantia de qualidade de software podem ser aplicadas desde as primeiras fases do desenvolvimento para aumentar a confiabilidade e qualidade antes da implantação de uma nova versão oficial em ambiente de produção (GALIN, 2004).

Alguns cenários podem deixar o processo de detecção de defeitos em software cada vez mais desafiador. Diversas empresas realizam entregas contínuas de novas versões de software como forma de otimizar o processo de desenvolvimento, exigindo também que o teste de software seja executado com mais frequência. Nesse contexto, sempre que um novo código é produzido, testes manuais e automatizados devem ser realizados para garantir a qualidade do código. Preferencialmente, um software deve ser testado a cada nova /versão lançada, mas testar completamente um software a cada novo lançamento é uma tarefa custosa e complexa. Além disso, para algumas organizações o teste de software por si só não é suficiente para garantir a qualidade de um software (BAHAROM, 2020).

Portanto, dado o tamanho e a complexidade dos projetos de software atuais, as abordagens para detectar defeitos devem ser escaláveis e robustas (LI; LESSMANN; BAESENS, 2019). Desta forma, a introdução de estratégias que auxiliem na predição de defeitos em software podem otimizar atividades de teste de software e reduzir o número de defeitos encontrados em ambientes de produção (TURHAN et al., 2009). A definição de defeito pode ser descrita a partir das definições de erro, defeito e falha (IEEE, 1990). Um erro é uma ação de um desenvolvedor que resulta em um defeito. Um defeito por sua vez é a manifestação de um erro no código que vai levar a uma falha em tempo de execução. A seguir, serão apresentadas as estratégias que auxiliam na detecção de defeitos em softwares.

2.1 ESTRATÉGIAS TRADICIONAIS DA ENGENHARIA DE SOFTWARE

Qualidade de software é um assunto bastante explorado na Engenharia de Software. Diversas estratégias e processos foram desenvolvidas ao longo dos anos a fim de reduzir o máximo

possível o número de defeitos em softwares. Nesse conjunto de estratégias encontram-se estratégias que usam diferentes níveis de abstração do software. O software pode ser analisado de forma estática em processos de revisão de código ou até mesmo através da execução de rotinas de testes no software em funcionamento. Nessa seção, serão apresentadas duas abordagens tradicionais de garantia de qualidade de software.

2.1.1 Revisão de Código

O processo de revisão de código, também conhecido como inspeção de código, é uma estratégia de análise de código feita por desenvolvedores que inspecionam cada linha do código fonte de um software a fim de sugerir melhorias ou até mesmo detectar defeitos (FAGAN, 1999) (SADOWSKI et al., 2018). A revisão de código foi incorporada muito bem no desenvolvimento de software ágil como uma das formas de garantir a qualidade de cada artefato de software desenvolvido. O processo de revisão de código pode ser realizado de algumas formas (COHEN et al., 2006):

- Em pares: nesse tipo de revisão dois desenvolvedores, geralmente um mais experiente que o outro, escrevem e revisam o código utilizando apenas uma estação de trabalho (FAGAN, 1999). Para ter sucesso nesse tipo de abordagem é necessário realizar discussões e revisões contínuas para detectar defeitos e também compartilhar conhecimento (WARDEN; SHORE, 2007);

- *Over-the-shoulder*: nessa estratégia o autor não realiza a revisão do seu código, sua missão é a de guiar alguns desenvolvedores no entendimento do código que foi escrito a partir de reuniões (COHEN et al., 2006). O autor do código pode realizar as alterações à medida que os revisores presentes nessa reunião vão sugerindo melhorias. Diferentemente da revisão em pares, aqui são envolvidos mais desenvolvedores e existe um compartilhamento ainda maior de conhecimento dentro do time. Entretanto, o fato do autor guiar os outros desenvolvedores na revisão do código pode não favorecer a identificação de problemas que poderiam ser identificados caso a revisão fosse feita de forma individual;

- *Email pass-around*: esse processo é bem semelhante ao *over-the-shoulder*, mas sem a necessidade de realização de uma reunião para revisão do código. Em vez disso, é enviado por email para os revisores os códigos que precisam de inspeção. Para facilitar esse processo, sistemas de controle de versão de software podem ser utilizados para

indicar quais arquivos foram alterados, adicionados ou deletados. Aqui os revisores podem realizar o exame do código de acordo com sua disponibilidade e ainda podem deixar comentários em linhas específicas dos códigos para facilitar a resolução dos problemas encontrados. Esse tipo de revisão foi utilizada por muitos anos pelos desenvolvedores de software *open source* (COHEN et al., 2006);

- Baseada em ferramentas: algumas ferramentas especializadas em revisão de código podem ser utilizadas para realizar inspeções no código. A partir de uma análise estática do código, artefatos de software que talvez possam apresentar problemas ou que não estejam seguindo boas práticas de programação podem ser evidenciados (AXELSSON et al., 2009).

- Processo moderno de revisão de código: recentemente ferramentas foram criadas para facilitar o processo de revisão de código. Em vez de utilizar email, agora desenvolvedores conseguem submeter o código diretamente para sistemas que auxiliam na revisão de código e no versionamento do projeto de software (SADOWSKI et al., 2018). Aqui desenvolvedores fazem a revisão de código adicionando comentários para sugerir melhorias e esses comentários podem ser visualizados por todos os membros do time. Após uma série de revisões, a alteração de código pode ser aprovada e integrada no código principal disponibilizados para todos os desenvolvedores (BELLER et al., 2014).

Estratégias baseadas em ferramentas são bastante utilizadas para a automatização do processo de revisão de código, uma vez que podem detectar problemas comuns e mais previsíveis (TUFAN et al., 2021). Em algumas situações, esse tipo de estratégia é utilizada antes de uma revisão de código mais formal feita por outros membros do time de desenvolvimento (FAN et al., 2018). Ferramentas de revisão de código também são utilizadas em ambientes de desenvolvimento fornecendo sugestões de melhoria e indicando problemas comumente encontrados durante o processo de escrita do código.

A decisão de qual estratégia de revisão de código utilizar pode ser tomada a partir das limitações e características dos times de desenvolvimento. Em alguns casos a programação em pares é utilizada para facilitar a inserção de novos colaboradores. Como os desenvolvedores já estão em pares, naturalmente inspeções vão ser executadas em pares. A criação de pares pode potencializar o desenvolvimento de um membro sem muita experiência, mas pode ser um problema para a revisão de código. Além de exigir muita interação entre dois desenvolvedores,

a revisão em pares possui como grande limitação a quantidade de indivíduos envolvidos. Os desenvolvedores atuam tanto no desenvolvimento como na revisão e acabam não realizando uma boa revisão do código por estarem muito imersos no problema (BEGEL; NAGAPPAN, 2008).

Com exceção das estratégias baseadas em ferramentas, todas as estratégias de revisão de código envolvem diretamente os desenvolvedores. Assim, a experiência de cada desenvolvedor é um fator determinante para uma boa revisão de código (CAULO et al., 2020) (PASCARELLA et al., 2018) (RUANGWAN et al., 2019) (CAULO et al., 2020). Como cada desenvolvedor provavelmente possui um nível diferente de experiência e dificilmente todos os desenvolvedores participarão do desenvolvimento de todos os módulos de um sistema, é importante que exista uma troca de conhecimento intensa durante o processo de revisão de código.

O processo de revisão de código é colaborativo (KALYAN et al., 2016). Desenvolvedores podem sugerir alterações no código para remover defeitos e outros desenvolvedores podem concordar ou discordar de sugestões de alterações realizadas por outros desenvolvedores. Ou seja, os comentários e revisões que são feitas ao longo do processo de inspeção de código até que um software seja aceito podem ser visualizados por todos os desenvolvedores (HOSSAIN et al., 2020). Desenvolvedores mais experientes geralmente são responsáveis pela aprovação final e integração do código, consequentemente possuem horas de trabalho mais caras e mais conhecimento no código desenvolvido (TÜZÜN; TEKINERDOGAN, 2015). Desta forma, partes do código que foram apontadas como propensas a defeitos por poucos desenvolvedores sem muita experiencia podem vir a ser definidas como defeituosas ou não após a revisão de desenvolvedores com mais experiência (MCINTOSH et al., 2014) (THONGTANUNAM et al., 2017).

Independente da estratégia utilizada, o processo de revisão do código tem como objetivos (BAUM et al., 2016):

- Melhorar a qualidade do software: aspectos analisados durante a revisão de código como uniformidade, legibilidade e compreensibilidade são fundamentais para aumentar a qualidade do software produzido e otimizar manutenções no futuro. Além disso, muitos códigos revisados serão de correções de defeitos, portanto a revisão coletiva de uma remoção de defeito em um código reduz as chances do mesmo defeito se repetir em novas implementações;

- Encontrar defeitos: é esperado que com o processo de revisão de código, defeitos difíceis de serem encontrados por testes sejam detectados. Apesar de não existir a execução de

código, como no processo tradicional de teste, as inspeções feitas por desenvolvedores experientes podem apontar problemas;

- Aprendizagem coletiva: o processo de inspeção de código é enriquecedor tanto para autores como para revisores. O autor de um código pode receber sugestões de melhoria de outros desenvolvedores com diferentes níveis de experiência. Ao mesmo tempo, desenvolvedores podem ter contato com implementações de diferentes módulos dentro de um software, o que pode ajudar no desenvolvimento de novas funcionalidades no futuro e no entendimento do software como um todo. Outro aspecto importante é o compartilhamento das revisões do mesmo código, i.e., todos os desenvolvedores podem visualizar as revisões feitas por outros desenvolvedores, podendo concordar ou até mesmo discordar dessas revisões. Portanto, o apontamento de um possível defeito feito por um desenvolvedor pode vir a ser confirmado por outros revisores.

Apesar do processo de revisão de código fornecer muitos benefícios, é preciso também atentar para algumas consequências desse processo. O esforço utilizado para revisar códigos poderia ser utilizado para desenvolver outras tarefas, portanto pode aumentar os custos e o tempo do ciclo de desenvolvimento do software (EGELMAN et al., 2020). Em geral, a revisão de código é um processo que exige atenção e experiência dos revisores. Além disso, é um processo que não inclui a execução do código, portanto muitas falhas presentes no software podem não ser encontradas. Estudos anteriores apontam que em grandes softwares comerciais apenas 20% das linhas de código chegam a ser revisadas em processos tradicionais de revisão de código (ORAM; WILSON, 2010). Ou seja, um processo caro que necessita de alocação de recursos e que não é muito eficiente. Desta forma, diversos times de desenvolvimento de software recorrem ao teste de software para encontrar defeitos.

2.1.2 Teste de Software

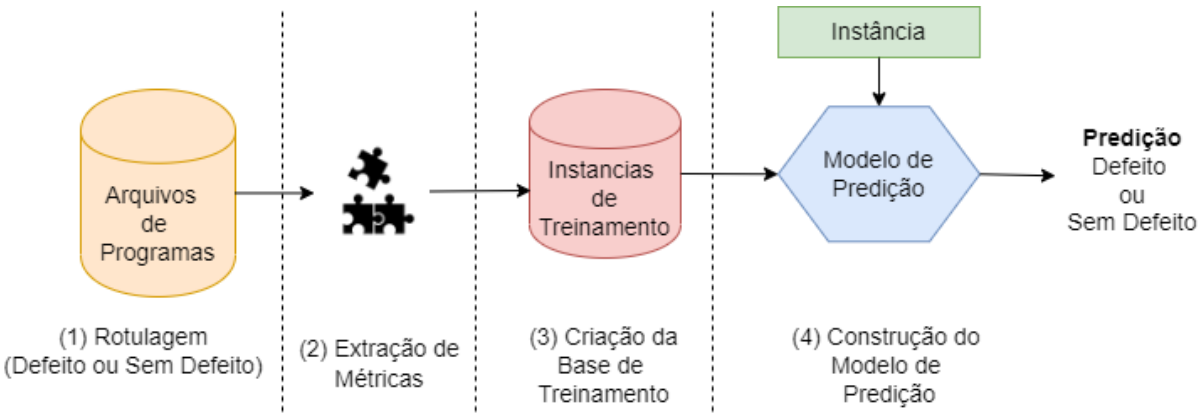
Atividades de teste são essenciais para garantia da qualidade de um software. Teste de software é o processo que executa o software simulando cenários de execução com o objetivo de encontrar problemas e de verificar se o software está sendo construído de acordo com suas especificações (MYERS; SANDLER; BADGETT, 2011). O teste de software pode ser aplicado em diferentes níveis do software (AMMANN; OFFUTT, 2016):

- Teste Funcional: também conhecido como teste caixa-preta ou teste orientado a dados, é o tipo de teste que avalia as entradas, saídas e estados de um software baseado nos requisitos especificados (MYERS; SANDLER; BADGETT, 2011). Neste tipo de teste, o software é visto como uma caixa preta cujo o conteúdo, ou seja, o código fonte, é completamente desconhecido.

- Teste Estrutural: este tipo de teste é conhecido como teste caixa-branca, em que o código fonte também é considerado para implementação dos testes (DELAMARO; JINO; MALDONADO, 2013). O teste estrutural é baseado em fluxos de dados que testam as diferentes regras e condições implementadas no software.

O teste de software envolve algumas variáveis que o tornam mais caro e demorado que processos de revisão de código. O número de possibilidades de execução de um software geralmente é alto, portanto é preciso garantir que os principais cenários estarão cobertos (FRASER; ROJAS, 2019). Tradicionalmente, a distribuição de defeitos em um software tende a ser desigual entre os artefatos, ou seja, utilizar o mesmo esforço para testar todos os elementos de um software pode não ser uma boa estratégia (GKORTZIS; FEITOSA; SPINELLIS, 2021) (MOHAGHEGHI et al., 2004). Além disso, testes de software precisam ser realizados seguindo alguns princípios para que falhas encontradas também sejam reproduzidas pelos desenvolvedores e que sejam corrigidas (JOORABCHI; MIRZAAGHAEI; MESBAH, 2014).

Figura 1 – Processo tradicional de predição de defeitos em software.



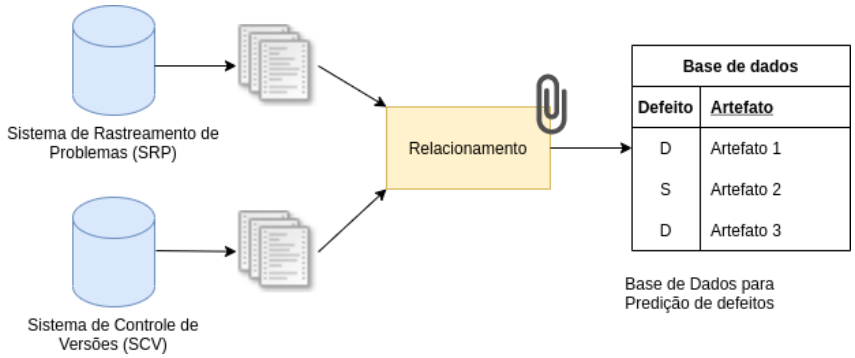
Fonte: (NAM, 2014)

2.2 ESTRATÉGIAS BASEADAS EM APRENDIZAGEM DE MÁQUINA

Além das estratégias tradicionais da engenharia de software apresentadas, existem iniciativas que visam aumentar a qualidade do software de forma automatizada e escalável sem necessariamente utilizar recursos humanos. Com o auxílio de abordagens de aprendizagem de máquina, sistemas de Predição de Defeito em Software (PDS) são utilizados para prever quais artefatos de um software são propensos a ter defeito (JURECZKO; MADEYSKI,) (TURHAN et al., 2009). Modelos tradicionais de PDS podem atuar de duas formas: (1) classificando artefatos de software como defeituoso ou não ou (2) estimando a quantidade de defeitos em artefatos de software através de modelos de regressão (NAM, 2014). Nesta seção, serão apresentadas as etapas necessárias para criação de um preditor de defeitos em software através de modelos de classificação.

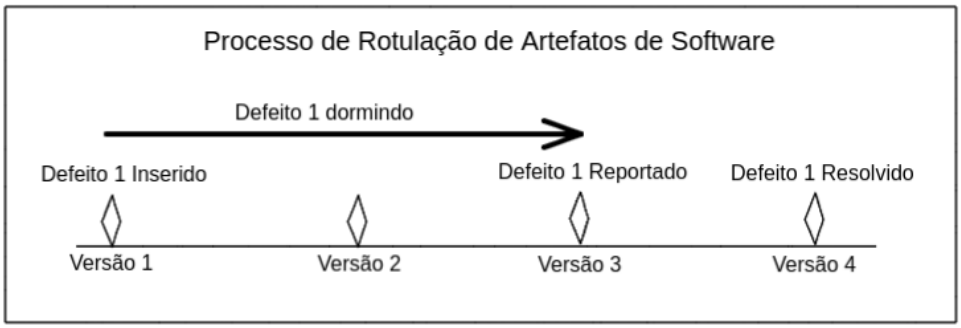
A Figura 1 apresenta o processo tradicional de construção de um classificador capaz de classificar artefatos de software como defeituosos ou não. Para a construção de uma base de treinamento, considera-se um conjunto de artefatos de software, que podem ser classes, métodos ou até mesmo arquivos que sejam necessários para o funcionamento do software. Como se trata de um problema de classificação, os artefatos do software devem ser etiquetados como defeituosos ou não defeituosos. Este processo geralmente é feito com o auxílio de sistemas de rastreamento de não conformidades usados para reportar problemas e de ferramentas de versionamento de código. Em seguida, o processo de extração de métricas extrai um conjunto de características de cada artefato de software, que são utilizadas como atributos preditores. Geralmente, são utilizadas medidas de complexidade de código como o número de linhas de código, número de métodos em uma classe, dentre outras. A partir disso, é criada uma base de dados para treinamento do modelo de predição com os artefatos de software caracterizadas a partir das métricas de software e rotulados. Por fim, um modelo de predição é treinado utilizando um algoritmo da aprendizagem de máquina. O modelo construído será usado para rotular novas instâncias de artefatos de software baseado nos valores das métricas descritivas. A seguir, serão apresentados mais detalhes de cada etapa do processo de criação de um sistema de PDS.

Figura 2 – Processo de criação de uma base de dados para PDS.



Fonte: (TANTITHAMTHAVORN et al., 2015)

Figura 3 – Processo de Rotulação de Artefatos de Software



Fonte: Elaborada pelo autor (2022)

2.2.1 Rotulação de Dados

A rotulação dos artefatos de software é uma das etapas iniciais do processo de criação de um sistema de PDS. Esse processo realiza a definição de quais instâncias de um software tem defeito, sendo fundamental para o treinamento do modelo de predição. Como é possível observar na Figura 2, a definição do rótulo de cada artefato de software necessita de dados um Sistema de Rastreamento de Problemas (SRP) e um Sistema de Controle de Versão (SCV). SRP podem ser utilizados por usuários para reportar problemas e também para auxiliar no gerenciamento do desenvolvimento do software a partir da criação de tarefas. Portanto, o SRP pode gerar relatórios de defeitos reportados, novos requisitos implementados ou até mesmo refatorações feitas em código. O SCV atua no gerenciamento das alterações de fato no código. A partir do SCV são extraídas informações de arquivos e linhas de código que foram alteradas para atender cada tarefa ou problema cadastrados no SRP.

Desta forma, a rotulação de um artefato de software como defeituoso ou não acontece a partir do relacionamento de um defeito reportado no SRP e o código alterado para resolver esse

defeito no SCV. Uma das abordagens de rotulação mais tradicionais é a realista. Essa utiliza a versão de software reportada como referência para rotular como defeituoso os artefatos de software que precisaram ser alterados para resolver o problema. Utilizando a Figura 3 como referência, o Defeito 1 foi relatado na Versão 3 de um software e será corrigido com o lançamento de Versão 4. Usando a Abordagem Realista, o conjunto de artefatos de software alterados para resolver o Defeito 1 deve ser rotulado na Versão 3 como defeituoso. Isso significa que a Abordagem Realista assume que o defeito foi incluído na versão de software em que o defeito foi reportado, neste caso Versão 3.

Portanto, a abordagem realista realiza a rotulagem na versão em que o defeito foi reproduzido, mas ignora quando o código problemático de fato foi introduzido. Nesse contexto, outras abordagens foram criadas ao longo do tempo para melhorar o processo de rotulação. Uma abordagem bastante explorada por sistemas de PDS é a abordagem SZZ (ŚLIWERSKI; ZIMMERMANN; ZELLER, 2005). O SZZ utiliza um mecanismo de anotações do SCV para identificar quais alterações de código inseriram linhas específicas de um artefato de software e auxiliar na identificação de todas as versões de software que já possuíam o defeito reportado. Isso significa que o SZZ pode detectar defeitos latentes e rotular artefatos de software usando as Versões Afetadas pelo problema de fato. Ainda usando a Figura 3, pela abordagem SZZ, Defeito 1 na verdade foi inserido na Versão 1 e não na Versão 3 como constatado pela Abordagem Realista. Portanto, segundo abordagens avançadas como o SZZ, os artefatos de software alterados para resolver o Defeito 1 devem ser rotulados como defeituosos nas Versão 1, Versão 2 e Versão 3.

Vale a pena salientar que defeitos podem ser inseridos na Versão 3 e permanecer na Versão 4 e também em novas versões do software sem ser descoberto pelo fato de nunca ter sido reportado no SRP. Nessas situações, tanto a abordagem SZZ como a realista não irão detectar esse ruído de rótulo. Além disso, mesmo sendo mais eficaz que a abordagem realista, a abordagem SZZ apenas retroage versões anteriores do software para identificar onde o defeito foi inserido. Ou seja, para identificar ruídos em versões de software recém lançadas, faz-se necessário o uso de estratégias mais robustas para detectar o ruído antes que o defeito seja reportado.

2.2.2 Métricas de Software

A Extração de Métricas de um software é uma etapa muito importante na criação de um sistema de PDS. Métricas são medidas quantitativas que podem avaliar diversos aspectos relacionados com o processo de desenvolvimento do software e que podem influenciar na inserção de defeitos. A seguir, serão apresentadas alguns tipos de métricas que podem ser extraídas de um software (CHOUDHARY et al., 2018):

- Métricas de código e complexidade: nesse grupo estão as métricas que indicam o quão complexo é um artefato de software. A complexidade de um código pode aumentar o esforço de manutenção e também aumentar as chances de encontrar defeito no software. Dentro desse categoria de métricas existem métricas simples, que apenas contam linhas e métodos por exemplo, como também métricas mais sofisticadas. Um bom exemplo de métrica mais sofisticada é a métrica de Complexidade Ciclomática (CC) que mede a complexidade da estrutura de decisão de um artefato de software através do cálculo do número de caminhos independentes alcançáveis na execução do código-fonte (EBERT et al., 2016);

- Métricas orientadas a objeto: aqui estão métricas que avaliam o código de acordo com o paradigma de orientação a objetos. Esse tipo de métrica pode avaliar a quantidade de classes, heranças, construtores, métodos públicos, dentre outros (CHIDAMBER; KEMERER, 1994) (BANSIYA; DAVIS, 2002);

- Métricas de processo: métricas de processo são métricas que focam nas alterações de código feitas ao longo do tempo (CHIDAMBER; KEMERER, 1994). Mudanças de código tendem a incluir novos defeitos mesmo sem aumentar a complexidade do código (MOSER; PEDRYCZ; SUCCI, 2008). São exemplos de métricas de processo: o número de linhas inspecionadas em revisões de código, quantidade de reinspeções de código devido a sugestões de desenvolvedores, números de desenvolvedores alterando o mesmo artefato de software em uma mesma versão, dentre outras;

- Métricas de dependência: esse tipo de métrica foca em medir o nível de dependência entre os artefatos de um software (SCHRÖTER; ZIMMERMANN; ZELLER, 2006). Níveis altos de dependência entre artefatos de software aumentam as chances de um artefato interferir no funcionamento de outro. A dependência pode aparecer em importações

de bibliotecas ou até mesmo devido à orientação a objeto através do uso de heranças, polimorfismos e interfaces.

Métricas de software podem também ser extraídas a partir dos testes realizados no software, podendo servir de referência para estimar a qualidade do software. Esse tipo de métrica pode avaliar a quantidade de testes executados para cada artefato de software ou até mesmo mensurar o percentual de cobertura de linhas de código do artefato de software nos testes executados (NAGAPPAN, 2004).

2.2.3 Modelos de Aprendizagem de Máquina para Sistemas de PDS

Analisar um artefato de software a partir das suas características e definir se o mesmo contém defeito ou não é um problema típico de classificação, ou seja, é preciso aprender um classificador que seja capaz de classificar novos artefatos de software a partir de um conjunto de artefatos de softwares rotulados e caracterizados por um conjunto de métricas. Mais especificamente, trata-se de um problema de classificação binária em que dois rótulos são possíveis: defeito ou sem defeito. Desta forma, diversos algoritmos de aprendizagem de máquina que realizam classificação foram explorados ao longo dos anos para aprimorar a detecção de defeitos em software (OKUTAN; YILDIZ, 2014) (LI; JING; ZHU, 2018) (LI et al., 2017) (WAHONO, 2015).

Estratégias tradicionais de classificação que utilizam similaridade como o *K-Nearest Neighbors* (KNN) podem ser utilizadas para aproximar artefatos de softwares semelhantes e ajudar na classificação de instâncias com base na sua vizinhança. Dado um artefato do software X, o KNN analisa o rótulo das K instâncias mais próximas com o objetivo de identificar qual seria o rótulo de X.

Apesar de ser uma técnica antiga e bastante explorada, o KNN ainda é uma técnica que apresenta bons resultados na detecção de defeitos (IQBAL et al., 2019) (MABAYOJE et al., 2019) (WAHONO; HERMAN; AHMAD, 2014). Dado o contexto da revisão de código em que são alterados alguns artefatos de software, o KNN pode detectar que existem outros artefatos defeituosos com as mesmas características dos artefatos problemáticos corrigidos que precisam ser corrigidos também. Ou seja, apenas um desenvolvedor com conhecimento em grande parte dos artefatos de um software conseguiria realizar esse tipo análise.

Além do KNN, abordagens que utilizam comitês de classificadores também foram bastante

exploradas em sistemas de PDS. Trata-se do uso de conjuntos de classificadores combinados para detecção de defeitos. Em bases de dados utilizadas por sistemas de PDS, uma técnica bastante explorada é o *AdaBoost* (RIAZ; ARSHAD; JIAO, 2018) (WANG; YAO, 2013) (ZHANG et al., 2020) (ALSAWALQAH et al., 2017). Diferentemente do KNN, no *AdaBoost* não é preciso analisar a vizinhança para classificar instâncias com defeituosas ou não. O *AdaBoost* tenta aprender novos classificadores a partir de algumas iterações, concentrando-se em instâncias consideradas a priori como de difícil classificação/aprendizagem, ou seja, que foram classificadas incorretamente em iterações anteriores. Através de um esquema de pesos, instâncias de difícil classificação são replicadas dentro do conjunto de dados fazendo com que classificadores tenham mais exemplos de cada instância problemática e pare de errar na classificação das mesmas. Ao final de algumas iterações, é esperado que o classificador minimize o erro de classificação e consiga aprender a classificar instâncias com o rótulo correto. Apesar de ser uma estratégia promissora, concentrar os recursos em amostras suspeitas, que podem ser de difícil classificação ou ter ruído de rótulo, pode levar ao sobreajuste (FRÉNAY; VERLEYSEN, 2014) (RÄTSCH; ONODA; MÜLLER, 2001) (DIETTERICH, 2000) (GROVE; SCHUURMANS, 1998).

2.2.4 Desafios

Em geral, o desempenho de modelos de classificação dependem de uma distribuição equilibrada de representantes das classes em questão, ou seja, o número de instâncias de cada classe deve ser balanceado o máximo possível para que o modelo de predição criado seja capaz de classificar corretamente todas as classes (BATISTA; PRATI; MONARD, 2004) (GALAR et al., 2011). Tradicionalmente, bases de dados utilizadas para treinamento de preditores de defeitos em artefatos de software têm poucas instâncias com defeito quando comparado com a quantidade de instâncias sem defeito, ou seja, existe um forte desbalanceamento entre as classes (ZHANG; ZHANG, 2007) (HE; GARCIA, 2009). Assim como em outras áreas, esse problema pode causar falhas na classificação de instâncias da classe minoritária. Em bases de dados utilizadas para treinar sistemas de PDS por exemplo, instâncias com defeito, mesmo sendo o foco dos classificadores e das medidas de desempenho, podem nunca ser encontradas ou ser preditas com muitos erros (PELAYO; DICK, 2007).

É possível dividir as estratégias utilizadas para lidar com o problema do desbalanceamento de classe em algumas categorias (SUN; WONG; KAMEL, 2009):

- Abordagens com foco nos dados: nesse subgrupo estão estratégias que buscam fazer reamostragem dos dados. Esse tipo de abordagem pode aumentar a representatividade da classe minoritária, diminuir a representação da classe majoritária ou até mesmo combinar essas duas estratégias para reduzir o nível de desbalanceamento entre as classes. A grande diferença entre essas estratégias é a forma de escolher as instâncias que serão duplicadas no caso da sobre-amostragem e de quais instâncias serão removidas quando se faz a subamostragem;

- Abordagens com foco nos classificadores: aqui estão as estratégias que tentam resolver o problema do desbalanceamento dentro do próprio algoritmo de classificação. Alterações em classificadores exigem um nível aprofundado de conhecimento do algoritmo de classificação, o sucesso desse grupo de estratégias está no entendimento das razões que levam o algoritmo de aprendizagem de máquina falhar quando os dados utilizados para treinamento não estão balanceados.

Outro problema comum na PDS é a sobreposição de classes (CHEN et al., 2018) (GONG et al., 2019) (GUPTA; GUPTA, 2017). Esse fenômeno ocorre quando elementos de um conjunto de dados rotulado aparentam-se como exemplos válidos para duas classes distintas, ou seja, possuem características semelhantes mas pertencentes a duas classes diferentes (VUTTIPITTAYAMONGKOL; ELYAN; PETROVSKI, 2021). Nas bases de dados utilizadas para PDS a classe minoritária é frequentemente sobreposta pela classe majoritária (CHEN et al., 2018). Além disso, existe uma forte ligação entre o nível de sobreposição com o desbalanceamento de classes (PRATI; BATISTA; MONARD, 2004). Muitas estratégias de sobreamostragem que visam diminuir o desbalanceamento entre as classes como o SMOTE (CHAWLA et al., 2002) na realidade tendem a aumentar a sobreposição (LÓPEZ et al., 2013) (WANG; JAPKOWICZ, 2004).

Alguns autores defendem que o problema da sobreposição está relacionado com a baixa qualidade dos dados utilizados para treinamento (GONG et al., 2019). Ou seja, a falta de uma vasta diversidade de métricas de software pode ser prejudicial para a separação de dois artefatos que pertencem a diferentes classes (CHEN et al., 2018). Uma solução possível é a remoção de instâncias em regiões de sobreposição, considerando-as como ruídos. Entretanto, a simples remoção artefatos de regiões com sobreposição pode remover completamente uma determinada classe. Além disso, artefatos em sobreposição podem ser apenas elementos de difícil aprendizagem (GUPTA; GUPTA, 2018).

É possível também identificar falhas no processo de rotulação de artefatos de software em bases de dados de sistemas de PDS. O SRP é utilizado por desenvolvedores e usuários de uma aplicação, usuários podem abrir chamados para requisitar novas funcionalidades e reportar problemas. Antes de atender um chamado, um processo de triagem é realizado por desenvolvedores para analisar as requisições, estimar esforço de desenvolvimento e análise final para definição se o chamado é de fato um defeito ou uma solicitação de um novo requisito. Desta forma, o processo de rotulação de artefatos de softwares que depende da análise dos usuários e desenvolvedores sobre o que foi reportado. Tendo em vista a fragilidade desse processo, alguns estudos investigaram o impacto que pode ser gerado pelo ruído gerado no processo de rotulação para sistemas de PDS (KHAN et al., 2020) (TANTITHAMTHAVORN et al., 2015) (XU et al., 2016) e estratégias para lidar com esse problema (KIM et al., 2011) (BASHIR et al., 2020) (PANDEY; TRIPATHI, 2021).

Desta forma, o processo de rotulagem é desafiador e propenso a erros pelo fato de utilizar recursos humanos como testadores e desenvolvedores, relatórios de erros, relatórios de versão e alterações de código (COSTA et al., 2016) (NETO; COSTA; KULESZA, 2018). Assim, os relatórios de erros podem introduzir alarmes falsos de defeitos, pois os desenvolvedores e usuários podem interpretar "defeitos" de maneira diferente. Como consequência, ruídos podem ser introduzidos no conjunto de dados de treinamento (HERZIG; JUST; ZELLER, 2013). Além disso, instâncias positivas (defeitos) abrangem apenas os defeitos já relatados pelos testadores ou usuários finais, o que podem introduzir outro tipo de ruído no rótulo: o artefato de software é rotulado como livre de defeitos mesmo contendo defeitos ainda não relatados (NETO; COSTA; KULESZA, 2018). Portanto, as instâncias de um software podem ser rotuladas com uma etiqueta incorreta devido a falhas de comunicação, problemas de categorização ou até falhas de especialistas no domínio. A necessidade de interpretação e inspeção manual em alterações de código torna difícil a criação de um conjunto de dados livre de ruídos sem o envolvimento de desenvolvedores e especialistas no domínio. Por essa razão, diversos trabalhos anteriores avaliaram o impacto do ruído apenas incluindo ruídos sintéticos nos conjunto de dados (HERZIG; JUST; ZELLER, 2013) (FAN et al., 2019) (NETO; COSTA; KULESZA, 2018) (NAZARI et al., 2018) (KIM et al., 2011).

2.3 CONSIDERAÇÕES FINAIS

Nesse capítulo, foi introduzida a predição de defeitos em software e como ela pode ser utilizada para otimizar os gastos com teste de software. Além disso, foi apresentado todo o processo

para que seja possível construir um preditor de defeitos em software a partir de controle de versões de código e sistemas de gerenciamento de defeitos. Diversas métricas de software podem ser utilizadas para medir a complexidade das instâncias de um software, tais como: métricas estáticas, métricas de processo, métricas de dependência e até métricas de teste. Tais métricas são essenciais para treinar classificadores que irão definir se um determinado artefato de software possui ou não defeito. Além disso, é necessário atentar para os diversos desafios encontrados na criação de um sistemas de PDS, em especial o problema de rotulação dos dados. Ruídos nos rótulos podem prejudicar bastante o desempenho de um sistema de PDS e até inviabilizar o uso de estratégias tradicionais para minimizar o desbalanceamento e sobreposição de classes presentes nessas bases de dados.

3 DETECÇÃO DE RUÍDOS EM RÓTULOS

Classificação é um dos tópicos mais estudados na área de aprendizagem de máquina (KOTSIANTIS et al., 2007). A abordagem tradicional consiste em treinar um classificador a partir de um conjunto de dados rotulados. Esses conjuntos são compostos por exemplos que possuem características, também chamados de atributos, e pertencem a uma classe específica. A partir desses exemplos um classificador aprende uma função que mapeia características dos exemplos com suas respectivas classes para realizar futuras classificações (BISHOP, 2006). Tendo em vista que a rotulação de dados de forma confiável costuma ser cara e demorada, é importante introduzir estratégias que garantam a qualidade e reduzam a quantidade de ruídos em base de dados utilizadas por algoritmos de classificação (KUHA; SKINNER; PALMGREN, 2014).

Ruídos em rótulos podem surgir a partir de falhas de comunicação, falhas de categorização ou até mesmo falhas de especialistas no domínio (FRÉNAY; VERLEYSEN, 2014). Tal problema pode aumentar a necessidade de instâncias de treinamento, prejudicar a precisão das classificações e até mesmo aumentar a complexidade do treinamento (ZHU; WU, 2004). Por estas razões, esse problema vem sendo tratado por diversos trabalhos nos últimos anos (FRÉNAY; VERLEYSEN, 2014) (XIA et al., 2020).

3.1 TAXONOMIA DO RUÍDO DE RÓTULO

Com base em trabalhos anteriores (NETTLETON; ORRIOLS-PUIG; FORNELLS, 2010) (BENTLER; YUAN, 1999), (FRÉNAY; VERLEYSEN, 2014) propuseram uma taxonomia para os tipos de ruído em rótulo:

- Ruídos Completamente Aleatórios: ocorre quando as instâncias com rótulos incorretos estão uniformemente presentes em todas as regiões de um conjunto de dados, ou seja, as chances de encontrar uma instância com rótulo incorreto é independente dos atributos e da própria classe. Em um problema de classificação binária, a mesma porcentagem de ruídos de rótulo devem ser encontrada para as duas classes;

- Ruídos Aleatórios: neste tipo de ruído a probabilidade do erro de rotulação depende da classe. Vale salientar que aqui ainda não existe nenhuma relação entre atributos de classe e rótulos incorretos. Isso permite a modelagem de ruídos de rótulo assimétrico

entre as classes, ou seja, quando instâncias de certas classes são mais propensas a serem rotuladas incorretamente;

- Ruídos Não Aleatórios: para esse grupo os ruídos não dependem apenas da classe da instância, mas também dos seus atributos. Aqui estão os ruídos de rótulo presentes em instâncias que geralmente são consideradas como de difícil classificação. Por exemplo, instâncias que são mais similares a instâncias de outras classes, instâncias em fronteira de decisão entre classes ou até mesmo instâncias com poucos exemplos semelhantes, ou seja, em regiões menos povoadas nos conjuntos de dados.

A taxonomia do ruído de rótulo é importante para determinar quais estratégias poderiam ser mais indicadas para lidar com o problema. A seguir, serão apresentadas algumas estratégias de detecção de ruídos de rótulo.

3.2 ESTRATÉGIAS DE DETECÇÃO DE RUÍDOS DE RÓTULO

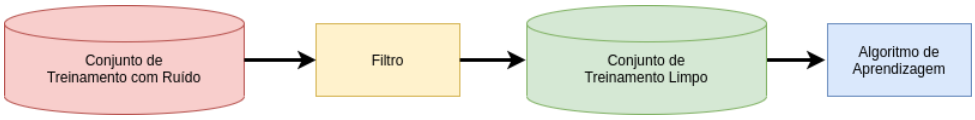
Diversas estratégias podem ser utilizadas para lidar com ruídos em rótulos. De acordo com Frénay e Verleysen (2014), é possível dividir as estratégias em três grupos: métodos robustos, estratégias de limpeza e modelos tolerantes a ruído de rótulo. A seguir, serão apresentados mais detalhes de cada grupo de estratégia.

3.2.1 Modelos robustos

Esse grupo de abordagens estão algoritmos que são considerados robustos o suficiente para classificar na presença de poucos ruídos de rótulo. Ou seja, não seria necessário realizar filtragem para remoção de instâncias ou até mesmo ajustes no modelo. Para classificação, modelos robustos podem ser criados utilizando comitês de classificadores. Na literatura, existem duas formas tradicionais para isso: *Bagging* e *Boosting*. O *Bagging* constrói modelos de forma independente e faz a combinação dos mesmos para classificação de uma nova instância, enquanto *Boosting* utiliza todos os dados para treinar um classificador e de forma sequencial cria modelos cada vez mais precisos. Estudos apontam que *Bagging* apresenta melhores resultados na presença de ruídos de rótulos que o *Boosting* (DIETTERICH, 2000).

A criação de diversos classificadores no *Bagging* pode fazer com que ruídos de classes se espalhem de forma aleatória, causando impactos de diferentes proporções em cada classificador

Figura 4 – Processo de limpeza de conjunto de dados de treinamento com ruído em rótulo.



Fonte: (BRODLEY; FRIEDL, 1999)

criado. Portanto, cada classificador no *Bagging* pode ter uma quantidade diferente ruídos de rotulo ou até mesmo instâncias repetidas. Por outro lado, no *Boosting* as instâncias com ruídos no rótulo acabam sendo consideradas como de difícil classificação e a cada iteração recebem um peso maior no classificador para reduzir o erro de classificação. Ou seja, além de aumentar a complexidade, esse aumento de peso de exemplos que podem ter o rótulo incorreto pode gerar falsas generalizações nos classificadores.

Alguns estudos apontam que a perda de desempenho de classificadores utilizando bases com e sem ruídos de rótulos é quase imperceptível em alguns casos (MANWANI; SASTRY, 2013). Apesar disso, a maioria dos algoritmos recentes de aprendizagem de máquina não são completamente robustos para classificar corretamente na presença de ruídos em rótulos (FRÉNAY; VERLEYSEN, 2014). Ruído em rótulo é um problema que não pode ser controlado facilmente. Afirmar que algoritmos modernos são imunes a esse tipo de problema pode não ser verdade para muitos casos. A solução para amenizar os impactos desse problema talvez seja não ignorar as instâncias problemáticas, e sim removê-las da base de dados. A seguir serão apresentadas métodos que visam realizar a detecção e eliminação de ruídos de rótulo.

3.2.2 Métodos de Limpeza

Métodos de limpeza são soluções simples e vastamente utilizadas para lidar com ruídos de rótulo que identificam instâncias com ruídos em rótulos nos conjuntos de dados para removê-las ou consertá-las. A Figura 4 apresenta como funciona o processo tradicional de limpeza de ruídos de rótulos. A partir de um conjunto de dados de treinamento que possui instâncias com ruído de rótulos, um filtro de limpeza é aplicado para criação de um novo conjunto de dados sem ruídos que pode vir a ser usado por algoritmos de aprendizagem de máquina. Diferentes estratégias podem ser utilizadas para realizar a limpeza de um conjunto de dados:

- Filtragem Baseada em previsões de modelo: utiliza previsões de classificadores para determinar se uma instância possui ruído de rótulo. Basicamente, se um classificador

estimar um rótulo diferente do atual de uma determinada instância implica que a mesma pode conter ruído de rótulo (MIRANDA et al., 2009);

- Baseada na influência do modelo: parte-se do pressuposto de que instâncias mal rotuladas causam grandes impactos na aprendizagem. Algumas estratégias aqui removem exemplos de conjuntos de dados de treinamento um a um e avaliam o desempenho dos classificadores criados aumentam ou diminuem (MALOSSINI; BLANZIERI; NG, 2006). Aumento de desempenho pode indicar que o exemplo removido possuía ruído de rótulo. Em contrapartida, diminuição de desempenho pode indicar que o exemplo removido está rotulado corretamente e deve ser adicionado novamente;

- Modelos baseados em vizinhança: esse grupo utiliza as instâncias vizinhas de uma dada instância para predizer seu rótulo. Falhas nessa predição podem indicar ruídos de rótulo na vizinhança ou até mesmo na própria instância que teve seu rótulo predito (DELANY; CUNNINGHAM, 2004). Ou seja, classificadores baseados em vizinhança são sensíveis a ruídos em rótulos e portanto falhas na classificação podem identificar ruídos de rótulo (SÁNCHEZ; PLA; FERRI, 1997);

- Métodos baseados em conjuntos de classificadores: para esse grupo de estratégias, um grupo de classificadores pode ter um desempenho melhor na detecção de ruídos que apenas um classificador. Existem duas maneiras de definir a localização de ruído de rótulo usando conjuntos de classificadores: (1) com base em votação, em que um exemplo pode ser considerado ruidoso quando diferentes classificadores fazem predições de rótulo diferentes; ou (2) baseado em pesos e limiares, quando é utilizado um filtro baseado em classificadores construídos a partir de re-amostragens do conjunto de dados de treinamento, para remover instâncias incorretamente rotuladas múltiplas vezes (MORALES et al., 2017);

- Complexidade de dados: para este grupo de estratégias, uma instância que aumenta a complexidade do conjunto de dados de treinamento pode ser um ruído (GAMBERGER; LAVRAC; GROSELJ, 1999).

Cada tipo de filtro concentra-se em aspectos bem específicos que podem levar a detecção de ruídos, fazendo com que diferentes tipos de ruídos sejam detectados em conjuntos de dados. Na PDS, como o número de estados alcançáveis dentro de um software aumenta

exponencialmente a complexidade do código (HOLZMANN, 2002), a alta complexidade pode aumentar a probabilidade de encontrar instâncias de difícil manutenção e que provavelmente podem apresentar defeitos.

3.2.3 Métodos tolerantes a ruídos de classe

Até agora, foram apresentadas modelos robustos que ignoram ruídos de rótulos com a premissa de que algoritmos de aprendizagem não robustos o suficientes para aprender mesmo com ruídos. Além dos modelos robustos, métodos de limpeza que buscam filtrar instâncias com ruídos de rótulo nos conjuntos de dados utilizados para treinamento de classificadores também foram discutidas.

Além dessas estratégias já discutidas ainda existem abordagens que modificam o algoritmo de aprendizagem para reduzir a influência dos ruídos de rótulos. Ou seja, em vez de executar técnicas de filtragem antes de treinar o classificador, métodos de filtragem também podem ser incluídos no algoritmo de aprendizagem, o que pode tornar esses modelos tolerantes a ruídos de classe. Para tal, pode-se utilizar basicamente dois tipos de métodos (FRÉNAY; VERLEYSEN, 2014):

- Probabilísticos: necessitam de alguma informação da probabilidade a priori para que se possa construir um classificador tolerante a ruídos de classe. Para isso, métodos Bayesianos podem ser utilizados (GABA; WINKLER, 1992) (SWARTZ et al., 2004);
- Baseados em modelos: aqui estão estratégias construídas a partir de algoritmos tradicionais de classificação para serem tolerantes a ruído de rótulo. Foram criados ao longo do tempo variações de algoritmos como de SVM (FRÉNAY; VERLEYSEN, 2013), Redes Neurais (KHARDON; WACHMAN, 2007) e até mesmo Árvores de Decisão (CLARK; NIBLETT, 1989).

3.3 RUÍDO DE RÓTULO NA PREDIÇÃO DE DEFEITOS EM SOFTWARE

Estudos anteriores revelaram que bases de dados utilizadas para treinamento de sistemas de PDS podem ter ruído de rótulo (BIRD et al., 2009) (HERZIG; JUST; ZELLER, 2013) (KIM et al., 2011). O processo de rotulação de dados que possui uma forte dependência de recursos humanos como testadores e desenvolvedores de software, relatórios de defeitos, relatórios de

lançamento de novas versões de software e alterações de código (KIM et al., 2011) (COSTA et al., 2016) (NETO; COSTA; KULESZA, 2018). Além disso, como foi visto na Seção 2.2.1, o processo de rotulação em bases de dados de PDS precisa que um defeito seja reportado em um SRP. Ou seja, instâncias defeituosas representam apenas problemas já relatados por testadores ou usuários finais.

Desta forma, o SRP pode introduzir alarmes falsos quando usuários confundem novos requisitos com defeitos, mas geralmente isso é detectado por desenvolvedores e gestores de projetos em um processo de triagem antes de qualquer alteração de código (HERZIG; JUST; ZELLER, 2013). Por outro lado, instâncias rotuladas como sem defeitos podem na verdade conter defeitos não reportados ainda (NETO; COSTA; KULESZA, 2018). Ou seja, defeitos podem permanecer em um software mesmo após o lançamento de novas versões de software até sejam reportados e consertados em versões seguintes (CHEN et al., 2014) (VANDEHEI; COSTA; FALESSI, 2021). Desta forma, existe uma probabilidade maior de existir ruídos de rótulo na classe sem defeito, apontando que trata-se de um Ruído Aleatório como foi visto na Seção 3.1.

Estratégias de limpeza de ruído de rótulo baseados em vizinhança foram bastante exploradas em bases de dados para PDS. Uma das primeiras propostas de detecção de ruídos de rótulo que serviu de base para diversas outras que utilizam vizinhança foi o *Edited Nearest Neighbors* (ENN) (WILSON, 1972). Segundo o ENN, uma instância terá ruído de rótulo se as classes das instâncias vizinhas forem majoritariamente diferentes da instância em questão. Algumas variações de ENN foram criadas e utilizadas na detecção de ruídos de rótulo em PDS (KHAN et al., 2020) (GARCIA et al., 2019), entre elas pode-se destacar o *All-k-ENN* (AENN) (TOMEK, 1976). No AENN o ENN é aplicado para todo o inteiro entre 1 e k vizinhos mais próximos para cada instância. A cada iteração, os exemplos que têm a maioria de seus vizinhos de outras classes são marcados como barulhentos para serem removidos.

Kim et al. (2011) criou o *Closest List Noise Identification* (CLNI), uma estratégia que utiliza a vizinhança de uma dada instância para determinar se ela possui ruído de rótulo semelhante ao ENN. Entretanto, em vez de utilizar o voto majoritário, o CLNI determina que se a percentagem de instâncias vizinhas com rótulo diferente de uma instância em questão for maior que um limiar pré-definido essa instância deve ter ruído de rótulo. Ou seja, ao contrário do ENN, aqui é possível fazer ajustes de tolerância de ruídos que podem ser importantes dependendo da quantidade de ruídos presentes nessas bases de dados de PDS.

O ENN também é utilizado em conjunto com técnicas de balanceamento de dados. Técnicas como o SMOTE-ENN (BATISTA; PRATI; MONARD, 2004) também foram exploradas na detecção

de ruídos de rótulo em bases de dados de PDS (KHAN et al., 2020). O SMOTE é uma técnica tradicional utilizada redução do desbalanceamento de classes em conjuntos de dados através da sobre-amostragem de classes minoritárias (CHAWLA et al., 2002). Logo, em bases de dados ruidosas o SMOTE também tende a replicar instâncias com rótulos incorretos. Para minimizar os efeitos de ruídos replicados, o SMOTE-ENN adicionou o ENN após o balanceamento de classes para reduzir ruídos de rótulos.

Modelos baseados em conjunto de classificadores também foram explorados na detecção de ruídos de rótulo. Em geral, esse grupo de estratégias assume que uma instância mal classificada por um conjunto de classificadores deve ter sido rotulada incorretamente. Uma das primeiras técnicas utilizadas em bases de dados de PDS foi o *Iterative Partitioning Filter* (IPF) (KHOSHGOFTAAR; REBOURS, 2007). Essa técnica divide o conjunto de dados de treinamento em n subconjuntos para treinar n classificadores utilizando cada parte desse subconjunto. Todos classificadores são utilizados para predizer o rótulo de cada instância, o ruído de rótulo é caracterizado quando majoritariamente o rótulo predito seja diferente do rótulo atual.

Sabzevari, Martínez-Muñoz e Suárez (2018) criou uma técnica utiliza comitês de classificadores e validação cruzada que também foi explorada para detecção de ruído de rótulo na PDS (KHAN et al., 2020). Esta técnica separa o conjunto de treinamento em n partes para executar n rodadas de treinamento e teste, em cada rodada um conjunto de dados é utilizado para teste e o restante para treinamento. Dependendo das características da base de dados em questão, um limiar de tolerância é criado para definir a quantidade tolerada de classificadores em desacordo com o rótulo atual de uma dada instância para determinar se existe ou não ruído de rótulo.

Ainda utilizando conjuntos de classificadores e validações cruzada, (SMITH; MARTINEZ; GIRAUD-CARRIER, 2014) criou uma técnica para detecção de ruídos considerando bases de dados desbalanceadas. Após alguns experimentos, foi demonstrado que em bases desbalanceadas a grande partes dos ruídos estão presentes na classe majoritária, caracterizando um Ruído Aleatório. Desta forma, apenas ruídos da classe majoritária deveriam considerados para remoção.

Técnicas baseadas em vizinhança podem também ser utilizadas em conjunto com técnicas baseadas em comitês de classificadores. Por exemplo, em Riaz, Arshad e Jiao (2018) inicialmente é aplicado um filtro baseado em vizinhança que considera o voto majoritário para detecção de ruídos de rótulo semelhante ao ENN e logo após é utilizado uma técnica de combinação de classificadores chamada *Adaboost* (FREUND; SCHAPIRE et al., 1996). Diferentemente

das técnicas que utilizaram validação cruzada, o *Adaboost* tenta aprender novos classificadores a partir de algumas iterações, concentrando-se em instâncias consideradas a priori como de difícil classificação/aprendizagem, ou seja, que foram classificadas incorretamente em iterações anteriores. Após algumas iterações, se uma instância for continuamente classificada incorretamente mesmo com o aumento de peso, ela pode ser marcada como uma instância com ruído de rótulo e não mais de difícil classificação.

3.4 CONSIDERAÇÕES FINAIS

Nesse capítulo, foram apresentados alguns tópicos relacionados com ruídos de rótulo. Apesar de ser um problema comum em bases de dados, é um problema que pode se manifestar de diferentes formas como foi visto na definição da taxonomia do ruído de rótulo. Devido as variações, diversas estratégias foram definidas para minimizar esse problema tendo em vista os impactos causados para modelos de aprendizagem de máquina que realizam classificações. Além disso, foi visto que esse é um problema que também atinge bases de dados utilizadas na PDS e as principais técnicas que podem ser utilizadas para detecção dos ruídos considerando os aspectos inerentes ao desenvolvimento e manutenção de um software, considerando vantagens e desvantagens de cada técnica apresentada.

4 FILTRAGEM ROBUSTA DE RUÍDO DE RÓTULO

Neste capítulo será apresentado o algoritmo de Filtragem Robusta de Ruído de Rótulo (F3R) criado levando em consideração os problemas observados em bases de dados utilizadas para PDS e os processos de garantia de qualidade de software.

O processo de produção de um software requer atenção desde os estágios iniciais de especificação de requisitos até a manutenção do sistema após o seu lançamento (SOMMERVILLE, 2011). Falhas em softwares são quase sempre inevitáveis e podem não ser descobertas durante os processos tradicionais de garantia de qualidade de software, como revisão de código e teste de software (BALOGUN et al., 2018). Defeitos produzem impactos negativos na confiabilidade e robustez dos produtos de software, portanto devem ser encontrados antes que se tornem problemas reais em ambientes de produção (BOWES; HALL; PETRIĆ, 2018).

No contexto acima, a predição de defeitos em software tem sido útil para identificar defeitos desde as primeiras fases do processo de desenvolvimento de um software. Por se tratar de um problema de aprendizagem de máquina, a qualidade do conjunto de dados utilizado para treinar os modelos de PDS é fator determinante para obtenção de bons resultados. Entretanto, trabalhos anteriores demonstraram que dados utilizados para treinamento de sistemas de PDS podem possuir ruídos de rótulo (KIM et al., 2011) (LIU et al., 2015) (RIAZ; ARSHAD; JIAO, 2018).

A solução proposta combina dois tipos de algoritmos para detecção de ruidos de rotulo: algoritmos baseados em vizinhança e algoritmos baseados em comitês. A escolha de cada uma dessas abordagens teve motivações diretamente relacionadas ao processo de revisão de código. Os algoritmos baseados em vizinhança são utilizados para simular o processo de revisão individual de cada desenvolvedor. Ou seja, algoritmos de vizinhança refletem a capacidade de um desenvolvedor realizar analogias entre os códigos desenvolvidos e revisados até o momento e os códigos que precisam de revisão a fim de encontrar defeitos. Em contrapartida, os algoritmos baseados em comitês representam o processo de revisão de código feito por diversos desenvolvedores que podem compartilhar opiniões e concluir que um artefato de software tem defeito ou não.

Inicialmente, em relação à escolha de algoritmos baseados em vizinhança, observa-se que existe prática comum no processo de desenvolvimento de software que pode aumentar ainda mais a dificuldade de encontrar e corrigir defeitos em software é a clonagem de código (MONDAL; ROY; SCHNEIDER, 2017) (ISLAM et al., 2019) (MONDAL et al., 2019). A replicação de

trechos de código semelhantes dentro de um mesmo software pode facilitar o desenvolvimento de funcionalidades comuns, mas exige atenção (DUBINSKY et al., 2013) (KRÜGER; BERGER, 2020) (LINSBAUER et al., 2021) (KRÜGER; MAHMOOD; BERGER, 2020). Apesar de acelerar o desenvolvimento de novas funcionalidades, essa prática também pode auxiliar na propagação de defeitos no software. Em se tratando de resolução de defeitos, uma vez que um trecho de código foi replicado em diversas partes do software, o desenvolvedor deve ter o cuidado de replicar também todas as correções no código para todas as partes que precisam do mesmo conserto.

Desta forma, a observação da similaridade entre artefatos de software pode facilitar a descoberta de defeitos ainda não reportados a partir dos defeitos já reportados (YOUM; AHN; LEE, 2017). Como foi visto na Seção 2.2.3, técnicas tradicionais de classificação que utilizam similaridade como o KNN podem ser utilizadas para aproximar artefatos de softwares semelhantes e ajudar na identificação de inconsistências de rótulo. Apesar de simples, métodos de aprendizado baseado em vizinhança como o KNN, também conhecidos como métodos locais, são bastante sensíveis a ruído. A falha na classificação de uma instância pode indicar uma falha no processo de rotulação da própria instância analisada ou até mesmo de seus vizinhos, ou seja, é encontrada uma inconsciência local.

Apesar de ser uma técnica antiga e bastante explorada, o KNN ainda apresenta resultados competitivos na detecção de ruídos de rótulo em trabalhos recentes (WANG; JHA; CHAUDHURI, 2018) (REEVE; KABÁN, 2019) (CANNINGS; FAN; SAMWORTH, 2020). Dado o contexto da revisão de código, o KNN pode detectar que existem artefatos com defeito a partir da similaridade com outros artefatos com defeitos já reportados. Ou seja, apenas um desenvolvedor com conhecimento em grande parte dos artefatos de um software conseguiria realizar esse tipo análise.

Além de ruídos de rótulo, artefatos de software podem apresentar características semelhantes e pertencerem a diferentes classes (CHEN et al., 2018), indicando uma sobreposição de instâncias. As instâncias em regiões de sobreposição de classes não são necessariamente ruídos, podendo ser apenas instâncias de difícil aprendizagem (GUPTA; GUPTA, 2018). Portanto, apenas uma abordagem local pode não ser suficiente para detecção de ruídos de rótulo em bases de dados de predição de defeito em software.

Em relação à escolha de algoritmos baseados em comitês, como foi visto na Seção 2.1, um dos processos de garantia de qualidade de software mais utilizados é o de revisão de código. Este processo geralmente utiliza comitês de desenvolvedores com diferentes níveis de

experiência proveniente dos códigos desenvolvidos e revisados anteriormente. Além disso, no processo moderno de revisão de código, cada desenvolvedor pode fazer a revisão de código através de comentários e ainda visualizar as revisões de outros desenvolvedores, podendo concordar ou até mesmo discordar com as revisões já feitas. Ou seja, um processo bastante semelhante ao da criação de comitês de classificadores com o *Adaboost* para detecção de ruídos em software.

Ao longo dos anos, comitês de classificadores também foram utilizados para detectar ruídos de rótulo em problemas de classificação, com destaque para o *Adaboost* (RIAZ; ARSHAD; JIAO, 2018) (KARMAKER; KWEK, 2006) (CAO; KWONG; WANG, 2012). O *Adaboost* aprende novos classificadores a partir de algumas iterações, concentrando-se em instâncias consideradas a priori como de difícil classificação, ou seja, que foram classificadas incorretamente em iterações anteriores (FREUND; SCHAPIRE et al., 1996). Através de um esquema de pesos, instâncias de difícil classificação são replicadas dentro do conjunto de dados fazendo com que classificadores tenham mais exemplos de cada instância problemática e pare de errar na classificação das mesmas. Ao final de algumas iterações, é esperado que o classificador minimize o erro de classificação e consiga aprender a classificar instâncias com o rótulo correto. Uma das formas de utilizar o *Adaboost* para detecção de ruídos de rótulos é através da definição de limiares de peso (KARMAKER; KWEK, 2006), assim se uma instância for continuamente classificada incorretamente mesmo com o aumento de peso, ela pode ser marcada como uma instância com ruído de rótulo e removida do conjunto de dados.

Desta forma, é proposto o algoritmo de F3R, uma nova abordagem para detecção de ruído de rótulo em conjuntos de dados utilizados para PDS que combina algoritmos baseados em vizinhança com algoritmos baseados em conjuntos de classificadores. Através dessa combinação, é criada uma estratégia robusta capaz de detectar ruído usando duas perspectivas diferentes. Algoritmos baseados em vizinhança representam a capacidade de cada desenvolvedor de fazer analogias entre o código em revisão com tudo que ele já desenvolveu e revisou no passado para identificação de possíveis falhas. Além disso, cada desenvolvedor pode ter atuado na codificação e revisão de partes diferentes do software, exigindo cada vez mais a troca de experiências e impressões sobre possíveis falhas dentro de um código que pode ser realizada a partir de comitês de desenvolvedores. Portanto, assim como no processo de revisão de software, o algoritmo de F3R realiza análises baseada em similaridade entre os artefatos de softwares feita a partir de comitês de desenvolvedores com níveis de experiência diferentes. A seguir, será apresentado em detalhes o algoritmo de F3R.

4.1 ALGORITMO

O algoritmo F3R recebe como entrada um conjunto de exemplos rotulados:

$$S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\} \tag{4.1}$$

onde cada exemplo $(\vec{x}, y)$ é composto por um vetor de atributos $\vec{x}$ e um rótulo y, no nosso caso defeito ou não defeito. Além disso, F3R possui alguns parâmetros: o limite de peso d, o número de iterações N e o tamanho da vizinhança K. Inicialmente será apresentado o filtro de ruído do rótulo KNNFilter que é executado a cada iteração do F3R. Em seguida, será apresentado o algoritmo F3R em detalhes.

4.1.1 Filtragem KNN (KNNFilter)

Essa função representa a capacidade de cada desenvolvedor de analisar a similaridade entre os códigos em revisão escritos por terceiros com tudo que o ele já desenvolveu ou revisou em um software. Ou seja, o principal objetivo desse filtro é identificar instâncias rotuladas como sem defeito que possuem outras instâncias similares rotuladas como com defeito. O algoritmo KNNFilter recebe como entrada as instâncias consideradas suspeitas de ter ruído de rótulo apontadas pelo Algoritmo de F3R e também previne a remoção de exemplos relevantes para a classificação de qualquer outra instância para evitar falsos positivos.

Portanto, a partir de um conjunto de dados S o algoritmo KNNFilter estima o rótulo de cada instância $\vec{x}_i$ com base nos rótulos das instâncias vizinhas. No algoritmo KNNFilter, $N(\vec{x}_i)$ é a lista de vizinhos da instância $\vec{x}_i$ e $KNN(\vec{x}_i, N(\vec{x}_i))$ é a classificação do rótulo da instância $\vec{x}_i$ através do algoritmo KNN utilizando a lista de vizinhos $N(\vec{x}_i)$. O algoritmo KNNFilter faz algumas verificações partir do resultado da classificação do rótulo da instância $\vec{x}_i$ para determinar os ruídos de rótulo:

- Predição incorreta: caso o rótulo predito da instância $\vec{x}_i$ pelo KNN seja diferente do rótulo atual, ou seja $KNN(\vec{x}_i, N(\vec{x}_i)) \neq y_i$, o algoritmo KNNFilter tenta definir quais instâncias vizinhas contribuíram para a falha na classificação. Para tal, o algoritmo adiciona no vetor m as instâncias em $N(\vec{x}_i)$ que possuem rótulo diferente do atual da instância $\vec{x}_i$ e que também estão no vetor de instâncias suspeitas s fornecido pela iteração atual do Algoritmo de F3R;

- Predição correta: caso o rótulo predito da instância $\vec{x}_i$ pelo KNN esteja correto o KNN-Filter verifica quais instâncias em $N(\vec{x}_i)$ são redundantes para a predição. Ou seja, novas rodadas de predição de rótulo são feitas com o mesmo algoritmo retirando uma a uma as instâncias com reposição para identificar quais instâncias são indispensáveis para a classificação correta. Caso a remoção de uma dada instância em $N(\vec{x}_i)$ resulte em um rótulo incorreto, essa instância será adicionada no vetor de instâncias não redundantes r.

Portanto, após executar o algoritmo KNNFilter em um conjunto de dados S serão consideradas instâncias com ruído de rótulo instâncias redundantes e que contribuíram para classificações incorretas, semelhante a proposta de Tomek (1976), caso também sejam consideradas suspeitas pelo algoritmo F3R.

Algoritmo 1: Algoritmo de KNNFilter

Dados: $S = (\vec{x}_1, y_1), ..., (\vec{x}_i, y_i)$, a quantidade de vizinhos K e a lista de instâncias suspeitas **s**;

for $i \leftarrow 1$ **to** $|S|$ **do**

Calcule $N(\vec{x}_i)$: os K vizinhos mais próximos de $\vec{x}_i$

/* Caso o rótulo predito esteja incorreto, identifique os vizinhos mais próximos que ajudaram no erro de classificação, ou seja, que possuem rótulo diferente do correto e que são suspeitos no Algoritmo 2 */

if $KNN(\vec{x}_i, N(\vec{x}_i)) \neq y_i$ **then**

foreach $(\vec{x}, y) \in N(\vec{x}_i)$ **do**

if $y \neq y_i$ *and* $\vec{x}_i \subset$ ***s*** **then**

add $(\vec{x}, y)$ in **m**;

end

end

end

/* Caso o rótulo predito esteja correto, identifique vizinhos que prejudicariam a classificação se removidos */

if $KNN(\vec{x}_i, N(\vec{x}_i)) = y_i$ **then**

foreach $(\vec{x}, y) \in N(\vec{x}_i)$ **do**

Calcule $\bar{N}(\vec{x}_i)$: conjunto de vizinhos de $\vec{x}_i$, excluindo $\vec{x}$

if $KNN\left(\vec{x}_i, \bar{N}(\vec{x}_i)\right) \neq y_i$ **then**

add $(\vec{x}, y)$ in **r**;

end

end

end

end

/* Conjunto de Ruídos E é composto pelas instâncias presentes em **m** mas ausentes em **r** */

$E = \mathbf{m} \setminus \mathbf{r}$

/* Remova todos os ruídos do conjunto de dados S */

$S = S \setminus E$

return S ;

4.1.2 Filtragem Robusta de Ruído de Rótulo (F3R)

O Algoritmo de F3R tem como objetivo simular o processo moderno de revisão de código visto na Seção 2.1.1 em que comitês de desenvolvedores realizam inspeções no código para apontar artefatos de softwares com defeito. A proposta do F3R é que sejam utilizados a cada iteração classificadores que representam desenvolvedores com diferentes níveis de experiência, ou seja, que já tenha participado do desenvolvimento e revisão de quantidades distintas de artefatos dentro do software. O objetivo do F3R é utilizar os princípios da revisão de código para analisar todo os artefatos de software disponíveis em busca de defeitos que ainda não foram reportados a partir de comitês de classificadores que representam os desenvolvedores.

A cada iteração do F3R novos classificadores são criados com foco em artefatos de software mal classificados nas últimas iterações. O erro de classificação ocorre quando um artefato de software possui o rótulo sem defeito mas foi classificado como com defeito, passando a ser considerados como suspeitos de ter ruído de rótulo. Assim como na revisão de código em que instâncias suspeitas recebem mais atenção nas próximas revisões, o F3R aumenta o peso de cada instância considerada suspeita por iterações anteriores para que os próximos classificadores tenham mais chances de acertar o rótulo. Após algumas iterações, caso esse peso supere um limiar previamente definido é definido que aquele artefato de software possui ruído de rótulo.

Ao mesmo tempo, em cada iteração do F3R é feita também uma verificação se os artefatos suspeitos são consideradas ruídos de rótulo para o algoritmo de KNNFilter apresentado na Seção 4.1.1. Na revisão tradicional de código, cada desenvolvedor pode considerar artefatos diferentes como suspeitos. De forma similar, KNNFilter em cada iteração vai receber uma lista diferente de artefatos suspeitos e vai ter conjuntos de dados diferentes para analisar devido a remoção em iterações anteriores de artefatos com ruídos de rótulo.

Em resumo, o Algoritmo F3R executará uma sequência de N iterações para produzir um conjunto de classificadores $h_1, ..., h_n$ a partir de um conjunto de dados $S = (\vec{x}_1, y_1), ..., (\vec{x}_m, y_m)$. Inicialmente para a primeira iteração, ou seja $n = 1$, todas as instâncias vão possuir peso 1 no vetor de pesos **w**. Em cada iteração N um classificador h_n que minimiza o erro de predição $\epsilon_n = Pr_{i \sim D_n} [h_n(\vec{x}_i) \neq y_i]$ é criado e o peso de cada instância é atualizado para as próximas iterações de acordo com o rótulo predito pelo classificador h_n para cada instância. Assim como em (KARMAKER; KWEK, 2006), instâncias que extrapolam o limite d de peso predefinido são consideradas com ruidosas e removidas do conjunto de dados. Além disso, o Algoritmo F3R

mantém uma lista **s** composta por instâncias suspeitas, ou seja, que ainda não extrapolaram o peso mas foram classificadas incorretamente pelo classificador h_n. Instâncias no conjunto **s** são utilizadas como entrada para o Algoritmo KNNFilter que vai avaliar se essas instâncias são inconsistências localmente para que sejam removidas o mais cedo possível sem a necessidade de aumento de peso dentro do Algoritmo F3R.

A cada nova iteração, os pesos daqueles exemplos classificados incorretamente são aumentados enquanto os pesos daqueles corretamente classificados são diminuídos. Além do vetor de pesos **w**, também é mantido um conjunto de distribuição de pesos por instância $D_t(i)$ que determinará a distribuição ponderada das instâncias em cada iteração N. Portanto, Z_n utilizado no final de cada iteração é um fator de normalização para garantir que D_n seja uma distribuição de probabilidade.

Desta forma, a cada iteração, o classificador passa a se concentrar em exemplos difíceis de aprender. Para tal, com base em $D_n(i)$, cada instância é repetida no conjunto de dados de treinamento para criar o novo classificador h_n na próxima iteração. O processo Adaboost aumenta os pesos das instâncias classificadas incorretamente, o que pode ser uma instância difícil de aprender ou ruidosa, ou seja, uma instância suspeita. Incluindo o limite d, é criado um limite superior nos pesos das instâncias de ruído e é dado para o classificador um conjunto de dados com cada vez menos ruídos e instâncias suspeitas que já foram consideradas também inconsistentes e redundantes localmente pelo KNNFilter.

Algoritmo 2: Algoritmo de F3R

Dados: $S = (\vec{x}_1, y_1), ..., (\vec{x}_m, y_m)$, limiar d, número de iterações N e número de vizinhos K;

Inicialize $D_1(i) = 1/m$, $w_1(i) = 1$;

for $n \leftarrow 1$ **to** N **do**

Treine o classificador h_t usando como base o conjunto de dados S e a distribuição D_t ;

Calcule $\epsilon_n = Pr_{i \sim D_n} [h_n(\vec{x}_i) \neq y_i]$;

foreach $(\vec{x}_i, y_i) \in S$ **do**

if $h_n(\vec{x}_i) \neq y_i$ **then**

$w_{n+1}(i) = w_n(i)\sqrt{\frac{1-\epsilon_n}{\epsilon_n}}$;

$p = -1$;

else

$w_{n+1}(i) = w_n(i)\sqrt{\frac{\epsilon_n}{1-\epsilon_n}}$;

$p = 1$;

end

if $w_{n+1}(i) \geq d$ **then**

Remova $(\vec{x}_i, y_i)$ de S;

end

if $w_{n+1}(i) \geq 1$ **then**

Adicione $\vec{x}$ in **s**;

end

$D_{n+1}(i) = D_n(i)(w_{n+1}(i))^p$;

end

$S = KNNFilter(S, K, \mathbf{s})$;

for $i \leftarrow 1$ **to** $|S|$ **do**

$D_{n+1}(i) = \frac{D_{n+1}(i)}{Z_n}$;

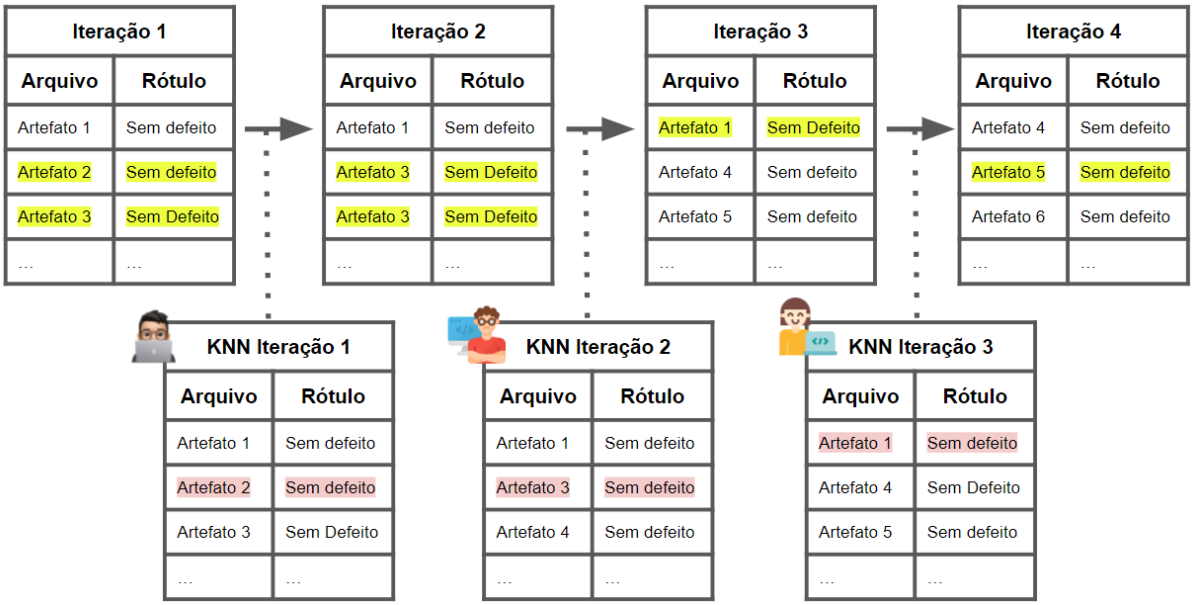
end

end

return S;

A Figura 5 ilustra todo o funcionamento do algoritmo F3R. A cada iteração, instâncias destacadas em amarelo são consideradas suspeitas, ou seja, instâncias apontadas como Com

Figura 5 – Ilustração do F3R



Fonte: Elaborada pelo autor (2022)

defeito quando possuem o rótulo Sem defeito. Após cada iteração, os artefatos de software passam por mais uma avaliação utilizando o KNNFilter que representa a capacidade de cada desenvolvedor de realizar analogias entre os artefatos já revisados e desenvolvidos e os artefatos sob revisão. Portanto, em cada iteração KNN, artefatos considerados suspeitos por iterações anteriores podem vir também a ser considerados defeitos, configurando assim um ruído de rótulo que faz com que essa artefato de software não venha mais ser considerado nas próximas iterações. Desta forma, a cada nova iteração, novos artefatos podem vir a ser considerados como suspeitos e novas vizinhanças são criadas como consequência das iterações anteriores. As novas vizinhanças representam novos desenvolvedores que podem ter revisado e desenvolvido artefatos diferentes de software e consequentemente destacar novos artefatos como ruidosos em conjuntos de dados cada vez mais limpos.

4.2 CONSIDERAÇÕES FINAIS

Foi apresentado nessa Seção o algoritmo de F3R. Trata-se de uma abordagem de detecção de ruídos de rótulos em bases de dados de PDS que simula o processo de garantia de qualidade de software de revisão de código. Tendo em vista a dinâmica do processo de revisão de código, foi criada uma estratégia capaz de simular diversos desenvolvedores com diferentes níveis de experiência dentro do software trabalhando de forma colaborativa para detectar o maior

número de defeitos presentes no código de um software que ainda não foram reportados. A seguir, serão apresentados os resultados obtidos da aplicação desse algoritmo em bases de dados de PDS.

5 EXPERIMENTOS

Neste Capítulo serão apresentados alguns experimentos realizados para avaliar o algoritmo de F3R apresentado no Capítulo 4.

5.1 MÉTODOS AVALIADOS

Para avaliar o algoritmo de F3R, foram realizados experimentos com diferentes técnicas de filtragem de ruído de rótulo. Cada estratégia concorrente ao algoritmo de F3R e sua configuração serão descritas a seguir:

- *Boost*: está técnica representa o uso de comitês de classificadores para filtrar o ruído de rótulo. Mais especificamente, foi adotada uma filtragem de ruído de rótulo baseada no *AdaBoost* proposta por (KARMAKER; KWEK, 2006). Essa estratégia treina classificadores ao longo de algumas iterações aumentando o peso de instâncias mal rotuladas para diminuir o erro de classificação. Cada instância que extrapole um limiar de peso pré definido será considerada como ruidosa e removida do conjunto de dados. Foram utilizados como parâmetros nos experimentos 5, 10 e 15 iterações e 5 e 10 como limites de peso. Para execução dessa estratégia foi utilizada a função ORBoostFilter implementada na biblioteca disponibilizada por Morales et al. (2017);

- KNN: para avaliar a filtragem de ruído de rótulo baseado em vizinhança foi utilizada uma técnica baseada em KNN proposta por (TOMEK, 1976). Este filtro considera como ruído de rótulo instâncias que se enquadram em duas condições: (a) se a instância participa de uma classificação incorreta (ou seja, a instância está entre os vizinhos de uma instância classificada incorretamente) e (b) se a remoção da instância não produz uma classificação incorreta (ou seja, é uma instância redundante). Essa técnica foi avaliada utilizando 5 e 10 como quantidade de vizinhos por instância a serem analisadas. Para execução dessa estratégia foi utilizada a função BBNR implementada na biblioteca disponibilizada por Morales et al. (2017);

- SEQ: representa a execução sequencial de técnicas baseadas em vizinhança e técnicas baseadas em comitês de classificadores, semelhante ao adotado por Riaz, Arshad e Jiao (2018) para detecção de ruídos de rótulo. Desta forma, foram utilizadas as duas técnicas

aqui já citadas de KNN e Boost seguindo os mesmos parâmetros: 5 e 10 para quantidade de vizinhos no KNN e para o algoritmo de Boost 5, 10 e 15 para a quantidade de iterações e 5 e 10 para limites de peso. Ou seja, primeiro foi aplicado o filtro KNN e logo após no mesmo conjunto de dados foi aplicado o filtro Boost. Para execução dessa estratégia foi utilizada a função BBNR e logo após a função ORBoostFilter no mesmo conjunto de dados ambas implementada na biblioteca disponibilizada por Morales et al. (2017);

- F3R: representa o algoritmo apresentado no Capítulo 4. Os parâmetros de configuração são iguais aos das estratégias concorrentes: 5 e 10 vizinhos para a análise de vizinhança, 5, 10 e 15 para as iterações Boost e 5 e 10 como o limite de peso para remover instâncias ruidosas. Para manter o equidade entre as estratégias, foi utilizado como classificador no F3R o Naive Bayes que é o padrão implementado pela função ORBoostFilter implementada pela biblioteca (MORALES et al., 2017) utilizada pelas estratégias concorrentes aqui comparadas.

Na Seção 5.3.2, d representa o limite de peso, N o número de iterações Boost utilizados nas estratégias de filtragem que envolvem comitês de classificadores e K o número de vizinhos analisados por instância para técnicas baseadas em vizinhança.

5.2 MEDIDA DE DESEMPENHO

As medidas de desempenho apresentadas na Seção 5.3.2 avaliam o desempenho das técnicas de filtragem de ruído de rótulo na tarefa de detectar os ruídos de rótulo presentes nos conjuntos de dados. Para isso, foram utilizadas três medidas de desempenho:

$$\text{Precisão} = \frac{\text{número de casos de ruídos de rótulo identificados corretamente}}{\text{número de todos os casos de ruídos de rótulo identificados}} \tag{5.1}$$

$$\text{Cobertura} = \frac{\text{número de casos de ruídos de rótulo identificados corretamente}}{\text{número de todos os casos de ruídos de rótulo no conjunto de dados}} \tag{5.2}$$

$$\text{F1 score} = \frac{2 \times \text{Precisão} \times \text{Cobertura}}{\text{Precisão} + \text{Cobertura}} \tag{5.3}$$

Além disso, na Seção 5.3.2, estimamos a relevância estatística dos resultados do F3R em comparação com as estratégias concorrentes através do teste pareado de *Wilcoxon Signed-Rank* considerando 0,05 como nível de significância. Para tal, o teste foi dividido em dois cenários:

Cenário 1:

$$H_0 : F3R = Concorrente \tag{5.4}$$

$$H_1 : F3R \neq Concorrente \tag{5.5}$$

Cenário 2:

$$H_0 : F3R <= Concorrente \tag{5.6}$$

$$H_1 : F3R > Concorrente \tag{5.7}$$

Em primeiro lugar, para verificar se os resultados do F3R e do Concorrente são estatisticamente diferentes, avaliamos o Cenário 1. Com base no nível de significância de 0,05, se o *p-value* for superior a 0,05, podemos aceitar H_0 e assumir que a diferença entre os resultados do F3R e do concorrente não é estatisticamente relevante (NER). Caso contrário, se o *p-value* for menor que 0,05, podemos rejeitar H_0 para o Cenário 1 e assumir que os resultados de F3R e do concorrente são diferentes, portanto precisamos verificar o Cenário 2. No Cenário 2, se o *p-value* for menor que 0,05, podemos considerar H_1 e rejeitar H_0. Isso significa que os resultados F3R são estatisticamente superiores aos do oponente. Em resumo, para interpretar nossos resultados estatísticos:

- NER: com base no Cenário 1, a diferença entre Concorrente e F3R não é significativa estatisticamente. Ou seja, significa dizer que o Cenário 2 não foi avaliado.
- *p-value* $<$ 0,05: utilizando o Cenário 2, significa que o resultado do F3R é estatisticamente superior ao do Concorrente;
- *p-value* $>=$ 0,05: utilizando o Cenário 2, significa que o resultado do Concorrente são estatisticamente superiores ao do F3R;

A seguir serão apresentados dois estudos de de caso utilizando diferentes conjuntos de dados para avaliar o algoritmo de F3R.

Tabela 1 – Defect prediction Metrics

Métrica	**Descrição**
Size	Linhas de código (LOC)
LOC Touched	Soma das LOC adicionadas/excluídas/modificadas
NR	Número de alterações de código
Nfix	Número de defeitos corrigidos
Nauth	Número de autores
LOC Added	Soma das LOC adicionadas nas alterações de código
MAX LOC Added	Máximo de LOC adicionadas nas alterações de código
AVG LOC Added	Média de LOC adicionadas por alterações de código
Churn	Soma do LOC adicionado - excluído nas revisões
Max Churn	Número máximo de Churn nas Revisões
Average Churn	Média de Churn nas revisões
Change Set Size	Número de arquivos alterados por alteração de código
Max Change Set	Número de máximo arquivos alterados por alteração de código
Average Change Set	Número médio de arquivos alterados por alteração de código
Age	Idade da versão de software
Weighted Age	Idade da versão de software ponderada pelo LOC touched

Fonte: Elaborada pelo autor (2022)

5.3 ESTUDO DE CASO 1: RUÍDOS DE RÓTULO EM CONJUNTOS DE DADOS PARA PDS

5.3.1 Conjunto de Dados

Para avaliar nossos resultados, foram utilizados 10 projetos de código aberto coletados por Falessi, Ahluwalia e Penta (2021) do ecossistema Apache. Para rotulação dos artefatos de software foi utilizado o JIRA (YATISH et al., 2019) como SRP e o Git (LOELIGER; MCCULLOUGH, 2012) como SCV. Para extrair características dos artefatos de software, o conjunto de dados usa 16 métricas definidas por D'Ambros, Lanza e Robbes (2012) e Falessi, Russo e Mullen (2017) que podem ser encontradas na Tabela 1. As métricas caracterizam o processo de desenvolvimento e o próprio produto de software e têm sido bastante exploradas por outros trabalhos que envolvem PDS (D'AMBROS; LANZA; ROBBES, 2012) (HE et al., 2015) (VASHISHT; RIZVI, 2021).

Como pode ser visto na Tabela 1, grande parte das métricas utilizadas estão relacionadas com alterações de código. Entre elas estão métricas que extraem de cada artefato de software

a quantidade de linhas de código alteradas, quantidade de alterações de código historicamente realizadas e até número de defeitos já reportados. Além de métricas retiradas diretamente do artefato de software, existem métricas que analisam ainda a quantidade de artefatos de softwares alterados por alteração de código. Como foi visto na Seção 2.2.1, alguns defeitos não são descobertos logo após serem adicionados em uma versão de software. Portanto, métricas que analisam aspectos relacionados as alterações de código podem auxiliar na detecção de ruídos de rótulo no momento de sua inserção.

Para rotular os artefatos de software como Com Defeito ou Sem Defeito, Falessi, Ahluwalia e Penta (2021) utilizou a abordagem realista. Como foi visto na Seção 2.2.1, a abordagem realista é a forma mais simples de realizar o processo de rotulação por utilizar apenas a versão afetada reportada no SRP para rotular artefatos de software. Além da abordagem realista, na Seção 2.2.1 foi abordada também a técnica SZZ que consegue identificar defeitos latentes a partir de alterações de código utilizadas para remoção de defeitos. Ou seja, o SZZ consegue definir quando o defeito foi de fato inserido para que seja possível não apenas rotular artefatos de software na versão afetada, mas também rotular os artefatos em versões anteriores que já possuíam o defeito mesmo que não tenha sido reportado no SRP. Desta forma, o SZZ foi utilizado para identificar nas versões anteriores quais artefatos de software estão rotulados como sem defeito quando na realidade possuem defeito, configurando assim o ruído de rótulo nesse trabalho.

Como foi visto na Seção 2.2.1, o SZZ é útil para revelar ruídos de rótulo em versões antigas de um software através da identificação de fato de todas as versões que já possuíam o defeito reportado mesmo que essas versões não sejam apontadas como afetadas por usuários no SRP. Em Falessi, Ahluwalia e Penta (2021) foi utilizada uma versão intermediária dentro do ciclo de vida do software, portanto uma versão que tivesse versões anteriores e mais novas em proporções semelhantes. A rotulação realista rotulou na versão intermediária os artefatos de software utilizados para resolver defeitos reportados no SRP que apontaram a versão intermediaria como afetada. Já para identificação de ruídos de rótulo, foram identificados os defeitos reportados após o lançamento da versão intermediária que já estavam presentes na mesma através da abordagem SZZ.

A Tabela 2 resume a distribuição de artefatos de software Sem Defeito, Com Defeito e Ruídos de rótulos por projeto obtida a partir do processo de rotulação. Na Coluna Ruídos está a quantidade de artefatos rotuladas como Sem Defeito e que na realidade são artefatos Com Defeito. Desta forma, é possível observar que para todos os projetos relatados aqui, há

Tabela 2 – Distribuição de ruído de rótulo por projeto

Projeto	Sem Defeito	Com Defeito	Ruídos	% Ruídos
AVRO	1111	16	50	4.44%
GIRAPH	2814	96	538	18.49%
IVY	4557	139	511	10.88%
OPENJPA	10518	196	778	7.26%
PROTON	6729	130	221	3.22%
SSHD	1050	47	262	23.88%
STORM	6429	367	1645	24.21%
WHIRR	661	19	37	5.44%
ZEPPELIN	2467	193	586	22.03%
ZOOKEEPER	1815	51	345	18.49%

Fonte: Elaborada pelo autor (2022)

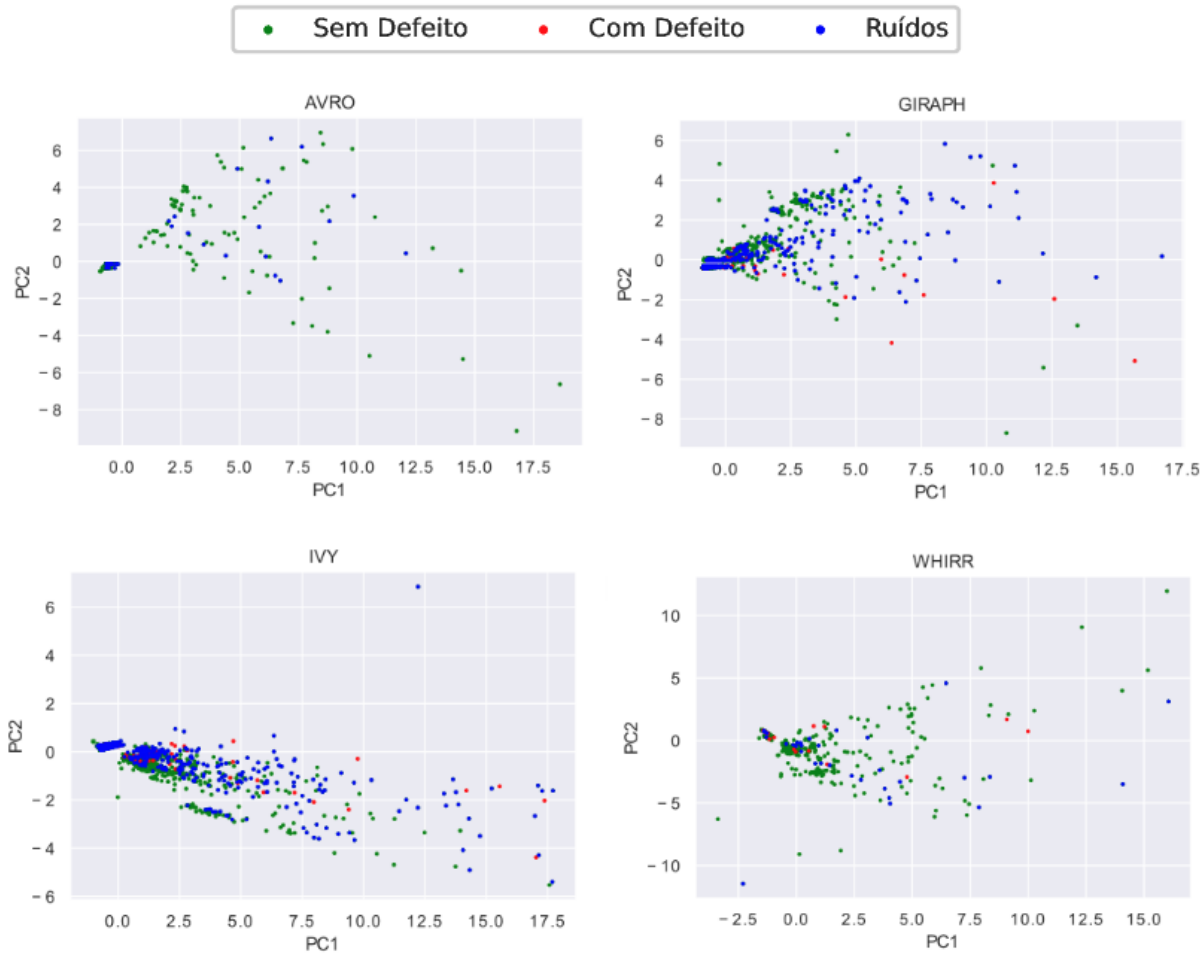
mais ruídos de rótulo do que defeito, o que pode indicar que muitos defeitos de fato estão presentes em algumas versões de softwares e são apenas encontrados algumas versões depois de sua inserção. Outra característica marcante em conjuntos de dados de PDS já relatada na Seção 2.2.4 e que também pode ser observado na Tabela 2 é o desbalanceamento de classes, a quantidade de artefatos Sem Defeito é muito superior que a dos Com Defeito.

Para ilustrar alguns dos desafios apontados na Seção 2.2.4, é esboçado na Figura 6 a distribuição de dados de quatro projetos feita a partir do uso de duas dimensões PC1 e PC2 extraídas com auxílio da técnica de *Principal Component Analysis* (PCA) e remoção de *outliers* para melhor visualização. Analisando a distribuição dos quatro projetos é possível notar que artefatos Com Defeitos e Ruídos não se concentram em regiões bem delimitadas no conjunto de dados, na realidade estão presentes em todas as regiões dos gráficos. Além disso, existe uma forte proximidade entre artefatos de diferentes classes, indicando que pode existir uma sobreposição de classes. As distribuições de todas as bases de dados podem ser encontradas no Apêndice C.

5.3.2 Resultados

Nessa seção, serão apresentados os resultados da avaliação de desempenho do algoritmo de F3R apresentado na Seção 4 em comparação com os concorrentes apresentados na Seção 5.1. Para isso foram utilizadas algumas medidas de desempenho como precisão, cobertura e F1 score e também um teste estatístico para avaliar a relevância dos resultados conforme o que

Figura 6 – Distribuição dos dados

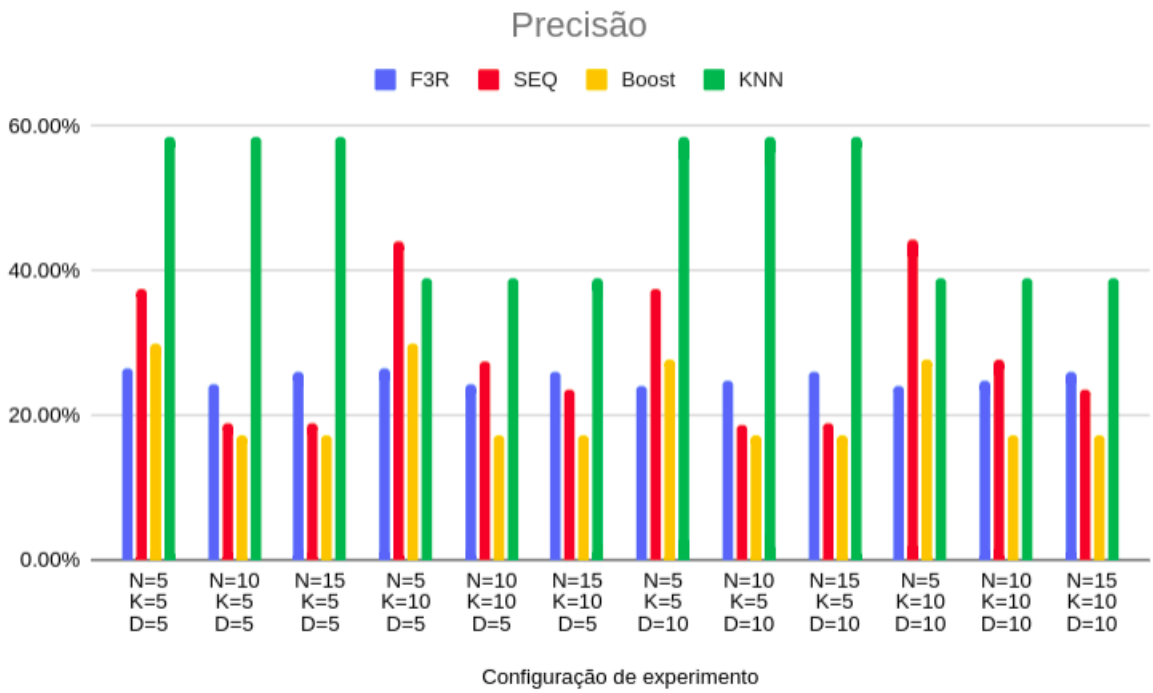


Fonte: Elaborada pelo autor (2022)

foi apresentado na Seção 5.2. Todos os resultados para todas as variações dos experimentos por projeto pode ser encontrados no Apêndice A.

A primeira medida de desempenho mensurada foi a precisão. Avaliar a precisão de uma abordagem de filtragem de rótulo de ruído pode ser valioso para reduzir o número de falsos positivos. No contexto da PDS, embora um dos grande objetivos seja remover a maior quantidade possível de ruídos de rótulo, remover várias instâncias sem ruído por engano pode diminuir a qualidade do preditor de defeitos. Pela Figura 7, a estratégia de KNN apresentou a melhor precisão para a maioria das variações do experimento. Ao contrário, as abordagens SEQ e Boost demonstraram uma degradação gradual de precisão ao longo das N iterações. Isso pode indicar que as abordagens que utilizam algoritmos baseados no *Adaboost* tendem a remover não apenas os ruídos de rótulo, mas também remover instâncias de difícil aprendizagem. Além disso, SEQ demonstrou um melhor desempenho em termos de precisão quando

Figura 7 – Resultados da avaliação de precisão



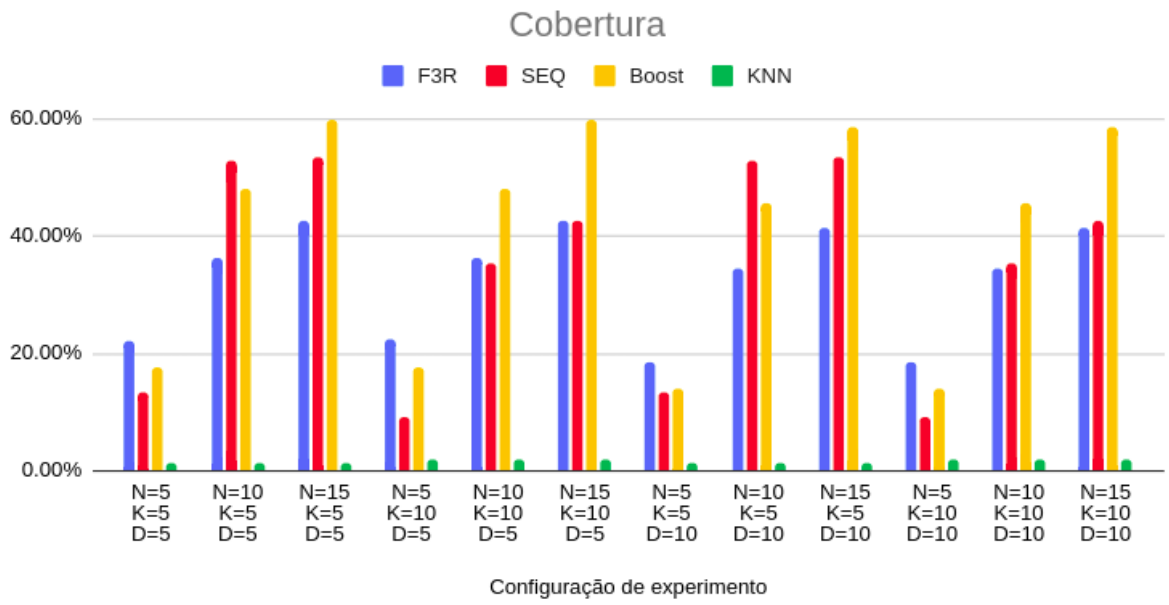
Fonte: Elaborada pelo autor (2022)

comparado com Boost, o que significa que o uso de KNN antes das execuções de Boost pode ser uma boa estratégia para reduzir inconsistências locais e fornecer um conjunto de dados mais limpo antes de passar por iterações de Boost.

Ainda referindo-se à Tabela 7, um fenômeno interessante pode ser observado nos resultados do algoritmo de F3R. Enquanto todas as estratégias que dependem de comitês de classificadores como SEQ e Boost obtiveram uma degradação de precisão ao longo de N iterações, o algoritmo de F3R demonstrou precisão estável e ao mesmo tempo bastante competitivo comparado com SEQ e Boost a partir de com 10 iterações.

Além da precisão, a cobertura pode ser útil para determinar quantos ruídos cada abordagem detectou em um conjunto de dados. Os resultados da cobertura podem ser encontrados na Figura 8 e mostram que, contrário dos resultados de precisão, todas as estratégias apresentaram um ganho de cobertura ao longo das iterações N, mas isso pode esconder algumas armadilhas. Para SEQ e Boost, cobertura e precisão são medidas inversamente proporcionais. Ao longo de cada iteração N, existe um aumento de cobertura e uma diminuição de precisão. Em conjuntos de dados fortemente desbalanceados como os utilizados para PDS a remoção de ruído do rótulo com uma baixa precisão pode diminuir a qualidade do conjunto de dados

Figura 8 – Resultados da avaliação da cobertura



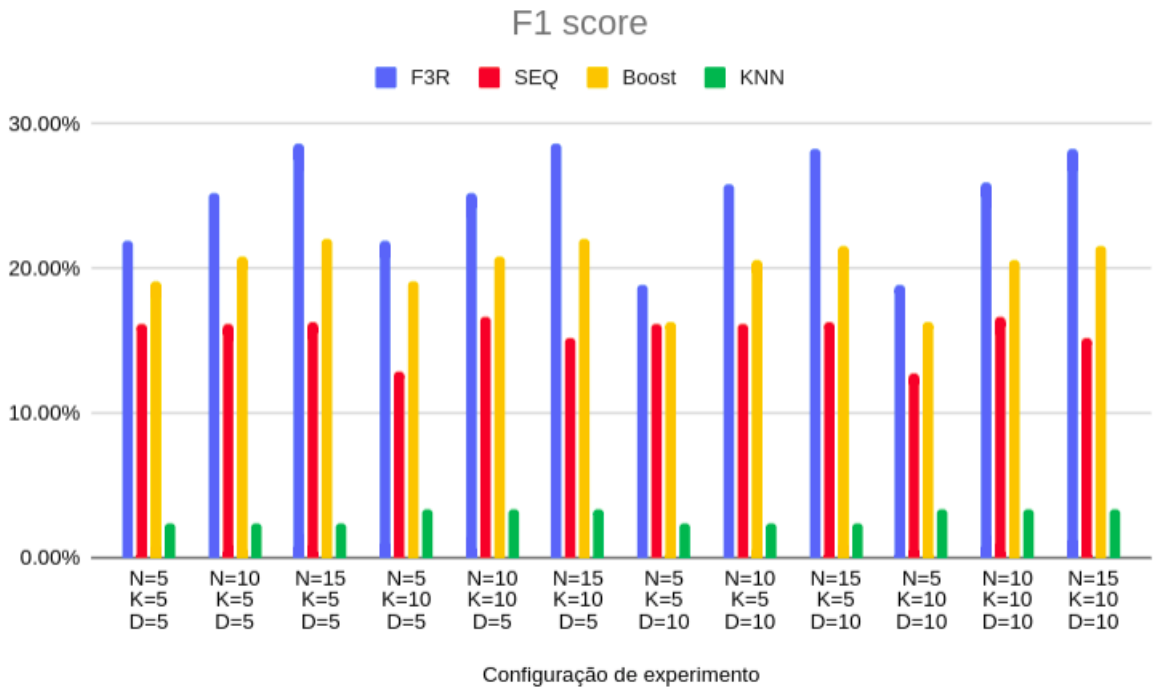
Fonte: Elaborada pelo autor (2022)

e, como consequência, criar um preditor de defeito com baixo desempenho. Ao contrário, o algoritmo de F3R apresentou uma precisão estável e também um ganho de cobertura ao longo das N interações. Isso indica que o algoritmo de F3R foi capaz de detectar mais ruídos sem danos de precisão.

Outro resultado interessante na Figura 8 é o desempenho do KNN. Apesar de ter apresentado maior precisão na Figura 7, KNN apresentou uma cobertura extremamente baixa em comparação com as outras estratégias detecção de ruído de rótulo. Aém disso, a melhoria da precisão na abordagem SEQ ao adicionar KNN ao Boost não refletiu também no aumento de cobertura. Isso indica que uma abordagem como SEQ em que é feita uma combinação simples de técnicas baseadas em vizinhança com técnicas baseadas em comitês de classificadores pode não ser suficiente para obter bons resultados.

Como foi apresentado na Seção 5.2, a partir da precisão e cobertura foi calculado o F1 score. Analisando os resultados de SEQ na Figura 9 é possível observar que a estratégia apresentou um F1 score estável, ou seja, mesmo com várias configurações nos experimentos e diferentes comportamentos em termos de precisão e cobertura, a cobertura progressivamente neutralizou a degradação de precisão. Além disso, Boost e F3R demonstraram um ganho de F-score ao longo de N iterações. No entanto, devido à degradação da precisão muito acentuada, o F1 score do Boost foi menor do que F3R em todas as configurações do experimento.

Figura 9 – Resultados do F1 score



Fonte: Elaborada pelo autor (2022)

Tabela 3 – Resultados estatísticos do F1 score

			F3R vs SEQ	F3R vs Boost	F3R vs KNN
D=5	K=5	N=5	**0.0488281**	**0.03125**	**0.00390625**
		N=10	NER	NER	0.00195313
		N=15	**0.0419922**	**0.0390625**	**0.000976563**
	K=10	N=5	**0.0078125**	**0.03125**	**0.00390625**
		N=10	0.0488281	NER	0.00195313
		N=15	**0.0419922**	**0.0390625**	**0.000976563**
D=10	K=5	N=5	NER	NER	0.00976563
		N=10	NER	0.03125	0.00195313
		N=15	0.0419922	NER	0.000976563
	K=10	N=5	0.0390625	NER	0.0078125
		N=10	**0.0488281**	**0.03125**	**0.00195313**
		N=15	0.0419922	NER	0.000976563

Fonte: Elaborada pelo autor (2022)

A fim de estimar a relevância estatística de nossos resultados, foi utilizado a medida de desempenho F1 score para comparar os resultados do algoritmo de F3R com o de todas as estratégias concorrentes. Foi realizado o teste estatístico através do calculo do *p-value* seguindo os cenários descritos na Seção 5.2. Todos os resultados do *p-value* de cada teste estatístico podem ser encontrados na Tabela 3. O testes estatísticos demonstraram que os resultados do algoritmo de F3R possui resultados estatisticamente superiores a SEQ, Boost e KNN em múltiplas configurações de experimentos destacadas em negrito na Tabela 3. Além disso, algumas comparações receberam o rótulo NER por não existir relevância estatística nos resultados da comparação entre o algoritmo de F3R com seu respectivo concorrente como foi visto na Seção 3. Vale salientar que, nenhuma estratégia obteve o *p-value* do teste estatístico superior a 0.05, o que indica que nenhum concorrente ao algoritmo de F3R apresentou um desempenho superior.

5.4 ESTUDO DE CASO 2: RUÍDOS DE RÓTULO EM CONJUNTOS DE DADOS DIVERSOS

O objetivo desse segundo estudo de caso é avaliar o algoritmo de F3R na detecção de ruídos de rótulo em outros conjuntos de dados com características semelhantes ao cenário da PDS. Parar tal, foi realizado alguns procedimentos de desbalanceamento e inserção de ruído de rótulo que serão detalhados a seguir.

5.4.1 Conjuntos de Dados

Para o estudo de caso 2 foram utilizados 10 conjuntos de dados retirados dos repositórios KEEL-dataset (ALCALÁ-FDEZ et al., 2011), UCI (DUA; GRAFF et al., 2017) e OpenML (VANSCHOREN et al., 2014). Devido ao uso de conjuntos de dados do mundo real, é preciso atentar aos ruídos de rótulo e desbalanceamento naturais de cada conjunto de dado. Para minimizar esses problemas e possibilitar a realização dos experimentos aqui descritos uniformizando a quantidade de ruídos e o índice de desbalanceamento, foram realizados alguns procedimentos bem semelhantes aos realizados por Moura, Prudencio e Cavalcanti (2018).

Tabela 4 – Algoritmos de aprendizagem para filtragem de ruídos de rótulo

Algoritmo	**Pacote R**
CN2 (rule learner)	RoughSets
kNN (nearest neighbor)	class
Naive Bayes	naivebayes
Random forest	randomForest
SVM (RBF Kernel)	e1071
J48	RWeka
JRip	RWeka
Multiplayer perceptron	RSNNS
Decision tree	party
SMO (linear Kernel)	RWeka

Fonte: Elaborada pelo autor (2022)

5.4.1.1 Limpeza

O processo de limpeza foi realizado inicialmente para remoção de ruídos já presentes em cada conjunto de dados. Para tal, assim como em Moura, Prudencio e Cavalcanti (2018), foram utilizados 10 classificadores de diferentes famílias apresentados na Tabela 4. Nesse conjunto de estão algoritmos de diversos tipos, tais como: árvores de decisão, modelos Bayesianos, redes neurais, máquinas de vetores de suporte, floresta aleatória, vizinhos mais próximos e métodos baseados em regras. Para limpeza foi utilizado o método *10-fold* e a votação por consenso. Ou seja, cada conjunto de dados foi dividido em 10 partes para realização de 10 rodadas de treinamento e teste.

Cada rodada utilizou 9 partes para treinamento e 1 para teste. Em cada uma das 10 rodadas de treinamento e teste, instâncias classificadas incorretamente pelos 10 classificadores foram consideradas ruídos e marcada para remoção a posteriormente. Os resultados do processo de limpeza podem ser visualizados na Tabela 5.

5.4.1.2 Desbalanceamento

O desbalanceamento foi realizado para uniformizar o nível de desbalanceamento entre os conjuntos de dados. Para tal, foi feito um procedimento de subamostragem aleatória da classe minoritária para obter a proporção de 80% de instâncias da classe majoritária e 20% de instâncias da classe minoritária. Desta forma, foram preservadas instâncias reais do conjunto

Tabela 5 – Processo de Limpeza de ruídos, onde I.D. representa o índice desbalanceamento de cada conjunto de dados, Att. a quantidade de atributos e % Rem refere-se a porcentagem de ruídos removida.

		Original		Após limpeza		
Conjunto de Dados	**Att.**	**I.D.**	**Total**	**I.D.**	**Total**	**% Rem**
breast-c-w	10	34:66	699	35:65	673	3,72%
column2C	7	32:68	310	32:68	308	0,65%
credit	16	44:56	690	45:55	644	6,67%
glass1	10	36:64	214	35:65	212	0,93%
heart-c	14	46:54	303	45:55	289	4,62%
hill-valley	101	50:50	1212	48:52	1184	2,31%
mushroom	23	48:52	8124	38:62	5644	30,53%
pima	9	35:65	768	32:68	732	4,69%
sonar	61	47:53	208	46:54	206	0,96%
tic-tac-toe	10	35:65	958	35:65	958	0.00%

Fonte: Elaborada pelo autor (2022)

Tabela 6 – Resultado do processo de desbalanceamento e inserção de ruídos para cada conjunto de dados utilizado.

	Desbalanceamento		Inserção de Ruídos			
Conjunto de Dados	**Classe 1**	**Classe 2**	**Classe 1**	**Classe 2**	**Ruído**	**% Ruído**
breast-c-w	435	109	462	82	27	4.96%
column2C	210	53	223	40	13	4.94%
credit	355	89	377	67	22	4.95%
glass1	138	35	147	26	9	5.20%
heart-c	160	40	170	30	10	5.00%
hill-valley	612	153	650	115	38	4.97%
mushroom	3488	872	3706	654	218	5.00%
pima	500	125	531	94	31	4.96%
sonar	111	28	118	21	7	5.04%
tic-tac-toe	626	157	665	118	39	4.98%

Fonte: Elaborada pelo autor (2022)

de dados já limpas pelo processo de remoção de ruídos, excluindo a necessidade de criação de instâncias sintéticas geralmente utilizadas em sobre-amostragens. A quantidade de instâncias de cada classe pode ser observada na Tabela 6.

5.4.1.3 Inserção de ruídos

A fim de inserir uma quantidade pequena de ruídos na classe majoritária semelhante ao estudo de caso 1, foram alterados aleatoriamente os rótulos de 5% das instâncias da classe minoritária para o rótulo da classe majoritária. A percentagem de ruídos e a nova quantidade de instâncias de cada classe é apresentada na Tabela 6.

Assim como no estudo de caso 1, foi esboçada a distribuição de dados dos conjuntos de dados utilizadas no estudo de caso 2 a partir do uso de duas dimensões PC1 e PC2 extraídas com auxílio da técnica de PCA e remoção de *outliers* para melhor visualização. De forma geral, como pode ser observado na Figura 10, existe uma distribuição mais controlada dos ruídos e uma separação mais bem definida entre classe 1 e classe 2 comparando com as bases de dados para PDS. Tais características auxiliam técnicas de detecção de ruídos, é possível observar que as instâncias em azul que representam os ruídos, ou seja, instâncias classificadas como classe 1 que na verdade pertencem a classe 2, estão localizadas em regiões predominantemente da classe 2. As distribuições de todas os conjuntos de dados podem ser encontradas no Apêndice D. A seguir serão apresentados os resultados dos experimentos realizados.

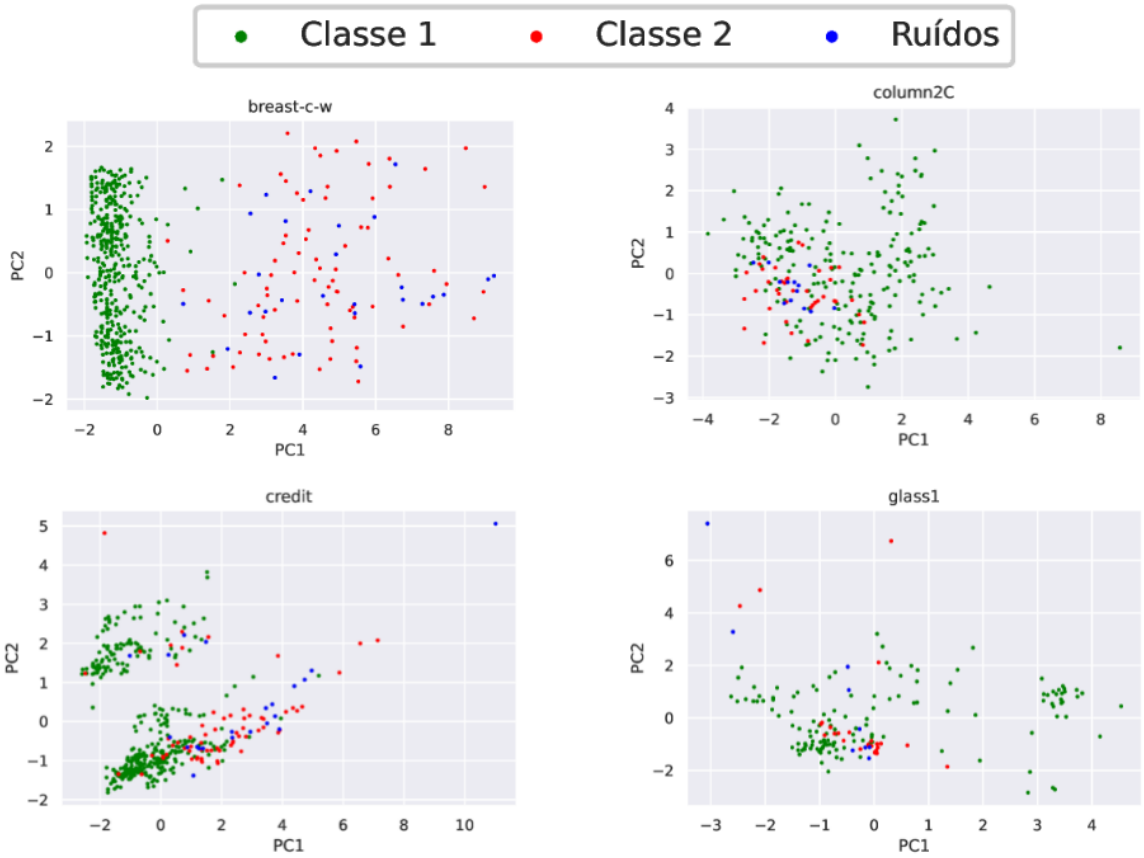
5.4.2 Resultados

Assim como no estudo de caso 1, foram realizados alguns experimentos para verificar o desempenho do algoritmo de F3R na detecção de ruídos no conjunto de dados. Todos os resultados para todas as variações dos experimentos por conjunto de dados pode ser encontrados no Apêndice B.

Assim como no estudo de caso 1, é possível observar na Figura 11 bons resultados em termos de precisão para o algoritmo de KNN. Ou seja, mais uma vez o KNN demonstra ser uma estratégia bastante precisa, principalmente com vizinhanças pequenas como em K=5. Além disso, a Figura 11 apresenta outro comportamento bastante similar ao estudo de caso 1 para todos os algoritmos avaliados. SEQ e Boost apresentam diminuição de precisão após algumas iterações de N nas variações de configuração do experimento. Esse fenômeno novamente não é também reproduzido pelo algoritmo de F3R que mantém uma precisão estável mesmo após algumas iterações.

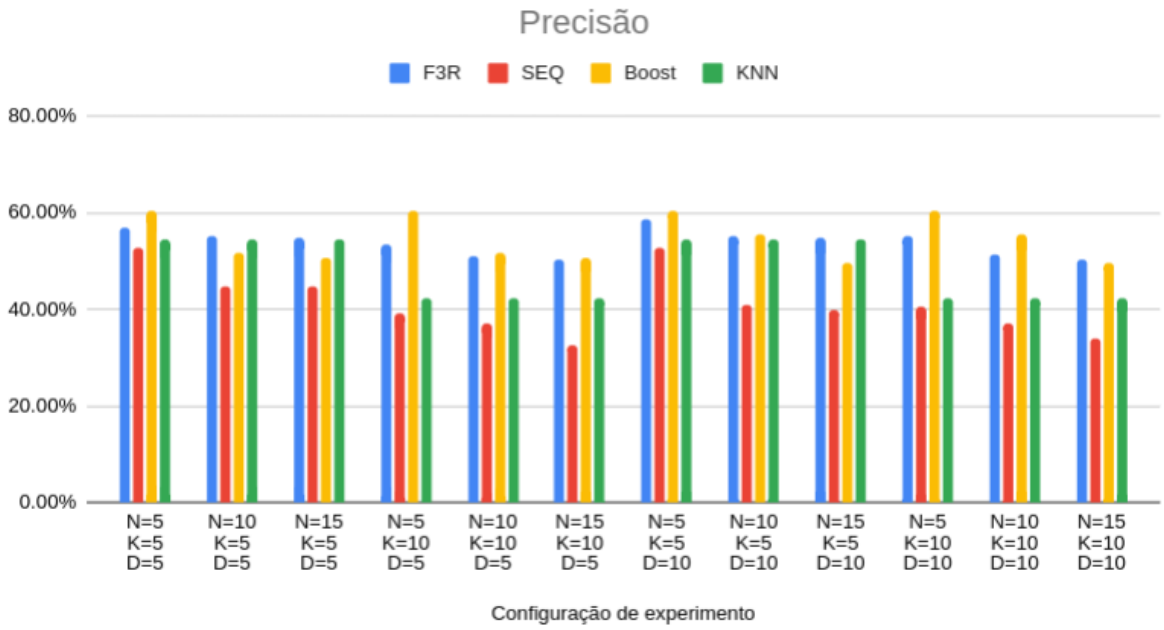
Ainda sobre a Figura 11, é possível observar que um dos maiores concorrentes do algoritmo de F3R é o KNN para k=5. Devido a inserção a inserção de ruídos sintéticos de forma

Figura 10 – Distribuição dos dados



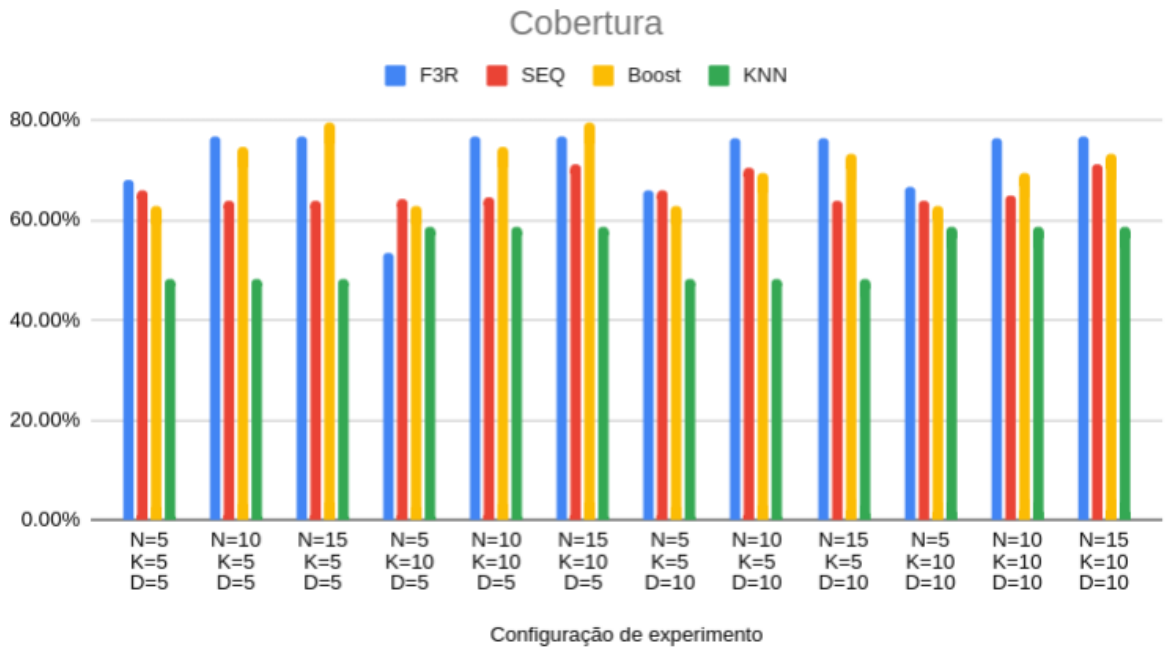
Fonte: Elaborada pelo autor (2022)

Figura 11 – Resultados da Precisão do estudo de caso 2



Fonte: Elaborada pelo autor (2022)

Figura 12 – Resultados da Cobertura do estudo de caso 2



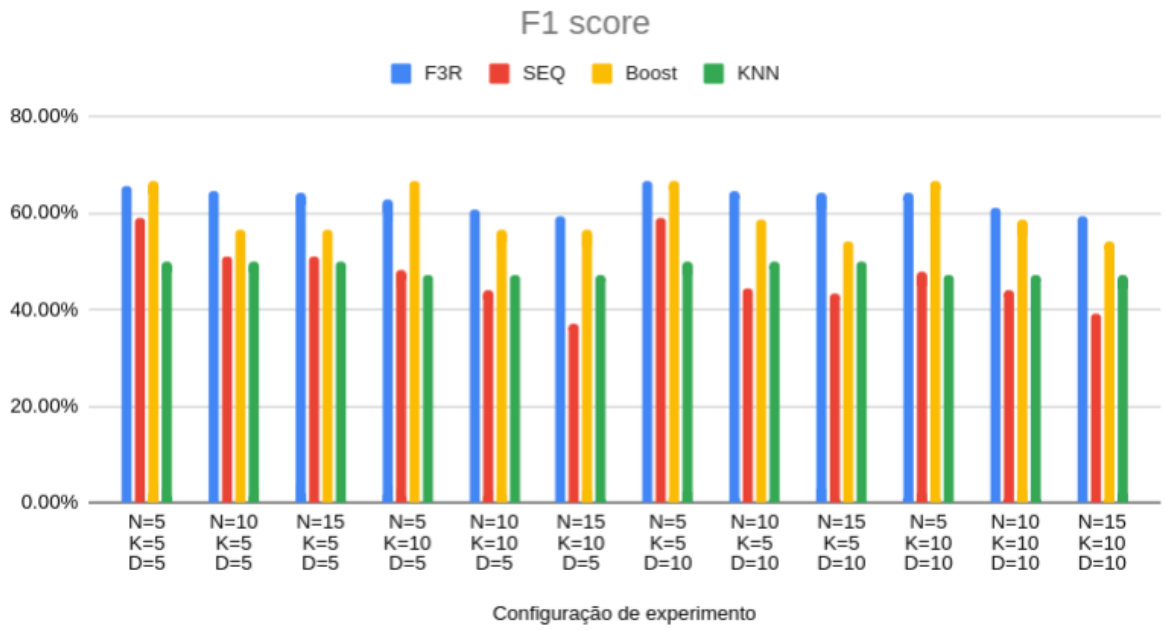
Fonte: Elaborada pelo autor (2022)

aleatória, a análise de vizinhanças, com destaque para vizinhanças curtas, aparenta ser uma boa estratégia para detecção de ruídos de rótulo. A inserção aleatória de ruídos tende a não criar sobreposições e regiões de difícil decisão para classificação. Vale a pena lembrar que o KNN no estudo de caso 1 com ruídos reais no problema de PDS obteve os piores resultados de F1 score.

Outro fato interessante sobre a Figura 11 é a sensibilidade do KNN a quantidade de vizinhos utilizada. Existe uma queda considerável de precisão quando são analisados K=10 vizinhos comparados com a precisão de K=5. Ou seja, a precisão é alta mas altamente dependente de parametrização. Além do KNN, o SEQ também teve sua precisão bastante reduzida com a mesma variação da quantidade de vizinhos analisadas. Em compensação, o algoritmo de F3R mesmo utilizando a análise de vizinhança e os mesmos parâmetros de KNN e SEQ apresentou estabilidade na precisão, apresentando resultados bastante competitivos ao Boost.

Assim como no estudo de caso 1, foi também estimada a cobertura de cada algoritmo na tarefa de detecção de ruídos. De forma semelhante, como pode ser observado na Figura 12, existe um ganho de cobertura para F3R, SEQ e Boost a cada iteração N na grande maioria dos experimentos. Os resultados de F3R e Boost são bastante competitivos, entretanto existe pouca diferença no algoritmo de F3R entre as iterações N=10 e N=15.

Figura 13 – Resultados do F1 score do estudo de caso 2



Fonte: Elaborada pelo autor (2022)

A última medida de desempenho analisada é o F1 score. Como é possível observar na Figura 13, o F1 score médio obtido pelo algoritmo de F3R apresenta um desempenho bem equilibrado mesmo após diversas alterações de configurações dos experimentos, com destaque para as variações de experimento que utilizaram N=10 e N=15 em que os resultados do algoritmo de F3R são sempre superiores aos concorrentes. Em contrapartida, o desempenho de Boost e SEQ apresentam uma diminuição do F1 score a cada iteração N. Esse fenômeno também foi observado no estudo de caso 1 nas bases de dados utilizadas para PDS.

A diminuição do F1 score de Boost e SEQ na Figura 13 é reflexo da queda de Precisão já apresentada na Figura 11. Ou seja, mesmo aumentando a cobertura após algumas iterações dos algoritmos inspirados no Boost, a grande maioria das estratégias não conseguem obter um equilíbrio entre precisão e cobertura. Por outro lado, o algoritmo de F3R apresentado o presente trabalho conseguiu equilibrar precisão e cobertura e mesmo assim ter coberturas competitivas ou até mesmo maiores utilizando N=10 e N=15 iterações.

Por fim, também foram realizados testes estatísticos comparando o F3R com os concorrentes. Como pode ser observado na Tabela 5.4.2, não houve diferença estatística entre os resultados de F1 score do Boost comparado com o algoritmo de F3R. Entretanto, é possível observar relevância estatística na diferença entre o algoritmo de F3R e o SEQ, um dos seus concorrentes, principalmente utilizando o limiar D=10.

Tabela 7 – Resultados dos testes estatísticos do F1 score do estudo de caso 2

			F3R vs SEQ	F3R vs Boost	F3R vs KNN
D=5	K=5	N=5	NER	NER	NER
		N=10	NER	NER	NER
		N=15	NER	NER	NER
	K=10	N=5	NER	NER	NER
		N=10	NER	NER	NER
		N=15	0.0244	NER	NER
D=10	K=5	N=5	NER	NER	0.0420
		N=10	0.0137	NER	NER
		N=15	0.0195	NER	NER
	K=10	N=5	0.0244	NER	NER
		N=10	NER	NER	NER
		N=15	0.0420	NER	NER

Fonte: Elaborada pelo autor (2022)

5.5 CONSIDERAÇÕES FINAIS

Foram apresentados nesse capítulo dois estudos de caso realizados para avaliar o desempenho do algoritmo de F3R em contraste com algumas estratégias concorrentes. O primeiro estudo de caso utilizou conjuntos de dados extraídos de softwares reais e com ruídos também reais sinalizados. No geral, todas as estratégias apresentaram ganho de cobertura ao longo das N iterações. Entretanto, houve perda de precisão em todas as técnicas baseadas no *Adaboost* concorrentes do F3R. Ao contrário, o F3R apresentou precisão estável mesmo após o aumento das iterações. Como consequência, o F3R apresentou F1 score médio superior para todas as variações de experimentos quando comparado com seus concorrentes com relevância estatística em diversas variações de experimento.

No segundo estudo de caso foram selecionadas alguns conjuntos de dados de repositórios populares para que através de simulações de desbalanceamento e da inserção de ruídos fosse possível avaliar o algoritmo de F3R em situações desfavoráveis mais próximas a realidade de conjuntos de dados para PDS. Para avaliar os resultados foram utilizados algumas medidas de desempenho como precisão, cobertura e F1 score e também um teste estatístico para comparar diretamente o desempenho do F1 score do algoritmo de F3R com seus concorrentes. Assim como no estudo de caso 1, o F3R apresentou F1 score na média superior aos concorrentes para a maioria dos experimentos e com destaque para N=10 e N=15 iterações.

6 CONCLUSÕES

O presente trabalho explorou as características de processos tradicionais de garantia de qualidade de software e características extraídas a partir do código-fonte de softwares para criar um algoritmo capaz de detectar ruídos de rótulo em conjuntos de dados utilizados para PDS. Mais especificamente, os ruídos aqui detectados nada mais são que defeitos ainda não detectados em softwares mas que estão presentes. A fim de explorar o potencial do algoritmo proposto, foram realizados dois estudos de caso. O estudo de caso 1 explorou conjuntos de dados de softwares com ruídos reais e o estudo de caso 2 utilizou conjuntos de dados de domínios não relacionados com PDS mas com características semelhantes de desbalanceamento e ruídos.

6.1 CONTRIBUIÇÕES

Fazendo uso de princípios básicos de processos tradicionais de garantia de qualidade de software, mais especificamente a revisão de código, o F3R demonstrou ser uma estratégia bastante promissora na tarefa de detecção de ruídos de rótulo. O algoritmo de F3R realiza inspeções em artefatos de softwares para detectar defeitos ainda não reportados simulando um comitê de desenvolvedores que interage entre si e possui capacidades de detecção de ruído a partir de diferentes perspectivas.

A experiência de cada desenvolvedor em diferentes partes do software, um aspecto bastante relevante dentro do processo de revisão de código, é simulado a partir da avaliação de vizinhanças de artefatos de software dentro de cada iteração do F3R. A análise de vizinhança simula as analogias feitas por desenvolvedores entre o código estático em análise e todos os outros códigos já desenvolvidos ou revisados anteriormente pelo desenvolvedor. A cada iteração, ruídos podem ser detectados e removidos, criando assim novas vizinhanças que serão avaliadas por novos comitês de classificadores mais especializados utilizando conjuntos de dados com cada vez menos ruídos.

Por outro lado, os algoritmos baseados em comitês representam bem o processo de revisão de código feito por diversos desenvolvedores com diferentes níveis de experiência que podem compartilhar opiniões e concluir que um artefato de software tem defeito ou não. Através do algoritmo de *Adaboost*, novos classificadores são treinados a cada iteração com foco em

instâncias consideradas a priori como de difícil classificação, ou seja, classificada incorretamente em iterações anteriores. Através de um esquema de pesos, instâncias de difícil classificação são replicadas dentro do conjunto de dados fazendo com que classificadores tenham mais exemplos de cada instância problemática e reduza a quantidade de erros de classificação. Em paralelo ao processo de revisão de código, cada desenvolvedor aqui é representado por um classificador que é treinado a cada iteração com diferentes amostragens do mesmo conjunto de dados de treinamento. Consequentemente, desenvolvedores com mais experiência têm mais chances de terem revisado ou desenvolvido artefatos de software similares aos de difícil classificação, podendo assim definir com mais clareza se é um ruído ou não.

No estudo de caso 1, foi observado que estratégias baseadas em vizinhança como o KNN possuem alta precisão e baixa cobertura. Ou seja, apesar de encontrar poucos ruídos, estratégias baseadas em vizinhança são confiáveis. Em paralelo com o processo de revisão de código, as chances de acerto na predição de defeitos pode aumentar se o desenvolvedor tem disponível facilmente alguns artefatos semelhantes aos sob revisão. Entretanto, a baixa cobertura reflete bem os problemas de sobreposição e desbalanceamento já apontados na Seção 2.2.4. Ou seja, as vizinhanças nem sempre são tão claras ao ponto de facilitar a detecção de ruídos de rótulo em conjuntos de dados de PDS.

No estudo de caso 1 também foi detectado uma considerável depreciação de precisão de técnicas baseadas em conjunto de classificadores como o SEQ e o Boost que utilizam como princípio básico o algoritmo de *Adaboost*. Em contraste, mesmo também utilizando o *Adaboost*, o F3R apresentou estabilidade na precisão e aumento de cobertura de ruídos encontrados a cada nova iteração. O algoritmo de F3R introduziu uma fase de análise de vizinhança que faz com que o ruído seja removido com segurança logo nas primeiras iterações do algoritmo, aproveitando a alta precisão de detecção de algoritmos de vizinhança. Portanto, essa antecipação favorece a detecção de ruídos mais complexos nas próximas iterações pelo fato de utilizar conjuntos de dados com cada vez menos ruídos. O algoritmo de F3R demonstrou ser uma estratégia robusta ao sobreajuste. Como foi visto na Seção 5.3.2, o Boost, que só utilizava o *Adaboost* e um limiar de pesos para remoção de ruídos, apresentou aumento de cobertura de ruídos com forte queda de precisão ao logo das iterações.

No estudo de caso 2 foram verificados fenômenos bem semelhantes ao estudo de caso 1. É possível observar ainda depreciação de precisão em técnicas que utilizam o *Adaboost* como o Boost e o SEQ e também precisão estável para o F3R. Entretanto, em comparação com o estudo de caso 1, o uso de ruídos sintéticos proporcionou um ganho de desempenho de todos

os algoritmos. De certa forma, os ruídos aleatórios sintéticos inseridos em uma um conjunto de dados já livre de ruídos e desbalanceado favoreceu a identificação de inconsistências de rótulo. Apesar disso, o algoritmo de F3R ainda apresentou resultados em média superiores aos concorrentes em múltiplas variações de experimento em termos de F1 score.

Um fato interessante no estudo de caso 2 em contraste ao estudo de caso 1 é o bom desempenho do KNN. O fato dos ruídos ficarem mais evidentes com poucas sobreposições faz com que estratégias baseadas em vizinhanças como KNN apresentem bons resultados. Além disso, a falta de complexidade na detecção dos ruídos também favorece o uso de técnicas como o Boost, mas mesmo assim foi observado uma queda de precisão após algumas execuções do algoritmo de *Adaboost* nos experimentos. Apesar de até mesmo visualmente os ruídos serem evidentes em muitos casos, ainda é grande a perda de precisão caso não exista uma regulação nas remoções feitas pelo Boost. Portanto, em termos de desempenho, levando em consideração a cobertura e precisão, Boost e KNN podem ser vistas como estratégias complementares, mas sua combinação pede conhecimento do conjunto de dados que será utilizado.

6.2 LIMITAÇÕES E TRABALHOS FUTUROS

Além das contribuições, é possível elencar algumas limitações e oportunidades de trabalhos futuros dessa tese. Com relação aos conjuntos de dados, cada estudo de caso analisou apenas 10 projetos diferentes, totalizando 20 conjuntos de dados estudados. Para validar com mais propriedade esta tese mais conjuntos de dados com diferentes distribuições podem ser requeridos. Além disso, no estudo de caso 1 apenas 16 métricas de software foram utilizadas. Apesar das métricas envolverem aspectos relacionados a revisão de código, processo de desenvolvimento e características de cada artefato de software, um estudo mais aprofundado em métricas de software talvez se faça necessário também.

Como foi mencionado na Seção 5.1, para representar o Boost, KNN e SEQ, foram utilizadas funções implementadas na biblioteca disponibilizada por Morales et al. (2017). Como consequência, o F3R utilizou o classificador Naive Bayes por ser o mesmo utilizado no Boost e SEQ. Portanto, fazer uso de mais classificadores pode ser também interessante para avaliar o desempenho do F3R. Além dos classificadores, foi utilizada uma quantidade limitada de parâmetros que poderia também ter sido largamente explorada para investigação dos impactos de mais que 15 execuções de algoritmos inspirados no *Adaboost* ou até mesmo mais variações de quantidade de vizinhos analisados.

Ainda sobre o *Adaboost*, é preciso salientar que não foram realizados experimentos que comparassem o desempenho do Boost com outras estratégias de comitês de classificadores como *Bagging*. O uso de *Bagging*, onde classificadores são treinados com partições diferentes do conjunto de dados, poderia representar um processo de revisão de código em que não existe compartilhamento de comentários e impressões entre os desenvolvedores. Ou seja, um comportamento não muito adotado em processos de revisão de código modernos mas que eram utilizadas no passado principalmente quando não existiam ferramentas para auxiliar o processo de revisão de código, como é o caso do *Email pass-around* apresentado na Seção 2.1.1.

Como foi visto no Capítulo 4, o algoritmo de F3R tenta reproduzir mecanismos básicos do processo de revisão de código utilizado para garantia de qualidade de software. Os detalhes desse processo levou as decisões de utilizar algoritmos baseados em vizinhança e em comitês de classificadores da maneira que foi implementado. Entretanto, o processo de revisão de código apenas analisa as alterações de código e não o código por completo. Desta forma, o foco no impacto da alteração de código no produto de software talvez seja mais compatível com o processo de revisão de código.

Por fim, a tese apresentada aqui considerou que existem apenas ruídos na classe sem defeito, ou seja, foram apontados como ruídos artefatos de software marcados como sem defeito que na verdade possuíam defeitos ainda não reportados. Entretanto, os defeitos rotulados nos conjuntos de dados podem representar também falhas em requisitos funcionais de um software, ou seja, não necessariamente uma falha inserida pelo desenvolvedor no processo de codificação do software. Uma falha de execução de um software que causa uma parada inesperada deve ser diferenciada de uma saída indesejada baseado em requisitos de domínio. Portanto, ruído na classe com defeito necessita de especialistas no domínio e em software para a devida rotulação como falha de codificação ou falha no entendimento do requisito que deveria ser implementado.

REFERÊNCIAS

ALCALÁ-FDEZ, J.; FERNÁNDEZ, A.; LUENGO, J.; DERRAC, J.; GARCÍA, S.; SÁNCHEZ, L.; HERRERA, F. Keel data-mining software tool: data set repository, integration of algorithms and experimental analysis framework. *Journal of Multiple-Valued Logic & Soft Computing*, Citeseer, v. 17, 2011.

ALSAWALQAH, H.; FARIS, H.; ALJARAH, I.; ALNEMER, L.; ALHINDAWI, N. Hybrid smote-ensemble approach for software defect prediction. In: SPRINGER. *Computer science on-line conference*. [S.l.], 2017. p. 355–366.

AMMANN, P.; OFFUTT, J. *Introduction to software testing*. [S.l.]: Cambridge University Press, 2016.

ARAR, Ö. F.; AYAN, K. Software defect prediction using cost-sensitive neural network. *Applied Soft Computing*, Elsevier, v. 33, p. 263–277, 2015.

AXELSSON, S.; BACA, D.; FELDT, R.; SIDLAUSKAS, D.; KACAN, D. Detecting defects with an interactive code review tool based on visualisation and machine learning. In: *the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*. [S.l.: s.n.], 2009.

BAHAROM, F. A survey on the current practices of software development process in malaysia. *Journal of Information and Communication Technology*, v. 4, p. 57–76, 2020.

BALOGUN, A.; BAJEH, A.; ORIE, V.; YUSUF-ASAJU, A. Software defect prediction using ensemble learning: An anp based evaluation method. *FUOYE Journal of Engineering and Technology*, v. 3, 09 2018.

BANSIYA, J.; DAVIS, C. G. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, IEEE, v. 28, n. 1, p. 4–17, 2002.

BASHIR, K.; LI, T.; YOHANNESE, C. W.; YAHAYA, M. Smotefris-inffc: Handling the challenge of borderline and noisy examples in imbalanced learning for software defect prediction. *Journal of Intelligent & Fuzzy Systems*, IOS Press, v. 38, n. 1, p. 917–933, 2020.

BATISTA, G. E.; PRATI, R. C.; MONARD, M. C. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD explorations newsletter*, ACM New York, NY, USA, v. 6, n. 1, p. 20–29, 2004.

BAUM, T.; LISKIN, O.; NIKLAS, K.; SCHNEIDER, K. Factors influencing code review processes in industry. In: *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*. [S.l.: s.n.], 2016. p. 85–96.

BEGEL, A.; NAGAPPAN, N. Pair programming: what's in it for me? In: *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. [S.l.: s.n.], 2008. p. 120–128.

BELLER, M.; BACCHELLI, A.; ZAIDMAN, A.; JUERGENS, E. Modern code reviews in open-source projects: Which problems do they fix? In: *Proceedings of the 11th working conference on mining software repositories*. [S.l.: s.n.], 2014. p. 202–211.

BENTLER, P. M.; YUAN, K.-H. Structural equation modeling with small samples: Test statistics. *Multivariate behavioral research*, Taylor & Francis, v. 34, n. 2, p. 181–197, 1999.

BIRD, C.; BACHMANN, A.; AUNE, E.; DUFFY, J.; BERNSTEIN, A.; FILKOV, V.; DEVANBU, P. Fair and balanced? bias in bug-fix datasets. In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2009. p. 121–130.

BISHOP, C. M. Pattern recognition. *Machine learning*, v. 128, n. 9, 2006.

BOWES, D.; HALL, T.; PETRIĆ, J. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*, Springer, v. 26, n. 2, p. 525–552, 2018.

BRODLEY, C. E.; FRIEDL, M. A. Identifying mislabeled training data. *Journal of artificial intelligence research*, v. 11, p. 131–167, 1999.

CANNINGS, T. I.; FAN, Y.; SAMWORTH, R. J. Classification with imperfect training labels. *Biometrika*, Oxford University Press, v. 107, n. 2, p. 311–330, 2020.

CAO, J.; KWONG, S.; WANG, R. A noise-detection based adaboost algorithm for mislabeled data. *Pattern Recognition*, Elsevier, v. 45, n. 12, p. 4451–4465, 2012.

CAULO, M.; LIN, B.; BAVOTA, G.; SCANNIELLO, G.; LANZA, M. Knowledge transfer in modern code review. In: *Proceedings of the 28th International Conference on Program Comprehension*. [S.l.: s.n.], 2020. p. 230–240.

CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, v. 16, p. 321–357, 2002.

CHEN, B.; XIA, S.; CHEN, Z.; WANG, B.; WANG, G. Rsmote: A self-adaptive robust smote for imbalanced problems with label noise. *Information Sciences*, Elsevier, v. 553, p. 397–428, 2021.

CHEN, L.; FANG, B.; SHANG, Z.; TANG, Y. Tackling class overlap and imbalance problems in software defect prediction. *Software Quality Journal*, Springer, v. 26, n. 1, p. 97–125, 2018.

CHEN, T.-H.; NAGAPPAN, M.; SHIHAB, E.; HASSAN, A. E. An empirical study of dormant bugs. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2014. p. 82–91.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, IEEE, v. 20, n. 6, p. 476–493, 1994.

CHOUDHARY, G. R.; KUMAR, S.; KUMAR, K.; MISHRA, A.; CATAL, C. Empirical analysis of change metrics for software fault prediction. *Computers & Electrical Engineering*, Elsevier, v. 67, p. 15–24, 2018.

CLARK, P.; NIBLETT, T. The cn2 induction algorithm. *Machine learning*, Springer, v. 3, n. 4, p. 261–283, 1989.

COHEN, J.; BROWN, E.; DURETTE, B.; TELEKI, S. *Best kept secrets of peer code review*. [S.l.]: Smart Bear Somerville, 2006.

COSTA, D. A. D.; MCINTOSH, S.; SHANG, W.; KULESZA, U.; COELHO, R.; HASSAN, A. E. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, IEEE, v. 43, n. 7, p. 641–657, 2016.

DALAL, P.; SHAH, A. Allocation of test cases using severity and data interactivity. *Int J Modern Trends Sci Technol*, v. 3, n. 1, 2017.

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. [S.l.]: Elsevier Brasil, 2013.

DELANY, S. J.; CUNNINGHAM, P. An analysis of case-base editing in a spam filtering system. In: SPRINGER. *European Conference on Case-Based Reasoning*. [S.l.], 2004. p. 128–141.

DENOEUX, T. A neural network classifier based on dempster-shafer theory. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, IEEE, v. 30, n. 2, p. 131–150, 2000.

DICK, S.; MEEKS, A.; LAST, M.; BUNKE, H.; KANDEL, A. Data mining in software metrics databases. *Fuzzy Sets and Systems*, Elsevier, v. 145, n. 1, p. 81–110, 2004.

DIETTERICH, T. G. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning*, Springer, v. 40, n. 2, p. 139–157, 2000.

DUA, D.; GRAFF, C. et al. Uci machine learning repository. 2017.

DUBINSKY, Y.; RUBIN, J.; BERGER, T.; DUSZYNSKI, S.; BECKER, M.; CZARNECKI, K. An exploratory study of cloning in industrial software product lines. In: IEEE. *2013 17th European Conference on Software Maintenance and Reengineering*. [S.l.], 2013. p. 25–34.

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

EBERT, C.; CAIN, J.; ANTONIOL, G.; COUNSELL, S.; LAPLANTE, P. Cyclomatic complexity. *IEEE software*, IEEE, v. 33, n. 6, p. 27–29, 2016.

EGELMAN, C. D.; MURPHY-HILL, E.; KAMMER, E.; HODGES, M. M.; GREEN, C.; JASPAN, C.; LIN, J. Predicting developers' negative feelings about code review. In: IEEE. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. [S.l.], 2020. p. 174–185.

FAGAN, M. E. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, IBM, v. 38, n. 2.3, p. 258–287, 1999.

FALESSI, D.; AHLUWALIA, A.; PENTA, M. D. The impact of dormant defects on defect prediction: A study of 19 apache projects. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, v. 31, n. 1, set. 2021. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/3467895>.

FALESSI, D.; RUSSO, B.; MULLEN, K. What if i had no smells? In: IEEE. *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.], 2017. p. 78–84.

FAN, Y.; XIA, X.; COSTA, D.; LO, D.; HASSAN, A. E.; LI, S. The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering*, PP, p. 1–1, 07 2019.

FAN, Y.; XIA, X.; LO, D.; LI, S. Early prediction of merged code changes to prioritize reviewing tasks. *Empirical Software Engineering*, Springer, v. 23, n. 6, p. 3346–3393, 2018.

FENG, S.; KEUNG, J.; LIU, J.; XIAO, Y.; YU, X.; ZHANG, M. Roct: Radius-based class overlap cleaning technique to alleviate the class overlap problem in software defect prediction. In: IEEE. *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. [S.l.], 2021. p. 228–237.

FRASER, G.; ROJAS, J. M. Software testing. In: *Handbook of Software Engineering*. [S.l.]: Springer, 2019. p. 123–192.

FRÉNAY, B.; VERLEYSEN, M. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems*, IEEE, v. 25, n. 5, p. 845–869, 2013.

FRÉNAY, B.; VERLEYSEN, M. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems*, IEEE, v. 25, n. 5, p. 845–869, 2014.

FREUND, Y.; SCHAPIRE, R. E. et al. Experiments with a new boosting algorithm. In: CITESEER. *icml*. [S.l.], 1996. v. 96, p. 148–156.

GABA, A.; WINKLER, R. L. Implications of errors in survey data: a bayesian model. *Management Science*, INFORMS, v. 38, n. 7, p. 913–925, 1992.

GALAR, M.; FERNANDEZ, A.; BARRENECHEA, E.; BUSTINCE, H.; HERRERA, F. A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, IEEE, v. 42, n. 4, p. 463–484, 2011.

GALIN, D. *Software quality assurance: from theory to implementation*. [S.l.]: Pearson Education India, 2004.

GAMBERGER, D.; LAVRAC, N.; GROSELJ, C. Experiments with noise filtering in a medical domain. In: *ICML*. [S.l.: s.n.], 1999. p. 143–151.

GARCIA, L. P.; LEHMANN, J.; CARVALHO, A. C. de; LORENA, A. C. New label noise injection methods for the evaluation of noise filters. *Knowledge-Based Systems*, Elsevier, v. 163, p. 693–704, 2019.

GKORTZIS, A.; FEITOSA, D.; SPINELLIS, D. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software*, Elsevier, v. 172, p. 110653, 2021.

GONG, L.; JIANG, S.; WANG, R.; JIANG, L. Empirical evaluation of the impact of class overlap on software defect prediction. In: IEEE. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2019. p. 698–709.

GROVE, A. J.; SCHUURMANS, D. Boosting in the limit: Maximizing the margin of learned ensembles. In: *AAAI/IAAI*. [S.l.: s.n.], 1998. p. 692–699.

GUPTA, S.; GUPTA, A. A set of measures designed to identify overlapped instances in software defect prediction. *Computing*, Springer, v. 99, n. 9, p. 889–914, 2017.

GUPTA, S.; GUPTA, A. Handling class overlapping to detect noisy instances in classification. *The Knowledge Engineering Review*, Cambridge University Press, v. 33, 2018.

HALL, T.; BEECHAM, S.; BOWES, D.; GRAY, D.; COUNSELL, S. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, IEEE, v. 38, n. 6, p. 1276–1304, 2012.

HE, H.; GARCIA, E. A. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, Ieee, v. 21, n. 9, p. 1263–1284, 2009.

HE, P.; LI, B.; LIU, X.; CHEN, J.; MA, Y. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, Elsevier, v. 59, p. 170–190, 2015.

HERZIG, K.; JUST, S.; ZELLER, A. It's not a bug, it's a feature: how misclassification impacts bug prediction. In: IEEE PRESS. *Proceedings of the 2013 international conference on software engineering*. [S.l.], 2013. p. 392–401.

HOLZMANN, G. J. The logic of bugs. *ACM SIGSOFT Software Engineering Notes*, v. 27, n. 6, p. 81–87, 2002.

HOSSAIN, S. S.; ARAFAT, Y.; HOSSAIN, M. E.; ARMAN, M. S.; ISLAM, A. Measuring the effectiveness of software code review comments. In: SPRINGER. *International Conference on Advances in Computing and Data Sciences*. [S.l.], 2020. p. 247–257.

IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, p. 1–84, Dec 1990.

IQBAL, A.; AFTAB, S.; ALI, U.; NAWAZ, Z.; SANA, L.; AHMAD, M.; HUSEN, A. Performance analysis of machine learning techniques on software defect prediction using nasa datasets. *Int. J. Adv. Comput. Sci. Appl*, v. 10, n. 5, p. 300–308, 2019.

ISLAM, J. F.; MONDAL, M.; ROY, C. K.; SCHNEIDER, K. A. Comparing bug replication in regular and micro code clones. In: IEEE. *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. [S.l.], 2019. p. 81–92.

JAPKOWICZ, N.; STEPHEN, S. The class imbalance problem: A systematic study. *Intelligent data analysis*, IOS Press, v. 6, n. 5, p. 429–449, 2002.

JOORABCHI, M. E.; MIRZAAGHAEI, M.; MESBAH, A. Works for me! characterizing non-reproducible bug reports. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2014. p. 62–71.

JURECZKO, M.; MADEYSKI, L. Towards identifying software project clusters with regard to defect prediction. In: *6th Intern. Conf. on Predictive Models in Software Engineering*. [S.l.: s.n.].

KALYAN, A.; CHIAM, M.; SUN, J.; MANOHARAN, S. A collaborative code review platform for github. In: IEEE. *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*. [S.l.], 2016. p. 191–196.

KARMAKER, A.; KWEK, S. A boosting approach to remove class label noise 1. *International Journal of Hybrid Intelligent Systems*, IOS Press, v. 3, n. 3, p. 169–177, 2006.

KHAN, S. S.; NILOY, N. T.; AZMAIN, M. A.; KABIR, A. Impact of label noise and efficacy of noise filters in software defect prediction. In: *SEKE*. [S.l.: s.n.], 2020. p. 347–352.

KHARDON, R.; WACHMAN, G. Noise tolerant variants of the perceptron algorithm. *Journal of Machine Learning Research*, v. 8, n. 2, 2007.

KHOSHGOFTAAR, T. M.; REBOURS, P. Improving software quality prediction by noise filtering techniques. *Journal of Computer Science and Technology*, Springer, v. 22, n. 3, p. 387–396, 2007.

KIM, S.; ZHANG, H.; WU, R.; GONG, L. Dealing with noise in defect prediction. In: IEEE. *2011 33rd International Conference on Software Engineering (ICSE)*. [S.l.], 2011. p. 481–490.

KOTSIANTIS, S. B.; ZAHARAKIS, I.; PINTELAS, P. et al. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, Amsterdam, v. 160, n. 1, p. 3–24, 2007.

KRAWCZYK, B. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence*, Springer, v. 5, n. 4, p. 221–232, 2016.

KRÜGER, J.; BERGER, T. An empirical analysis of the costs of clone-and platform-oriented software reuse. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2020. p. 432–444.

KRÜGER, J.; MAHMOOD, W.; BERGER, T. Promote-pl: a round-trip engineering process model for adopting and evolving product lines. In: *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*. [S.l.: s.n.], 2020. p. 1–12.

KUHA, J.; SKINNER, C.; PALMGREN, J. Misclassification error. *Wiley StatsRef: Statistics Reference Online*, Wiley Online Library, 2014.

KUMAR, V.; MINZ, S. Feature selection: a literature review. *SmartCR*, v. 4, n. 3, p. 211–229, 2014.

LENARDUZZI, V.; LOMIO, F.; MORESCHINI, S.; TAIBI, D.; TAMBURRI, D. A. Software quality for ai: Where we are now? In: SPRINGER. *International Conference on Software Quality*. [S.l.], 2021. p. 43–53.

LI, J.; HE, P.; ZHU, J.; LYU, M. R. Software defect prediction via convolutional neural network. In: IEEE. *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. [S.l.], 2017. p. 318–328.

LI, L.; LESSMANN, S.; BAESENS, B. Evaluating software defect prediction performance: an updated benchmarking study. *arXiv preprint arXiv:1901.01726*, 2019.

LI, Z.; JING, X.-Y.; ZHU, X. Progress on approaches to software defect prediction. *Iet Software*, IET, v. 12, n. 3, p. 161–175, 2018.

LINSBAUER, L.; SCHWÄGERL, F.; BERGER, T.; GRÜNBACHER, P. Concepts of variation control systems. *Journal of Systems and Software*, Elsevier, v. 171, p. 110796, 2021.

LIU, S.; CHEN, X.; LIU, W.; CHEN, J.; GU, Q.; CHEN, D. Fecar: A feature selection framework for software defect prediction. In: IEEE. *2014 IEEE 38th Annual Computer Software and Applications Conference*. [S.l.], 2014. p. 426–435.

LIU, W.; LIU, S.; GU, Q.; CHEN, X.; CHEN, D. Fecs: A cluster based feature selection method for software fault prediction with noises. In: IEEE. *2015 IEEE 39th Annual Computer Software and Applications Conference*. [S.l.], 2015. v. 2, p. 276–281.

LOELIGER, J.; MCCULLOUGH, M. *Version Control with Git: Powerful tools and techniques for collaborative software development*. [S.l.]: "O'Reilly Media, Inc.", 2012.

LÓPEZ, V.; FERNÁNDEZ, A.; GARCÍA, S.; PALADE, V.; HERRERA, F. An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. *Information sciences*, Elsevier, v. 250, p. 113–141, 2013.

MABAYOJE, M. A.; BALOGUN, A. O.; JIBRIL, H. A.; ATOYEBI, J. O.; MOJEED, H. A.; ADEYEMO, V. E. Parameter tuning in knn for software defect prediction: an empirical analysis. Department of Computer Engineering, Universitas Diponegoro, Indonesia., 2019.

MALHOTRA, R. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, Elsevier, v. 27, p. 504–518, 2015.

MALOSSINI, A.; BLANZIERI, E.; NG, R. T. Detecting potential labeling errors in microarrays by data perturbation. *Bioinformatics*, Oxford University Press, v. 22, n. 17, p. 2114–2121, 2006.

MANWANI, N.; SASTRY, P. Noise tolerance under risk minimization. *IEEE transactions on cybernetics*, IEEE, v. 43, n. 3, p. 1146–1151, 2013.

MCINTOSH, S.; KAMEI, Y.; ADAMS, B.; HASSAN, A. E. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2014. p. 192–201.

MICHAELS, P. Faulty software can lead to astronomic costs. *Computer Weekly (June), http://www. computerweekly. com/Articles/2009/01/13/230950/Faulty-software-can-lead-to-astronomic-costs. htm*, 2008.

MIRANDA, A. L.; GARCIA, L. P. F.; CARVALHO, A. C.; LORENA, A. C. Use of classification algorithms in noise detection and elimination. In: SPRINGER. *International Conference on Hybrid Artificial Intelligence Systems*. [S.l.], 2009. p. 417–424.

MOHAGHEGHI, P.; CONRADI, R.; KILLI, O. M.; SCHWARZ, H. An empirical study of software reuse vs. defect-density and stability. In: IEEE. *Proceedings. 26th International Conference on Software Engineering*. [S.l.], 2004. p. 282–291.

MONDAL, M.; ROY, B.; ROY, C. K.; SCHNEIDER, K. A. Investigating context adaptation bugs in code clones. In: IEEE. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2019. p. 157–168.

MONDAL, M.; ROY, C. K.; SCHNEIDER, K. A. Bug propagation through code cloning: An empirical study. In: IEEE. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2017. p. 227–237.

MORALES, P.; LUENGO, J.; LUíS, P.; ANA, C.; CARVALHO, A. de; HERRERA, F. The noisefiltersr package: Label noise preprocessing in r. *The R Journal*, v. 9, p. 219, 01 2017.

MOSER, R.; PEDRYCZ, W.; SUCCI, G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: ACM. *Proceedings of the 30th international conference on Software engineering*. [S.l.], 2008. p. 181–190.

MOURA, K. G. de; PRUDENCIO, R. B.; CAVALCANTI, G. D. Ensemble methods for label noise detection under the noisy at random model. In: IEEE. *2018 7th Brazilian Conference on Intelligent Systems (BRACIS)*. [S.l.], 2018. p. 474–479.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The art of software testing*. [S.l.]: John Wiley & Sons, 2011.

NAGAPPAN, N. Toward a software testing and reliability early warning metric suite. In: IEEE COMPUTER SOCIETY. *Proceedings of the 26th international conference on software engineering*. [S.l.], 2004. p. 60–62.

NAM, J. Survey on software defect prediction. *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

NAZARI, Z.; NAZARI, M.; DANISH, M. S. S.; KANG, D. Evaluation of class noise impact on performance of machine learning algorithms. *IJCSNS*, v. 18, n. 8, p. 149, 2018.

NETO, E. C.; COSTA, D. A. da; KULESZA, U. The impact of refactoring changes on the szz algorithm: An empirical study. In: IEEE. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.], 2018. p. 380–390.

NETTLETON, D. F.; ORRIOLS-PUIG, A.; FORNELLS, A. A study of the effect of different types of noise on the precision of supervised learning techniques. *Artificial intelligence review*, Springer, v. 33, n. 4, p. 275–306, 2010.

OKUTAN, A.; YILDIZ, O. T. Software defect prediction using bayesian networks. *Empirical Software Engineering*, Springer, v. 19, n. 1, p. 154–181, 2014.

ORAM, A.; WILSON, G. *Making software: What really works, and why we believe it*. [S.l.]: "O'Reilly Media, Inc.", 2010.

ÖZAKINCI, R.; TARHAN, A. Early software defect prediction: A systematic map and review. *Journal of Systems and Software*, Elsevier, v. 144, p. 216–239, 2018.

PANDEY, S. K.; TRIPATHI, A. K. An empirical study toward dealing with noise and class imbalance issues in software defect prediction. *Soft Computing*, Springer, v. 25, n. 21, p. 13465–13492, 2021.

PASCARELLA, L.; SPADINI, D.; PALOMBA, F.; BRUNTINK, M.; BACCHELLI, A. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction*, ACM New York, NY, USA, v. 2, n. CSCW, p. 1–27, 2018.

PELAYO, L.; DICK, S. Applying novel resampling strategies to software defect prediction. In: IEEE. *NAFIPS 2007-2007 Annual Meeting of the North American Fuzzy Information Processing Society*. [S.l.], 2007. p. 69–72.

PIETRANTUONO, R.; POTENA, P.; PECCHIA, A.; RODRIGUEZ, D.; RUSSO, S.; FERNÁNDEZ-SANZ, L. Multiobjective testing resource allocation under uncertainty. *IEEE Transactions on Evolutionary Computation*, IEEE, v. 22, n. 3, p. 347–362, 2018.

PRATI, R. C.; BATISTA, G. E.; MONARD, M. C. Class imbalances versus class overlapping: an analysis of a learning system behavior. In: SPRINGER. *Mexican international conference on artificial intelligence*. [S.l.], 2004. p. 312–321.

PRESSMAN, R.; MAXIM, B. *Engenharia de Software-8ª Edição*. [S.l.]: McGraw Hill Brasil, 2016.

RÄTSCH, G.; ONODA, T.; MÜLLER, K.-R. Soft margins for adaboost. *Machine learning*, Springer, v. 42, n. 3, p. 287–320, 2001.

REEVE, H.; KABÁN, A. Fast rates for a knn classifier robust to unknown asymmetric label noise. In: PMLR. *International Conference on Machine Learning*. [S.l.], 2019. p. 5401–5409.

RIAZ, S.; ARSHAD, A.; JIAO, L. Rough noise-filtered easy ensemble for software fault prediction. *IEEE Access*, IEEE, v. 6, p. 46886–46899, 2018.

RUANGWAN, S.; THONGTANUNAM, P.; IHARA, A.; MATSUMOTO, K. The impact of human factors on the participation decision of reviewers in modern code review. *Empirical Software Engineering*, Springer, v. 24, n. 2, p. 973–1016, 2019.

SABZEVARI, M.; MARTÍNEZ-MUÑOZ, G.; SUÁREZ, A. A two-stage ensemble method for the detection of class-label noise. *Neurocomputing*, Elsevier, v. 275, p. 2374–2383, 2018.

SADOWSKI, C.; SÖDERBERG, E.; CHURCH, L.; SIPKO, M.; BACCHELLI, A. Modern code review: a case study at google. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. [S.l.: s.n.], 2018. p. 181–190.

SAIFUDIN, A.; HERYADI, Y. et al. Ensemble undersampling to handle unbalanced class on cross-project defect prediction. In: IOP PUBLISHING. *IOP Conference Series: Materials Science and Engineering*. [S.l.], 2019. v. 662, n. 6, p. 062012.

SÁNCHEZ, J. S.; PLA, F.; FERRI, F. J. Prototype selection for the nearest neighbour rule through proximity graphs. *Pattern Recognition Letters*, Elsevier, v. 18, n. 6, p. 507–513, 1997.

SCHRÖTER, A.; ZIMMERMANN, T.; ZELLER, A. Predicting component failures at design time. In: ACM. *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. [S.l.], 2006. p. 18–27.

ŚLIWERSKI, J.; ZIMMERMANN, T.; ZELLER, A. When do changes induce fixes? In: ACM. *ACM sigsoft software engineering notes*. [S.l.], 2005. v. 30, n. 4, p. 1–5.

SMITH, H. T. G. *Software Quality Assurance: A Guide for Developers and Auditors*. [S.l.]: CRC Press, 2020.

SMITH, M. R.; MARTINEZ, T.; GIRAUD-CARRIER, C. An instance level analysis of data complexity. *Machine learning*, Springer, v. 95, n. 2, p. 225–256, 2014.

SOMMERVILLE, I. Software engineering 9th edition. *ISBN-10*, v. 137035152, p. 18, 2011.

SUN, Y.; WONG, A. K.; KAMEL, M. S. Classification of imbalanced data: A review. *International journal of pattern recognition and artificial intelligence*, World Scientific, v. 23, n. 04, p. 687–719, 2009.

SWARTZ, T. B.; HAITOVSKY, Y.; VEXLER, A.; YANG, T. Y. Bayesian identifiability and misclassification in multinomial data. *Canadian Journal of Statistics*, Wiley Online Library, v. 32, n. 3, p. 285–302, 2004.

TANTITHAMTHAVORN, C.; MCINTOSH, S.; HASSAN, A. E.; IHARA, A.; MATSUMOTO, K. The impact of mislabelling on the performance and interpretation of defect prediction models. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.], 2015. v. 1, p. 812–823.

THONGTANUNAM, P.; MCINTOSH, S.; HASSAN, A. E.; IIDA, H. Review participation in modern code review. *Empirical Software Engineering*, Springer, v. 22, n. 2, p. 768–817, 2017.

THOTA, M. K.; SHAJIN, F. H.; RAJESH, P. et al. Survey on software defect prediction techniques. *International Journal of Applied Science and Engineering*, Chaoyang University of Technology, v. 17, n. 4, p. 331–344, 2020.

TOMEK, I. An experiment with the edited nearest-neighbor rule. *IEEE Transactions on systems, Man, and Cybernetics*, n. 6, p. 448–452, 1976.

TUFAN, R.; PASCARELLA, L.; TUFANOY, M.; POSHYVANYKZ, D.; BAVOTA, G. Towards automating code review activities. In: IEEE. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. [S.l.], 2021. p. 163–174.

TURHAN, B.; MENZIES, T.; BENER, A. B.; STEFANO, J. D. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, Springer, v. 14, n. 5, p. 540–578, 2009.

TÜZÜN, E.; TEKINERDOGAN, B. Analyzing impact of experience curve on roi in the software product line adoption process. *Information and Software Technology*, Elsevier, v. 59, p. 136–148, 2015.

VANDEHEI, B.; COSTA, D. A. D.; FALESSI, D. Leveraging the defects life cycle to label affected versions and defective classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, USA, v. 30, n. 2, p. 1–35, 2021.

VANSCHOREN, J.; RIJN, J. N. V.; BISCHL, B.; TORGO, L. Openml: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, ACM New York, NY, USA, v. 15, n. 2, p. 49–60, 2014.

VASHISHT, R.; RIZVI, S. A. M. An empirical study of heterogeneous cross-project defect prediction using various statistical techniques. *International Journal of e-Collaboration (IJeC)*, IGI Global, v. 17, n. 2, p. 55–71, 2021.

VUTTIPITTAYAMONGKOL, P.; ELYAN, E.; PETROVSKI, A. On the class overlap problem in imbalanced data classification. *Knowledge-based systems*, Elsevier, v. 212, p. 106631, 2021.

WAHONO, R. S. A systematic literature review of software defect prediction. *Journal of Software Engineering*, v. 1, n. 1, p. 1–16, 2015.

WAHONO, R. S.; HERMAN, N. S.; AHMAD, S. A comparison framework of classification models for software defect prediction. *Advanced Science Letters*, American Scientific Publishers, v. 20, n. 10-11, p. 1945–1950, 2014.

WANG, B. X.; JAPKOWICZ, N. Imbalanced data set learning with synthetic samples. In: SN. *Proc. IRIS Machine Learning Workshop*. [S.l.], 2004. v. 19, p. 435.

WANG, H.; KHOSHGOFTAAR, T. M.; NAPOLITANO, A. A comparative study of ensemble feature selection techniques for software defect prediction. In: IEEE. *2010 Ninth International Conference on Machine Learning and Applications*. [S.l.], 2010. p. 135–140.

WANG, S.; YAO, X. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, IEEE, v. 62, n. 2, p. 434–443, 2013.

WANG, Y.; JHA, S.; CHAUDHURI, K. Analyzing the robustness of nearest neighbors to adversarial examples. In: PMLR. *International Conference on Machine Learning*. [S.l.], 2018. p. 5133–5142.

WARDEN, S.; SHORE, J. *The art of agile development*. [S.l.]: O'Reilly Media, Incorporated, 2007.

WILSON, D. L. Asymptotic properties of nearest neighbor rules using edited data. *IEEE Transactions on Systems, Man, and Cybernetics*, IEEE, n. 3, p. 408–421, 1972.

XIA, X.; LIU, T.; HAN, B.; WANG, N.; GONG, M.; LIU, H.; NIU, G.; TAO, D.; SUGIYAMA, M. Part-dependent label noise: Towards instance-dependent label noise. *Advances in Neural Information Processing Systems*, v. 33, 2020.

XU, Z.; LIU, J.; LUO, X.; YANG, Z.; ZHANG, Y.; YUAN, P.; TANG, Y.; ZHANG, T. Software defect prediction based on kernel pca and weighted extreme learning machine. *Information and Software Technology*, Elsevier, v. 106, p. 182–200, 2019.

XU, Z.; LIU, J.; YANG, Z.; AN, G.; JIA, X. The impact of feature selection on defect prediction performance: An empirical comparison. In: IEEE. *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.], 2016. p. 309–320.

YANG, X.; TANG, K.; YAO, X. A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability*, IEEE, v. 64, n. 1, p. 234–246, 2015.

YATISH, S.; JIARPAKDEE, J.; THONGTANUNAM, P.; TANTITHAMTHAVORN, C. Mining software defects: should we consider affected releases? In: IEEE. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. [S.l.], 2019. p. 654–665.

YOUM, K. C.; AHN, J.; LEE, E. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, Elsevier, v. 82, p. 177–192, 2017.

ZHANG, H.; ZHANG, X. Comments on"data mining static code attributes to learn defect predictors". *IEEE Transactions on Software Engineering*, IEEE, v. 33, n. 9, p. 635–637, 2007.

ZHANG, N.; ZHU, K.; YING, S.; WANG, X. Kaea: A novel three-stage ensemble model for software defect prediction. *Computers, Materials and Continua*, Tech Science Press, v. 64, n. 1, p. 471–499, 2020.

ZHU, X.; WU, X. Class noise vs. attribute noise: A quantitative study. *Artificial intelligence review*, Springer, v. 22, n. 3, p. 177–210, 2004.

APÊNDICE A – RESULTADOS DO ESTUDO DE CASO 1

Tabela 8 – Resultados do KNN utilizando K=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	1.47%	80.00%	0.74%
IVY	1.15%	30.00%	0.59%
OPENJPA	2.02%	57.14%	1.03%
PROTON	2.65%	60.00%	1.36%
SSHD	1.52%	100.00%	0.76%
STORM	5.87%	86.21%	3.04%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	8.05%	71.43%	4.27%
ZOOKEEPER	1.72%	100.00%	0.87%
Média	**2.45%**	**58.48%**	**1.27%**

Fonte: Elaborada pelo autor (2022)

Tabela 9 – Resultados do KNN utilizando K=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	3.38%	42.86%	1.76%
OPENJPA	2.25%	42.86%	1.16%
PROTON	5.22%	66.67%	2.71%
SSHD	0.00%	0.00%	0.00%
STORM	9.81%	78.90%	5.23%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	11.39%	78.26%	6.14%
ZOOKEEPER	2.29%	80.00%	1.16%
Média	**3.43%**	**38.95%**	**1.82%**

Fonte: Elaborada pelo autor (2022)

Tabela 10 – Resultados do Boost utilizando N=5 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	30.09%	61.08%	19.96%
OPENJPA	30.27%	44.72%	22.88%
PROTON	26.07%	24.70%	27.60%
SSHD	6.31%	7.69%	5.34%
STORM	10.19%	73.77%	5.47%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	44.91%	53.41%	38.74%
ZOOKEEPER	27.98%	23.15%	35.36%
Média	**19.07%**	**30.04%**	**17.54%**

Fonte: Elaborada pelo autor (2022)

Tabela 11 – Resultados do Boost utilizando N=5 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	30.09%	61.08%	19.96%
OPENJPA	30.27%	44.72%	22.88%
PROTON	26.07%	24.70%	27.60%
SSHD	6.31%	7.69%	5.34%
STORM	10.19%	73.77%	5.47%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	44.91%	53.41%	38.74%
ZOOKEEPER	0.00%	0.00%	0.00%
Média	**16.28%**	**27.73%**	**14.00%**

Fonte: Elaborada pelo autor (2022)

Tabela 12 – Resultados do Boost utilizando N=10 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	18.93%	10.57%	90.80%
OPENJPA	12.69%	6.84%	88.69%
PROTON	26.11%	24.41%	28.05%
SSHD	6.31%	7.69%	5.34%
STORM	37.53%	23.76%	89.36%
WHIRR	15.76%	8.76%	78.38%
ZEPPELIN	47.36%	54.05%	42.15%
ZOOKEEPER	29.02%	23.84%	37.10%
Média	**20.86%**	**17.18%**	**47.99%**

Fonte: Elaborada pelo autor (2022)

Tabela 13 – Resultados do Boost utilizando N=10 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	18.93%	10.57%	90.80%
OPENJPA	12.69%	6.84%	88.69%
PROTON	26.27%	24.70%	28.05%
SSHD	6.31%	7.69%	5.34%
STORM	37.68%	23.84%	89.91%
WHIRR	16.36%	9.48%	59.46%
ZEPPELIN	44.93%	53.15%	38.91%
ZOOKEEPER	27.98%	23.15%	35.36%
Média	**20.61%**	**17.13%**	**45.65%**

Fonte: Elaborada pelo autor (2022)

Tabela 14 – Resultados do Boost utilizando N=15 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	28.40%	20.91%	44.24%
IVY	18.93%	10.57%	90.80%
OPENJPA	12.69%	6.84%	88.69%
PROTON	5.84%	3.02%	89.59%
SSHD	6.31%	7.69%	5.34%
STORM	40.75%	25.59%	100.00%
WHIRR	15.76%	8.76%	78.38%
ZEPPELIN	47.74%	53.14%	43.34%
ZOOKEEPER	29.50%	24.13%	37.97%
Média	**22.09%**	**17.25%**	**59.84%**

Fonte: Elaborada pelo autor (2022)

Tabela 15 – Resultados do Boost utilizando N=15 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	28.40%	20.91%	44.24%
IVY	18.93%	10.57%	90.80%
OPENJPA	12.69%	6.84%	88.69%
PROTON	5.84%	3.02%	89.59%
SSHD	6.31%	7.69%	5.34%
STORM	37.90%	23.95%	90.82%
WHIRR	15.76%	8.76%	78.38%
ZEPPELIN	45.96%	53.51%	40.27%
ZOOKEEPER	29.38%	24.07%	37.68%
Média	**21.61%**	**17.12%**	**58.58%**

Fonte: Elaborada pelo autor (2022)

Tabela 16 – Resultados do SEQ utilizando N=5, K=5 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	1.47%	80.00%	0.74%
IVY	28.25%	49.51%	19.77%
OPENJPA	30.10%	46.03%	22.37%
PROTON	19.95%	20.48%	19.46%
SSHD	19.42%	63.83%	11.45%
STORM	21.41%	78.16%	12.40%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	22.69%	20.95%	24.74%
ZOOKEEPER	18.89%	16.22%	22.61%
Média	**16.22%**	**37.52%**	**13.35%**

Fonte: Elaborada pelo autor (2022)

Tabela 17 – Resultados do SEQ utilizando N=5, K=10 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	31.78%	52.97%	22.70%
OPENJPA	2.25%	42.86%	1.16%
PROTON	15.44%	52.63%	9.05%
SSHD	6.59%	81.82%	3.44%
STORM	26.29%	81.13%	15.68%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	43.69%	50.68%	38.40%
ZOOKEEPER	2.29%	80.00%	1.16%
Média	**12.83%**	**44.21%**	**9.16%**

Fonte: Elaborada pelo autor (2022)

Tabela 18 – Resultados do SEQ utilizando N=5, K=5 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	1.47%	80.00%	0.74%
IVY	28.25%	49.51%	19.77%
OPENJPA	30.10%	46.03%	22.37%
PROTON	19.95%	20.48%	19.46%
SSHD	19.42%	63.83%	11.45%
STORM	21.41%	78.16%	12.40%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	22.69%	20.95%	24.74%
ZOOKEEPER	18.89%	16.22%	22.61%
Média	**16.22%**	**37.52%**	**13.35%**

Fonte: Elaborada pelo autor (2022)

Tabela 19 – Resultados do SEQ utilizando N=5, K=10 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	31.78%	52.97%	22.70%
OPENJPA	2.25%	42.86%	1.16%
PROTON	15.44%	52.63%	9.05%
SSHD	6.59%	81.82%	3.44%
STORM	26.29%	81.13%	15.68%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	43.62%	50.79%	38.23%
ZOOKEEPER	2.29%	80.00%	1.16%
Média	**12.83%**	**44.22%**	**9.14%**

Fonte: Elaborada pelo autor (2022)

Tabela 20 – Resultados do SEQ utilizando N=10, K=5 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	1.47%	80.00%	0.74%
IVY	20.15%	11.20%	99.80%
OPENJPA	13.23%	7.12%	93.32%
PROTON	6.06%	3.13%	94.12%
SSHD	39.85%	24.93%	99.24%
STORM	38.81%	24.50%	93.25%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	22.69%	20.95%	24.74%
ZOOKEEPER	19.30%	16.53%	23.19%
Média	**16.16%**	**18.84%**	**52.84%**

Fonte: Elaborada pelo autor (2022)

Tabela 21 – Resultados do SEQ utilizando N=10, K=10 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	34.11%	50.19%	25.83%
OPENJPA	2.25%	42.86%	1.16%
PROTON	6.16%	3.18%	96.38%
SSHD	39.63%	24.78%	98.85%
STORM	38.82%	24.53%	93.07%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	43.76%	50.56%	38.57%
ZOOKEEPER	2.29%	80.00%	1.16%
Média	**16.70%**	**27.61%**	**35.50%**

Fonte: Elaborada pelo autor (2022)

Tabela 22 – Resultados do SEQ utilizando N=10, K=5 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	1.47%	80.00%	0.74%
IVY	20.15%	11.20%	99.80%
OPENJPA	13.23%	7.12%	93.32%
PROTON	6.11%	3.16%	94.57%
SSHD	39.85%	24.93%	99.24%
STORM	39.03%	24.64%	93.92%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	22.69%	20.95%	24.74%
ZOOKEEPER	18.89%	16.22%	22.61%
Média	**16.14%**	**18.82%**	**52.89%**

Fonte: Elaborada pelo autor (2022)

Tabela 23 – Resultados do SEQ utilizando N=10, K=10 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	34.11%	50.19%	25.83%
OPENJPA	2.25%	42.86%	1.16%
PROTON	6.16%	3.18%	96.38%
SSHD	39.63%	24.78%	98.85%
STORM	38.82%	24.53%	93.07%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	43.58%	50.68%	38.23%
ZOOKEEPER	2.29%	80.00%	1.16%
Média	**16.68%**	**27.62%**	**35.47%**

Fonte: Elaborada pelo autor (2022)

Tabela 24 – Resultados do SEQ utilizando N=15, K=5 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	1.47%	80.00%	0.74%
IVY	20.15%	11.20%	99.80%
OPENJPA	13.23%	7.12%	93.32%
PROTON	6.06%	3.13%	94.12%
SSHD	39.85%	24.93%	99.24%
STORM	40.75%	25.59%	100.00%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	22.69%	20.95%	24.74%
ZOOKEEPER	19.30%	16.53%	23.19%
Média	**16.35%**	**18.95%**	**53.52%**

Fonte: Elaborada pelo autor (2022)

Tabela 25 – Resultados do SEQ utilizando N=15, K=10 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	19.66%	10.94%	97.06%
OPENJPA	2.25%	42.86%	1.16%
PROTON	6.16%	3.18%	96.38%
SSHD	39.63%	24.78%	98.85%
STORM	38.82%	24.53%	93.07%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	43.82%	50.44%	38.74%
ZOOKEEPER	2.29%	80.00%	1.16%
Média	**15.26%**	**23.67%**	**42.64%**

Fonte: Elaborada pelo autor (2022)

Tabela 26 – Resultados do SEQ utilizando N=15, K=5 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	1.47%	80.00%	0.74%
IVY	20.15%	11.20%	99.80%
OPENJPA	13.23%	7.12%	93.32%
PROTON	6.11%	3.16%	94.57%
SSHD	39.85%	24.93%	99.24%
STORM	40.75%	25.59%	100.00%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	22.69%	20.95%	24.74%
ZOOKEEPER	19.32%	16.56%	23.19%
Média	**16.36%**	**18.95%**	**53.56%**

Fonte: Elaborada pelo autor (2022)

Tabela 27 – Resultados do SEQ utilizando N=15, K=10 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	0.00%	0.00%	0.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	19.66%	10.94%	97.06%
OPENJPA	2.25%	42.86%	1.16%
PROTON	6.16%	3.18%	96.38%
SSHD	39.63%	24.78%	98.85%
STORM	38.82%	24.53%	93.07%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	43.58%	50.68%	38.23%
ZOOKEEPER	2.29%	80.00%	1.16%
Média	**15.24%**	**23.70%**	**42.59%**

Fonte: Elaborada pelo autor (2022)

Tabela 28 – Resultados do F3R utilizando N=5, K=5 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	35.38%	44.51%	29.35%
OPENJPA	33.25%	32.36%	34.19%
PROTON	26.09%	19.51%	39.37%
SSHD	6.31%	7.69%	5.34%
STORM	27.32%	72.32%	16.84%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	47.51%	54.15%	42.32%
ZOOKEEPER	27.98%	23.15%	35.36%
Média	**21.88%**	**26.56%**	**22.28%**

Fonte: Elaborada pelo autor (2022)

Tabela 29 – Resultados do F3R utilizando N=5, K=10 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	35.38%	44.51%	29.35%
OPENJPA	33.25%	32.36%	34.19%
PROTON	26.35%	19.69%	39.82%
SSHD	6.31%	7.69%	5.34%
STORM	27.32%	72.32%	16.84%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	47.61%	54.13%	42.49%
ZOOKEEPER	27.98%	23.15%	35.36%
Média	**21.91%**	**26.58%**	**22.34%**

Fonte: Elaborada pelo autor (2022)

Tabela 30 – Resultados do F3R utilizando N=5, K=5 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	35.38%	44.51%	29.35%
OPENJPA	33.25%	32.36%	34.19%
PROTON	26.09%	19.51%	39.37%
SSHD	6.31%	7.69%	5.34%
STORM	27.32%	72.32%	16.84%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	44.91%	53.41%	38.74%
ZOOKEEPER	0.00%	0.00%	0.00%
Média	**18.82%**	**24.17%**	**18.38%**

Fonte: Elaborada pelo autor (2022)

Tabela 31 – Resultados do F3R utilizando N=5, K=10 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	35.38%	44.51%	29.35%
OPENJPA	33.25%	32.36%	34.19%
PROTON	26.35%	19.69%	39.82%
SSHD	6.31%	7.69%	5.34%
STORM	27.32%	72.32%	16.84%
WHIRR	0.00%	0.00%	0.00%
ZEPPELIN	44.86%	53.29%	38.74%
ZOOKEEPER	0.00%	0.00%	0.00%
Média	**18.84%**	**24.18%**	**18.43%**

Fonte: Elaborada pelo autor (2022)

Tabela 32 – Resultados do F3R utilizando N=10, K=5 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	40.44%	35.74%	46.58%
OPENJPA	35.61%	29.10%	45.89%
PROTON	24.75%	16.33%	51.13%
SSHD	6.31%	7.69%	5.34%
STORM	41.33%	62.48%	30.88%
WHIRR	9.62%	5.14%	75.68%
ZEPPELIN	48.67%	50.55%	46.93%
ZOOKEEPER	30.31%	24.51%	39.71%
Média	**25.20%**	**24.34%**	**36.21%**

Fonte: Elaborada pelo autor (2022)

Tabela 33 – Resultados do F3R utilizando N=10, K=10 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	40.44%	35.74%	46.58%
OPENJPA	35.61%	29.10%	45.89%
PROTON	25.23%	16.84%	50.23%
SSHD	6.31%	7.69%	5.34%
STORM	41.33%	62.48%	30.88%
WHIRR	9.62%	5.14%	75.68%
ZEPPELIN	48.76%	50.55%	47.10%
ZOOKEEPER	30.31%	24.51%	39.71%
Média	**25.25%**	**24.40%**	**36.14%**

Fonte: Elaborada pelo autor (2022)

Tabela 34 – Resultados do F3R utilizando N=10, K=5 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	40.44%	35.74%	46.58%
OPENJPA	35.61%	29.10%	45.89%
PROTON	24.75%	16.33%	51.13%
SSHD	6.31%	7.69%	5.34%
STORM	41.33%	62.48%	30.88%
WHIRR	16.36%	9.48%	59.46%
ZEPPELIN	48.67%	50.55%	46.93%
ZOOKEEPER	30.28%	24.46%	39.71%
Média	**25.87%**	**24.77%**	**34.59%**

Fonte: Elaborada pelo autor (2022)

Tabela 35 – Resultados do F3R utilizando N=10, K=10 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	0.00%	0.00%	0.00%
IVY	40.44%	35.74%	46.58%
OPENJPA	35.61%	29.10%	45.89%
PROTON	25.23%	16.84%	50.23%
SSHD	6.31%	7.69%	5.34%
STORM	41.33%	62.48%	30.88%
WHIRR	16.36%	9.48%	59.46%
ZEPPELIN	48.76%	50.55%	47.10%
ZOOKEEPER	30.28%	24.46%	39.71%
Média	**25.92%**	**24.83%**	**34.52%**

Fonte: Elaborada pelo autor (2022)

Tabela 36 – Resultados do F3R utilizando N=15, K=5 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	31.60%	22.51%	52.97%
IVY	41.16%	34.92%	50.10%
OPENJPA	35.54%	29.00%	45.89%
PROTON	24.49%	16.10%	51.13%
SSHD	6.31%	7.69%	5.34%
STORM	42.15%	60.07%	32.46%
WHIRR	9.57%	5.11%	75.68%
ZEPPELIN	49.92%	48.40%	51.54%
ZOOKEEPER	30.31%	24.51%	39.71%
Média	**28.60%**	**26.02%**	**42.48%**

Fonte: Elaborada pelo autor (2022)

Tabela 37 – Resultados do F3R utilizando N=15, K=10 e D=5

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	31.60%	22.51%	52.97%
IVY	41.16%	34.92%	50.10%
OPENJPA	35.54%	29.00%	45.89%
PROTON	24.49%	16.10%	51.13%
SSHD	6.31%	7.69%	5.34%
STORM	42.15%	60.07%	32.46%
WHIRR	9.57%	5.11%	75.68%
ZEPPELIN	50.00%	48.40%	51.71%
ZOOKEEPER	30.31%	24.51%	39.71%
Média	**28.60%**	**26.02%**	**42.50%**

Fonte: Elaborada pelo autor (2022)

Tabela 38 – Resultados do F3R utilizando N=15, K=5 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	28.40%	20.91%	44.24%
IVY	41.16%	34.92%	50.10%
OPENJPA	35.54%	29.00%	45.89%
PROTON	24.49%	16.10%	51.13%
SSHD	6.31%	7.69%	5.34%
STORM	42.15%	60.07%	32.46%
WHIRR	9.61%	5.13%	75.68%
ZEPPELIN	50.04%	49.42%	50.68%
ZOOKEEPER	30.28%	24.46%	39.71%
Média	**28.29%**	**25.96%**	**41.52%**

Fonte: Elaborada pelo autor (2022)

Tabela 39 – Resultados do F3R utilizando N=15, K=10 e D=10

Projeto	F1 Score	Precisão	Cobertura
AVRO	14.93%	11.90%	20.00%
GIRAPH	28.40%	20.91%	44.24%
IVY	41.16%	34.92%	50.10%
OPENJPA	35.54%	29.00%	45.89%
PROTON	24.49%	16.10%	51.13%
SSHD	6.31%	7.69%	5.34%
STORM	42.15%	60.07%	32.46%
WHIRR	9.61%	5.13%	75.68%
ZEPPELIN	50.13%	49.42%	50.85%
ZOOKEEPER	30.28%	24.46%	39.71%
Média	**28.30%**	**25.96%**	**41.54%**

Fonte: Elaborada pelo autor (2022)

APÊNDICE B – RESULTADOS DO ESTUDO DE CASO 2

Tabela 40 – Resultados do KNN utilizando K=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	86.27%	91.67%	81.48%
column2C	91.67%	100.00%	84.62%
credit	50.00%	55.56%	45.45%
glass1	33.33%	26.67%	44.44%
heart-c	30.00%	30.00%	30.00%
hill-valley	0.00%	0.00%	0.00%
mushroom	91.09%	98.92%	84.40%
pima	20.00%	26.32%	16.13%
sonar	50.00%	60.00%	42.86%
tic-tac-toe	48.35%	42.31%	56.41%
Média	**50.07%**	**53.14%**	**48.58%**

Fonte: Elaborada pelo autor (2022)

Tabela 41 – Resultados do KNN utilizando K=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	92.86%	89.66%	96.30%
column2C	83.87%	72.22%	100.00%
credit	45.90%	35.90%	63.64%
glass1	25.81%	18.18%	44.44%
heart-c	47.62%	45.45%	50.00%
hill-valley	0.00%	0.00%	0.00%
mushroom	93.04%	88.43%	98.17%
pima	15.15%	14.29%	16.13%
sonar	27.27%	20.00%	42.86%
tic-tac-toe	41.27%	29.89%	66.67%
Média	**47.28%**	**41.40%**	**57.82%**

Fonte: Elaborada pelo autor (2022)

Tabela 42 – Resultados do Boost utilizando N=5 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	89.66%	83.87%	96.30%
column2C	100.00%	100.00%	100.00%
credit	52.17%	50.00%	54.55%
glass1	77.78%	77.78%	77.78%
heart-c	57.14%	44.44%	80.00%
hill-valley	12.80%	9.20%	21.05%
mushroom	63.24%	71.93%	56.42%
pima	35.14%	30.23%	41.94%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	96.00%	100.00%	92.31%
Média	**66.73%**	**66.75%**	**69.18%**

Fonte: Elaborada pelo autor (2022)

Tabela 43 – Resultados do Boost utilizando N=5 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	89.66%	83.87%	96.30%
column2C	100.00%	100.00%	100.00%
credit	52.17%	50.00%	54.55%
glass1	77.78%	77.78%	77.78%
heart-c	57.14%	44.44%	80.00%
hill-valley	12.80%	9.20%	21.05%
mushroom	63.24%	71.93%	56.42%
pima	35.14%	30.23%	41.94%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	96.00%	100.00%	92.31%
Média	**66.73%**	**66.75%**	**69.18%**

Fonte: Elaborada pelo autor (2022)

Tabela 44 – Resultados do Boost utilizando N=10 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	10.90%	5.78%	96.30%
column2C	100.00%	100.00%	100.00%
credit	22.07%	13.01%	72.73%
glass1	77.78%	77.78%	77.78%
heart-c	57.14%	44.44%	80.00%
hill-valley	12.80%	9.20%	21.05%
mushroom	68.64%	74.33%	63.76%
pima	35.14%	30.23%	41.94%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**56.52%**	**55.48%**	**71.99%**

Fonte: Elaborada pelo autor (2022)

Tabela 45 – Resultados do Boost utilizando N=10 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	10.90%	5.78%	96.30%
column2C	100.00%	100.00%	100.00%
credit	47.06%	41.38%	54.55%
glass1	77.78%	77.78%	77.78%
heart-c	57.14%	44.44%	80.00%
hill-valley	12.80%	9.20%	21.05%
mushroom	63.24%	71.93%	56.42%
pima	35.14%	30.23%	41.94%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**58.48%**	**58.07%**	**69.43%**

Fonte: Elaborada pelo autor (2022)

Tabela 46 – Resultados do Boost utilizando N=15 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	10.90%	5.78%	96.30%
column2C	100.00%	100.00%	100.00%
credit	21.77%	12.80%	72.73%
glass1	77.78%	77.78%	77.78%
heart-c	57.14%	44.44%	80.00%
hill-valley	13.45%	7.31%	84.21%
mushroom	67.78%	61.98%	74.77%
pima	35.14%	30.23%	41.94%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	98.70%	100.00%	97.44%
Média	**56.60%**	**54.03%**	**79.66%**

Fonte: Elaborada pelo autor (2022)

Tabela 47 – Resultados do Boost utilizando N=15 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	10.90%	5.78%	96.30%
column2C	100.00%	100.00%	100.00%
credit	18.82%	10.81%	72.73%
glass1	77.78%	77.78%	77.78%
heart-c	45.71%	32.00%	80.00%
hill-valley	12.80%	9.20%	21.05%
mushroom	55.85%	50.56%	62.39%
pima	38.46%	31.91%	48.39%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	98.70%	100.00%	97.44%
Média	**54.24%**	**51.80%**	**72.75%**

Fonte: Elaborada pelo autor (2022)

Tabela 48 – Resultados do SEQ utilizando N=5, K=5 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	84.38%	72.97%	100.00%
column2C	100.00%	100.00%	100.00%
credit	52.83%	45.16%	63.64%
glass1	40.00%	31.25%	55.56%
heart-c	44.44%	35.29%	60.00%
hill-valley	5.15%	3.21%	13.16%
mushroom	87.53%	81.18%	94.95%
pima	33.80%	30.00%	38.71%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	71.03%	55.88%	97.44%
Média	**59.06%**	**52.64%**	**69.49%**

Fonte: Elaborada pelo autor (2022)

Tabela 49 – Resultados do SEQ utilizando N=5, K=10 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	77.14%	62.79%	100.00%
column2C	83.87%	72.22%	100.00%
credit	41.67%	30.00%	68.18%
glass1	25.81%	18.18%	44.44%
heart-c	56.00%	46.67%	70.00%
hill-valley	2.86%	3.13%	2.63%
mushroom	85.83%	75.17%	100.00%
pima	27.16%	22.00%	35.48%
sonar	27.27%	20.00%	42.86%
tic-tac-toe	52.94%	37.11%	92.31%
Média	**48.05%**	**38.73%**	**65.59%**

Fonte: Elaborada pelo autor (2022)

Tabela 50 – Resultados do SEQ utilizando N=5, K=5 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	84.38%	72.97%	100.00%
column2C	100.00%	100.00%	100.00%
credit	52.83%	45.16%	63.64%
glass1	40.00%	31.25%	55.56%
heart-c	44.44%	35.29%	60.00%
hill-valley	5.15%	3.21%	13.16%
mushroom	86.81%	80.95%	93.58%
pima	33.80%	30.00%	38.71%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	71.03%	55.88%	97.44%
Média	**58.99%**	**52.61%**	**69.35%**

Fonte: Elaborada pelo autor (2022)

Tabela 51 – Resultados do SEQ utilizando N=5, K=10 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	77.14%	62.79%	100.00%
column2C	83.87%	72.22%	100.00%
credit	41.67%	30.00%	68.18%
glass1	25.81%	18.18%	44.44%
heart-c	56.00%	46.67%	70.00%
hill-valley	2.86%	3.13%	2.63%
mushroom	85.38%	75.00%	99.08%
pima	27.16%	22.00%	35.48%
sonar	27.27%	20.00%	42.86%
tic-tac-toe	52.94%	37.11%	92.31%
Média	**48.01%**	**38.71%**	**65.50%**

Fonte: Elaborada pelo autor (2022)

Tabela 52 – Resultados do SEQ utilizando N=10, K=5 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	80.60%	67.50%	100.00%
column2C	100.00%	100.00%	100.00%
credit	13.93%	7.82%	63.64%
glass1	0.00%	0.00%	0.00%
heart-c	44.44%	35.29%	60.00%
hill-valley	5.15%	3.21%	13.16%
mushroom	88.00%	81.32%	95.87%
pima	33.80%	30.00%	38.71%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	71.03%	55.88%	97.44%
Média	**50.84%**	**45.25%**	**64.02%**

Fonte: Elaborada pelo autor (2022)

Tabela 53 – Resultados do SEQ utilizando N=10, K=10 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	70.13%	54.00%	100.00%
column2C	83.87%	72.22%	100.00%
credit	8.82%	4.69%	72.73%
glass1	25.81%	18.18%	44.44%
heart-c	56.00%	46.67%	70.00%
hill-valley	2.86%	3.13%	2.63%
mushroom	85.83%	75.17%	100.00%
pima	27.16%	22.00%	35.48%
sonar	27.27%	20.00%	42.86%
tic-tac-toe	52.94%	37.11%	92.31%
Média	**44.07%**	**35.32%**	**66.05%**

Fonte: Elaborada pelo autor (2022)

Tabela 54 – Resultados do SEQ utilizando N=10, K=5 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	11.25%	5.96%	100.00%
column2C	100.00%	100.00%	100.00%
credit	9.66%	5.22%	63.64%
glass1	11.54%	6.12%	100.00%
heart-c	44.44%	35.29%	60.00%
hill-valley	5.15%	3.21%	13.16%
mushroom	86.81%	80.95%	93.58%
pima	33.80%	30.00%	38.71%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	71.03%	55.88%	97.44%
Média	**44.51%**	**39.41%**	**73.79%**

Fonte: Elaborada pelo autor (2022)

Tabela 55 – Resultados do SEQ utilizando N=10, K=10 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	68.35%	51.92%	100.00%
column2C	83.87%	72.22%	100.00%
credit	8.82%	4.69%	72.73%
glass1	25.81%	18.18%	44.44%
heart-c	56.00%	46.67%	70.00%
hill-valley	2.86%	3.13%	2.63%
mushroom	85.83%	75.17%	100.00%
pima	29.27%	23.53%	38.71%
sonar	27.27%	20.00%	42.86%
tic-tac-toe	52.94%	37.11%	92.31%
Média	**44.10%**	**35.26%**	**66.37%**

Fonte: Elaborada pelo autor (2022)

Tabela 56 – Resultados do SEQ utilizando N=15, K=5 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	80.60%	67.50%	100.00%
column2C	100.00%	100.00%	100.00%
credit	13.93%	7.82%	63.64%
glass1	0.00%	0.00%	0.00%
heart-c	44.44%	35.29%	60.00%
hill-valley	5.15%	3.21%	13.16%
mushroom	88.00%	81.32%	95.87%
pima	33.80%	30.00%	38.71%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	71.03%	55.88%	97.44%
Média	**50.84%**	**45.25%**	**64.02%**

Fonte: Elaborada pelo autor (2022)

Tabela 57 – Resultados do SEQ utilizando N=15, K=10 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	68.35%	51.92%	100.00%
column2C	83.87%	72.22%	100.00%
credit	8.82%	4.69%	72.73%
glass1	25.81%	18.18%	44.44%
heart-c	56.00%	46.67%	70.00%
hill-valley	11.05%	5.85%	100.00%
mushroom	11.21%	5.94%	100.00%
pima	27.16%	22.00%	35.48%
sonar	27.27%	20.00%	42.86%
tic-tac-toe	52.94%	37.11%	92.31%
Média	**37.25%**	**28.46%**	**75.78%**

Fonte: Elaborada pelo autor (2022)

Tabela 58 – Resultados do SEQ utilizando N=15, K=5 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	11.25%	5.96%	100.00%
column2C	100.00%	100.00%	100.00%
credit	9.66%	5.22%	63.64%
glass1	0.00%	0.00%	0.00%
heart-c	44.44%	35.29%	60.00%
hill-valley	5.15%	3.21%	13.16%
mushroom	86.81%	80.95%	93.58%
pima	33.80%	30.00%	38.71%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	71.03%	55.88%	97.44%
Média	**43.36%**	**38.79%**	**63.79%**

Fonte: Elaborada pelo autor (2022)

Tabela 59 – Resultados do SEQ utilizando N=15, K=10 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	11.13%	5.90%	100.00%
column2C	83.87%	72.22%	100.00%
credit	8.82%	4.69%	72.73%
glass1	25.81%	18.18%	44.44%
heart-c	56.00%	46.67%	70.00%
hill-valley	11.06%	5.86%	100.00%
mushroom	85.83%	75.17%	100.00%
pima	29.27%	23.53%	38.71%
sonar	27.27%	20.00%	42.86%
tic-tac-toe	52.94%	37.11%	92.31%
Média	**39.20%**	**30.93%**	**76.10%**

Fonte: Elaborada pelo autor (2022)

Tabela 60 – Resultados do F3R utilizando N=5, K=5 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	76.06%	61.36%	100.00%
column2C	100.00%	100.00%	100.00%
credit	37.50%	24.32%	81.82%
glass1	84.21%	80.00%	88.89%
heart-c	62.50%	45.45%	100.00%
hill-valley	12.80%	9.20%	21.05%
mushroom	68.30%	73.54%	63.76%
pima	34.09%	26.32%	48.39%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**65.62%**	**62.02%**	**77.02%**

Fonte: Elaborada pelo autor (2022)

Tabela 61 – Resultados do F3R utilizando N=5, K=10 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	76.06%	61.36%	100.00%
column2C	100.00%	100.00%	100.00%
credit	37.62%	24.05%	86.36%
glass1	76.19%	66.67%	88.89%
heart-c	55.56%	38.46%	100.00%
hill-valley	12.80%	9.20%	21.05%
mushroom	68.30%	73.54%	63.76%
pima	32.18%	25.00%	45.16%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**62.75%**	**56.97%**	**77.15%**

Fonte: Elaborada pelo autor (2022)

Tabela 62 – Resultados do F3R utilizando N=5, K=5 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	76.06%	61.36%	100.00%
column2C	100.00%	100.00%	100.00%
credit	47.27%	39.39%	59.09%
glass1	84.21%	80.00%	88.89%
heart-c	62.50%	45.45%	100.00%
hill-valley	12.80%	9.20%	21.05%
mushroom	68.64%	74.33%	63.76%
pima	34.67%	29.55%	41.94%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**66.68%**	**63.93%**	**74.10%**

Fonte: Elaborada pelo autor (2022)

Tabela 63 – Resultados do F3R utilizando N=5, K=10 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	76.06%	61.36%	100.00%
column2C	100.00%	100.00%	100.00%
credit	44.78%	33.33%	68.18%
glass1	76.19%	66.67%	88.89%
heart-c	58.82%	41.67%	100.00%
hill-valley	12.80%	9.20%	21.05%
mushroom	68.64%	74.33%	63.76%
pima	35.14%	30.23%	41.94%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**64.12%**	**58.82%**	**75.01%**

Fonte: Elaborada pelo autor (2022)

Tabela 64 – Resultados do F3R utilizando N=10, K=5 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	66.67%	50.00%	100.00%
column2C	100.00%	100.00%	100.00%
credit	36.36%	22.73%	90.91%
glass1	84.21%	80.00%	88.89%
heart-c	62.50%	45.45%	100.00%
hill-valley	9.90%	6.10%	26.32%
mushroom	69.34%	61.57%	79.36%
pima	35.56%	27.12%	51.61%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**64.52%**	**59.30%**	**80.34%**

Fonte: Elaborada pelo autor (2022)

Tabela 65 – Resultados do F3R utilizando N=10, K=10 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	66.67%	50.00%	100.00%
column2C	100.00%	100.00%	100.00%
credit	35.71%	22.22%	90.91%
glass1	76.19%	66.67%	88.89%
heart-c	45.45%	29.41%	100.00%
hill-valley	10.10%	6.25%	26.32%
mushroom	69.34%	61.57%	79.36%
pima	35.16%	26.67%	51.61%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**60.74%**	**53.42%**	**80.34%**

Fonte: Elaborada pelo autor (2022)

Tabela 66 – Resultados do F3R utilizando N=10, K=5 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	66.67%	50.00%	100.00%
column2C	100.00%	100.00%	100.00%
credit	35.51%	22.35%	86.36%
glass1	84.21%	80.00%	88.89%
heart-c	62.50%	45.45%	100.00%
hill-valley	10.93%	6.90%	26.32%
mushroom	69.62%	62.01%	79.36%
pima	35.56%	27.12%	51.61%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**64.57%**	**59.38%**	**79.88%**

Fonte: Elaborada pelo autor (2022)

Tabela 67 – Resultados do F3R utilizando N=10, K=10 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	66.67%	50.00%	100.00%
column2C	100.00%	100.00%	100.00%
credit	34.23%	21.35%	86.36%
glass1	76.19%	66.67%	88.89%
heart-c	47.62%	31.25%	100.00%
hill-valley	11.17%	7.09%	26.32%
mushroom	69.62%	62.01%	79.36%
pima	35.56%	27.12%	51.61%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**60.99%**	**53.69%**	**79.88%**

Fonte: Elaborada pelo autor (2022)

Tabela 68 – Resultados do F3R utilizando N=15, K=5 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	66.67%	50.00%	100.00%
column2C	100.00%	100.00%	100.00%
credit	36.36%	22.73%	90.91%
glass1	84.21%	80.00%	88.89%
heart-c	62.50%	45.45%	100.00%
hill-valley	9.22%	5.59%	26.32%
mushroom	67.98%	59.45%	79.36%
pima	35.56%	27.12%	51.61%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**64.32%**	**59.03%**	**80.34%**

Fonte: Elaborada pelo autor (2022)

Tabela 69 – Resultados do F3R utilizando N=15, K=10 e D=5

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	66.67%	50.00%	100.00%
column2C	100.00%	100.00%	100.00%
credit	35.71%	22.22%	90.91%
glass1	76.19%	66.67%	88.89%
heart-c	31.75%	18.87%	100.00%
hill-valley	9.43%	5.75%	26.32%
mushroom	67.98%	59.45%	79.36%
pima	35.16%	26.67%	51.61%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**59.17%**	**52.10%**	**80.34%**

Fonte: Elaborada pelo autor (2022)

Tabela 70 – Resultados do F3R utilizando N=15, K=5 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	66.67%	50.00%	100.00%
column2C	100.00%	100.00%	100.00%
credit	34.55%	21.59%	86.36%
glass1	84.21%	80.00%	88.89%
heart-c	62.50%	45.45%	100.00%
hill-valley	9.30%	5.65%	26.32%
mushroom	68.24%	59.86%	79.36%
pima	35.56%	27.12%	51.61%
sonar	83.33%	100.00%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**64.17%**	**58.97%**	**79.88%**

Fonte: Elaborada pelo autor (2022)

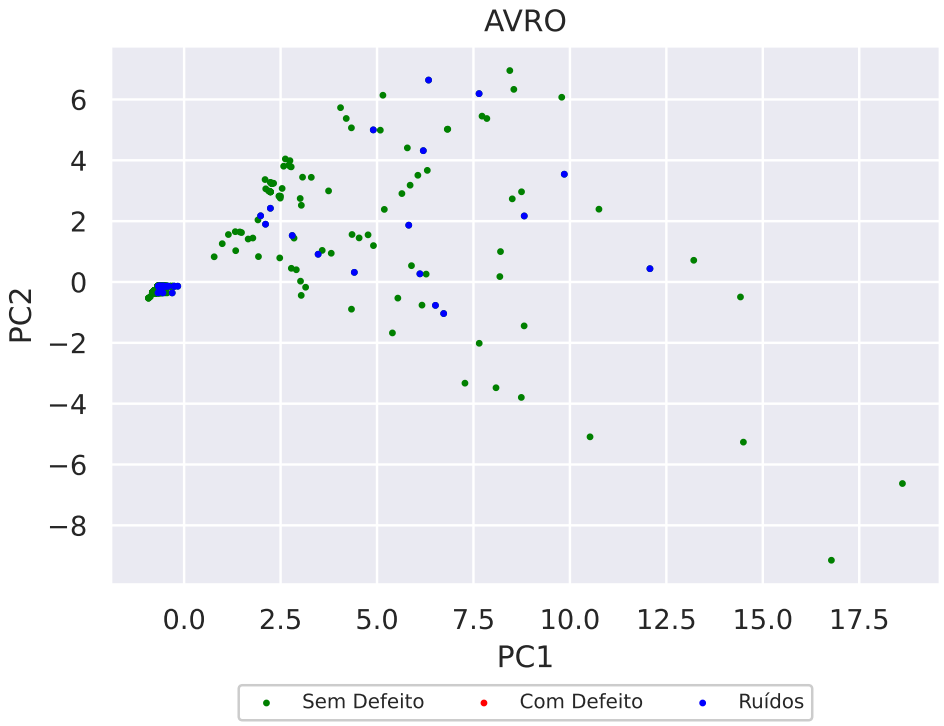
Tabela 71 – Resultados do F3R utilizando N=15, K=10 e D=10

Conjunto de Dados	F1 Score	Precisão	Cobertura
breast-c-w	66.67%	50.00%	100.00%
column2C	100.00%	100.00%	100.00%
credit	35.09%	21.74%	90.91%
glass1	76.19%	66.67%	88.89%
heart-c	33.90%	20.41%	100.00%
hill-valley	9.52%	5.81%	26.32%
mushroom	68.24%	59.86%	79.36%
pima	35.16%	26.67%	51.61%
sonar	71.43%	71.43%	71.43%
tic-tac-toe	97.37%	100.00%	94.87%
Média	**59.36%**	**52.26%**	**80.34%**

Fonte: Elaborada pelo autor (2022)

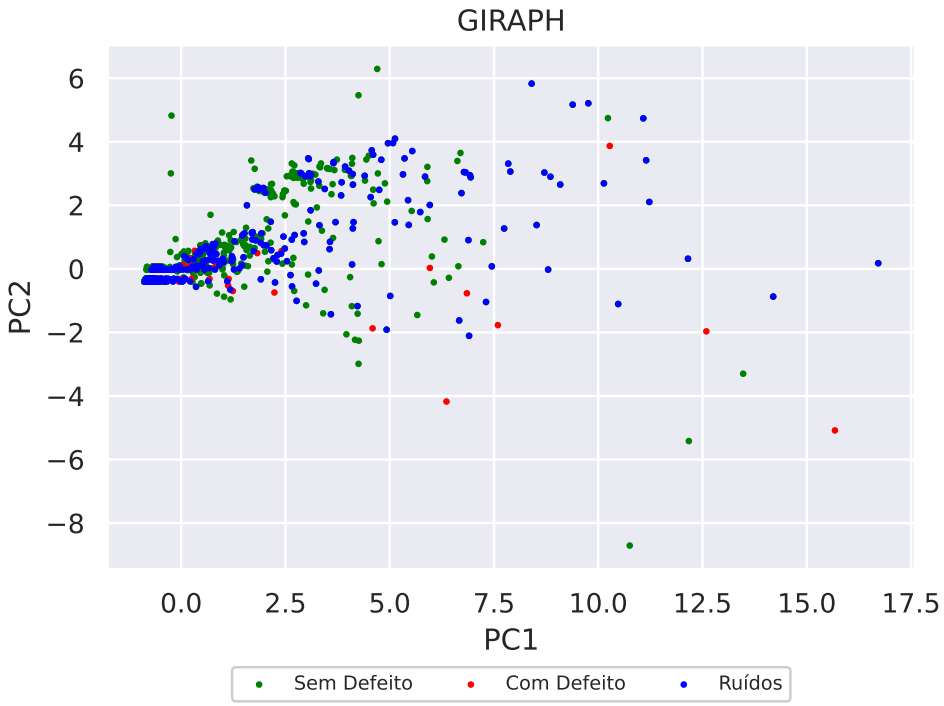
APÊNDICE C – CONJUNTO DE DADOS ESTUDO DE CASO 1

Figura 14 – Distribuição de dados do Conjunto de dados AVRO



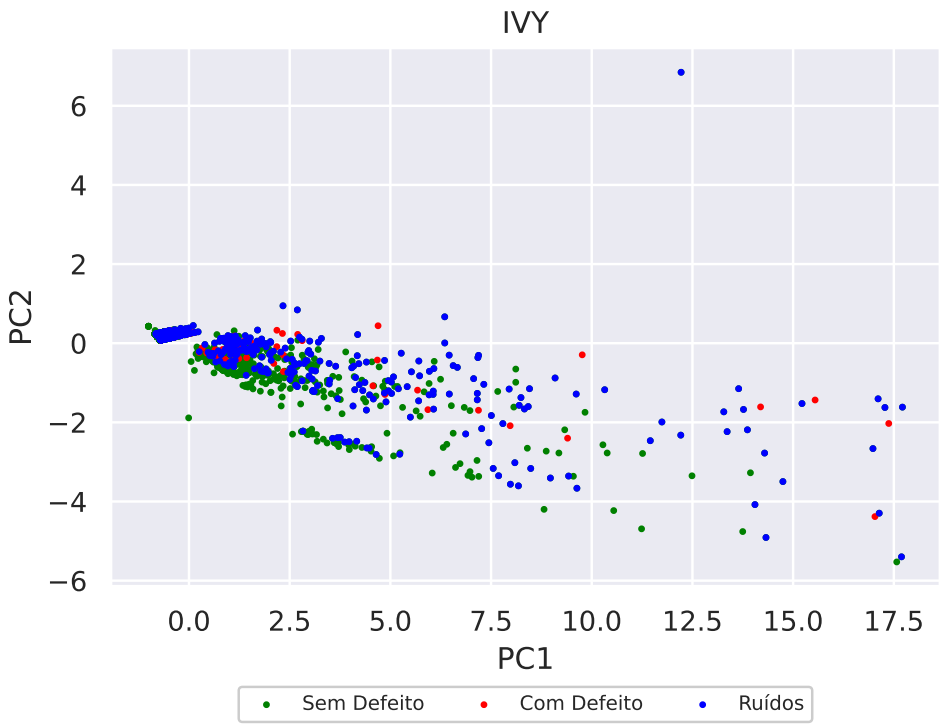
Fonte: Elaborada pelo autor (2022)

Figura 15 – Distribuição de dados do Conjunto de dados GIRAPH



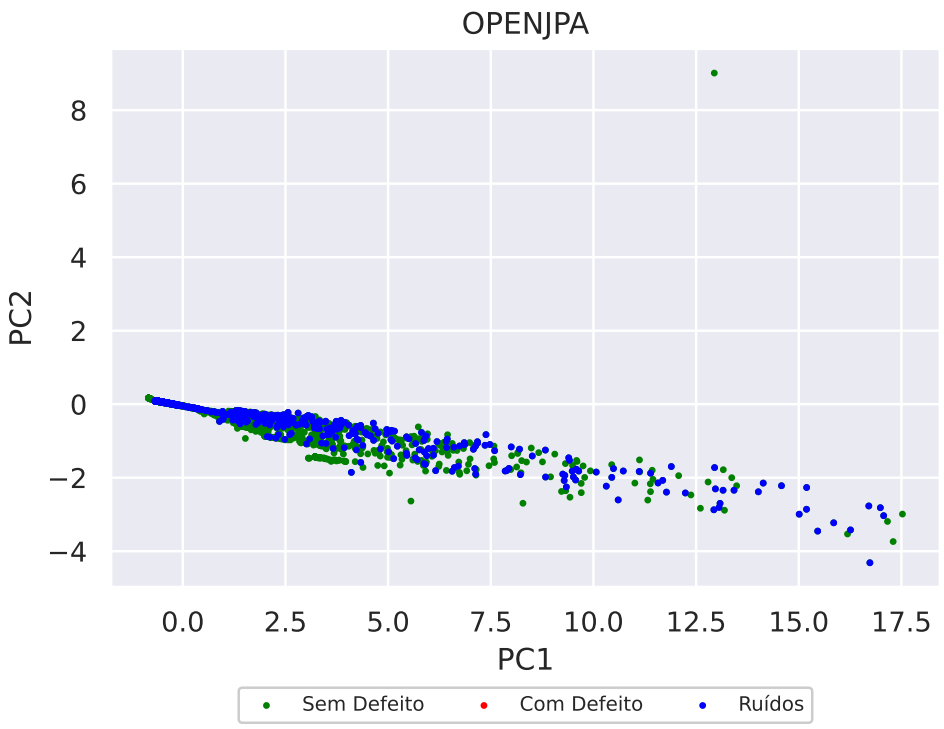
Fonte: Elaborada pelo autor (2022)

Figura 16 – Distribuição de dados do Conjunto de dados IVY



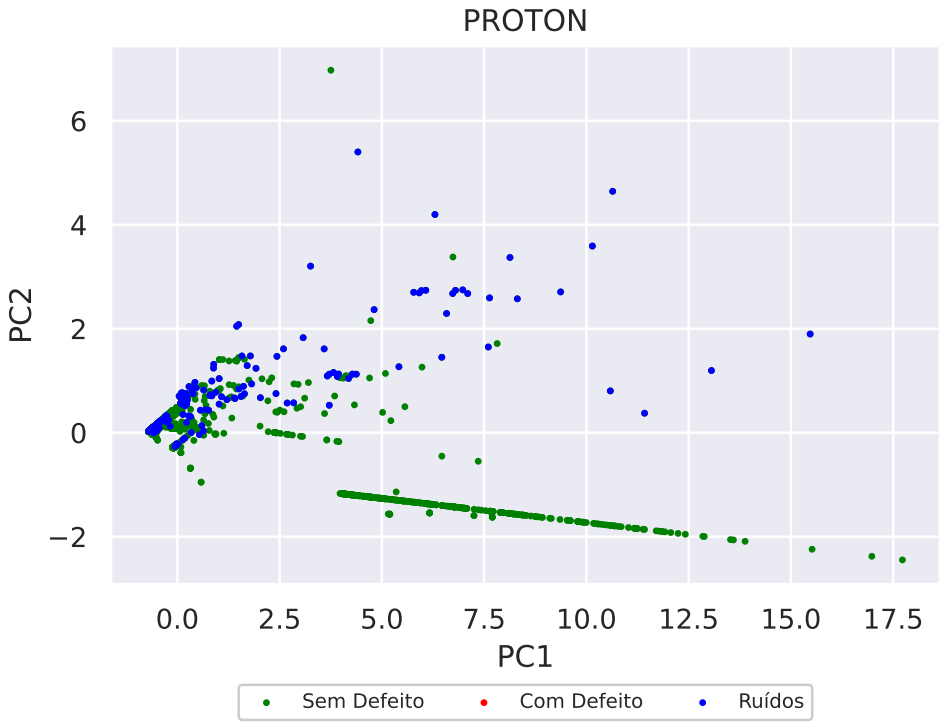
Fonte: Elaborada pelo autor (2022)

Figura 17 – Distribuição de dados do Conjunto de dados OPENJPA



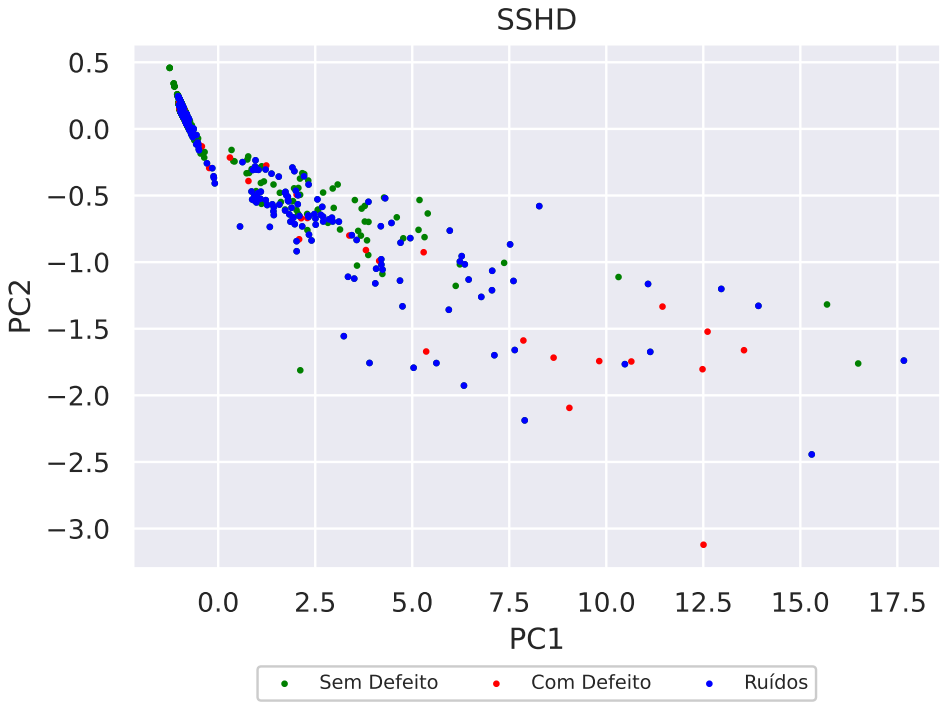
Fonte: Elaborada pelo autor (2022)

Figura 18 – Distribuição de dados do Conjunto de dados PROTON



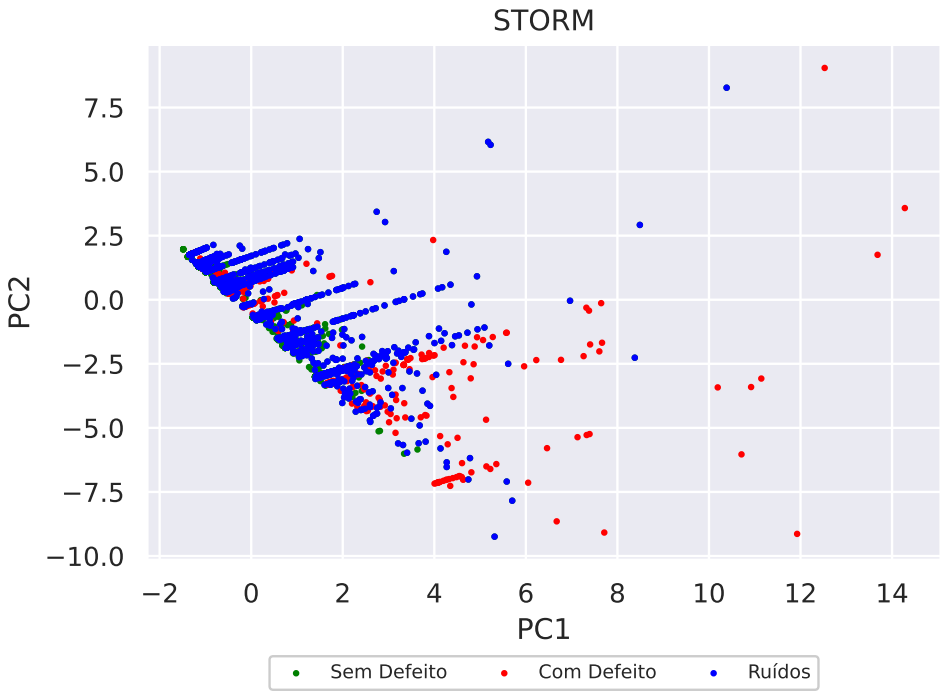
Fonte: Elaborada pelo autor (2022)

Figura 19 – Distribuição de dados do Conjunto de dados SSHD



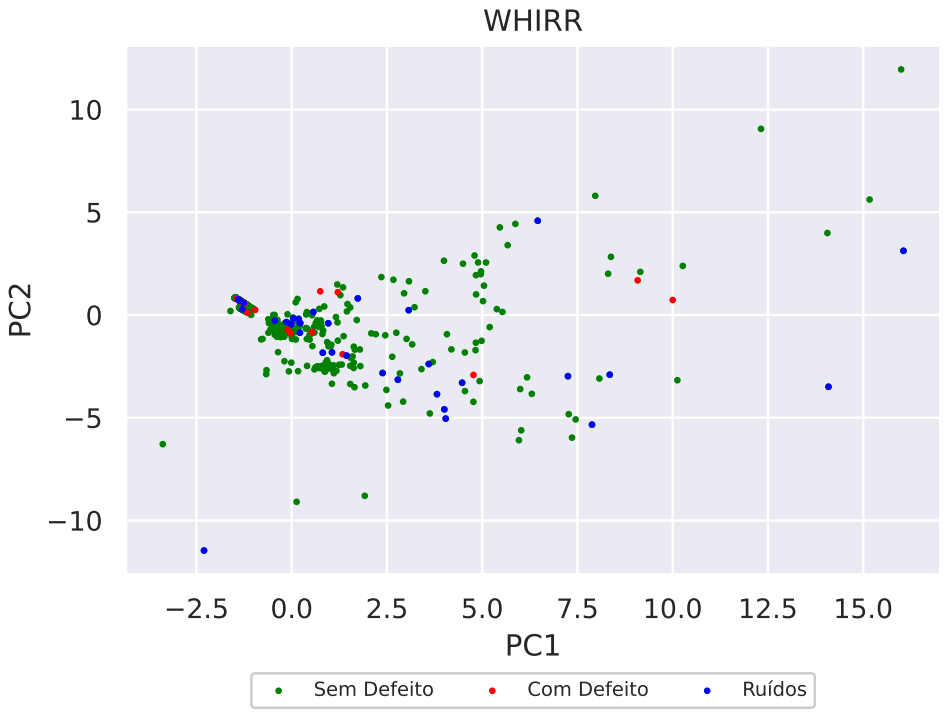
Fonte: Elaborada pelo autor (2022)

Figura 20 – Distribuição de dados do Conjunto de dados STORM



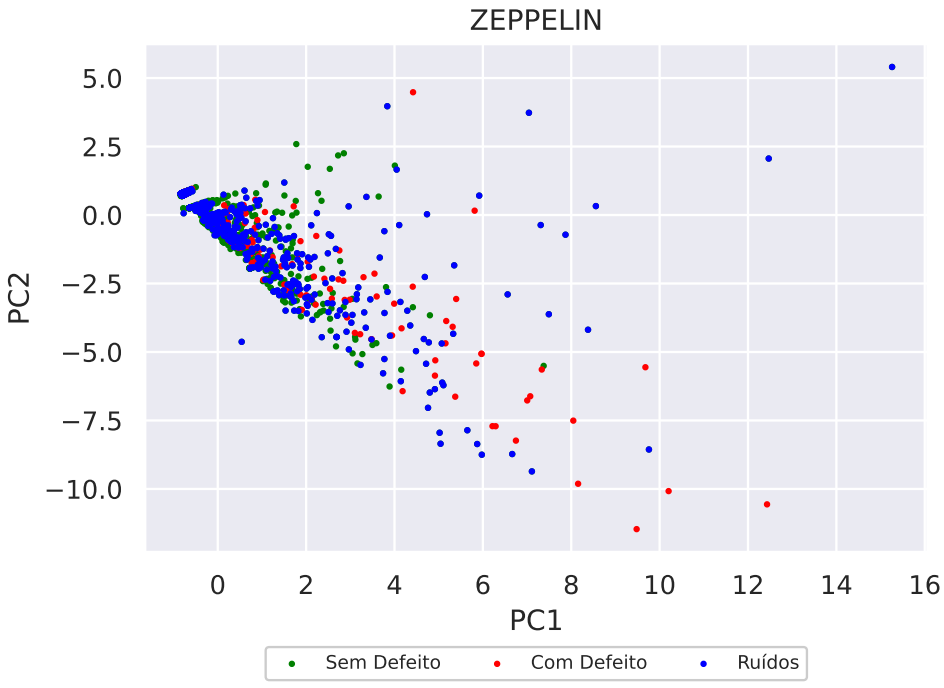
Fonte: Elaborada pelo autor (2022)

Figura 21 – Distribuição de dados do Conjunto de dados WHIRR



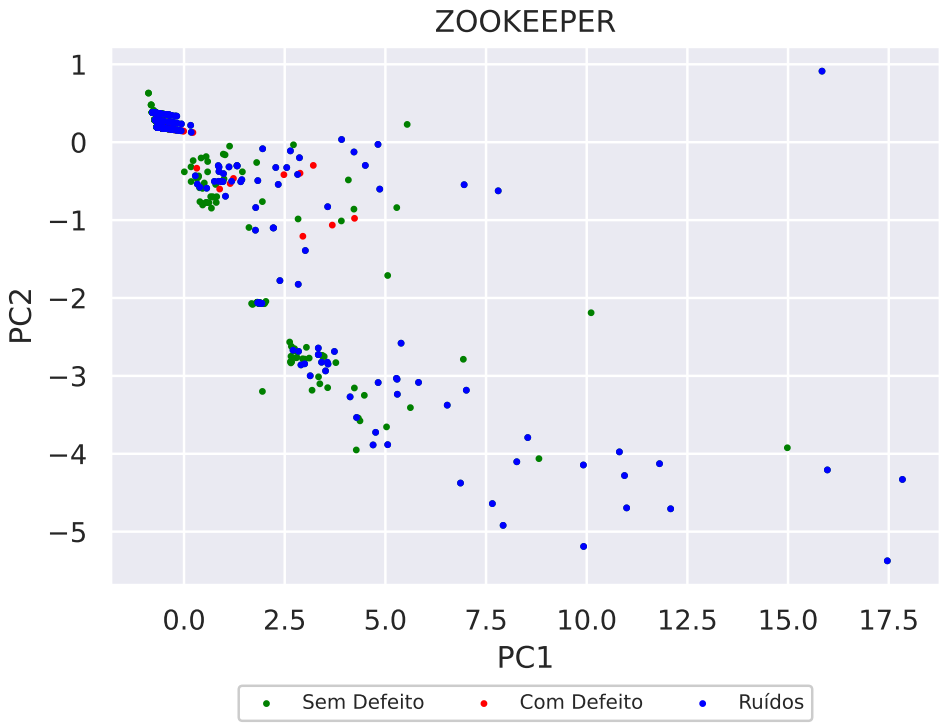
Fonte: Elaborada pelo autor (2022)

Figura 22 – Distribuição de dados do Conjunto de dados ZEPPELIN



Fonte: Elaborada pelo autor (2022)

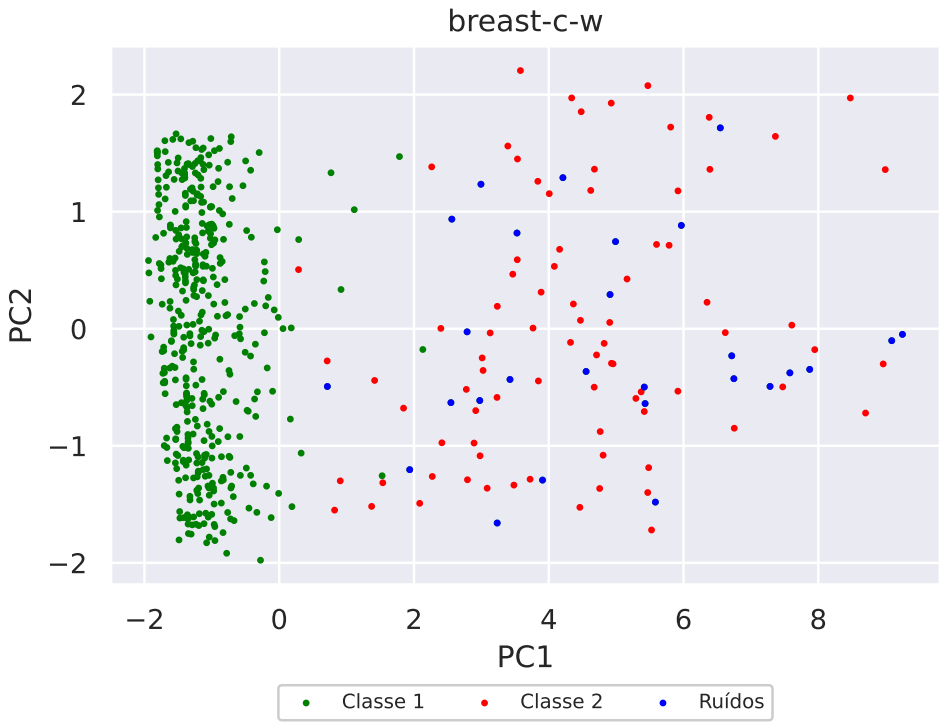
Figura 23 – Distribuição de dados do Conjunto de dados ZOOKEEPER



Fonte: Elaborada pelo autor (2022)

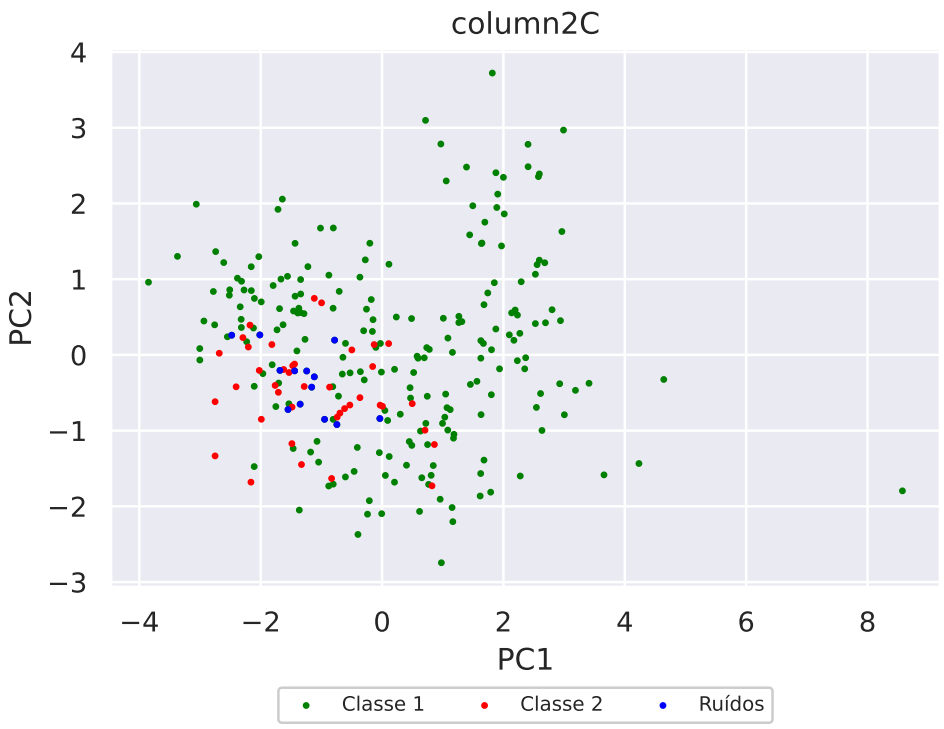
APÊNDICE D – CONJUNTO DE DADOS ESTUDO DE CASO 2

Figura 24 – Distribuição de dados do Conjunto de dados breast-c-w



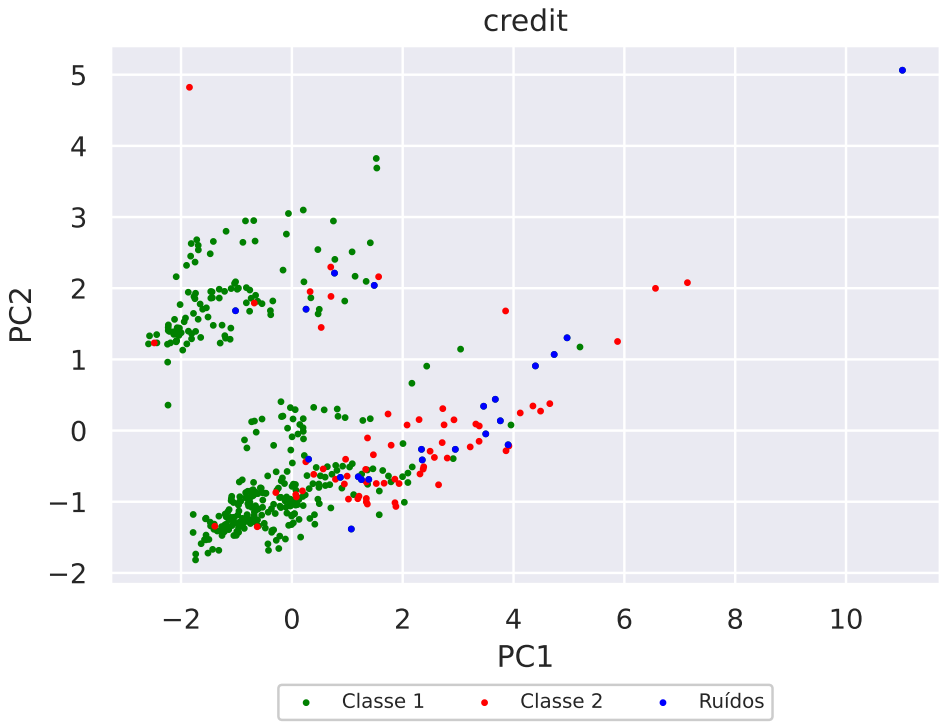
Fonte: Elaborada pelo autor (2022)

Figura 25 – Distribuição de dados do Conjunto de dados column2C



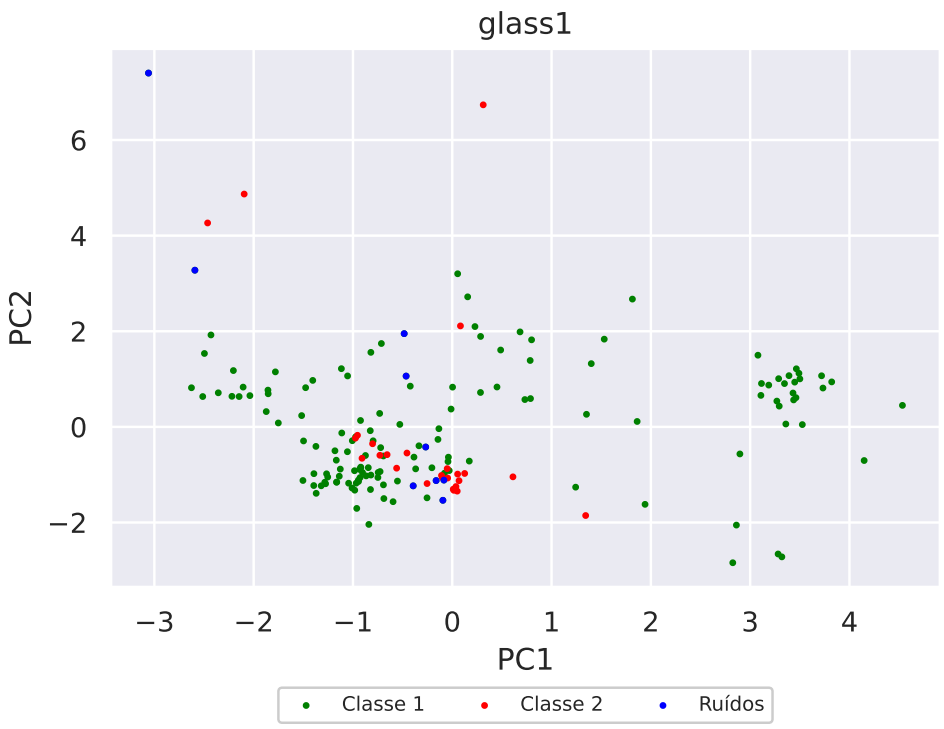
Fonte: Elaborada pelo autor (2022)

Figura 26 – Distribuição de dados do Conjunto de dados credit



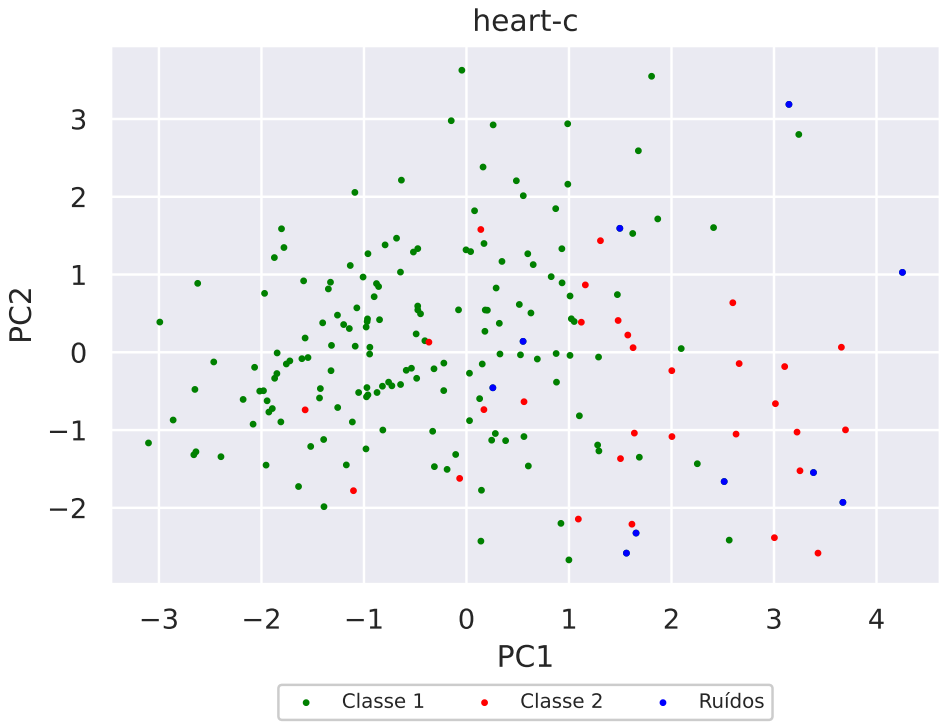
Fonte: Elaborada pelo autor (2022)

Figura 27 – Distribuição de dados do Conjunto de dados glass1



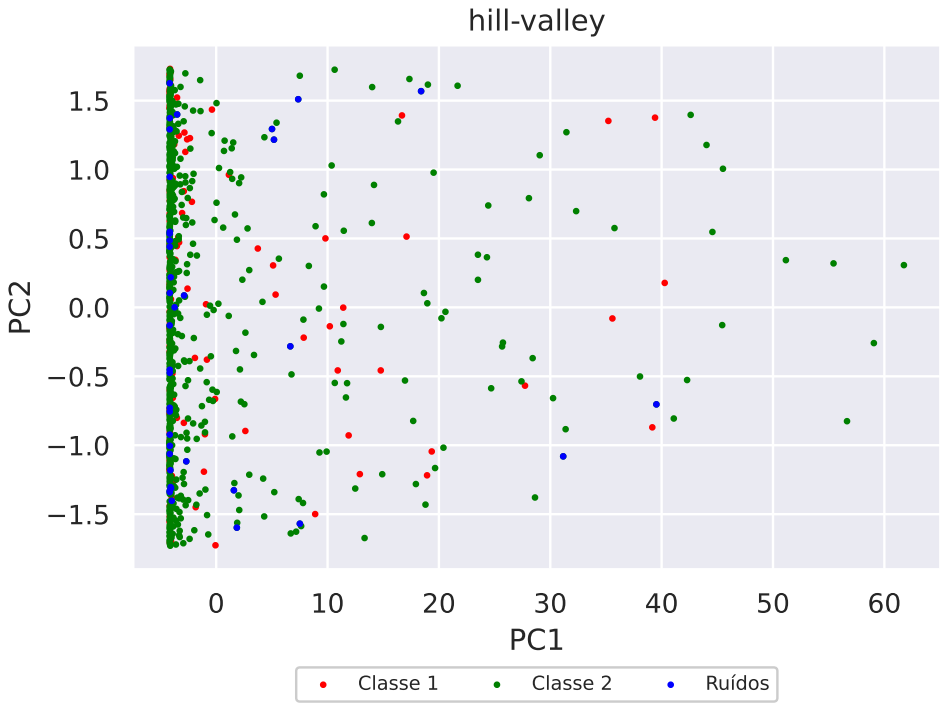
Fonte: Elaborada pelo autor (2022)

Figura 28 – Distribuição de dados do Conjunto de dados heart-c



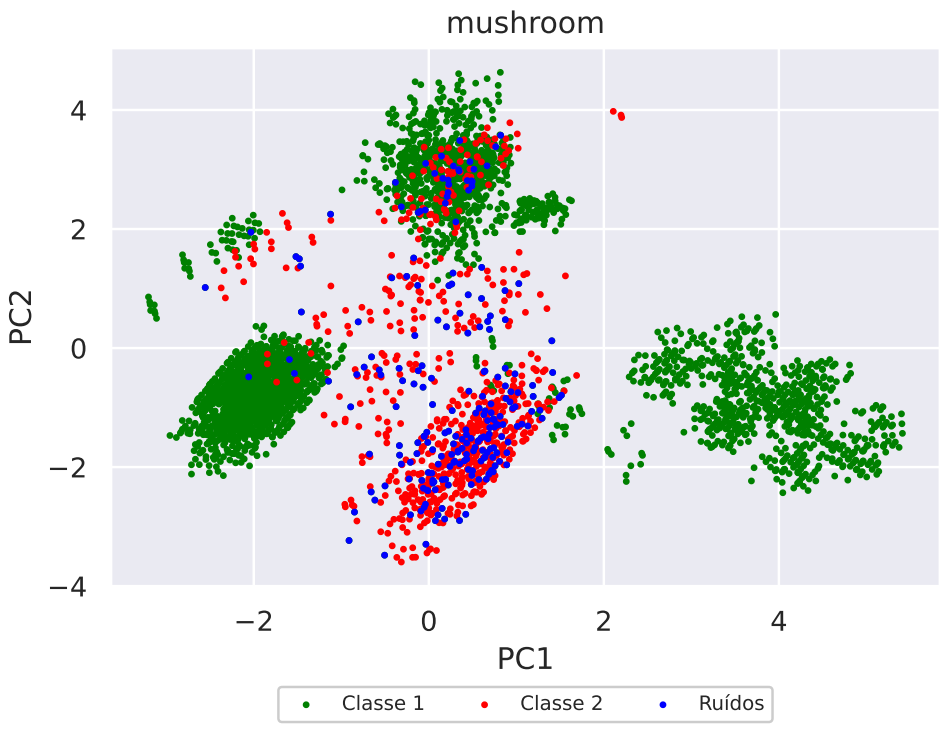
Fonte: Elaborada pelo autor (2022)

Figura 29 – Distribuição de dados do Conjunto de dados hill-valley



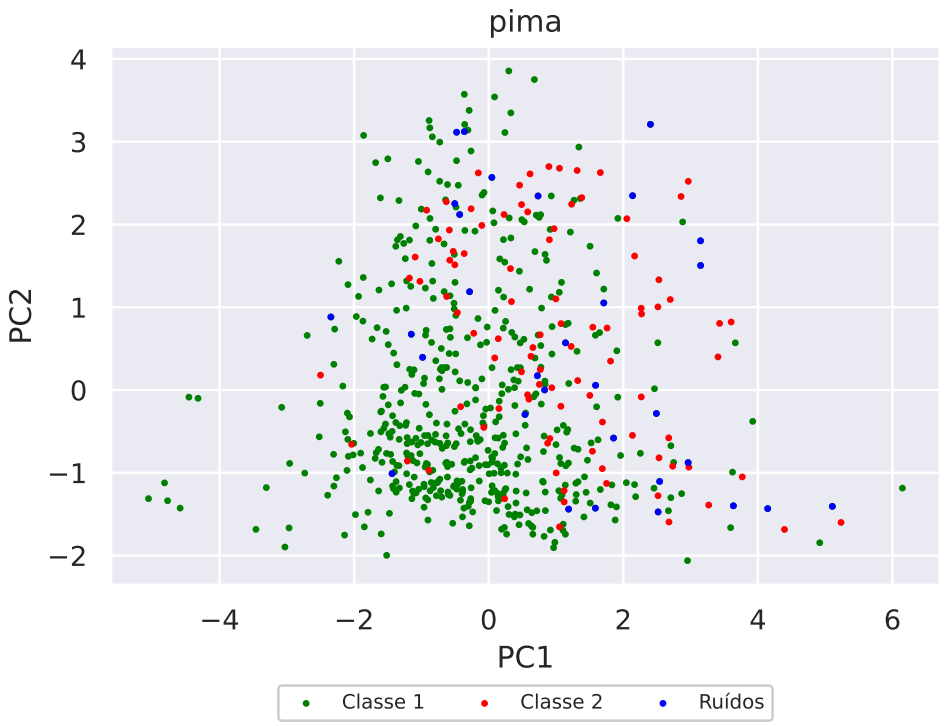
Fonte: Elaborada pelo autor (2022)

Figura 30 – Distribuição de dados do Conjunto de dados mushroom



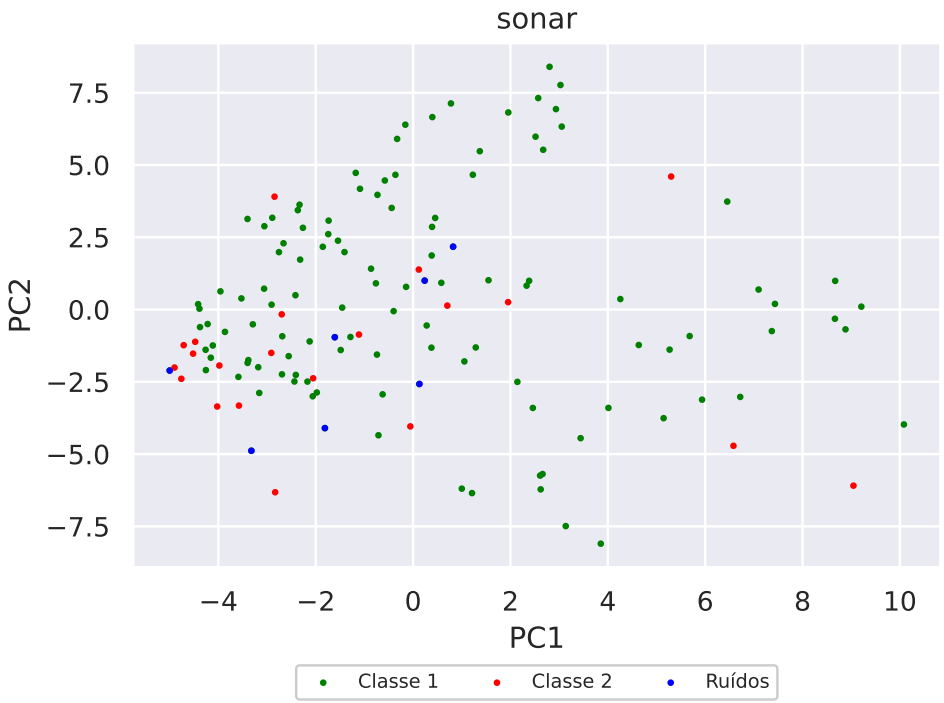
Fonte: Elaborada pelo autor (2022)

Figura 31 – Distribuição de dados do Conjunto de dados pima



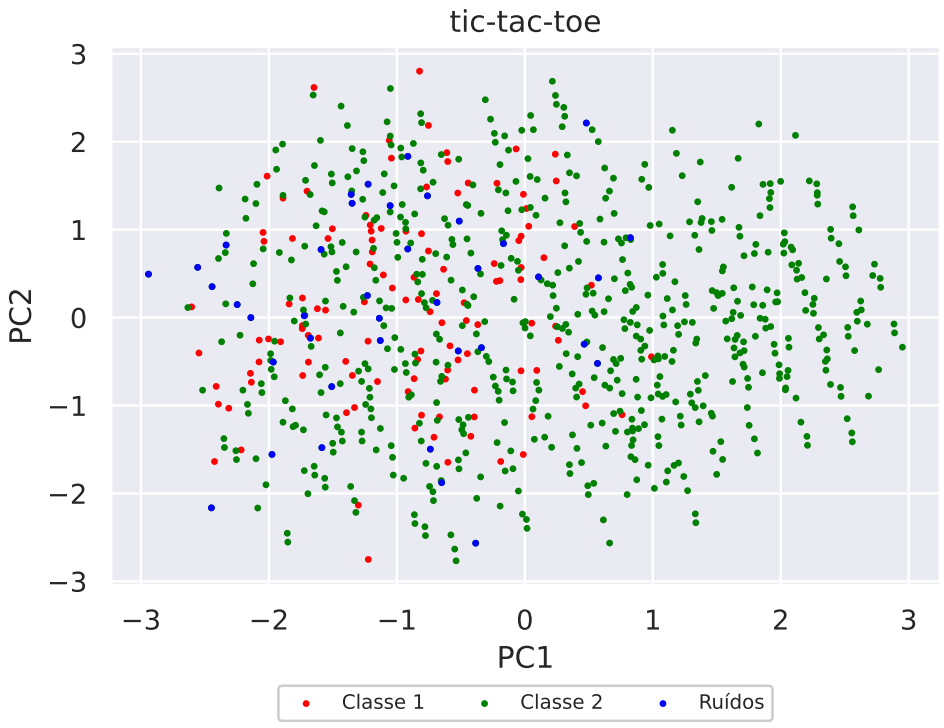
Fonte: Elaborada pelo autor (2022)

Figura 32 – Distribuição de dados do Conjunto de dados sonar



Fonte: Elaborada pelo autor (2022)

Figura 33 – Distribuição de dados do Conjunto de dados tic-tac-toe



Fonte: Elaborada pelo autor (2022)