MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

FÁBIO DO ROSARIO SANTOS

UMA ABORDAGEM PARA DETECÇÃO E AVALIAÇÃO DA GRAVIDADE DE *CODE SMELLS* COM BASE EM TRANSFERÊNCIA DE APRENDIZADO

RIO DE JANEIRO
2024

FÁBIO DO ROSARIO SANTOS

UMA ABORDAGEM PARA DETECÇÃO E AVALIAÇÃO DA GRAVIDADE DE *CODE SMELLS* COM BASE EM TRANSFERÊNCIA DE APRENDIZADO

Dissertação apresentada ao Programa de Pós-graduação em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador(es): Ricardo Choren Noya, D.Sc.

Rio de Janeiro

2024

©2024
INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 Praia Vermelha
Rio de Janeiro RJ CEP: 22290-270

Este exemplar é de propriedade do Instituto Militar de Engenharia, que poderá incluí-lo em base de dados, armazenar em computador, microfilmar ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es) e do(s) orientador(es).

Santos, Fábio do Rosario.
Uma abordagem para detecção e avaliação da gravidade de *code smells* com base em Transferência de Aprendizado / Fábio do Rosario Santos. – Rio de Janeiro, 2024.
197 f.

Orientador(es): Ricardo Choren Noya.

Dissertação (mestrado) – Instituto Militar de Engenharia, Sistemas e Computação, 2024.

1. *code smells*. 2. transferência de aprendizado. 3. métodos de comitê. 4. pré-processamento de dados. 5. otimização de hiperparâmetros. i. Choren Noya, Ricardo (orient.) ii. Título

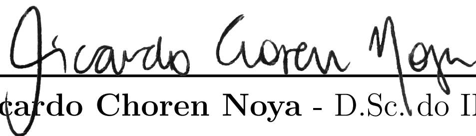
FÁBIO DO ROSARIO SANTOS

Uma abordagem para detecção e avaliação da gravidade de *code smells* com base em Transferência de Aprendizado

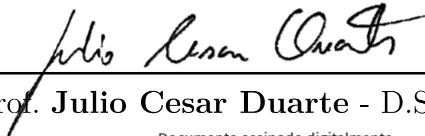
Dissertação apresentada ao Programa de Pós-graduação em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador(es): Ricardo Choren Noya.

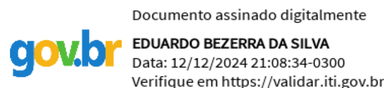
Aprovada em 03 de dezembro de 2024, pela seguinte banca examinadora:



Prof. **Ricardo Choren Noya** - D.Sc. do IME - Presidente



Prof. **Julio Cesar Duarte** - D.Sc. do IME



Prof. **Eduardo Bezerra da Silva** - D.Sc. do CEFET/RJ

Rio de Janeiro

2024

Este trabalho é dedicado a Deus, à minha esposa,
e aos meus pais que sempre me apoiaram neste desafio.

AGRADECIMENTOS

Os agradecimentos principais são direcionados a Deus, por me conceder saúde e as capacidades necessárias para alcançar este objetivo. Agradeço profundamente à minha esposa, Daniella Ribeiro Ferreira Santos, pelo suporte contínuo, mesmo nos momentos mais difíceis. E, aos meus queridos pais, Melquiades Pereira dos Santos e Maria Helena do Rosario Santos, por tudo.

Agradecimentos especiais são direcionados ao Escritório de Projetos e Sistemas Digitais do CFN, nas pessoas dos meus comandantes e chefes imediatos (Capitão de Mar e Guerra Terra, Capitão de Fragata Soransso e Capitão de Corveta Flávio Matias), pelo empenho na concretização dessa oportunidade ímpar de me dedicar a este curso, e aos meus professores do IME, em especial ao meu orientador, Professor Ricardo Choren Noya, pela transferência de conhecimento, paciência e orientação despendidas.

“Hoje vemos como por um espelho, confusamente;
mas então veremos face a face.
Hoje conheço em parte;
mas então conhecerei totalmente,
como eu sou conhecido”
(Bíblia Sagrada, 1 Coríntios 13, 12)

RESUMO

A detecção de *code smells* e a avaliação de gravidade são importantes para categorizar e priorizar esforços de manutenção de software. Nesse sentido, há uma pesquisa considerável com foco em modelos de Aprendizado Profundo e Transformadores para detecção de *code smells*. Este trabalho visa não apenas detectar, mas também fazer uma avaliação da gravidade de *code smells* usando uma abordagem de dois estágios empregando Métodos de Comitê e Transferência de Aprendizado. Este trabalho também explora o impacto da aplicação de dimensionamento de dados, técnicas de seleção de atributos, otimização de hiperparâmetros e sobreamostragem de dados para aprimorar os Métodos Comitê para detecção de*code smells* e avaliação de gravidade. Além disso, a abordagem proposta funciona nos dois níveis de *code smells* (classe e método) e é adequada para conjuntos de dados Java e C#. Este trabalho revela que a Transferência de Aprendizado melhorou a generalização dos modelos, com precisão de detecção no conjunto de dados C# correspondendo ou excedendo a do conjunto de dados Java com perda mínima de desempenho. Experimentos indicam que a abordagem proposta fornece resultados promissores para detecção de *code smells* e avaliação de gravidade em níveis de classe e método.

Palavras-chave: *code smells*. transferência de aprendizado. métodos de comitê. pré-processamento de dados. otimização de hiperparâmetros.

ABSTRACT

Code smell detection and severity assessment are important for categorizing and prioritizing software maintenance efforts. In this sense, there is considerable research focusing on Deep Learning and Transformer-based models for code smell detection. This work aims not only to detect but also to make a severity assessment of code smells using a two-stage approach employing Ensembles and Transfer Learning. This work also explores the impact of applying data scaling, feature selection techniques, hyperparameter optimization, and data oversampling to enhance the Ensembles for code smell detection and severity assessment. Moreover, the proposed approach works at the two code smell levels (class and method) and is suitable for both Java and C# datasets. This work reveals that Transfer Learning improved model generalization, with detection accuracy on the C# dataset matching or exceeding that of the Java dataset with minimal performance loss. Experiments indicate that the proposed approach delivers promising results for code smell detection and severity assessment at both class and method levels.

Keywords: code smells. transfer learning. ensemble methods. data preprocessing. hyperparameter optimization.

LISTA DE ILUSTRAÇÕES

Figura 1 – Visão geral do processo de combinação do conjunto de dados Java . . . 59
Figura 2 – Visão geral da abordagem proposta 63
Figura 3 – Pré-processamento de Dados 64
Figura 4 – Otimização de Hiperparâmetros 66
Figura 5 – Detecção de *Code Smell* 69
Figura 6 – Avaliação de Gravidade 70
Figura 7 – Matriz de Confusão da Tarefa de Detecção de *Code Smell* Java 78
Figura 8 – Matriz de Confusão da Tarefa de Avaliação de Gravidade Java 78
Figura 9 – Matriz de Confusão da Tarefa de Detecção de *Code Smell* C# 81
Figura 10 – Matriz de Confusão da Tarefa de Avaliação de Gravidade C# 82
Figura 11 – Produção Científica Anual 110
Figura 12 – Processo de seleção de estudos primários. 111
Figura 13 – Contribuição de cada fonte de pesquisa. 113
Figura 14 – Frequência de Seleção de Métricas para os Melhores Detectores de *Code Smell* 116
Figura 15 – Evolução da Acurácia - ANOVA 126
Figura 16 – Evolução da Acurácia - ANOVA com Atributos Normalizados 126
Figura 17 – Evolução da Média de Desempenho - ANOVA 127
Figura 18 – Evolução da Média de Desempenho - ANOVA com Atributos Normalizados 127
Figura 19 – Evolução da Desvio Padrão do Desempenho - ANOVA 128
Figura 20 – Evolução da Desvio Padrão do Desempenho - ANOVA com Atributos Normalizados 128
Figura 21 – Evolução do Tempo de Otimização de Hiperparâmetros - ANOVA . . . 129
Figura 22 – Evolução do Tempo de Otimização de Hiperparâmetros - ANOVA com Atributos Normalizados 129
Figura 23 – Evolução da Acurácia - Qui-Quadrado 131
Figura 24 – Evolução da Acurácia - Qui-Quadrado com Atributos Normalizados . . 131
Figura 25 – Evolução da Média de Desempenho - Qui-Quadrado 132
Figura 26 – Evolução da Média de Desempenho - Qui-Quadrado com Atributos Normalizados 132
Figura 27 – Evolução da Desvio Padrão do Desempenho - Qui-Quadrado 133
Figura 28 – Evolução da Desvio Padrão do Desempenho - Qui-Quadrado com Atributos Normalizados 133
Figura 29 – Evolução do Tempo de Otimização de Hiperparâmetros - Qui-Quadrado 134
Figura 30 – Evolução do Tempo de Otimização de Hiperparâmetros - Qui-Quadrado com Atributos Normalizados 134

Figura 31 – Diagrama de Classes do Analisador de Métricas de Código 135
Figura 32 – Diagrama das Principais Classes do Analisador de Métricas de Código 136
Figura 33 – Diagrama de Classes dos Analisadores de Métricas de Métodos 137
Figura 34 – Diagrama de Classes dos Analisadores de Métricas de Classes 138
Figura 35 – Importância dos Atributos para Detecção de FE Java 152
Figura 36 – Importância dos Atributos para Avaliação das Gravidades FE Java . . 153
Figura 37 – Impacto na saída do modelo de detecção de FE Java - SHAP 154
Figura 38 – Detalhamento de instância de FE não Grave Java com LIME 156
Figura 39 – Detalhamento de instância de Gravidade FE Java com LIME 157
Figura 40 – Detalhamento de instância de FE Grave Java com LIME 158
Figura 41 – Importância dos Atributos para Detecção de LM Java 159
Figura 42 – Importância dos Atributos para Avaliação das Gravidades LM Java . . 160
Figura 43 – Impacto na saída do modelo de detecção de LM Java - SHAP 161
Figura 44 – Detalhamento de instância de LM não Grave Java com LIME 162
Figura 45 – Detalhamento de instância de Gravidade LM Java com LIME 163
Figura 46 – Detalhamento de instância de LM Grave Java com LIME 164
Figura 47 – Importância dos Atributos para Detecção de DC Java 165
Figura 48 – Importância dos Atributos para Avaliação das Gravidades DC Java . . 166
Figura 49 – Impacto na saída do modelo de detecção de DC Java - SHAP 167
Figura 50 – Detalhamento de instância de DC não Grave Java com LIME 168
Figura 51 – Detalhamento de instância de Gravidade DC Java com LIME 169
Figura 52 – Detalhamento de instância de DC Grave Java com LIME 170
Figura 53 – Importância dos Atributos para Detecção de GC Java 171
Figura 54 – Importância dos Atributos para Avaliação das Gravidades GC Java . . 172
Figura 55 – Impacto na saída do modelo de detecção de GC Java - SHAP 173
Figura 56 – Detalhamento de instância de GC não Grave Java com LIME 174
Figura 57 – Detalhamento de instância de Gravidade GC Java com LIME 175
Figura 58 – Detalhamento de instância de GC Grave Java com LIME 176
Figura 59 – Importância dos Atributos para Detecção de FE C# 177
Figura 60 – Importância dos Atributos para Avaliação das Gravidades FE C# . . . 178
Figura 61 – Detalhamento de instância de FE não Grave C# com LIME 179
Figura 62 – Detalhamento de instância de Gravidade FE C# com LIME 180
Figura 63 – Importância dos Atributos para Detecção de LM C# 181
Figura 64 – Importância dos Atributos para Avaliação das Gravidades LM C# . . . 182
Figura 65 – Detalhamento de instância de LM não Grave C# com LIME 184
Figura 66 – Detalhamento de instância de Gravidade LM C# com LIME 185
Figura 67 – Detalhamento de instância de LM Grave C# com LIME 186
Figura 68 – Importância dos Atributos para Detecção de DC C# 187
Figura 69 – Importância dos Atributos para Avaliação das Gravidades DC C# . . . 188

Figura 70 – Impacto na saída do modelo de detecção de DC C# - SHAP 189
Figura 71 – Detalhamento de instância de DC não Grave C# com LIME 190
Figura 72 – Detalhamento de instância de Gravidade DC C# com LIME 191
Figura 73 – Importância dos Atributos para Detecção de GC C# 192
Figura 74 – Importância dos Atributos para Avaliação das Gravidades GC C# . . . 193
Figura 75 – Impacto na saída do modelo de detecção de GC C# - SHAP 194
Figura 76 – Detalhamento de instância de GC não Grave C# com LIME 195
Figura 77 – Detalhamento de instância de Gravidade GC C# com LIME 196
Figura 78 – Detalhamento de instância de GC Grave C# com LIME 197

LISTA DE TABELAS

Tabela 1 – Síntese do Estado da Arte para Detecção e Avaliação de Gravidade de *code smell* baseado em AM . . . 52
Tabela 2 – Ferramenta de detecção automática ("Conselheiros") . . . 57
Tabela 3 – Detalhes das instâncias corrigidas de GC e DC . . . 57
Tabela 4 – Detalhamento das classes geradas - Conjunto de Dados Java . . . 60
Tabela 5 – Detalhamento das classes geradas - Conjunto de Dados C# . . . 62
Tabela 6 – Instâncias recuperadas dos conjuntos de dados C# originais . . . 62
Tabela 7 – Atributos Selecionados com ANOVA e Qui-Quadrado . . . 65
Tabela 8 – Detalhamento dos Hiperparâmetros . . . 67
Tabela 9 – Configuração dos Conjuntos de Dados de Treinamento, Otimização de Hiperparâmetros e Teste . . . 72
Tabela 10 – Os melhores Métodos de Comitê otimizados para detecção de *code smell* 73
Tabela 11 – Métodos de Comitê Otimizados para Transferência de Aprendizado de C# para Java . . . 74
Tabela 12 – Métodos de Comitê Otimizados para Transferência de Aprendizado de Java para C# . . . 74
Tabela 13 – Resultados para a TA de C# para Java . . . 76
Tabela 14 – Detalhamento das Avaliações de Gravidades - TA C# para Java . . . 76
Tabela 15 – Resultados para a TA de Java para C# . . . 80
Tabela 16 – Detalhamento das Avaliações de Gravidades - TA Java para C# . . . 80
Tabela 17 – Resumo das Análises de Explicabilidade por *Code Smell* Java . . . 84
Tabela 18 – Resumo das Análises de Explicabilidade por *Code Smell* C# . . . 87
Tabela 19 – Avaliação das medidas de Precisão, Sensibilidade e Pontuação F1 - TA C# para Java . . . 91
Tabela 20 – Avaliação das medidas de Precisão, Sensibilidade e Pontuação F1 - TA Java para C# . . . 92
Tabela 21 – Critérios de Inclusão . . . 108
Tabela 22 – Critério de Exclusão . . . 108
Tabela 23 – Avaliação de Qualidade . . . 109
Tabela 24 – Detalhamento dos estudos excluídos . . . 112
Tabela 25 – Fontes de dados e resultados de pesquisa . . . 112
Tabela 26 – Formulário de Extração de Dados . . . 113
Tabela 27 – Localização dos atributos selecionados no vetor de atributos do conjunto de dados . . . 115
Tabela 28 – Hiperparâmetros selecionados para os detectores de *code smell RF* . . 117
Tabela 29 – Hiperparâmetros selecionados para os detectores de *code smell XGB* . 117

Tabela 30 – Hiperparâmetros selecionados para os detectores de *code smell CB* . . 118
Tabela 31 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *QQ-Norm-25%-binário_RS* 119
Tabela 32 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *QQ-Original_RS* 120
Tabela 33 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *QQ-Norm-30%-binário_BS* 121
Tabela 34 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *Anova-30%-multiclasse_BS* 122
Tabela 35 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *QQ-Norm-20%-multiclasse_BS* 123
Tabela 36 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *Anova-25%-binário_BS* 124
Tabela 37 – Resultados para conjunto de dados GC Java 139
Tabela 38 – Resultados para conjunto de dados LM Java 139
Tabela 39 – Resultados para conjunto de dados DC Java 139
Tabela 40 – Resultados para conjunto de dados FE Java 140
Tabela 41 – Resultados para conjunto de dados GC C# 140
Tabela 42 – Resultados para conjunto de dados LM C# 140
Tabela 43 – Resultados para conjunto de dados DC C# 140
Tabela 44 – Resultados para conjunto de dados FE C# 141

LISTA DE ABREVIATURAS E SIGLAS

Acc	Acurácia
AD	Árvore de Decisão
AdaBoost	*Adaptive Boosting*
AM	Aprendizado de Máquina
ANOVA	Análise de Variância
AP	Aprendizado Profundo
AvGrav	Avaliação de Gravidade
Bagging	*Booststrap Aggregating*
Bin	Binário
BS	Busca Bayesiana
B-SMOTE	*Borderline-SMOTE*
CB	*Categorical Boosting*
DC	*Data Class*
DetCS	Detecção de *Code Smell*
Escal	Escalonamento de Dados
F1	Pontuação F1
FE	*Feature Envy*
GB	*Grandient Boosting*
GC	*God Class*
LIME	*Local Interpretable Model-Agnostic Explanations*
LM	*Long Method*
MC	Métodos de Comitê
Mean	Média do Desempenho para os MC
Mult	Multiclasse

NCS	Número de tipos de *Code Smells*
Norm	Normalização
OH	Otimização de Hiperparâmetros
OO	Orientado a Objeto
P	Precisão
Padr	Padronização
PCA	Análise de Componentes Principais
PLN	Processamento de Linguagem Natural
Pred	Preditores
QQ	Qui-Quadrado
RF	*Random Forest*
RS	Busca Randomizada
S	Sensibilidade
SelAtr	Seleção de Atributos
SHAP	*SHapley Additive exPlanations*
SMOTE	*Synthetic Minority Oversampling Technique*
Sobr	Sobreamostragem
std	Desvio Padrão do Desempenho para os MC
TA	Tranferência de Aprendizado
Test	Conjunto de Dados de Teste
Train	Conjunto de Dados de Treinamento
XGB	*Extreme Gradient Boosting*

LISTA DE SÍMBOLOS

μ	É a média dos dados. O valor médio de todos os elementos no conjunto de dados.
σ	É o desvio padrão dos dados, que mede a dispersão dos valores em torno da média.
NCS	Número de *Code Smells*
X	É o valor original do dado que será padronizado ou normalizado.
X'	É o valor normalizado. O novo valor após a transformação.
$X_{\min}$	É o menor valor no conjunto de dados.
$X_{\max}$	É o maior valor no conjunto de dados.
Z	É o valor padronizado (ou Z-score), que representa quantos desvios padrão um valor está da média.

SUMÁRIO

1 INTRODUÇÃO 21
1.1 MOTIVAÇÃO 21
1.2 OBJETIVOS 22
1.2.1 JUSTIFICATIVA 24
1.3 ESTRUTURA DO TEXTO 25

2 FUNDAMENTAÇÃO TEÓRICA 26
2.1 *CODE SMELL* 26
2.2 PRÉ-PROCESSAMENTO DE DADOS PARA CLASSIFICAÇÃO EM AM . . 31
2.3 MÉTODOS DE COMITÊ 34
2.4 TRANSFERÊNCIA DE APRENDIZADO 37
2.5 EXPLICABILIDADE EM MODELOS DE AM 39

3 TRABALHOS RELACIONADOS 42
3.1 PRÉ-PROCESSAMENTO DE DADOS PARA DETECÇÃO E AVALIAÇÃO DA GRAVIDADE DE *CODE SMELLS* 42
3.1.1 BALANCEAMENTO DE DADOS 42
3.1.2 SELEÇÃO DE ATRIBUTOS 43
3.1.3 ESCALONAMENTO DE DADOS 44
3.2 DETECÇÃO DE *CODE SMELL* COM AM 45
3.3 AVALIAÇÃO DA GRAVIDADE DE *CODE SMELL* BASEADO EM AM . . . 48
3.4 TA PARA DETECÇÃO E AVALIAÇÃO DA GRAVIDADE DE *CODE SMELLS* 49
3.5 ESTADO DA ARTE: LIMITAÇÕES E ESTRATÉGIAS DE MELHORIA . . 51
3.5.1 PRÉ-PROCESSAMENTO DE DADOS 51
3.5.2 DETECÇÃO DE *CODE SMELLS* 53
3.5.3 AVALIAÇÃO DE GRAVIDADE DE *CODE SMELLS* 53
3.5.4 TRANSFERÊNCIA DE APRENDIZADO PARA DETECÇÃO E AVALIAÇÃO DA GRAVIDADE DE *CODE SMELLS* 54

4 CONJUNTOS DE DADOS 56
4.1 O CONJUNTO DE DADOS JAVA 57
4.1.1 COMBINAÇÃO DOS CONJUNTOS DE DADOS JAVA 58
4.2 O CONJUNTO DE DADOS C# 60
4.2.1 COMBINAÇÃO DOS CONJUNTOS DE DADOS C# 61

5 **O MÉTODO DE AM PROPOSTO PARA DETECÇÃO E AVALIAÇÃO DA GRAVIDADE DE *CODE SMELLS*** **63**
5.1 PRÉ-PROCESSAMENTO DE DADOS 63
5.2 OTIMIZAÇÃO DE HIPERPARÂMETROS 66
5.3 DETECÇÃO DE *CODE SMELL* 69
5.4 AVALIAÇÃO DA GRAVIDADE 70

6 **EXPERIMENTOS E RESULTADOS** **72**
6.1 CONFIGURAÇÃO DOS EXPERIMENTOS 72
6.2 RESULTADOS 75
6.2.1 TRANSFERÊNCIA DE APRENDIZADO DE C# PARA JAVA 75
6.2.2 TRANSFERÊNCIA DE APRENDIZADO DE JAVA PARA C# 79
6.3 EXPLICABILIDADE DOS MODELOS AM 83
6.3.1 EXPLICABILIDADE DOS MODELOS PARA PROJETOS JAVA 83
6.3.2 EXPLICABILIDADE DOS MODELOS PARA PROJETOS C# 86
6.4 AMEAÇAS À VALIDADE 89
6.5 DISCUSSÃO 90

7 **CONCLUSÃO** **93**

REFERÊNCIAS **96**

APÊNDICE A – METODOLOGIA DA REVISÃO SISTEMÁTICA DA LITERATURA **107**
A.1 ESTRATÉGIA DE BUSCA 107
A.2 SELEÇÃO DE ESTUDOS 108
A.2.1 CRITÉRIOS DE INCLUSÃO/EXCLUSÃO 108
A.2.2 AVALIAÇÃO DE QUALIDADE 109
A.2.3 *SNOWBALLING* 109
A.3 VISÃO GERAL DOS ESTUDOS PRIMÁRIOS 110
A.4 ANÁLISE DE DADOS 112
A.4.1 EXTRAÇÃO DE DADOS 112
A.4.2 SÍNTESE DE DADOS 114

APÊNDICE B – LOCALIZAÇÃO E FREQUÊNCIA DOS ATRIBUTOS SELECIONADOS **115**

APÊNDICE C – HIPERPARÂMETROS SELECIONADOS PARA OS MÉTODOS DE COMITÊ UTILIZADOS NA DETECÇÃO DE *CODE SMELL* **117**

APÊNDICE D – HIPERPARÂMETROS SELECIONADOS PARA OS MÉTODOS DE COMITÊ UTILIZADOS NA AVALIAÇÃO DE GRAVIDADE . . . 119

APÊNDICE E – EVOLUÇÃO DAS MEDIDAS DE DESEMPENHO E DO TEMPO DE OTIMIZAÇÃO DE HIPERPARÂMETROS PARA A DETECÇÃO DE *CODE SMELL* COM ANOVA . . . 125

APÊNDICE F – EVOLUÇÃO DAS MEDIDAS DE DESEMPENHO E DO TEMPO DE OTIMIZAÇÃO DE HIPERPARÂMETROS PARA A DETECÇÃO DE *CODE SMELL* COM QUI-QUADRADO . . . 130

APÊNDICE G – DIAGRAMAS DE CLASSES DO ANALISADOR DE MÉTRICAS DE CÓDIGO . . . 135

APÊNDICE H – RESULTADOS DETALHADOS DA TRANSFERÊNCIA DE APRENDIZADO . . . 139

H.1 RESULTADOS DA TRANSFERÊNCIA DE APRENDIZADO DE C# PARA JAVA . . . 139

H.2 RESULTADOS DA TRANSFERÊNCIA DE APRENDIZADO DE JAVA PARA C# . . . 140

APÊNDICE I – AVALIAÇÃO DA GRAVIDADE DO MÉTODO CAPTUREWITHCURSOR . . . 142

APÊNDICE J – GRÁFICOS DE EXPLICABILIDADE - FE JAVA . 152

J.1 IMPORTÂNCIA DOS ATRIBUTOS . . . 152

J.2 SHAP . . . 153

J.3 LIME . . . 155

APÊNDICE K – GRÁFICOS DE EXPLICABILIDADE - LM JAVA . 159

K.1 IMPORTÂNCIA DOS ATRIBUTOS . . . 159

K.2 SHAP . . . 160

K.3 LIME . . . 160

APÊNDICE L – GRÁFICOS DE EXPLICABILIDADE - DC JAVA . 165

L.1 IMPORTÂNCIA DOS ATRIBUTOS . . . 165

L.2 SHAP . . . 166

L.3 LIME . . . 166

APÊNDICE M – GRÁFICOS DE EXPLICABILIDADE - GC JAVA . 171
M.1 IMPORTÂNCIA DOS ATRIBUTOS 171
M.2 SHAP 172
M.3 LIME 172

APÊNDICE N – GRÁFICOS DE EXPLICABILIDADE - FE C# . . 177
N.1 IMPORTÂNCIA DOS ATRIBUTOS 177
N.2 LIME 178

APÊNDICE O – GRÁFICOS DE EXPLICABILIDADE - LM C# . . 181
O.1 IMPORTÂNCIA DOS ATRIBUTOS 181
O.2 LIME 182

APÊNDICE P – GRÁFICOS DE EXPLICABILIDADE - DC C# . . 187
P.1 IMPORTÂNCIA DOS ATRIBUTOS 187
P.2 SHAP 188
P.3 LIME 188

APÊNDICE Q – GRÁFICOS DE EXPLICABILIDADE - GC C# . . 192
Q.1 IMPORTÂNCIA DOS ATRIBUTOS 192
Q.2 SHAP 193
Q.3 LIME 193

1 INTRODUÇÃO

O conceito de *code smell* foi inicialmente introduzido por Fowler et al.(1) e alude a indicadores primários de dívidas técnicas no código-fonte, destacando áreas que necessitam de manutenção completa. Desta maneira, *code smell* refere-se a um sintoma no código causado por falhas de design ou práticas de codificação inadequadas, podendo comprometer a qualidade do software e dificultar o desenvolvimento e a manutenção de sistemas. Alguns fatores contribuem para o surgimento dessas dívidas técnicas, como prazos rigorosos e introdução de novos requisitos durante o desenvolvimento de software (1).

A detecção de *code smell* é um processo baseado em dados que visa identificar violações dos princípios fundamentais de *design* em trechos de código. Diversas ferramentas para detecção de *code smell*, incluindo tanto ferramentas comerciais quanto protótipos de pesquisa, foram propostas. Essas ferramentas adotam diversas técnicas para identificar *code smell*: algumas baseiam-se em métricas (2, 3), enquanto outras utilizam uma linguagem de especificação dedicada (4), análise de programa para identificar oportunidades de refatoração (5, 6), análise de repositórios de software (7), ou técnicas de aprendizado de máquina (AM) (8). A maioria dessas abordagens utiliza heurísticas e diferencia artefatos de código afetados (ou não) por um determinado tipo de *code smell*, aplicando regras de detecção que comparam os valores de métricas relevantes extraídas do código-fonte com limites identificados empiricamente (9).

A gravidade de um *code smell* refere-se ao nível de impacto que suas características específicas exercem sobre a qualidade do software. Cada *code smell* carrega atributos que, em maior ou menor intensidade, contribuem para potenciais problemas de manutenibilidade, legibilidade e desempenho. Assim, a gravidade é uma medida que estima a quantidade e a intensidade desses atributos em uma ocorrência de *code smell* (10). A avaliação da gravidade categoriza os problemas associados a cada *code smell*, permitindo a priorização de custos e esforços de manutenção de software com base na gravidade enfrentada. Neste sentido, a abordagem baseada em AM possui a vantagem de poder explorar qualquer característica do código que um desenvolvedor considere importante para definir, incluindo a gravidade de um *code smell*, na medida em que essa característica pode ser estimada por meio de uma ou mais métricas de software (10).

1.1 Motivação

Apesar das diversas abordagens existentes, quando feito utilizando-se AM, a detecção e a avaliação de gravidade de *code smells* enfrenta desafios devido à natureza desequilibrada dos dados, quando uma ou mais classes estão desproporcionalmente re-

presentadas em um conjunto de dados, e à propensão a vieses de interpretação (9). Este desequilíbrio se apresenta, basicamente, de duas formas. Primeiro, os conjuntos de dados de *code smells* disponíveis são frequentemente rotulados manualmente usando um procedimento personalizado ou centrado em crenças (11). Além de ser dispendiosa, a rotulagem manual cria conjuntos de dados desequilibrados, propostos a vieses de interpretação (9).

Segundo, os conjuntos de dados utilizados pelos estudos existentes para a detecção e a avaliação da gravidade de *code smells* se valem de conjuntos de dados que compreendem instâncias de apenas um tipo específico de *code smell.* Essa orientação separa essas abordagens de cenários mais realista dos softwares existentes, pois negligenciam a influência que pode ser exercida por diferentes tipos de *code smells* uns sobre os outros. De fato, alguns trabalhos se concentraram na detecção de *code smells* apenas em um nível (e.g., (12, 13)), enquanto outros, mesmo focando nos dois níveis, a detecção e avaliação da gravidade dos conjuntos de dados ocorreram separadamente para cada tipo de *code smell* (e.g., (10, 14, 15)).

Além do problema do desequilíbrio, os conjuntos de dados disponíveis para experimentação sobre a detecção e a avaliação da gravidade de *code smells* são limitados principalmente a Java (11, 16). Assim, é necessário ampliar esses conjuntos de dados para outras linguagens de programação, incluindo C# que é amplamente utilizada no desenvolvimento de softwares e os seus desenvolvedores estão cada vez mais conscientes dos impactos de *code smells* (17). Entretanto, os softwares em C# tem poucas ferramentas que suportem análise de código para enfrentar esse problema (17, 18).

Esses desafios apontam para a necessidade de métodos mais abrangentes que abordem a detecção e avaliação de gravidade de forma conjunta e considerem múltiplas linguagens de programação.

1.2 Objetivos

O objetivo deste trabalho é apresentar um método de AM, que se vale de Métodos de Comitê (MC), para a detecção e a avaliação da gravidade de *code smells* em dois níveis distintos: níveis de classe e método. Além disso, este trabalho visa ainda aplicar o método proposto para a detecção e a avaliação da gravidade de *code smells* em projetos de software desenvolvidos com C#, através da aplicação de Transferência de Aprendizado (TA).

Para tanto, primeiro, criou-se um novo conjunto de dados Java para apresentar uma representação mais realista de *code smells* de software, uma vez que o software pode apresentá-los nesses dois níveis. O pré-processamento deste conjunto de dados incluiu a seleção dos atributos com mais influência na detecção de *code smells* e na avaliação de sua gravidade

Depois, desenvolveu-se um método de AM para a detecção e a avaliação da gravidade de *code smells* para os níveis de classe e método. O método proposto se diferencia dos estudos existentes para a detecção e a avaliação da gravidade de *code smells* que se valem de AM, pois estes aplicam um único método supervisionado. Normalmente, um método supervisionado explora um conjunto de variáveis independentes para determinar o valor de uma variável dependente (i.e, presença de um *code smell* ou de sua gravidade em um trecho de código) usando um único método de AM (19).

Este trabalho se vale do uso de MC para a detecção e a avaliação da gravidade de *code smells*. Um MC é uma técnica, no contexto de AM, que combina múltiplos modelos (ou "membros do comitê") para melhorar a precisão e a robustez das previsões. A ideia central é que, ao reunir a "opinião" de vários modelos, é possível reduzir o erro individual de cada modelo e obter um desempenho geral superior (20, 21).

Para aumentar a precisão do método de MC utilizado, foi abordado o escalonamento de dados, o balanceamento de dados e a otimização de hiperparâmetros. Isso permitiu a detecção de instâncias sem ou com *code smell* e a avaliação de sua gravidade.

Em seguida, combinou-se dois conjuntos de dados existentes de Slivka et al.(11) e Prokic et al.(22) para criar um conjunto de dados de *code smell* C#. Este novo conjunto de dados contém instâncias com os atributos selecionados durante o pré-processamento do conjunto de dados Java.

Finalmente, para que o método proposto seja capaz de detectar e avaliar a gravidade de *code smells* em projetos C#, aplicou-se a técnica de TA. Essa é uma técnica de AM onde o conhecimento adquirido ao treinar um modelo em uma tarefa é reaproveitado para melhorar o desempenho em uma tarefa diferente, mas relacionada (23). Como as características dessa técnica permitem que um algoritmo aproveite similaridades entre diferentes tarefas para facilitar a transferência de conhecimento entre elas (23), TA foi aplicada para treinar os modelos com instâncias Java e testar a detecção e a avaliação da gravidade de *code smells* em instâncias C#.

Assim, as contribuições esperadas para este trabalho são:

- um conjunto de dados Java que contenha instâncias de múltiplos tipos de gravidade de *code smells*, que seja mais realista e facilite aos algoritmos de AM aprenderem melhor as nuances dessas instâncias, com base em um conjunto de métricas, i.e., variáveis independentes, abrangendo diversos aspectos da qualidade de software, como tamanho, complexidade, coesão, acoplamento, encapsulamento e herança.
- um conjunto de dados contendo instâncias de vários tipos de gravidades de *code smell* de projetos C#, com base em um conjunto de atributos selecionados de um conjunto de dados com instâncias de projetos Java.

- um método de AM em duas etapas (para projetos Java e C#) para detecção de *code smell* e avaliação da gravidade baseado em MC e TA. Esse método usa padronização de dados, sobreamostragem, seleção de atributos e otimização de hiperparâmetros.

1.2.1 Justificativa

A manutenção de sistemas corporativos em organizações como a Marinha do Brasil (MB) e o Exército Brasileiro (EB) enfrenta desafios singulares devido à sua complexidade e às limitações de recursos disponíveis. No caso específico do Escritório de Projetos e Sistemas Digitais do Corpo de Fuzileiros Navais da Marinha do Brasil (EPSD-CFN-MB), a responsabilidade pela manutenção dos Softwares Corporativos do CFN recai sobre uma equipe de desenvolvedores restrita, dado que o principal objetivo do CFN não é o desenvolvimento ou a manutenção de software. Esse cenário também se aplica ao Exército Brasileiro, que possui sistemas igualmente estratégicos para suporte às suas operações.

Nesse contexto, o uso de ferramentas baseadas em modelos de AM surge como uma solução promissora para enfrentar os desafios relacionados à detecção de *code smells* e à avaliação de sua gravidade. Essas ferramentas não apenas ajudam a identificar problemas que comprometem a qualidade do código, mas também permitem priorizar esforços de manutenção com base na gravidade dos problemas encontrados. Essa abordagem contribui para uma alocação mais eficiente dos recursos humanos, maximizando o impacto das intervenções e assegurando maior qualidade e continuidade no ciclo de vida dos softwares utilizados tanto pela Marinha quanto pelo Exército.

O estudo da gravidade de *code smells* é, portanto, essencial, pois permite categorizar e hierarquizar problemas relacionados à manutenção de software. Além disso, a relevância desse tema transcende o contexto das Forças Armadas, sendo aplicável a diversos projetos de software, como aqueles desenvolvidos em C#.

A relevância científica do tema também é corroborada pelo crescente interesse da comunidade acadêmica. Desde o estudo pioneiro de Fontana e Zanoni(10), que propôs a avaliação de gravidade de *code smells* utilizando AM, houve um aumento significativo no número de publicações nessa área. Nos últimos dois anos, seis estudos se destacaram (14, 15, 24, 25, 26, 27), evidenciando a importância da avaliação de gravidade como um dos principais desafios no campo da manutenção de software.

Portanto, investigar métodos mais robustos e automatizados de detecção e avaliação de gravidade de *code smells*, considerando múltiplos tipos de problemas e diferentes linguagens de programação, é importante para atender às demandas específicas da MB, do EB e de outras organizações estratégicas.

1.3 Estrutura do Texto

A dissertação está estruturada de maneira a fornecer os conceitos e experimentos necessários ao entendimento do modelo nela proposto. Para isto, além desta introdução, o Capítulo 2 apresenta uma fundamentação teórica abordando os conceitos básicos que nortearão o presente estudo. O Capítulo 3 apresenta a revisão da literatura e o estado da arte para a detecção de *code smell* e avaliação de gravidade. O Capítulo 4 detalha os conjuntos de dados e o processo de combinação deles tanto para Java quanto para C#. A metodologia utilizada para atingir os objetivos propostos, no qual é descrita cada uma das etapas do processo de pesquisa é apresentada no Capítulo 5. O Capítulo 6 especifica os experimentos realizados e a análise dos resultados obtidos. Finalmente, o Capítulo 7 descreve as principais conclusões deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados e discutidos os principais conceitos e técnicas que embasam esta pesquisa, incluindo *code smells*, pré-processamento de dados, métodos de comitê, transferência de aprendizado e explicabilidade em AM. Inicialmente, é introduzido o conceito de *code smells* e alguns dos seus tipos mais estudados na literatura (28) (seção 2.1). Em seguida, são exploradas técnicas de pré-processamento de dados, como balanceamento, seleção de atributos e escalonamento de dados, ressaltando sua relevância para a melhoria da qualidade dos dados e do desempenho dos modelos (seção 2.2). Também são discutidos métodos de comitê, como *Bootstrap Aggregating (Bagging)* e *Boosting*, que visam aprimorar a robustez e a precisão dos algoritmos de AM (seção 2.3). Na seção 2.4 a transferência de aprendizado é abordada em termos de seu impacto, destacando como modelos pré-treinados podem ser utilizados para enfrentar diferentes contextos e linguagens de programação. Por fim, na seção 2.5, a explicabilidade em AM é tratada por meio de ferramentas como SHAP e LIME, que permitem interpretar os modelos e compreender a influência das métricas nos resultados.

2.1 *Code Smell*

Code Smell refere-se a sinais iniciais de dívidas técnicas – metáfora usada para descrever os impactos negativos de longo prazo causados por decisões subótimas durante o desenvolvimento de software (29) – presentes no código-fonte, indicando partes que requerem atenção em manutenção. Este conceito foi introduzido por Fowler et al.(1). Desse modo, um *code smell* representa um sintoma causado por falhas de design ou práticas de codificação inadequadas, com potencial para comprometer a qualidade do software, além de dificultar o desenvolvimento e a manutenção dos sistemas. A presença de *code smells* em um projeto acarreta uma série de riscos que podem afetar diretamente a produtividade da equipe e a sustentabilidade do software a longo prazo. Esses sintomas de má qualidade de código, quando não tratados, tendem a se acumular, aumentando a complexidade e tornando o sistema mais propenso a erros (30). Como consequência, a manutenção e evolução do software se tornam tarefas mais difíceis, demoradas e custosas (1). Além disso, *code smells* podem reduzir a legibilidade e a previsibilidade do código, dificultando a compreensão dos desenvolvedores e elevando a probabilidade de introdução de novos defeitos (31). Em contextos de desenvolvimento ágil, onde mudanças e adaptações frequentes são necessárias, a negligência com *code smells* pode limitar a capacidade de resposta do sistema, comprometendo sua qualidade e, em última análise, a satisfação do usuário final (32).

Os *code smells* são divididos em dois níveis: classe e método. Os *code smells* de nível de classe referem-se a problemas estruturais que afetam uma classe inteira, indicando um design inadequado ou responsabilidades mal distribuídas. Exemplos comuns incluem classes que centralizam excessivas responsabilidades, dificultando a manutenção e a testabilidade, e as classes que armazenam apenas dados, dependendo de outras classes para manipulação, o que leva a um acoplamento indesejável. Já os *code smells* de nível de método ocorrem dentro de métodos individuais e geralmente indicam implementações complexas e pouco coesas. Exemplos incluem métodos que executam múltiplas responsabilidades em um único bloco de código, e os métodos que acessam excessivamente dados de outra classe, violando o encapsulamento. A seguir, são apresentadas as definições de instâncias de *code smells* incluídas nos conjuntos de dados deste trabalho:

***God-class* (GC)**: classes que centralizam grande parte da lógica do sistema, caracterizadas por alta complexidade, tamanho extenso e múltiplas responsabilidades. Podem utilizar muitos dados de outras classes e implementar funcionalidades variadas (27). Abaixo é apresentado um exemplo de GC, onde uma classe assume responsabilidades de usuários, produtos, pedidos, notificações e geração de relatórios, que poderiam ser delegadas a classes mais específicas.

```
1  public class GodClass {
2  private List<String> usuarios = new ArrayList<>();
3      private List<String> produtos = new ArrayList<>();
4
5      public void adicionarUsuario(String usuario) {
6          usuarios.add(usuario);
7          System.out.println("Usuário␣adicionado:␣" + usuario);
8      }
9
10     public void adicionarProduto(String produto) {
11         produtos.add(produto);
12         System.out.println("Produto␣adicionado:␣" + produto);
13     }
14
15     public void processarPedido(String usuario, String produto) {
16         if (usuarios.contains(usuario) && produtos.contains(
               produto)) {
17             System.out.println("Pedido␣processado␣para␣o␣usuário:
                   ␣" + usuario + "␣com␣o␣produto:␣" + produto);
18         } else {
19             System.out.println("Usuário/Produto␣não␣encontrado");
20         }
21     }
```

```
22
23      public void enviarNotificacao(String mensagem) {
24          System.out.println("Notificação␣enviada:␣" + mensagem);
25      }
26
27      public void gerarRelatorio() {
28          System.out.println("Gerando␣relatório...");
29          System.out.println("Total␣de␣usuários:␣" + usuarios.size
               ());
30          System.out.println("Total␣de␣produtos:" + produtos.size()
               );
31      }
32  }
```

***Data-class* (DC)**: classes que funcionam principalmente como contêineres de dados, possuem funcionalidades limitadas e frequentemente fornecem acesso a seus atributos via métodos acessadores (27). No caso de DC, as classes não possuem lógica além dos acessadores e modificadores. Isso pode se tornar um problema caso esses dados estejam espalhados pelo código e modifiquem sua lógica em vários lugares.

```
1   public class DataClass {
2       private String nome;
3       private double preco;
4
5       public DadosProduto(String nome, double preco) {
6           this.nome = nome;
7           this.preco = preco;
8       }
9
10      public String getNome() {
11          return nome;
12      }
13
14      public void setNome(String nome) {
15          this.nome = nome;
16      }
17
18      public double getPreco() {
19          return preco;
20      }
21
22      public void setPreco(double preco) {
```

```
23         this.preco = preco;
24     }
25 }
```

***Feature-envy* (FE)**: métodos que utilizam majoritariamente dados de outras classes em vez dos seus próprios, frequentemente acessando atributos de outras classes através de métodos acessadores (27). No exemplo a seguir de método com o *code smell* FE, *ServicoNotificacao* depende de dados de *Cliente* para enviar a notificação. Seria mais adequado que a própria classe *Cliente* tivesse uma função para verificar se o saldo está baixo e enviar uma notificação.

```
1  public class Cliente {
2      private String nome;
3      private String email;
4      private double saldo;
5
6      public Cliente(String nome, String email, double saldo) {
7          this.nome = nome;
8          this.email = email;
9          this.saldo = saldo;
10     }
11
12     public String getEmail() {
13         return email;
14     }
15
16     public double getSaldo() {
17         return saldo;
18     }
19 }
20
21 public class ServicoNotificacao {
22     public void enviarNotificacaoSaldoBaixo(Cliente cliente) {
23         if (cliente.getSaldo() < 50) {
24             System.out.println("Enviando␣notificação␣de␣saldo␣
                   baixo␣para␣" + cliente.getEmail());
25         }
26     }
27 }
```

***Long-method* (LM)**: métodos que agregam muitas funcionalidades em uma única unidade, tornando-se longos, complexos e difíceis de entender, com alto grau de

dependência de dados externos (27). O código abaixo é um exemplo de LM, onde o método é de difícil entendimento e pode indicar que o método está lidando com mais de uma responsabilidade (validação, verificação de disponibilidade e cálculo de desconto). Esse método pode ser refatorado para dividir cada uma das operações em métodos separados.

```java
public class ExemploLongMethod {

    public void processarTransacao(String usuario, String produto, int quantidade) {
        System.out.println("Validando transação...");

        if (usuario == null || produto == null) {
            System.out.println("Transação inválida. Usuário ou produto está nulo.");
            return;
        }

        System.out.println("Verificando disponibilidade do produto...");
        if (quantidade < 1) {
            System.out.println("A quantidade deve ser pelo menos 1.");
            return;
        }

        System.out.println("Calculando desconto...");
        double desconto = 0.1 * quantidade;
        double preco = 100 * quantidade - desconto;

        System.out.println("Processando pagamento...");
        if (preco <= 0) {
            System.out.println("Erro no processamento do pagamento.");
            return;
        }

        System.out.println("Transação concluída. Preço: " + preco);
    }
}
```

2.2 Pré-Processamento de Dados para Classificação em AM

A classificação é uma tarefa de AM que tem o objetivo de prever uma categoria ou classe a que uma nova observação pertence, com base em um conjunto de dados de treinamento com observações já categorizadas, assumindo valores de um conjunto discreto e não ordenado (33). Existem três tipos principais de classificação: i) binária, com apenas duas classes possíveis; ii) multiclasse, com mais de duas classes; e iii) multirótulo, onde uma observação pode pertencer a várias classes simultaneamente. Além disso, para a classificação alguns passos são utilizados, como: o pré-processamento do conjunto de dados; o treinamento dos modelos de classificação; a validação e o teste dos modelos usados para avaliar seu desempenho; e a predição, utilizando os modelos treinados para preverem as classes de novas observações. Entretanto, o pré-processamento de dados destaca-se como um aspecto fundamental na construção de modelos de AM de alto desempenho. Esse processo envolve a limpeza e transformação de dados brutos para facilitar a análise e a identificação de informações relevantes, melhorando, em última análise, o desempenho do modelo (34)

O pré-processamento de dados foca principalmente em duas áreas principais: o balanceamento de dados e a seleção de atributos (35); e, também, no escalonamento de dados. Balanceamento de dados refere-se ao processo de ajustar a distribuição das classes em um conjunto de dados para garantir que cada classe tenha uma representação proporcional ou igual e visa melhorar a performance do modelo de AM máquina, garantindo que ele não seja tendencioso ou enviesado em favor das classes majoritárias (33, 36). As principais técnicas utilizadas para o balanceamento de dados buscam aumentar a quantidade de exemplos na classe minoritária de um conjunto de dados, com o objetivo de equilibrar a distribuição de classes (35, 37) (sobreamostragem), e.g., *Synthetic Minority Oversampling Technique* (SMOTE) e Borderline-SMOTE (B-SMOTE). Além dessas técnicas de sobreamostragem, foram utilizadas técnicas de subamostragem, que buscam balancear classes em um conjunto de dados desbalanceado, reduzindo o número de exemplos da classe majoritária (38), bem como técnicas que combinam a sobreamostragem e a subamostragem, e.g., *Synthetic Minority Over-sampling Technique and Edited Nearest Neighbors* (SMOTEEN) (39).

SMOTE enfrenta o problema do desbalanceamento de dados criando novas amostras na classe minoritária até que sua quantidade se equipare à da classe majoritária, preservando a qualidade dos dados e uma distribuição mais balanceada de classes como entrada (40). Inicialmente SMOTE faz a seleção de exemplos minoritários, para cada exemplo minoritário no conjunto de dados, SMOTE seleciona aleatoriamente alguns de seus vizinhos mais próximos (*k-nearest neighbors*).

Depois, novos exemplos sintéticos são gerados interpolando entre o exemplo minoritário original e seus vizinhos selecionados. A fórmula básica para a criação de um novo exemplo sintético é: novo exemplo = exemplo original + $\lambda \times$ (vizinho − exemplo original),

onde λ é um valor aleatório entre 0 e 1, que determina a posição do novo exemplo ao longo da linha entre o exemplo original e o vizinho. E, por fim, os novos exemplos sintéticos são então adicionados ao conjunto de dados, aumentando o número de exemplos na classe minoritária até que a proporção entre as classes fique mais balanceada (37).

B-SMOTE é uma versão aprimorada do SMOTE. Essa técnica de sobreamostragem prioriza a otimização das amostras de borda, aproveitando as características de SMOTE para lidar mais adequadamente com instâncias de limite das classes minoritárias (41). Essa técnica é útil para melhorar a capacidade do modelo de distinguir entre as classes, especialmente em áreas onde a separação não é clara. O funcionamento deste algoritmo começa com a identificação dos exemplos da classe minoritária que estão perto das fronteiras de decisão, determinando os k-vizinhos mais próximos para cada exemplo minoritário.

Em seguida, cada exemplo minoritário é classificado em três categorias: i) *Safe*, se a maioria dos k-vizinhos mais próximos pertencer à classe minoritária; ii) *Danger*, se aproximadamente metade dos k-vizinhos pertencer à classe majoritária e metade à minoritária; e iii) *Noise*, se a maioria dos k-vizinhos pertencer à classe majoritária. Então, apenas os exemplos classificados como "*Danger*" são utilizados para gerar novos exemplos sintéticos, pois esses exemplos estão nas regiões mais difíceis onde a distinção entre classes é mais importante. Assim, novo exemplos sintéticos são criados interpolando entre os exemplos "*Danger*" e seus vizinhos da classe minoritária, de maneira semelhante ao SMOTE: $\text{novo exemplo} = \text{exemplo danger} + \lambda \times (\text{vizinho} - \text{exemplo danger})$, onde λ é um valor aleatório entre 0 e 1. Finalmente, os novos exemplos sintéticos são então adicionados ao conjunto de dados, ajudando a equilibrar a distribuição das classes, especialmente nas áreas *borderline* (42).

SMOTEEN é uma técnica de sobreamostragem e subamostragem combinadas, que usa a técnica de sobreamostragem SMOTE para criar instâncias sintéticas da classe minoritária e, em seguida, aplica o método *Edited Nearest Neighbors* (ENN) para limpar as instâncias de ambas as classes, removendo aquelas que estão próximas aos seus vizinhos de classes opostas. Dessa forma, SMOTEEN ajuda a equilibrar conjuntos de dados desbalanceados, tanto aumentando a representatividade da classe minoritária quanto removendo ruído do conjunto de dados (39).

A seleção de atributos, e.g., com Qui-quadrado ou Análise de Variância (ANOVA), identifica os elementos mais relevantes em um conjunto de dados, focando na melhoria do desempenho e das métricas de software (43). Os atributos mais impactantes são as características do conjunto de dados que mais influenciam a variável alvo ou as previsões do modelo (33). Essa prática é fundamental para melhorar a compreensão dos padrões nos dados e otimizar o desempenho dos modelos, ajudando no melhor entendimento do problema e na melhoria da performance do modelo (43).

Qui-quadrado é uma técnica usada para atributos categóricos, que calcula o valor

do qui-quadrado entre cada métrica e seleciona as métricas com as melhores pontuações para cada conjunto de dados (44). A fórmula do teste qui-quadrado é apresentada na Equação 2.1 (15):

$$X^2 = \frac{(FrequenciaObservada - FrequenciaEsperada)^2}{FrequenciaEsperada} \tag{2.1}$$

Por fim, ANOVA é uma técnica estatística comumente usada para comparar diferenças entre médias de três ou mais grupos ou populações independentes (45). O funcionamento desta técnica passa por quatro etapas (45):

- **Definição de Hipóteses:** Hipótese Nula (H0), afirma que todas as médias dos grupos são iguais. Hipótese Alternativa (H1), afirma que pelo menos uma das médias dos grupos é diferente.
- **Cálculo das Variâncias:** Variância Entre os Grupos (*Between-group variance*), mede a variação das médias dos grupos em relação à média geral. Variância Dentro dos Grupos (*Within-group variance*), mede a variação dentro de cada grupo individualmente.
- **Cálculo do Estatístico F:** O valor F é calculado dividindo a variância entre os grupos pela variância dentro dos grupos, Equação 2.2:

  $$F = \frac{\text{Variância Entre os Grupos}}{\text{Variância Dentro dos Grupos}} \tag{2.2}$$

  Um valor F alto indica que há uma grande diferença entre as médias dos grupos, enquanto um valor F baixo sugere que as médias são semelhantes.
- **Determinação da Significância:** O valor F calculado é comparado a um valor crítico da distribuição F. Se o valor F calculado for maior que o valor crítico, rejeita-se a hipótese nula. O p-valor associado ao valor F também pode ser usado para determinar a significância. Se o p-valor for menor que o nível de significância (geralmente 0.05), rejeita-se a hipótese nula.

Além do balanceamento de dados e da seleção de atributos, para o pré-processamento dos dados foram utilizadas técnicas de escalonamento de dados. A utilização dessas técnicas é uma etapa de pré-processamento no AM que visa padronizar o intervalo de variáveis independentes ou características do conjunto de dados. Este processo envolve a transformação dos valores dos atributos para um intervalo específico, normalmente entre 0 e 1 ou -1 e 1, para garantir que tenham escalas semelhantes. Além disso, técnicas de escalonamento são aplicadas para evitar que atributos com escalas diferentes influenciem

de forma desproporcional o aprendizado dos modelos. As principais técnicas para detecção e avaliação da gravidade de *code smells* são normalização e padronização (35).

A normalização de dados é o processo de transformar as características de um conjunto de dados para que fiquem em uma escala comum, para assegurar que todas as características tenham a mesma influência no modelo, independentemente da sua escala original (46). A normalização Min-Max é empregada para redimensionar os valores de um atributo para que fiquem dentro do intervalo [0, 1]. A Equação 2.3 mostra a técnica Min-Max.

$$X' = \frac{X - X_{\min}}{X_{\max} - X_{\min}} \tag{2.3}$$

Padronização de dados é uma técnica que reduz os valores dos atributos a uma distribuição normal padrão, média é 0 e desvio padrão é 1 (47). Além disso, a Equação 2.4 descreve a padronização:

$$Z = \frac{X - \mu}{\sigma} \tag{2.4}$$

Onde Z é a variável padronizada, X é a variável original, μ é a média da variável original, e σ é o desvio padrão da variável original.

2.3 Métodos de Comitê

Ao tratar a questão dos *code smells*, diversos modelos de AM foram empregados pelos pesquisadores, incluindo *Naive Bayes* (NB), Máquinas de Vetores de Suporte (SVM), K-Nearest Neighbour (KNN), regressão logística (RL) e árvores de decisão (AD) (8, 19, 9, 44).

- **NB** (48): fundamentado no teorema de Bayes, o NB assume independência entre os recursos de software. Este algoritmo requer a estimativa de um número limitado de parâmetros e é reconhecido por sua robustez em situações de dados faltantes, o que favorece uma implementação relativamente simples.

- **SVM** (49): visa separar instâncias pertencentes a diferentes classes, identificando o hiperplano ideal em um espaço de recursos transformado. A transformação é realizada por meio de uma função *kernel*, que mapeia os dados originais para um espaço onde se tornam linearmente separáveis.

- **KNN** (50): classifica amostras com base nos rótulos das K amostras mais próximas, avaliando a similaridade entre elas pela distância no espaço de características.

- **RL** (51): é uma técnica de classificação binária que estima a probabilidade de um elemento pertencer a uma categoria específica, atribuindo-o a uma de duas possíveis classes.

- **AD** (52): constrói modelos de classificação por meio de uma estrutura hierárquica. Os dados são particionados em subconjuntos menores, enquanto uma árvore conectada é desenvolvida incrementalmente. Os nós folha representam os resultados finais ou as classificações.

Essas abordagens utilizam um único modelo de AM supervisionado para detectar *code smells*, explorando um conjunto de variáveis independentes para prever o valor de uma variável dependente, como a presença de um *code smell* ou sua gravidade em um trecho de código (19). Entre essas abordagens, destaque-se AD, que apresenta bons resultados tanto para detecção de *code smells* (44) quanto para avaliação da gravidade (15, 26), além de servir como modelo-base para técnicas de AM mais complexas. Contudo, avanços recentes nas técnicas e algoritmos de AM têm introduzido novas abordagens nesta área de pesquisa, incluindo métodos de comitê (MC) (24) e transferência de aprendizado (TA) (53) (seção 2.4).

Um MC é uma técnica de AM que combina múltiplos modelos (ou “membros do comitê”) para melhorar a precisão e a robustez das previsões dos modelos (20). A ideia central desses métodos é combinar vários classificadores, proporcionando resultados superiores em comparação com um único classificador. Essa abordagem é bem estabelecida e tutoriais detalhados foram desenvolvidos para orientar profissionais interessados na construção de sistemas de classificação baseados em comitês (20, 21).

Um comitê pode ser caracterizado como homogêneo ou heterogêneo. Um comitê homogêneo é construído com classificadores do mesmo tipo, treinados em subconjuntos distintos do conjunto de dados (54). Um comitê heterogêneo utiliza diferentes tipos de classificadores. Este trabalho se vale de comitês homogêneos pois são os mais utilizados para enfrentar o problema de detecção de *code smell* e avaliação de gravidade (35). Além disso, esses comitês têm mostrado desempenho consistente tanto para detecção de *code smell* (13) quanto para a avaliação de sua gravidade (15, 27, 24). Dentre eles destacam-se: *Bagging, Random Forest (RF), Boosting, Adaptive Boosting (AdaBoost), Gradient Boosting (GB), Light Gradient Boosting (LightGBM), Extreme Gradient Boosting (XGB) e o Categorical Boosting (CB)* (35).

Bagging é um meta-algoritmo de comitê poderoso usado para melhorar a estabilidade e precisão de algoritmos de AM (55). Esta técnica gera múltiplas amostras de *bootstrap* simultaneamente, começando a partir de um conjunto de dados de treinamento. Em seguida, ele treina vários classificadores binários, e a classificação final é determinada através de um mecanismo de votação (55, 56).

O RF é uma variação do *Bagging* que utiliza múltiplas árvores de decisão (AD) configuradas com parâmetros aleatórios para melhorar a estabilidade e precisão de algoritmos e reduzir o *overfitting* (55) – *overfitting* ocorre quando um modelo é treinado excessivamente bem em um conjunto de dados específico, a ponto de captar o ruído e as variações específicas desse conjunto, em vez de aprender os padrões gerais, resultando em um modelo que se sai bem nos dados de treinamento, mas mal nos dados não vistos (de teste) (33). Assim como no *bagging*, as árvores podem ser treinadas com réplicas do conjunto de treinamento ou em subconjuntos de diferentes características, aumentando a diversidade e robustez do modelo (20). O RF consiste de três etapas: construção de árvores, seleção de atributos e agregação de resultados. Primeiro, várias AD são construídas utilizando subconjuntos diferentes do conjunto de dados de treinamento. Cada árvore é construída com uma amostra aleatória (com reposição) dos dados.

Segundo, para cada divisão em uma árvore, um subconjunto aleatório de características é considerado. Isso garante que as árvores sejam diversificadas. E, então, as previsões de todas as árvores são combinadas. No caso de classificação, a previsão final é baseada na votação majoritária das árvores. Para regressão, a média das previsões das árvores é utilizada. Assim, utilizar RF trás algumas vantagens, como: redução de *overfitting* comparado a AD individuais; funciona bem com dados de alta dimensionalidade; e é robusto a *outliers* (valores extremos em um conjunto de dados que se desviam do restante dos dados (57)) e variáveis ruidosas. Entretanto, RF pode ter uma alta demanda computacional com grandes conjuntos de dados e muitas árvores, além de ser menos interpretável em comparação com AD únicas.

Boosting é considerado um dos algoritmos mais importantes da história do AM (20), é empregado na detecção de *code smell*. Semelhante ao *bagging*, o *boosting* cria um conjunto de classificadores por meio da reamostragem de dados combinados por meio de um mecanismo de votação por maioria. No entanto, ao contrário do *bagging*, o *boosting* concentra-se estrategicamente na reamostragem para fornecer os dados de treinamento mais informativos para cada classificador sucessivo.

Adaptive Boosting (AdaBoost), um algoritmo proeminente na família *boosting*, desempenha um papel importante na detecção de *code smell*. Este algoritmo treina modelos sequencialmente, gerando um novo modelo treinado para cada rodada. Assim, as instâncias identificadas incorretamente recebem maior peso nas rodadas subsequentes. *AdaBoost* aproveita essa característica para selecionar algoritmos de aprendizado típicos, como AD e Máquinas de Vetores de Suporte (SVM), como "aprendizes fracos" (58).

Para a detecção de *code smell*, Luiz, Oliveira e Parreiras(59) enfatizou a utilização do *Gradient Boosting*, que é uma técnica poderosa que amalgama um conjunto de alunos fracos para construir um aluno mais robusto. Este método de conjunto foi projetado para atingir menor viés e variância, aumentando sua eficácia.

LightGBM constrói um modelo aditivo usando AD simples, que são então generalizadas otimizando uma função de perda definida pelo usuário para aumentar a robustez da previsão. As vantagens do *LightGBM* incluem velocidade de treinamento, maior eficiência e melhor precisão em comparação com muitos outros algoritmos de *boosting*, juntamente com menor consumo de memória. Além disso, apresenta compatibilidade aprimorada com grandes conjuntos de dados (60).

XGB é um algoritmo de aprendizado baseado em árvores que se destaca pela velocidade, flexibilidade, portabilidade e desempenho. Desenvolvido pela *Distributed Machine-Learning Community (DMLC)*, é amplamente utilizado na detecção de *code smells* devido à sua eficácia em dados estruturados (15). O algoritmo começa com uma previsão inicial (geralmente a média para regressão ou proporção para classificação). Depois em cada iteração, os resíduos (diferenças entre as previsões e os valores reais) são calculados. Uma nova árvore é treinada para prever esses resíduos. Em seguida, as previsões do modelo são atualizadas adicionando-se uma fração das previsões da nova árvore às previsões existentes. E, por fim, são incluídos termos de regularização (L1 e L2) para evitar *overfitting*, tornando o modelo mais robusto. Nesse sentido, XGB é: focado na eficiência de tempo e memória; inclui regularização para reduzir *overfitting*; e suporta paralelização durante o treinamento. Contudo, XGB pode ser complexo de configurar devido aos muitos hiperparâmetros, além de ser sensível a *outliers*.

CB é um algoritmo de *boosting* projetado especificamente para lidar com características categóricas. Este MC utiliza o *Sort Boosting*, que substitui o método tradicional de estimativa de gradiente. Essa técnica reduz o desvio na estimativa, melhorando a capacidade de generalização do modelo (61). As vantagens do CB são: utilização de uma técnica interna para transformar características categóricas em uma forma numérica adequada para a construção de AD; similar ao XGB, utiliza uma abordagem de *boosting* para melhorar iterativamente as previsões do modelo; e a aplicação de técnicas avançadas para reduzir o *overfitting*, incluindo a injeção de ruído durante o treinamento. Em contrapartida, CB pode ser de alto custo computacional dependendo do tamanho dos dados. Além disso, este algoritmo é menos flexível em comparação com algoritmos que requerem pré-processamento manual.

2.4 Transferência de Aprendizado

Embora a tecnologia tradicional de AM tenha alcançado grande sucesso em diversas aplicações práticas, ela enfrenta limitações em cenários mais realistas. O cenário ideal para AM envolve a disponibilidade de um grande volume de instâncias de treinamento rotuladas, com a mesma distribuição dos dados de teste. No entanto, a coleta de um conjunto suficientemente amplo de dados rotulados é muitas vezes cara, demorada e, em

alguns casos, inviável. O aprendizado semi-supervisionado é uma abordagem que visa mitigar essa limitação ao combinar um número restrito de instâncias rotuladas com uma grande quantidade de dados não rotulados para melhorar a precisão do aprendizado. Essa técnica situa-se entre o aprendizado supervisionado, que depende inteiramente de dados rotulados, e o aprendizado não supervisionado, que não utiliza rótulos. No entanto, mesmo a obtenção de instâncias não rotuladas pode ser desafiadora em certos domínios, limitando o desempenho dos métodos tradicionais (62).

Para superar essas restrições, a TA emerge como uma alternativa, permitindo a reutilização do conhecimento adquirido em uma tarefa para outra, aproveitando similaridades entre domínios (23). De acordo com o estudo de Zhuang et al.(62) e de Weiss, Khoshgoftaar e Wang(63) a categorização da TA pode ser baseada em três formas principais: no problema, na similaridade dos domínios e na abordagem de solução.

A categorização com base no problema é focada na configuração do problema de aprendizado, considerando a disponibilidade e a similaridade dos rótulos entre os domínios de origem e destino:

- **Transferência Indutiva:** a informação de rótulos está disponível no domínio de origem e de destino. O objetivo é melhorar a performance no domínio de destino utilizando conhecimento do domínio de origem, mesmo que as distribuições dos domínios sejam diferentes.
- **Transferência Transdutiva:** a informação de rótulos está presente apenas no domínio de origem. Os domínios de origem e destino compartilham a mesma tarefa, mas as distribuições são diferentes.
- **Transferência Não Supervisionada:** nenhuma informação de rótulos está disponível tanto no domínio de origem quanto no de destino. O foco está na descoberta de estruturas subjacentes nos dados.

A categorização com base na similaridade dos domínios é baseada nas características e na similaridade entre os domínios:

- **Transferência Homogênea:** o domínio de origem e o domínio de destino compartilham o mesmo espaço de atributos ($X_S = X_T$) e o mesmo espaço de rótulos ($Y_S = Y_T$). Neste caso, as diferenças residem nas distribuições marginais ou condicionais, exigindo técnicas de adaptação de distribuição.
- **Transferência Heterogênea:** os domínios de origem e destino possuem diferentes espaços de atributos ($X_S \neq X_T$) ou diferentes espaços de rótulos ($Y_S \neq Y_T$). Este tipo de transferência é mais desafiador, pois exige mapeamento e adaptação entre diferentes representações de atributos e rótulos.

A categorização com base na abordagem de solução está relacionada ao método utilizado para realizar a transferência de conhecimento:

- **Baseada em Instâncias:** utiliza uma estratégia de ponderação para ajustar a relevância das instâncias do domínio de origem.
- **Baseada em Atributos:** transforma os atributos originais para criar uma nova representação que facilite a transferência de conhecimento.
- **Baseada em Parâmetros:** transfere conhecimento por meio do compartilhamento de parâmetros ou ajustes de modelos previamente treinados.
- **Baseada em Relações:** foca na transferência de conhecimento em domínios relacionais, utilizando regras e estruturas lógicas.

2.5 Explicabilidade em Modelos de AM

Interpretar corretamente a saída de um modelo preditivo auxilia no aumento da confiança dos usuários, oferece *insights* sobre possíveis melhorias no modelo e facilita a compreensão do processo modelado. A explicabilidade em AM refere-se à capacidade de um modelo em justificar suas decisões e previsões de forma compreensível para humanos. Essas técnicas permitem que desenvolvedores e outros interessados compreendam as características que mais influenciaram as decisões do modelo, aumentando a confiança no sistema e facilitando a aplicação prática dos resultados (64).

A explicabilidade em modelos de AM pode estar relacionada ao processo ou ao resultado (65). A explicabilidade do processo refere-se à capacidade de entender como o algoritmo aprende a partir dos dados de entrada e toma decisões. Isso permite rastrear o estado interno do algoritmo, compreender o processo de aprendizagem e identificar os fundamentos das decisões, esclarecendo como o resultado final é produzido. Já a explicabilidade dos resultados diz respeito à compreensão de como as saídas do algoritmo se relacionam com os dados de entrada. Essa explicabilidade possibilita entender a base e a importância atribuída pelo algoritmo a diferentes características dos dados de entrada, facilitando a avaliação da razoabilidade e credibilidade das decisões tomadas.

As abordagens para alcançar a explicabilidade nos modelos de AM dividem-se em duas categorias principais: design transparente e interpretação *ex post facto*. Por um lado, o design transparente envolve métodos que aumentam a clareza e facilitam a compreensão do processo de tomada de decisão dos sistemas de AM, permitindo que os usuários entendam como esses sistemas operam (66). Isso contribui para aumentar a confiança e aceitação dos modelos, além de melhorar a interação humano-computador (65). Por outro lado, a interpretação *ex post facto* é aplicada após o modelo de AM tomar uma decisão, utilizando

métodos que explicam tanto o processo quanto os resultados, ajudando o usuário a entender o raciocínio por trás da decisão e, assim, fortalecendo a confiança no modelo.

As abordagens de explicação *ex post facto* podem ser aplicadas de duas formas principais: diretamente pelos próprios métodos dos classificadores, como RF, XGB e CB, para analisar a importância dos atributos para os modelos de AM; ou por meio de modelos agnósticos, que independem do classificador, como o *LIME (Local Interpretable Model-Agnostic Explanations)*(67) e o *SHAP (SHapley Additive exPlanations)*(64).

A análise de importância dos atributos ajuda a identificar quais características têm maior impacto nas previsões dos classificadores, oferecendo uma visão geral da relevância de cada métrica utilizada na detecção de *code smells*. Para calcular a importância dos atributos dos classificadores, tanto RF quanto XGB e CB fornecem uma análise interna sobre a importância das variáveis. RF[1] calcula a importância de variáveis com base na quantidade de redução de impureza (como Gini ou Entropia) ao longo das árvores. Já o XGB e o CB determinam a importância de uma variável com base na frequência com que ela foi usada para divisões ou pela redução de perda (e.g., *log-loss, mean squared error*) que cada variável proporciona.

No caso de CB, para calcular a importância dos atributos foi utilizado o *LossFunctionChange* para o método *get_feature_importance*. Esse tipo de importância de atributo pode ser usado para qualquer modelo, mas é particularmente útil para classificar modelos, onde outros tipos de importância de atributo podem fornecer resultados enganosos. Para cada atributo, o valor representa a diferença entre o valor de perda do modelo com esse atributo e sem ele[2]. Já para XGB, o cálculo da importância dos atributos foi utilizado o valor padrão do parâmetro importance_type, "*weight*", medindo a importância pelo número de vezes que um atributo aparece nos nós das AD.

Tanto LIME quanto SHAP são técnicas de interpretação pós-modelo (*model-agnostic*) que são independentes do tipo de classificador utilizado, permitindo uma análise detalhada das previsões mesmo em modelos complexos como os baseados em comitê. O LIME é um algoritmo que explica previsões locais de um modelo ajustando um modelo interpretável simples (como uma regressão linear) próximo ao ponto de interesse. Ele é útil para entender previsões específicas e para modelos complexos, como RF e *Boosting*. O funcionamento de LIME pode ser resumido em alguns passos principais (67):

- **Seleção de Amostras**: LIME escolhe um conjunto de amostras próximas ao ponto de interesse (a instância a ser interpretada).
- **Perturbação das Amostras**: As amostras são perturbadas, ou seja, são modificadas

[1] <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.feature_importances_>

[2] <https://catboost.ai/en/docs/concepts/fstr#regular-feature-importances___lossfunctionchange>

de pequenas maneiras.

- **Previsões do Modelo**: O modelo original faz previsões para essas amostras perturbadas.
- **Modelo Simples**: Um modelo simples (como uma regressão linear) é ajustado às previsões do modelo original para essas amostras perturbadas.
- **Interpretação Local**: As características que influenciam a previsão do modelo simples são usadas para interpretar a previsão do modelo original para a instância original.

O SHAP aplica uma abordagem da teoria dos jogos para explicar a saída de qualquer modelo de AM, conectando a alocação de crédito ótima com explicações locais usando os valores clássicos de Shapley. O SHAP é especialmente útil para entender o impacto de cada variável tanto em nível global quanto local em qualquer tipo de modelo. O funcionamento de SHAP pode ser resumido em alguns passos principais (64):

- **Valor Base**: previsão média do modelo para todas as instâncias.
- **Conjuntos de Características**: SHAP considera todos os subconjuntos possíveis de características e calcula a contribuição de cada característica para a previsão.
- **Valor de Shapley**: cada característica recebe um valor de Shapley, que representa sua contribuição marginal para a previsão, considerando todas as combinações possíveis de características.
- **Atribuição de Importância**: os valores de Shapley são somados para atribuir uma importância a cada característica.

3 TRABALHOS RELACIONADOS

Foi realizada uma Revisão Sistemática da Literatura (RSL) (35) para examinar empiricamente e sintetizar as práticas e soluções utilizadas para pré-processamento de dados para a detecção de *code smells* e avaliação da gravidade com base em AM, assim como identificar lacunas no estado da arte a fim de sugerir áreas a serem mais investigadas. Para tanto, a revisão foi pautada em identificar os estudos primários na área e os trabalhos relacionados às questões de pesquisa, para sintetizar os dados e responder às referidas questões.

A revisão bibliográfica (Apêndice A) forneceu uma visão geral dos trabalhos que aplicam pré-processamento de dados para a detecção de *code smells* e a avaliação de sua gravidade. Na seção 3.1, são descritos os estudos que empregam AM e técnicas de pré-processamento de dados para detecção e avaliação da gravidade de *code smells*. Na seção 3.2 são apresentados os trabalhos que buscaram detectar *code smells* baseados em modelos de AM. A seção 3.3 oferece uma visão geral dos trabalhos que investigaram a avaliação da gravidade de *code smells* com base em técnicas de AM. Por fim, a seção 3.4 explora pesquisas voltadas para a detecção e avaliação da gravidade de *code smells* utilizando TA.

3.1 Pré-processamento de Dados para Detecção e Avaliação da Gravidade de *Code Smells*

O pré-processamento de dados é um aspecto importante na construção de modelos de AM de alto desempenho, particularmente na detecção e na avaliação de gravidade de *code smells*. Nos últimos anos, a pesquisa tem focado principalmente em duas frentes: técnicas de balanceamento de dados e de seleção de atributos (35). O balanceamento de dados aborda o desafio da natureza desbalanceada dos conjuntos de dados disponíveis, enquanto a seleção de atributos visa identificar as principais características que influenciam a detecção de *code smells*. Ademais, outras abordagens de pré-processamento também foram investigadas além dessas técnicas fundamentais. Em especial, o escalonamento dos dados foi investigado como uma estratégia viável (68). Esses esforços refletem um compromisso em refinar e otimizar os dados de entrada para os modelos, com o objetivo de melhorar o desempenho e a eficácia dos sistemas de detecção de *code smells*.

3.1.1 Balanceamento de Dados

Estudos recentes indicam que o uso de algoritmos como o SMOTE pode produzir resultados ambíguos na detecção de *code smells*. Nesse sentido, Pecorelli et al.(9) decidiram

não utilizar o SMOTE (37), no conjunto de treino, pois testes experimentais indicaram que o uso do SMOTE poderia introduzir um viés nas previsões de *code smells*. Além disso, o SMOTE enfrentou limitações computacionais, o que resultou em um balanceamento incompleto e impacto nas métricas de desempenho do modelo. Em casos onde houve falta de previsões, a interpretação dos resultados foi afetada.

Em contraste, em um estudo subsequente, Pecorelli et al.(36) observaram que o SMOTE gerou um desempenho ideal em determinados modelos, embora com algumas limitações no treinamento. No entanto, a melhoria resultante do balanceamento de dados foi mínima, sugerindo um impacto limitado no desempenho geral. Descobertas semelhantes foram relatadas por Alkharabsheh et al.(13), onde o SMOTE não melhorou a detecção da GC, destacando a complexidade e a dependência de contexto dos efeitos do balanceamento de dados nos resultados dos modelos.

Boutaib et al.(69) observaram que métodos de subamostragem e sobreamostragem poderiam introduzir imprecisões nos dados. Eles abordaram o problema do desbalanceamento no nível do algoritmo, focando em métodos para gerenciar o desbalanceamento durante o processo de treinamento em vez de modificar a distribuição de dados por meio de reamostragem. Contudo, Rao et al.(26) relataram resultados positivos com o uso de SMOTE combinado com seleção de atributos baseada em PCA, alcançando uma acurácia de severidade de 0,99 para o *code smell* LM usando os algoritmos RF e AD. PCA é uma técnica de AM não supervisionado que determina as características mais importantes reduzindo a dimensionalidade dos dados (70). Esse resultado ressalta o potencial do SMOTE para melhorar a acurácia do modelo quando combinado com seleção de atributos, sugerindo um efeito complementar.

Por fim, Yu, Mao e Ye(71) destacaram a influência do número de amostras positivas na eficiência de treinamento e no desempenho de previsão dos modelos. Concluíram que um excesso de dados poderia reduzir a velocidade de convergência e aumentar o tempo de treinamento, sublinhando a importância de um equilíbrio no tamanho das amostras para otimizar a eficiência e o desempenho dos modelos de detecção de *code smells*.

3.1.2 Seleção de Atributos

A seleção de atributos desempenha um papel fundamental na construção de modelos para detecção de *code smells*. A exploração inicial sobre o uso prático de técnicas de seleção de atributos remonta ao trabalho de Amorim et al.(72). O objetivo do estudo era aprimorar o desempenho do algoritmo de AD ao combiná-lo com um Algoritmo Genético (GA) para selecionar métricas de software. GA gera soluções para otimizar um problema de seleção de atributos usando técnicas inspiradas pela evolução natural, como herança, mutação, seleção e cruzamento (73). Esse algoritmo foi aplicado para detectar 12 tipos de *code smells* em quatro projetos *open-source*.

É importante observar, no entanto, que a generalização dos resultados obtidos por Amorim et al.(72) depende da representatividade do software considerado e da qualidade dos dados utilizados. Nesse experimento, a quantidade de dados foi relativamente pequena, o que pode limitar a aplicabilidade dos achados. Esse estudo ressalta a importância de considerar a qualidade, quantidade e representatividade dos dados ao avaliar o impacto das técnicas de seleção de atributos no desempenho de modelos de AM para detecção de *code smells*.

Fontana e Zanoni(10) optaram por uma abordagem simplificada de seleção de atributos, utilizando variância e correlação linear para eliminar mais da metade dos atributos. Embora essa abordagem possa levantar preocupações sobre a confiabilidade dos resultados, o estudo sugeriu o uso da técnica de seleção de atributos *backward*, que permite selecionar atributos com base no desempenho de um classificador específico. Esse tipo de técnica de seleção de atributos em AM, começa com um conjunto completo dos atributos e remove iterativamente aquelas que têm menor importância ou menor contribuição para o modelo.

No mesmo sentido, Dewangan et al.(44) implementaram técnicas de seleção de atributos, incluindo Qui-quadrado. Os resultados indicaram que os algoritmos RF e Regressão Logística (RL) tiveram um desempenho melhor ao utilizar todos os atributos. Entretanto, em um estudo posterior (74), os autores destacaram a sensibilidade do Qui-quadrado a frequências baixas, enfatizando o potencial de erros ao lidar com características onde o valor esperado é inferior a cinco. Além disso, em uma abordagem distinta, Dewangan et al.(15) aplicaram seleção de atributos pelo método qui-quadrado em dados de gravidade de *code smells* sem balanceamento. Embora a seleção de atributos tenha melhorado a acurácia em alguns algoritmos, outros apresentaram desempenho inferior, sugerindo que o balanceamento de dados poderia mitigar o desbalanceamento de classes e melhorar a consistência entre os modelos.

3.1.3 Escalonamento de Dados

O escalonamento de dados tem por finalidade padronizar o intervalo de variáveis independentes ou características do conjunto de dados. Nesse sentido, para otimizar os dados de entradas dos modelos RF, *K-Nearest Neighbors* (KNN), *Naive Bayes* e *Multi-Layer Perceptron* (MLP), Dewangan et al.(44) usaram normalização de dados. Assim, os resultados deste estudo mostraram uma melhora na acurácia com a aplicação das técnicas de seleção de atributos.

Esses achados sugerem que, além da seleção cuidadosa de atributos, a utilização de MC e a normalização de dados são igualmente relevantes na detecção de *code smells*. Isso destaca a natureza holística do desenvolvimento dos modelos, onde fatores como relevância dos atributos, técnicas de comitê e normalização contribuem coletivamente para a precisão

e desempenho dos sistemas de detecção de *code smells*.

No entanto, é importante ressaltar que Fontana e Zanoni(10) experimentaram várias abordagens de normalização nos conjuntos de dados durante a fase inicial de exploração. Apesar dos esforços, não foram observadas diferenças relevantes, especialmente em termos da avaliação da gravidade de *code smells* dos classificadores. Isso destaca possibilidades interessantes para aprimorar os conjuntos de entrada para modelos de AM.

3.2 Detecção de *Code Smell* com AM

A detecção de *code smells* baseada em AM visa identificar e detectar padrões de código que indicam possíveis problemas ou oportunidades de aprimoramento em aspectos como design, legibilidade, manutenibilidade ou desempenho de bases de código. Nesse sentido, contribui para a qualidade do desenvolvimento de software, auxiliando desenvolvedores a identificar e resolver problemas de qualidade de código logo nas fases iniciais do ciclo de desenvolvimento.

Diferente dos métodos tradicionais de detecção de *code smells*, que se baseiam em regras ou heurísticas predefinidas aplicadas ao código para sinalizar possíveis problemas – a maioria dessas abordagens usa análise de programas para comparar métricas com limites empiricamente identificados, que podem ser tendenciosos (9)–, as abordagens baseadas em AM utilizam algoritmos que aprendem autonomamente padrões e características que sugerem a presença de *code smells* (9).

Em geral, esse processo começa pela coleta de trechos de código rotulados manualmente ou de forma automática para indicar se contêm um *code smell* específico (variável dependente). As variáveis independentes costumam incluir métricas de software e informações sintáticas e semânticas do código. A partir disso, selecionam-se técnicas de AM para treinamento e validação dos modelos. A avaliação do desempenho é feita com um conjunto de dados de validação, comumente utilizando métricas como precisão, sensibilidade, F1 e acurácia para verificar a eficácia do modelo na detecção de *code smells* (28).

Inicialmente os estudos utilizavam abordagens baseadas em AM supervisionada para detectar *code smells*. Normalmente, um método supervisionado explora um conjunto de variáveis independentes para determinar o valor de uma variável dependente (i.e, presença ou não de um *code smell* em um trecho de código) usando um único método de AM (19). Nesse sentido, foram empregados uma ampla gama de modelos de AM, e.g., baseados em AD, aprendizes de regras, máquinas de vetores de suporte, redes bayesianas (8, 19, 9, 44), e também baseados em RL e redes neurais (44). Entretanto, com os avanços dos últimos anos das técnicas e algoritmos de AM, métodos mais robustos e abrangentes têm sido empregados para a detecção de *code smells*, incluindo MC, aprendizado profundo (AP) e TA (conforme apresentado na seção 3.3).

MC, como RF, são amplamente valorizados pela sua habilidade em lidar com dados desbalanceados, como evidenciado por Cruz, Santana e Figueiredo(75). Esses métodos têm se destacado em relação a outros classificadores devido à sua robustez. Diferentemente de Fontana et al.(8), que aplicou *Adaboost* uniformemente em todos os classificadores, Cruz, Santana e Figueiredo(75) exploraram algoritmos de *boosting* de forma isolada, o que permitiu um maior controle sobre os resultados.

Para otimizar o desempenho dos classificadores de *code smells*, Sukkasem e Soomlek(76) aplicaram técnicas de otimização de hiperparâmetros, incluindo a otimização Bayesiana, especificamente em AD e RF. Esse estudo destacou que melhorias substanciais podem ser alcançadas nos classificadores baseados em aprendizado supervisionado, o que facilita a detecção precisa de *code smells* em diferentes cenários.

No contexto do *bagging*, Liu et al.(12) descobriram que esta técnica melhora a precisão e o desempenho geral em termos de pontuação F1, embora seu impacto na sensibilidade tenha sido pequeno e inconsistente. Além disso, neste estudo os resultados indicam que a técnica de agregação por *bootstrap* é útil, mas não infalível, dependendo do objetivo de detecção.

Chen et al.(77) sugeriram que o uso de classificadores simples como "alunos fracos" no *AdaBoost* pode reduzir preocupações com *overfitting* em comparação a outros algoritmos de AM. Eles também identificaram que os parâmetros mais cruciais para o *AdaBoost* são a taxa de aprendizado e o número de "alunos fracos", o que evidencia a importância de ajustar esses fatores para otimizar o desempenho.

O estudo de Barbez, Khomh e Gueheneuc(78) utilizou redes neurais convolucionais (CNNs) com inicializações aleatórias, e aplicou uma técnica de *boosting* para melhorar o desempenho final. Embora essa abordagem tenha resultado em melhor desempenho geral, o uso de conjunto de dados manuais pode limitar a generalização dos resultados. Esses conjuntos de dados manuais exigem tempo para produzir *code smells*, restringindo o tamanho da amostra.

Para simplificar a coleta de conjunto de dados apropriados e reduzir o viés de seleção, Luiz, Oliveira e Parreiras(59) propuseram uma configuração positiva/não rotulada (PU), que elimina a necessidade de anotações negativas. Nesse contexto, modelos baseados em *LightGBM* foram destacados como adequados para detecção de *code smells*. Além disso, modelos de *boosting* como XGB e CB também demonstraram bom desempenho em cenários com dados altamente desbalanceados. Porém, a limitação na padronização de nomes das anotações afetou o tamanho do conjunto de dados, o que pode ter impactado o desempenho.

Alkharabsheh et al.(13) conduziram um estudo comparativo para investigar o efeito do balanceamento de dados na acurácia e comportamento dos classificadores, testando

técnicas de *bagging*, *boosting* e RF. Os experimentos indicaram que o balanceamento de dados teve pouco impacto na acurácia, ressaltando a aplicabilidade de técnicas de AM em contextos de software mais reais, onde os dados frequentemente são desbalanceados devido à natureza dos *code smells*.

Portanto, espera-se que o emprego do aprendizado de comitê melhore o desempenho dos modelos de detecção, facilitando assim o processo de refatoração e contribuindo para uma melhoria geral na qualidade do software. De acordo com Kaur e Kaur(79), a utilização do MC auxilia os desenvolvedores na alocação ideal de recursos, levando a economias substanciais de custos no projeto. No entanto, há limitações associadas a essas técnicas: i) diversidade dos "alunos" a partir dos quais os comitês é construído; ii) presença de restrições de memória e tempo ao empregar comitês; iii) desafios na compreensão do processo de aprendizado desses comitês; e iv) complexidade na compreensão da correlação entre os "alunos" usados para criar o comitê.

AP é uma subárea do AM que utiliza redes neurais artificiais com múltiplas camadas (*deep neural networks*) para modelar e compreender padrões complexos e dados de alta dimensão. Diferente das abordagens tradicionais de AM, que requerem engenharia manual de características, o AP permite que o modelo aprenda automaticamente representações hierárquicas dos dados, aprimorando sua capacidade de reconhecer e interpretar informações complexas (33). A aplicação de técnicas de AP na detecção de *code smells* tem sido frequentemente associada à extração de características semânticas relevantes do código-fonte, visando complementar as características estruturais. Essas características semânticas ampliam a compreensão do significado do código e complementam as métricas orientadas a objetos, já amplamente empregadas (80).

Liu, Xu e Zou(80) introduziu a primeira abordagem de detecção baseada em AP, utilizando tanto características textuais quanto métricas de código como entrada. Para a entrada textual, os autores concentraram-se em identificar nomes, aplicando a técnica *word2vector* – técnica de PLN que transforma palavras em vetores numéricos de forma a capturar suas relações semânticas (81) – em conjunto com redes neurais convolucionais (CNN) para detectar o *code smell* FE.

Diferentemente, Hadj-Kacem e Bouassida(82) adotaram uma perspectiva mais ampla ao considerar todo o código-fonte, o que oferece uma riqueza semântica maior. Embora o estudo tenha excluído comentários do código-fonte, o que pode limitar a riqueza semântica capturada, esses comentários poderiam agregar valor, oferecendo um ponto de aprimoramento futuro.

Guo, Shi e Jiang(83) ampliaram o uso de características textuais, incluindo informações do nome do projeto, pacote, classe e método. Os elementos semânticos contidos nos nomes, especialmente os das funções, foram considerados valiosos na detecção de *code smells*. Nomes de métodos, por exemplo, costumam refletir a funcionalidade do código,

e nomes mais descritivos podem apontar para a presença de *code smells* como LM. A abordagem proposta utilizou entradas de informações textuais e métricas, aproveitando-se da semântica para aprimorar a acurácia.

Os resultados experimentais obtidos por Hamdy e Tazy(84) revelaram que três redes de AP superaram três modelos de AM tradicionais, como *Naïve Bayes*, RF e AD. No entanto, eles também observaram que, embora o aumento dos recursos alocados possa favorecer o desempenho, isso não é uma regra. O estudo mostrou melhorias nas redes *Long Short-Term Memory (LSTM)* e *Gated Recurrent Unit* (GRU), mas a performance da CNN foi negativamente afetada com o aumento de complexidade.

De acordo com Ho et al.(85), muitos estudos tratam o código-fonte como texto, aplicando métodos de PLN. No entanto, código-fonte possui diferenças sintáticas em relação à linguagem natural, como estruturas de controle de fluxo aninhadas, que trazem implicações semânticas importantes. Ignorar essas estruturas pode comprometer a correta preservação do significado do código, destacando a importância de capturar essas nuances estruturais para uma detecção precisa de *code smells*.

Esses estudos revelam uma evolução dinâmica na detecção de *code smells*, integrando AP, enriquecimento semântico e análise estrutural. A área se beneficia da combinação intrincada desses elementos, apresentando um caminho promissor para aprimorar a acurácia da detecção e avançar na compreensão das nuances complexas do código-fonte.

3.3 Avaliação da gravidade de *code smell* baseado em AM

O estudo de Fontana e Zanoni(10) focou na avaliação da gravidade de *code smells* utilizando técnicas de aprendizado de máquina aplicadas em diversos experimentos. Foram testados diferentes modelos, abrangendo desde classificação multinomial até regressão, além de um método de adaptação de classificações binárias para classificação ordinal. Eles modelaram a gravidade do *code smell* como uma variável ordinal, expandindo os modelos propostos por Fontana et al.(8), onde resultados promissores foram obtidos com o uso de modelos de classificação binária para detecção de *code smells*.

O estudo de Nanda e Chhabra(14) realizou uma análise detalhada e correção dos conjuntos de dados utilizados por Fontana e Zanoni(10), eliminando inconsistências nos dados relacionados a GC e DC. Essas intervenções melhoraram o desempenho das técnicas de AM aplicadas para a classificação da gravidade dos *code smells*. Em seguida, propuseram a abordagem SSHM (*Synthetic Minority Over-sampling Technique - SMOTE and Stacking in combination*), aplicando-a na classificação da gravidade para quatro tipos de *code smells*, i.e., GC, DC, FE e LM.

O trabalho de Abdou e Darwish(24) focou na avaliação de modelos de classificação

para gravidade de *code smells* após a aplicação da técnica de sobreamostragem SMOTE. Adicionalmente, o estudo incluiu uma análise comparativa para selecionar os melhores modelos e a extração de regras de detecção, visando examinar a eficácia das métricas de software na identificação de *code smells*.

No estudo de Hu et al.(27), foram avaliados 21 modelos de predição para estimar a gravidade de *code smells*, incluindo 10 métodos de classificação e 11 algoritmos de regressão. As principais métricas de desempenho utilizadas foram a ferramenta visual usada para avaliar a eficácia de um modelo preditivo –*Cumulative Lift Chart* (CLC)– e o *Severity@20%*, que mede a porcentagem da gravidade total nas 20 principais instâncias previstas. A acurácia foi considerada uma métrica secundária.

Rao et al.(26) aplicaram a técnica SMOTE para lidar com o desequilíbrio de classe e usaram Análise de Componentes Principais (PCA) para seleção de atributos, visando melhorar a precisão. Enquanto Dewangan et al.(15) adotaram modelos de comitê de AM, utilizando seleção de atributos baseada no teste qui-quadrado em cada conjunto de dados, e avaliaram o impacto da otimização de hiperparâmetros nos resultados de classificação para gravidade de *code smells*.

3.4 TA para detecção e avaliação da gravidade de *code smells*

A pesquisa sobre TA aplicada à detecção de *code smells* ainda é limitada, com poucos estudos explorando o tema. Quanto à avaliação da gravidade, até onde se sabe, não foram encontrados estudos a esse respeito. Sharma et al.(53) mostraram que é possível empregar AP e TA para identificar características de qualidade no código sem o uso de métricas derivadas, sugerindo um caminho para ferramentas de detecção em linguagens com poucos recursos de análise. Mais especificamente, Sharma et al.(53) investigaram a eficácia dos métodos AP na detecção de *code smells* em projetos Java e C# com TA, explorando diferentes arquiteturas de redes neurais. As Redes Neurais Convolucionais (CNNs) foram utilizadas para identificar padrões locais em dados estruturados, enquanto as Redes Neurais Recorrentes (RNNs) foram aplicadas para lidar com dados sequenciais, como a análise de dependências em código. Além disso, *autoencoders*, que são redes neurais projetadas para aprender representações compactas dos dados, foram empregados para explorar características latentes associadas a *code smells*. Ao final, esse estudo indicou que futuras abordagens poderiam considerar o contexto e as opiniões dos desenvolvedores para aumentar a precisão.

Kovacevic et al.(86) diferenciaram-se ao focar na transferência de conhecimento de modelos de compreensão de código, utilizando um conjunto de dados rotulado manualmente para uma avaliação prática mais confiável. Já Stefano et al.(87) exploraram TA entre diferentes projetos, embora os resultados iniciais não tenham demonstrado benefícios claros.

Essa pesquisa, no entanto, sugere que a abordagem pode ter potencial em estudos futuros.

Ren, Shi e Zhao(88) propuseram uma solução para a confiabilidade dos conjuntos de dados, combinando dados sintéticos para pré-treinamento com dados reais para *fine-tuning* –que consiste em ajustar um modelo pré-treinado com dados específicos para melhorar seu desempenho em uma tarefa específica –, o que melhorou a precisão na detecção de GC. Gupta e Singh(89) exploraram a TA para lidar com dados heterogêneos, mostrando como a técnica pode ser útil quando os dados de treino e teste têm características distintas. Finalmente, Ma et al.(90) utilizaram modelos pré-treinados, incluindo CodeT5, para detectar FE, baseando-se em relações semânticas entre métodos e classes, o que promoveu maior coesão no código. CodeT5 (91) é um modelo codificador-decodificador pré-treinado inspirado na arquitetura T5 do campo de PLN (92) que demonstrou beneficiar tarefas de compreensão e geração de linguagem natural.

O estudo de Kovacevic et al.(93) investigou a eficácia das representações de código geradas pelo *CodeT5* para a detecção automática de *code smells*. Eles combinaram métricas de código-fonte com as representações do *CodeT5* para treinar modelos de AM, comparando o desempenho dessas abordagens baseadas em AM com heurísticas tradicionais para detecção de *code smells*. Nos experimentos, ambas as abordagens de AM superaram as heurísticas de detecção.

O classificador de AM treinado com métricas de código-fonte isoladas de Kovacevic et al.(93) apresentou melhor desempenho em relação ao modelo baseado nas representações do *CodeT5* nas tarefas LM e *Large Class* (a.k.a GC), atingindo pontuação F1 de 0,87 e 0,91, respectivamente. Esse resultado diverge dos achados anteriores de Kovacevic et al.(86), nos quais as representações *embeddings* (representações vetoriais de objetos em um espaço contínuo de alta dimensão, onde a semelhança entre esses objetos é preservada) de código do CuBERT (94) capturaram melhor a semântica necessária para a detecção de *code smells*. O CuBERT é um modelo pré-treinado baseado na arquitetura BERT (*Bidirectional Encoder Representations from Transformers*) adaptado especificamente para a análise de código-fonte. Ele foi treinado em um corpus de código-fonte de várias linguagens de programação, como Python, JavaScript, Java, C++, entre outras.

Eles atribuíram essa diferença aos conjuntos de dados utilizados. No estudo de Kovacevic et al.(93), o conjunto de dados de Slivka et al.(11) foi rotulado utilizando um procedimento de anotação rigoroso, garantindo alta concordância entre anotadores. Em contraste, o estudo de Kovacevic et al.(86) utilizou o conjunto de dados MLCQ (95), que visa capturar diferentes perspectivas de desenvolvedores sobre *code smells*, mas que apresenta rótulos inconsistentes devido às variações na percepção dos anotadores sobre esses *code smells*.

3.5 Estado da Arte: Limitações e Estratégias de Melhoria

Esta seção apresenta uma síntese dos principais trabalhos relacionados à detecção e avaliação da gravidade de *code smells* com base em AM, conforme a Tabela 1. A seguir, discute as principais limitações observadas nos processos de pré-processamento de dados, detecção e avaliação de gravidade de *code smells*, bem como no uso de TA nesses contextos. Também são propostas estratégias para superação dessas limitações, destacando as contribuições deste trabalho.

3.5.1 Pré-processamento de Dados

O pré-processamento de dados desempenha um papel importante na detecção de *code smells*, envolvendo uma interação complexa entre técnicas, algoritmos e as especificidades dos conjuntos de dados. Estratégias adaptadas às características desses dados são fundamentais para avanços neste campo de pesquisa. Contudo, algumas limitações têm sido identificadas na literatura.

Uma das principais limitações é o balanceamento de dados. Técnicas como o SMOTE nem sempre resultam em melhorias consistentes no desempenho dos modelos de AM (9, 13). Além disso, SMOTE pode introduzir vieses, como o vazamento de dados do conjunto de treinamento para o teste, ou falhar na criação de equilíbrio adequado devido a limitações computacionais ou características específicas dos dados (26, 36). Neste trabalho, essas limitações foram abordadas com o uso de B-SMOTE, uma variação do SMOTE que prioriza amostras de borda para melhorar a separação entre classes. Essa técnica foi aplicada apenas nos conjuntos de treinamento, especialmente no contexto de avaliação de gravidade de *code smells*, evitando o vazamento de dados.

A seleção de atributos também apresenta desafios, como a sensibilidade de técnicas estatísticas (e.g., qui-quadrado) a baixas frequências, que podem resultar em erros na análise (44). Para mitigar esses problemas, foi utilizada a análise ANOVA, especialmente em dados padronizados, buscando melhorar a representatividade dos atributos e reduzir a influência de *outliers*.

Por fim, o escalonamento de dados, embora amplamente utilizado, nem sempre resulta em melhorias na eficácia dos modelos (10, 15). Este trabalho adota a padronização como alternativa à normalização, ajustando os valores para escalas comparáveis e reduzindo a influência de *outliers*, visando aumentar o desempenho dos modelos de AM (33).

Tabela 1 – Síntese do Estado da Arte para Detecção e Avaliação de Gravidade de *code smell* baseado em AM

Estudo	**Balanc. Dados**	**Seleção de Atributos**	**Escalon. Dados**	Escopo Pré-proc.	**MC**	TA	Domínio	Objetivo	Validação dos Dados
Fontana e Zanoni(10)	-	Filtro de baixa variância e de alta correlação linear	Norm	Completo	*Adaboost*, RF	-	Java	AvGrav	Especialista
Nanda e Chhabra(14)	SMOTE	-	-	Completo	*Adaboost*, RF	-	Java	AvGrav	Especialista
Abdou e Darwish(24)	SMOTE	Ganho de Informação	-	Treino	Boosting e RF	-	Java	AvGrav	Especialista
Hu et al.(27)	-	-	-	-	*Adaboost*, *Bagging*, RF, XGB	-	Java	AvGrav	Especialista
Rao et al.(26)	SMOTE	PCA	Norm	Completo	RF	Não	Java	AvGrav	Especialista
Dewangan et al.(15)	-	Qui-quadrado	Norm	-	*Adaboost*, GB, RF, XGB	Não	Java	AvGrav	Especialista
Sharma et al.(53)	Subamostragem	*Autoencoder*	-	Treino	-	AP (CNN e RNN), entre domínios, com dados semânticos do código	Java e C#	DetCS	Não validado
Kovacevic et al.(93)	SMOTE e SMOTEENN -	-	Padr	Treino	*Bagging*, RF, GB e CB	CodeT5 com dados estruturais e semântico do código	C#	DetCS	Especialista
Abordagem Proposta	SMOTE e B-SMOTE	Qui-quadrado e ANOVA	Norm e Padr	Treino	RF, XGB e CB	MC em duas etapas, entre domínios, com dados estruturais do código	Java e C#	DetCS e AvGrav	Especialista

3.5.2 Detecção de *Code Smells*

A detecção de *code smells* por AM enfrenta limitações como a necessidade de grandes volumes de dados rotulados e altos custos computacionais (13, 75), principalmente para AP. Além disso, a interpretação das decisões de modelos de AP continua sendo um desafio, devido à complexidade das camadas de processamento envolvidas.

Para contornar esses problemas, este estudo adotou MC, como RF, XGB e CB, que demandam menos recursos computacionais e apresentam resultados promissores tanto para detecção de *code smells* quanto para avaliação de gravidade. A explicabilidade dos modelos foi abordada com técnicas como análise de importância de atributos, LIME e SHAP, que identificam os fatores mais influentes nos resultados, aumentando a confiança nos modelos. Além disso, a otimização de hiperparâmetros foi realizada com buscas randomizada e bayesiana, permitindo explorar combinações de parâmetros de forma eficiente (68, 96).

3.5.3 Avaliação de Gravidade de *Code Smells*

A avaliação da gravidade de *code smells* apresenta desafios relacionados ao desbalanceamento de classes e à ausência de comparação entre métodos de escalonamento de dados (10, 14, 26). Muitos estudos avaliaram a gravidade de *code smells* em conjuntos de dados de apenas um tipo específico, ignorando a interação entre diferentes tipos de *code smells*, o que se afasta de cenários reais de desenvolvimento de software.

Para mitigar esses problemas, técnicas de sobreamostragem como SMOTE e B-SMOTE foram aplicadas, apenas nos conjuntos de treinamento, tanto na detecção quanto na avaliação de gravidade. Além disso, este trabalho utilizou conjuntos de dados combinados, abrangendo múltiplos tipos e gravidades de *code smells*, para Java (68) e C# (11, 22). Essa abordagem representa mais fielmente o desafio encontrado em ambientes reais, permitindo uma análise mais abrangente.

Adicionalmente, foi adotada a abordagem de duas etapas proposta por Santos, Duarte e Choren(68) para detectar e avaliar a gravidade de *code smells*. Essa abordagem, integrada ao fluxo metodológico deste trabalho consiste em:

1. **Detecção binária de *code smells***: Primeiramente, realiza-se a detecção binária de *code smells* (com e sem *code smell*). Esta etapa inicial é determinante para filtrar as instâncias que realmente apresentam *code smells*, concentrando a análise subsequente apenas nos casos positivos, com presença de *code smell*.

2. **Avaliação de gravidade multiclasse nas instâncias positivas**: Uma vez identificadas as instâncias com *code smells*, a segunda etapa foca na avaliação da gravidade desses *code smells* apenas nas instâncias positivas. Isso permite uma análise mais de-

talhada e precisa da gravidade dos problemas encontrados, utilizando uma abordagem multiclasse.

Essa metodologia em duas etapas não apenas simplifica o processo de detecção, mas também otimiza os recursos computacionais ao focar a análise de gravidade apenas nas instâncias pertinentes. Ao combinar técnicas de escalonamento de dados, seleção de atributos, sobreamostragem dos dados e otimização de hiperparâmetros, os modelos de AM construídos são mais robustos e precisos, refletindo melhor os desafios encontrados em ambientes reais de desenvolvimento de software.

3.5.4 Transferência de Aprendizado para Detecção e Avaliação da Gravidade de *Code Smells*

A TA aplicada à detecção de *code smells* enfrenta limitações como a falta de padronização nos conjuntos de dados rotulados, divergências entre abordagens baseadas em *embeddings* (86, 93), introdução de viés quando os dados reais para ajuste fino não são representativos o suficiente (88) e a adaptação limitada a diferentes contextos de software (87). Além disso, até onde se sabe, não foram encontrados trabalhos que apliquem TA para avaliação de gravidade.

Neste trabalho, a TA foi utilizada para aprimorar tanto a detecção quanto a avaliação da gravidade de *code smells*, aproveitando dois conjuntos de dados oriundos de diferentes linguagens de programação: Java e C#. O domínio de origem (Java) foi empregado para transferir conhecimento ao domínio de destino (C#), explorando similaridades nas tarefas e nos atributos preditores. Para assegurar a consistência entre os domínios, os atributos preditores do conjunto de dados de C# foram transformados para se alinhar aos do conjunto de dados Java, criando uma configuração homogênea. Nesse cenário, ambos os domínios compartilham o mesmo espaço de atributos ($X_S = X_T$) e de rótulos ($Y_S = Y_T$).

Dessa forma, a abordagem de TA adotada pode ser classificada como:

- **Transferência Indutiva**, devido à presença de rótulos nos domínios de origem e destino;
- **Transferência Homogênea**, graças à correspondência entre os atributos preditores;
- **Baseada em Atributos**, uma vez que os atributos do conjunto de dados de C# foram modificados para garantir uma representação comum, facilitando a transferência de conhecimento entre os domínios.

Essa abordagem possibilitou uma análise mais robusta e alinhada com cenários reais, permitindo investigar instâncias de *code smells* em diferentes linguagens de programação.

Este trabalho se diferencia de estudos como o de Sharma et al.(53), que aplicaram TA para detecção de *code smells* sem o uso de métricas de software, e de Kovacevic et al.(93), que exploraram *embeddings* pré-treinados para extração de recursos em C#.

4 CONJUNTOS DE DADOS

Dentre as contribuições deste trabalho está a criação dos conjuntos de dados Java e C#. Os conjuntos Java são maioria para o estudo de *code smell* (16, 11), porém, os estudos de pesquisa anteriores (10, 14, 24, 27, 26, 15) visavam principalmente avaliar a gravidade dos *code smells* em conjuntos de dados que continham instâncias de apenas um tipo específico de *code smell.* Assim, esse foco estreito separa essas abordagens de situações mais realistas no desenvolvimento de software, ignorando a influência entre diferentes tipos de *code smells* uns sobre os outros.

Os conjuntos de dados Java originais utilizados nesta pesquisa estão disponíveis em *Evolution of Software Systems and Reverse Engineering (ESSeRE Lab)*[1] –Fontana e Zanoni(10) e *Google Drive*[2] –Nanda e Chhabra(14). Os conjuntos de dados em C# de Slivka et al.(11)[3] e Prokic et al.(22)[4] podem ser encontrados em Zenodo.

Cada conjunto de dados inclui um valor de gravidade de *code smell* em quatro níveis, definido por Fontana e Zanoni(10) da seguinte forma:

0 - Sem *code smell:* a classe ou método está isento de quaisquer características de *code smells.* Exemplo: um método de 10 linhas que calcula o total de produtos vendidos e, ao final, grava o resultado em um arquivo; a responsabilidade adicional é pequena, portanto, não representa um problema (11).

1 - *Code smell* não grave: a classe ou método apresenta características de *code smell*, mas de forma leve. Exemplo: um método de 30 linhas com três blocos de código independentes, cada um desempenhando uma responsabilidade que pode ser extraída (11).

2 - *Code smell*: as características do *code smell* estão todas presentes e impactam no entendimento da classe ou método. Exemplo: um método com 100 linhas, estruturas de controle aninhadas e funcionalidades variadas, exigindo refatoração do código (11).

3 - *Code smell* grave: o *code smell* está fortemente presente, com valores elevados de tamanho, complexidade ou acoplamento. Exemplo: um método com centenas de linhas, múltiplas estruturas aninhadas e responsabilidades extensivas, tornando-o difícil de compreender e propenso a efeitos colaterais (11).

---

[1] <http://essere.disco.unimib.it/reverse/MLCSD.html>
[2] <https://drive.google.com/drive/folders/16BqUdNlKNgdM_qrrJqGWKdQ_NfEdRPVD?usp=sharing>
[3] <https://zenodo.org/records/10475432>
[4] <https://zenodo.org/records/6520056>

4.1 O Conjunto de Dados Java

Os conjuntos de dados Java originais utilizados neste trabalho são o de Fontana e Zanoni(10) e Nanda e Chhabra(14). Estes conjuntos de dados foram escolhidos pois são bem estabelecidos na literatura e focam nos *code smells* apresentados na seção 2.1.

Para detectar e avaliar a gravidade de *code smells*, os autores empregaram, no estudo de Fontana e Zanoni(10), 76 sistemas do corpus *Qualitas Corpus* (versão 20120401r), desenvolvido por Tempero et al.(97). Esse conjunto foi avaliado considerando um amplo conjunto de métricas de orientação a objetos, com *code smells* de GC, DC, FE e LM identificados por ferramentas e métodos, "conselheiros", como *iPlasma* (98), *Fluid Tool* (99), *JSpIRIT* (100), PMD (98) e regras de detecção de Marinescu(101). A Tabela 2 apresenta as ferramentas de detecção automática utilizadas no estudo.

Tabela 2 – Ferramenta de detecção automática ("Conselheiros")

Classe	"Conselheiros"
DC	*iPlasma*
GC	*iPlasma, JSpIRIT, PMD*
FE	*iPlasma, Fluid Tool*
LM	*iPlasma (Brain Method)*, *PMD*, Regras de Detecção de (101)

No estudo conduzido por Nanda e Chhabra(14), a análise dos conjuntos de dados fornecidos por Fontana e Zanoni(10) revelou discrepâncias entre os conjuntos de dados binários e multinomiais de GC e DC. Essas discrepâncias foram corrigidas e os demais casos foram avaliados quanto a erros de classificação, exigindo ajustes na gravidade quando necessário. Vários "conselheiros" foram empregados durante esta fase de correção: *iPlasma* (98), *JSpIRIT* (100) e PMD[5] para GC, enquanto a ferramenta *iPlasma* foi usada para DC; e a decisão final e os rótulos de gravidade foram atribuídos com base na avaliação de especialistas. A Tabela 3(14) resume as diferentes correções realizadas, indicando os motivos de cada ajuste.

Tabela 3 – Detalhes das instâncias corrigidas de GC e DC

Motivo	*God-class*	*Data-class*
A. Conflito entre conjunto de dados	128	129
B. Classificado incorretamente como instância negativa	5	12
C. Classificado incorretamente como instância positiva	6	6
D. Reclassificação de gravidade	18	20
E. Mudanças não conflitantes (B+C+D)	29	38
Total (A + E)	157	167

[5] https://github.com/pmd/pmd

Os conjuntos de dados, com 420 instâncias cada, representam classes ou métodos e são descritos por vetores de características, atributos, contendo 63 valores para GC e DC, e 84 para LM e FE.

4.1.1 Combinação dos Conjuntos de Dados Java

Para os conjuntos de dados originais Java, o método de seleção de segmentos de código, resultou em uma distribuição de classes considerada irreal (11). Essa distribuição distorcida pode comprometer a capacidade de generalização dos modelos de AM treinados com esses conjuntos de dados (19). Mais especificamente, a composição das instâncias nos conjuntos de dados pode influenciar no desempenho do modelo e difere de um ambiente de projeto de software realista (102). Neste contexto, a combinação dos conjuntos de dados Java visa adaptar os dados para técnicas de AM aplicáveis a cenários reais, onde dados desbalanceados são comuns devido à natureza dos *code smells* (9). Para representar melhor os modelos de avaliação de gravidade, foi necessário criar um novo conjunto de dados com instâncias de diferentes gravidades e tipos de *code smells*.

Os vetores de características, atributos, para GC e DC foram expandidos para incluir os 84 valores presentes em FE e LM, resultando em valores ausentes para algumas métricas de método (NOP=1, CC=0, ATFD=0, FDP=0, CM=1, MAXNESTING=1, LOC=3, CYCLO=1, NMCS=0, NOLV=1, MaMCL=0, NOAV=1, LAA=1, FANOUT=0, CFNAMM=0, ATLD=0, CLNAMM=0, CINT=0, MeMCL=0, CDISP=0) nas instâncias GC e DC. Esses valores foram imputados com o valor de moda, pois os valores de amostra para cada atributo não seguem uma distribuição normal. O valor de moda foi calculado considerando apenas instâncias sem *code smell* para as classes FE e LM. Além disso, o atributo "is_Static_method" recebeu um valor zero em cada instância de GC e DC. A hipótese subjacente é que os modelos de AM interpretarão que estes casos não são de instâncias FE ou LM e com os outros valores independentes relacionados a GC e DC, os modelos serão capazes de classificá-los corretamente.

A Figura 1 ilustra os conjuntos de dados de Fontana e Zanoni(10) e Nanda e Chhabra(14), que inicialmente contém 420 instâncias cada. Essas instâncias são classificadas em ordem crescente por atributos: "project", "package", "complex_type" e "method". No primeiro estágio do processo, cada par de conjuntos de dados de nível de classe (GC e DC) e nível de método (FE, LM) são combinados. Para isso, foi seguido os passos do Algoritmo 1. Este algoritmo remove instâncias duplicadas de nível de método ou classe com diferentes gravidades, removendo a menor gravidade. Além disso, no caso de instâncias com gravidades iguais, instâncias das classes LM e GC são priorizadas sobre FE e DC, respectivamente, uma vez que são códigos que têm o maior impacto nos atributos de qualidade e propensão a erros (103), além de serem consideradas as mais problemáticas pelos desenvolvedores (104).

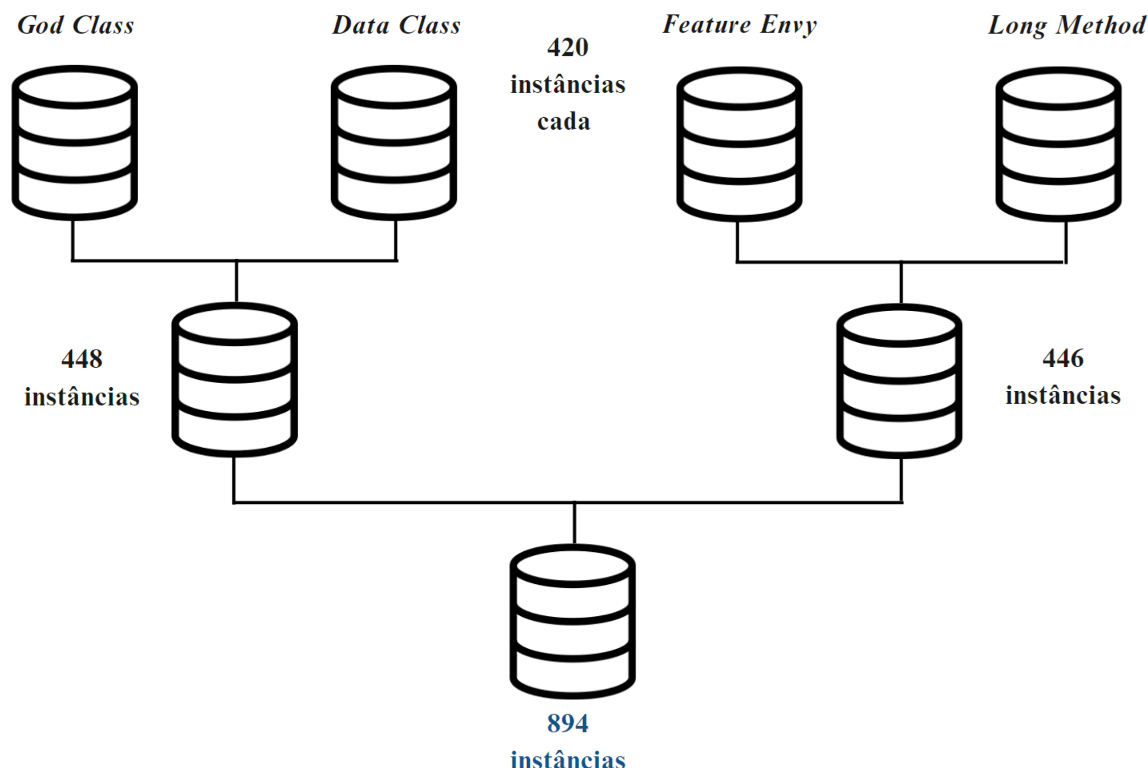


Figura 1 – Visão geral do processo de combinação do conjunto de dados Java

Após a execução completa do algoritmo, os subconjuntos de dados em nível de classe (com 448 instâncias) e em nível de método (com 446 instâncias) são combinados para formar um conjunto de dados final composto por 894 instâncias. O algoritmo é capaz de gerar até $((3 * NCS) + 1)$ classes distintas de gravidade de *code smell*, com uma delas representando a classe "sem *code smell*". No caso específico deste estudo, foram geradas 13 classes, uma vez que o número de *code smells* (NCS) distintos (GC, DC, FE e LM) é 4.

A Tabela 4 apresenta os detalhes das classes geradas, incluindo o número de instâncias e a distribuição percentual de cada uma, categorizadas por gravidade. É relevante observar que as instâncias negativas, sem *code smell*, representam 46,20% do conjunto de dados combinado, enquanto as instâncias positivas, com *code smell*, correspondem a 53,80%. Contudo, alguns atributos do conjunto de dados apresentam *outliers* consideráveis, que podem enviesar os resultados dos modelos de AM (68). Para melhorar o equilíbrio entre as classes, foram ajustadas a proporção de instâncias negativas e positivas, excluindo algumas instâncias com *outliers* muito discrepantes das classes de gravidade 5, 8 e 12 (LM, DC e GC Grave - as classes com mais instâncias de gravidade). Foram eliminadas 23, 20 e 25 instâncias dessas classes, respectivamente. Esse ajuste visa estabelecer uma proporção de 1:1 entre instâncias negativas e positivas no conjunto de dados (413 instâncias em cada categoria).

Algorithm 1 Algoritmo de Combinação de Conjuntos de Dados

```
1:  while Houver instâncias a serem analisadas do
2:      Procurar por instâncias duplicadas (no nível de classe ou método)
3:      if Instâncias duplicadas encontradas são encontradas then
4:          if Os níveis de gravidade dos casos são diferentes then
5:              Excluir a instância de menor gravidade
6:              if A gravidade é diferente de sem code smell then
7:                  Alterar o nome da classe da instância remanescente para “gravidade + code
                    smell
8:              end if
9:          else if A gravidade é diferente de sem code smell then
10:             Excluir a instância de FE ou DC
11:             Alterar o nome da classe da instância remanescente para “gravidade + GC ou
                LM
12:         else
13:             Excluir qualquer uma das duas instâncias
14:         end if
15:     else if A gravidade é diferente de sem code smell then
16:         Alterar o nome da classe da instância para “gravidade + code smell
17:     end if
18: end while
```

Tabela 4 – Detalhamento das classes geradas - Conjunto de Dados Java

Code Smell	ID Gravidade	Nome	Instâncias Originais (% gravidades)	Instâncias Combinadas (% gravidades)
0	0	Sem *Code Smell*	1115	413
1	1	FE não Grave	23 (4.07)	13 (3.15)
1	2	FE	95 (16.81)	39 (9.44)
1	3	FE Grave	22 (3.89)	11 (2.66)
1	4	LM não Grave	11 (1.95)	8 (1.94)
1	5	LM	95 (16.81)	67 (16.22)
1	6	LM Grave	34 (6.02)	34 (8.23)
1	7	DC não Grave	32 (5.66)	32 (7.75)
1	8	DC	77 (13.63)	57 (13.80)
1	9	DC Grave	37 (6.55)	37 (8.96)
1	10	GC não Grave	9 (1.59)	9 (2.18)
1	11	GC	34 (6.02)	33 (7.99)
1	12	GC Grave	96 (16.99)	73 (17.68)

4.2 O Conjunto de Dados C#

Embora o C# seja amplamente utilizado no desenvolvimento e os desenvolvedores estejam cada vez mais conscientes de *code smells* nessa linguagem (17), a maioria dos conjuntos de dados anotados manualmente ainda concentra-se em instâncias de código de projetos Java (16). Este trabalho propôs, então, a criação de um conjunto de dados de *code*

smell específico para C#, utilizando uma abordagem semelhante à aplicada na construção de conjuntos de dados Java. O novo conjunto de dados combina dois conjuntos previamente apresentados em Slivka et al.(11) e Prokic et al.(22). Slivka et al.(11) aplicaram uma metodologia de anotação prescritiva de PLN para identificar os *code smells* LM e GC. A partir de 8 projetos C# de código aberto, eles construíram um conjunto de dados de tamanho médio contendo 2.574 instâncias anotadas de *code smells* LM e 920 de *code smells* GC. Em Slivka et al.(11), cada instância do conjunto de dados foi rotulada manualmente, envolvendo múltiplos anotadores para garantir precisão. Além disso, a proporção entre instâncias com e sem *code smell* foi mantida para refletir o equilíbrio observado em projetos de software reais.

A pesquisa apresentada em Prokic et al.(22) representa uma extensão dos trabalhos de Slivka et al.(11), com o objetivo de expandir os resultados originais e estabelecer um procedimento prescritivo para a anotação manual de *code smells* que possa ser aplicado a qualquer tipo de *code smell* definido por Fowler et al.(1). Esse procedimento visa auxiliar pesquisadores em AM na construção de modelos para a detecção de *code smells*, ferramentas que engenheiros de software podem utilizar para aprimorar a manutenção de software. Nesse contexto, os autores desenvolveram uma ferramenta que facilita o trabalho dos anotadores. Três anotadores utilizaram essa ferramenta para expandir o conjunto de dados original, que continha *code smells* LM e GC, adicionando também as categorias de *code smells* DC, FE e *Refused Bequest*, sendo que as instâncias do último *code smell* foram desconsideradas neste trabalho. Foram anotadas 231 instâncias de *code smell* DC e 220 instâncias de *code smell* FE.

4.2.1 Combinação dos Conjuntos de Dados C#

O processo de combinação dos conjuntos de dados C# começou com a exclusão de todas as métricas de software (preditores desses conjuntos de dados). Em seguida, foram utilizados os principais atributos selecionados do conjunto de dados balanceado Java para realizar a análise dos trechos de código. Essa análise foi feita com a ajuda de analisadores da plataforma do compilador .NET (Roslyn)[6], que calcularam as métricas para os projetos de software C# relacionados às instâncias de *code smells* descritas por Slivka et al.(11) e Prokic et al.(22). Os diagramas de classes dos Analisadores de Métricas de Código utilizados neste trabalho estão disponíveis no Apêndice G. Esses analisadores inspecionam o código C# ou *Visual Basic* em busca de problemas de segurança, desempenho, design e outros. Finalmente, os novos preditores gerados foram incluídos no conjunto de dados C# anotado, resultando na combinação dos quatro conjuntos de dados em um único conjunto que abrange todas as instâncias de gravidade de *code smell* apresentados na seção 2.1, exceto por DC e FE Graves, que não tem instâncias rotuladas. A Tabela 5 apresenta a

[6] <https://github.com/fabiorosario/MetricsAnalyzer>

distribuição dessas instâncias no conjunto de dados combinado.

Tabela 5 – Detalhamento das classes geradas - Conjunto de Dados C#

Code Smell	ID Gravidade	Nome	Instâncias Originais (% gravidades)	Instâncias Combinadas (% gravidades)
0	0	Sem *Code Smell*	6097	2301
1	1	FE não Grave	28 (1.10)	26 (3.30)
1	2	FE	8 (0.31)	7 (0.89)
1	3	FE Grave	0 (0.00)	0 (0.00)
1	4	LM não Grave	1243 (48.63)	335 (42.51)
1	5	LM	498 (19.48)	150 (19.04)
1	6	LM Grave	131 (5.13)	41 (5.20)
1	7	DC não Grave	12 (0.47)	11 (1.40)
1	8	DC	3 (0.12)	2 (0.25)
1	9	DC Grave	0 (0.00)	0 (0.00)
1	10	GC não Grave	393 (15.38)	124 (15.74)
1	11	GC	174 (6.81)	64 (8.12)
1	12	GC Grave	66 (2.58)	28 (3.55)

As instâncias negativas, que não apresentam *code smell*, constituem 74,49% do conjunto de dados combinado, enquanto as instâncias positivas, que contêm *code smell*, totalizam 25,51%. Além disso, é importante observar que as instâncias não graves de LM (335) e GC (124) têm mais registros do que o número de instâncias no conjunto de dados Java, que possui apenas 8 e 9, respectivamente (conforme a Tabela 4). A Tabela 6 demonstra que o processo de combinação que foi empregado recuperou uma porcentagem considerável de instâncias do conjunto de dados original, permitindo a utilização do conjunto gerado sem comprometer suas características de amostra originais.

Tabela 6 – Instâncias recuperadas dos conjuntos de dados C# originais

Code Smell	Instâncias Originais	Instâncias Recuperadas	% Instâncias Recuperadas
Long-method	2574	1943	75.49
God-class	920	786	85.43
Data-class	231	181	78.35
Feature-envy	220	179	81.36

5 O MÉTODO DE AM PROPOSTO PARA DETECÇÃO E AVALIAÇÃO DA GRAVIDADE DE *CODE SMELLS*

Este trabalho propõe uma abordagem para a detecção de *code smells* e a avaliação de sua gravidade em cenários com conjuntos de dados desbalanceados. A ideia principal é utilizar MCs para detectar *code smells* (tanto instâncias positivas quanto negativas) e avaliar a gravidade dos *code smells*, baseando-se apenas nas instâncias positivas detectadas anteriormente.

A abordagem é dividida em quatro tarefas. A primeira tarefa consiste no pré-processamento de dados, que envolve a seleção dos atributos mais importantes do conjunto de dados Java, com ou sem a aplicação de escalonamento de dados. Em seguida, foi realizada a otimização de hiperparâmetros, utilizando técnicas de otimização randomizada ou bayesiana, além de sobreamostragem de dados e seleção de atributos.

As terceira e quarta tarefas são responsáveis pela detecção de *code smells* e pela avaliação de sua gravidade, utilizando MC (RF, XGB ou CB), além do modelo-base desses algoritmos, AD. A Figura 2 ilustra a visão geral da abordagem proposta.

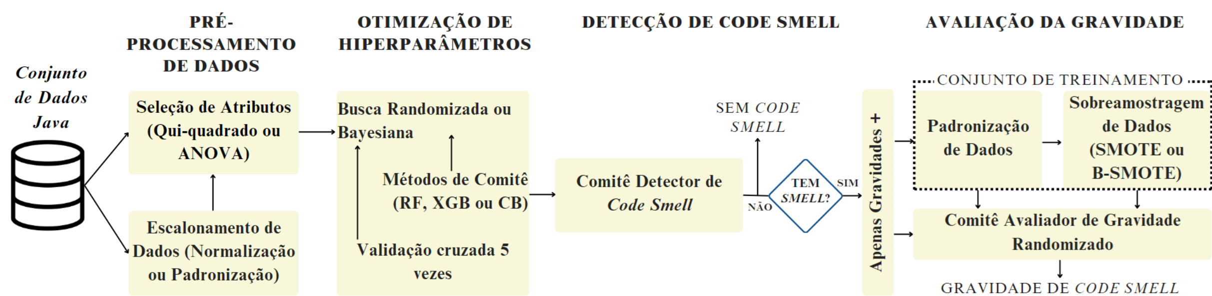


Figura 2 – Visão geral da abordagem proposta

5.1 Pré-processamento de Dados

A tarefa de pré-processamento de dados (Figura 3) é um aspecto importante na construção de modelos de AM mais precisos, pois envolve a limpeza e a transformação dos dados brutos. Essa etapa facilita a análise e a identificação das informações mais relevantes, contribuindo assim para a melhoria do desempenho do modelo (34).

Antes de aplicar outras técnicas de pré-processamento ao conjunto de dados Java combinado, que inclui rótulos e métricas como atributos, é essencial abordar questões específicas, como a presença de dados duplicados e valores ausentes. Além disso, a análise exploratória do conjunto de dados mostrou que os intervalos de valores entre os atributos

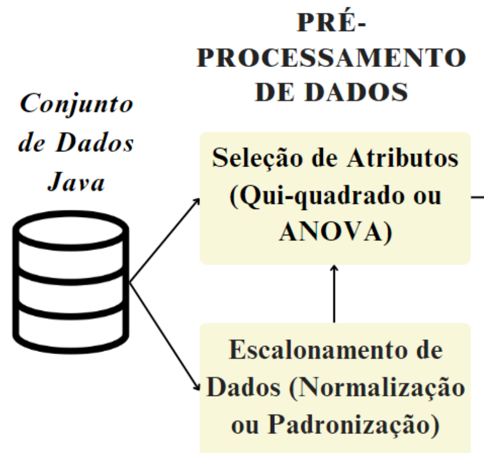


Figura 3 – Pré-processamento de Dados

são distintos, e a diferença entre os valores mínimo e máximo de cada atributo (e.g., LOC_method e LOC_package) pode apresentar variações consideráveis. Adicionalmente, alguns atributos do conjunto de dados apresentam vários *outliers*, como ATFD_Method e AMWNAMM_TYPE. Essas características ocultas podem influenciar os resultados dos modelos de AM.

O processo foi iniciado pela eliminação das instâncias de *code smell* duplicadas, (subseção 4.1.1), que estão sempre relacionadas ao mesmo nível e nunca entre diferentes níveis. Em seguida, foram realizadas imputações para os valores ausentes de 21 atributos, uma vez que esses atributos são específicos para os *code smells* de método (84 atributos) e não contêm valores nos conjuntos de dados de *code smells* de classe (63 atributos). Essa imputação foi necessária para garantir que os modelos de AM pudessem operar adequadamente, sem comprometer as classificações dos conjuntos de dados de *code smells* de classe.

Em seguida, a segunda parte desta tarefa visa realizar a seleção de atributos, distinguindo entre funções semelhantes nos padrões de design (43). Inicialmente, selecionou-se os atributos com o maior impacto no conjunto de dados Java, dividindo os subconjuntos em 10, 15, 20, 25 e 30% dos atributos originais. Para nos concentrar nas métricas de software que ajudam na diferenciação de atributos semelhantes, foram utilizados ANOVA e qui-quadrado. Além disso, como a abordagem proposta emprega um detector binário de *code smell* e um avaliador de gravidade multiclasse, cada técnica de seleção de atributos foi configurada com alvo binário ou multiclasse. No caso do qui-quadrado, também se considerou os atributos selecionados por Santos, Duarte e Choren(68) (QQ-Original), com base em Dewangan et al.(44). Outro aspecto considerado na seleção de atributos foi o escalonamento dos preditores. Para isso, foram incluídos preditores normalizados, padronizados ou não escalonados no processo de seleção de atributos.

A Tabela 7 apresenta os atributos selecionados para os diversos MCs voltados à detecção de *code smell* e à avaliação de gravidade.

Tabela 7 – Atributos Selecionados com ANOVA e Qui-Quadrado

Dimensão de Qualidade	**Rótulo da Métrica e Citação**	**Nome da Métrica**	**Granularidade**
Tamanho	LOC (105)	Linhas de Código	Projeto, Pacote, Classe, Método
	LOCNAMM (106)	Linhas de Código sem Métodos Acessadores ou Modificadores	Classe
	NOA	Número de Atributos	Classe
	NOCS (107, 105)	Número de Classes	Projeto, Pacote
	NOM (107, 105)	Número de Métodos	Projeto
	NOMNAMM (106)	Número de Métodos não Acessadores ou Modificadores	Projeto, Classe
Complexidade	AMW (108, 107)	Peso Médio dos Métodos	Classe
	AMWNAMM (107)	Peso Médio dos Métodos sem Acessadores ou Modificadores	Classe
	ATLD (106)	Acesso a Dados Locais	Método
	CLNAMM (106)	Métodos Locais Chamados sem serem Acessadores ou Modificadores	Método
	CYCLO (107, 109)	Complexidade Ciclomática	Método
	MAXNETING (110)	Nível Máximo de Aninhamento	Método
	NOAV (110)	Número de Variáveis Acessadas	Método
	NOLV (101)	Número de Variáveis Locais	Método
	NOP (8)	Número de Parâmetros	Método
	WMC (108, 107)	Contagem de Métodos Ponderados	Classe
	WMCNAMM (106)	Contagem de Métodos Ponderados sem Métodos Acessadores ou Modificadores	Classe
	WOC (101)	Peso da Classe	Classe
Acoplamento	ATFD (101)	Acesso a Dados Externos	Classe, Método
	CDISP (101)	Dispersão de Acoplamento	Método
	CFNAMM (106)	Métodos Externos Chamados sem serem Acessadores ou Modificadores	Classe, Método
	CINT (101)	Intensidade de Acoplamento	Método
	FANOUT (110)	Número de Classes Chamadas	Classe, Método
	FDP (110)	Provedores de Dados Externos	Método
	MaMCL (106)	Comprimento Máximo da Cadeia de Mensagens	Método
	MeMCL (106)	Comprimento Médio da Cadeia de Mensagens	Método
	RFC (108, 107)	Resposta para uma Classe	Classe
Encapsulamento	NOAM (110)	Número de Métodos Acessadores	Classe
Coesão	LCOM5 (107, 111)	Falta de Coesão nos Métodos	Classe
	TCC (111, 112, 107)	Coesão Forte da Classe	Classe
Herança	DIT (108, 107)	Profundidade da Árvore de Herança	Classe
	NIM (107, 105)	Número de Métodos Herdados	Classe

Esses atributos refletem algumas dimensões de qualidade na engenharia de software orientada a objetos (OO), como tamanho, complexidade, acoplamento, encapsulamento e

coesão. Essas métricas são calculadas em nível de projeto, pacote, método, classe ou em todos os níveis. O Apêndice B apresenta: i) a Tabela 27, com a localização dos atributos selecionados no vetor de atributos do conjunto de dados; e ii) A Figura 14, com o gráfico da frequência de seleção de atributos para os melhores detectores de *code smells*.

Além das métricas originais, a abordagem proposta incorpora outras duas métricas personalizadas definidas em Fontana et al.(8): i) o número de atributos não-final e não estáticos por classe (NONFNSA) e ii) o número de atributos privados por classe (NOPVA). Essas duas métricas foram selecionadas utilizando a técnica ANOVA para capturar propriedades estruturais adicionais do código-fonte (8).

Em seguida, seguiu-se as definições os detalhes computacionais necessários para calculá-las, conforme estabelecido por Fontana et al.(8), disponíveis em *Evolution of Software Systems and Reverse Engineering (ESSeRE Lab)*[1]. Essa abordagem foi utilizada para extrair os valores das métricas selecionadas para as instâncias do conjunto de dados C# gerado.

5.2 Otimização de Hiperparâmetros

Após a seleção dos principais atributos do conjunto de dados Java, foi realizada a otimização dos hiperparâmetros, (Figura 4). Utilizou-se os algoritmos de busca randomizada (RS) e busca bayesiana (BS) para identificar as configurações mais adequadas para os hiperparâmetros da AD de cada MC destinado à detecção de *code smells*.

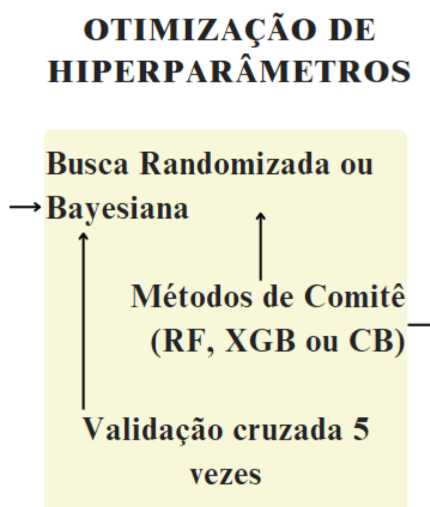


Figura 4 – Otimização de Hiperparâmetros

A Tabela 8 apresenta os detalhes dos hiperparâmetros que devem ser ajustados para cada MC. Esses hiperparâmetros foram configurados com o objetivo de otimizar

[1] <https://essere.disco.unimib.it/wp-content/uploads/sites/71/2019/04/metric-definitions.pdf>

o modelo com base na métrica F1, sem levar em consideração o desequilíbrio entre as classes-alvo.

Tabela 8 – Detalhamento dos Hiperparâmetros

AM	Hiperparâmetros	Valor Inicial	Valor Final	Passos
AD	max_depth	1	20	1
	min_samples_split	2	20	1
	min_samples_leaf	1	20	1
	criterion	entropy/gini/log_loss		
	max_features	sqrt/log2/None		
RF	n_estimators	50	1000	2
	max_depth	1	20	1
	min_samples_split	2	20	1
	min_samples_leaf	1	20	1
	bootstrap	True/False		
	criterion	entropy/gini/log_loss		
	max_features	sqrt/log2/None		
XGB	n_estimators	50	1000	2
	max_depth	1	20	1
	learning_rate	0.01	0.5	30
	criterion	entropy/gini/log_loss		
	booster	gbtree/dart		
	tree_method	approx/hist		
	grow_policy	depthwise/lossguide		
	objective*	multi:softprob/multi:softmax		
	num_class*	12		
CB	iterations	50	1000	2
	depth	1	16	1
	learning_rate	0.01	0.5	30
	l2_leaf_reg	1	10	1
	border_count	32	256	16
	feature_border_type	Median/Uniform/GreedyLogSum/UniformAndQuantiles/MaxLogSum/MinEntropy		
	leaf_estimation_method	Newton/Gradient		
	auto_class_weights	Balanced/SqrtBalanced		
	grow_policy	SymmetricTree/Lossguide/Depthwise		
	bootstrap_type	Bayesian/Bernoulli/MVS/No		
	objective*	MultiClass		
	classes_count*	12		

* Usado apenas para otimização de hiperparâmetros para avaliação de gravidade.

O modelo-base, AD, e os três modelos de MC (RF[2], XGB[3] e CB[4]) compartilham

[2] <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
[3] <https://xgboost.readthedocs.io/en/stable/parameter.html>
[4] <https://catboost.ai/en/docs/references/training-parameters/>

diversos hiperparâmetros que impactam seu desempenho. Entre esses, destacam-se:

- *n_estimators/iterations* (para RF, XGB e CB): define o número de árvores (ou rodadas de aumento) a serem construídas. Um valor maior geralmente melhora a precisão, embora possa aumentar o tempo de treinamento.
- *max_depth/depth* (para todos os modelos de AM): controla a profundidade máxima de cada árvore, equilibrando a complexidade do modelo e o risco de *overfitting* (excesso de ajuste).
- *learning_rate* (para XGB e CB), também conhecido como *shrinkage*, determina a contribuição de cada árvore para o modelo final, com valores menores resultando em atualizações mais conservadoras.
- *criterion* (para RF e CB): define a função utilizada para avaliar as divisões ou medir erros, podendo variar entre funções como Gini, entropia e regressão.

Além disso, para XGB e CB, o parâmetro *objective* especifica se o modelo é destinado à classificação, regressão ou outras tarefas, enquanto *num_class/classes_count* define o número de classes-alvo.

Para AD e RF, foram utilizados alguns hiperparâmetros exclusivos, incluindo *min_samples_split*, *min_samples_leaf* e *max_features*, que controlam o número mínimo de amostras necessárias para dividir um nó, formar uma folha e limitar o número de recursos considerados para cada divisão para acelerar o treinamento, respectivamente. Esses hiperparâmetros tem a finalidade de evitar *overfitting*. Além destes, o RF também possui o *bootstrap*, que determina se a amostragem inicial é usada para construir árvores.

Em XGB, hiperparâmetros específicos incluem *booster*, que permite escolher entre modelos baseados em árvores ou modelos lineares e *tree_method*, que determina o algoritmo para a construção de árvores (e.g., *exact, approximate*). O XGB também possui *grow_policy*, que define a estratégia para o crescimento de árvores (*depth-wise or loss-guided*) e oferece mais flexibilidade no treinamento.

Para CB, hiperparâmetros exclusivos incluem *l2_leaf_reg*, que adiciona a regularização de L2 aos valores das folhas para reduzir o *overfitting* e, *border_count* e *feature_border_type*, que lidam com o armazenamento de recursos e a divisão para recursos numéricos, respectivamente. *Leaf_estimation_method* controla como os valores das folhas são calculados, enquanto *auto_class_weights* ajuda a equilibrar os pesos das classes automaticamente em conjuntos de dados desequilibrados. O CB também possui seu próprio *grow_policy* e *bootstrap_type*, oferecendo flexibilidade na forma como as árvores crescem e as amostras são selecionadas.

Concluindo, para melhorar o desempenho dos modelos, foi adotado um método de validação cruzada de 5 vezes durante as técnicas de otimização de hiperparâmetros. Essa abordagem foi escolhida devido à presença de classes com um número pequeno de instâncias, o que pode afetar a robustez das avaliações de desempenho.

5.3 Detecção de *Code Smell*

O processo de detecção de *code smells* começa com a análise de desempenho dos modelos de AM, considerando tanto aqueles combinados com técnicas de seleção de atributos quanto os modelos sem essa integração. Essa análise foi importante para identificar os modelos mais adequados para a fase de detecção dentro da abordagem proposta, como ilustrado na Figura 5.

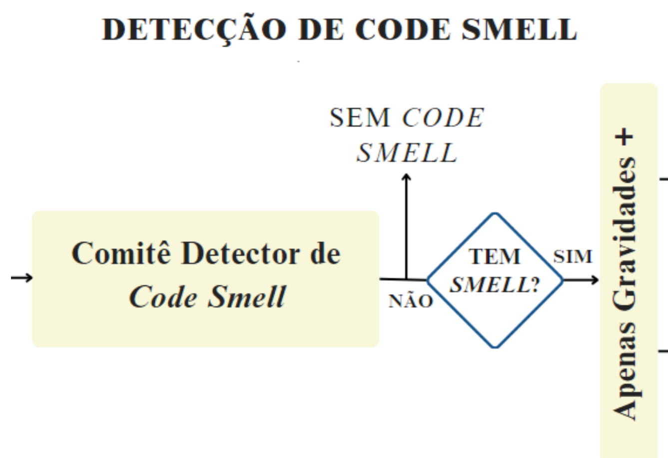


Figura 5 – Detecção de *Code Smell*

Os modelos de AM mais eficazes foram selecionados com base em sua configuração, avaliando combinações de algoritmos de otimização, técnicas de seleção de atributos, escalonamento de dados, porcentagens de atributos selecionados e tipos de alvos (binário ou multiclasse).

Detectores construídos com RF, XGB e CB se destacaram em comparação com aqueles que utilizam apenas o modelo-base destes classificadores, AD, utilizando uma seleção otimizada de atributos do conjunto de dados Java. A seleção dos atributos foi realizada por meio de técnicas como qui-quadrado normalizado e ANOVA não escalonado, resultando em uma representação mais eficiente para a detecção. Detalhes adicionais sobre os atributos selecionados e suas frequências estão disponíveis no Apêndice C

Os detectores foram otimizados por estratégias como otimização bayesiana e randomizada, dependendo da configuração de cada modelo. Hiperparâmetros específicos para esses detectores podem ser consultados no Apêndice C, enquanto gráficos ilustrando o

desempenho dos modelos de AM ao longo do processo estão no Apêndice E (ANOVA) e no Apêndice F (Qui-Quadrado).

Finalmente, o processo é encerrado se uma instância negativa de *code smell* for detectada. Caso contrário, ao identificar instâncias positivas, um novo subconjunto de dados é criado contendo apenas essas instâncias positivas. Esse subconjunto segue para a tarefa de avaliação da gravidade, livre da influência de instâncias negativas.

5.4 Avaliação da Gravidade

Assim, com os dez modelos selecionados, passou-se para a próxima etapa de otimização de hiperparâmetros para os classificadores que avaliam a gravidade de *code smell* em uma abordagem multiclasse, (Figura 6), uma vez que o conjunto de dados foi modificado na etapa anterior.

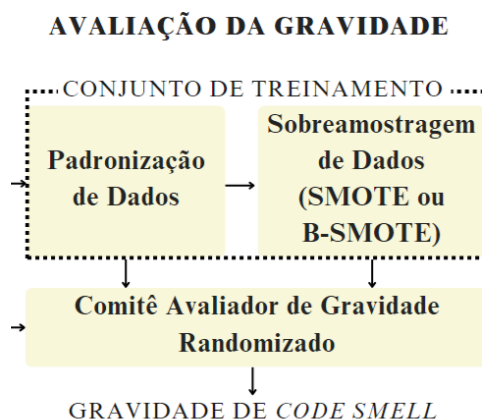


Figura 6 – Avaliação de Gravidade

Os hiperparâmetros selecionados para esta tarefa estão disponíveis na Apêndice D. Nesta fase, foi utilizada apenas a otimização randomizada, que apresentou resultados comparáveis à otimização bayesiana, mas com um custo de tempo menor (consulte a Tabela 10). É importante notar que essa avaliação da gravidade não é afetada pelas instâncias negativas comuns nesse tipo de problema, pois essas instâncias são descartadas previamente antes desta segunda etapa da abordagem, permanecendo apenas as instâncias positivas a serem analisadas (68).

Contudo, as instâncias de gravidade são desbalanceadas e em quantidade reduzida, conforme ilustrado na Tabela 4. Para lidar com essa situação, empregou-se as técnicas de sobreamostragem SMOTE e B-SMOTE nos dados de treinamento, visando equilibrar as instâncias de gravidade antes de aplicar a otimização randomizada para selecionar os melhores modelos para os classificadores no processo de avaliação da gravidade. O SMOTE é um método estatístico projetado para tratar conjuntos de dados desbalanceados, gerando

instâncias equilibradas e sintetizando novos pontos de dados com base em instâncias minoritárias existentes como entrada (40). Enquanto B-SMOTE é uma versão aprimorada do SMOTE, e prioriza a otimização de amostras de limite, alavancando o SMOTE, lidando com esse tipo de amostras da classe minoritária (41).

Por fim, realizou-se o treinamento do comitê avaliador de gravidade e a predição das novas observações contidas no conjunto de teste para determinar não apenas o tipo de *code smell*, mas, também, o nível de gravidade relacionado. A abordagem em duas etapas, que inclui a detecção binária de *code smells* seguida pela avaliação da gravidade nas instâncias positivas, busca mitigar os problemas de desbalanceamento, vazamento de dados de treinamento e melhorar a precisão dos modelos. Para isso, aplicam-se técnicas de sobreamostragem como SMOTE e B-SMOTE no conjunto de treinamento antes da otimização randomizada, com o objetivo de equilibrar as instâncias de gravidade e construir classificadores robustos.

6 EXPERIMENTOS E RESULTADOS

Este capítulo detalha os experimentos realizados utilizando conjuntos de dados Java e C#, aplicando técnicas de pré-processamento como seleção de atributos, balanceamento e escalonamento. Além disso, são empregados MC para detectar e avaliar a gravidade de *code smells* com base na TA.

6.1 Configuração dos Experimentos

Para os experimentos, foi utilizado um conjunto de dados Java combinado, conforme proposto por Santos, Duarte e Choren(68), que abrange os conjuntos de dados de Fontana e Zanoni(10) e Nanda e Chhabra(14). Este conjunto foi balanceado para assegurar uma proporção de 1:1 entre instâncias negativas e positivas, resultando em um total de 826 instâncias, que contemplam quatro tipos de *code smells*: GC, DC, FE e LM. O conjunto de dados foi dividido em 70% para treinamento e otimização de hiperparâmetros, e 30% para teste. A Tabela 9 apresenta a configuração desses conjuntos dados utilizados nas etapa de otimização de hiperparâmetros, detecção e avaliação da gravidade. A relação entre as instâncias com e sem *code smell* para treinamento e otimização de hiperparâmetros é no máximo de 17,56% (12-GC Grave), no mínimo de 2,15% (LM não Grave) e na média de 8,93%.

Tabela 9 – Configuração dos Conjuntos de Dados de Treinamento, Otimização de Hiperparâmetros e Teste

Dados	Sem CS	Feature Envy			Long Method			Data Class			God Class		
		1	2	3	1	2	3	1	2	3	1	2	3
OH/Train.	279	10	31	9	6	43	26	25	39	31	7	23	49
Teste	134	3	8	2	2	24	8	7	18	6	2	10	24

Além disso, foi utilizada validação cruzada com 5 repetições para calcular os desempenhos dos modelos, dividindo os dados em cinco segmentos e realizando cinco iterações. Em cada replicação, uma parte é avaliada como o conjunto de teste, enquanto as demais são utilizadas para o treinamento. Essa configuração garante uma quantidade adequada de dados para o aprendizado e um conjunto robusto para teste, assegurando a representatividade de cada classe nas avaliações.

Após a configuração dos hiperparâmetros, a validação dos modelos e a análise das melhorias de desempenho de cada MC, os modelos mais adequados foram selecionados para cada fase desta abordagem. A Tabela 10 apresenta os melhores detectores para cada MC, considerando, também, todas as características. No entanto, os modelos que

utilizaram todas as características ou àqueles que se baseiam apenas em AD não superaram os resultados obtidos com uma menor porcentagem de atributos selecionados com MC. Especificamente, destacam-se: i) cinco modelos que apresentaram a melhor acurácia, com valores iguais ou superiores a 94,35%, e ii) cinco modelos que apresentaram a melhor acurácia média na validação cruzada, com média maior ou igual a 93,58%.

Tabela 10 – Os melhores Métodos de Comitê otimizados para detecção de *code smell*

MC	SelAtr	Escal. SelAtr	% Pred.	Alvo SelAtr	OH	Acc	Mean	Std	Time(s)
XGB	QQ	Norm	25%	Bin	RS	**95.56**	92.98	1.75	448
RF	QQ*	Norm	30%	Bin	RS	**95.16**	92.73	1.39	33
XGB	QQ	Norm	30%	Bin	BS	**95.16**	92.73	2.37	3551
RF	QQ	Norm	25%	Bin	RS	**94.76**	92.61	1.41	35
CB	QQ*	Norm	30%	Bin	RS	**94.35**	92.49	2.05	1697
XGB		todos-atributos			BS	94.35	93.58	0.98	8169
CB		todos-atributos			BS	94.35	93.58	1.98	8407
RF		todos-atributos			RS	93.95	92.37	2.19	53
XGB	Anova	-	30%	Mult	BS	93.95	**93.83**	1.77	379
CB	Anova	-	30%	Mult	BS	92.74	**93.58**	1.13	3395
RF	QQ	Norm	20%	Mult	BS	94.76	**93.58**	1.26	123
CB	QQ	Norm	20%	Mult	BS	93.55	**93.58**	1.26	3239
CB	Anova	-	25%	Bin	BS	93.55	**93.58**	1.56	2511
AD	Anova	-	15%	Bin	RS	91.13	87.04	2.87	1
AD	QQ*	Norm	30%	Bin	RS	90.73	89.22	1.57	3
AD	QQ	Norm	30%	Mult	BS	90.73	86.20	2.78	346

*Qui-Quadrado de Santos, Duarte e Choren(68) baseado em Dewangan et al.(44).

Nesse contexto, foram empregadas técnicas de seleção de atributos, escalonamento de dados e sobreamostragem no conjunto de dados Java. Essa etapa deste trabalho resultou na seleção dos melhores modelos para a detecção e avaliação da gravidade dos *code smells* GC, DC, FE e LM em instâncias de cada uma das linguagens de programação (Java e C#), conforme ilustrado na Tabela 11 e na Tabela 12.

Foram realizados ajustes nos modelos para a TA entre conjuntos de dados com instâncias de *code smells* de uma linguagem para outra. Para TA do conjunto de dados Java para C#, foram realizados os passos abaixo nos dados tanto para detectores de *code smell* quanto para avaliadores da gravidade, a saber:

1. Carregamento de todo o conjunto de dados Java;
2. Carregamento de apenas um dos quatro subconjuntos de dados C# de *code smell* (GC, DC, FE ou LM) para testar, que servem como ajuste fino para os modelos baseados em Java selecionados;

Tabela 11 – Métodos de Comitê Otimizados para Transferência de Aprendizado de C# para Java

Code Smell	Ação	MC	SelAtr	Escal. SelAtr	% Pred.	Alvo SelAtr	OH	Escal. Treino?	Sobr.
GC	DetCS	CB	QQ	Norm	20	Mult	BS	Sim	B-SMOTE
	AvGrav	CB	ANOVA	-	30	Mult	RS	Não	B-SMOTE
LM	DetCS	XGB	ANOVA	-	30	Mult	BS	Sim	B-SMOTE
	AvGrav	XGB	QQ	Norm	25	Bin	RS	Não	B-SMOTE
DC	DetCS	XGB	QQ	Norm	30	Bin	BS	Não	-
	AvGrav	RF	QQ	Norm	25	Bin	RS	Não	SMOTE
FE	DetCS	CB	ANOVA	-	30	Mult	BS	Não	-
	AvGrav	RF	ANOVA	-	25	Bin	RS	Sim	SMOTE

Tabela 12 – Métodos de Comitê Otimizados para Transferência de Aprendizado de Java para C#

Code Smell	Ação	MC	SelAtr	Escal. SelAtr	% Pred.	Alvo SelAtr	OH	Escal. Treino?	Sobr.
GC	DetCS	CB	QQ	Norm	20	Mult	BS	Sim	SMOTE
	AvGrav	XGB	QQ	Norm	30	Bin	RS	Sim	SMOTE
LM	DetCS	CB	QQ*	Norm	30	Bin	RS	Sim	-
	AvGrav	RF	ANOVA	-	25	Bin	RS	Não	B-SMOTE
DC	DetCS	CB	ANOVA	-	25	Bin	BS	Não	SMOTE
	AvGrav	RF	QQ	Norm	25	Bin	RS	Não	-
FE	DetCS	CB	QQ*	Norm	30	Bin	RS	Sim	-
	AvGrav	XGB	QQ*	Norm	30	Bin	RS	Sim	B-SMOTE

*Qui-Quadrado usado por Santos, Duarte e Choren(68) baseado em Dewangan et al.(44).

3. Divisão dos conjuntos de dados Java e C# em 70% para treinamento e 30% para teste;

4. Padronização ou não dos conjuntos de dados de treinamento e teste Java e C#;

5. Sobreamostragem (SMOTE ou B-SMOTE) ou não dos conjuntos de treinamento Java e C#;

6. Combinação dos dados de treinamento Java e C# em um único conjunto de treinamento;

7. Treinamento dos Modelos AM; e

8. Predição com os modelos de AM para o conjunto de dados de teste C#.

Invertem-se os conjuntos de dados nos itens 1, 2 e 8 para a TA de C# para Java. Quatro métricas de desempenho precisão (P), sensibilidade (S), pontuação F1 (F1)

e acurácia (Acc) - foram utilizadas para avaliar a qualidade dos MCs. Essas métricas são calculadas com base em uma matriz de confusão, que inclui as classificações reais e previstas para a detecção de padrões de design (113), i.e., Verdadeiro Positivo (VP), Falso Positivo (FP), Verdadeiro Negativo (VN) e Falso Negativo (FN). Por fim, utilizou-se a análise de importância de atributos dos classificadores (RF, XGB e CB) e algoritmos de explicabilidade (LIME e SHAP) para detalhar os resultados dos modelos na detecção de *code smells* e na avaliação de gravidade.

6.2 Resultados

Os experimentos com os MC foram realizados de seis maneiras diferentes para cada possível detector de *code smell* ou avaliador de gravidade, utilizando os conjuntos de dados combinados (Java e C#) com e sem TA: i) sem sobreamostragem, padronizado ou não padronizado; ii) com SMOTE, padronizado ou não padronizado; e iii) com B-SMOTE, padronizado ou não padronizado.

Esses experimentos determinaram a melhor combinação de seleção de atributos, otimização de hiperparâmetros, escalonamento de dados, balanceamento de dados e técnicas de AM para a detecção de *code smells* e a avaliação de gravidade nos conjuntos de dados em cada estágio da abordagem proposta.

Primeiro, foram selecionados os melhores atributos para a detecção de *code smells* e para a avaliação de gravidade, utilizando as técnicas de Qui-quadrado e ANOVA. Em seguida, cada MC foi configurado com os melhores hiperparâmetros, usando algoritmos de otimização randomizada ou bayesiana.

Após isso, a padronização e a sobreamostragem dos dados de treinamento foram aplicadas em conjunto ou separadamente. Com esses novos dados de treinamento os MC foram treinados e testados para cada linguagem de programação, assim como na abordagem de TA, utilizando os conjuntos de dados Java e C# combinados. Os resultados detalhados deste trabalho estão disponíveis no repositório GitHub[1]. Veja README.md para mais detalhes.

6.2.1 Transferência de Aprendizado de C# para Java

A Tabela 13, apresenta os resultados dos MCs com e sem TA para o conjunto de dados de projetos Java referentes aos *code smells* GC, LM, DC e FE, respectivamente. Esses resultados capturam aspectos importantes da TA de um conjunto de dados C# para um conjunto de dados Java. Os resultados detalhados encontram-se na seção H.1.

[1] https://github.com/fabiorosario/code-smell-severity-classification-based-transfer-learning

Tabela 13 – Resultados para a TA de C# para Java

Uso de TA	GC (P, S, F1)	LM (P, S, F1)	DC (P, S, F1)	FE (P, S, F1)
Sem TA	(95, 93, 93)	(94, 91, 93)	(90, 84, 86)	(91, 76, 82)
Com TA	(**97, 95, 95**)	(**95, 95, 95**)	(**90, 89, 88**)	(**97, 93, 94**)

Nesse contexto, observou-se que a precisão, a sensibilidade e a pontuação F1 aumentaram com a aplicação de TA nos conjuntos de dados de projetos Java. Destaca-se um aumento de quase 15% na pontuação F1 para o *code smell* FE, que subiu de 82% para 94%, além de um crescimento superior a 22% na sensibilidade do FE, que passou de 76% para 93%.

Além disso, é importante ressaltar que os outros *code smells* em projetos Java apresentaram aumentos menores na pontuação F1, em torno de 2%. Outro ponto relevante é que os avaliadores de gravidade com TA para os *code smells* GC, LM e FE alcançaram 90% ou mais na pontuação F1, enquanto essa marca não foi alcançada por nenhum dos avaliadores de gravidade sem a aplicação de TA.

A Tabela 14 apresenta o detalhamento das avaliações de gravidade nos três níveis para cada *code smell* analisado neste trabalho. A maioria das medidas de desempenho alcançou índices superiores a 90%, com destaque para as avaliações das gravidades *3-FE Grave* e *7-DC não Grave*, que atingiram 100% nas pontuações de precisão, sensibilidade e F1, apesar de contarem com poucas instâncias de suporte, que foram 2 e 9, respectivamente. Isso sugere que a utilização das técnicas de sobreamostragem (SMOTE para FE e B-SMOTE para os outros *code smells*) na avaliação da gravidade, mesmo em classes com pouca representatividade, contribuiu para a melhoria dos resultados dos modelos de AM utilizados.

Tabela 14 – Detalhamento das Avaliações de Gravidades - TA C# para Java

Gravidade	P	S	F1	Suporte
1-FE não Grave	**100**	60	75	5
2-FE	**93**	**93**	**93**	15
3-FE Grave	**100**	**100**	**100**	2
4-LM não Grave	67	50	57	4
5-LM	**92**	**92**	**92**	25
6-LM Grave	**93**	**100**	**96**	13
7-DC não Grave	**100**	**100**	**100**	9
8-DC	68	**94**	79	16
9-DC Grave	89	50	64	16
10-GC não Grave	**100**	60	75	5
11-GC	62	**100**	77	5
12-GC Grave	**100**	**96**	**98**	23

Em relação à precisão: i) as gravidades 1, 10 e 12 também alcançaram 100%; ii) as avaliações de gravidades 2 e 6 atingiram 93%; e iii) a gravidade 5 obteve 92%. Quanto aos resultados de sensibilidade: i) as gravidades 6 e 11 também alcançaram 100%; ii) índices acima de 90%, foram obtidos nas avaliações de gravidade para os tipos 2, 5, 8 e 12, com valores de sensibilidade de 93, 92, 94 e 96%, respectivamente. No que diz respeito à pontuação F1, as avaliações de gravidade para os tipos 2, 5, 6 e 12 atingiram 93, 92, 96 e 98%, respectivamente.

As matrizes de confusão apresentadas na Figura 7 indicam que os detectores de *code smells* em Java, utilizados na abordagem, conseguem identificar instâncias negativas, alcançando as medidas de desempenho de precisão, sensibilidade, pontuação F1 e acurácia, conforme abaixo:

- para FE, P=98%, S=99%, F1=98% e Acc=97%;
- para LM, P=100%, S=100%, F1=100% e Acc=100%;
- para DC, P=96%, S=99%, F1=98% e Acc=96%; e
- para GC, P=100%, S=98%, F1=99% e Acc=99%.

Esses resultados são promissores para a tarefa de avaliação de gravidade em projetos Java, considerando apenas as instâncias positivas. A ocorrência de falsos negativos foi 3 de 344 possíveis, o que representa uma proporção de 0,009% do total das instâncias testadas.

A Figura 8 apresenta a matriz de confusão da avaliação das gravidades, onde as frequências de acerto e erro alcançadas para as gravidades em seus três níveis, em relação a cada *code smell*, são mostradas.

O pior cenário ocorre quando o modelo avalia uma gravidade como menor do que a real, o que pode impactar na qualidade e no ciclo de vida do software, já que uma gravidade mais séria não é devidamente reconhecida e tratada. Nesse contexto, no modelo proposto essa situação ocorreu em algumas ocasiões, exceto na avaliação da gravidade DC Grave, conforme descrito a seguir:

- 1 instância foi classificada como LM não Grave, enquanto deveria ter sido classificada como LM, representando 4% de 25 instâncias.
- 1 instância foi classificada como GC não Grave e 7 instâncias foram classificadas como DC, quando deveriam ter sido classificadas como DC Grave, o que corresponde a 6 e 44% de 16 instâncias, respectivamente.
- 1 instância foi classificada como GC, quando deveria ter sido classificada como GC Grave, representando 4% de 23 instâncias.

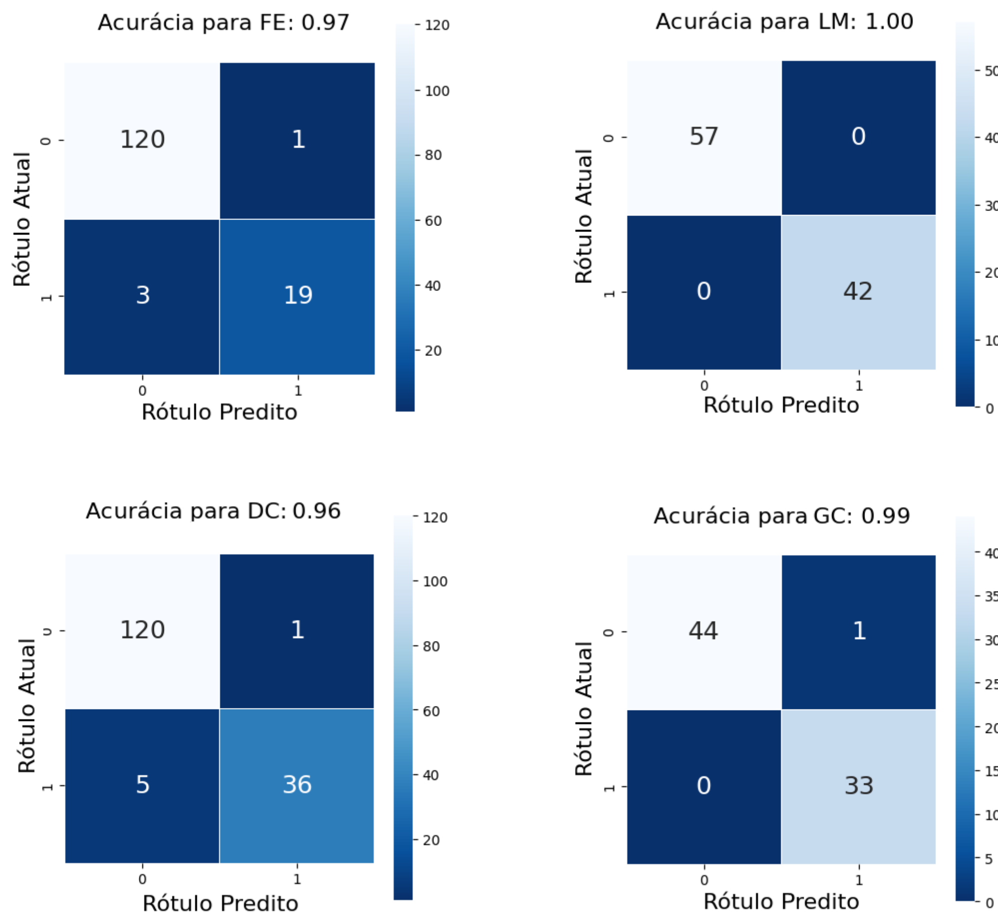


Figura 7 – Matriz de Confusão da Tarefa de Detecção de *Code Smell* Java

	FE não Grave	FE	FE Grave	LM não Grave	LM	LM Grave	DC não Grave	DC	DC Grave	GC não Grave	GC	GC Grave
FE não Grave	3	1	0	1	0	0	0	0	0	0	0	0
FE	0	14	0	0	1	0	0	0	0	0	0	0
FE Grave	0	0	2	0	0	0	0	0	0	0	0	0
LM não Grave	0	0	0	2	2	0	0	0	0	0	0	0
LM	0	0	0	1	23	1	0	0	0	0	0	0
LM Grave	0	0	0	0	0	13	0	0	0	0	0	0
DC não Grave	0	0	0	0	0	0	9	0	0	0	0	0
DC	0	0	0	0	0	0	0	15	1	0	0	0
DC Grave	0	0	0	0	0	0	0	7	8	1	0	0
GC não Grave	0	0	0	0	0	0	0	0	0	3	2	0
GC	0	0	0	0	0	0	0	0	0	0	5	0
GC Grave	0	0	0	0	0	0	0	0	0	0	1	22

Figura 8 – Matriz de Confusão da Tarefa de Avaliação de Gravidade Java

Outro cenário que pode resultar em custos de manutenção desnecessários ocorre quando uma gravidade de *code smell* menos grave é priorizada em detrimento de uma mais séria, devido à classificação incorreta. Isso também aconteceu algumas vezes, especialmente em classes de gravidades que contavam com poucas instâncias de suporte, exceto no caso das gravidades de LM, conforme detalhado a seguir:

- 1 instância foi classificada como FE quando deveria ter sido classificada como FE não Grave, representando 20% de 5 instâncias.
- 2 instâncias foram classificadas como LM quando deveriam ter sido classificadas como LM não Grave, correspondendo a 50% de 4 instâncias.
- 1 instância foi classificada como LM Grave quando deveria ter sido classificada como LM, o que equivale a 4% de 25 instâncias.
- 2 instâncias foram classificadas como GC quando deveriam ter sido classificadas como GC não Grave, totalizando 40% de 5 instâncias.

Outra informação importante que pode ser extraída da matriz de confusão é que o modelo proposto conseguiu prever todas as instâncias das classes LM, DC e GC em um de seus três níveis de gravidade, mesmo com a interferência de outras instâncias em nível de método e classe. No entanto, o mesmo fato não foi observado nas gravidades da classe FE, onde em 9% dos casos o modelo previu as gravidades de LM quando deveria ter previsto gravidades de FE, em 2 dos 22 casos.

Além disso, é importante notar que a abordagem proposta propiciou acertos em todas as instâncias ao considerar cada nível, seja método ou classe; ou seja, instâncias no nível de classe não interferiram na classificação de instâncias no nível de método e vice-versa. Isso indica, e.g., que a imputação de dados ausentes em instâncias no nível de classe resultou no comportamento esperado dos modelos.

6.2.2 Transferência de Aprendizado de Java para C#

Os resultados dos MC sem e com TA para o conjunto de dados de projetos C# são apresentados na Tabela 15. Assim como no conjunto de dados de projetos Java, a precisão, a sensibilidade e a pontuação F1 aumentaram com a aplicação de TA ao conjunto de dados de projetos C#.

É importante destacar que houve um aumento na sensibilidade e na pontuação F1 da avaliação da gravidade dos *code smells* DC e FE: i) a sensibilidade aumentou em 45% (de 60% para 87%) para DC e em 56% (de 61% para 95%) para FE; e ii) a pontuação F1 aumentou de 68% para 87% (um aumento de 28%) para DC e de 70% para 97% (um aumento de 39%) para FE. Os resultados detalhados encontram-se na seção H.2.

Tabela 15 – Resultados para a TA de Java para C#

Uso de TA	GC (P, S, F1)	LM (P, S, F1)	DC (P, S, F1)	FE (P, S, F1)
Sem TA	(90, 90, 90)	(87, 87, 87)	(88, 60, 68)	(90, 61, 70)
Com TA	(**92, 91, 91**)	(**88, 87, 87**)	(**88, 87, 87**)	(**100, 95, 97**)

A Tabela 16 apresenta o detalhamento das avaliações de gravidades em seus três níveis para cada *code smell* analisado neste trabalho. Quase metade das medidas de desempenho alcançou índices superiores a 90%, com destaque para as avaliações das gravidades 2-FE e 7-DC não Grave, que atingiram 100% nas pontuações de precisão, sensibilidade e F1, mesmo com poucas instâncias de suporte, que foram 2 e 3, respectivamente. Isso sugere que a utilização das técnicas de sobreamostragem, em conjunto com a maior quantidade de instâncias Java, que são mais representativas do que as de C# para essas gravidades, contribuiu para a melhoria dos resultados dos modelos de AM utilizados, mesmo em classes com pouca representatividade.

Tabela 16 – Detalhamento das Avaliações de Gravidades - TA Java para C#

Gravidade	P	S	F1	Suporte
1-FE não Grave	**100**	88	**93**	8
2-FE	**100**	**100**	**100**	2
3-FE Grave	-	-	-	0
4-LM não Grave	86	**93**	89	94
5-LM	71	55	62	44
6-LM Grave	53	67	59	12
7-DC não Grave	**100**	**100**	**100**	3
8-DC	0	0	0	1
9-DC Grave	-	-	-	0
10-GC não Grave	87	**97**	**92**	34
11-GC	82	70	76	20
12-GC Grave	**100**	71	83	7

Além disso, é importante destacar outros resultados de medidas de desempenho. Para precisão, as classes 1-FE não Grave e 12-GC Grave atingiram 100% de precisão. No caso da sensibilidade, as classes 4-LM não Grave e 10-GC não Grave superaram os 90% de sensibilidade, com 93 e 97%, respectivamente. Em relação à pontuação F1, as avaliações de gravidade nas classes 1-FE não Grave e 10-GC não Grave também se destacaram, com índices de 93 e 92%, respectivamente.

As matrizes de confusão apresentadas na Figura 9 indicam que os detectores de *code smells* em C# propostos conseguem identificar as instâncias negativas, alcançando resultados promissores nas métricas de precisão, sensibilidade, pontuação F1 e acurácia para todos os *code smells* analisados.

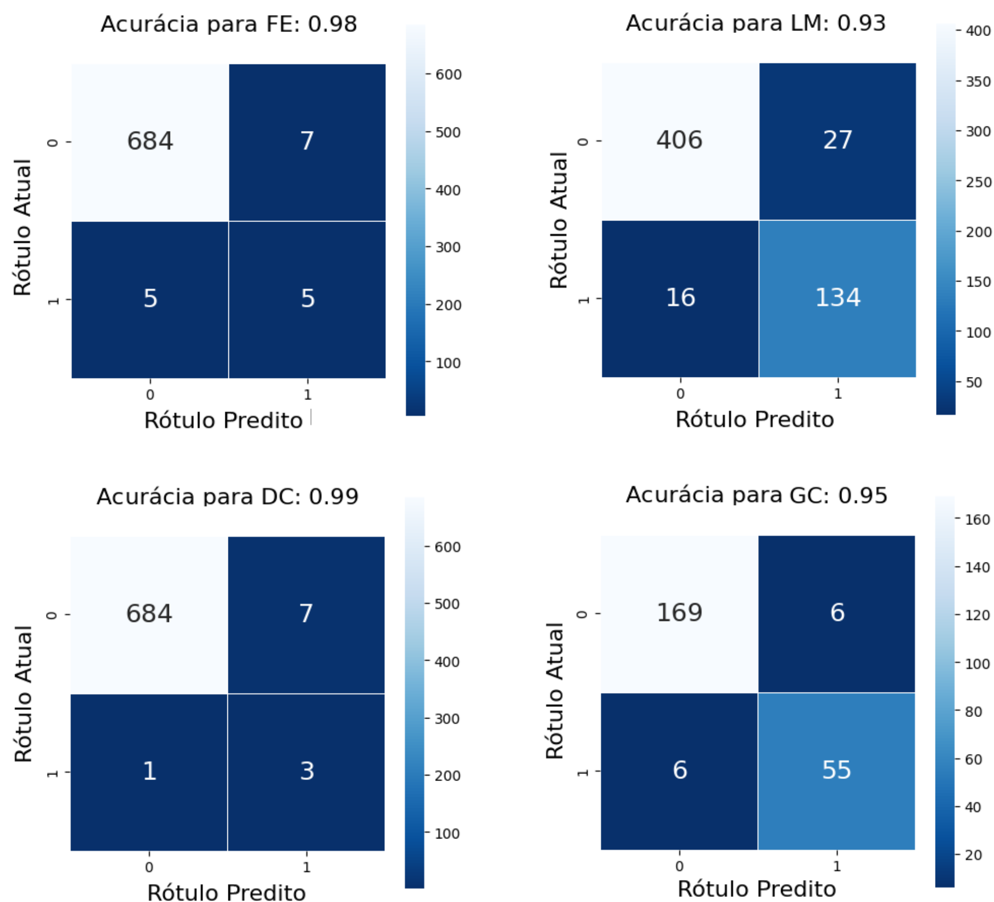


Figura 9 – Matriz de Confusão da Tarefa de Detecção de *Code Smell* C#

Em comparação com os detectores para Java, os detectores em C# apresentaram desempenho superior nas métricas para FE e DC. No entanto, foram superados nas medidas de desempenho para LM e GC (conforme a Figura 7).

- para FE, P=99%, S=99%, F1=99% e Acc=98%;
- para LM, P=96%, S=94%, F1=95% e Acc=93%;
- para DC, P=100%, S=99%, F1=99% e Acc=99%; e
- para GC, P=97%, S=97%, F1=97% e Acc=95%.

Os resultados supracitados contribuem para a tarefa de avaliação de gravidade em projetos C#, pois a ocorrência de Falsos Negativos é de 47 de 1990 possíveis, i.e., aproximadamente 0,024% do total das instâncias que serviram de suporte para estes testes.

A Figura 10 apresenta a matriz de confusão da avaliação das gravidades em três níveis para cada *code smell* para projetos C#.

	FE não Grave	FE	FE Grave	LM não Grave	LM	LM Grave	DC não Grave	DC	DC Grave	GC não Grave	GC	GC Grave
FE não Grave	7	0	0	0	0	1	0	0	0	0	0	0
FE	0	2	0	0	0	0	0	0	0	0	0	0
FE Grave	0	0	0	0	0	0	0	0	0	0	0	0
LM não Grave	0	0	0	87	6	1	0	0	0	0	0	0
LM	0	0	0	14	24	6	0	0	0	0	0	0
LM Grave	0	0	0	0	4	8	0	0	0	0	0	0
DC não Grave	0	0	0	0	0	0	3	0	0	0	0	0
DC	0	0	0	0	0	0	0	0	1	0	0	0
DC Grave	0	0	0	0	0	0	0	0	0	0	0	0
GC não Grave	0	0	0	0	0	0	0	0	0	33	1	0
GC	0	0	0	0	0	1	0	0	0	5	14	0
GC Grave	0	0	0	0	0	0	0	0	0	0	2	5

Figura 10 – Matriz de Confusão da Tarefa de Avaliação de Gravidade C#

O pior cenário, quando o modelo indica uma gravidade com nível menor do que ela realmente é, aconteceu nos modelos para TA de Java para C#, conforme abaixo:

- 14 instâncias foram classificadas como LM não Grave quando deveria ter sido classificada como LM, i.e., 32% de 44 instâncias.
- 4 instâncias foram classificadas como LM quando deveriam ter sido classificadas como LM Grave, 34% de 12 instâncias.
- 5 instâncias foram classificadas como GC não Grave quando deveriam ter sido classificadas como GC, 26% de 19 instâncias.
- 2 instâncias foram classificadas como GC quando deveriam ter sido classificadas como GC Grave, 29% de 7 instâncias.

No cenário onde uma gravidade de *code smell* menos grave é priorizado em detrimento de uma mais séria devido à classificação incorreta, foram identificados menos ocorrências que no cenário anterior, a saber:

- 1 instância foi classificada como LM Grave quando deveria ter sido classificada como FE não Grave, 13% de 8 instâncias. Entretanto, uma análise aprofundada (conforme Apêndice I) deste método, *CaptureWithCursor*, que está disponível, também, no GitHub[2], mostra que a predição em LM Grave é mais adequada ao que foi proposto

[2] <https://github.com/NickeManarin/ScreenToGif/blob/2d318f837946f730e1b2e5c708ae9f776b9e360b/ScreenToGif/Capture/DirectCachedCapture.cs#L195-L401>

neste trabalho, priorizando a avaliação de instâncias mais graves de *code smells*. Isto ocorre pelo tamanho, complexidade, responsabilidade múltiplas e seções distintas deste método.

- 6 instâncias foram classificadas como LM e 1 instância foi classificada como LM Grave quando deveriam ter sido classificadas como LM não Grave, 6 e 1% de 94 instâncias.
- 6 instâncias foram classificadas como LM Grave quando deveria ter sido classificada como LM, 14% de 44 instâncias.
- 1 instância foi classificada como DC Grave quando deveria ter sido classificada como DC. Havia apenas uma instâncias para ser avaliada quanto à gravidade no conjunto de teste.
- 1 instância foi classificada como GC quando deveria ter sido classificada como GC não Grave, 3% de 34 instâncias.

Assim, como nos modelos de TA de C# para Java, a matriz de confusão da avaliação das gravidades nos modelos de TA de Java para C# mostra que os modelos conseguiram prever todas as instâncias das classes LM, DC e GC em um de seus três níveis de gravidade. Contudo, para as gravidades de FE houve um caso onde o modelo classificou a gravidade como sendo de LM, 10% dos casos de um total de 10 instâncias. Por fim, vale a pena destacar que, assim como a TA para Java, a TA para C# obteve êxito em avaliar as instâncias de gravidade de *code smell* em seu respectivo nível (seja de classe ou de método).

6.3 Explicabilidade dos Modelos AM

Neste trabalho, foram utilizados as abordagens de explicação *ex post facto* de importância dos atributos dos classificadores (RF, XGB e CB), LIME e SHAP para analisar a explicabilidade dos resultados MCs proposta para detecção e avaliação da gravidade de *code smells* tanto para Java quanto para C#.

6.3.1 Explicabilidade dos Modelos para Projetos Java

A Tabela 17 apresenta um resumo das análises de explicabilidade por *Code Smell* em projetos Java. Para isso, foram verificadas a importância dos atributos nos classificadores XGB e CB, tanto para a detecção quanto para a avaliação de gravidade. Além disso, para a detecção de *code smells*, foi utilizado o gráfico SHAP resumo para analisar o impacto global das variações dos atributos nas saídas dos modelos. O gráfico de LIME foi aplicado às instâncias com maior probabilidade de predição em cada nível de gravidade. O detalhamento dos resultados obtidos por cada tipo de explicabilidade dos

modelos para os *code smells* FE, LM, DC e GC pode ser encontrado, respectivamente, no Apêndice J, Apêndice K, Apêndice L e Apêndice M. A seguir, são apresentados os resultados da análise da explicabilidade para os modelos de AM utilizados para detecção e avaliação da gravidade de *code smells* Java.

Tabela 17 – Resumo das Análises de Explicabilidade por *Code Smell* Java

Code Smell	**Tipo de Explicabilidade**	**Atributos ou Valores de Atributos que mais impactaram nos modelos**
FE	Importância de Variáveis	Os atributos mais importantes para **detecção e avaliação de gravidade** foram AFTD_type e MAXNESTING_method.
	SHAP Resumo	Para **detecção**, valores altos de ATFD_type, AFTD_method e FDP_method impactam positivamente.
	LIME (Níveis de Gravidade)	Para **FE não Grave**: $NOA_type > 0.19$, $-0.23 < ATFD_method \leq 0.64$ e $-0.39 < CINT_method \leq 1.18$; para **FE**: $-0.13 < NOAV_method \leq 1.28$, $-0.22 < NOLV_method \leq 1.43$ e $-0.47 < NOA_type \leq -0.24$; e para **FE Grave**: $ATFD_method > 0.64$, $NOAV_method > 1.28$, $ATFD_type > 0.34$ e $RFC_type > 0.00$.
LM	Importância de Variáveis	Os atributos mais importantes foram: para **detecção** –MAXNESTING_method, FANOUT_method e ATFD_type; e para **avaliação de gravidade** —LOC_method, WMCNAMM_type, CYCLO_method e ATFD_method.
	SHAP Resumo	Para **detecção**, valores altos de MAXNESTING_method, LOC_method, FANOUT_method e FDP_method impactam positivamente.
	LIME (Níveis de Gravidade)	Para **LM não Grave**: $NOM_project > 4624.00$ e $0.00 < CINT_method \leq 10.26$; para **LM**: $CYCLO_method > 16.17$, $NOAV_method > 20.89$, $9.52 < CFNAMM_type \leq 28.00$, $28.00 < LOC_method \leq 104.00$, $LOCNAMM_type > 148.26$, $NIM_type \leq 2.20$ e $WOC_type \leq 0.26$; e para **LM Grave**: $AMW_type \leq 1.47$, $WMCNAMM_type \leq 6.00$, $CFNAMM_type \leq 1.98$ e $LCOM5_type > 0.93$.
DC	Importância de Variáveis	Os atributos mais importantes foram: para **detecção** – LOC_method, LOC_type e NOCS_project; e para **avaliação de gravidade** –WMC_type, LOC_method, WMCNAMM_type, LOC_type. e CYCLO_method.
	SHAP Resumo	Para **detecção**: i) valores altos de NOCS_project e LCOM5_type; e ii) valores baixos de LOC_method e NOAM_type.
	LIME (Níveis de Gravidade)	Para **DC não Grave**: $WMC_type \leq 12.70$, $LOC_type \leq 107.00$, $WMCNAMM_type \leq 6.00$, $AMW_type \leq 1.47$, $LOC_method \leq 3.00$, $CYCLO_me\text{-}thod \leq 1.00$, $CFNAMM_type \leq 1.98$ e $NOAV_method \leq 1.00$; para **DC**: $WMCNAMM_type \leq 6.00$, $AMW_type \leq 1.47$, $CFNAMM_type \leq 1.98$, $CYCLO_method \leq 1.00$, $LOC_me\text{-}thod \leq 3.00$, $ATFD_type \leq 1.00$, $NOAV_method \leq 1.00$ e $LCOM5_type > 0.85$; e para **DC Grave**: $AMW_type \leq 1.47$, $WMCNAMM_type \leq 6.00$, $CFNAMM_type \leq 1.98$, $LCOM5_type > 0.93$, $CYCLO_method \leq 1.00$, $NOM_project > 5064.00$, $NIM_type \leq 0.25$, $ATFD_type \leq 1.00$ e $LOC_method \leq 3.00$.
GC	Importância de Variáveis	Os atributos mais importantes foram: para **detecção** –LOC_method, CYCLO_method, CDISP_method e WMCNAMM_type; e para **avaliação de gravidade** –FANOUT_type, MeMCL_method, ATFD_method e ATFD_type.
	SHAP Resumo	Para **detecção**: valores mais altos de WMC_type, WMCNAMM_type e LOCNAMM_type.
	LIME (Níveis de Gravidade)	Para **GC não Grave**: $MAXNESTING_method \leq 3.00$, $ATFD_type \leq 68.50$, $FANOUT_method \leq 0.00$, $NOLV_method \leq 1.00$ e $CLNAMM_method \leq 2.74$; para **GC**: $MAXNESTING_method \leq 3.00$ e $FANOUT_method \leq 0.00$; e para **GC Grave**: $MAXNESTING_method \leq 3.00$ e $LOC_type > 3.00$.

Para FE (seção J.1), os atributos ATFD_type e MAXNESTING_method são os mais influentes para a detecção e avaliação de gravidade de *code smells*. O gráfico SHAP (seção J.2) mostra que valores altos de ATFD_type, ATFD_method e FDP_method impactam positivamente a detecção de FE. Enquanto LIME (seção J.3) indica: i) para gravidade FE não Grave, há uma tendência de valores moderados, especialmente para NOA_type e ATFD_method; ii) valores moderados de NOAV_method e NOLV_method indicam a gravidade FE; e iii) em instâncias de FE Grave, há um aumento nos valores de ATFD_method, NOAV_method e RFC_type, indicando alta complexidade e acoplamento.

Para LM (seção K.1), os atributos como MAXNESTING_method, FANOUT_method e ATFD_ type são importantes para a detecção de *code smells*, enquanto, para a avaliação de gravidade, LOC_method, WMCNAMM_type e CYCLO_method destacam-se. O gráfico SHAP (seção K.2) mostra que a complexidade e o tamanho do método, indicada por valores altos dos atributos MAXNESTING_method e LOC_method, impactam positivamente a detecção de *code smells*. Na análise do gráfico gerado por LIME (seção K.3) para os níveis de gravidade é importante destacar que os métodos NOM_project e CINT_method são importantes para a gravidade LM não Grave. Métodos com alta complexidade (e.g., CYCLO_method e NOAV_method) influenciam na gravidade LM. Enquanto os baixos valores de AMW_type e WMCNAMM_type para instâncias de LM Grave indicam uma estrutura de código deficiente.

Para DC (seção L.1), a explicabilidade ligada à importância de atributos indica que os atributos que mais influenciam na detecção de *code smells* incluem LOC_type e NOCS_project. Já na avaliação de gravidade, destaca-se atributos de complexidade, como WMC_type e WMCNAMM_type. SHAP (seção L.2) atribui importância a valores altos de NOCS_project e LCOM5_type e baixos de NOAM_type, como indicadores de baixa coesão e alta complexidade como fatores determinantes. Com LIME (seção L.3), instâncias DC não Grave apresentam baixos valores de atributos como WMC_type. Além disso, para DC Grave, altos valores de LCOM5_type e baixos de AMW_type sugerem classes com alta falta de coesão.

Por fim, para GC (seção M.1), atributos de complexidade, como WMCNAMM_type, são destacados na detecção de *code smells*. Enquanto, para avaliação de gravidade, atributos de acoplamento, FANOUT_type e ATFD_type, são mais relevantes. Da análise do gráfico SHAP (seção M.2) é possível atribuir alta influência a valores elevados de WMC_type e LOCNAMM_type, indicando complexidade e falta de modularidade. E, com a análise do gráfico LIME (seção M.3), observam-se que baixos valores para atributos de acoplamento, como ATFD_type indicam a presença da gravidade GC não Grave. O aumento desses valores de acoplamento influenciam na gravidade GC. Enquanto, em instâncias GC Grave, valores elevados de tamanho e acoplamento (LOC_type e ATFD_type) são influentes para o modelo AM .

A análise dos atributos mais impactantes para detecção e avaliação de gravidade dos *code smells* destaca o uso de atributos relacionados à complexidade e tamanho (e.g., MAXNESTING_method, LOC_method), acoplamento (e.g., ATFD_type, FANOUT_method) e coesão (e.g., LCOM5_type). SHAP e LIME complementam a explicabilidade ao fornecerem *insights* sobre o impacto dos atributos e suas faixas de valores para diferentes níveis de gravidade. Enquanto a Importância de Atributos oferece uma visão mais geral e robusta, identificando os principais atributos globais que influenciam os modelos. Este resumo (conforme a Tabela 17) sugere que os atributos de complexidade, tamanho, acoplamento e coesão desempenham um papel importante na explicabilidade dos modelos de detecção e avaliação de gravidade para *code smells* em projetos Java.

6.3.2 Explicabilidade dos Modelos para Projetos C#

A Tabela 18 apresenta um resumo das análises de explicabilidade por *Code Smell* em projetos C#. Para isso, foram verificadas a importância dos atributos nos classificadores CB, tanto para a detecção quanto para a avaliação de gravidade. Para a detecção de *code smells*, foi utilizado o gráfico SHAP resumo nos *code smells* em nível de classe, a fim de verificar o impacto global das variações dos atributos nas saídas dos modelos. Além disso, para uma análise mais detalhada dos resultados da avaliação de gravidade, foram utilizados os gráficos gerados pelo algoritmo LIME. Esses gráficos foram elaborados com base nas instâncias com maior probabilidade de predição em cada nível de gravidade. O detalhamento dos resultados obtidos por cada tipo de explicabilidade dos modelos para os *code smells* em C# FE, LM, DC e GC pode ser encontrado, respectivamente, no Apêndice N, Apêndice O, Apêndice P e Apêndice Q. A seguir, são apresentados a análise de explicabilidade para os modelos de AM para projetos C#.

Para FE (seção N.1), os atributos mais relevantes foram AMW_type e WOC_type para detecção, ambos relacionados à complexidade, enquanto, para avaliação de gravidade, destacam-se ATFD_method, CYCLO_method, NOAV_method e ATFD_type, indicando a importância de atributos de complexidade e acoplamento. A análise do gráfico LIME (seção N.2): para instâncias de FE não Grave, há um padrão com valores baixos de TCC_type e AMW_type, sugerindo maior coesão e menor complexidade destes métodos; para gravidade FE, atributos como ATFD_method e CFNAMM_type se destacam com valores elevados, indicando alto acoplamento. Vale destacar que não há instâncias de FE grave no conjunto de dados C#.

Para LM (seção O.1), os principais atributos para detecção são LOC_method e CYCLO_method, refletindo o tamanho e a complexidade do método. Já para avaliação de gravidade, destacam-se CYCLO_method, LOC_method, NOAV_method e RFC_method, indicando a influência da complexidade, tamanho e acoplamento do método. No gráfico gerado por LIME (seção O.2), as instâncias de LM não Grave, indicam que os métodos

Tabela 18 – Resumo das Análises de Explicabilidade por *Code Smell* C#

Code Smell	**Tipo de Explicabilidade**	**Atributos ou Valores de Atributos que mais impactaram nos modelos**
FE	Importância de Variáveis	Os atributos mais importantes foram: para **detecção** –AMW_type e WOC_type; e para **avaliação de gravidade** –ATFD_method, CYCLO_method, NOAV_method e ATFD_type
	SHAP Resumo	-
	LIME (Níveis de Gravidade)	Para **FE não Grave**: $-0.05 < TCC_type \leq -0.02$, $AMW_type \leq -0.08$, $-0.39 < CINT_method \leq 0.62$, $-0.38 < ATFD_method \leq 0.54$, $-0.17 < RFC_type \leq 0.46$ e $NOLV_method \leq -0.38$; para **FE**: $ATFD_method > 0.54$, $CFNAMM_type > 0.71$, $WOC_type > -0.16$ e $AMW_type \leq -0.08$; e para **FE Grave**: não há instâncias desta gravidade no conjunto de dados C#.
LM	Importância de Variáveis	Os atributos mais importantes foram: para **detecção** –LOC_method e CYCLO_method; e para **avaliação de gravidade** –CYCLO_method, LOC_method, NOAV_method e RFC_method.
	SHAP Resumo	-
	LIME (Níveis de Gravidade)	Para **LM não Grave**: $RFC_type \leq 0.00$, $3.00 < LOC_method \leq 37.00$, $1.00 < CYCLO_method \leq 6.00$, $ATFD_method \leq 0.00$ e $1.00 < NOAV_method \leq 8.00$; para **LM**: $CYCLO_method > 13.00$, $LOC_method > 83.00$, $RFC_type \leq 0.00$, $NOLV_method > 12.00$, $ATFD_method \leq 0.00$, $NOAV_method > 19.50$ e $CFNAMM_method > 8.00$; e para **LM Grave**: $CYCLO_method > 13.00$, $LOC_method > 83.00$, $NOLV_method > 12.00$, $CINT_method > 11.00$ e $FDP_method > 5.00$
DC	Importância de Variáveis	Os atributos mais importantes foram: para **detecção** –NOA_type, NOAM_type e LOC_method; e para **avaliação de gravidade** –LOC_method, NOAV_method e WMC_type.
	SHAP Resumo	Para **detecção**: i) valores altos de NOAM_type e NONFNSA; e ii) valores baixos de NOPVA.
	LIME (Níveis de Gravidade)	Para **DC não Grave**: $WMC_type \leq 13.00$, $LOC_method \leq 3.00$, $NOAV_method \leq 1.00$, $3.00 < WMCNAMM_type \leq 50.00$, $LCOM5_type \leq 0.83$ e $LOC_package \leq 1367.50$; para **DC**: há apenas uma instância no conjunto de teste C# e a predição foi errada (DC Grave), principalmente pelo impacto do valor do atributo WMC_type, > 13.00, exerce em cada tipo de gravidade de DC, sendo maior para a gravidade DC Grave, 0.13.; e para **DC Grave**: não há instâncias desta gravidade no conjunto de dados C#.
GC	Importância de Variáveis	Os atributos mais importantes foram: para **detecção** – WMCNAMM_type, LOC_package e LCOM5_type; e para **avaliação de gravidade** –AFTD_method, CYCLO_method e LOC_method.
	SHAP Resumo	Para **detecção**: valores mais altos de LOC_package, WMCNAMM_type, LCOM5_type e AMW_type.
	LIME (Níveis de Gravidade)	Para **GC não Grave**: $AMW_type \leq -0.09$, $NOCS_package \leq -0.28$, $LCOM5_type > -0.10$ e $-0.38 < ATFD_method \leq 0.00$; para **GC**: $LOC_type > 0.75$, $-0.45 < NOAV_method \leq -0.04$, $-0.16 < NOAM_type \leq 0.00$, $NOMNAMM_project > 0.39$, $0.10 < WMC_type \leq 0.71$, $0.02 < LOCNAMM_type \leq 1.03$, $-0.01 < NIM_type \leq 0.95$, $LOC_package \leq -0.45$ e $-0.38 < ATFD_method \leq 0.00$; e para **GC Grave**: $LOC_type > 0.75$, $ATFD_type > 0.56$, $CFNAMM_type > 0.79$, $-0.45 < NOAV_method \leq -0.04$, $LOCNAMM_type > 1.03$ e $LCOM5_type > -0.10$. Entretanto, $AMW_type \leq -0.09$, $-0.09 < WOC_type \leq 0.17$ e $NOCS_package \leq -0.28$

possuem acoplamento,complexidade e tamanho moderado, refletida por valores baixos de RFC_type. LOC_method, CYCLO_method e ATFD_method. Para a gravidade LM a complexidade (CYCLO_method) e o tamanho (LOC_method) dos métodos aumentam.

Em casos de LM Grave, observa-se um aumento de outros atributos de complexidade (NOLV_method e FDP_method) e acoplamento (CINT_method), indicando métodos longos e complexos.

Para DC (seção P.1), a detecção é mais influenciada por NOA_type e NOAM_type, indicando atributos relacionados ao tamanho e ao encapsulamento. Para avaliação de gravidade, destaca-se WMC_type. O gráfico de resumo SHAP (seção P.2) mostra que valores altos de NOAM_type e baixos de NOPVA impactam positivamente na detecção, indicando baixo encapsulamento. O gráfico LIME (seção P.3) sugere que instâncias de DC não Grave apresentam valores baixos de WMC_type, indicando classes menores e menos complexas. Não há instâncias para DC Grave, mas um erro de predição indica que WMC_type alto influenciou para isso.

Finalmente, para GC (seção Q.1), para detecção, os atributos mais relevantes são WMCNAMM_type, LOC_package e LCOM5_type, todos relacionados à complexidade, tamanho e coesão. Já para a avaliação de gravidade destaca AFTD_method, CYCLO_method e LOC_method, indicando alto acoplamento e complexidade. A análise do gráfico SHAP (seção Q.2) mostra que valores altos de LOC_package, WMCNAMM_type, LCOM5_type e AMW_type são determinantes para a detecção de GC, sugerindo classes grandes, complexas e com alta coesão. Enquanto, o gráfico LIME (seção Q.3): para instâncias de GC não Grave, valores baixos de AMW_type e NOCS_package são observados, indicando menor complexidade; para a gravidade GC, há um aumento do tamanho e diminuição do encapsulamento (LOC_type e NOAM_type); e em instâncias de GC Grave, são observados valores elevados de LOC_type, ATFD_type e CFNAMM_type (classes grandes, complexas e acopladas).

A explicabilidade para os *code smells* C# revela que atributos relacionados à complexidade (e.g., CYCLO_method e WMC_type), tamanho (e.g. LOC_method), coesão (e.g., LCOM5_type) e acoplamento (e.g., ATFD_method, RFC_method) são essenciais tanto para a detecção quanto para a avaliação de gravidade. LIME fornece *insights* detalhados sobre os diferentes níveis de gravidade, enquanto o SHAP destaca os impactos globais dos atributos no modelo. O comportamento dos atributos varia de acordo com a gravidade, com instâncias mais graves apresentando valores mais altos de complexidade e acoplamento. Assim, este resumo confirma a importância da análise de atributos de complexidade e coesão para explicar o comportamento dos modelos de detecção e avaliação de gravidade de *code smells* em projetos C#.

Concluindo, uma comparação dos resultados da explicabilidade para projetos em Java e C#, primeiramente, indica que, em ambos os casos, atributos de complexidade (e.g., CYCLO_method), tamanho (e.g., , LOC_method) e acoplamento (e.g., ATFD_method) são determinantes tanto para detecção quanto para avaliação de gravidade. As ferramentas de explicabilidade (LIME e SHAP) forneceram *insights* complementares, com LIME

detalhando níveis de gravidade e SHAP oferecendo uma visão global das influências dos atributos.

Além disso, algumas diferenças puderam ser capturadas: C# apresentou uma dependência maior de atributos relacionados à estrutura do código (e.g., WMCNAMM_type, LOC_package), enquanto Java focou mais em métricas de complexidade de métodos. A modularidade e a organização dos projetos C# parecem ter influenciado a explicabilidade, resultando em uma maior diversidade de atributos relevantes. Essa análise comparativa sugere que, embora as métricas de complexidade e acoplamento sejam universais para a detecção de *code smells*, as características específicas das linguagens e dos projetos influenciam a seleção e a importância dos atributos explicativos.

6.4 Ameaças à Validade

Diversas ameaças à validade podem afetar a integridade e a precisão dos resultados e das conclusões desta pesquisa. Esta seção resume potenciais implicações para a abordagem de detecção de *code smells* e avaliação de gravidade baseada em TA que foi utilizada.

A validade de construção deste estudo está principalmente ligada à combinação dos conjuntos de dados. Foram selecionados conjuntos bem conhecidos: i) para Java, utilizou-se o conjunto de dados de Santos, Duarte e Choren(68), que combinou os conjuntos de dados FE e LM de Fontana e Zanoni(10) com os conjuntos GC e DC de Nanda e Chhabra(14); e ii) para C#, empregou-se o conjunto de dados GC e LM de Slivka et al.(11) e os conjuntos DC e FE de Prokic et al.(22). Para mitigar esta ameaça, as instâncias duplicadas foram eliminadas e as imputações para valores ausentes realizadas, garantindo que os modelos de AM pudessem operar adequadamente. Seguiu-se o algoritmo de combinação de dados de Santos, Duarte e Choren(68) para o conjunto de dados Java, para assegurar a verificabilidade e a reprodutibilidade do processo. Além disso, as métricas selecionadas para o conjunto de dados C# foram calculadas de acordo com as especificações computacionais apresentadas por Fontana et al.(8).

Em relação à validade externa, esses novos conjuntos de dados representam *code smells* tanto em nível de método quanto em nível de classe, proporcionando uma visão mais realista de diferentes cenários de programação. Ao abordar *code smells* nesses dois níveis e garantir que os dados reflitam uma diversidade de casos, aumentou-se a validade externa dos achados. Essa abordagem fortalece a confiabilidade dos resultados, ampliando sua aplicabilidade para outros projetos e contextos de desenvolvimento de software além dos conjuntos de dados específicos utilizados, incluindo potenciais extensões para outras linguagens de programação. Adicionalmente, as modificações realizadas nos dados e nos modelos reduziram o viés de classificação, eliminando duplicatas e padronizando as escalas dos atributos para lidar com *outliers*.

Outro fator importante é a confiabilidade da conclusão, que diz respeito a ameaças que possam afetar a possibilidade de obter as mesmas conclusões se o procedimento for repetido nas mesmas condições. O principal desafio relacionado a essa categoria está na configuração experimental dos conjuntos de dados Java e C#. Para mitigar essa ameaça, todas as etapas do processo, incluindo seleção de atributos, otimização de hiperparâmetros, sobreamostragem de dados, validação de modelos e comparação e escolha das técnicas empregadas, foram detalhadas, analisadas e disponibilizadas publicamente.

6.5 Discussão

Esta seção tem como objetivo apresentar o impacto da TA na detecção e avaliação da gravidade de *code smells*, além de detalhar uma análise comparativa da abordagem proposta.

A aplicação de TA resultou em melhorias no desempenho dos MC propostos para a detecção de *code smells* e avaliação de gravidade em conjuntos de dados Java e C#. Os experimentos demonstraram que a combinação de diferentes conjuntos de dados, juntamente com o uso de técnicas avançadas de sobreamostragem, como SMOTE e B-SMOTE, além da seleção cuidadosa de atributos e a otimização de hiperparâmetros, contribuíram para uma elevação nos indicadores de desempenho, como acurácia, precisão, sensibilidade e pontuação F1 dos modelos de AM.

Especificamente, no conjunto de dados Java, observou-se um aumento na pontuação F1 (aumento de quase 15%) e na sensibilidade (aumento superior a 22%) na avaliação de gravidade de FE, sugerindo que a TA lidou adequadamente com a variabilidade do conjunto de dados. Já no conjunto de dados C#, houve ganhos nas medidas de desempenho para a avaliação de gravidade de DC e FE – a sensibilidade aumentou em 45% para DC e 56% para FE, a pontuação F1 aumentou em 28% para DC e 39% para FE e a acurácia teve ganhos em 45% e 54% para DC e FE, respectivamente –, o que sugere que os benefícios da TA se estendem para diferentes domínios de código.

Esses resultados sublinham o potencial da TA não apenas para aumentar a eficácia das estruturas de detecção de *code smells*, mas também para melhorar a avaliação de sua gravidade. Isso abre novos horizontes para a implementação de práticas de garantia de qualidade de software mais robustas e escaláveis, especialmente em cenários multidomínio.

Para a análise comparativa foram utilizados os trabalhos de Dewangan et al.(15) e Santos, Duarte e Choren(68) abordam a detecção de *code smells* e a avaliação de gravidade em conjuntos de dados Java, enquanto o trabalho de Kovacevic et al.(93) foca na detecção de *code smells* para conjuntos de dados C#. Até onde se sabe, não foram encontrados trabalhos que abordem a avaliação de gravidade de *code smells* em conjuntos de dados C#.

A Tabela 19 resume os resultados de precisão, sensibilidade e pontuação F1. Os resultados alcançados neste estudo superaram aqueles obtidos por Santos, Duarte e Choren(68) para todos os *code smells*. Comparados com Dewangan et al.(15), os resultados desta pesquisa foram melhores para DC e GC, enquanto os resultados deles foram superiores para FE e LM. É importante destacar que a pontuação F1 obtida nos experimentos deste trabalho para GC superou a de Dewangan et al.(15) em 5,56%. Além disso, a pontuação F1 (94%) para FE foi apenas 2% inferior à do trabalho de Dewangan et al.(15).

No entanto, é necessário enfatizar que a abordagem proposta, ao contrário da de Dewangan et al.(15), não negligência a influência que diferentes tipos de *code smells* podem exercer uns sobre os outros. Para isso, utilizou-se instâncias de várias gravidades de *code smell* no treinamento dos modelos de AM, aproximando-se, assim, de cenários mais realistas em softwares existentes.

Tabela 19 – Avaliação das medidas de Precisão, Sensibilidade e Pontuação F1 - TA C# para Java

Estudos	DC (P, S, F1)	GC (P, S, F1)	FE (P, S, F1)	LM (P, S, F1)
Dewangan et al.(15)	(88, 87, 87)	(90, 90, 90)	(**97, 96, 96**)	(**100, 100, 100**)
Santos, Duarte e Choren(68)	(82, 81, 81)	(81, 76, 78)	(88, 74, 79)	(80, 88, 83)
Abordagem proposta	(**90, 89, 88**)	(**97, 95, 95**)	(**97**, 93, 94)	(95, 95, 95)

A Tabela 20 mostra que a abordagem proposta de TA baseada em MC para projetos C# superou as abordagens *ML_metrics* e *ML_CodeT5* de Kovacevic et al.(93). Em comparação com *ML_metrics*, a abordagem proposta alcançou uma pontuação F1 de 95% para a detecção de LM e 97% para a detecção de GC, enquanto *ML_metrics* obteve 87% e 88%, respectivamente. Em relação ao *ML_CodeT5*, a abordagem proposta apresentou um aumento de aproximadamente 13% e 14% nas pontuações F1 para LM e GC, respectivamente. Além disso, esta abordagem alcançou resultados relevantes de pontuação F1 para a avaliação de gravidade de DC (87%) e FE (97%) em um conjunto de dados mais realista e complexo.

Concluindo, os resultados da abordagem proposta demonstram desempenho superior na detecção de *code smells* em comparação com estudos anteriores. Para o conjunto de dados Java, as métricas de precisão, sensibilidade e pontuação F1, alcançadas neste trabalho, superaram as de Santos, Duarte e Choren(68) em todos os *code smells* e, também, superaram o trabalho de Dewangan et al.(15) na detecção e avaliação da gravidade dos *code smells* DC e GC. Embora Dewangan et al.(15) tenha obtido resultados ligeiramente melhores para FE e LM, a abordagem proposta forneceu uma análise mais realista ao considerar múltiplas gravidades de *code smells* no treinamento dos modelos utilizados.

Tabela 20 – Avaliação das medidas de Precisão, Sensibilidade e Pontuação F1 - TA Java para C#

Estudos	GC (P, S, F1)	LM (P, S, F1)	DC (P, S, F1)	FE (P, S, F1)
ML_metrics(93)	(89, 88, 88)	(92, 83, 87)	(-)	(-)
ML_CodeT5(93)	(89, 82, 85)	(81, 87, 84)	(-)	(-)
Detector proposto	(**97, 97, 97**)	(**96, 94, 95**)	(100, 99, 99)	(99, 99, 99)
Avaliador de gravidade proposto	(92, 91, 91)	(88, 87, 87)	(88, 87, 87)	(100, 95, 97)

Para o conjunto de dados C#, o método proposto excedeu o desempenho das abordagens *ML_metrics* e *ML_CodeT5* de Kovacevic et al.(93), particularmente para GC e LM, obtendo uma melhoria na pontuação F1 de aproximadamente 12%. Além disso, a abordagem produziu altas pontuações F1 para avaliação da gravidade de DC e FE, validando sua eficácia em cenários mais complexos e realistas. No geral, os resultados deste trabalho refletem a robustez e a generalização da abordagem proposta, oferecendo avanços para a detecção e avaliação da gravidade de *code smells*.

7 CONCLUSÃO

Avaliar a gravidade de *code smells* é essencial para categorizar problemas e priorizar esforços de manutenção, o que impacta diretamente o ciclo de vida do software. Neste estudo, foi proposta uma abordagem para detectar e classificar *code smells* utilizando MC em duas etapas, baseados em TA. A primeira etapa foca na detecção do *code smell*, enquanto a segunda avalia sua gravidade. Utilizou-se modelos de alta precisão para detecção e adotou-se técnicas de sobreamostragem para lidar com o desbalanceamento de dados na avaliação de gravidade. Além disso, foram empregados conjuntos de dados bem estabelecidos de projetos em Java e C#, buscando garantir a representatividade dos resultados em cenários reais e diferentes contextos de programação.

Este trabalho enfrentou algumas limitações observadas em estudos anteriores na detecção e avaliação de *code smells*, particularmente nas etapas de pré-processamento de dados e na aplicação de métodos de AM. Em relação ao balanceamento de dados, a utilização de técnicas como o SMOTE nem sempre proporcionou melhorias no desempenho dos modelos, e em alguns casos introduziu vieses, não conseguindo garantir um balanceamento adequado entre as classes, especialmente devido a restrições de recursos computacionais. No que tange à seleção de atributos, houve dificuldades em assegurar a representatividade dos dados, e o uso do teste de qui-quadrado apresentou problemas ao lidar com dados de baixa frequência, comprometendo a precisão dos resultados. Além disso, as técnicas de escalonamento de dados, como a normalização, não demonstraram impacto relevante na eficácia dos modelos. Estudos comparativos também foram insuficientes, não analisando adequadamente outras abordagens, como a padronização, o que limita a análise do impacto dessas técnicas.

Os MC demonstraram desempenho comparável ou superior aos algoritmos de AP, na detecção e avaliação da gravidade de *code smells*, com o custo computacional menor. No entanto, uma limitação desses métodos foi a falta de explicabilidade, o que dificulta a interpretação dos processos e resultados. No contexto da TA, a ausência de padronização e rotulação dos conjuntos de dados prejudicou a generalização dos modelos. O uso de dados sintéticos para pré-treinamento pode ter introduzido vieses, e a adaptação dos modelos para diferentes contextos de projetos apresentou desafios, com benefícios limitados. Além disso, a análise dos estudos relacionados revelou que as abordagens existentes, para detecção de *code smells* e avaliação de gravidade, geralmente focam em apenas um tipo de *code smell*, sem considerar a interação entre diferentes tipos, o que reduz a aplicabilidade prática dos resultados obtidos.

Depois de enfrentar essas limitações, os resultados deste trabalho indicam que o TA aprimorou a generalização dos modelos de AM utilizados. A precisão da detecção

no conjunto de dados C# foi semelhante ou até melhor do que os resultados obtidos no conjunto de dados Java, com perda mínima de desempenho. Os modelos avaliaram efetivamente a gravidade do *code smell* em diferentes níveis, demonstrando robustez no tratamento de distribuições de dados variáveis entre projetos Java e C#.

Quanto a explicabilidade, a análise dos atributos mais impactantes para detecção e avaliação de gravidade dos *code smells* Java e C# destaca o uso de atributos relacionados à complexidade, tamanho, acoplamento e coesão. Entretanto, C# apresentou maior dependência de atributos estruturais, enquanto Java focou mais em métricas de complexidade de métodos. A organização dos projetos em C# favoreceu maior diversidade de atributos relevantes. Isso indica que, embora métricas de complexidade e acoplamento sejam universais para *code smells*, características das linguagens e dos projetos influenciam a importância dos atributos explicativos. As ferramentas de explicabilidade, como LIME e SHAP, ofereceram *insights* complementares: LIME detalhou níveis de gravidade, enquanto SHAP proporcionou uma visão global das influências dos atributos para detecção de *code smells*.

Assim, a abordagem proposta teve resultados relevantes comparado à trabalhos anteriores nesta área de pesquisa. Por um lado, na comparação com o trabalho baseado em projetos Java, Dewangan et al.(15), os resultados de pontuação F1 de 95% para GC e 88% para DC superaram os deles em 5,56% e 1,15%, respectivamente. No entanto, os resultado de pontuação F1 (94%) para FE foi superado pelo resultado de Dewangan et al.(15) (96%) em apenas 2%. Por outro lado, em comparação com o trabalho baseado em projetos C#, o método proposto superou a abordagem *ML_metrics* de Kovacevic et al.(93), alcançando uma pontuação F1 de 95% para LM e 97% para detecção de GC. Além disso, a abordagem proposta superou a abordagem *ML_CodeT5* de Kovacevic et al.(93) em aproximadamente 13% e 14%, respectivamente, para os valores de pontuação F1 de LM e GC. Vale destacar, também, que a abordagem deste trabalho alcançou os seguintes resultados de pontuação F1 para avaliação de gravidade DC (87%) e FE (97%). Além dos resultados mencionados anteriormente, as principais descobertas deste trabalho são:

- a abordagem multiclasse foi aprimorada apesar do desequilíbrio entre a quantidade de gravidades menores e maiores de *code smell* presentes nos conjuntos de dados. As técnicas de sobreamostragem SMOTE e B-SMOTE ajudaram a equilibrar os dados de treinamento sem comprometer a qualidade da amostra ou causar vazamento de dados.

- A normalização de dados é crucial para a seleção de atributos usando a técnica Qui-quadrado. No entanto, para a seleção de atributos baseada em ANOVA, o escalonamento de dados (normalização ou padronização) é desnecessário, pois melhores resultados podem ser obtidos sem a aplicação desta técnica de pré-processamento de

dados.

Dados esses resultados, a abordagem proposta aprimora a detecção da qualidade do código e fornece *insights* valiosos sobre os padrões de *code smell*, que são essenciais para desenvolvedores que buscam melhorar a manutenibilidade do código e o desempenho do software. Além disso, os resultados alcançados neste trabalho destacam o potencial da TA não apenas para aprimorar a eficácia das estruturas de detecção de *code smells*, mas também para melhorar a avaliação de sua gravidade.

Neste trabalho, a TA foi aplicada para aprimorar a detecção de *code smells* em dois conjuntos de dados de diferentes linguagens: Java e C#. O domínio de origem (Java) transferiu conhecimento para o domínio de destino (C#), alinhando atributos preditores para garantir consistência ($X_S = X_T$, $Y_S = Y_T$). A TA utilizada é categorizada como Transferência Indutiva, por envolver rótulos nos dois domínios, e Transferência Homogênea, devido à correspondência entre os atributos.

Ajustes nos atributos preditores de C# garantiram uma representação comum, caracterizando a abordagem como Baseada em Atributos e permitindo melhorias na detecção de *code smells* no domínio de destino. Os experimentos mostraram que a combinação de conjuntos de dados, técnicas como SMOTE e B-SMOTE, seleção de atributos e otimização de hiperparâmetros elevaram os indicadores de desempenho dos modelos de AM.

Além disso, a pesquisa incentiva novos estudos em garantia de qualidade de software, oferecendo publicamente os dados e métodos utilizados, ampliando sua aplicabilidade em cenários multidomínio.

Pesquisas futuras podem explorar técnicas adicionais de AM, expandir o conjunto de dados para incluir mais linguagens de programação e testar a abordagem em diferentes contextos de desenvolvimento de software. O uso de outros MC, arquiteturas de redes neurais mais complexas, ou modelos de linguagem grande escala (LLMs) pode trazer avanços significativos, especialmente considerando a capacidade dos LLMs de capturar contextos complexos e gerar *insights* detalhados sobre o código.

Além disso, futuras pesquisas poderiam adotar uma abordagem híbrida, combinando representações baseadas em métricas com *embeddings* pré-treinados, como os gerados por LLMs, e integrar múltiplos conjuntos de dados rotulados por especialistas para aumentar a robustez e consistência. A inclusão de *feedback* dos desenvolvedores no processo de treinamento e o uso de LLMs para personalizar as detecções e avaliações também poderiam aumentar a precisão e relevância das ferramentas para detectar e avaliar a gravidade de *code smells*.

REFERÊNCIAS

1 FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201485672.

2 LANZA, M.; MARINESCU, R. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. [S.l.]: Springer Science & Business Media, 2007.

3 FONTANA, F. A.; FERME, V.; ZANONI, M.; ROVEDA, R. Towards a prioritization of code debt: A code smell intensity index. In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. [S.l.: s.n.], 2015. p. 16–24.

4 MOHA, N.; GUEHENEUC, Y.-G.; DUCHIEN, L.; MEUR, A.-F. L. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, v. 36, n. 1, p. 20–36, 2010.

5 TSANTALIS, N.; CHATZIGEORGIOU, A. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, v. 35, n. 3, p. 347–367, 2009.

6 TSANTALIS, N.; CHATZIGEORGIOU, A. Ranking refactoring suggestions based on historical volatility. In: *2011 15th European Conference on Software Maintenance and Reengineering*. [S.l.: s.n.], 2011. p. 25–34.

7 PALOMBA, F.; NUCCI, D. D.; TUFANO, M.; BAVOTA, G.; OLIVETO, R.; POSHYVANYK, D.; LUCIA, A. D. Landfill: An open dataset of code smells with public evaluation. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2015. p. 482–485.

8 FONTANA, F. A.; MäNTYLä, M. V.; ZANONI, M.; MARINO, A. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 21, n. 3, p. 11431191, jun 2016. ISSN 1382-3256.

9 PECORELLI, F.; PALOMBA, F.; NUCCI, D. D.; LUCIA, A. D. Comparing heuristic and machine learning approaches for metric-based code smell detection. In: *IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. [S.l.]: IEEE Computer Society, 2019. v. 2019-May, p. 93104. ISBN 978-172811519-1.

10 FONTANA, F. A.; ZANONI, M. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, Elsevier B.V., v. 128, p. 4358, 2017. ISSN 09507051.

11 SLIVKA, J.; LUBURI, N.; PROKI, S.; GRUJI, K.-G.; KOVAEVI, A.; SLADI, G.; VIDAKOVI, D. Towards a systematic approach to manual annotation of code smells. *Science of Computer Programming*, v. 230, p. 102999, 2023. ISSN 0167-6423. 20/03/2024. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642323000813>.

12 LIU, H.; JIN, J.; XU, Z.; ZOU, Y.; BU, Y.; ZHANG, L. Deep learning based code smell detection. *IEEE Transactions on Software Engineering*, Institute of Electrical and Electronics Engineers Inc., v. 47, n. 9, p. 18111837, 2021. ISSN 00985589.

13 ALKHARABSHEH, K.; ALAWADI, S.; KEBANDE, V. R.; CRESPO, Y.; FERNáNDEZ-DELGADO, M.; TABOADA, J. A. A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of god class. *Information and Software Technology*, Elsevier B.V., v. 143, p. 106736, 2022. ISSN 09505849. 01/04/2024. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584921001865>.

14 NANDA, J.; CHHABRA, J. K. Sshm: Smote-stacked hybrid model for improving severity classification of code smell. *International Journal of Information Technology (Singapore)*, Springer Science and Business Media B.V., v. 14, n. 5, p. 27012707, 2022. ISSN 25112104.

15 DEWANGAN, S.; RAO, R. S.; CHOWDHURI, S. R.; GUPTA, M. Severity classification of code smells using machine-learning methods. *SN Computer Science*, Springer, v. 4, n. 5, 2023. ISSN 2662995X. 20/03/2024. Disponível em: <https://doi.org/10.1007/s42979-023-01979-8>.

16 ZAKERI-NASRABADI, M.; PARSA, S.; ESMAILI, E.; PALOMBA, F. A systematic literature review on the code smells datasets and validation mechanisms. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 55, n. 13s, jul 2023. ISSN 0360-0300. 02/03/2024. Disponível em: <https://doi.org/10.1145/3596908>.

17 TAHIR, A.; DIETRICH, J.; COUNSELL, S.; LICORISH, S.; YAMASHITA, A. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. *Information and Software Technology*, v. 125, p. 106333, 2020. ISSN 0950-5849. 04/04/2024. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584920300926>.

18 ABUHASSAN, A.; ALSHAYEB, M.; GHOUTI, L. Software smell detection techniques: A systematic literature review. *Journal of Software: Evolution and Process*, John Wiley and Sons Ltd, v. 33, n. 3, 2021. ISSN 20477481. 04/03/2024. Disponível em: <https://doi.org/10.1002/smr.2320>.

19 NUCCI, D. D.; PALOMBA, F.; TAMBURRI, D. A.; SEREBRENIK, A.; LUCIA, A. D. Detecting code smells using machine learning techniques: Are we there yet? In: *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2018. v. 2018-March, p. 612621. ISBN 978-153864969-5.

20 POLIKAR, R. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, v. 6, n. 3, p. 21–45, 2006.

21 ROKACH, L. Ensemble-based classifiers. *Artificial Intelligence Review*, v. 33, p. 1–39, 2010.

22 PROKIC, S.; LUBURI, N.; SLIVKA, J.; KOVAEVI, A. Prescriptive procedure for manual code smell annotation. *Science of Computer Programming*, v. 238, p. 103168, 2024. ISSN 0167-6423. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642324000911>.

23 BENGIO, Y.; COURVILLE, A.; VINCENT, P. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 35, n. 8, p. 1798–1828, 2013.

24 ABDOU, A.; DARWISH, N. Severity classification of software code smells using machine learning techniques: A comparative study. *Journal of Software: Evolution and Process*, John Wiley and Sons Ltd, v. 36, 2022. ISSN 20477481. 04/03/2024. Disponível em: <https://doi.org/10.1002/smr.2454>.

25 GUPTA, A.; CHAUHAN, N. K. A severity-based classification assessment of code smells in kotlin and java application. *Arabian Journal for Science and Engineering*, Springer Science and Business Media Deutschland GmbH, v. 47, n. 2, p. 18311848, 2022. ISSN 2193567X.

26 RAO, R. S.; DEWANGAN, S.; MISHRA, A.; GUPTA, M. A study of dealing class imbalance problem with machine learning methods for code smell severity detection using pca-based feature selection technique. *Scientific Reports*, Nature Research, v. 13, n. 1, 2023. ISSN 20452322. 02/03/2024. Disponível em: <https://doi.org/10.1038/s41598-023-43380-8>.

27 HU, W.; LIU, L.; YANG, P.; ZOU, K.; LI, J.; LIN, G.; XIANG, J. Revisiting "code smell severity classification using machine learning techniques". In: H., S.; Y., T.; A., C.; M., S.; D., T.; AKM.J.A., M.; H., K.; J.-J., Y.; M., T.; N., S.; R., B.; S.I., A. (Ed.). *IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. [S.l.]: IEEE Computer Society, 2023. v. 2023-June, p. 840849. ISBN 979-835032697-0. ISSN 07303157.

28 AZEEM, M. I.; PALOMBA, F.; SHI, L.; WANG, Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, v. 108, p. 115–138, 2019. ISSN 0950-5849.

29 CUNNINGHAM, W. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, Association for Computing Machinery, New York, NY, USA, v. 4, n. 2, p. 2930, dez. 1992. ISSN 1055-6400. Disponível em: <https://doi.org/10.1145/157710.157715>.

30 YAMASHITA, A.; MOONEN, L. To what extent can maintenance problems be predicted by code smell detection? an empirical study. *Information and Software Technology*, v. 55, n. 12, p. 2223–2242, 2013. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584913001614>.

31 TUFANO, M.; PALOMBA, F.; BAVOTA, G.; OLIVETO, R.; PENTA, M. D.; LUCIA, A. D.; POSHYVANYK, D. When and why your code starts to smell bad. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.: s.n.], 2015. v. 1, p. 403–414.

32 ALBUQUERQUE, D.; GUIMARãES, E.; PERKUSICH, M.; ALMEIDA, H.; PERKUSICH, A. An approach for integrating interactive detection of code smells on agile software development. In: *2023 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. [S.l.: s.n.], 2023. p. 1–6.

33 FACELI, K.; LORENA, A. C.; GAMA, J.; ALMEIDA, T. A. d.; CARVALHO, A. C. P. d. L. F. d. *Inteligência artificial: uma abordagem de aprendizado de máquina*. Brazil: LTC, 2021.

34 PARASHAR, A.; PARASHAR, A.; DING, W.; SHABAZ, M.; RIDA, I. Data preprocessing and feature selection techniques in gait recognition: A comparative study of machine learning and deep learning approaches. *Pattern Recognition Letters*, v. 172, p. 65–73, 2023. ISSN 0167-8655.

35 SANTOS, F. R.; CHOREN, R. Data preprocessing for machine learning based code smell detection: A systematic literature review. *Available at SSRN 4756315.*

36 PECORELLI, F.; NUCCI, D. D.; ROOVER, C. D.; LUCIA, A. D. On the role of data balancing for machine learning-based code smell detection. In: F.A., F.; B., W.; A., A.; F., P.; G., P.; M., A.; M., C.; X., D. (Ed.). *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2019).* [S.l.]: Association for Computing Machinery, Inc, 2019. p. 1924. ISBN 978-145036855-1.

37 CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, AI Access Foundation, El Segundo, CA, USA, v. 16, n. 1, p. 321357, jun 2002. ISSN 1076-9757.

38 MONIZ, N.; BRANCO, P.; TORGO, L. Resampling strategies for imbalanced time series. In: *IEEE International Conference on Data Science and Advanced Analytics (DSAA).* [S.l.: s.n.], 2016. p. 282–291.

39 BATISTA, G. E. A. P. A.; PRATI, R. C.; MONARD, M. C. A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explor. Newsl.*, Association for Computing Machinery, New York, NY, USA, v. 6, n. 1, p. 2029, jun. 2004. ISSN 1931-0145. Disponível em: <https://doi.org/10.1145/1007730.1007735>.

40 MALHOTRA, R.; JAIN, B.; KESSENTINI, M. Examining deep learnings capability to spot code smells: A systematic literature review. *Cluster Computing*, Kluwer Academic Publishers, USA, v. 26, n. 6, p. 34733501, oct 2023. ISSN 1386-7857.

41 ZHANG, D.; SONG, S.; ZHANG, Y.; LIU, H.; SHEN, G. Code smell detection research based on pre-training and stacking models. *IEEE Latin America Transactions*, v. 22, n. 1, p. 22–30, 2024.

42 HAN, H.; WANG, W.-Y.; MAO, B.-H. Borderline-smote: a new over-sampling method in imbalanced data sets learning. In: *Proceedings of the 2005 International Conference on Advances in Intelligent Computing - Volume Part I.* Berlin, Heidelberg: Springer-Verlag, 2005. (ICIC'05), p. 878887. ISBN 3540282262. Disponível em: <https://doi.org/10.1007/11538059_91>.

43 ROMERO, E.; SOPENA, J. M. Performing feature selection with multilayer perceptrons. *IEEE Transactions on Neural Networks*, v. 19, n. 3, p. 431–441, 2008.

44 DEWANGAN, S.; RAO, R. S.; MISHRA, A.; GUPTA, M. A novel approach for code smell detection: An empirical study. *IEEE Access*, Institute of Electrical and Electronics Engineers Inc., v. 9, p. 162869162883, 2021. ISSN 21693536. 25/03/2024. Disponível em: <https://doi.org/10.1109/ACCESS.2021.3133810>.

45 REIS, J. Pereira dos; ABREU, F. Brito e; CARNEIRO, G. d. F. Code smells incidence: Does it depend on the application domain? In: *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC).* [S.l.: s.n.], 2016. p. 172–177.

46 ALI, P. M.; FARAJ, R. *Data Normalization and Standardization: A Technical Report.* [S.l.]: Machine Learning Technical Reports, 2014.

47 AFRIN, M.; ASMA, S. A.; AKHTER, N.; RIDOY, J. H.; SAUDA, S. S.; TAHER, K. A. A hybrid approach to investigate anti-pattern from source code. In: *25th International Conference on Computer and Information Technology (ICCIT)*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2022. p. 888892. ISBN 979-835034602-2.

48 RISH, I. et al. An empirical study of the naive bayes classifier. In: *IJCAI 2001 workshop on empirical methods in artificial intelligence*. [S.l.: s.n.], 2001.

49 JAKKULA, V. Tutorial on support vector machine (svm). *School of EECS, Washington State University*, v. 37, n. 2.5, p. 3, 2006.

50 PETERSON, L. E. K-nearest neighbor. *Scholarpedia*, v. 4, n. 2, p. 1883, 2009.

51 JR, D. W. H.; LEMESHOW, S.; STURDIVANT, R. X. *Applied logistic regression*. [S.l.]: John Wiley & Sons, 2013. v. 398.

52 CHAITRA, P.; KUMAR, R. S. A review of multi-class classification algorithms. *Int. J. Pure Appl. Math*, v. 118, n. 14, p. 17–26, 2018.

53 SHARMA, T.; EFSTATHIOU, V.; LOURIDAS, P.; SPINELLIS, D. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, Elsevier Inc., v. 176, 2021. ISSN 01641212. 05/03/2024. Disponível em: <https://doi.org/10.1016/j.jss.2021.110936>.

54 ALAZBA, A.; ALJAMAAN, H. Code smell detection using feature selection and stacking ensemble: An empirical investigation. *Information and Software Technology*, Elsevier B.V., v. 138, p. 106648, 2021. ISSN 09505849. 01/03/2024. Disponível em: <https://doi.org/10.1016/j.infsof.2021.106648>.

55 WANG, S.; MINKU, L. L.; YAO, X. Resampling-based ensemble methods for online class imbalance learning. *IEEE Transactions on Knowledge and Data Engineering*, v. 27, n. 5, p. 1356–1368, 2015.

56 SAGI, O.; ROKACH, L. Ensemble learning: A survey. *WIREs Data Mining and Knowledge Discovery*, v. 8, n. 4, p. e1249, 2018. 22/04/2024. Disponível em: <https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1249>.

57 M., L. F.; ., A.; ., D.; ., C. R.; ., M. C. Métricas científicas em estudos bibliométricos: detecção de outliers para dados univariados. *Em Questão*, v. 23, p. 254–273, 2017. ISSN 1807-8893. Disponível em: <https://www.redalyc.org/articulo.oa?id=465650499014>.

58 FREUND, Y.; SCHAPIRE, R. E. A decision-theoretic generalization of on-line learning and an application to boosting. In: *Proceedings of the Second European Conference on Computational Learning Theory*. Berlin, Heidelberg: Springer-Verlag, 1995. (EuroCOLT '95, 2), p. 2337. ISBN 3540591192.

59 LUIZ, F. C.; OLIVEIRA, B. R. D.; PARREIRAS, F. S. Machine learning techniques for code smells detection: An empirical experiment on a highly imbalanced setup. In: S. de Avila e S.; R., S.; D.V. dos S.; F., G. da R.; I., V.; A.P., G.; C., B.; F.O., V.; S., S.; H., C.; F., B.; A., B.; L.R., C.; L.G., A.; M., C.; F.S., B. (Ed.). *Proceedings of the XV Brazilian Symposium on Information Systems (SBSI '19)*. [S.l.]: Association for Computing Machinery, 2019. p. 18. ISBN 978-145037237-4.

60 MCCARTY, D. A.; KIM, H. W.; LEE, H. K. Evaluation of light gradient boosted machine learning technique in large scale land use and land cover classification. *Environments*, v. 7, n. 10, 2020. ISSN 2076-3298. 22/04/2024. Disponível em: <https://doi.org/10.3390/environments710008>.

61 WANG, Z.; REN, H.; LU, R.; HUANG, L. Stacking based lightgbm-catboost-randomforest algorithm and its application in big data modeling. In: *4th International Conference on Data-driven Optimization of Complex Systems (DOCS)*. Chengdu, China: IEEE, 2022. p. 1–6.

62 ZHUANG, F.; QI, Z.; DUAN, K.; XI, D.; ZHU, Y.; ZHU, H.; XIONG, H.; HE, Q. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, PP, p. 1–34, 07 2020.

63 WEISS, K.; KHOSHGOFTAAR, T. M.; WANG, D. A survey of transfer learning. *Journal of Big Data*, v. 3, 2016. 03/03/2024. Disponível em: <https://doi.org/10.1186/s40537-016-0043-6>.

64 LUNDBERG, S. M.; LEE, S.-I. A unified approach to interpreting model predictions. In: GUYON, I.; LUXBURG, U. V.; BENGIO, S.; WALLACH, H.; FERGUS, R.; VISHWANATHAN, S.; GARNETT, R. (Ed.). *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017. p. 4765–4774. Disponível em: <http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf>.

65 LI, Y.; XIAO, Y.; GONG, Y.; ZHANG, R.; HUO, Y.; WU, Y. Explainable ai: A way to achieve trustworthy ai. In: *2024 IEEE 10th Conference on Big Data Security on Cloud (BigDataSecurity)*. [S.l.: s.n.], 2024. p. 150–155.

66 LIANG, W.; LI, Y.; XU, J.; QIN, Z.; ZHANG, D.; LI, K.-C. Qos prediction and adversarial attack protection for distributed services under dlaas. *IEEE Transactions on Computers*, v. 73, n. 3, p. 669–682, 2024.

67 RIBEIRO, M. T.; SINGH, S.; GUESTRIN, C. "why should I trust you?": Explaining the predictions of any classifier. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. [S.l.: s.n.], 2016. p. 1135–1144.

68 SANTOS, F.; DUARTE, J.; CHOREN, R. Code smell severity classification at class and method level with a single manually labeled imbalanced dataset. In: *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*. Porto Alegre, RS, Brasil: SBC, 2024. p. 12–23. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/sbes/article/view/30345>.

69 BOUTAIB, S.; BECHIKH, S.; PALOMBA, F.; ELARBI, M.; MAKHLOUF, M.; SAID, L. B. Code smell detection and identification in imbalanced environments. *Expert Systems with Applications*, v. 166, p. 114076, 2021. ISSN 0957-4174.

70 PATNAIK, A.; PADHY, N. Does code complexity affect the quality of real-time projects? detection of code smell on software projects using machine learning algorithms. In: *Proceedings of the International Conference on Data Science, Machine Learning and Artificial Intelligence*. New York, NY, USA: Association for Computing Machinery, 2022. (DSMLAI '21', 21), p. 178185. ISBN 9781450387637.

71 YU, J.; MAO, C.; YE, X. A novel tree-based neural network for android code smells detection. In: *IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. [S.l.]: Institute of Electrical and Electronics Engineers, 2021. v. 2021-December, p. 738748. ISBN 978-166545813-9. ISSN 26939177.

72 AMORIM, L.; COSTA, E.; ANTUNES, N.; FONSECA, B.; RIBEIRO, M. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In: *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE)*. USA: IEEE Computer Society, 2015. (ISSRE '15), p. 261269. ISBN 9781509004065.

73 MHAWISH, M. Y.; GUPTA, M. Predicting code smells and analysis of predictions: Using machine learning techniques and software metrics. *Journal of Computer Science and Technology*, Springer, v. 35, n. 6, p. 14281445, 2020. ISSN 10009000.

74 DEWANGAN, S.; RAO, R. S.; MISHRA, A.; GUPTA, M. Code smell detection using ensemble machine learning algorithms. *Applied Sciences (Switzerland)*, MDPI, v. 12, n. 20, 2022. ISSN 20763417. 04/04/2024. Disponível em: <https://www.mdpi.com/2076-3417/12/20/10321>.

75 CRUZ, D.; SANTANA, A.; FIGUEIREDO, E. Detecting bad smells with machine learning algorithms: An empirical study. In: *IEEE/ACM International Conference on Technical Debt (TechDebt)*. [S.l.]: Association for Computing Machinery, Inc, 2020. p. 3140. ISBN 978-145037960-1.

76 SUKKASEM, P.; SOOMLEK, C. Enhanced machine learning-based code smell detection through hyper-parameter optimization. In: *20th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2023. p. 297302. ISBN 979-835030050-5.

77 CHEN, H.; REN, Z.; QIAO, L.; ZHOU, Z.; GAO, G.; MA, Y.; JIANG, H. Adaboost-based refused bequest code smell detection with synthetic instances. In: *7th International Conference on Dependable Systems and Their Applications (DSA)*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2020. p. 7889. ISBN 978-073812422-3.

78 BARBEZ, A.; KHOMH, F.; GUEHENEUC, Y.-G. Deep learning anti-patterns from code metrics history. In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2019. p. 114124. ISBN 978-172813094-1.

79 KAUR, I.; KAUR, A. A novel four-way approach designed with ensemble feature selection for code smell detection. *IEEE Access*, Institute of Electrical and Electronics Engineers Inc., v. 9, p. 86958707, 2021. ISSN 21693536.

80 LIU, H.; XU, Z.; ZOU, Y. Deep learning based feature envy detection. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018. (ASE '18, 33), p. 385396. ISBN 9781450359375.

81 ALAZBA, A.; ALJAMAAN, H.; ALSHAYEB, M. Deep learning approaches for bad smell detection: a systematic literature review. *Empirical Software Engineering*, v. 28, 05 2023. 01/03/2024. Disponível em: <https://doi.org/10.1007/s10664-023-10312-z>.

82 HADJ-KACEM, M.; BOUASSIDA, N. Deep representation learning for code smells detection using variational auto-encoder. In: *International Joint Conference on Neural Networks (IJCNN)*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2019. v. 2019-July, p. 1–8. ISBN 978-172811985-4.

83 GUO, X.; SHI, C.; JIANG, H. Deep semantic-based feature envy identification. In: *Proceedings of the 11th Asia-Pacific Symposium on Internetware (Internetware '19)*. [S.l.]: Association for Computing Machinery, 2019. p. 16. ISBN 978-145037701-0.

84 HAMDY, A.; TAZY, M. Deep hybrid features for code smells detection. *Journal of Theoretical and Applied Information Technology*, Little Lion Scientific, v. 98, n. 14, p. 26842696, 2020. ISSN 19928645.

85 HO, A.; BUI, A. M. T.; NGUYEN, P. T.; SALLE, A. D. Fusion of deep convolutional and lstm recurrent neural networks for automated detection of code smells. In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE '23)*. [S.l.]: Association for Computing Machinery, 2023. p. 229234. ISBN 979-840070044-6.

86 KOVACEVIC, A.; SLIVKA, J.; VIDAKOVI, D.; GRUJI, K.-G.; LUBURI, N.; PROKI, S.; SLADI, G. Automatic detection of long method and god class code smells through neural source code embeddings. *Expert Systems with Applications*, Elsevier Ltd, v. 204, 2022. ISSN 09574174.

87 STEFANO, M. D.; PECORELLI, F.; PALOMBA, F.; LUCIA, A. D. Comparing within-and cross-project machine learning algorithms for code smell detection. In: A., A.; D., F.; G., C.; V., L. (Ed.). *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution (MaLTESQuE 2021)*. [S.l.]: Association for Computing Machinery, Inc, 2021. p. 16. ISBN 978-145038625-8.

88 REN, S.; SHI, C.; ZHAO, S. Exploiting multi-aspect interactions for god class detection with dataset fine-tuning. In: W.K., C.; B., C.; H., T.; J.-J., Y.; Y., T.; D., T.; S., S.; H., S.; S., R.; S.I., A. (Ed.). *IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2021. p. 864873. ISBN 978-166542463-9.

89 GUPTA, R.; SINGH, S. K. A novel transfer learning method for code smell detection on heterogeneous data: A feasibility study. *SN Computer Science*, Springer, v. 4, n. 6, 2023. ISSN 2662995X. 03/03/2024. Disponível em: <https://doi.org/10.1007/s42979-023-02157-6>.

90 MA, W.; YU, Y.; RUAN, X.; CAI, B. Pre-trained model based feature envy detection. In: *IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2023. p. 430440. ISBN 979-835031184-6.

91 WANG, Y.; WANG, W.; JOTY, S.; HOI, S. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. [S.l.: s.n.], 2021. p. 8696–8708.

92 RAFFEL, C.; SHAZEER, N.; ROBERTS, A.; LEE, K.; NARANG, S.; MATENA, M.; ZHOU, Y.; LI, W.; LIU, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, JMLR.org, v. 21, n. 1, jan. 2020. ISSN 1532-4435.

93 KOVACEVIC, A.; LUBURI, N.; SLIVKA, J.; PROKI, S.; GRUJI, K.-G.; VIDAKOVI, D.; SLADI, G. Automatic detection of code smells using metrics and codet5 embeddings: a case study in c#. *Neural Comput. Appl.*, Springer-Verlag, Berlin, Heidelberg, v. 36, n. 16, p. 92039220, feb 2024. ISSN 0941-0643. Disponível em: <https://doi.org/10.1007/s00521-024-09551-y>.

94 KANADE, A.; MANIATIS, P.; BALAKRISHNAN, G.; SHI, K. Learning and evaluating contextual embedding of source code. In: *Proceedings of the 37th International Conference on Machine Learning*. [S.l.]: JMLR.org, 2020. (ICML'20).

95 MADEYSKI, L.; LEWOWSKI, T. Mlcq: Industry-relevant code smell data set. In: *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (EASE '20, 24), p. 342347. ISBN 9781450377317. 03/04/2024. Disponível em: <https://doi.org/10.1145/3383219.3383264>.

96 SHEN, L.; LIU, W.; CHEN, X.; GU, Q.; LIU, X. Improving machine learning-based code smell detection via hyper-parameter optimization. In: *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. [S.l.: s.n.], 2020. p. 276–285.

97 TEMPERO, E.; ANSLOW, C.; DIETRICH, J.; HAN, T.; LI, J.; LUMPE, M.; MELTON, H.; NOBLE, J. The qualitas corpus: A curated collection of java code for empirical studies. In: *2010 Asia Pacific Software Engineering Conference*. Sydney, NSW, Australia: IEEE, 2010. p. 336–345.

98 MARINESCU, C.; MARINESCU, R.; MIHANCEA, P.; RATIU, D.; WETTEL, R. iplasma: An integrated platform for quality assessment of object-oriented design. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary*. [S.l.]: IEEE, 2005. v. 1, n. 14, p. 77–80.

99 NONGPONG, K. *Integrating "code smells"detection with refactoring tool support*. Tese (Doutorado) — University of Wisconsin at Milwaukee, USA, 2012. AAI3523928.

100 VIDAL, S.; VAZQUEZ, H.; DIAZ-PACE, J. A.; MARCOS, C.; GARCIA, A.; OIZUMI, W. Jspirit: a flexible tool for the analysis of code smells. In: *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*. Santiago, Chile: IEEE, 2015. p. 1–6.

101 MARINESCU, R. Measurement and quality in object-oriented design. In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*. Budapest, Hungary: IEEE, 2005. p. 701–704.

102 PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; FASANO, F.; OLIVETO, R.; LUCIA, A. D. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, Springer New York LLC, v. 23, n. 3, p. 1188 1221, 2018. ISSN 13823256. Cited by:

187; All Open Access, Green Open Access, Hybrid Gold Open Access. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85026909182&doi=10.1007%2fs10664-017-9535-z&partnerID=40&md5=395ef0f7bfbcd60cc9aca55d34d3a88a>.

103 LACERDA, G.; PETRILLO, F.; PIMENTA, M.; GUéHéNEUC, Y. G. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, v. 167, p. 110610, 2020. ISSN 0164-1212. 25/03/2024. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121220300881>.

104 PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D. Do they really smell bad? a study on developers' perception of bad code smells. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. Victoria, BC, Canada: IEEE, 2014. p. 101–110.

105 LORENZ, M.; KIDD, J. *Object-oriented software metrics: a practical guide*. USA: Prentice-Hall, Inc., 1994. ISBN 013179292X.

106 FERME, V. *JCodeOdor: A Software Quality Advisor Through Design Flaws Detection*. Tese (Doutorado) — Università degli Studi di Milano-Bicocca, 09 2013.

107 EMDEN, E. van; MOONEN, L. Java quality assurance by detecting code smells. In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* Richmond, VA, USA: IEEE, 2002. p. 97–106.

108 CIUPKE, O. Automatic detection of design problems in object-oriented reengineering. In: *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30 (Cat. No.PR00278)*. Santa Barbara, CA, USA: IEEE, 1999. p. 18–32.

109 MCCABE, T. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2, n. 4, p. 308–320, 1976.

110 LANZA, R. M. M. *Object-Oriented Metrics in Practice*. Berlin, Heidelberg: Springer, 2006. XIV, 207 p.

111 BRIAND, L.; DALY, J.; WUST, J. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, v. 25, n. 1, p. 91–121, 1999.

112 BROWN, W. H.; MALVEAU, R. C.; MCCORMICK, H. W. S.; MOWBRAY, T. J. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st. ed. USA: John Wiley & Sons, Inc., 1998. ISBN 0471197130.

113 CATAL, C. Performance evaluation metrics for software fault prediction studies. *Acta Polytechnica Hungarica*, v. 9, 01 2012. 05/04/2024. Disponível em: <https://www.semanticscholar.org/paper/Performance-Evaluation-Metrics-for-Software-Fault-Catal/dbd4b674fbd4bb5bc91b95a59eac9d1f3f81209b>.

114 KITCHENHAM, B. A.; CHARTERS, S. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. [S.l.], 2007.

115 KITCHENHAM, B.; Pearl Brereton, O.; BUDGEN, D.; TURNER, M.; BAILEY, J.; LINKMAN, S. Systematic literature reviews in software engineering a systematic literature review. *Information and Software Technology*, v. 51, n. 1, p. 7–15, 2009. ISSN 0950-5849. Special Section - Most Cited Articles in 2002 and Regular Research Papers.

116 WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (EASE '14,), p. 10. ISBN 9781450324762.

APÊNDICE A – METODOLOGIA DA REVISÃO SISTEMÁTICA DA LITERATURA

Para obter uma melhor compreensão das abordagens de pré-processamento de dados para esse tipo de detecção, foi conduzida uma RSL, na qual foram examinados e sintetizados empiricamente as práticas e soluções atuais para pré-processamento de dados de *code smell*. Para conduzir esta RSL, foi seguida a orientação metodológica fornecida por Kitchenham e Charters(114) para ajudar a manter a integridade e a credibilidade do processo de revisão sistemática, levando a resultados mais robustos e confiáveis. Para orientar a análise, foram abordadas as duas Questões de Pesquisa (QPs) apresentadas abaixo:

- QP1 - Quais técnicas de pré-processamento de dados foram aplicadas em modelos de Aprendizado de Máquina para detecção de *code smell*?
- QP2 - Quais técnicas de AM para detecção de *code smell* foram usadas com técnicas de pré-processamento de dados?

A.1 Estratégia de Busca

A RSL iniciou-se com uma estratégia de busca para extrair todos os artigos de pesquisa potencialmente relevantes de bibliotecas digitais acadêmicas. Depois, foram identificadas palavras-chave relevantes para o foco desta RSL e, então, foi formulada a sequência de pesquisa a partir de questões de pesquisa seguindo a abordagem de Kitchenham e Charters(114) PICO (*Population, Intervention, Comparison, and Outcomes*). Além disso, a RSL se centrou na detecção de *code smells* baseada em AM; assim, os sinônimos das técnicas de AM foram identificados. Em seguida, foi realizada a busca em todas as bibliotecas digitais especificadas, utilizando a *string* de busca abaixo:

Population: (code smell OR bad smell) AND; *Intervention:* (machine-learning OR machine learning OR structural-feature OR structural feature OR supervised) AND; *Comparation:* (deep-learning OR deep learning OR semantic-feature OR semantic feature OR un- supervised OR semisupervised OR semi-supervised OR semi supervised) AND; *Outcomes:* (recall OR f-measure OR f-score OR F1 OR precision OR mcc).

Estudos relevantes foram identificados utilizando a *string* de busca nas duas bibliotecas digitais acadêmicas mais utilizadas para engenharia de software: *IEEE Xplore* e *ACM Digital* (ZHANG et al.). Em seguida, a biblioteca *SCOPUS* foi incluída, pois

ela indexa diversas outras bases de dados acadêmicas menores. A busca foi realizada em novembro de 2023.

A.2 Seleção de Estudos

Para fornecer uma base para responder às questões de pesquisa, foram estabelecidos critérios específicos de inclusão e exclusão de estudos primários, delineando o conjunto de fontes consideradas para este estudo. Além disso, foram considerados diversos critérios de qualidade para que os estudos selecionados fornecessem dados qualificados para responder às questões de pesquisa formuladas. O processo de inclusão sistemática de referências foi integrado ao processo de seleção dos estudos para complementar os resultados da estratégia de busca. E por fim, foi apresentado um panorama dos estudos primários encontrados.

A.2.1 Critérios de inclusão/exclusão

Um estudo primário foi incluído somente se preenchesse todos os critérios descritos na Tabela 21.

Tabela 21 – Critérios de Inclusão

ID	Descrição do Critério de Inclusão
IC1	Escrito em inglês
IC2	Revisado por pares e publicado em Conferência ou Periódico
IC3	Resultados empíricos relatados
IC4	Técnicas de pré-processamento de dados, métodos de conjunto, aprendizado profundo ou transferência de aprendizado relatados

Foram excluídos os estudos que preencheram pelo menos um dos critérios detalhados na Tabela 22.

Tabela 22 – Critério de Exclusão

ID	Descrição do Critério de Exclusão
EC1	Não escrito em inglês
EC2	Publicação que não estava em fase final
EC3	Estudo de revisão (e.g., RSL, *survey*, etc.)
EC4	Estudo que se concentrou em outras técnicas de detecção, em vez de aprendizado de máquina
EC5	Incapaz de acessar
EC6	Procedimento ou índice
EC7	Estudo duplicado

A.2.2 Avaliação de Qualidade

De acordo com Alazba, Aljamaan e Alshayeb(81), avaliar a qualidade dos estudos selecionados auxilia na interpretação dos principais dados da pesquisa, ao mesmo tempo que avalia a qualidade das informações extraídas. Kitchenham et al.(115) observaram que não existem definições uniformes para as principais métricas de qualidade do estudo. Portanto, vários critérios de qualidade foram considerados, conforme detalhado na Tabela 23.

Tabela 23 – Avaliação de Qualidade

ID	Questões da Avaliação de Qualidade
QA1	Os objetivos e questões da pesquisa estão claramente definidos?
QA2	As medidas de desempenho utilizadas para avaliar os modelos do estudo são especificadas?
QA3	As variáveis independentes e variáveis dependentes estão claramente definidas?
QA4	O método de validação foi especificado?
QA5	Os conjuntos de dados estão adequadamente descritos?
QA6	Os detalhes do método experimental estão descritos adequadamente?
QA7	As ameaças à validade do estudo são especificadas?
QA8	As técnicas de pré-processamento de dados usadas são especificadas?
QA9	São usados métodos de comitê, aprendizado profundo ou transferência de aprendizado?

Os critérios de qualidade incluem a clareza das questões de pesquisa, as medidas de desempenho, variáveis independentes/dependentes, os métodos de validação, os conjuntos de dados, os detalhes de experimentos e as ameaças à validade, conforme usado por Alazba, Aljamaan e Alshayeb(81). Além disso, foram considerados o uso de pré-processamento de dados, métodos de comitê, aprendizado profundo e transferência de aprendizado. Foram atribuídas pontuações de acordo com o cumprimento desses critérios: “Sim” = 1, “Parcialmente” = 0,5 e ”Não” = 0.

Esta abordagem visa auxiliar na seleção de estudos primários que forneçam dados suficientemente qualificados para abordar as questões de pesquisa formuladas. Por isso, foi estabelecido um limite de 50% (ou seja, qualquer estudo que obtivesse pontuação inferior a 4,5 na avaliação da qualidade seria excluído), seguindo a abordagem utilizada por Alazba, Aljamaan e Alshayeb(81).

A.2.3 *Snowballing*

Com os estudos da seleção final derivados da *string* de busca, integrou-se o processo de seleção dos estudos adotando a inclusão sistemática de referências, ou seja, *snowballing*, conforme definido por Wohlin(116) para busca de possíveis artigos faltantes. No contexto desta RSL, foram aplicados *forward snowballing* aos artigos inicialmente selecionados, que incluiu todos os artigos que os referenciavam, e o *backward snowballing*, que incluiu todos os artigos que foram referenciados por eles.

A.3 Visão Geral dos Estudos Primários

A figura 11 mostra o número de artigos selecionados ao longo dos anos.

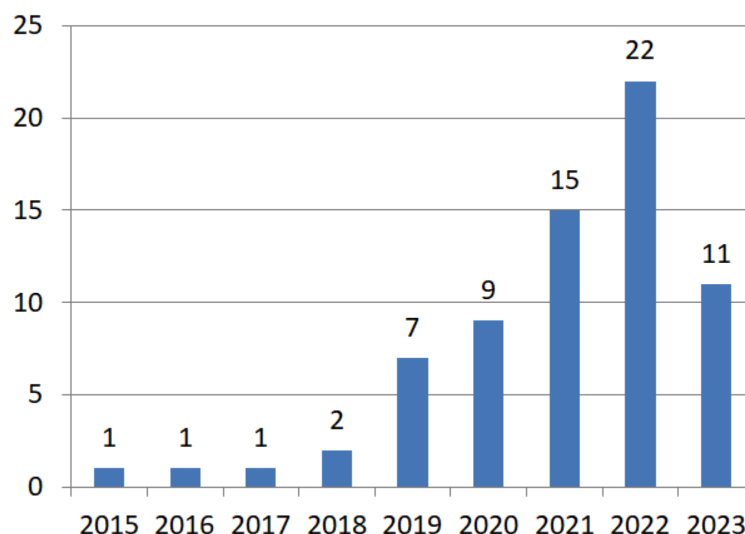


Figura 11 – Produção Científica Anual

Observa-se que esta área de pesquisa tem ganhado importância. Percebe-se que o número de estudos publicados entre 2015 e 2017 manteve-se estável, sendo um estudo publicado por ano, enquanto em 2018 foram publicados dois estudos. Porém, em 2019, houve um aumento de três vezes nas publicações, i.e., sete estudos publicados. Além disso, tem havido um aumento no número de estudos publicados nesta área nos últimos anos (9, 15 e 22, respectivamente, em 2020, 2021 e 2022), o que indica que a atenção da investigação nesta área está crescendo. E por fim, em 2023, já foram publicados 11 estudos.

Um total de 328 estudos potenciais foram identificados aplicando a *string* de busca detalhada na Subseção A.1. Além disso, foram incluídos filtros na biblioteca digital *SCOPUS*, para limitar os estudos àqueles escritos em inglês, artigos ou conferências, e aqueles publicados em periódico ou conferências.

A figura 12 ilustra o processo de seleção dos estudos primários empregado.

Partindo do conjunto de estudos potenciais, filtrou-se inicialmente os artigos que atendiam a todos os critérios de inclusão, resultando na seleção inicial de 313 estudos (15 foram excluídos por não terem sido submetidos à revisão por pares e por não terem sido publicados em congresso ou periódico). Posteriormente, foram eliminados desta seleção os estudos irrelevantes, aplicando-se os critérios de exclusão: i) 7 estudos não escritos em inglês (EC1); ii) 3 estudos que não estavam em fase final (EC2); iii) 38 estudos de revisão (EC3); iv) 185 estudos que centraram-se em outras técnicas de detecção em vez de aprendizado de máquina (EC4); v) 1 estudo sem acesso (EC5); vi) 33 anais ou índice (EC6); e vii) 9 estudos duplicados (EC7).

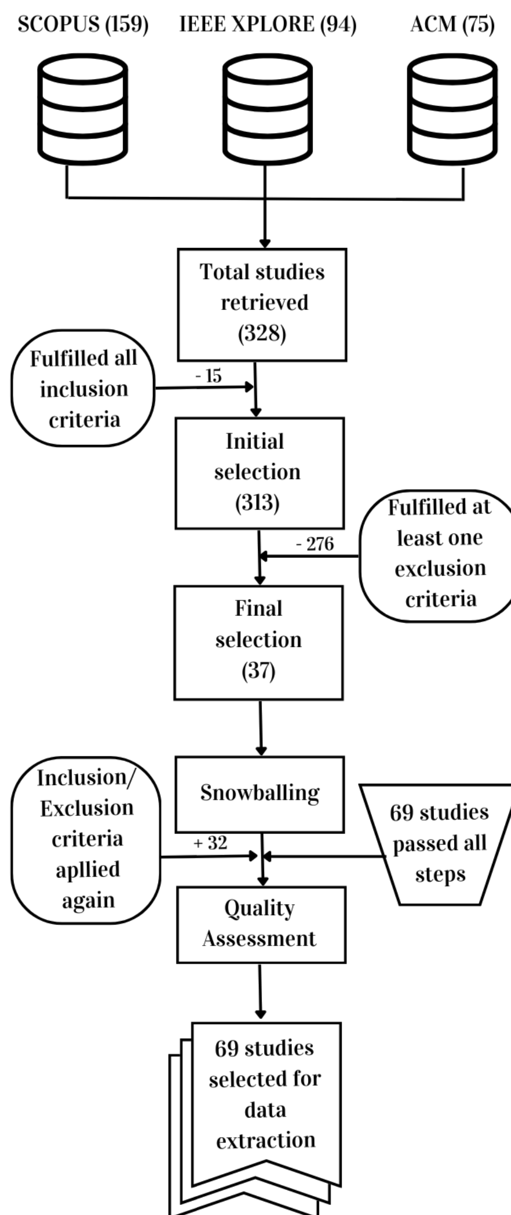


Figura 12 – Processo de seleção de estudos primários.

Isso resultou na exclusão de um total de 276 estudos, conforme detalhado na Tabela 24. Depois, 37 estudos foram selecionados por *snowballing* e submetidos aos mesmos critérios de exclusão/inclusão. Como resultado, foram selecionados 32 novos artigos (ver Tabela 25), totalizando 69 artigos a serem analisados na avaliação de qualidade. Nenhum artigo foi excluído após a avaliação.

A figura 13 ilustra a contribuição de cada fonte de pesquisa em cada etapa do processo de seleção dos estudos primários. O *IEEE Xplore* teve uma participação elevada e constante no percentual de estudos selecionados, variando de 28,66% dos estudos potenciais a 30,43% da seleção final. O *SCOPUS* começou com estudos de maior potencial do que outras fontes de pesquisa, mas teve uma queda acentuada de quase 30% na participação,

Tabela 24 – Detalhamento dos estudos excluídos

BIBLIOTECA DIGITAL	EC1	EC2	EC3	EC4	EC5	EC6	EC7	TOTAL
ACM	0	0	9	31	0	29	3	72
IEEE Xplore	4	0	15	50	0	4	0	73
SCOPUS	3	3	14	104	1	0	6	131
TOTAL	7	3	38	185	1	33	9	276

Tabela 25 – Fontes de dados e resultados de pesquisa

FONTE DE DADOS	ESTUDOS POTENCIAIS	SELEÇÃO INICIAL	SELEÇÃO FINAL
ACM	75	75	3
IEEE Xplore	94	94	21
Scopus	159	144	13
Subtotal	328	313	37
Backward Snowballing	-	-	7
Forward Snowballing	-	-	25
Total Geral	328	313	69

terminando com 18,84% dos estudos primários. *ACM Digital* foi a fonte de pesquisa com menor número de contribuições, chegando à seleção final com apenas 4,35% dos estudos primários selecionados. Vale destacar que 53,62% dos estudos primários foram selecionados através da *string* de busca, e os 46,38% restantes foram encontrados através do processo de *Snowballing*.

A.4 Análise de Dados

A.4.1 Extração de dados

Após finalizar a seleção de estudos primários para a RSL, extraiu-se os dados necessários desses estudos para responder às questões de pesquisa. Este processo de extração foi facilitado pela utilização do formulário de extração de dados detalhado na Tabela 26. Dentro deste formulário, alocou-se uma seção designada para documentar possíveis limitações observadas nos estudos, com o objetivo geral de desenvolver futuras diretrizes de pesquisa.

Além disso, coletou-se sistematicamente informações sobre atributos específicos em consideração. Esses atributos abrangeram uma gama de técnicas empregadas em pré-processamento de dados, métodos de comitê, aprendizado profundo e transferência de aprendizado, todos destinados a aprimorar modelos de aprendizado de máquina para detecção de *code smell*. Ao reunir metodicamente esta informação, as metodologias e estratégias predominantes na literatura foram analisadas de forma abrangente, enriquecendo

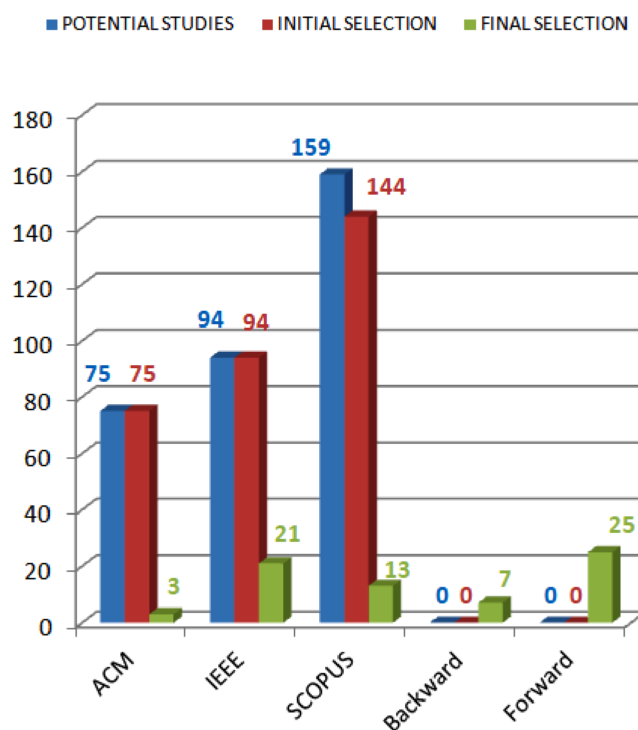


Figura 13 – Contribuição de cada fonte de pesquisa.

Tabela 26 – Formulário de Extração de Dados

DADOS EXTRAÍDOS	DESCRIÇÃO
Metadados	ID do Estudo, Título, Autores, Ano de Publicação, Tipo de Publicação Local de Publicação e Resumo.
Técnicas de Pré-processamento de dados aplicadas	Quais técnicas de pré-processamento de dados foram utilizadas nos estudos?
Melhoria dos modelos de AM	Quais métodos de comitê, aprendizado profundo ou transferência de aprendizado foram utilizados nos estudos?
Limitações ou Oportunidades	Quais são as limitações e oportunidades para melhoria das abordagens aplicadas?
Progresso alcançado	Quais avanços foram alcançados pelos estudos na detecção de *code smell*?
Problema a ser resolvido	Quais foram os detalhes contextuais do problema a ser resolvido?

assim a profundidade desta revisão.

A.4.2 Síntese de Dados

A síntese de dados envolveu a coleta e o resumo dos resultados dos estudos primários incluídos. A síntese pode ser descritiva (não quantitativa) e, apesar disso, é ocasionalmente viável aprimorar uma síntese descritiva com um resumo quantitativo (114).

Para sintetizar os dados extraídos relativos às questões de pesquisa, foi definido a estratégia de síntese narrativa para QP1 e QP2. Nesse sentido, buscou-se extrair informações sobre os estudos (e.g., técnicas de pré-processamento de dados utilizadas, contexto, limitação e qualidade do estudo). Tabulou-se, então, os dados em estilo compatível com as questões. Além disso, as tabelas geradas foram estruturadas para evidenciar os resultados obtidos pelo estudo.

APÊNDICE B – LOCALIZAÇÃO E FREQUÊNCIA DOS ATRIBUTOS SELECIONADOS

Tabela 27 – Localização dos atributos selecionados no vetor de atributos do conjunto de dados

Métrica	QQ-Original		QQ-Norm30		QQ-Norm25		QQ-Norm20		QQ-Anova30		QQ-Anova25	
	Java	C#	Java	C#	Java	C#	Java	C#	Java	C#	Java	C#
ATFD_method	10	8	10	8	10	8	10	8	10	8	10	8
FDP_method	11	9					11	9			11	9
MAXNESTING_method	13	10					13	10			13	10
LOC_method	14	11	14	11	14	11	14	11	14	11	14	11
CYCLO_method	15	12	15	12	15	12	15	12	15	12	15	12
NOLV_method	17	13					17	13			17	13
NOAV_method	19	14	19	14	19	14	19	14	19	14	19	14
FANOUT_method	21	15					21	15			21	15
CFNAMM_method	22	16					22	16			22	16
ATLD_method	23	17					23	17				
CINT_method	25	18	25	18			25	18	25	18	25	18
CDISP_method			27	19	27	19			27	19		
AMW_type	30	20	30	20	30	20			30	20		
ATFD_type	31	21	31	21			31	21	31	21	31	21
CFNAMM_type	33	22	33	22			33	22	33	22	33	22
DIT_type	34	23							34	23		
FANOUT_type	35	24					35	24			35	24
LCOM5_type			36	25	36	25			36	25		
LOCNAMM_type	37	26	37	26	37	26	37	26	37	26	37	26
LOC_package			38	27	38	27			38	27	38	27
LOC_type	40	28	40	28	40	28	40	28	40	28	40	28
NIM_type			41	29	41	29			41	29		
NOA_type	44	30					44	30			44	30
NOCS_package			45	31					45	31		
NOCS_project			46	32					46	32		
NOMNAMM_project			53	33	53	33			53	33		
NOMNAMM_type	54	34										
NOM_project			56	35	56	35			56	35		
RFC_type	60	36					60	36	60	36	60	36
TCC_type	61	37										
WMCNAMM_type	62	38	62	38	62	38			62	38		
WMC_type	63	39	63	39	63	39			63	39		
WOC_type	64	40	64	40	64	40			64	40		
AMWNAMM_type					29	42						
LOC_project									39	47		
NOAM_type							43	49	43	49	43	49
NOP_method							8	50				
MaMCL_method							18	51				
MeMCL_method							26	52				
CLNAMM_method							24	53				
NONFNSA							70	54			70	54
NOPVA							84	55			84	55

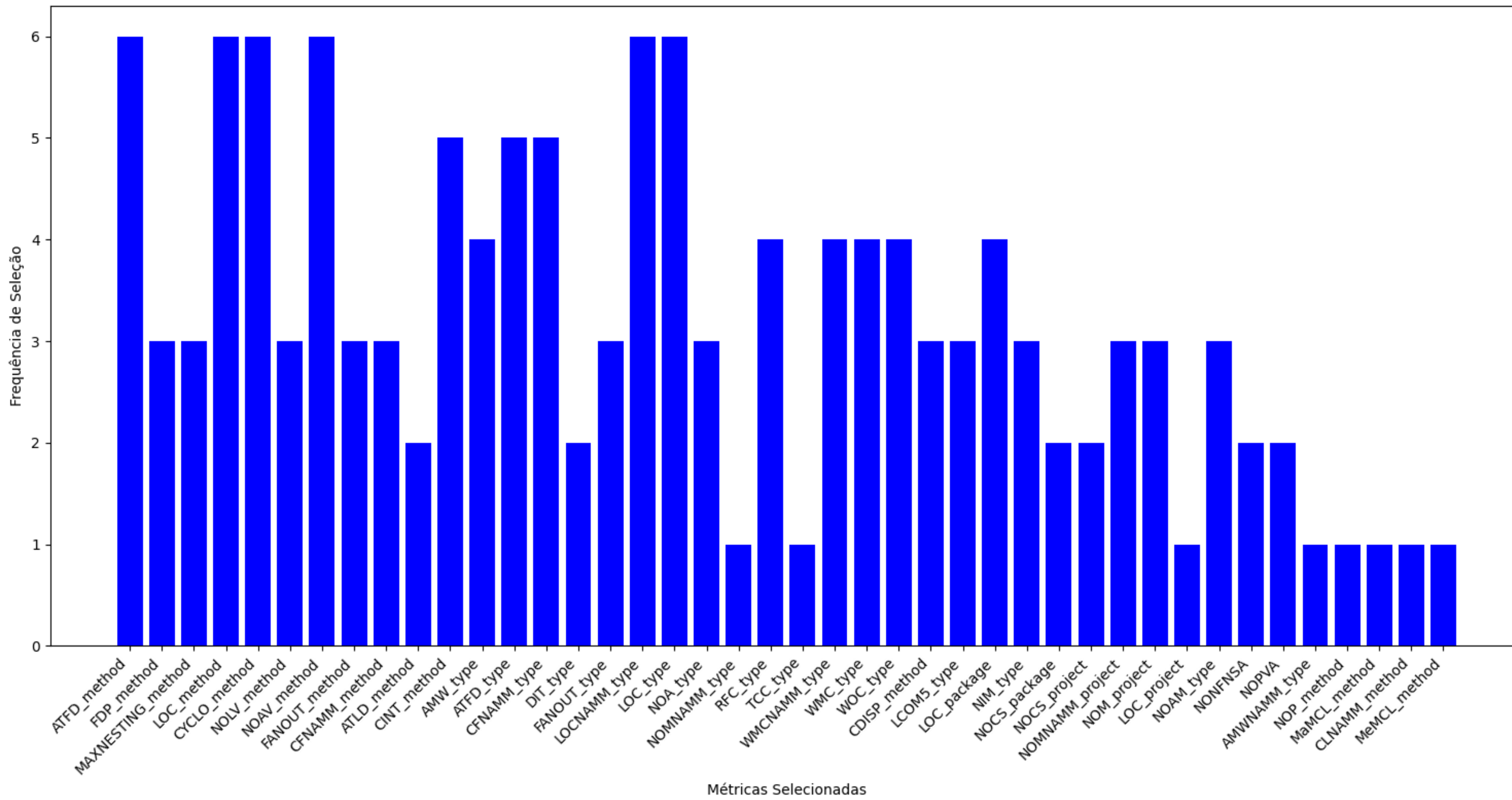


Figura 14 – Frequência de Seleção de Métricas para os Melhores Detectores de *Code Smell*

APÊNDICE C – HIPERPARÂMETROS SELECIONADOS PARA OS MÉTODOS DE COMITÊ UTILIZADOS NA DETECÇÃO DE *CODE SMELL*

Tabela 28 – Hiperparâmetros selecionados para os detectores de *code smell RF*

MC_SelAtr_OH	Hiperparâmetros Selecionados
RF_QQ-Original_RS	‘n_estimators’: 524, ‘min_samples_split’: 12, ‘min_samples_leaf’: 4, ‘max_features’: ‘sqrt’, ‘max_depth’: 13, ‘criterion’: ‘entropy’, ‘bootstrap’: False
RF_QQ-Norm-25%-binário_RS	‘n_estimators’: 524, ‘min_samples_split’: 12, ‘min_samples_leaf’: 4, ‘max_features’: ‘sqrt’, ‘max_depth’: 13, ‘criterion’: ‘entropy’, ‘bootstrap’: False
RF_QQ-Norm-20%-multiclasse_BS	OrderedDict([(‘bootstrap’, False), (‘criterion’, ‘entropy’), (‘max_depth’, 15), (‘max_features’, ‘log2’), (‘min_samples_leaf’, 1), (‘min_samples_split’, 20), (‘n_estimators’, 1000)])

Tabela 29 – Hiperparâmetros selecionados para os detectores de *code smell XGB*

MC_SelAtr_OH	Hiperparâmetros Selecionados
XGB_QQ-Norm-25%-binário_RS	‘tree_method’: ‘approx’, ‘n_estimators’: 58, ‘max_depth’: 5, ’learning_rate’: 0.24655172413793103, ‘grow_policy’: ‘depthwise’, ‘booster’: ‘gbtree’
XGB_QQ-Norm-30%-binário_BS	OrderedDict([(‘booster’, ‘gbtree’), (‘grow_policy’, ‘lossguide’), (‘learning_rate’, 0.061380987690824756), (‘max_depth’, 9), (‘n_estimators’, 229), (‘tree_method’, ‘approx’)])
XGB_Anova-30%-multiclasse_BS	OrderedDict([(‘booster’, ‘dart’), (‘grow_policy’, ‘depthwise’), (‘learning_rate’, 0.01), (‘max_depth’, 20), (‘n_estimators’, 503), (‘tree_method’, ’approx’)])

Tabela 30 – Hiperparâmetros selecionados para os detectores de *code smell CB*

MC_SelAtr_OH	Hiperparâmetros Selecionados
CB_QQ-Original_RS	‘learning_rate’: 0.01, ‘leaf_estimation_method’: ‘Newton’, ‘l2_leaf_reg’: 6, ‘iterations’: 454, ‘grow_policy’: ‘Lossguide’, ‘feature_border_type’: ‘UniformAndQuantiles’, ‘depth’: 5, ‘border_count’: 192, ‘bootstrap_type’: ‘MVS’, ‘auto_class_weights’: ‘Balanced’
CB_Anova-30%-multiclasse_BS	OrderedDict([(‘auto_class_weights’, ‘Balanced’), (‘bootstrap_type’, ‘MVS’), (‘border_count’, 251), (‘depth’, 10), (‘feature_border_type’, ‘GreedyLogSum’), (‘grow_policy’, ‘SymmetricTree’), (‘iterations’, 915), (‘l2_leaf_reg’, 7), (‘leaf_estimation_method’, ‘Newton’), (‘learning_rate’, 0.5)])
CB_QQ-Norm-20%-multiclasse_BS	OrderedDict([(‘auto_class_weights’, ‘SqrtBalanced’), (‘bootstrap_type’, ‘Bayesian’), (‘border_count’, 32), (‘depth’, 16), (‘feature_border_type’, ‘MaxLogSum’), (‘grow_policy’, ‘Lossguide’), (‘iterations’, 1000), (‘l2_leaf_reg’, 10), (‘leaf_estimation_method’, ‘Newton’), (‘learning_rate’, 0.01)])
CB_Anova-25%-binário_BS	OrderedDict([(‘auto_class_weights’, ‘SqrtBalanced’), (‘bootstrap_type’, ’No’), (‘border_count’, 112), (‘depth’, 14), (‘feature_border_type’, ‘GreedyLogSum’), (‘grow_policy’, ‘Lossguide’), (‘iterations’, 666), (‘l2_leaf_reg’, 5), (‘leaf_estimation_method’, ‘Gradient’), (‘learning_rate’, 0.3208067898602168)])

APÊNDICE D – HIPERPARÂMETROS SELECIONADOS PARA OS MÉTODOS DE COMITÊ UTILIZADOS NA AVALIAÇÃO DE GRAVIDADE

Tabela 31 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *QQ-Norm-25%-binário_RS*

MC	Sobr.	Hiperparâmetros Selecionados
RF	Sem Sobr.	‘n_estimators’: 970, ‘min_samples_split’: 7, ‘min_samples_leaf’: 3, ‘max_features’: None, ‘max_depth’: 6, ‘criterion’: ‘log_loss’, ‘bootstrap’: True
	B-SMOTE	‘n_estimators’: 524, ‘min_samples_split’: 12, ‘min_samples_leaf’: 4, ‘max_features’: ‘sqrt’, ‘max_depth’: 13, ‘criterion’: ‘entropy’, ‘bootstrap’: False
	SMOTE	‘n_estimators’: 118, ‘min_samples_split’: 12, ‘min_samples_leaf’: 3, ‘max_features’: ‘sqrt’, ‘max_depth’: 9, ‘criterion’: ‘gini’, ‘bootstrap’: False
XGB	Sem Sobr.	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softprob’, ‘n_estimators’: 356, ‘max_depth’: 9, ‘learning_rate’: 0.29724137931034483, ‘grow_policy’: ‘depthwise’, ‘booster’: ‘dart’
	B-SMOTE	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softprob’, ‘n_estimators’: 998, ‘max_depth’: 6, ‘learning_rate’: 0.31413793103448273, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘dart’
	SMOTE	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softmax’, ‘n_estimators’: 406, ‘max_depth’: 3, ‘learning_rate’: 0.3310344827586207, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘gbtree’
CB	Sem Sobr.	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.43241379310344824, ‘leaf_estimation_method’: ‘Newton’, ‘l2_leaf_reg’: 5, ‘iterations’: 166, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘MaxLogSum’, ‘depth’: 14, ‘border_count’: 48, ‘bootstrap_type’: ‘MVS’, ‘auto_class_weights’: ‘SqrtBalanced’
	B-SMOTE	‘task_type’: t‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.07758620689655171, ‘leaf_estimation_method’: ‘Gradient’, ‘l2_leaf_reg’: 8, ‘iterations’: 548, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘GreedyLogSum’, ‘depth’: 6, ‘border_count’: 208, ‘bootstrap_type’: ‘Bayesian’, ‘auto_class_weights’: ‘Balanced’
	SMOTE	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.21275862068965518, ‘leaf_estimation_method’: ‘Gradient’, ‘l2_leaf_reg’: 5, ‘iterations’: 502, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘MinEntropy’, ‘depth’: 12, ‘border_count’: 192, ‘bootstrap_type’: ‘Bernoulli’, ‘auto_class_weights’: ‘Balanced’

Foram utilizados 25% dos atributos selecionados por meio da técnica Qui-quadrado, com preditores normalizados e alvos binários. Os hiperparâmetros dos modelos foram otimizados com a busca randomizada.

Tabela 32 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *QQ-Original_RS*

MC	Sobr.	Hiperparâmetros Selecionados
RF	Sem Sobr.	‘n_estimators’: 970, ‘min_samples_split’: 7, ‘min_samples_leaf’: 3, ‘max_features’: None, ‘max_depth’: 6, ‘criterion’: ‘log_loss’, ‘bootstrap’: True
	B-SMOTE	‘n_estimators’: 524, ‘min_samples_split’: 12, ‘min_samples_leaf’: 4, ‘max_features’: ‘sqrt’, ‘max_depth’: 13, ‘criterion’: ‘entropy’, ‘bootstrap’: False
	SMOTE	‘n_estimators’: 118, ‘min_samples_split’: 12, ‘min_samples_leaf’: 3, ‘max_features’: ‘sqrt’, ‘max_depth’: 9, ‘criterion’: ‘gini’, ‘bootstrap’: False
XGB	Sem Sobr.	‘num_class’: 12, ‘tree_method’: ‘approx’, ‘objective’: ‘multi:softmax’, ‘n_estimators’: 420, ‘max_depth’: 5, ‘learning_rate’: 0.5, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘gbtree’
	B-SMOTE	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softmax’, ‘n_estimators’: 406, ‘max_depth’: 3, ‘learning_rate’: 0.3310344827586207, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘gbtree’
	SMOTE	‘num_class’: 12, ‘tree_method’: ‘approx’, ‘objective’: ‘multi:softprob’, ‘n_estimators’: 586, ‘max_depth’: 5, ‘learning_rate’: 0.5, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘dart’
CB	Sem Sobr.	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.16206896551724137, ‘leaf_estimation_method’: ‘Newton’, ‘l2_leaf_reg’: 7, ‘iterations’: 592, ‘grow_policy’: ‘Lossguide’, ‘feature_border_type’: ‘Uniform’, ‘depth’: 6, ‘border_count’: 192, ‘bootstrap_type’: ‘MVS’, ‘auto_class_weights’: ‘Balanced’
	B-SMOTE	‘task_type’: t’CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.3310344827586207, ‘leaf_estimation_method’: ‘Newton’, ‘l2_leaf_reg’: 6, ‘iterations’: 734, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘GreedyLogSum’, ‘depth’: 15, ‘border_count’: 96, ‘bootstrap_type’: ‘Bayesian’, ‘auto_class_weights’: ‘SqrtBalanced’
	SMOTE	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.07758620689655171, ‘leaf_estimation_method’: ‘Gradient’, ‘l2_leaf_reg”: 8, ‘iterations’: 548, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘GreedyLogSum’, ‘depth’: 6, ‘border_count’: 208, ‘bootstrap_type’:‘ Bayesian’, ‘auto_class_weights’: ‘Balanced’

Foram utilizados 30% dos atributos selecionados por meio da técnica Qui-quadrado, com preditores normalizados e alvos binários (68). Os hiperparâmetros dos modelos foram otimizados com a busca randomizada.

Tabela 33 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *QQ-Norm-30%-binário_BS*

MC	Sobr.	Hiperparâmetros Selecionados
RF	Sem Sobr.	‘n_estimators’: 970, ‘min_samples_split’: 7, ‘min_samples_leaf’: 3, ‘max_features’: None, ‘max_depth’: 6, ‘criterion’: ‘log_loss’, ‘bootstrap’: True
	B-SMOTE	‘n_estimators’: 118, ‘min_samples_split’: 12, ‘min_samples_leaf’: 3, ‘max_features’: ‘sqrt’, ‘max_depth’: 9, ‘criterion’: ‘gini’, ‘bootstrap’: False
	SMOTE	‘n_estimators’: 118, ‘min_samples_split’: 12, ‘min_samples_leaf’: 3, ‘max_features’: ‘sqrt’, ‘max_depth’: 9, ‘criterion’: ‘gini’, ‘bootstrap’: False
XGB	Sem Sobr.	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softmax’, ‘n_estimators’: 140, ‘max_depth’: 2, ‘learning_rate’: 0.2803448275862069, ‘grow_policy’: ‘depthwise’, ‘booster’: ‘dart’
	B-SMOTE	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softmax’, ‘n_estimators’: 58, ‘max_depth’: 8, ‘learning_rate’: 0.38172413793103444, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘dart’
	SMOTE	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softmax’, ‘n_estimators’: 720, ‘max_depth’: 18, ‘learning_rate’: 0.36482758620689654, ‘grow_policy’: ‘depthwise’, ‘booster’: ‘gbtree’
CB	Sem Sobr.	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.16206896551724137, ‘leaf_estimation_method’: ‘Newton’, ‘l2_leaf_reg’: 7, ‘iterations’: 592, ‘grow_policy’: ‘Lossguide’, ‘feature_border_type’: ‘Uniform’, ‘depth’: 6, ‘border_count’: 192, ‘bootstrap_type’: ‘MVS’, ‘auto_class_weights’: ‘Balanced’
	B-SMOTE	‘task_type’: t‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.026896551724137928, ‘leaf_estimation_method’: ‘Gradient’, ‘l2_leaf_reg’: 7, ‘iterations’: 140, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘UniformAndQuantiles’, ‘depth’: 13, ‘border_count’: 176, ‘bootstrap_type’: ‘MVS’, ‘auto_class_weights’: ‘Balanced’
	SMOTE	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.16206896551724137, ‘leaf_estimation_method’: ‘Gradient’, ‘l2_leaf_reg’: 2, ‘iterations’: 182, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘MaxLogSum’, ‘depth’: 11, ‘border_count’: 176, ‘bootstrap_type’: ‘Bernoulli’, ‘auto_class_weights’: ‘Balanced’

Foram utilizados 30% dos atributos selecionados por meio da técnica Qui-quadrado, com preditores normalizados e alvos binários. Os hiperparâmetros dos modelos foram otimizados com a busca Bayesiana.

Tabela 34 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *Anova-30%-multiclasse_BS*

MC	Sobr.	Hiperparâmetros Selecionados
RF	Sem Sobr.	‘n_estimators’: 970, ‘min_samples_split’: 7, ‘min_samples_leaf’: 3, ‘max_features’: None, ‘max_depth’: 6, ‘criterion’: ‘log_loss’, ‘bootstrap’: True
	B-SMOTE	‘n_estimators’: 118, ‘min_samples_split’: 12, ‘min_samples_leaf’: 3, ‘max_features’: ‘sqrt’, ‘max_depth’: 9, ‘criterion’: ‘gini’, ‘bootstrap’: False
	SMOTE	‘n_estimators’: 416, ‘min_samples_split’: 8, ‘min_samples_leaf’: 1, ‘max_features’: ‘log2’, ‘max_depth’: 6, ‘criterion’: ‘log_loss’, ‘bootstrap’: False
XGB	Sem Sobr.	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softmax’, ‘n_estimators’: 786, ‘max_depth’: 5, ‘learning_rate’: 0.11137931034482758, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘dart’
	B-SMOTE	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softprob’, ‘n_estimators’: 164, ‘max_depth’: 1, ‘learning_rate’: 0.48310344827586205, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘dart’
	SMOTE	‘num_class’: 12, ‘tree_method’: ‘approx’, ‘objective’: ‘multi:softmax’, ‘n_estimators’: 852, ‘max_depth’: 4, ‘learning_rate’: 0.19586206896551722, ‘grow_policy’: ‘depthwise’, ‘booster’: ‘dart’
CB	Sem Sobr.	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.04379310344827586, ‘leaf_estimation_method’: ‘Newton’, ‘l2_leaf_reg’: 8, ‘iterations’: 354, ‘grow_policy’: ‘Lossguide’, ‘feature_border_type’: ‘MaxLogSum’, ‘depth’: 6, ‘border_count’: 96, ‘bootstrap_type’: ‘No’, ‘auto_class_weights’: ‘SqrtBalanced’
	B-SMOTE	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.3986206896551724, ‘leaf_estimation_method’: ‘Newton’, ‘l2_leaf_reg’: 7, ‘iterations’: 396, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘UniformAndQuantiles’, ‘depth’: 15, ‘border_count’: 128, ‘bootstrap_type’: ‘Bernoulli’, ‘auto_class_weights’: ‘SqrtBalanced’
	SMOTE	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.3479310344827586, ‘leaf_estimation_method’: ‘Gradient’, ‘l2_leaf_reg’: 5, ‘iterations’: 910, ‘grow_policy’: ‘Lossguide’, ‘feature_border_type’: ‘Uniform’, ‘depth’: 12, ‘border_count’: 192, ‘bootstrap_type’: ‘Bayesian’, ‘auto_class_weights’: ‘Balanced’

Foram utilizados 30% dos atributos selecionados por meio da técnica ANOVA, com preditores sem escalonamento de dados e alvos multiclasse. Os hiperparâmetros dos modelos foram otimizados com a busca Bayesiana.

Tabela 35 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *QQ-Norm-20%-multiclasse_BS*

MC	Sobr.	Hiperparâmetros Selecionados
RF	Sem Sobr.	‘n_estimators’: 970, ‘min_samples_split’: 7, ‘min_samples_leaf’: 3, ‘max_features’: None, ‘max_depth’: 6, ‘criterion’: ‘log_loss’, ‘bootstrap’: True
	B-SMOTE	‘n_estimators’: 524, ‘min_samples_split’: 12, ‘min_samples_leaf’: 4, ‘max_features’: ‘sqrt’, ‘max_depth’: 13, ‘criterion’: ‘entropy’, ‘bootstrap’: False
	SMOTE	‘n_estimators’: 524, ‘min_samples_split’: 12, ‘min_samples_leaf’: 4, ‘max_features’: ‘sqrt’, ‘max_depth’: 13, ‘criterion’: ‘entropy’, ‘bootstrap’: False
XGB	Sem Sobr.	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softprob’, ‘n_estimators’: 94, ‘max_depth’: 17, ‘learning_rate’: 0.1789655172413793, ‘grow_policy’: ‘depthwise’, ‘booster’: ‘dart’
	B-SMOTE	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softprob’, ‘n_estimators’: 406, ‘max_depth’: 17, ‘learning_rate’: 0.060689655172413794, ‘grow_policy’: ‘depthwise’, ‘booster’: ‘dart’
	SMOTE	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softmax’, ‘n_estimators’: 920, ‘max_depth’: 17, “learning_rate’: 0.2803448275862069, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘gbtree’
CB	Sem Sobr.	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.19586206896551722, ‘leaf_estimation_method’: ‘Newton’, ‘l2_leaf_reg’: 3, ‘iterations’: 918, ‘grow_policy’: ‘Lossguide’, ‘feature_border_type’: ‘GreedyLogSum’, ‘depth’: 6, ‘border_count’: 32, ‘bootstrap_type’: ‘No’, ‘auto_class_weights’: ‘Balanced’
	B-SMOTE	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.21275862068965518, ‘leaf_estimation_method’: ‘Gradient’, ‘l2_leaf_reg’: 5, ‘iterations’: 502, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘MinEntropy’, ‘depth’: 12, ‘border_count’: 192, ‘bootstrap_type’: ‘Bernoulli’, ‘auto_class_weights’: ‘Balanced’
	SMOTE	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.43241379310344824, ‘leaf_estimation_method’: ‘Newton’, ‘l2_leaf_reg’: 5, ‘iterations’: 166, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘MaxLogSum’, ‘depth’: 14, ‘border_count’: 48, ‘bootstrap_type’: ‘MVS’, ‘auto_class_weights’: ‘SqrtBalanced’

Foram utilizados 20% dos atributos selecionados por meio da técnica Qui-quadrado, com preditores normalizados e alvos multiclasse. Os hiperparâmetros dos modelos foram otimizados com a busca Bayesiana.

Tabela 36 – Hiperparâmetros selecionados para a Avaliação de Gravidade - *Anova-25%-binário_BS*

MC	Sobr.	Hiperparâmetros Selecionados
RF	Sem Sobr.	‘n_estimators’: 970, ‘min_samples_split’: 7, ‘min_samples_leaf’: 3, ‘max_features’: None, ‘max_depth’: 6, ‘criterion’: ‘log_loss’, ‘bootstrap’: True
	B-SMOTE	‘n_estimators’: 416, ‘min_samples_split’: 8, ‘min_samples_leaf’: 1, ‘max_features’: ‘log2’, ‘max_depth’: 6, ‘criterion’: ‘log_loss’, ‘bootstrap’: False
	SMOTE	‘n_estimators’: 118, ‘min_samples_split’: 12, ‘min_samples_leaf’: 3, ‘max_features’: ‘sqrt’, ‘max_depth’: 9, ‘criterion’: ‘gini’, ‘bootstrap’: False
XGB	Sem Sobr.	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softprob’, ‘n_estimators’: 94, ‘max_depth’: 18, ‘learning_rate’: 0.026896551724137928, ‘grow_policy’: ‘depthwise’, ‘booster’: ‘dart’
	B-SMOTE	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softprob’, ‘n_estimators’: 310, ‘max_depth’: 3, ‘learning_rate’: 0.07758620689655171, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘dart’
	SMOTE	‘num_class’: 12, ‘tree_method’: ‘hist’, ‘objective’: ‘multi:softmax’, ‘n_estimators’: 998, ‘max_depth’: 2, ‘learning_rate’: 0.43241379310344824, ‘grow_policy’: ‘lossguide’, ‘booster’: ‘dart’
CB	Sem Sobr.	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.09448275862068964, ‘leaf_estimation_method’: ‘Newton’, ‘l2_leaf_reg’: 2, ‘iterations’: 378, ‘grow_policy’: ‘Lossguide’, ‘feature_border_type’: ‘Uniform’, ‘depth’: 15, ‘border_count’: 48, ‘bootstrap_type’: ‘No’, ‘auto_class_weights’: ‘SqrtBalanced’
	B-SMOTE	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.09448275862068964, ‘leaf_estimation_method’: ‘Gradient’, ‘l2_leaf_reg’: 9, ‘iterations’: 566, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘Uniform’, ‘depth’: 6, ‘border_count’: 128, ‘bootstrap_type’: ‘No’, ‘auto_class_weights’: ‘Balanced’
	SMOTE	‘task_type’: ‘CPU’, ‘verbose’: False, ‘classes_count’: 12, ‘objective’: ‘MultiClass’, ‘learning_rate’: 0.3986206896551724, ‘leaf_estimation_method’: ‘Gradient’, ‘l2_leaf_reg’: 6, ‘iterations’: 288, ‘grow_policy’: ‘Depthwise’, ‘feature_border_type’: ‘MaxLogSum’, ‘depth’: 10, ‘border_count’: 240, ‘bootstrap_type’: ‘Bernoulli’, ‘auto_class_weights’: ‘SqrtBalanced’

Foram utilizados 25% dos atributos selecionados por meio da técnica ANOVA, sem escalonamento de dados e alvos binários. Os hiperparâmetros dos modelos foram otimizados com a busca Bayesiana.

APÊNDICE E – EVOLUÇÃO DAS MEDIDAS DE DESEMPENHO E DO TEMPO DE OTIMIZAÇÃO DE HIPERPARÂMETROS PARA A DETECÇÃO DE *CODE SMELL* COM ANOVA

As figuras 15, 16, 17, 18, 19, 20, 21 e 22 mostram a evolução da acurácia, da média de desempenho, do desvio padrão do desempenho e do tempo de otmização de hiperarâmetros, com busca randomizada (RS) ou bayesiana (BS), para os modelos de AM utilizados na fase de detecção de *code smell* do modelo proposto, com a utilização da técnica ANOVA para Seleção de Atributos. Esses atributos constam do Conjunto de Dados Java e foram selecionados na proporção de 10, 15, 20, 25, 30 e 100% dos atributos possíveis sem escalonamento de dados, normalizados ou padronizados. Para melhor entendimento dos rótulos dos gráficos é importante mencionar que:

i) preditores_todos_atributos: utilização de todos os atributos possíveis do conjunto de dados;

ii) preditores_anova_10, 15, 20, 25 ou 30: utilização de uma porcentagem de 10 a 30% dos atributos originais;

iii) preditores_anova_norm_10, 15, 20, 25 ou 30: utilização de uma porcentagem de 10 a 30% dos atributos originais normalizados;

iv) AD_BS: Árvore de Decisão com Busca Bayesiana;

v) AD_RS: Árvore de Decisão com Busca Randomizada;

vi) RF_BS: *Random Forest* com Busca Bayesiana;

vii) RF_RS: *Random Forest* com Busca Randomizada;

viii) XGB_BS: *Extreme Gradient Boosting* com Busca Bayesiana;

ix) XGB_RS: *Extreme Gradient Boosting* com Busca Randomizada;

x) CB_BS: *Categorical Boosting* com Busca Bayesiana; e

xi) CB_RS: *Categorical Boosting* com Busca Randomizada;

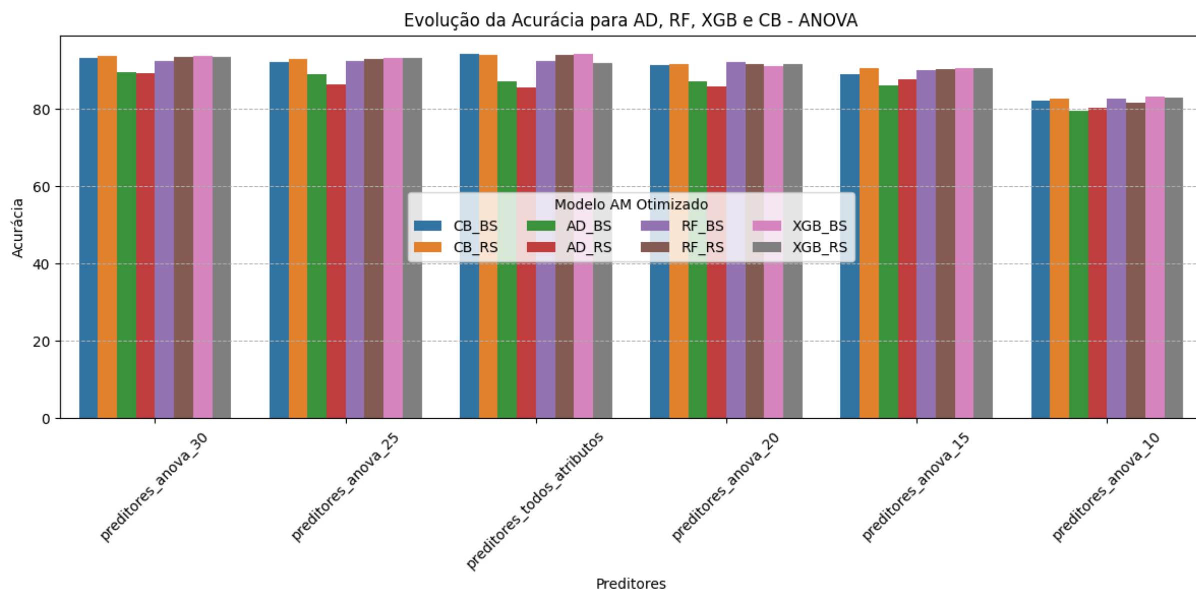


Figura 15 – Evolução da Acurácia - ANOVA

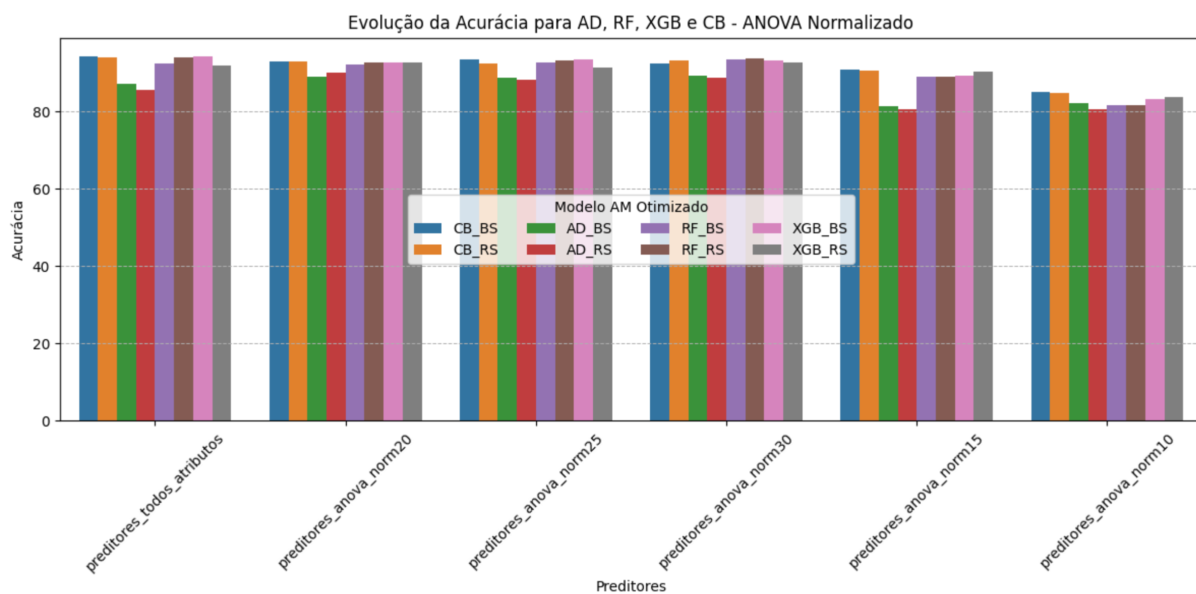


Figura 16 – Evolução da Acurácia - ANOVA com Atributos Normalizados

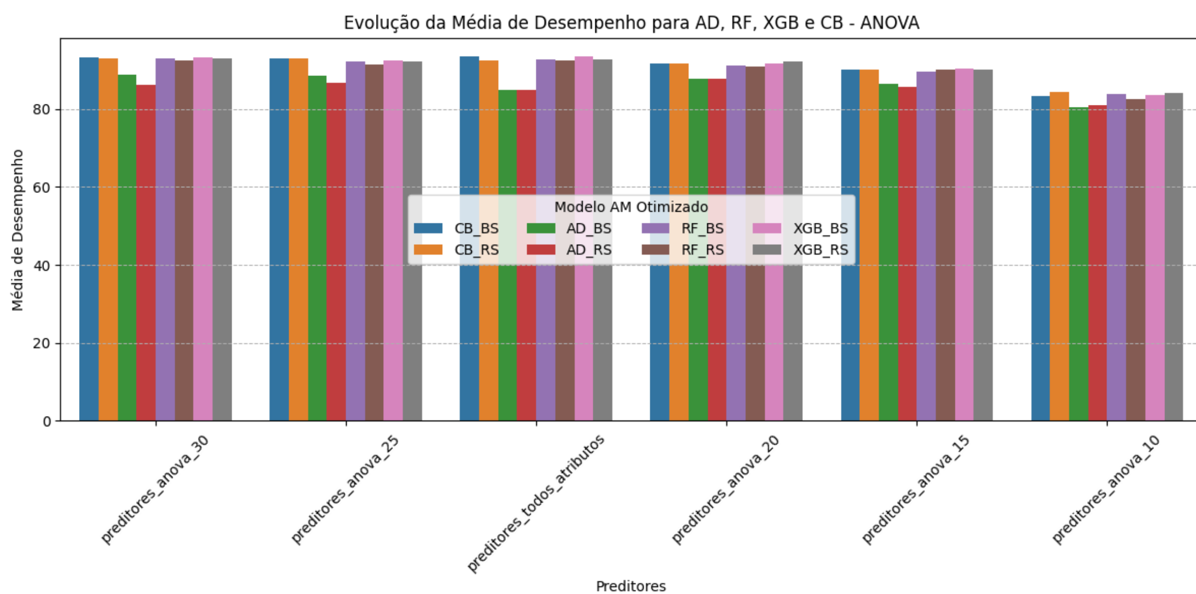


Figura 17 – Evolução da Média de Desempenho - ANOVA

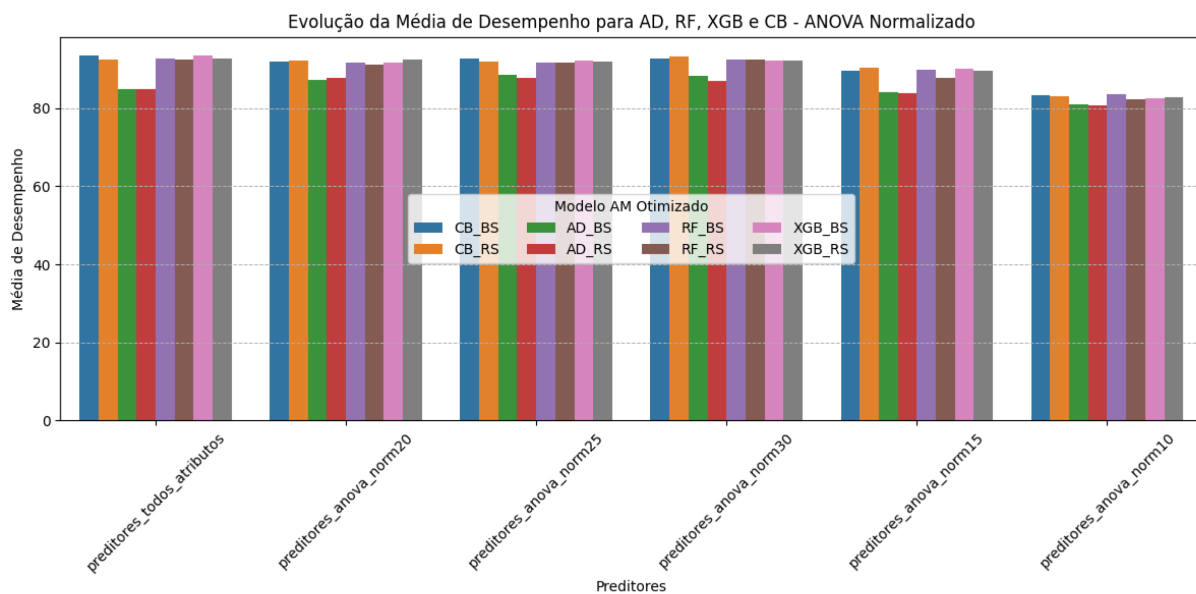


Figura 18 – Evolução da Média de Desempenho - ANOVA com Atributos Normalizados

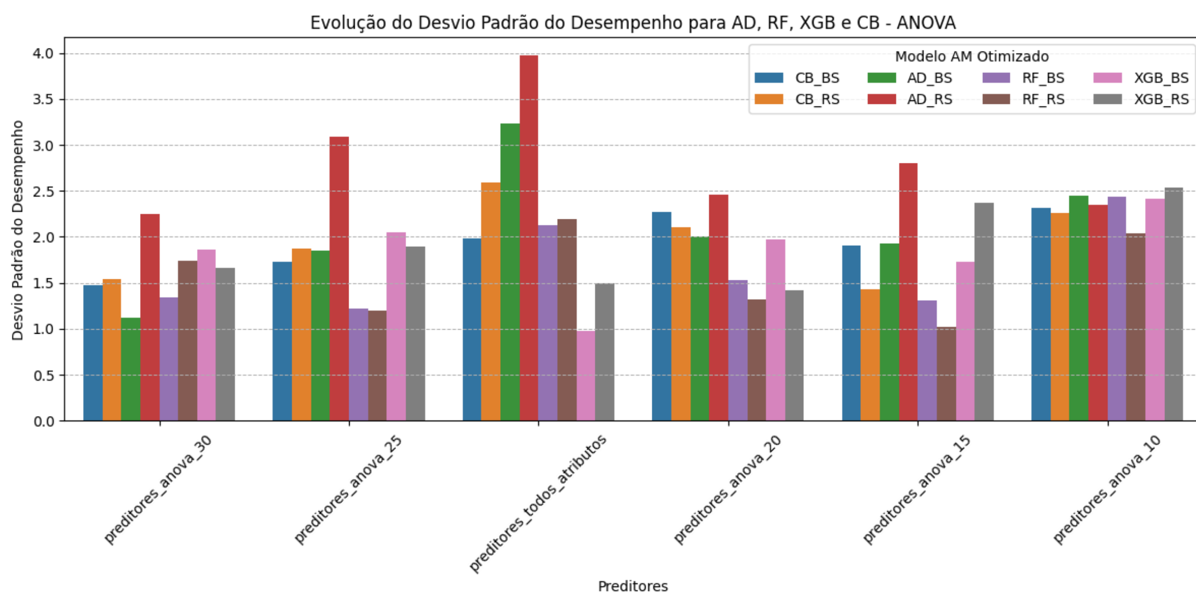


Figura 19 – Evolução da Desvio Padrão do Desempenho - ANOVA

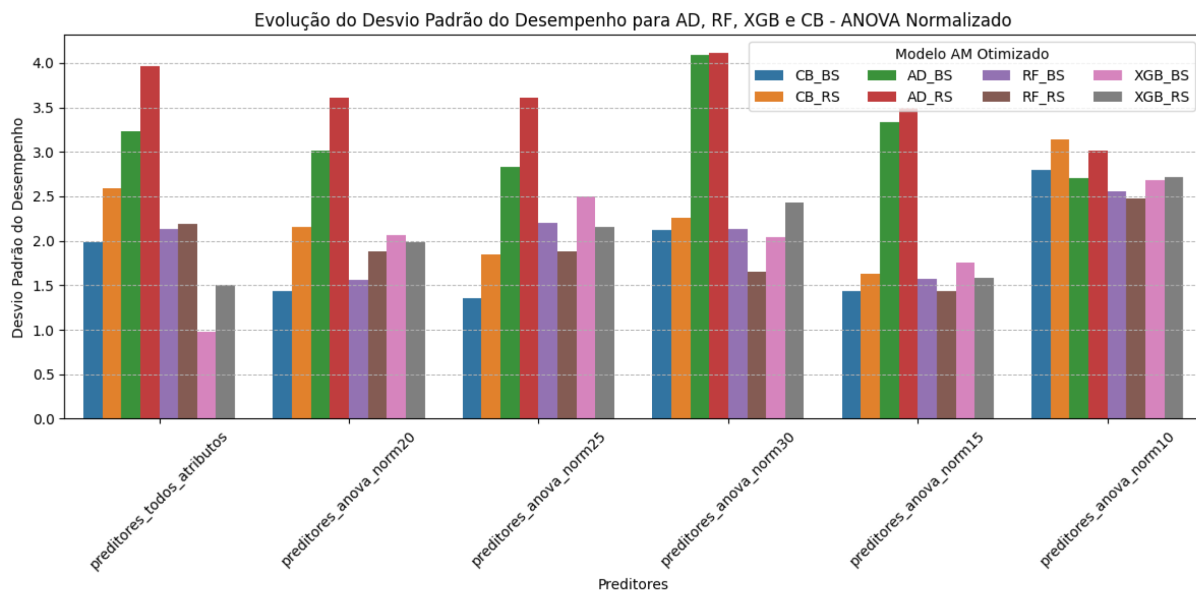


Figura 20 – Evolução da Desvio Padrão do Desempenho - ANOVA com Atributos Normalizados

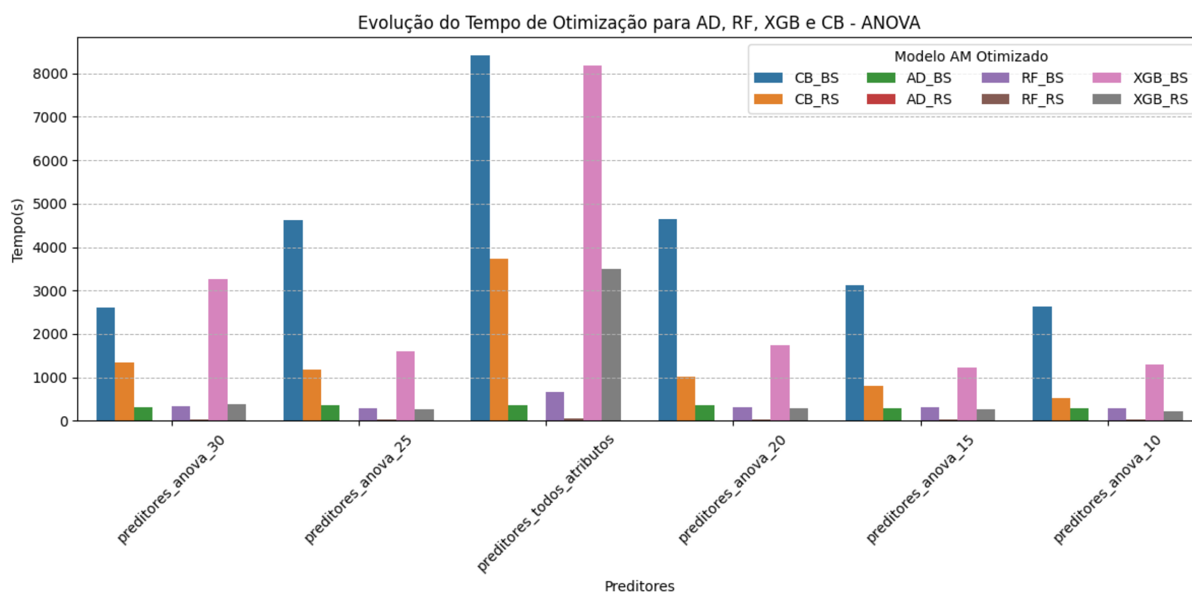


Figura 21 – Evolução do Tempo de Otimização de Hiperparâmetros - ANOVA

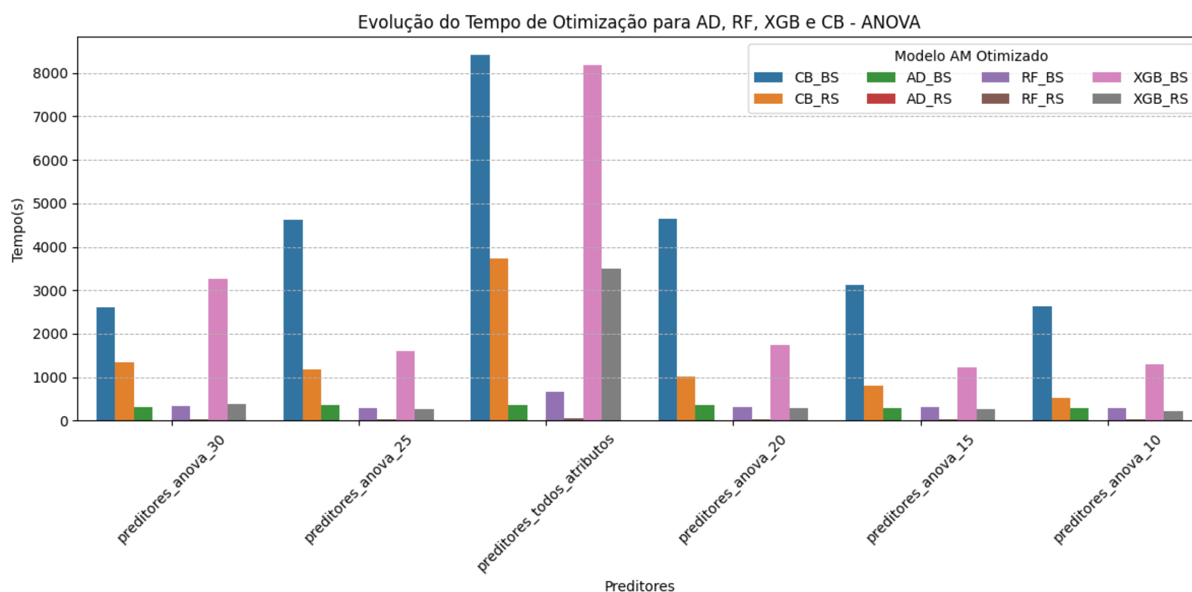


Figura 22 – Evolução do Tempo de Otimização de Hiperparâmetros - ANOVA com Atributos Normalizados

APÊNDICE F – EVOLUÇÃO DAS MEDIDAS DE DESEMPENHO E DO TEMPO DE OTIMIZAÇÃO DE HIPERPARÂMETROS PARA A DETECÇÃO DE *CODE SMELL* COM QUI-QUADRADO

As figuras 23, 24, 25, 26, Figura 27, Figura 28, Figura 29 e Figura 30 mostram a evolução da acurácia, da média de desempenho e do desvio padrão do desempenho para os modelos de AM utilizados para detecção de *code smell*, com a utilização da técnica Qui-Quadrado para Seleção de Atributos. Esses atributos constam do Conjunto de Dados Java e foram selecionados na proporção de 10, 15, 20, 25, 30 e 100% dos atributos possíveis sem ou com a normalização dos dados. Para melhor entendimento dos rótulos dos gráficos é importante mencionar que:

i) preditores_todos_atributos: utilização de todos os atributos possíveis do conjunto de dados;

ii) preditores_qq_10, 15, 20, 25 ou 30: utilização de uma porcentagem de 10 a 30% dos atributos originais selecionados com a técnica qui-quadrado;

iii) preditores_qq_norm_10, 15, 20, 25 ou 30: utilização de uma porcentagem de 10 a 30% dos atributos originais normalizados selecionados com a técnica qui-quadrado;

iv) preditores_qq_original: utilização dos atributos selecionados por Santos, Duarte e Choren(68);

v) AD_BS: Árvore de Decisão com Busca Bayesiana;

vi) AD_RS: Árvore de Decisão com Busca Randomizada;

vii) RF_BS: *Random Forest* com Busca Bayesiana;

viii) RF_RS: *Random Forest* com Busca Randomizada;

ix) XGB_BS: *Extreme Gradient Boosting* com Busca Bayesiana;

x) XGB_RS: *Extreme Gradient Boosting* com Busca Randomizada;

xi) CB_BS: *Categorical Boosting* com Busca Bayesiana; e

xii) CB_RS: *Categorical Boosting* com Busca Randomizada;

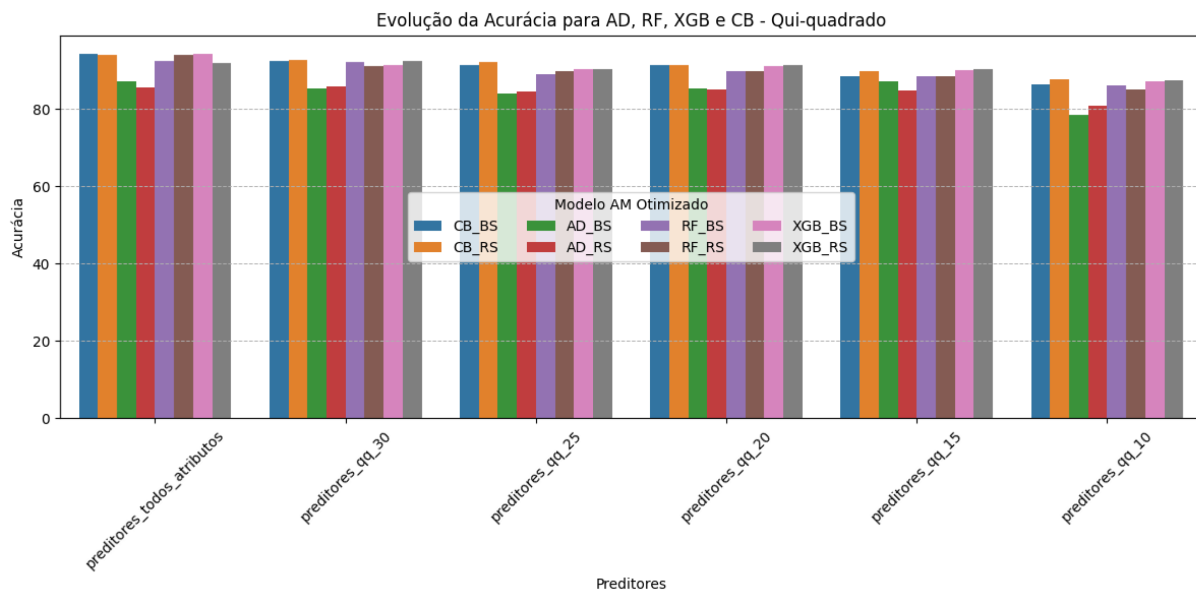


Figura 23 – Evolução da Acurácia - Qui-Quadrado

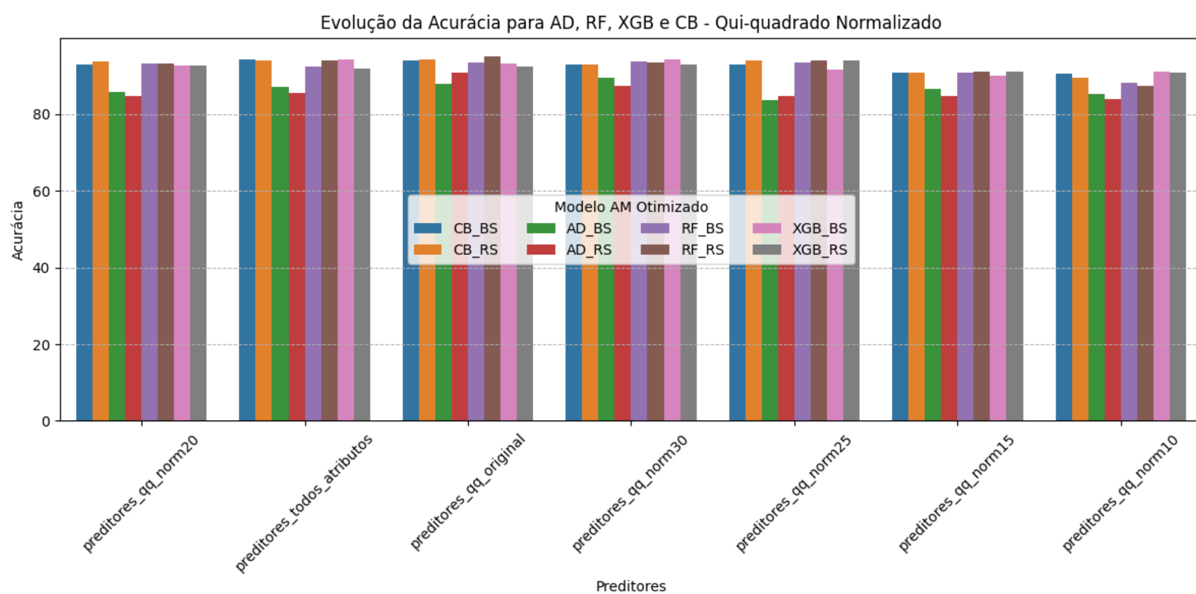


Figura 24 – Evolução da Acurácia - Qui-Quadrado com Atributos Normalizados

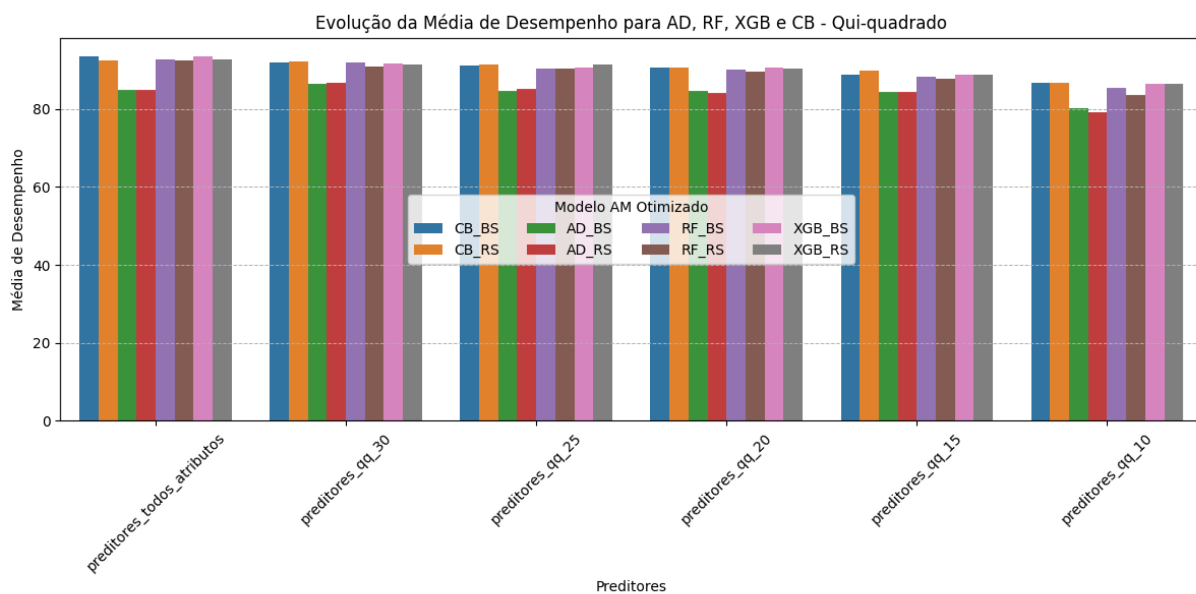


Figura 25 – Evolução da Média de Desempenho - Qui-Quadrado

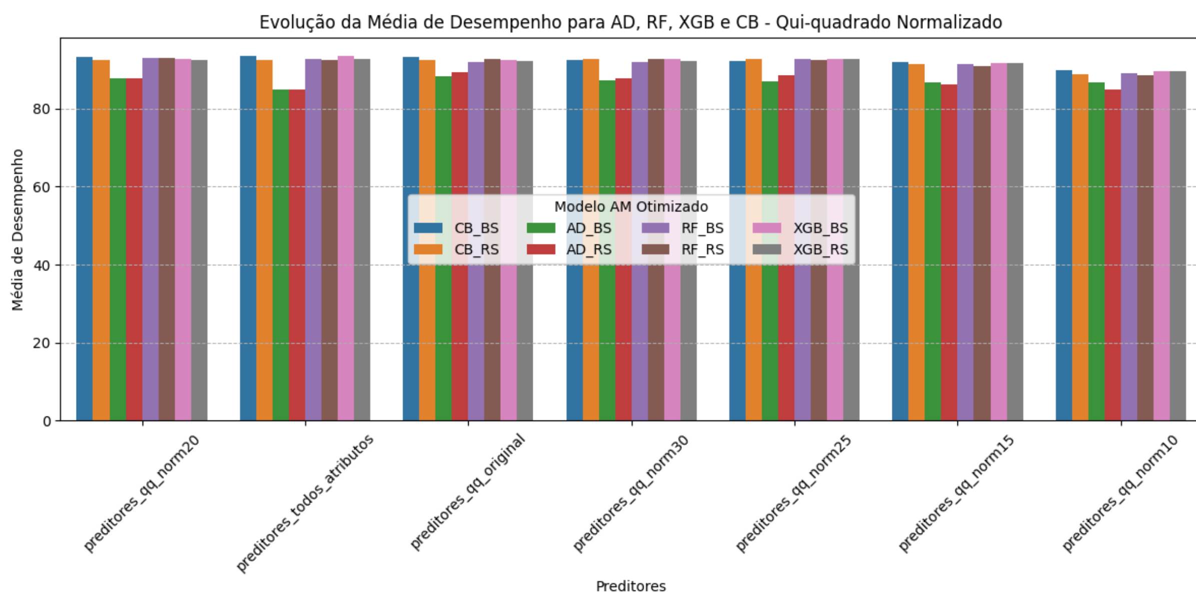


Figura 26 – Evolução da Média de Desempenho - Qui-Quadrado com Atributos Normalizados

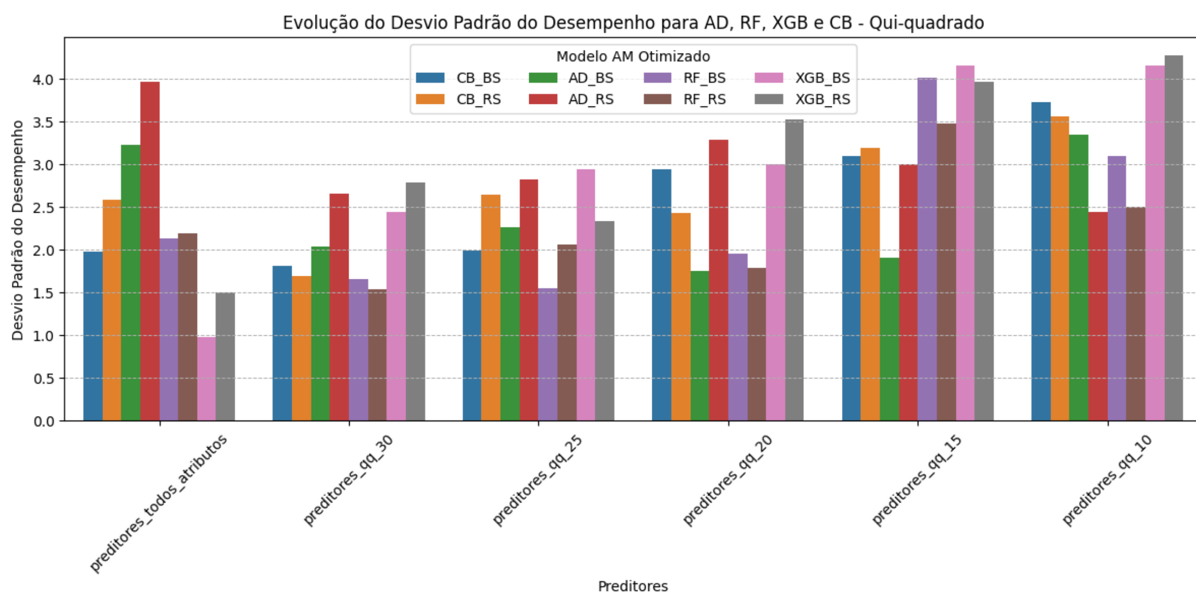


Figura 27 – Evolução da Desvio Padrão do Desempenho - Qui-Quadrado

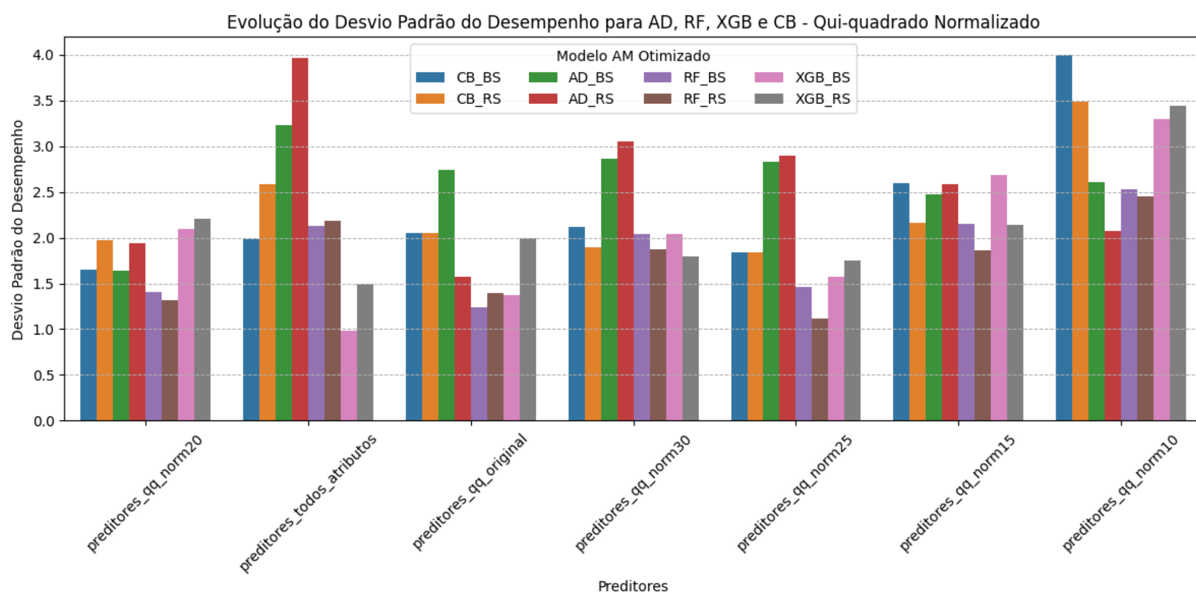


Figura 28 – Evolução da Desvio Padrão do Desempenho - Qui-Quadrado com Atributos Normalizados

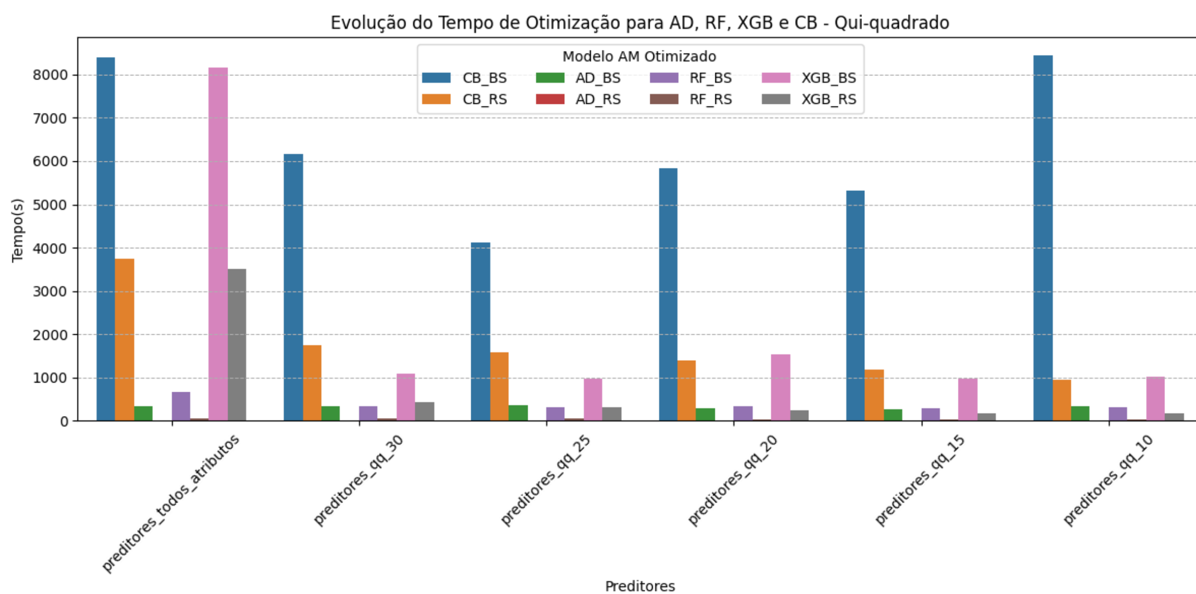


Figura 29 – Evolução do Tempo de Otimização de Hiperparâmetros - Qui-Quadrado

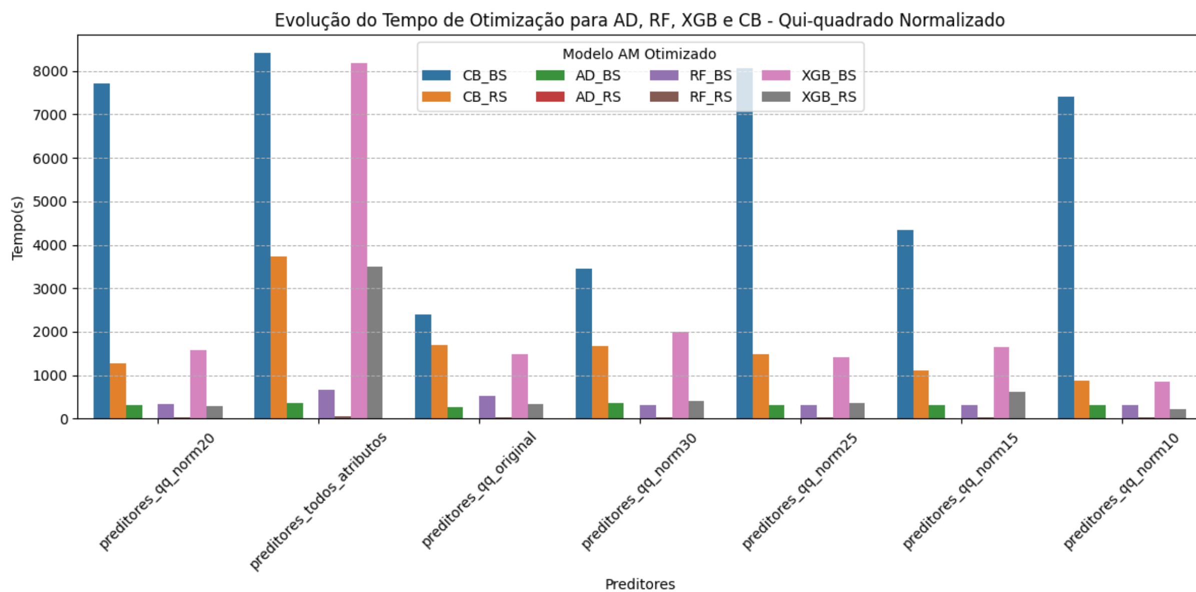


Figura 30 – Evolução do Tempo de Otimização de Hiperparâmetros - Qui-Quadrado com Atributos Normalizados

APÊNDICE G – DIAGRAMAS DE CLASSES DO ANALISADOR DE MÉTRICAS DE CÓDIGO

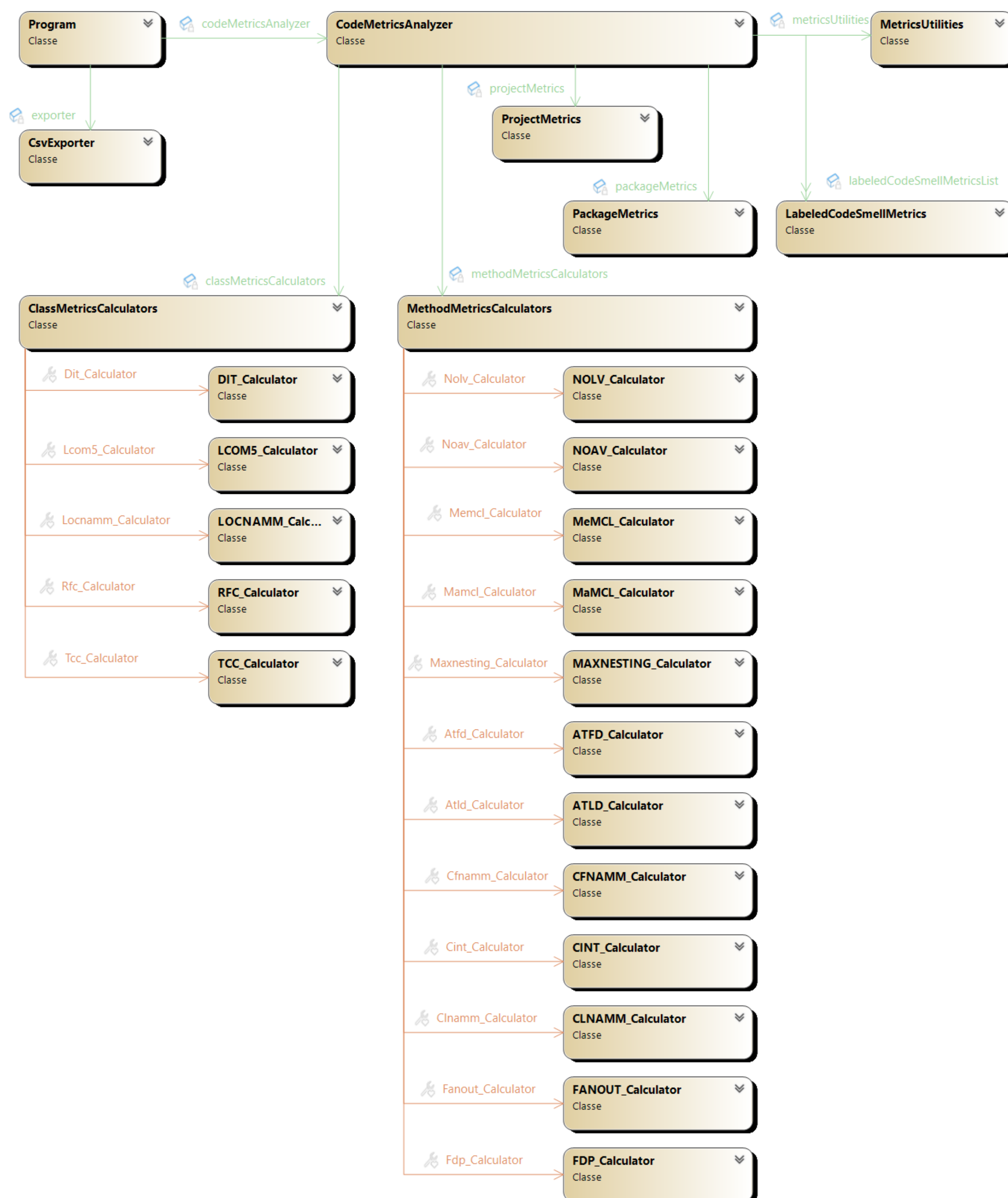


Figura 31 – Diagrama de Classes do Analisador de Métricas de Código

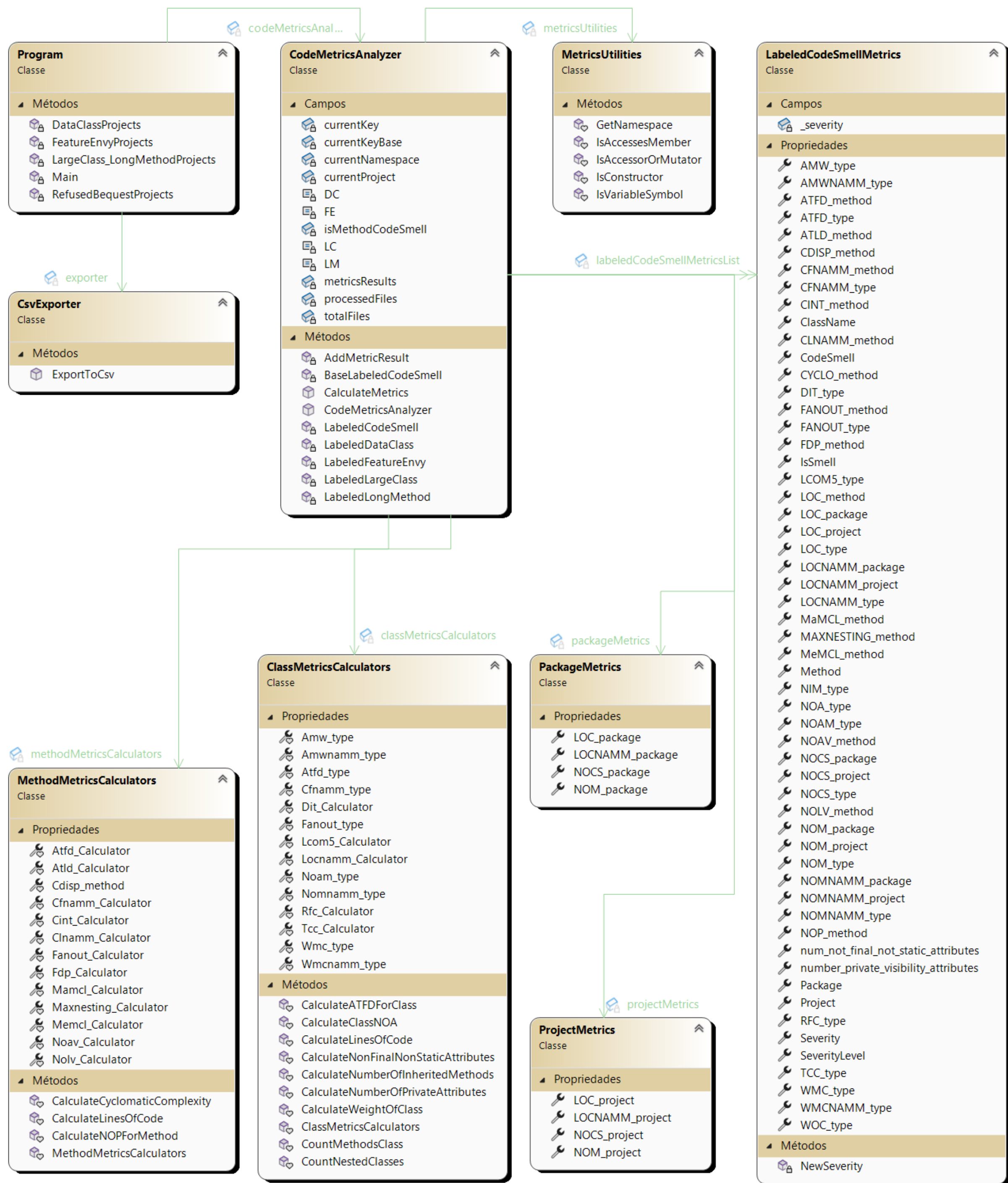


Figura 32 – Diagrama das Principais Classes do Analisador de Métricas de Código

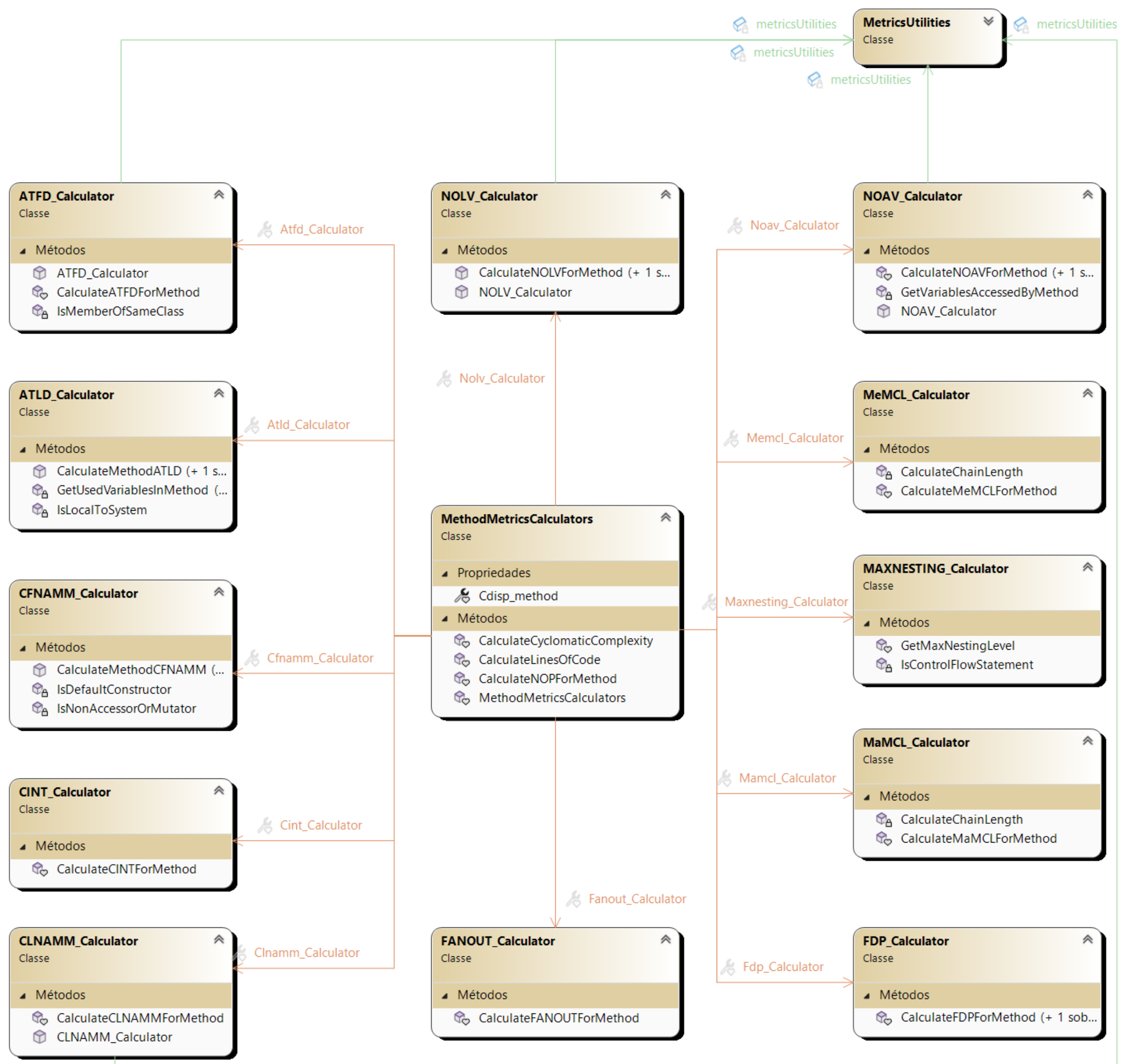


Figura 33 – Diagrama de Classes dos Analisadores de Métricas de Métodos

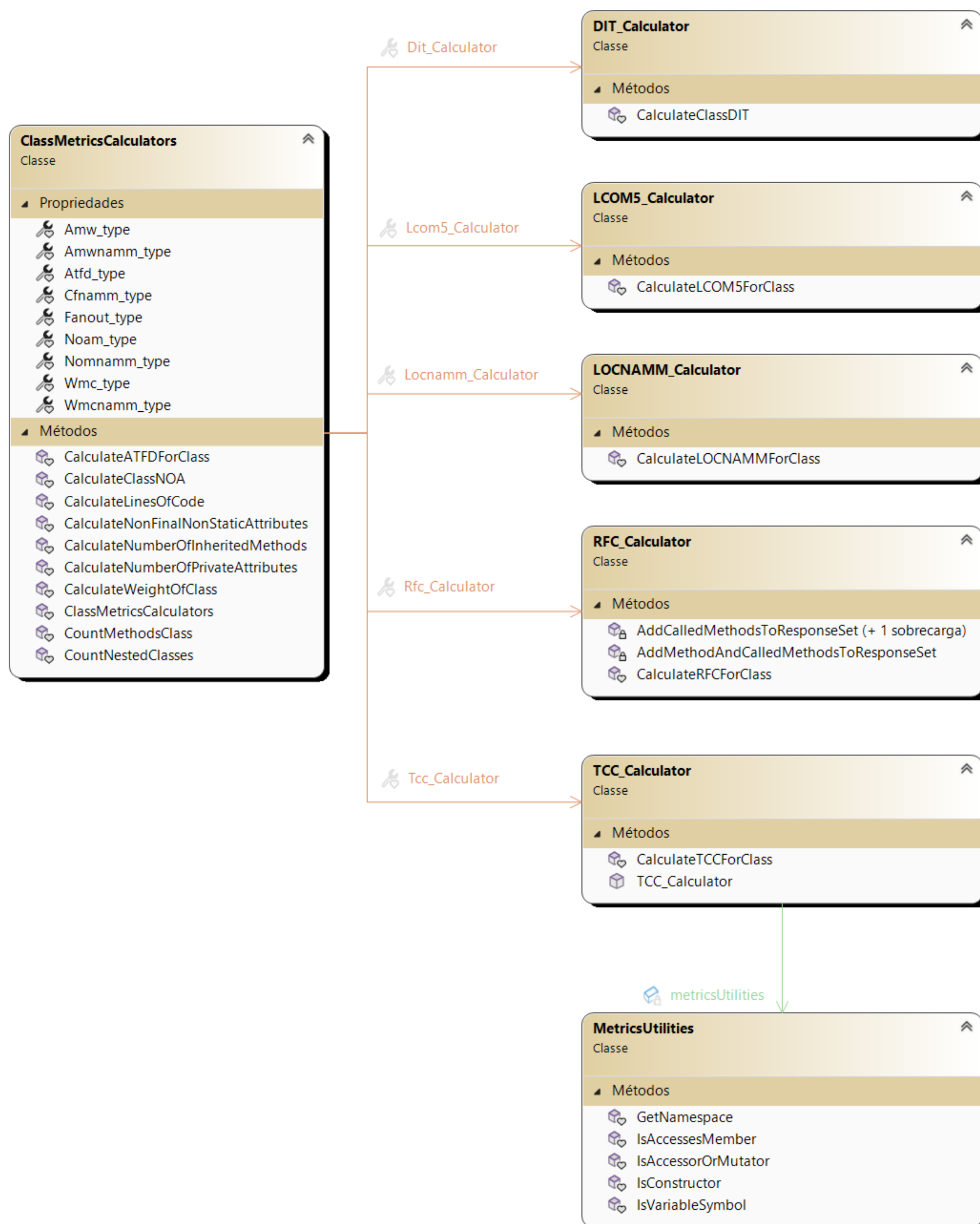


Figura 34 – Diagrama de Classes dos Analisadores de Métricas de Classes

APÊNDICE H – RESULTADOS DETALHADOS DA TRANSFERÊNCIA DE APRENDIZADO

H.1 Resultados da Transferência de Aprendizado de C# para Java

As tabelas 37, 38, 39 e 40 apresentam os resultados dos MCs com e sem TA para o conjunto de dados de projetos Java referentes aos *code smells* GC, LM, DC e FE, respectivamente.

Tabela 37 – Resultados para conjunto de dados GC Java

Uso de TA	Etapa	P	S	F1	Acc
Sem TA	DetCS	0.96	0.98	0.97	0.96
	AvGrav	0.93	0.88	0.89	0.88
	Duas-etapas	0.95	0.93	0.93	0.92
Com TA	DetCS	1.00	0.98	0.99	0.99
	AvGrav	0.94	0.91	0.91	0.91
	Duas-etapas	**0.97**	**0.95**	**0.95**	**0.95**

Tabela 38 – Resultados para conjunto de dados LM Java

Uso de TA	Etapa	P	S	F1	Acc
Sem TA	DetCS	0.98	0.96	0.97	0.97
	AvGrav	0.90	0.86	0.88	0.86
	Duas-etapas	0.94	0.91	0.93	0.92
Com TA	DetCS	1.00	1.00	1.00	1.00
	AvGrav	0.90	0.90	0.90	0.90
	Duas-etapas	**0.95**	**0.95**	**0.95**	**0.95**

Tabela 39 – Resultados para conjunto de dados DC Java

Uso de TA	Etapa	P	S	F1	Acc
Sem TA	DetCS	1.00	0.94	0.97	0.96
	AvGrav	0.79	0.73	0.74	0.73
	Duas-etapas	0.90	0.84	0.86	0.85
Com TA	DetCS	0.96	0.99	0.98	0.96
	AvGrav	0.83	0.78	0.78	0.78
	Duas-etapas	**0.90**	**0.89**	**0.88**	**0.87**

Tabela 40 – Resultados para conjunto de dados FE Java

Uso de TA	Etapa	P	S	F1	Acc
	DetCS	1.00	0.91	0.95	0.92
Sem TA	AvGrav	0.81	0.60	0.68	0.60
	Duas-etapas	0.91	0.76	0.82	0.76
	DetCS	0.98	0.99	0.98	0.97
Com TA	AvGrav	0.95	0.86	0.90	0.86
	Duas-etapas	**0.97**	**0.93**	**0.94**	**0.92**

H.2 Resultados da Transferência de Aprendizado de Java para C#

Os resultados dos MC sem e com TA para o conjunto de dados de projetos C# são apresentados nas Tabelas 41, 42, 43 e 44.

Tabela 41 – Resultados para conjunto de dados GC C#

Uso de TA	Etapa	P	S	F1	Acc
	DetCS	0.93	0.98	0.96	0.94
Sem TA	AvGrav	0.86	0.82	0.84	0.82
	Duas-etapas	0.90	0.90	0.90	0.88
	DetCS	0.97	0.97	0.97	0.95
Com TA	AvGrav	0.87	0.85	0.85	0.85
	Duas-etapas	**0.92**	**0.91**	**0.91**	**0.90**

Tabela 42 – Resultados para conjunto de dados LM C#

Uso de TA	Etapa	P	S	F1	Acc
	DetCS	0.95	0.95	0.95	0.92
Sem TA	AvGrav	0.79	0.78	0.78	0.78
	Duas-etapas	0.87	0.87	0.87	0.85
	DetCS	0.96	0.94	0.95	0.93
Com TA	AvGrav	0.79	0.79	0.79	0.79
	Duas-etapas	**0.88**	**0.87**	**0.87**	**0.86**

Tabela 43 – Resultados para conjunto de dados DC C#

Uso de TA	Etapa	P	S	F1	Acc
	DetCS	1.00	0.94	0.97	0.94
Sem TA	AvGrav	0.75	0.25	0.38	0.25
	Duas-etapas	0.88	0.60	0.68	0.60
	DetCS	1.00	0.99	0.99	0.99
Com TA	AvGrav	0.75	0.75	0.75	0.75
	Duas-etapas	**0.88**	**0.87**	**0.87**	**0.87**

Tabela 44 – Resultados para conjunto de dados FE C#

Uso de TA	Etapa	P	S	F1	Acc
Sem TA	DetCS	0.99	0.92	0.96	0.92
	AvGrav	0.80	0.30	0.44	0.30
	Duas-etapas	0.90	0.61	0.70	0.61
Com TA	DetCS	0.99	0.99	0.99	0.98
	AvGrav	1.00	0.90	0.95	0.90
	Duas-etapas	**1.00**	**0.95**	**0.97**	**0.94**

APÊNDICE I – AVALIAÇÃO DA GRAVIDADE DO MÉTODO CAPTUREWITHCURSOR

Este apêndice apresenta uma análise do método *CaptureWithCursor(FrameInfo frame)*, que foi predito como LM Grave, mas que está rotulado como FE não Grave no conjunto de dados C#. Este método está disponível ao final deste capítulo, apresentando uma combinação de características que podem ser analisadas para avaliar a sua gravidade tanto como FE não Grave quanto LM Grave. Destaca-se os pontos-chave para facilitar a análise:

1. **Análise para FE nao Grave:** o método faz várias chamadas a objetos externos como *DuplicatedOutput, Device, ResourceRegion, BackingTexture*, entre outros. Essas chamadas indicam uma dependência significativa de outras classes para executar tarefas importantes, o que pode ser um indicador de FE. Contudo, é necessário avaliar se essas chamadas são adequadas ao contexto da classe atual. Embora existam múltiplas interações, elas parecem ser realizadas de forma legítima para capturar e processar os quadros da tela, o que pode ser parte do propósito da classe.
2. **Análise LM Grave:** o método contém mais de 200 linhas de código, com múltiplas seções (e.g., *try/catch, loops* e condicionais). Há diversas estruturas de controle (*if/else, loops* e regiões de código) que tornam o método mais difícil de seguir. Além disso, o método executa várias tarefas, como capturar quadros, processar retângulos movidos, retângulos sujos, e gerenciar texturas. E, ainda, as regiões de código sugerem a possibilidade de dividir o método em submétodos menores com responsabilidades mais específicas, o que tornaria o código mais modular e legível.

Com base nas características do método apresentadas, a gravidade deste método pode ser considerado como FE não grave, pois há uma interação com os objetos externos de forma legítima e parte da responsabilidade da classe. Além disso, o método apesar de estar acessando repetidamente dados de outra classe, não há indícios de quebra de encapsulamento. Entretanto, há forte indicação de que o *code smell* de gravidade LM Grave, devido ao tamanho excessivo, à complexidade e à falta de modularidade do método. Neste sentido é recomendável dividir o método em partes menores, cada uma lidando com uma tarefa específica. Além de revisar as dependências externas para confirmar se são essenciais ou se podem ser minimizadas.

```
1  public override int CaptureWithCursor(FrameInfo frame)
2  {
3      var res = new Result(-1);
4
5      try
6      {
7          //Try to get the duplicated output frame within given time.
8          res = DuplicatedOutput.TryAcquireNextFrame(0, out var info, out var resource);
9
10         //Checks how to proceed with the capture. It could have failed, or the screen, cursor or both
              could have been captured.
11         if (FrameCount == 0 && info.LastMouseUpdateTime == 0 && (res.Failure || resource == null))
12         {
13             //Somehow, it was not possible to retrieve the resource, frame or metadata.
14             resource?.Dispose();
15             return FrameCount;
16         }
17         else if (FrameCount == 0 && info.TotalMetadataBufferSize == 0 && info.LastMouseUpdateTime > 0)
18         {
19             //Sometimes, the first frame has cursor info, but no screen changes.
20             GetCursor(null, info, frame);
21             resource?.Dispose();
22             return FrameCount;
23         }
24
25         #region Process changes
```

```
26
27          //Something on screen was moved or changed.
28          if (info.TotalMetadataBufferSize > 0)
29          {
30              //Copies the screen data into memory that can be accessed by the CPU.
31              using (var screenTexture = resource.QueryInterface<Texture2D>())
32              {
33                  #region Moved rectangles
34
35                  var movedRectangles = new OutputDuplicateMoveRectangle[info.TotalMetadataBufferSize];
36                  DuplicatedOutput.GetFrameMoveRects(movedRectangles.Length, movedRectangles, out var
                        movedRegionsLength);
37
38                  for (var movedIndex = 0; movedIndex < movedRegionsLength / Marshal.SizeOf(typeof(
                        OutputDuplicateMoveRectangle)); movedIndex++)
39                  {
40                      //Crop the destination rectangle to the screen area rectangle.
41                      var left = Math.Max(movedRectangles[movedIndex].DestinationRect.Left, Left -
                            OffsetLeft);
42                      var right = Math.Min(movedRectangles[movedIndex].DestinationRect.Right, Left +
                            Width - OffsetLeft);
43                      var top = Math.Max(movedRectangles[movedIndex].DestinationRect.Top, Top -
                            OffsetTop);
44                      var bottom = Math.Min(movedRectangles[movedIndex].DestinationRect.Bottom, Top +
                            Height - OffsetTop);
45
46                      //Copies from the screen texture only the area which the user wants to capture.
```

```
47                     if (right > left && bottom > top)
48                     {
49                         //Limit the source rectangle to the available size within the destination
                              rectangle.
50                         var sourceWidth = movedRectangles[movedIndex].SourcePoint.X + (right - left);
51                         var sourceHeight = movedRectangles[movedIndex].SourcePoint.Y + (bottom - top);
52 
53                         Device.ImmediateContext.CopySubresourceRegion(screenTexture, 0,
54                             new ResourceRegion(movedRectangles[movedIndex].SourcePoint.X,
                                  movedRectangles[movedIndex].SourcePoint.Y, 0, sourceWidth, sourceHeight
                                  , 1),
55                             BackingTexture, 0, left - (Left - OffsetLeft), top - (Top - OffsetTop));
56                     }
57                 }
58 
59                 #endregion
60 
61                 #region Dirty rectangles
62 
63                 var dirtyRectangles = new RawRectangle[info.TotalMetadataBufferSize];
64                 DuplicatedOutput.GetFrameDirtyRects(dirtyRectangles.Length, dirtyRectangles, out var
                      dirtyRegionsLength);
65 
66                 for (var dirtyIndex = 0; dirtyIndex < dirtyRegionsLength / Marshal.SizeOf(typeof(
                      RawRectangle)); dirtyIndex++)
67                 {
68                     //Crop screen positions and size to frame sizes.
```

```
69                      var left = Math.Max(dirtyRectangles[dirtyIndex].Left, Left - OffsetLeft);
70                      var right = Math.Min(dirtyRectangles[dirtyIndex].Right, Left + Width - OffsetLeft)
                           ;
71                      var top = Math.Max(dirtyRectangles[dirtyIndex].Top, Top - OffsetTop);
72                      var bottom = Math.Min(dirtyRectangles[dirtyIndex].Bottom, Top + Height - OffsetTop
                           );
73
74                      //int left, right, top, bottom;
75                      //switch (DisplayRotation)
76                      //{
77                      //    case DisplayModeRotation.Rotate90:
78                      //    {
79                      //        //TODO:
80                      //        left = Math.Max(dirtyRectangles[dirtyIndex].Left, Left - OffsetLeft);
81                      //        right = Math.Min(dirtyRectangles[dirtyIndex].Right, Left + Width -
                           OffsetLeft);
82                      //        top = Math.Max(dirtyRectangles[dirtyIndex].Top, Top - OffsetTop);
83                      //        bottom = Math.Min(dirtyRectangles[dirtyIndex].Bottom, Top + Height -
                           OffsetTop);
84
85                      //        //left = Math.Min(dirtyRectangles[dirtyIndex].Bottom, Top + Height -
                           OffsetTop);
86                      //        //right = Math.Max(dirtyRectangles[dirtyIndex].Top, Top - OffsetTop);
87                      //        //top = Math.Max(dirtyRectangles[dirtyIndex].Left, Left - OffsetLeft);
88                      //        //bottom = Math.Min(dirtyRectangles[dirtyIndex].Right, Left + Width -
                           OffsetLeft);
89
```

```
90                      //        break;
91                      //    }
92
93                      //    case DisplayModeRotation.Rotate180:
94                      //    {
95                      //        //TODO:
96                      //        left = Math.Max(dirtyRectangles[dirtyIndex].Top + OffsetTop, Top);
97                      //        right = Math.Min(dirtyRectangles[dirtyIndex].Bottom + OffsetTop, Top +
                           Height);
98                      //        top = Math.Min(dirtyRectangles[dirtyIndex].Right + OffsetLeft, Left +
                           Width);
99                      //        bottom = Math.Max(dirtyRectangles[dirtyIndex].Left + OffsetLeft, Left);
100                     //        break;
101                     //    }
102
103                     //    default:
104                     //    {
105                     //        //In this context, the screen positions are relative to the current
                           screen, not to the whole set of screens (virtual space).
106                     //        left = Math.Max(dirtyRectangles[dirtyIndex].Left, Left - OffsetLeft);
107                     //        right = Math.Min(dirtyRectangles[dirtyIndex].Right, Left + Width -
                           OffsetLeft);
108                     //        top = Math.Max(dirtyRectangles[dirtyIndex].Top, Top - OffsetTop);
109                     //        bottom = Math.Min(dirtyRectangles[dirtyIndex].Bottom, Top + Height -
                           OffsetTop);
110                     //        break;
111                     //    }
```

```
112                     //}
113 
114                     //Copies from the screen texture only the area which the user wants to capture.
115                     if (right > left && bottom > top)
116                         Device.ImmediateContext.CopySubresourceRegion(screenTexture, 0, new
                               ResourceRegion(left, top, 0, right, bottom, 1), BackingTexture, 0, left - (
                               Left - OffsetLeft), top - (Top - OffsetTop));
117                 }
118 
119                 #endregion
120             }
121         }
122 
123         if (info.TotalMetadataBufferSize > 0 || info.LastMouseUpdateTime > 0)
124         {
125             //Copy the captured desktop texture into a staging texture, in order to show the mouse
                   cursor and not make the captured texture dirty with it.
126             Device.ImmediateContext.CopyResource(BackingTexture, StagingTexture);
127 
128             //Gets the cursor image and merges with the staging texture.
129             GetCursor(StagingTexture, info, frame);
130         }
131 
132         //Saves the most recent capture time.
133         LastProcessTime = Math.Max(info.LastPresentTime, info.LastMouseUpdateTime);
134 
135         #endregion
```

```
136
137         #region Gets the image data
138
139         //Gets the staging texture as a stream.
140         var data = Device.ImmediateContext.MapSubresource(StagingTexture, 0, MapMode.Read, MapFlags.
               None, out var stream);
141
142         if (data.IsEmpty)
143         {
144             Device.ImmediateContext.UnmapSubresource(StagingTexture, 0);
145             stream?.Dispose();
146             resource?.Dispose();
147             return FrameCount;
148         }
149
150         //Sets the frame details.
151         FrameCount++;
152         frame.Path = $"{Project.FullPath}{FrameCount}.png";
153         frame.Delay = FrameRate.GetMilliseconds();
154         frame.DataLength = stream.Length;
155         frame.Data = new byte[stream.Length];
156
157         //BGRA32 is 4 bytes.
158         for (var height = 0; height < Height; height++)
159         {
160             stream.Position = height * data.RowPitch;
161             Marshal.Copy(new IntPtr(stream.DataPointer.ToInt64() + height * data.RowPitch), frame.Data
```

```
                    , height * Width * 4, Width * 4);
162           }
163
164           BlockingCollection.Add(frame);
165
166           #endregion
167
168           Device.ImmediateContext.UnmapSubresource(StagingTexture, 0);
169           stream.Dispose();
170           resource?.Dispose();
171
172           return FrameCount;
173       }
174       catch (SharpDXException se) when (se.ResultCode.Code == SharpDX.DXGI.ResultCode.WaitTimeout.Result
              .Code)
175       {
176           return FrameCount;
177       }
178       catch (SharpDXException se) when (se.ResultCode.Code == SharpDX.DXGI.ResultCode.DeviceRemoved.
              Result.Code || se.ResultCode.Code == SharpDX.DXGI.ResultCode.DeviceReset.Result.Code)
179       {
180           //When the device gets lost or reset, the resources should be instantiated again.
181           DisposeInternal();
182           Initialize();
183
184           return FrameCount;
185       }
```

```
186     catch (Exception ex)
187     {
188         LogWriter.Log(ex, "It␣was␣not␣possible␣to␣finish␣capturing␣the␣frame␣with␣DirectX.");
189 
190         MajorCrashHappened = true;
191         Application.Current.Dispatcher.Invoke(() => OnError.Invoke(ex));
192         return FrameCount;
193     }
194     finally
195     {
196         try
197         {
198             //Only release the frame if there was a sucess in capturing it.
199             if (res.Success)
200                 DuplicatedOutput.ReleaseFrame();
201         }
202         catch (Exception e)
203         {
204             LogWriter.Log(e, "It␣was␣not␣possible␣to␣release␣the␣frame.");
205         }
206     }
207 }
```

APÊNDICE J – GRÁFICOS DE EXPLICABILIDADE - FE JAVA

J.1 Importância dos Atributos

A Figura 35 mostra a importância dos atributos para detecção de FE em projetos Java. Os atributos foram selecionados com a utilização da técnica ANOVA, 30% dos atributos disponíveis. Para calcular a importância dos atributos foi utilizado o *LossFunctionChange* para o método *get_feature_importance* de CatBoost. Esse tipo de importância de atributo pode ser usado para qualquer modelo, mas é particularmente útil para classificar modelos, onde outros tipos de importância de atributo podem fornecer resultados enganosos. Para cada atributo, o valor representa a diferença entre o valor de perda do modelo com esse atributo e sem ele. As métricas que mais se destacaram para detecção de FE em projetos Java foram AFTD_type e MAXNESTING_method.

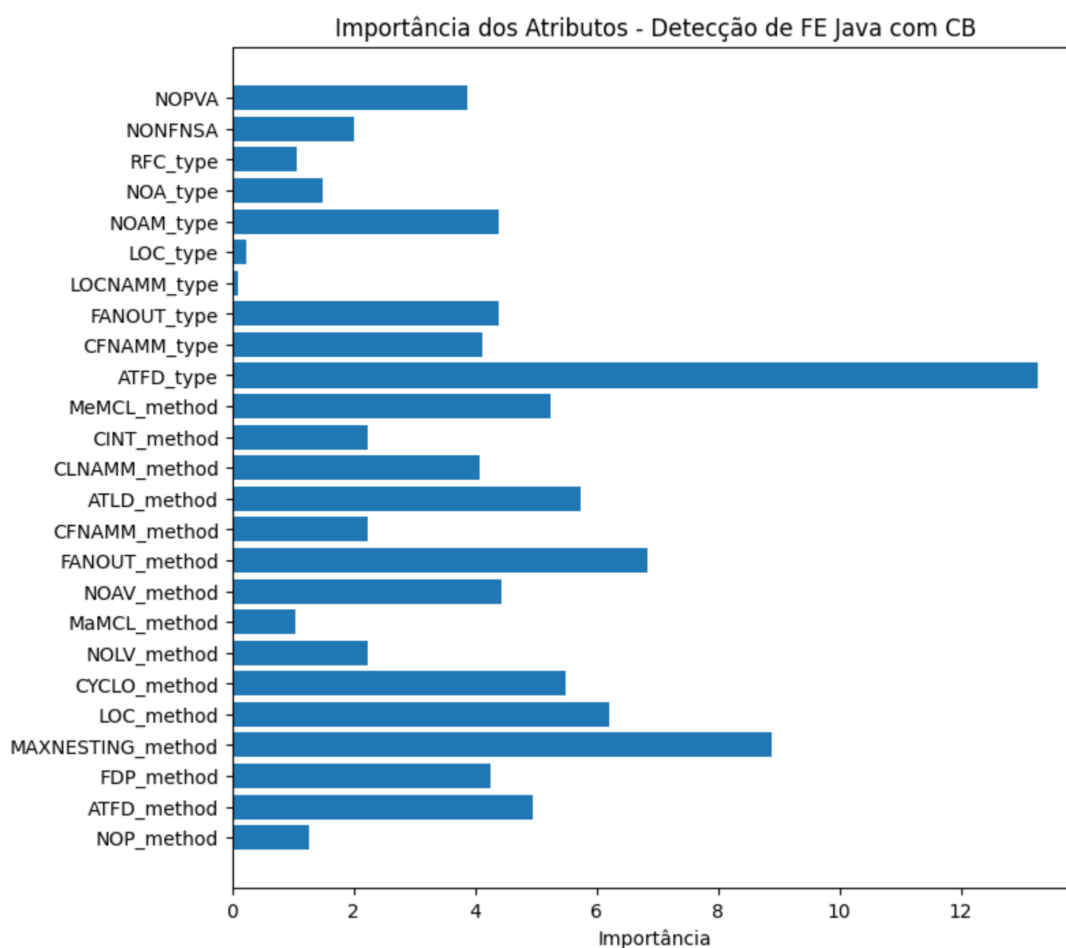


Figura 35 – Importância dos Atributos para Detecção de FE Java

A Figura 36 mostra a importância dos atributos para avaliação de gravidade de FE

em projetos Java. Os atributos foram selecionados com a utilização da técnica ANOVA, 25% dos atributos disponíveis. O Calculo da importância de atributos, com base em um classificador RF, foram realizadas computando a redução total de impureza do critério Gini ao longo das árvores do modelo. As métricas que mais se destacaram para avaliação de gravidade FE em projetos Java foram AFTD_type e MAXNESTING_method.

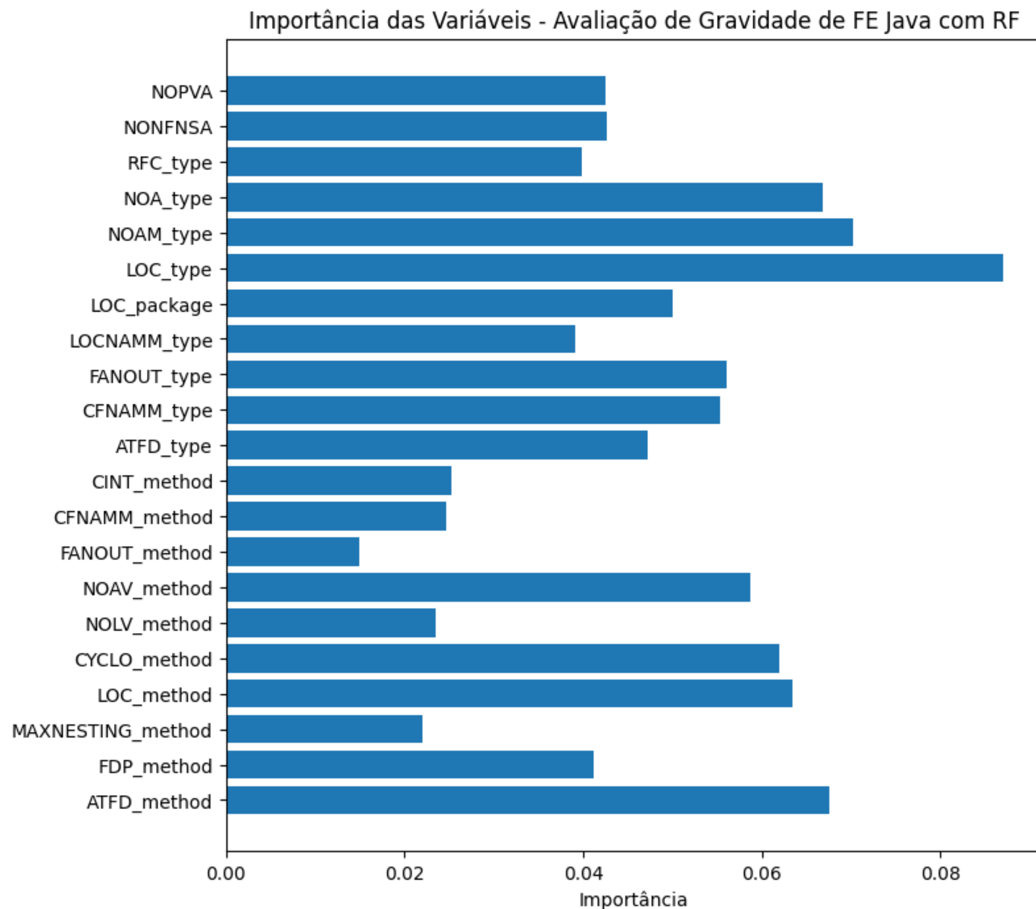


Figura 36 – Importância dos Atributos para Avaliação das Gravidades FE Java

J.2 SHAP

SHAP é uma abordagem teórica de jogos para explicar a saída de qualquer modelo de aprendizado de máquina. Ela conecta alocação de crédito ótima com explicações locais usando os valores clássicos de Shapley da teoria dos jogos e suas extensões relacionadas. Além disso, SHAP é particularmente útil para entender o impacto de cada variável tanto em nível global quanto local, em qualquer tipo de modelo (64). Para extrair alguma informação global a partir das importâncias locais, foi utilizado o gráfico resumo (*summary plot*) do SHAP para medir o impacto da saída do modelo CB na detecção de FE, veja Figura 37. Nos gráficos de resumo de SHAP, cada linha representa uma variável e cada ponto corresponde a uma instância do conjunto de dados. A cor do ponto indica o valor

do atributo para aquele exemplo: azul para valores baixos e vermelho para valores altos. A posição no eixo horizontal mostra o valor de SHAP, i.e., o efeito do atributo: quanto mais à direita, maior a contribuição positiva daquele atributo para a instância (64).

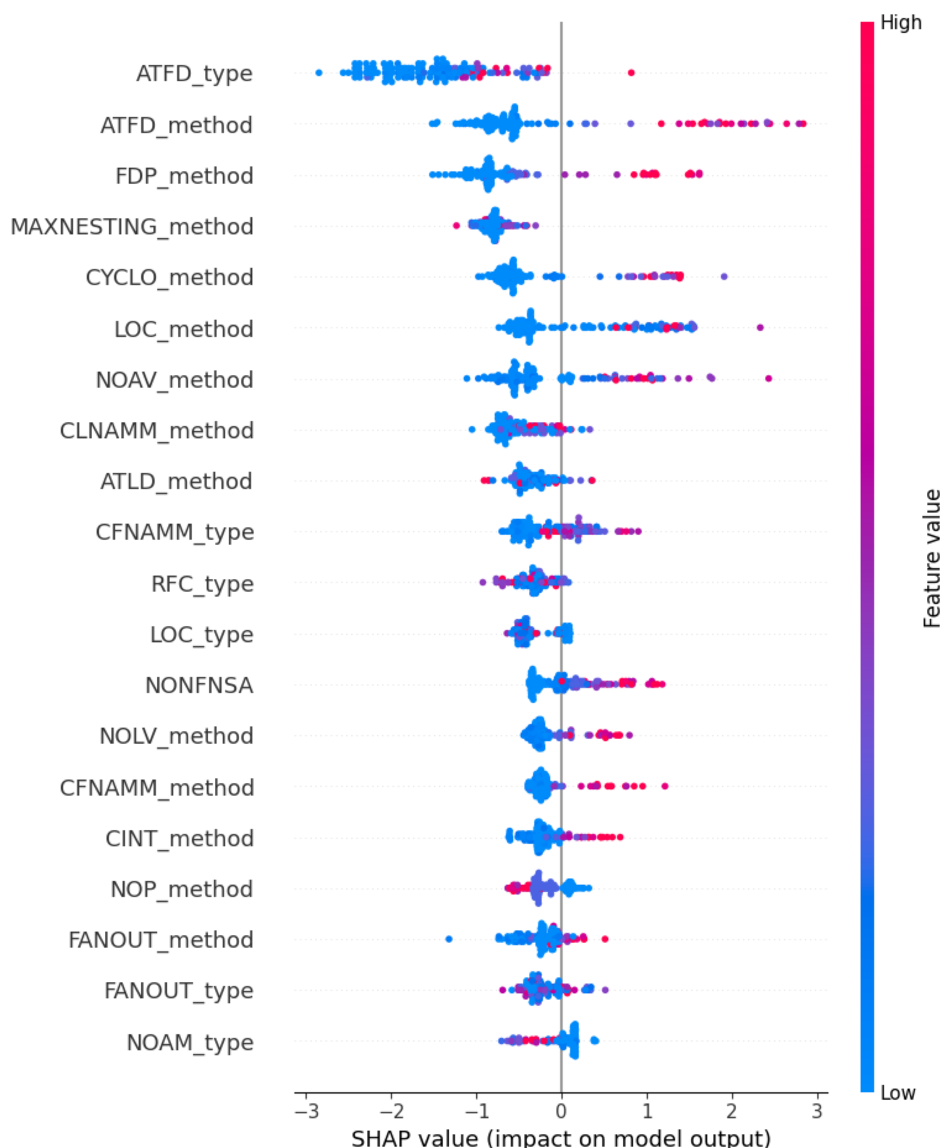


Figura 37 – Impacto na saída do modelo de detecção de FE Java - SHAP

No gráfico acima, Figura 37, os atributos ATFD_type, AFTD_method e FDP_method contribuem de forma diferente dependendo do seu valor para a resposta do modelo: i) para valores mais baixos (mais azul), contribuem negativamente e ii) para valores mais altos (mais vermelho), contribuem positivamente. De forma geral, é seguro assumir que variáveis com maior dispersão de contribuições no eixo horizontal tendem a ser mais importantes. Isso ocorre porque uma ampla variação nos valores de SHAP indica que a variável está afetando a predição de diferentes exemplos, sugerindo uma influência maior na tomada de decisão do modelo[1].

[1] <https://medium.com/big-data-blog/shap-o-que-%C3%A9-e-por-que-usar-6b01d37ae592>

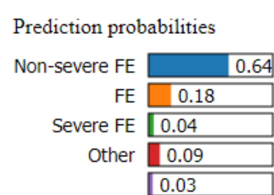
J.3 LIME

O LIME é um algoritmo que explica as previsões locais do modelo ajustando um modelo interpretável simples (como uma regressão linear) próximo ao ponto de interesse. É útil para entender previsões específicas e para modelos complexos como RF e *Boosting* (67). Por isso, utilizou-se este algoritmo para a explicabilidade dos classificadores para avaliação de gravidade de *code smells*. Para gravidades de FE em projetos Java foi utilizado um modelo baseado em RF com os dados padronizados.

A Figura 38 apresenta o gráfico LIME para uma instância de FE não grave possui, neste caso, 64% de probabilidade de ser classificada corretamente pelo modelo de avaliação de gravidade. Esta instância possui $NOA_type > 0.19$, $-0.23 < ATFD_method \leq 0.64$ e $-0.39 < CINT_method \leq 1.18$ de valores padronizados, o que aumenta a sua probabilidade de acerto. Entretanto, os valores de $ATFD_type > 0.34$ e $RFC_type > 0.00$, tem importância para reduzir a probabilidade de acerto desta instância pelo modelo.

No caso de instância de gravidade FE, veja Figura 39, houve uma probabilidade de acerto de 81%. Sendo os valores de $-0.13 < NOAV_method \leq 1.28$, $-0.22 < NOLV_method \leq 1.43$ e $-0.47 < NOA_type \leq -0.24$ relevantes para uma probabilidade maior de acerto do modelo. Enquanto, os valores de $CINT_method > 1.18$ e $FDP_method > 1.56$ são fatores que diminuem a probabilidade de acerto.

Para instância de FE grave, que é detalhada na Figura 40, os valores de $ATFD_method > 0.64$, $NOAV_method > 1.28$, $ATFD_type > 0.34$ e $RFC_type > 0.00$ são preponderantes a probabilidade de predição de 76% para este nível mais severo de gravidade FE.



NOT Non-severe FE | Non-severe FE

atfd.t > 0.34 0.03
rfc.t > 0.00 0.03
noa.t > 0.19 0.03
-0.23 < atfd.m <= 0.64 0.02
-0.39 < cint.m <= 1.18 0.02
nonfnsa > 0.33 0.02
-0.13 < noav.m <= 1.28 0.02
fanout.t > 0.70 0.01
loc.t > 0.22 0.01
-0.09 < cyclo.m <= 0.99 0.01
cfnamm.t > 0.42 0.01
loc.k > 0.06 0.01
-0.02 < fdp.m <= 1.56 0.01
-0.21 < nopva <= 0.55 0.01
-0.30 < maxnest <= 1.73 0.01
-0.33 < noam.t <= -0.12 0.01
-0.22 < nolv.m <= 1.43 0.01
-0.67 < fanout.m <= 1.49 0.01
locnamm.t > 0.64 0.01
0.05 < loc.m <= 1.56 0.00
-0.35 < cfnam.m <= 1.37 0.00

NOT FE | FE

-0.13 < noav.m <= 1.28 0.05
nonfnsa > 0.33 0.03
noa.t > 0.19 0.03
-0.22 < nolv.m <= 1.43 0.03
-0.21 < nopva <= 0.55 0.01
-0.39 < cint.m <= 1.18 0.01
fanout.t > 0.70 0.01
0.05 < loc.m <= 1.56 0.01
-0.33 < noam.t <= -0.12 0.01
-0.02 < fdp.m <= 1.56 0.01
-0.09 < cyclo.m <= 0.99 0.01
atfd.t > 0.34 0.01
-0.23 < atfd.m <= 0.64 0.01
locnamm.t > 0.64 0.01
rfc.t > 0.00 0.01
loc.t > 0.22 0.01
-0.67 < fanout.m <= 1.49 0.00
-0.30 < maxnest <= 1.73 0.00
-0.35 < cfnam.m <= 1.37 0.00
cfnamm.t > 0.42 0.00
loc.k > 0.06 0.00

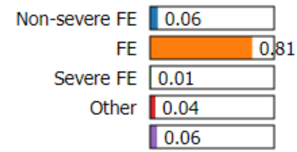
NOT Severe FE | Severe FE

atfd.t > 0.34 0.03
rfc.t > 0.00 0.03
-0.23 < atfd.m <= 0.64 0.01
-0.13 < noav.m <= 1.28 0.01
loc.k > 0.06 0.01
-0.39 < cint.m <= 1.18 0.00
cfnamm.t > 0.42 0.00
fanout.t > 0.70 0.00
-0.22 < nolv.m <= 1.43 0.00
noa.t > 0.19 0.00
locnamm.t > 0.64 0.00
-0.30 < maxnest <= 1.73 0.00
-0.02 < fdp.m <= 1.56 0.00
-0.67 < fanout.m <= 1.49 0.00
nonfnsa > 0.33 0.00
0.05 < loc.m <= 1.56 0.00
-0.09 < cyclo.m <= 0.99 0.00
-0.35 < cfnam.m <= 1.37 0.00
-0.21 < nopva <= 0.55 0.00
loc.t > 0.22 0.00
-0.33 < noam.t <= -0.12 0.00

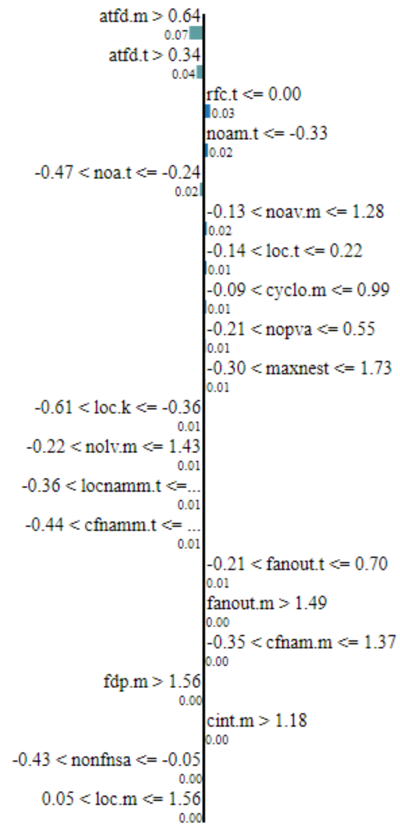
Feature	Value
atfd.t	2.30
rfc.t	3.53
noa.t	0.20
atfd.m	0.43
cint.m	0.33
nonfnsa	0.72
noav.m	0.39
fanout.t	5.78
loc.t	4.04
cyclo.m	0.10
cfnamm.t	3.32
loc.k	0.35
fdp.m	0.22
nopva	0.39
maxnest	0.86
noam.t	-0.14
nolv.m	-0.07
fanout.m	-0.11
locnamm.t	4.25
loc.m	0.09
cfnam.m	-0.15

Figura 38 – Detalhamento de instância de FE não Grave Java com LIME

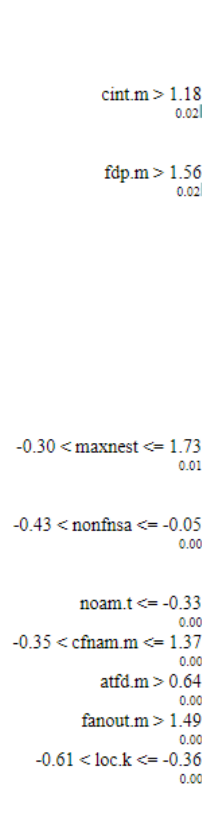
Prediction probabilities

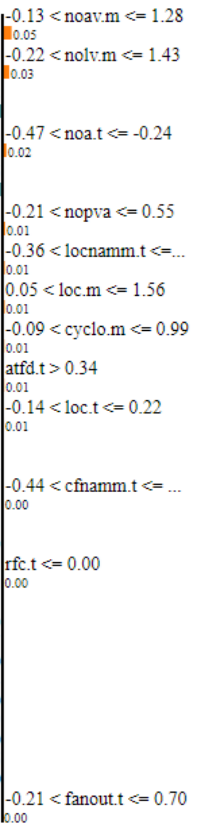


NOT Non-severe FE Non-severe FE

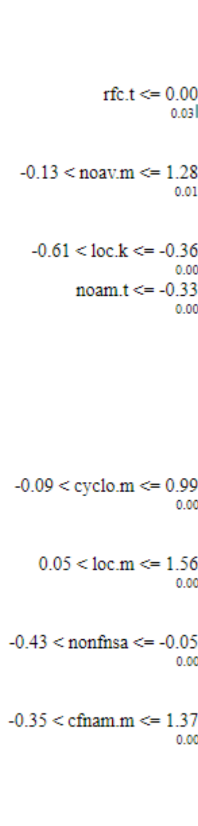


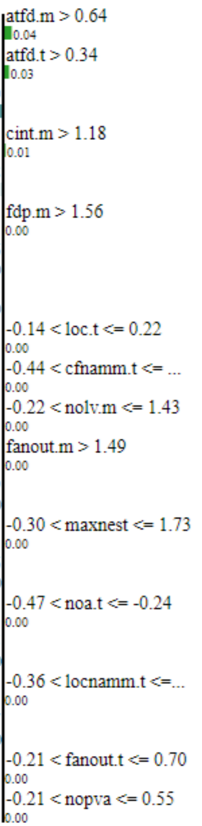
NOT FE FE





NOT Severe FE Severe FE





Feature	Value
atfd.m	1.21
atfd.t	0.34
rfc.t	-0.18
noam.t	-0.33
noa.t	-0.33
noav.m	0.95
loc.t	-0.06
cyclo.m	0.64
nopva	-0.12
maxnest	0.86
loc.k	-0.44
nolv.m	0.81
locnamm.t	-0.02
cfnamm.t	-0.39
fanout.t	0.32
fanout.m	1.51
cfnam.m	0.30
fdp.m	2.06
cint.m	1.43
nonfnsa	-0.32
loc.m	1.31

Figura 39 – Detalhamento de instância de Gravidade FE Java com LIME

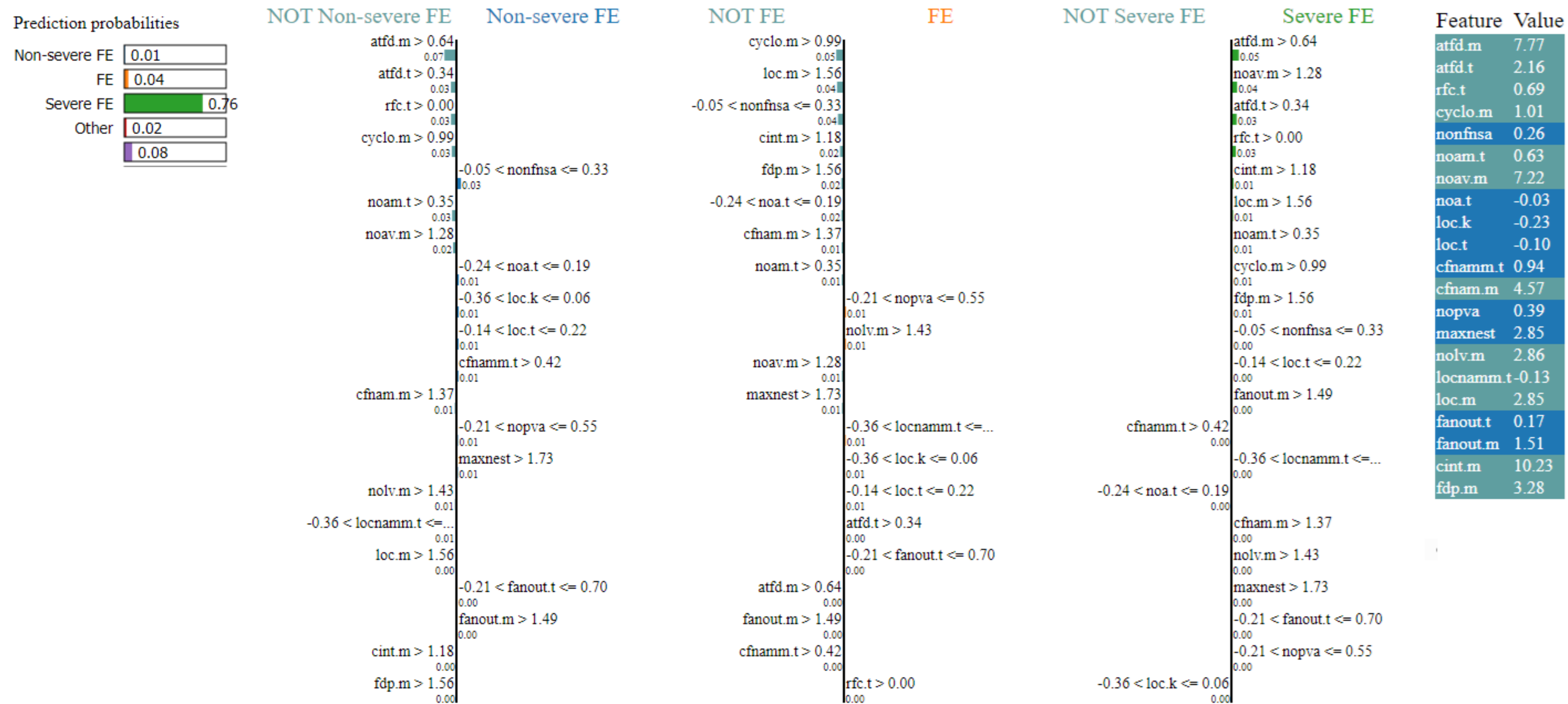


Figura 40 – Detalhamento de instância de FE Grave Java com LIME

APÊNDICE K – GRÁFICOS DE EXPLICABILIDADE - LM JAVA

K.1 Importância dos Atributos

Para calcular a importância dos atributos tanto para detecção de LM em projetos Java quanto para avaliação da sua gravidade foi utilizado o valor padrão do parâmetro *importance_type* para o XGB, “*weight*”, medindo a importância pelo número de vezes que um atributo aparece nos nós das árvores de decisão.

A Figura 41 mostra a importância dos atributos para detecção de LM em projetos Java. 25 atributos foram selecionados com a utilização da técnica ANOVA, o que corresponde a 30% dos atributos disponíveis. MAXNESTING_method, FANOUT_method e ATFD_type foram as métricas de maior importância para detecção de LM em projetos Java.

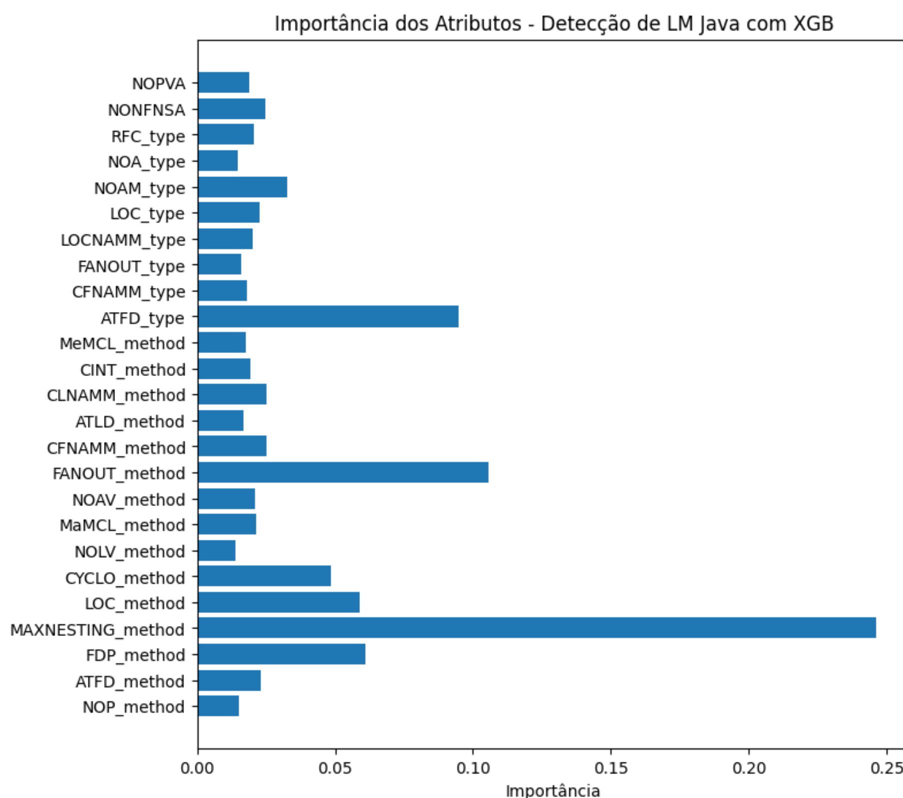


Figura 41 – Importância dos Atributos para Detecção de LM Java

A Figura 42 mostra a importância dos atributos para avaliação de gravidade de LM em projetos Java. Os atributos foram selecionados com a utilização da técnica Qui-Quadrado, 25% dos atributos disponíveis. As métricas que mais se destacaram para avaliação de gravidade LM em projetos Java foram LOC_method, WMCNAMM_type,

CYCLO_method e ATFD_method.

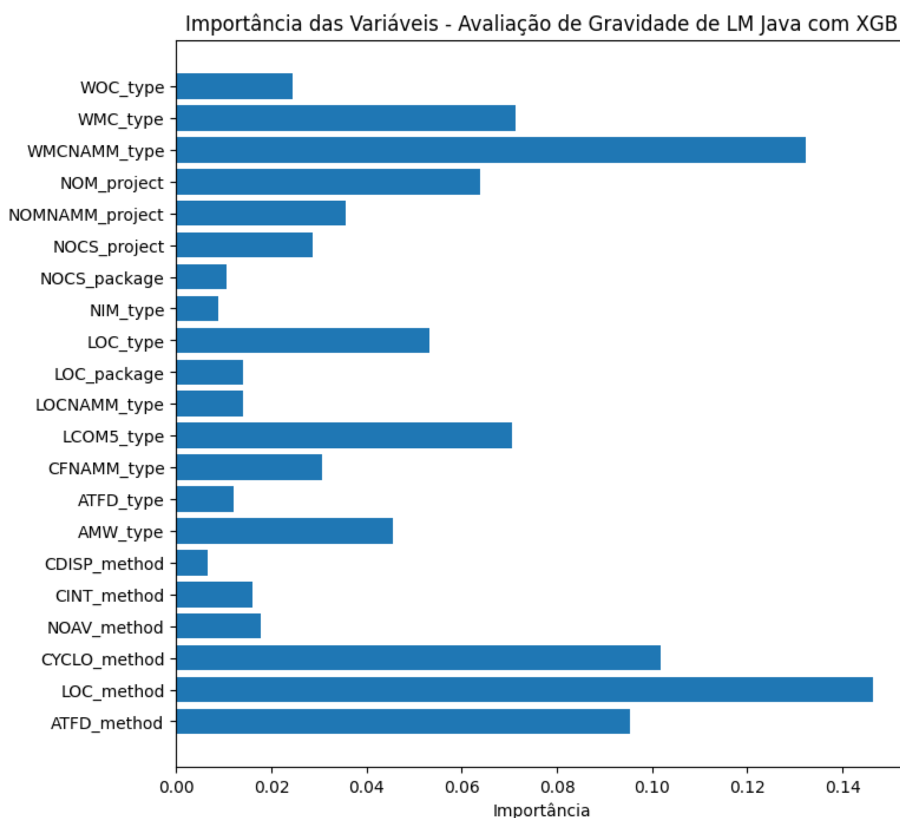


Figura 42 – Importância dos Atributos para Avaliação das Gravidades LM Java

K.2 SHAP

A Figura 43 mostra o gráfico resumo (*summary plot*) do SHAP com o impacto dos atributos na saída do modelo XGB na detecção de LM. MAXNESTING_method, LOC_method, FANOUT_method e FDP_method foram os atributos que mais impactaram na saída do modelo de detecção. Sendo que para esses atributos os valores menores (azuis) exercem influência negativa na saída do modelo, enquanto os valores maiores (vermelhos) influenciam positivamente.

K.3 LIME

Para a avaliação da gravidade LM em projetos Java foi utilizado um modelo baseado em XGB sem escalonamento dos dados. A Figura 44 apresenta o gráfico LIME para uma instância de LM não Grave que alcançou 82% de probabilidade de ser classificada corretamente pelo modelo de avaliação de gravidade. Para esta instância se destacam os valores dos atributos $NOM_project > 4624.00$ e $0.00 < CINT_method \leq 10.26$, que influenciam positivamente na probabilidade de predição. Enquanto os seguintes valores dos

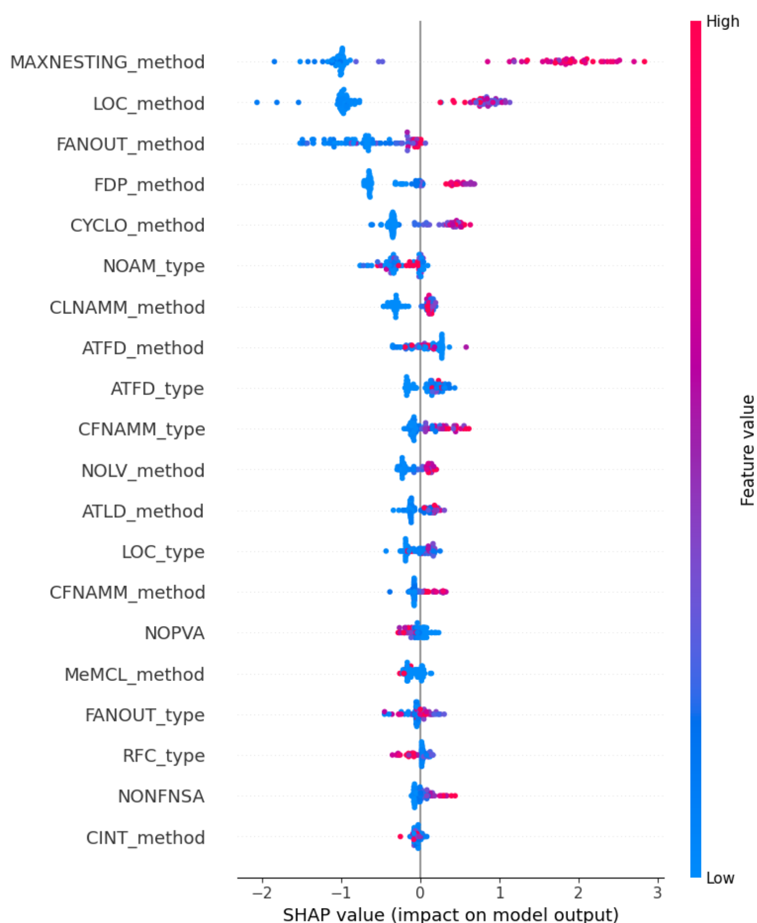


Figura 43 – Impacto na saída do modelo de detecção de LM Java - SHAP

atributos: $5.00 < NOAV_method \leq 20.89$ e $WMCNAMM_type \leq 21.00$ influenciam negativamente neste processo de predição.

A (Figura 45) mostra uma probabilidade de predição de 100% para a avaliação da gravidade LM. Sendo os valores de atributos, como: $CYCLO_method > 16.17$, $NOAV_method > 20.89$, $9.52 < CFNAMM_type \leq 28.00$, $28.00 < LOC_method \leq 104.00$, $LOCNAMM_type > 148.26$, $NIM_type \leq 2.20$ e $WOC_type \leq 0.26$, são importantes para a probabilidade máxima de predição. É importante notar que mesmo com alguns valores de atributos – $0.00 < CINT_method \leq 10.26$ e $LCOM5_type > 0.94$ – contribuindo negativamente, eles não foram suficientes para diminuir o índice de predição desta instância de gravidade LM.

No caso de instância de gravidade LM Grave, detalhada na Figura 46, os valores de $AMW_type \leq 1.47$, $WMCNAMM_type \leq 6.00$, $CFNAMM_type \leq 1.98$ e $LCOM5_type > 0.93$ são características que aumentam a probabilidade de acerto. Neste caso específico, a probabilidade de predição é de 84%. Essa probabilidade de predição foi afetada negativamente pelos valores dos seguintes atributos: $NOCS_package > 48.00$, $LOC_package > 7478.00$ e $0.11 < WOC_type \leq 0.40$.

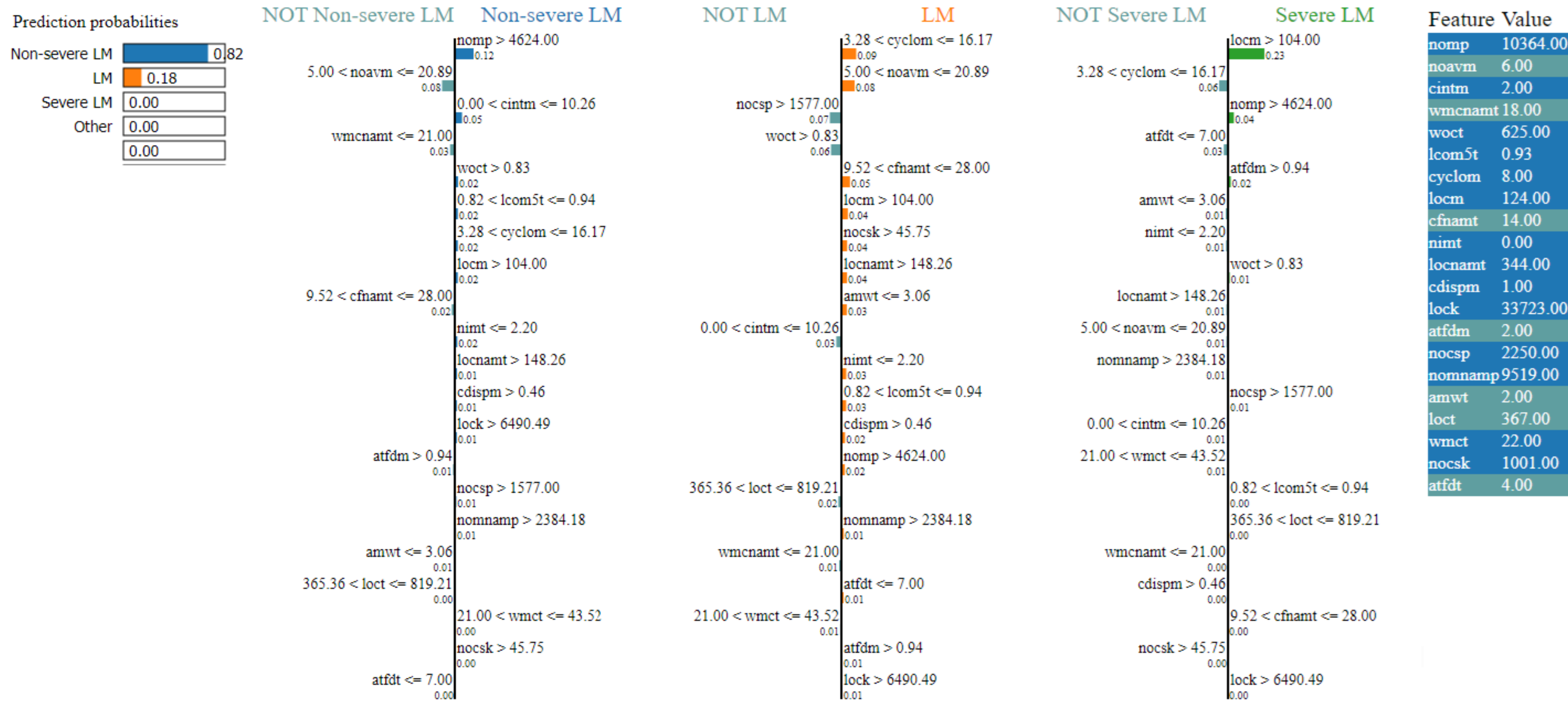


Figura 44 – Detalhamento de instância de LM não Grave Java com LIME

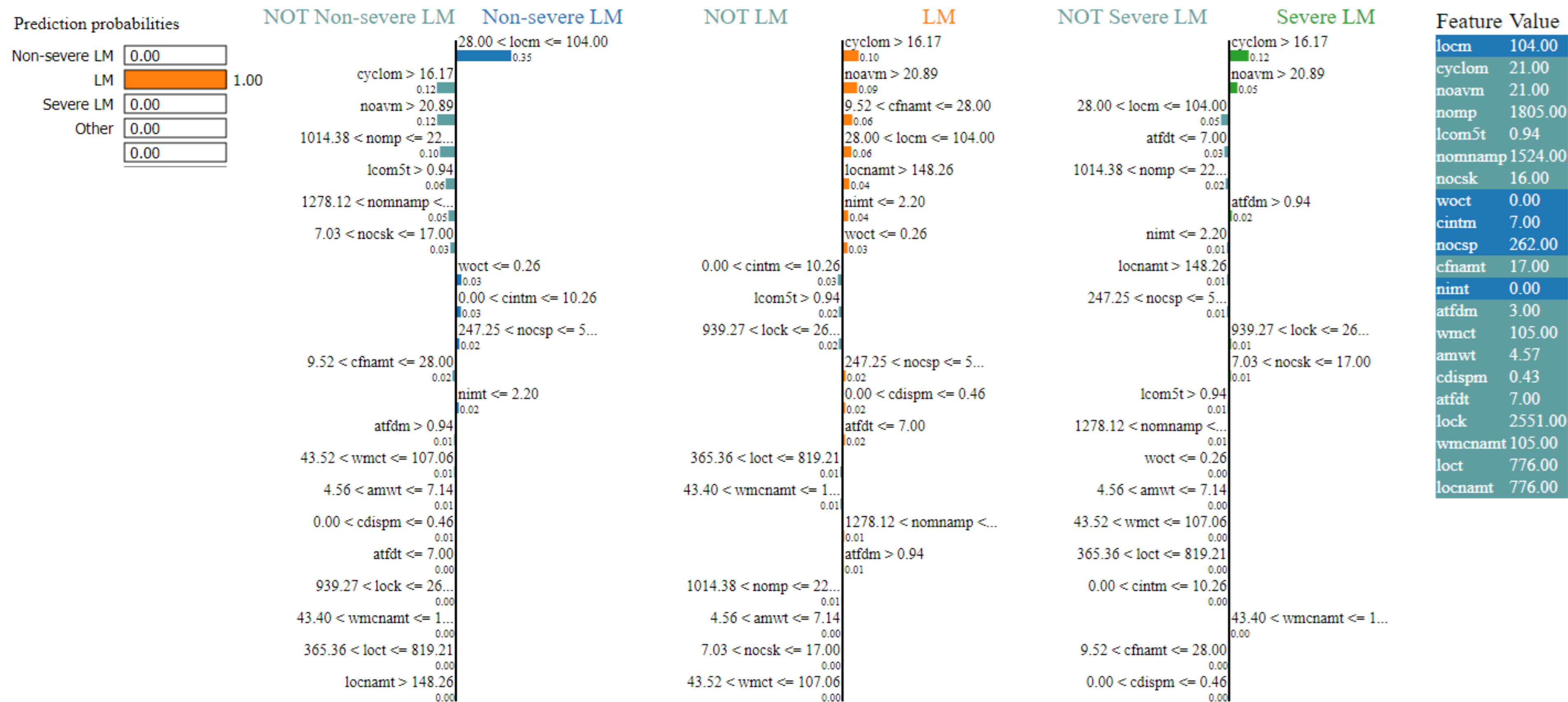


Figura 45 – Detalhamento de instância de Gravidade LM Java com LIME

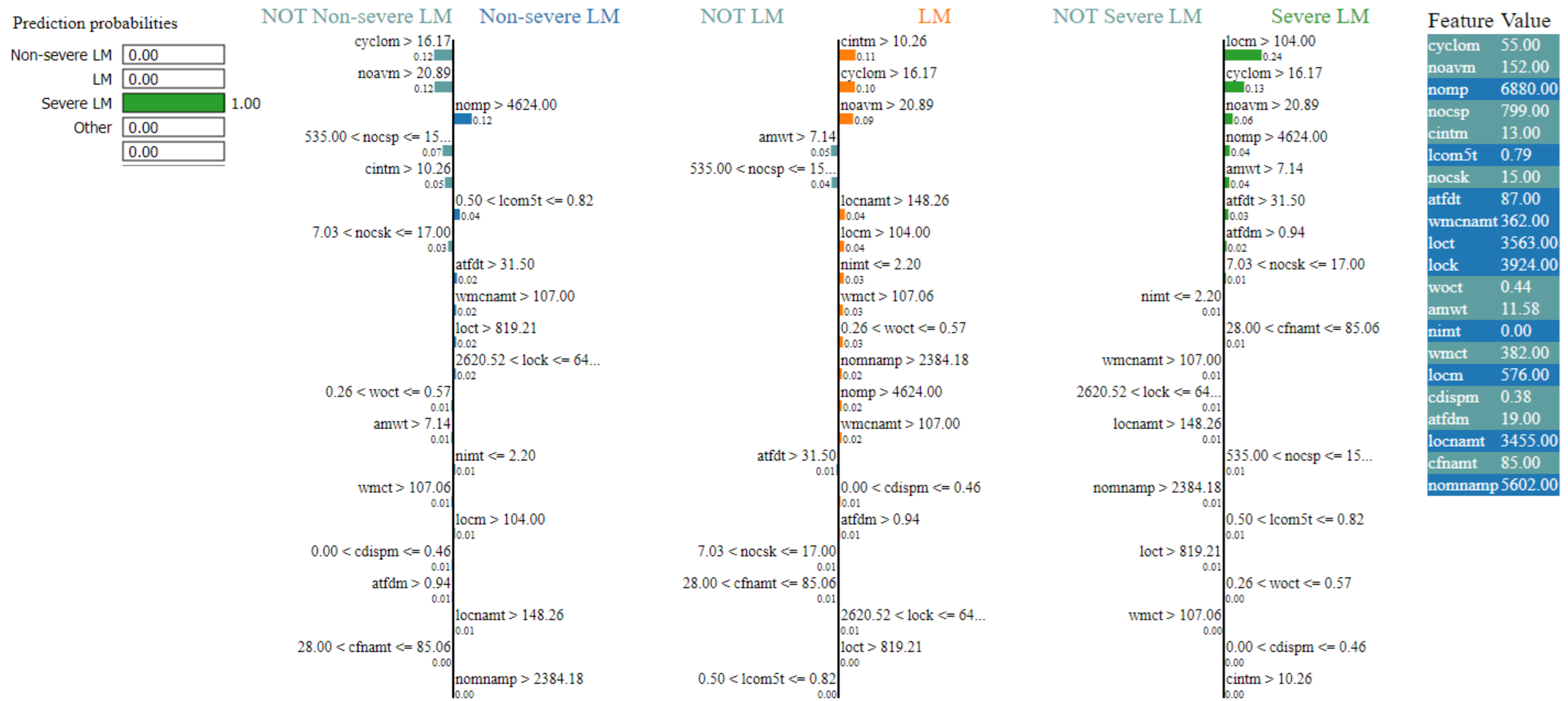


Figura 46 – Detalhamento de instância de LM Grave Java com LIME

APÊNDICE L – GRÁFICOS DE EXPLICABILIDADE - DC JAVA

L.1 Importância dos Atributos

A Figura 47 mostra a importância dos atributos para detecção de DC em projetos Java. 25 atributos foram selecionados com a utilização da técnica Qui-Quadrado, o que corresponde a 30% dos 84 atributos disponíveis. Para calcular a importância dos atributos foi utilizado o valor padrão do parâmetro *importance_type* para o XGB, "*weight*", medindo a importância pelo número de vezes que um atributo aparece nos nós das árvores de decisão. As métricas que mais se destacaram para detecção de DC em projetos Java foram LOC_method, LOC_type e NOCS_project.

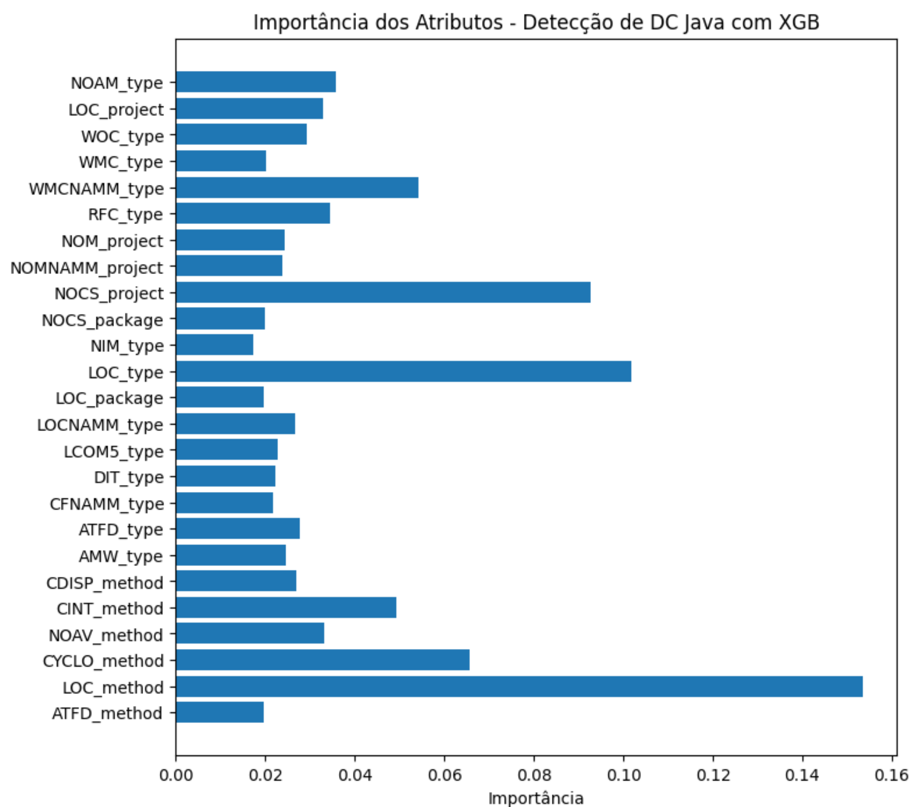


Figura 47 – Importância dos Atributos para Detecção de DC Java

A Figura 48 mostra a importância dos atributos para avaliação de gravidade de DC em projetos Java. Os atributos foram selecionados com a utilização da técnica Qui-Quadrado, 25% dos atributos disponíveis. O Calculo da importância de atributos, com base em um classificador RF, foram realizadas computando a redução total de impureza do critério Gini ao longo das árvores do modelo. As métricas que mais se destacaram para avaliação de gravidade DC em projetos Java foram WMC_type, LOC_method,

WMCNAMM_type, LOC_type e CYCLO_method.

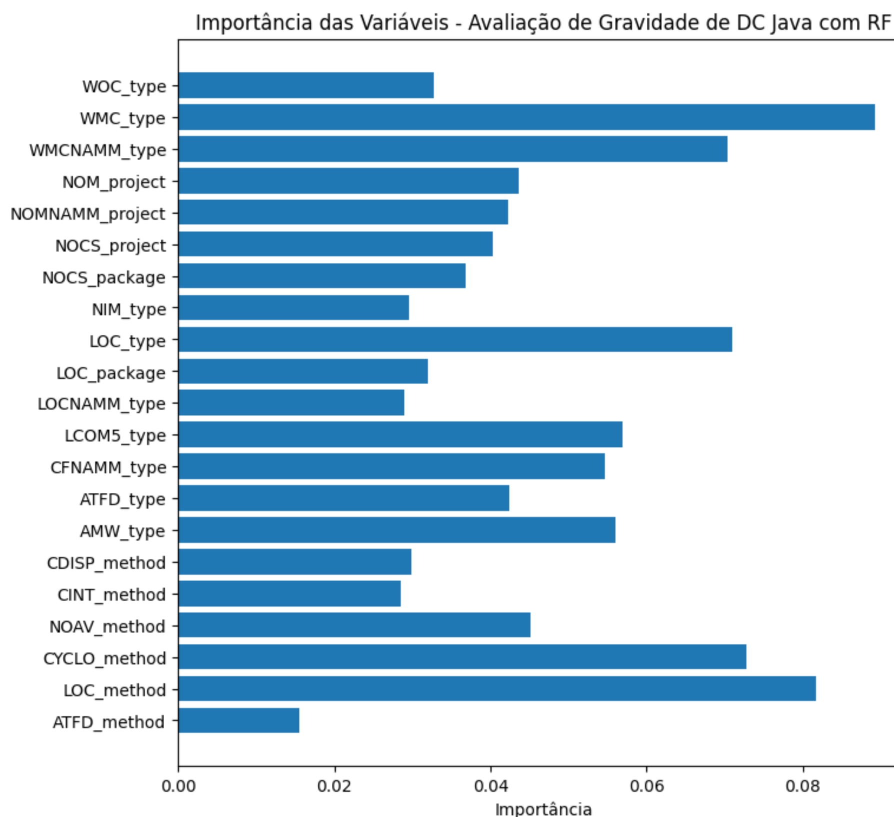


Figura 48 – Importância dos Atributos para Avaliação das Gravidades DC Java

L.2 SHAP

A Figura 49 mostra o gráfico resumo (*summary plot*) do SHAP com o impacto dos atributos na saída do modelo XGB na detecção de DC. NOCS_project, LOC_method, NOAM_project e LCOM5_type foram os atributos que mais impactaram na saída do modelo de detecção. Para os valores de NOCS_project e LCOM5_type os valores mais baixos (azuis) impactam negativamente, enquanto os valores mais altos (vermelhos) impactam positivamente. Para LOC_method e NOAM_type, a maioria dos valores mais baixos impactam positivamente, e a maioria dos valores mais altos, impactam negativamente.

L.3 LIME

Para a avaliação das gravidades DC em projetos Java foi utilizado um modelo baseado em RF sem escalonamento dos dados. A Figura 50 apresenta o gráfico LIME para uma instância de DC não Grave que possui, neste caso, 99% de probabilidade de ser classificada corretamente pelo modelo de avaliação de gravidade. Esta instância possui $WMC_type \leq 12.70$, $LOC_type \leq 107.00$, $WMCNAMM_type \leq 6.00$, $AMW_type \leq$

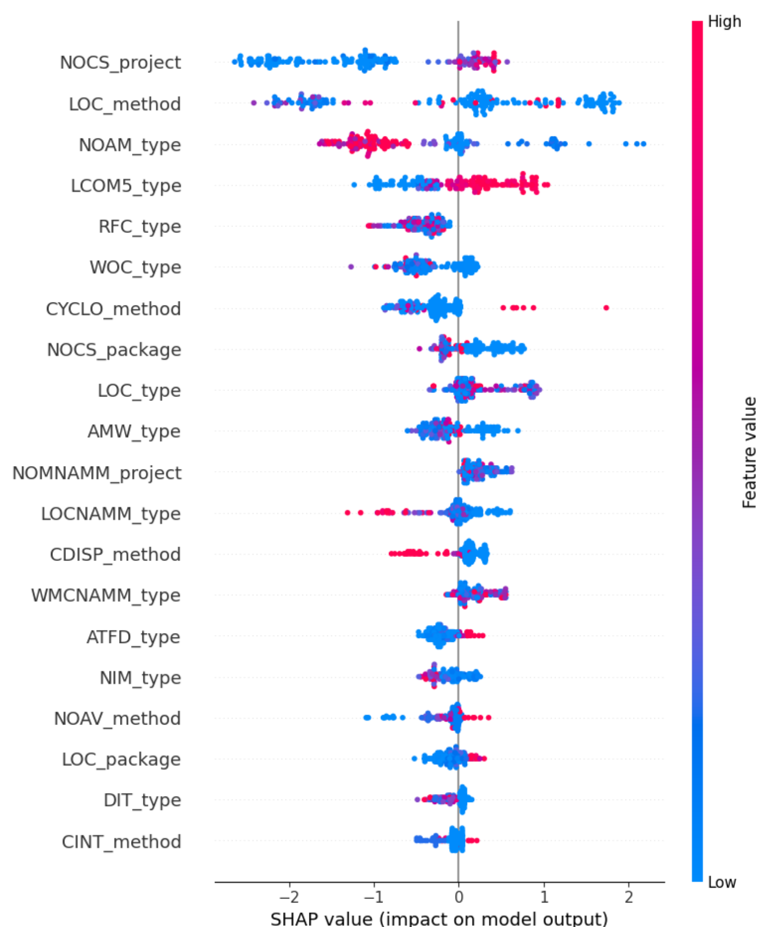


Figura 49 – Impacto na saída do modelo de detecção de DC Java - SHAP

1.47, $LOC_method \leq 3.00$, $CYCLO_me$- $thod \leq 1.00$, $CFNAMM_type \leq 1.98$ e $NOAV_method \leq 1.00$, respectivamente, que contribuem para alcançar este alto índice de probabilidade.

Para a instância de gravidade DC (Figura 51), a probabilidade de acerto é de 87%, com a contribuição de valores de atributos, como: $WMCNAMM_type \leq 6.00$, $AMW_type \leq 1.47$, $CFNAMM_type \leq 1.98$, $CYCLO_method \leq 1.00$, $LOC_method \leq 3.00$, $ATFD_type \leq 1.00$, $NOAV_method \leq 1.00$ e $LCOM5_type > 0.85$. Enquanto, os valores de $LOC_package \leq 1006.71$ e $WMC_type \leq 12.70$ são fatores que diminuem a probabilidade de acerto.

No caso de instância de gravidade DC Grave, detalhada na Figura 52, a probabilidade de predição é de 84%. Para tanto, os valores de $AMW_type \leq 1.47$, $WMCNAMM_type \leq 6.00$, $CFNAMM_type \leq 1.98$, $LCOM5_type > 0.93$, $CYCLO_method \leq 1.00$, $NOM_project > 5064.00$, $NIM_type \leq 0.25$, $ATFD_type \leq 1.00$ e $LOC_method \leq 3.00$ são características que aumentam a probabilidade de acerto.

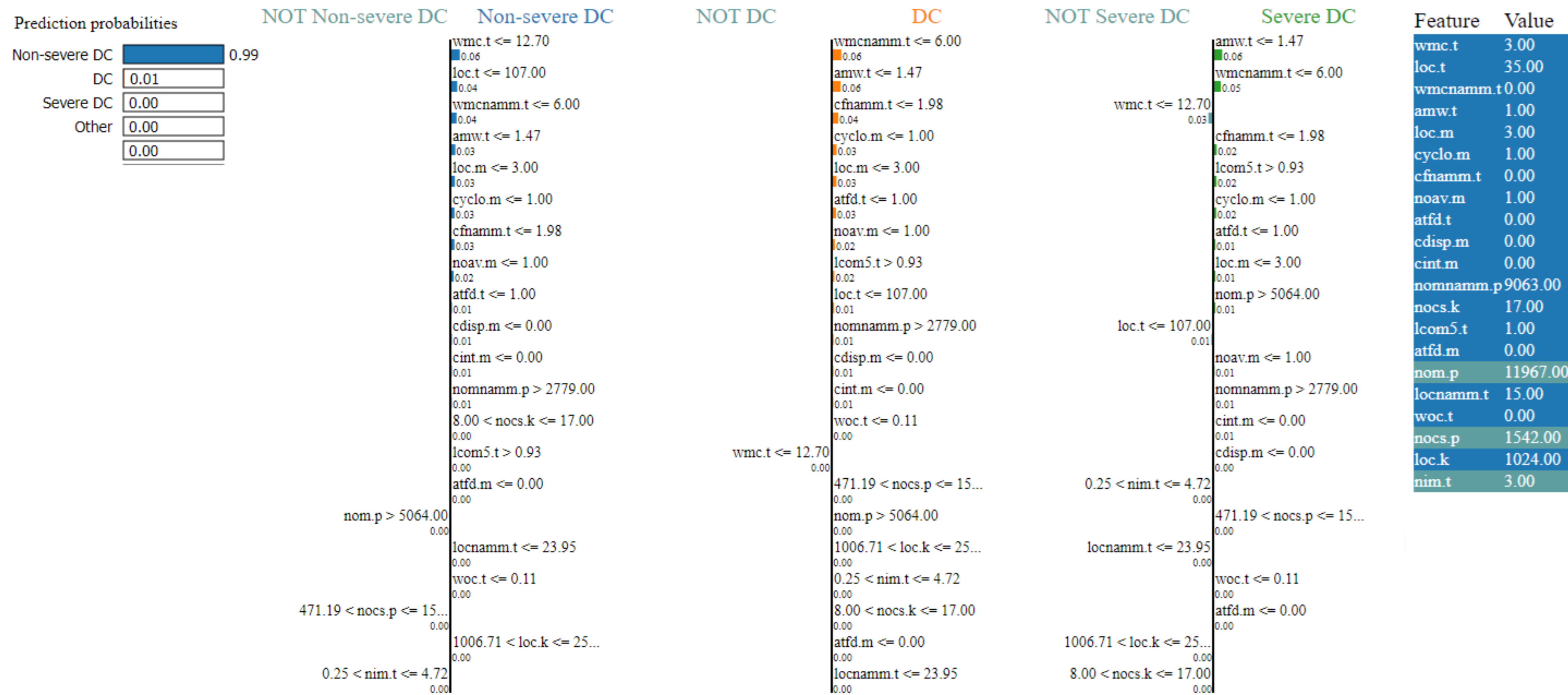


Figura 50 – Detalhamento de instância de DC não Grave Java com LIME

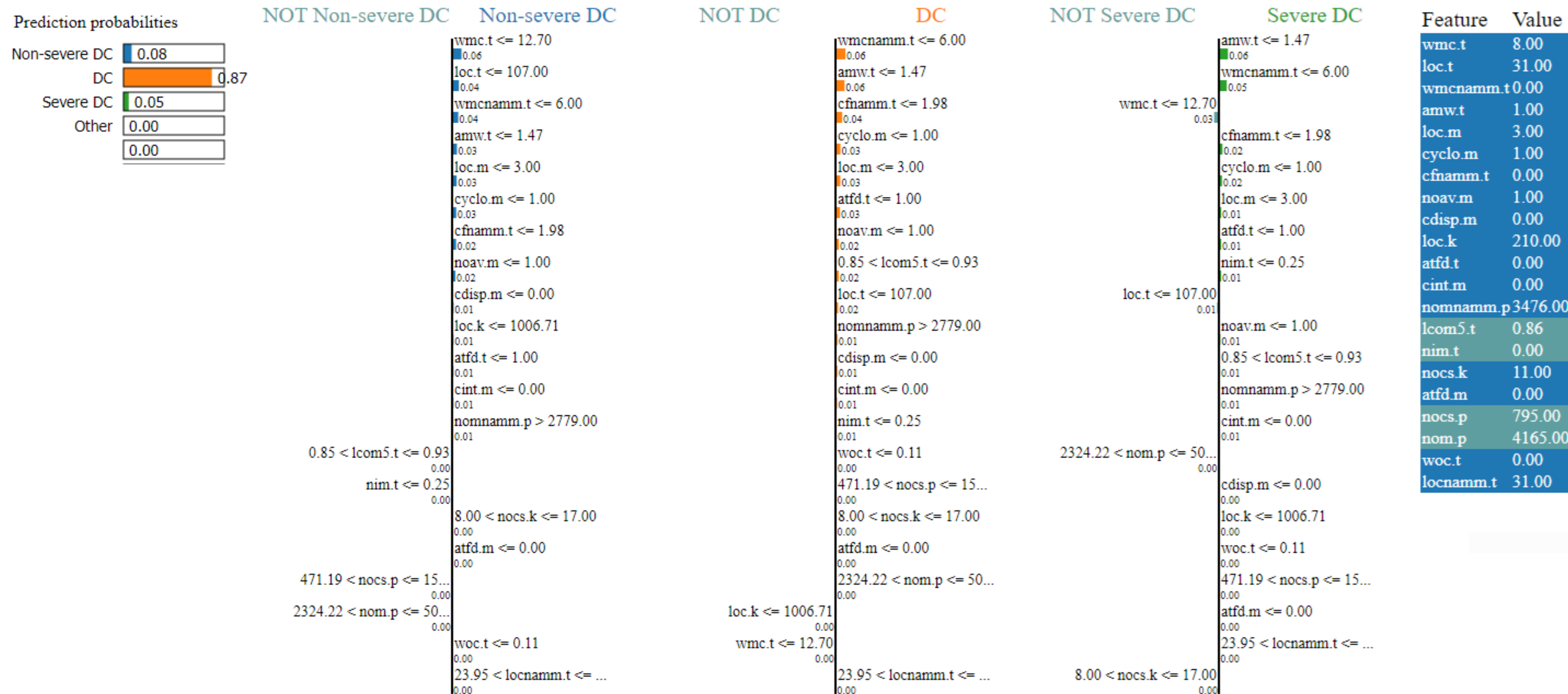


Figura 51 – Detalhamento de instância de Gravidade DC Java com LIME

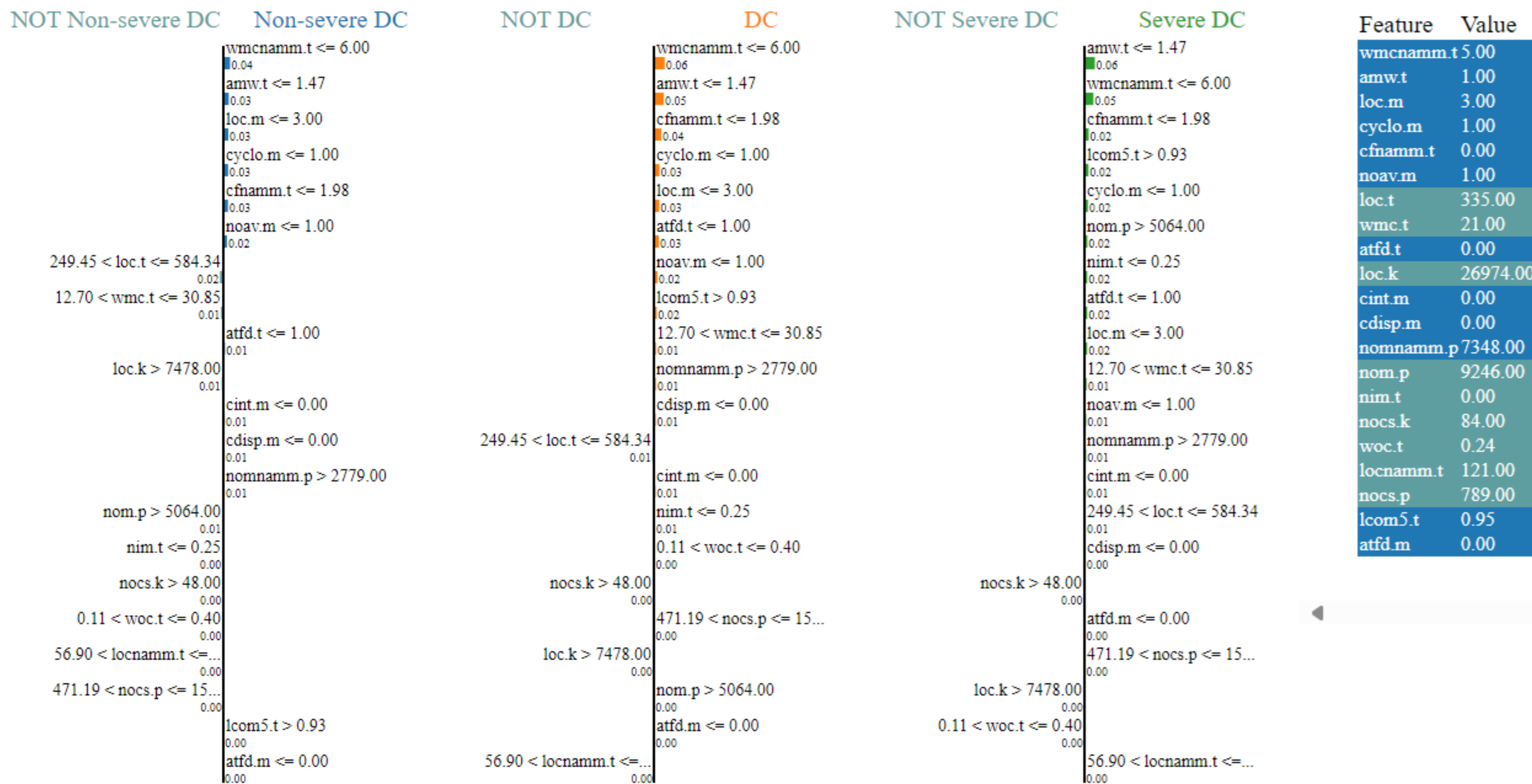


Figura 52 – Detalhamento de instância de DC Grave Java com LIME

APÊNDICE M – GRÁFICOS DE EXPLICABILIDADE - GC JAVA

M.1 Importância dos Atributos

Para calcular a importância dos atributos tanto para detecção de GC em projetos Java quanto para a avaliação da gravidade foi utilizado o *LossFunctionChange* para o método *get_feature_importance* de CB. Para cada atributo, o valor representa a diferença entre o valor de perda do modelo com ou sem esse atributo.

A Figura 53 mostra a importância dos atributos para detecção de GC em projetos Java. O subconjunto dos atributos selecionados corresponde a 20% dos atributos disponíveis. Esses atributos foram selecionados com a utilização da técnica Qui-Quadrado. As métricas que mais tiveram importância para detecção de GC em projetos Java foram LOC_method, CYCLO_method, CDISP_method e WMCNAMM_type.

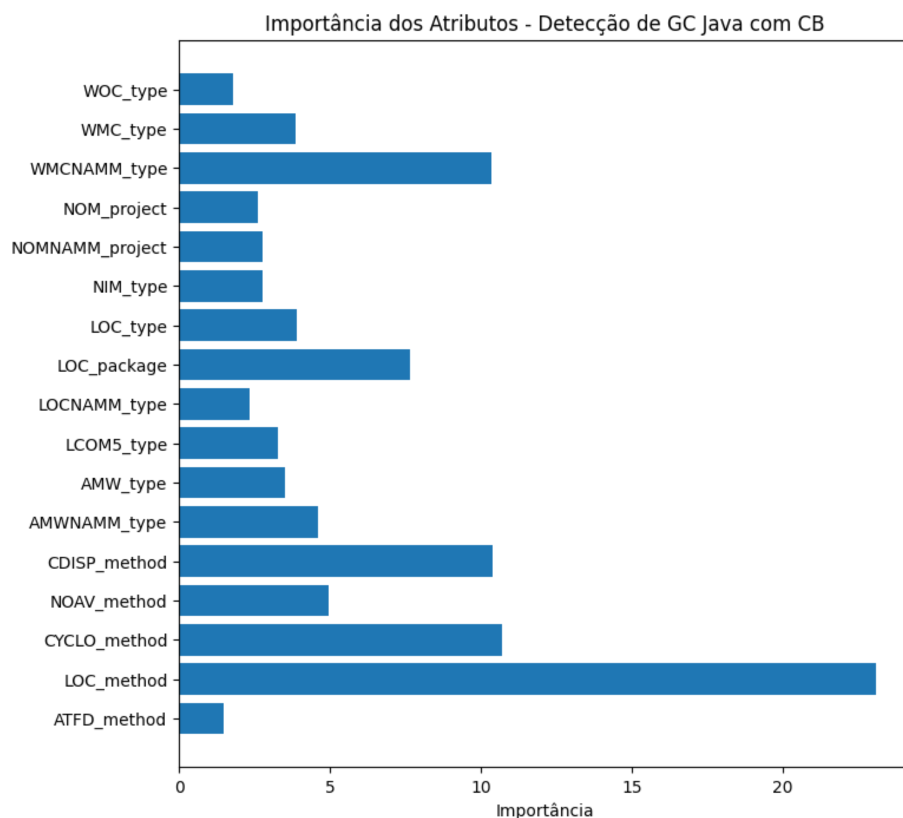


Figura 53 – Importância dos Atributos para Detecção de GC Java

A Figura 54 mostra a importância dos atributos para avaliação de gravidade de GC em projetos Java. Os atributos foram selecionados com a utilização da técnica ANOVA, 30% dos atributos disponíveis. FANOUT_type, MeMCL_method, ATFD_method e

ATFD_type foram as métricas que obtiveram o melhor resultado no calculo de importância na avaliação de gravidade GC em projetos Java.

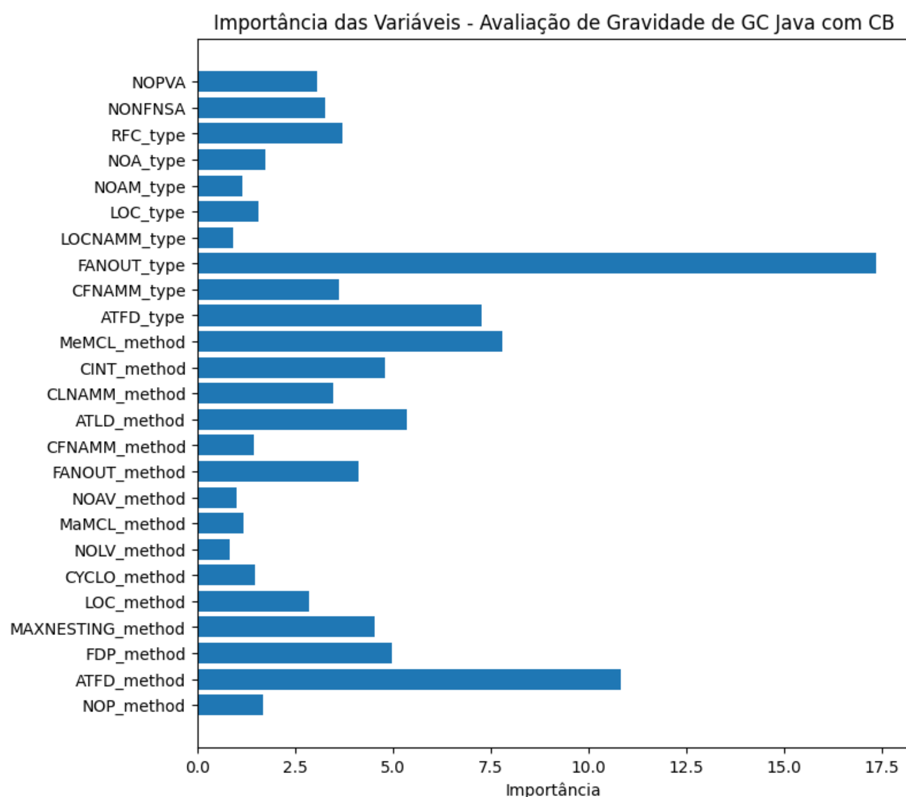


Figura 54 – Importância dos Atributos para Avaliação das Gravidades GC Java

M.2 SHAP

A Figura 55 mostra o gráfico resumo (*summary plot*) do SHAP com o impacto dos atributos na saída do modelo CB na detecção de GC. WMC_type, WMCNAMM_type e LOCNAMM_type foram os atributos que mais impactaram na saída do modelo de detecção. Nesse sentido, os valores de SHAP mais baixos, em sua maioria, para esses atributos, influenciam negativamente a saída dos modelos. No entanto, os valores mais altos, em sua maioria, exercem influência positiva sobre a saída do modelo.

M.3 LIME

Para a avaliação da gravidade DC em projetos Java foi utilizado um modelo baseado no modelo CB sem escalonamento dos dados. A Figura 56 apresenta o gráfico LIME para uma instância de GC não Grave possui, neste caso, 90% de probabilidade de ser classificada corretamente pelo modelo de avaliação de gravidade. Esta instância possui valores de $MAXNESTING_method \leq 3.00$, $ATFD_type \leq 68.50$, $FANOUT_method \leq 0.00$,

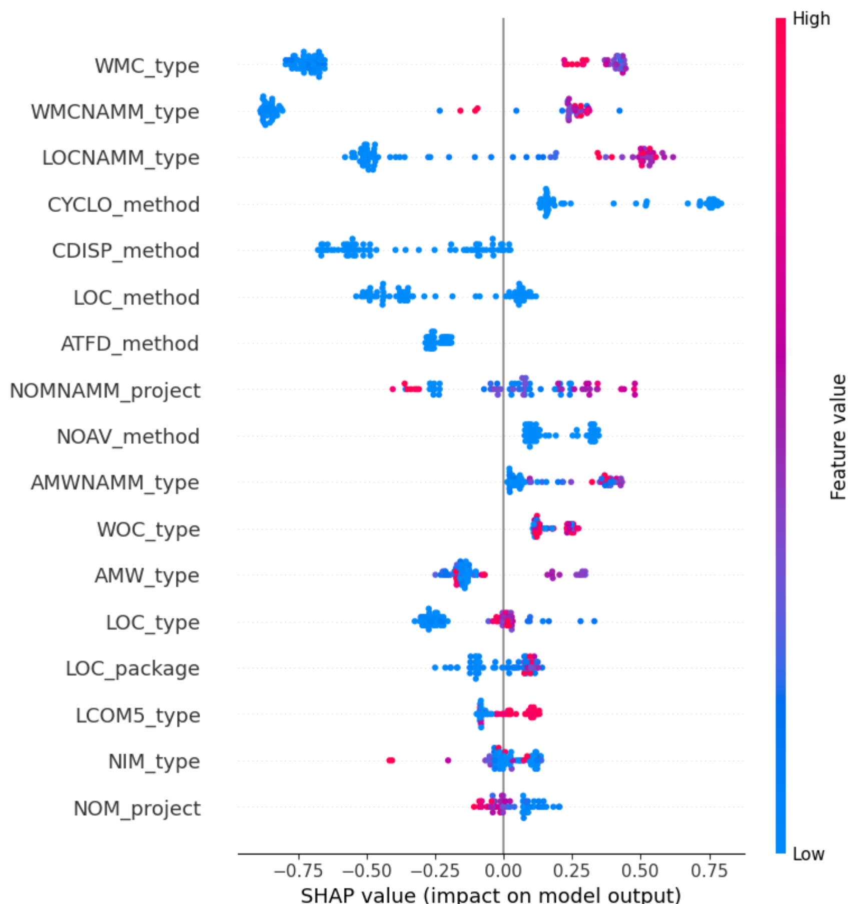
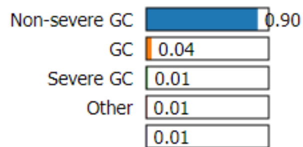


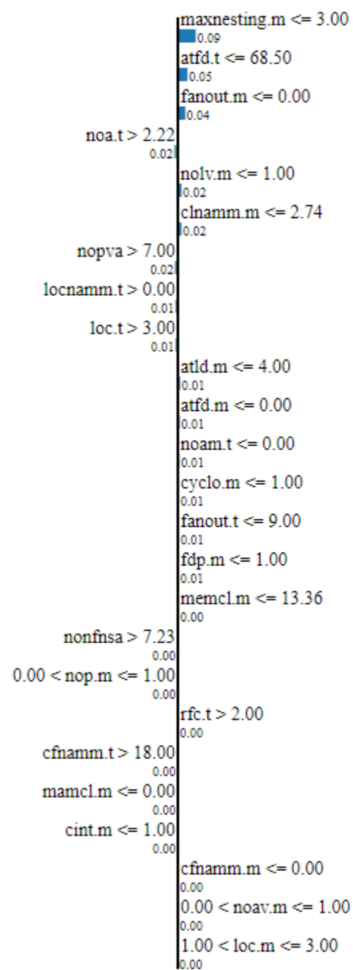
Figura 55 – Impacto na saída do modelo de detecção de GC Java - SHAP

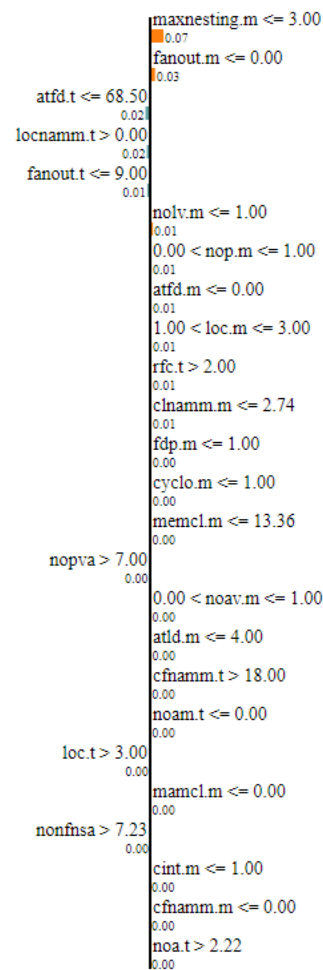
$NOLV_method \leq 1.00$ e $CLNAMM_method \leq 2.74$, que contribuem para alcançar este alto índice de probabilidade.

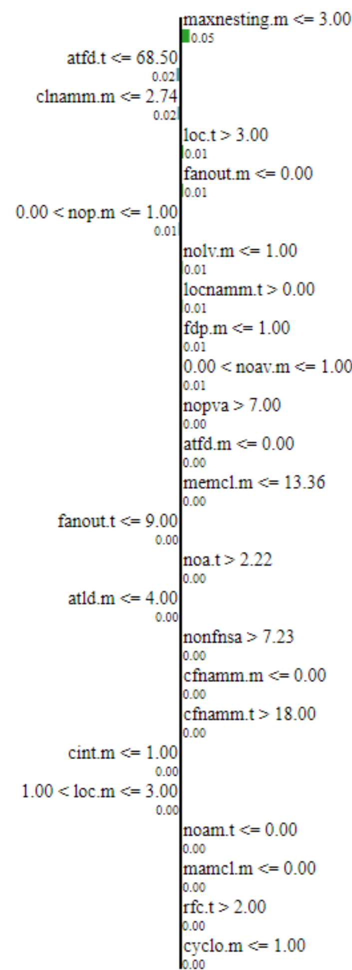
Para a instância de gravidade GC (Figura 57), a probabilidade de acerto é de 99%, com a contribuição de valores de atributos, como: $MAXNESTING_method \leq 3.00$ e $FANOUT_method \leq 0.00$. Enquanto, os valores de $ATFD_type \leq 68.50$ e $LOCNAMM_type > 0.00$ são fatores que diminuem a probabilidade de acerto.

No caso de instância de gravidade GC Grave, detalhada na Figura 58, os valores de $MAXNESTING_method \leq 3.00$ e $LOC_type > 3.00$ são características que aumentam a probabilidade de acerto. Neste caso específico, a probabilidade de predição é de 99%. Essa probabilidade de predição foi afetada negativamente pelos valores de alguns atributos (e.g., $ATFD_type \leq 68.50$ e $CLNAMM_method \leq 2.74$).







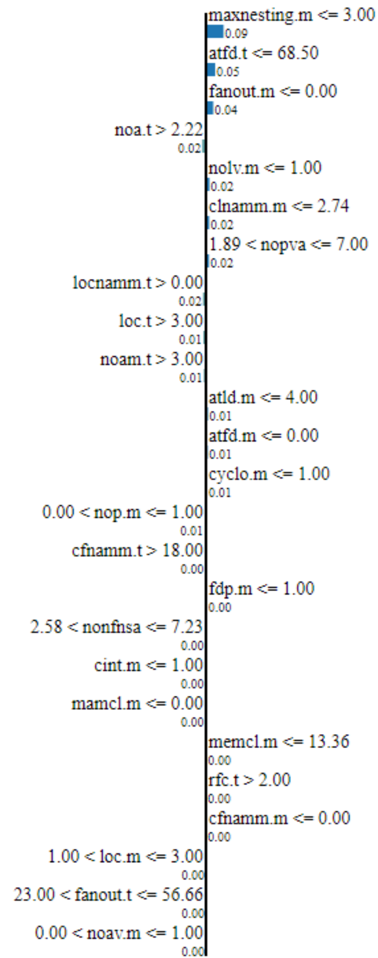


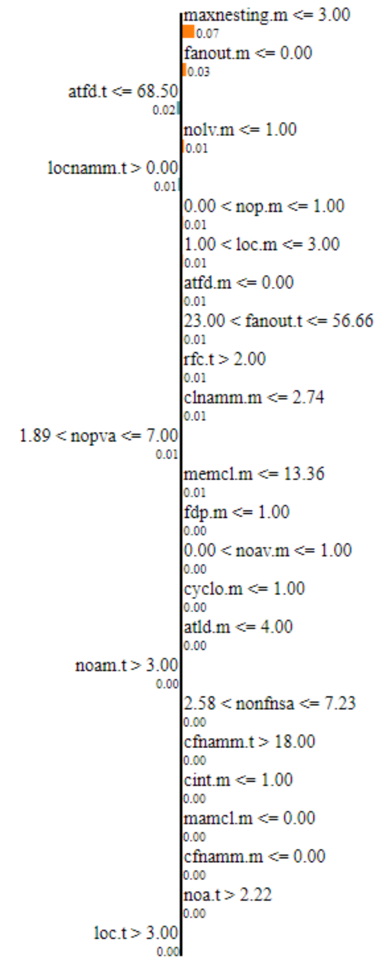
Feature	Value
maxnesting.m	1.00
atfd.t	3.00
fanout.m	0.00
noa.t	10.00
nolv.m	1.00
clnamm.m	0.00
nopva	10.00
locnamm.t	571.00
loc.t	571.00
atld.m	0.00
atfd.m	0.00
noam.t	0.00
cyclo.m	1.00
fanout.t	7.00
fdp.m	0.00
memcl.m	0.00
nonfnsa	10.00
nop.m	1.00
rfc.t	39.00
cfnamm.t	19.00
mamcl.m	0.00
cint.m	0.00
cfnamm.m	0.00
noav.m	1.00
loc.m	3.00

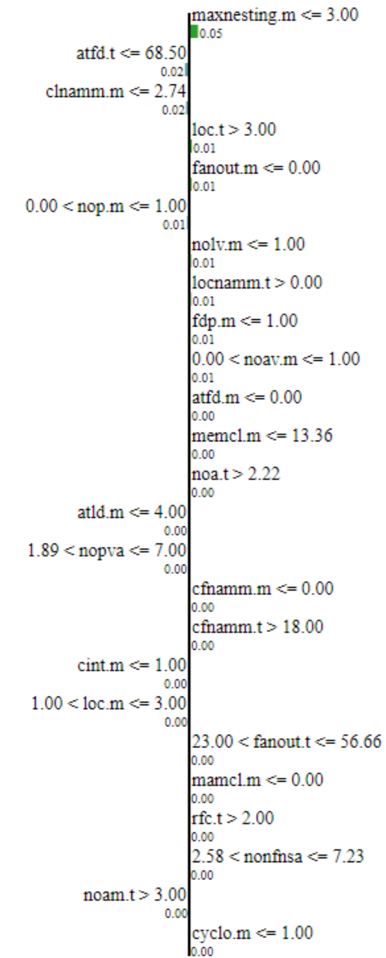
Figura 56 – Detalhamento de instância de GC não Grave Java com LIME

Prediction probabilities

Non-severe GC	0.00
GC	0.99
Severe GC	0.00
Other	0.00
	0.00







Feature	Value
maxnesting.m	1.00
atfd.t	14.00
fanout.m	0.00
noa.t	6.00
nolv.m	1.00
clnamm.m	0.00
nopva	5.00
locnamm.t	997.00
loc.t	1091.00
noam.t	7.00
atld.m	0.00
atfd.m	0.00
cyclo.m	1.00
nop.m	1.00
cfnamm.t	80.00
fdp.m	0.00
nonfnsa	5.00
cint.m	0.00
mamcl.m	0.00
memcl.m	0.00
rfc.t	217.00
cfnamm.m	0.00
loc.m	3.00
fanout.t	30.00
noav.m	1.00

Figura 57 – Detalhamento de instância de Gravidade GC Java com LIME

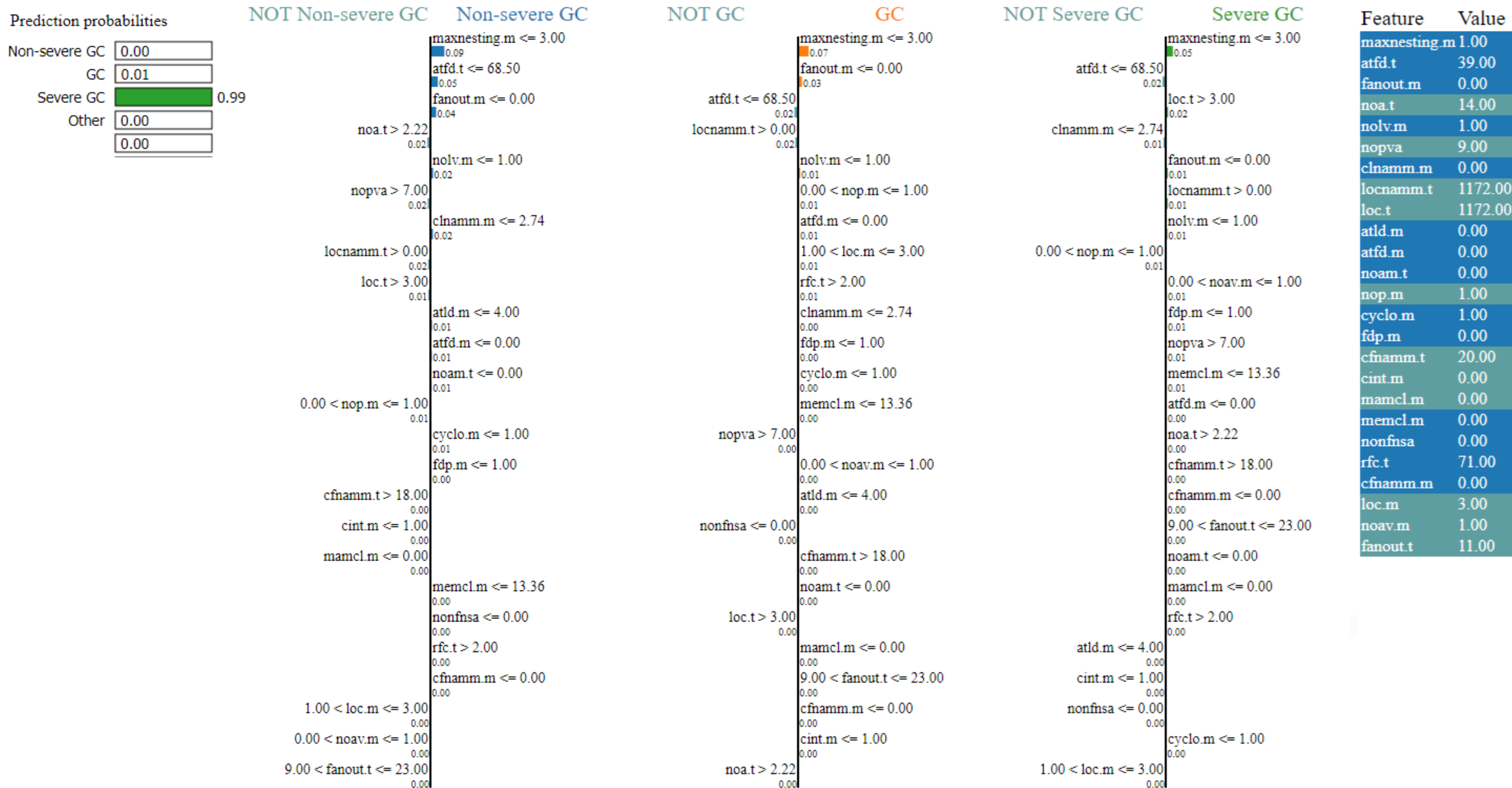


Figura 58 – Detalhamento de instância de GC Grave Java com LIME

APÊNDICE N – GRÁFICOS DE EXPLICABILIDADE - FE C#

N.1 Importância dos Atributos

Os atributos, padronizados, tanto para detecção de FE em projetos C# quanto para avaliação de suas gravides foram selecionados com a utilização da técnica Qui-Quadrado (68), 30% dos atributos disponíveis. A Figura 59 mostra a importância dos atributos para detecção de FE em projetos C#. Para calcular a importância dos atributos foi utilizado o *LossFunctionChange* para o método *get_feature_importance* de CB. Para cada atributo, o valor representa a diferença entre o valor de perda do modelo com esse atributo e sem ele. Os atributos que mais se destacaram para detecção de FE em projetos C# foram AMW_type e WOC_type.

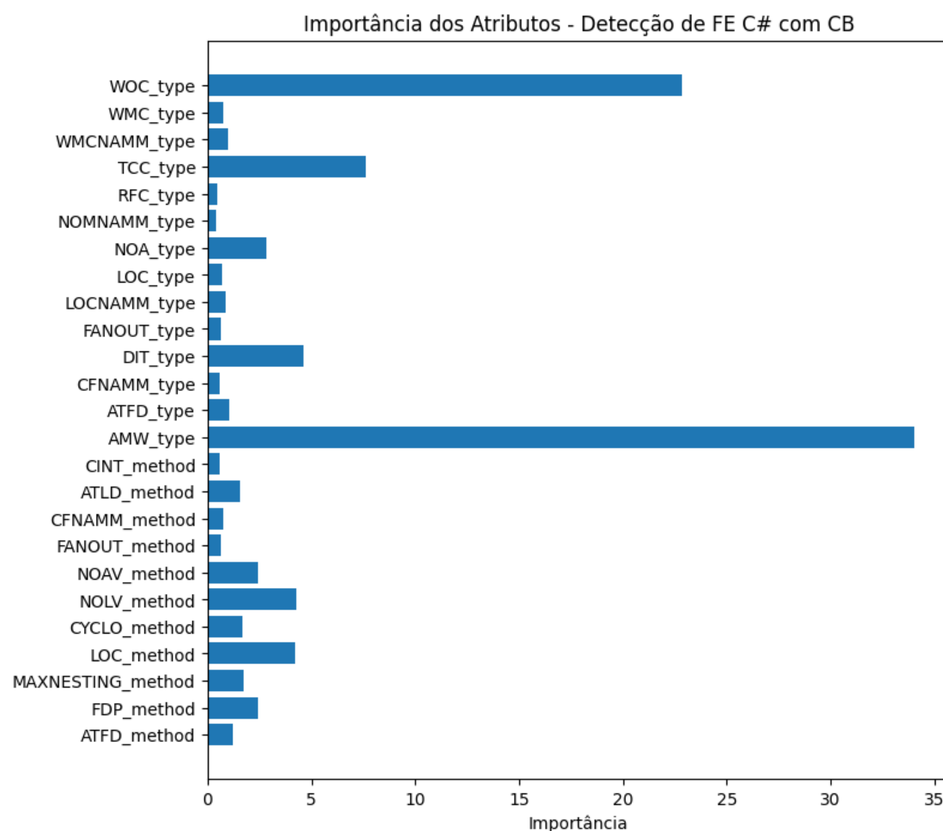


Figura 59 – Importância dos Atributos para Detecção de FE C#

A Figura 60 mostra a importância dos atributos para avaliação de gravidade de FE em projetos C#. Para calcular a importância dos atributos foi utilizado o valor padrão do parâmetro *importance_type* para o XGB, “*weight*”, medindo a importância pelo número de vezes que um atributo aparece nos nós das árvores de decisão. Os atributos mais

importantes para avaliação de gravidade FE em projetos C# foram ATFD_method, CYCLO_method, NOAV_method e ATFD_type.

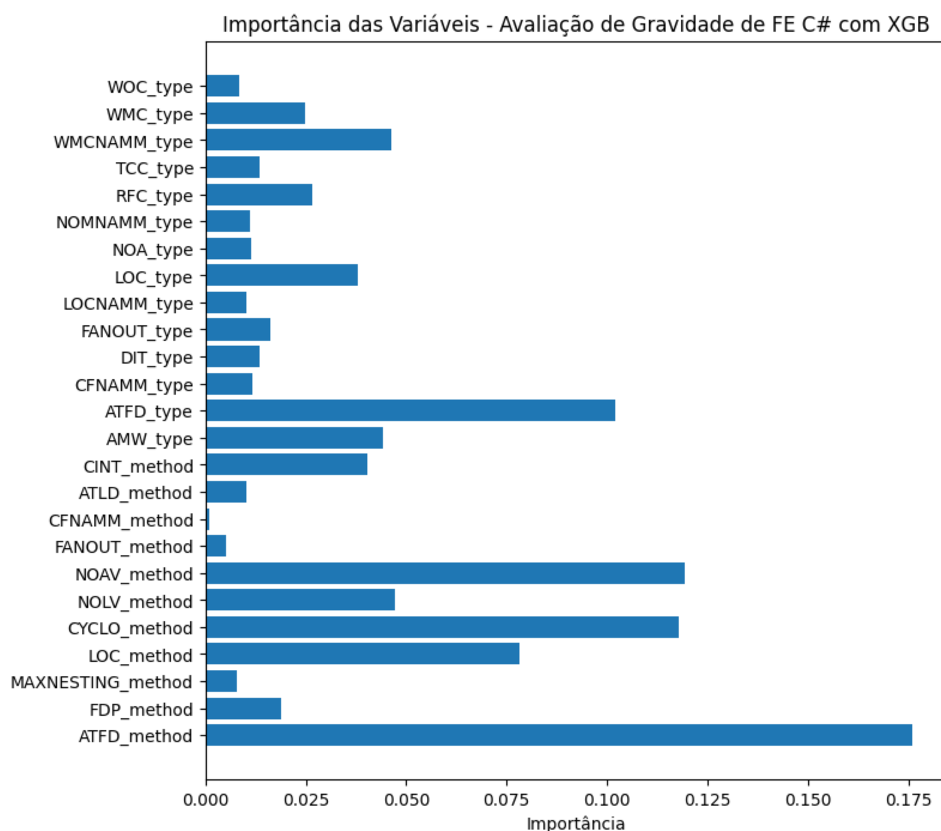


Figura 60 – Importância dos Atributos para Avaliação das Gravidades FE C#

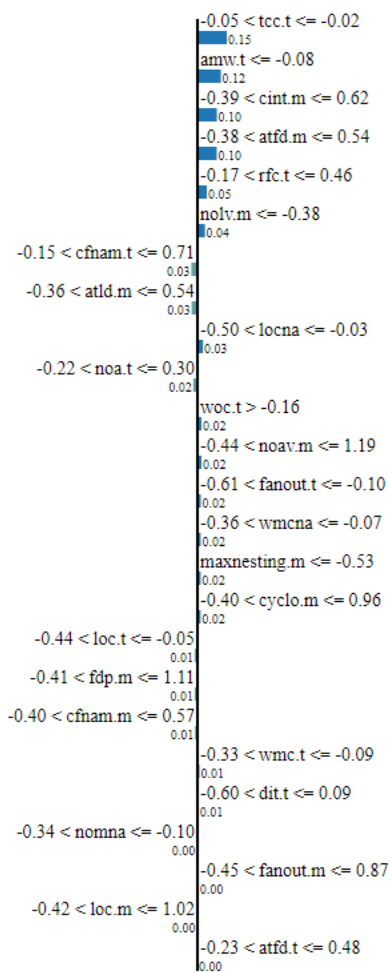
N.2 LIME

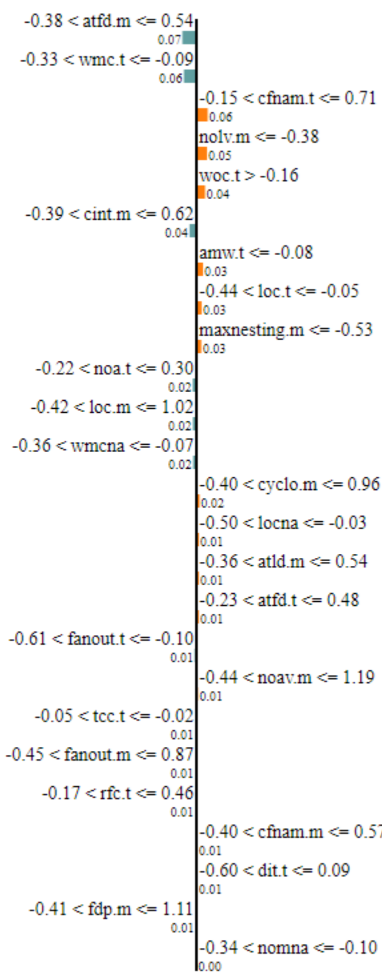
Para a avaliação da gravidade FE em projetos C# foi utilizado um modelo baseado em XGB com padronização dos dados. A Figura 61 apresenta o gráfico LIME para uma instância de FE não Grave, que alcançou 100% de probabilidade de predição pelo modelo de avaliação de gravidade. Esta instância possui alguns valores de atributos importantes para sua alta probabilidade, como: $-0.05 < TCC_type \leq -0.02$, $AMW_type \leq -0.08$, $-0.39 < CINT_method \leq 0.62$, $-0.38 < ATFD_method \leq 0.54$, $-0.17 < RFC_type \leq 0.46$ e $NOLV_method \leq -0.38$.

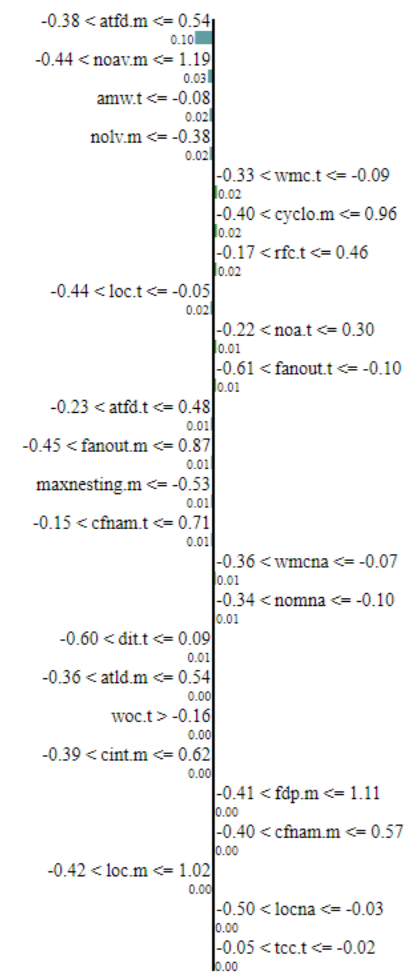
No caso de instância de gravidade FE, veja Figura 62, houve uma probabilidade de acerto de 62%. Sendo os valores de $ATFD_method > 0.54$, $CFNAMM_type > 0.71$, $WOC_type > -0.16$ e $AMW_type \leq -0.08$ exercem influência relevante na probabilidade de acerto do modelo. Contudo, os valores dos atributos seguintes diminuem esta probabilidade: $NOLV_method > 0.88$, $-0.03 < LOCNAMM_type \leq 0.59$, $-0.39 < CINT_method \leq 0.62$ e $NOA_type > 0.30$. Além disso, para essa instância há uma probabilidade considerável, 33%, para predição equivocada como FE não Grave.

Prediction probabilities

Non-severe FE	1.00
FE	0.00
Severe FE	0.00
Other	0.00
	0.00





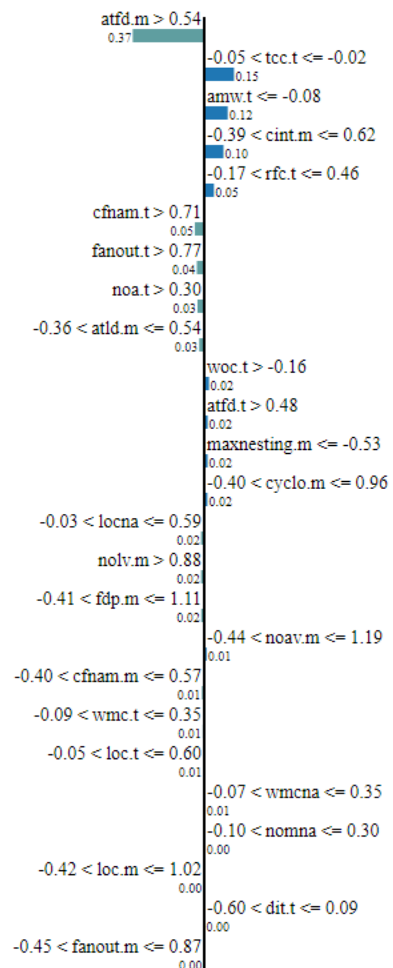


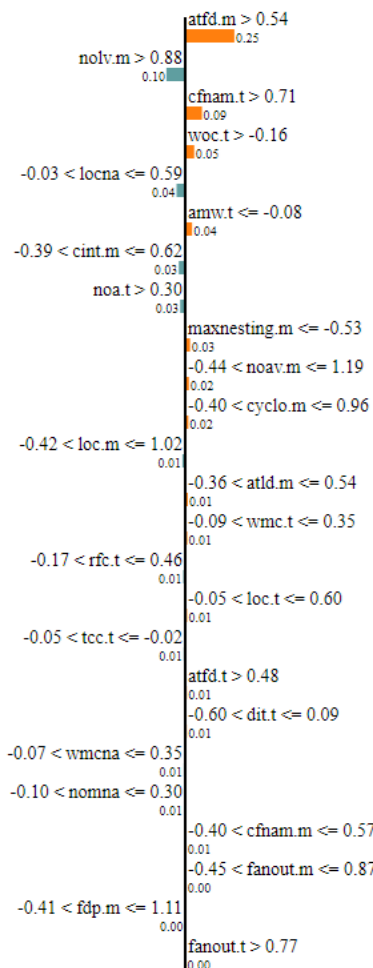
Feature	Value
tcc.t	-0.04
amw.t	-0.13
cint.m	0.23
atfd.m	-0.24
rfc.t	0.12
nolv.m	-0.45
cfnam.t	-0.11
atld.m	-0.27
locna	-0.48
noa.t	-0.14
woc.t	-0.11
noav.m	-0.43
fanout.t	-0.10
wmcna	-0.32
maxnesting.m	-1.11
cyclo.m	-0.37
loc.t	-0.28
fdp.m	0.75
cfnam.m	0.40
wmc.t	-0.32
dit.t	-0.34
nomna	-0.28
fanout.m	0.50
loc.m	-0.29
atfd.t	0.01

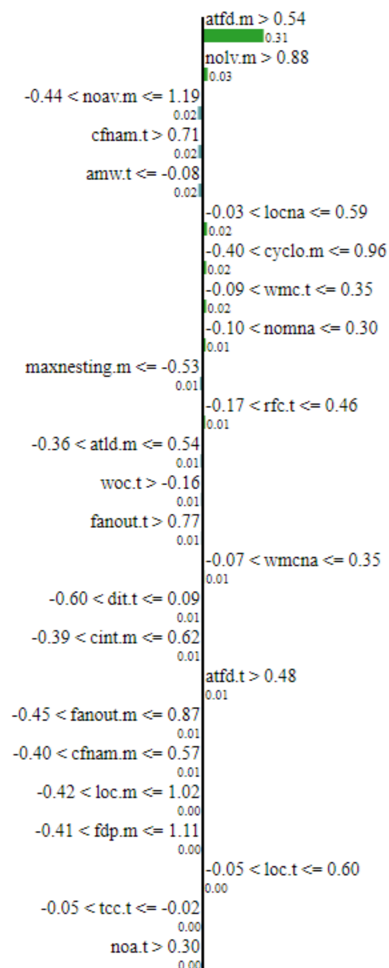
Figura 61 – Detalhamento de instância de FE não Grave C# com LIME

Prediction probabilities

Non-severe FE	0.33
FE	0.62
Severe FE	0.00
Other	0.04
	0.00







Feature	Value
atfd.m	1.13
tcc.t	-0.04
amw.t	-0.13
cint.m	-0.19
rfc.t	0.20
cfnam.t	0.72
fanout.t	0.95
noa.t	0.60
atld.m	0.37
woc.t	-0.12
atfd.t	0.97
maxnesting.m	-1.11
cyclo.m	-0.37
locna	0.48
nolv.m	1.90
fdp.m	0.75
noav.m	0.90
cfnam.m	-0.24
wmc.t	0.23
loc.t	0.59
wmcna	0.23
nomna	-0.07
loc.m	0.74
dit.t	-0.34
fanout.m	-0.08

Figura 62 – Detalhamento de instância de Gravidade FE C# com LIME

APÊNDICE O – GRÁFICOS DE EXPLICABILIDADE - LM C#

O.1 Importância dos Atributos

Os atributos, padronizados, para detecção de LM em projetos C# foram selecionados com a utilização da técnica Qui-Quadrado (68), 30% dos atributos disponíveis. A Figura 63 mostra a importância dos atributos para detecção de LM em projetos C#. Para calcular a importância dos atributos foi utilizado o *LossFunctionChange* para o método *get_feature_importance* de CB. Os atributos que tiveram mais importância para esta detecção foram LOC_method e CYCLO_method.

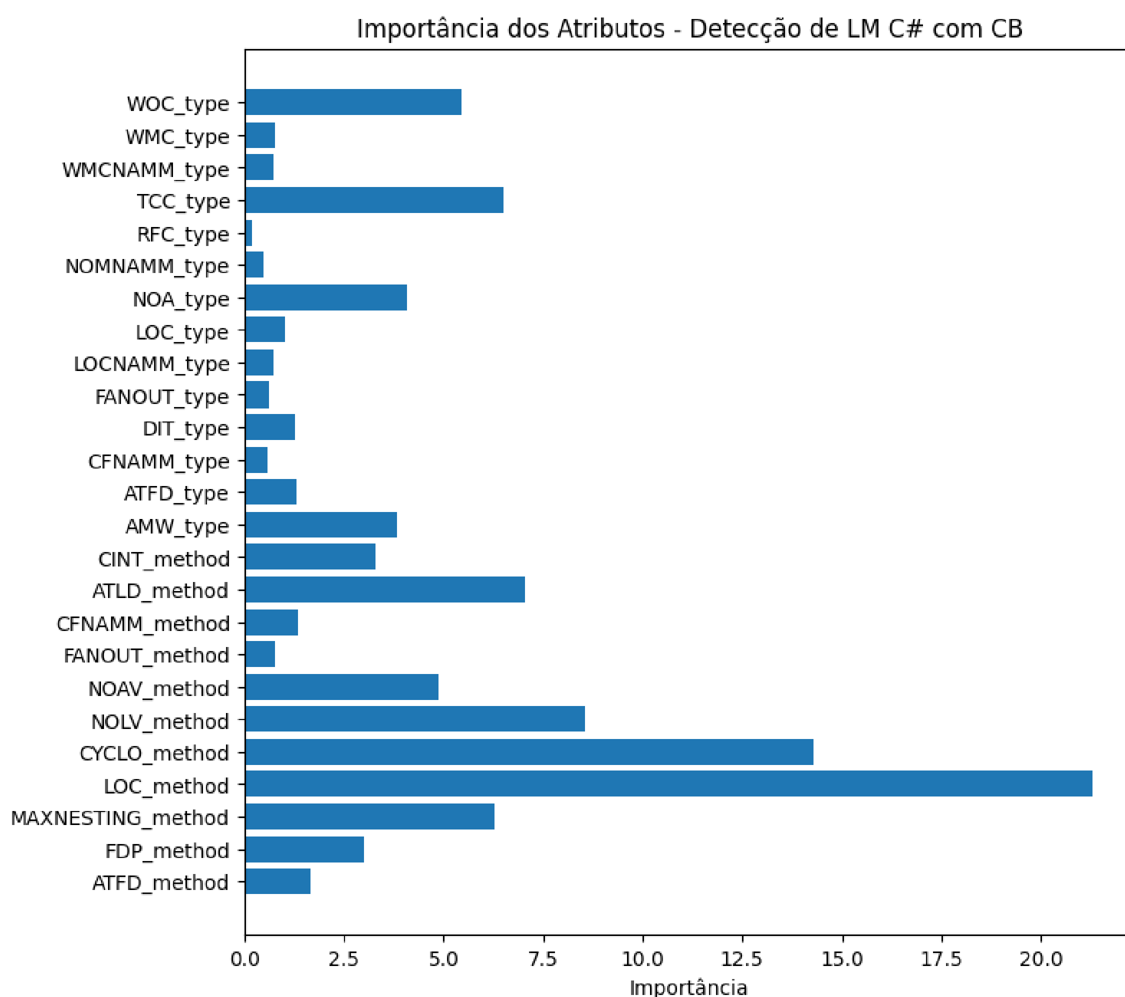


Figura 63 – Importância dos Atributos para Detecção de LM C#

A Figura 64 mostra a importância dos atributos para avaliação de gravidade de LM em projetos C#. Os atributos, 25% do total, foram selecionados com a utilização da técnica ANOVA. Para calcular a importância dos atributos foi utilizado a redução

total de impureza do critério *Log Loss* ao longo das árvores do modelo. CYCLO_method, LOC_method, NOAV_method e RFC_method foram os atributos mais importantes para o modelo de MC de avaliação de gravidade LM em projetos C#.

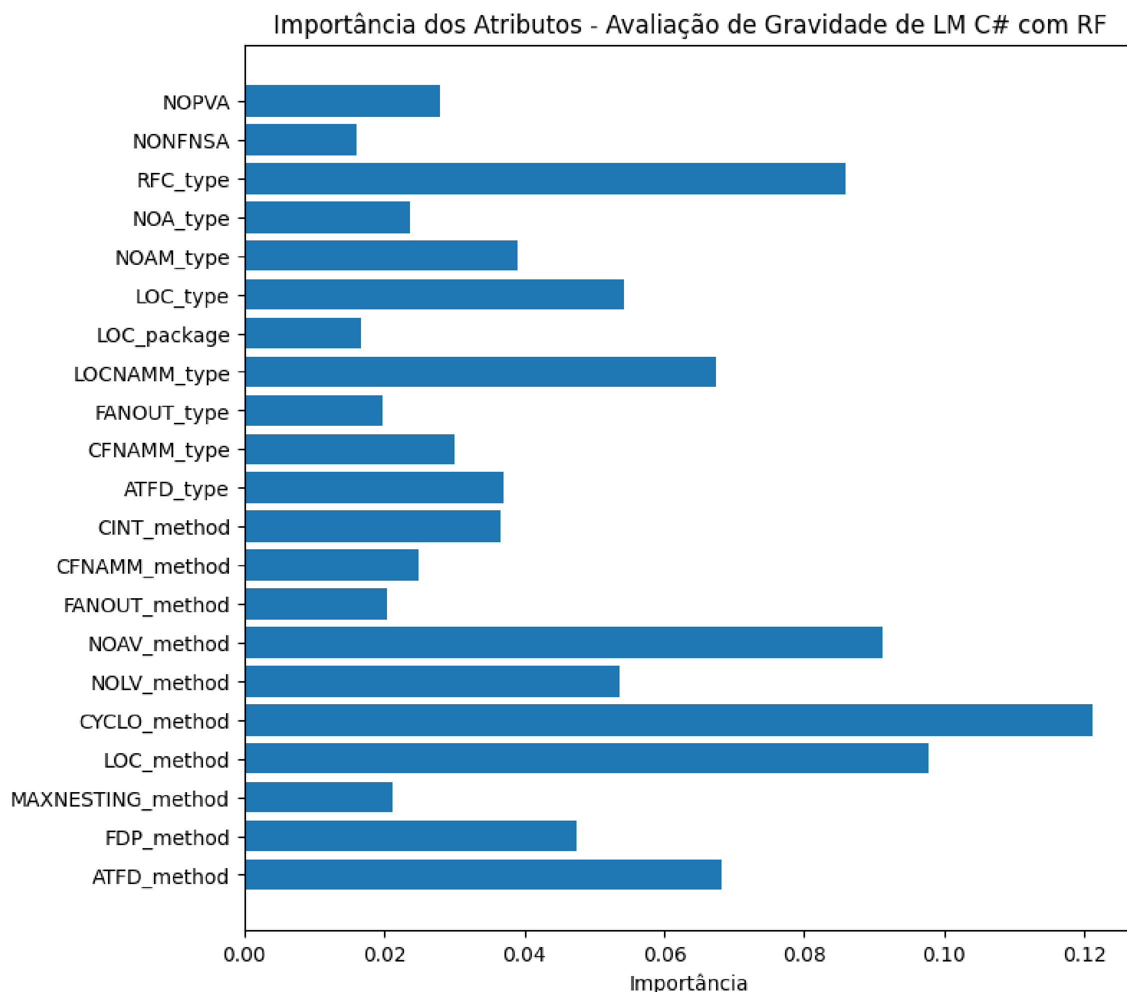


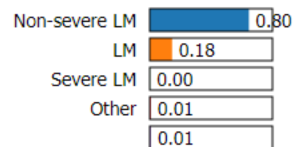
Figura 64 – Importância dos Atributos para Avaliação das Gravidades LM C#

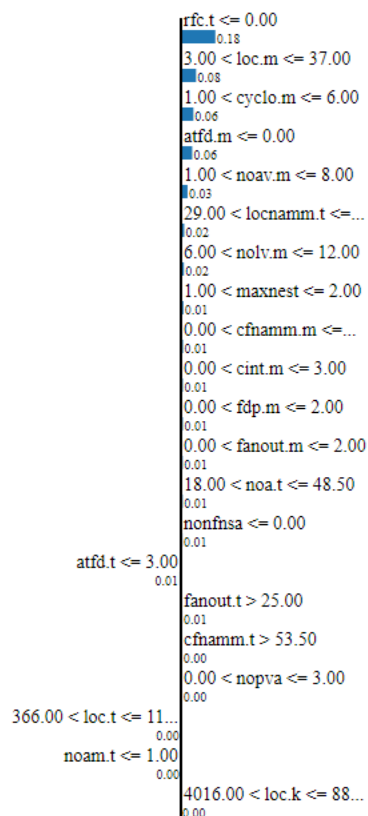
O.2 LIME

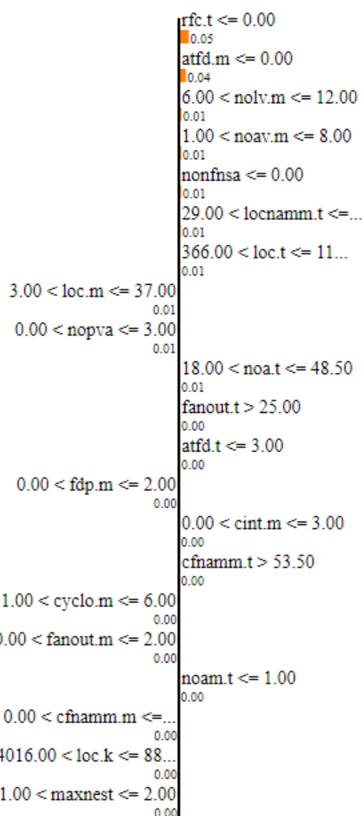
Para a avaliação da gravidade LM em projetos C# foi utilizado um modelo baseado em RF sem a necessidade do escalonamento dos dados. A Figura 65 apresenta o gráfico LIME para uma instância de LM não Grave que alcançou 80% de probabilidade de predição pelo modelo de avaliação de gravidade. Esta instância possui alguns valores de atributos importantes, como: $RFC_type \leq 0.00$, $3.00 < LOC_method \leq 37.00$, $1.00 < CYCLO_method \leq 6.00$, $ATFD_method \leq 0.00$ e $1.00 < NOAV_method \leq 8.00$. Além disso, houve uma probabilidade de 18% da instância ser considerada uma gravidade LM.

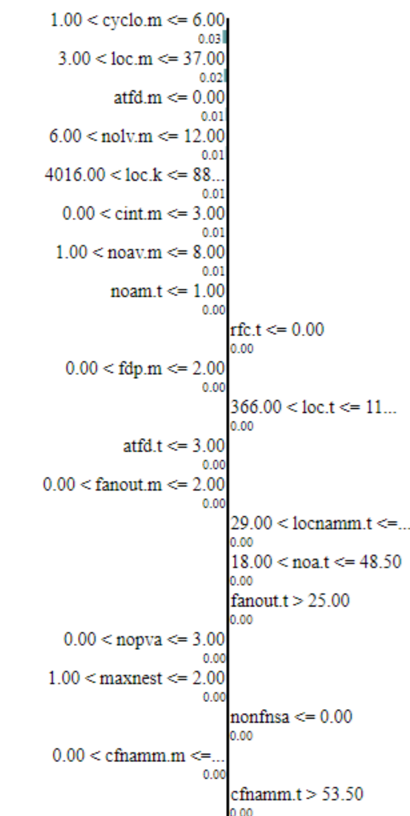
No caso de instância de gravidade LM, veja Figura 66, houve uma probabilidade de predição acertada de 59%. Sendo os valores dos seguintes atributos influenciaram positivamente na probabilidade de acerto do modelo: $CYCLO_method > 13.00$, $LOC_method > 83.00$, $RFC_type \leq 0.00$, $NOLV_method > 12.00$, $ATFD_method \leq 0.00$, $NOAV_method > 19.50$ e $CFNAMM_method > 8.00$. Entretanto, alguns desses atributos, como CYCLO_method e RFC_type, na mesma faixa de valores, também contribuíram para probabilidades de predição consideráveis das outras duas gravidades, i.e., LM não Grave (16%) e LM Grave (21%).

A Figura 67 detalha o gráfico LIME para uma instância de gravidade LM Grave, que atingiu 77% de probabilidade de predição acertada. Neste sentido, os valores de $CYCLO_method > 13.00$, $LOC_method > 83.00$, $NOLV_method > 12.00$, $CINT_method > 11.00$ e $FDP_method > 5.00$ foram a base desta probabilidade de acerto. Ademais, vale destacar que houve uma probabilidade de predição de 17% para que esta instância fosse considerada como gravidade LM, pois ela contém valores semelhantes à gravidade LM Grave para os atributos CYCLO_method, LOC_method e NOLV_method, entretanto com pesos menores no cálculo da probabilidade de predição.









Feature	Value
rfc.t	0.00
loc.m	26.00
cyclo.m	5.00
atfd.m	0.00
noav.m	8.00
locnamm.t	51.00
nolv.m	8.00
maxnest	2.00
cfnamm.m	2.00
cint.m	1.00
fdp.m	1.00
fanout.m	1.00
noa.t	48.00
nonfnsa	0.00
atfd.t	2.00
fanout.t	55.00
cfnamm.t	83.00
nopva	1.00
loc.t	692.00
noam.t	1.00
loc.k	8465.00

Figura 65 – Detalhamento de instância de LM não Grave C# com LIME

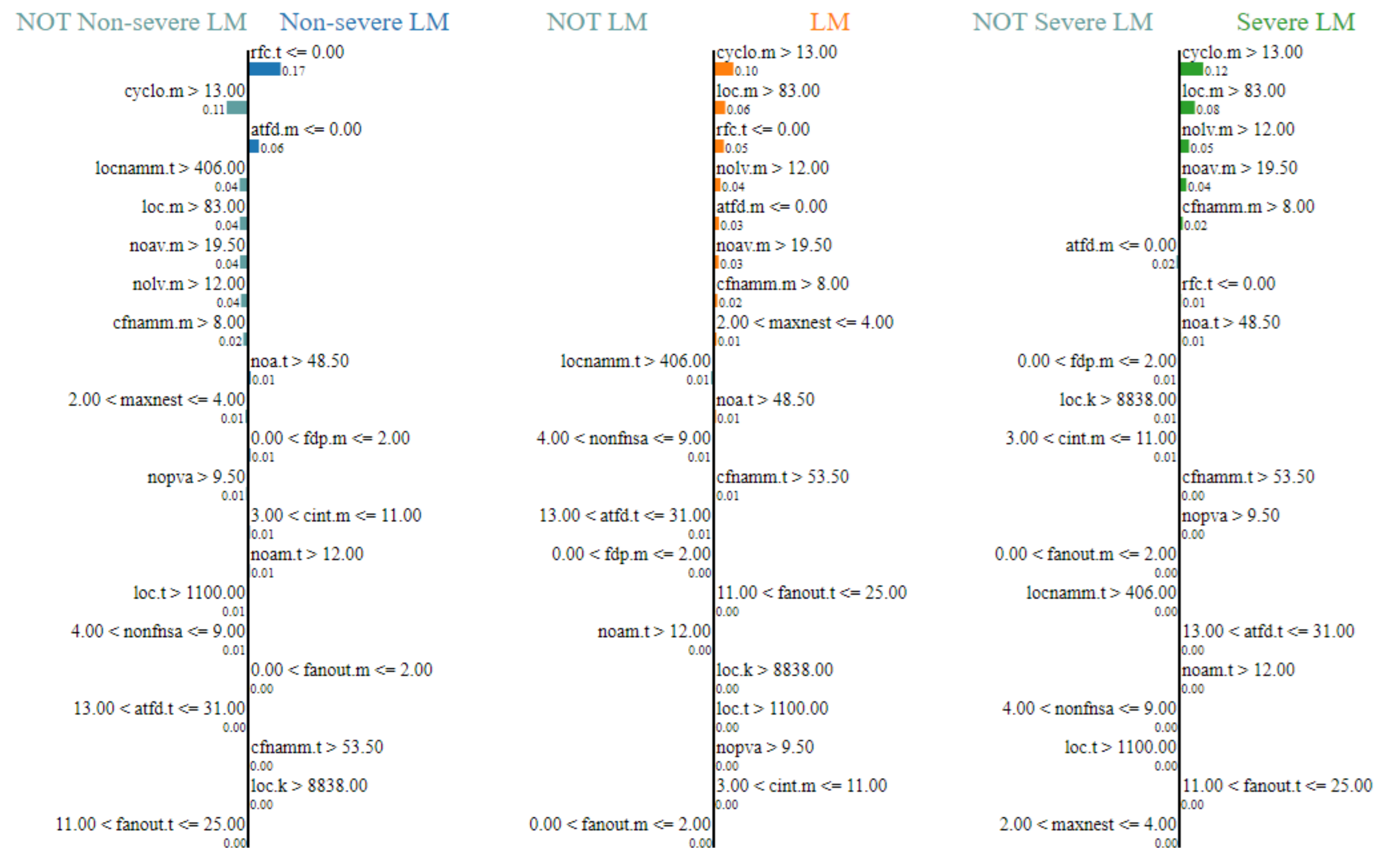
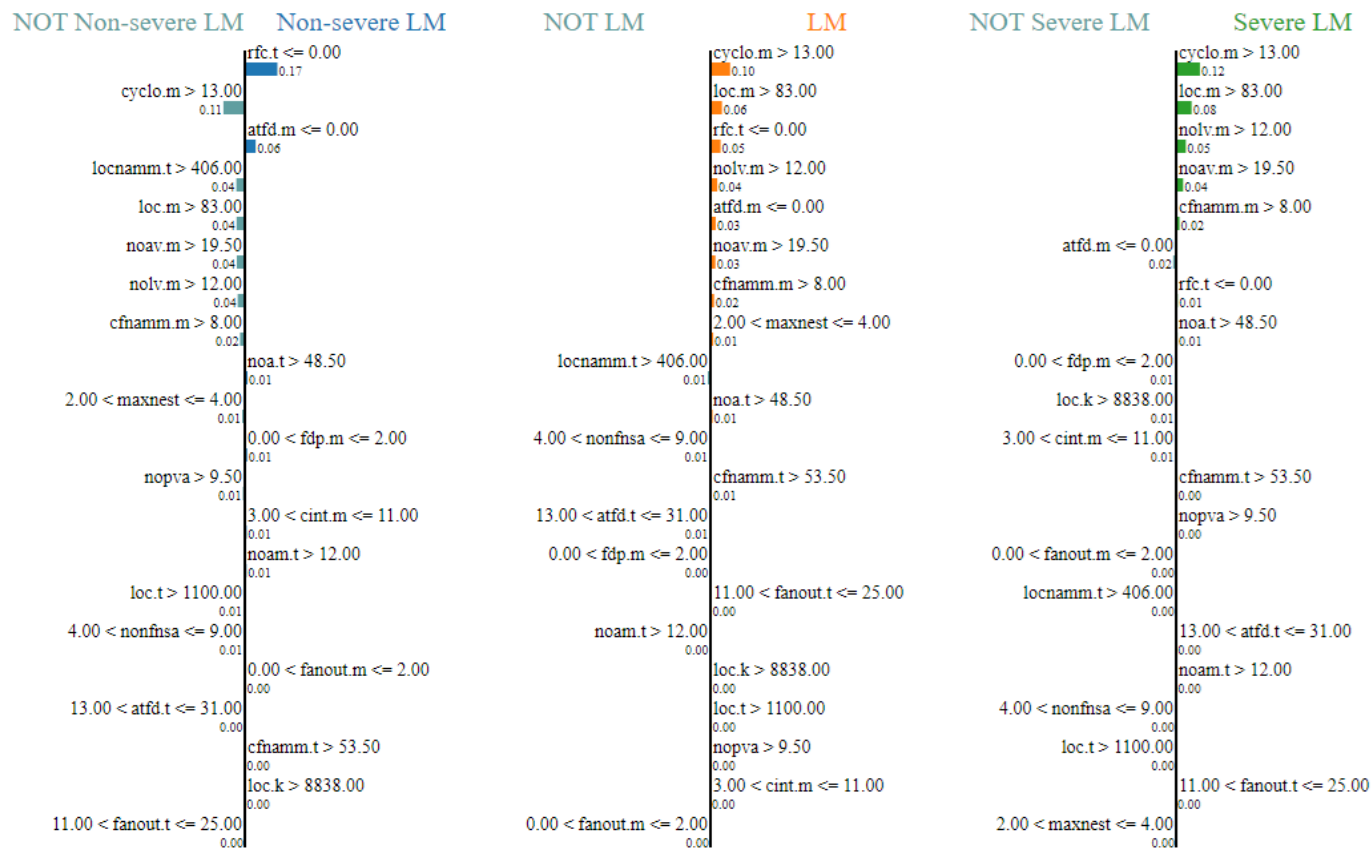


Figura 66 – Detalhamento de instância de Gravidade LM C# com LIME

Prediction probabilities

Non-severe LM	0.16
LM	0.59
Severe LM	0.21
Other	0.01
	0.02



Feature	Value
rfc.t	0.00
cyclo.m	21.00
atfd.m	0.00
locnamm.t	419.00
loc.m	164.00
noav.m	77.00
nolv.m	14.00
cfnamm.m	13.00
noa.t	75.00
maxnest	4.00
fdp.m	1.00
nopva	21.00
cint.m	7.00
noam.t	26.00
loc.t	2325.00
nonfnsa	5.00
fanout.m	1.00
atfd.t	27.00
cfnamm.t	137.00
loc.k	12637.00
fanout.t	19.00

Figura 67 – Detalhamento de instância de LM Grave C# com LIME

APÊNDICE P – GRÁFICOS DE EXPLICABILIDADE - DC C#

P.1 Importância dos Atributos

A Figura 68 mostra a importância dos atributos para detecção de DC em projetos C#. 21 atributos foram selecionados com a utilização da técnica ANOVA, o que corresponde a 25% dos atributos disponíveis (84). Para calcular a importância dos atributos foi utilizado o *LossFunctionChange* para o método *get_feature_importance* de CB. As métricas que mais se destacaram para detecção de DC em projetos C# foram NOA_type, NOAM_type e LOC_method.

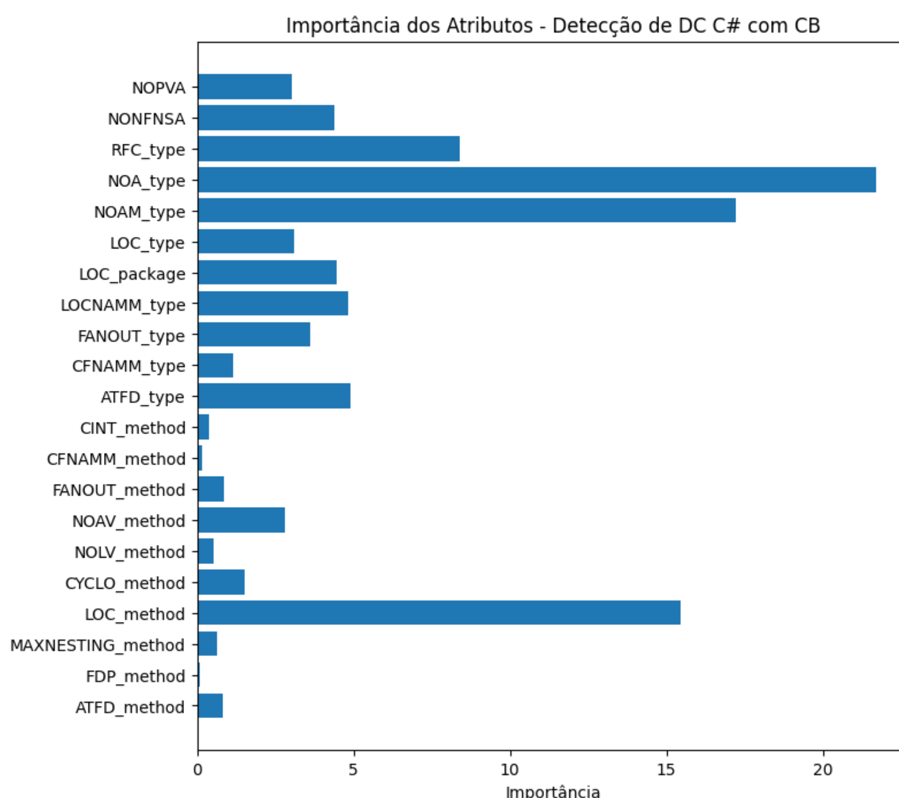


Figura 68 – Importância dos Atributos para Detecção de DC C#

A Figura 69 mostra a importância dos atributos para avaliação de gravidade de DC em projetos C# com um classificador RF. Os atributos foram selecionados com a utilização da técnica Qui-Quadrado, 25% dos atributos disponíveis. O Calculo da importância, de cada atributo, foi realizado computando a redução total de impureza do critério *Log Loss* ao longo das árvores do modelo. LOC_method, NOAV_method e WMC_type foram os atributos que mais contribuíram para avaliação de gravidade DC em projetos C#.

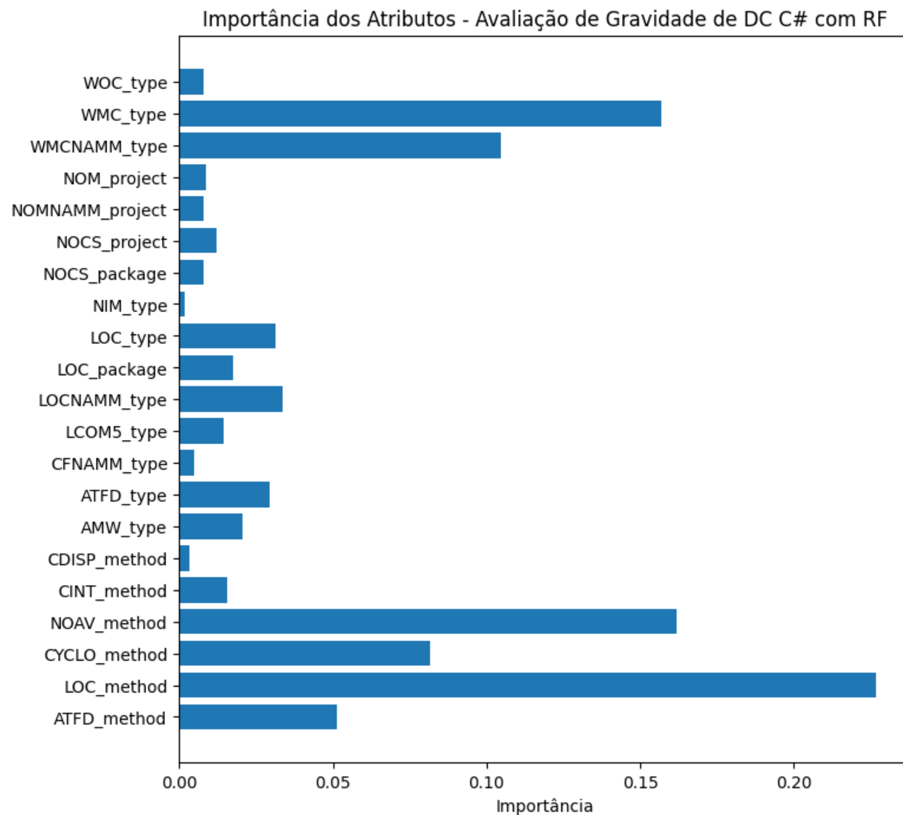


Figura 69 – Importância dos Atributos para Avaliação das Gravidades DC C#

P.2 SHAP

A Figura 70 mostra o gráfico resumo (*summary plot*) do SHAP com o impacto dos atributos na saída do modelo CB para a detecção de DC. NOAM_type, NOPVA, NONFNSA foram os atributos que mais impactaram na saída do modelo de detecção. Por um lado, os valores de NOAM_type e NONFNSA mais baixos (azuis) há um impacto negativo na saída do modelo. Por outro lado, os valores mais altos (vermelhos) destes atributos impactam positivamente. Entretanto, no caso de NOPVA a relação é inversa, com os valores mais altos impactando negativamente, enquanto os valores mais baixos influenciam positivamente a saída do modelo CB para detecção de DC C#.

P.3 LIME

Para a avaliação das gravidades DC em projetos C#, foi utilizado um modelo baseado em RF sem escalonamento dos dados dos preditores. A Figura 71 apresenta o gráfico LIME para uma instância de DC não Grave, que possui 74% de probabilidade de ser classificada corretamente pelo modelo de avaliação de gravidade. Esta instância tem os seguintes atributos como os mais relevantes: $WMC_type \leq 13.00$, $LOC_method \leq 3.00$, $NOAV_method \leq 1.00$, $3.00 < WMCNAMM_type \leq 50.00$, $LCOM5_type \leq 0.83$ e

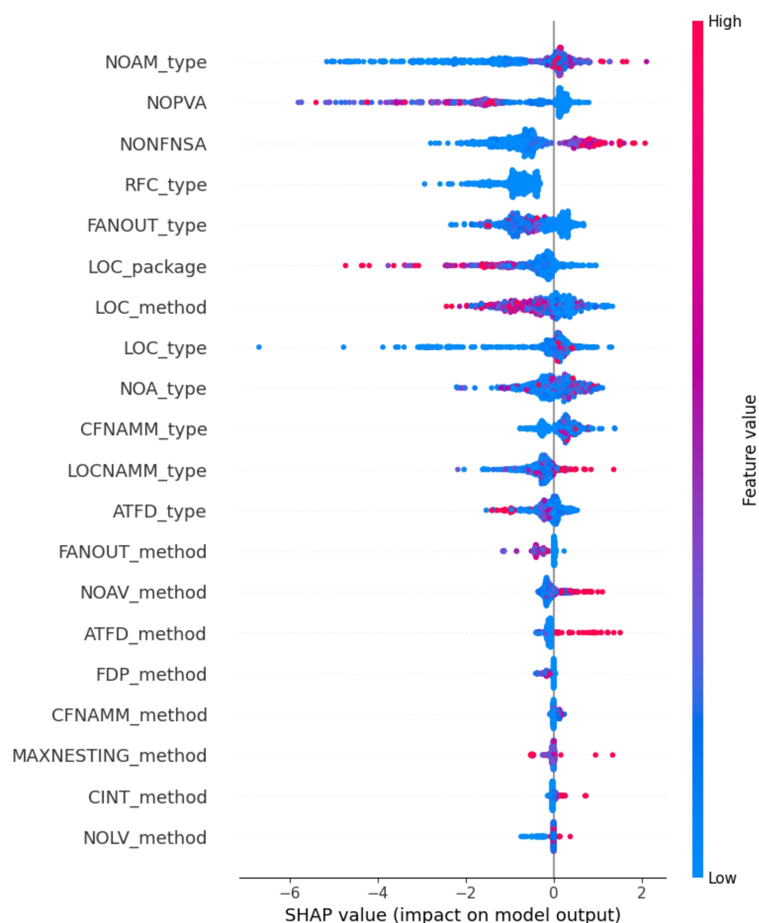
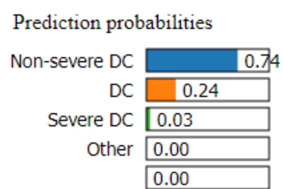


Figura 70 – Impacto na saída do modelo de detecção de DC C# - SHAP

LOC_package ≤ 1367.50. Esses valores de atributos contribuem para o modelo alcançar um alto índice de probabilidade de acerto. Entretanto, estes mesmos atributos, exceto LOC_package, também contribuem (mas em menor grau) para que as instâncias possam, erradamente, ser consideradas como de gravidade DC (probabilidade de 24%).

Para a instância de gravidade DC (Figura 72), a probabilidade de acerto é baixa, 36%. Isto fez o modelo indicar, equivocadamente, esta instância como DC Grave (probabilidade de 49%). Além disso, há uma probabilidade mais baixa, de 15%, desta instância ser predita como DC não Grave. A principal razão deste resultado, está no impacto positivo que o valor do atributo WMC_type, > 13.00, exerce em cada tipo de gravidade de DC, sendo maior para a gravidade DC Grave, 0.13.



NOT Non-severe DC | Non-severe DC

wmc.t <= 13.00 0.17
loc.m <= 3.00 0.06
noav.m <= 1.00 0.06
3.00 < wmcna <= 50.00 0.03
lcom5_t <= 0.83 0.03
loc.k <= 1367.50 0.02
0.00 < nim.t <= 3.25 0.01
woc.t > 0.98 0.01
104.75 < loc.t <= 398.50 0.01
1.00 < amw.t <= 3.56 0.01
cdisp.m <= 0.00 0.00
9.50 < atfd.t <= 27.00 0.00
58.00 < locanmm.t <=... 0.00
atfd.m <= 0.00 0.00
9.00 < nocs.k <= 20.00 0.00
nomnamm.p <= 1137.00 0.00
nocs.p <= 203.75 0.00
cyclo.m <= 1.00 0.00
0.00 < cfnamm.t <= ... 0.00
cint.m <= 0.00 0.00
nom.p <= 1447.00 0.00

NOT DC | DC

loc.m <= 3.00 0.09
wmc.t <= 13.00 0.09
noav.m <= 1.00 0.08
3.00 < wmcna <= 50.00 0.04
loc.k <= 1367.50 0.02
0.00 < nim.t <= 3.25 0.01
woc.t > 0.98 0.01
1.00 < amw.t <= 3.56 0.01
9.50 < atfd.t <= 27.00 0.01
9.00 < nocs.k <= 20.00 0.01
58.00 < locanmm.t <=... 0.01
cint.m <= 0.00 0.01
nocs.p <= 203.75 0.00
nom.p <= 1447.00 0.00
nomnamm.p <= 1137.00 0.00
0.00 < cfnamm.t <= ... 0.00
atfd.m <= 0.00 0.00
104.75 < loc.t <= 398.50 0.00
lcom5_t <= 0.83 0.00
cdisp.m <= 0.00 0.00
cyclo.m <= 1.00 0.00

NOT Severe DC | Severe DC

loc.m <= 3.00 0.09
noav.m <= 1.00 0.08
wmc.t <= 13.00 0.06
3.00 < wmcna <= 50.00 0.04
lcom5_t <= 0.83 0.02
loc.k <= 1367.50 0.01
nom.p <= 1447.00 0.01
nocs.p <= 203.75 0.01
1.00 < amw.t <= 3.56 0.01
0.00 < nim.t <= 3.25 0.01
woc.t > 0.98 0.00
58.00 < locanmm.t <=... 0.00
104.75 < loc.t <= 398.50 0.00
0.00 < cfnamm.t <= ... 0.00
9.00 < nocs.k <= 20.00 0.00
nomnamm.p <= 1137.00 0.00
atfd.m <= 0.00 0.00
9.50 < atfd.t <= 27.00 0.00
cint.m <= 0.00 0.00
cyclo.m <= 1.00 0.00
cdisp.m <= 0.00 0.00

Feature	Value
wmc.t	5.00
loc.m	3.00
noav.m	1.00
wmcna	5.00
lcom5_t	0.71
loc.k	332.00
nim.t	3.00
woc.t	1.00
loc.t	110.00
amw.t	1.67
cdisp.m	0.00
atfd.t	17.00
locanmm.t	89.00
atfd.m	0.00
nocs.k	11.00
nomnamm.p	70.00
nocs.p	173.00
cyclo.m	1.00
cfnamm.t	2.00
cint.m	0.00
nom.p	262.00

Figura 71 – Detalhamento de instância de DC não Grave C# com LIME

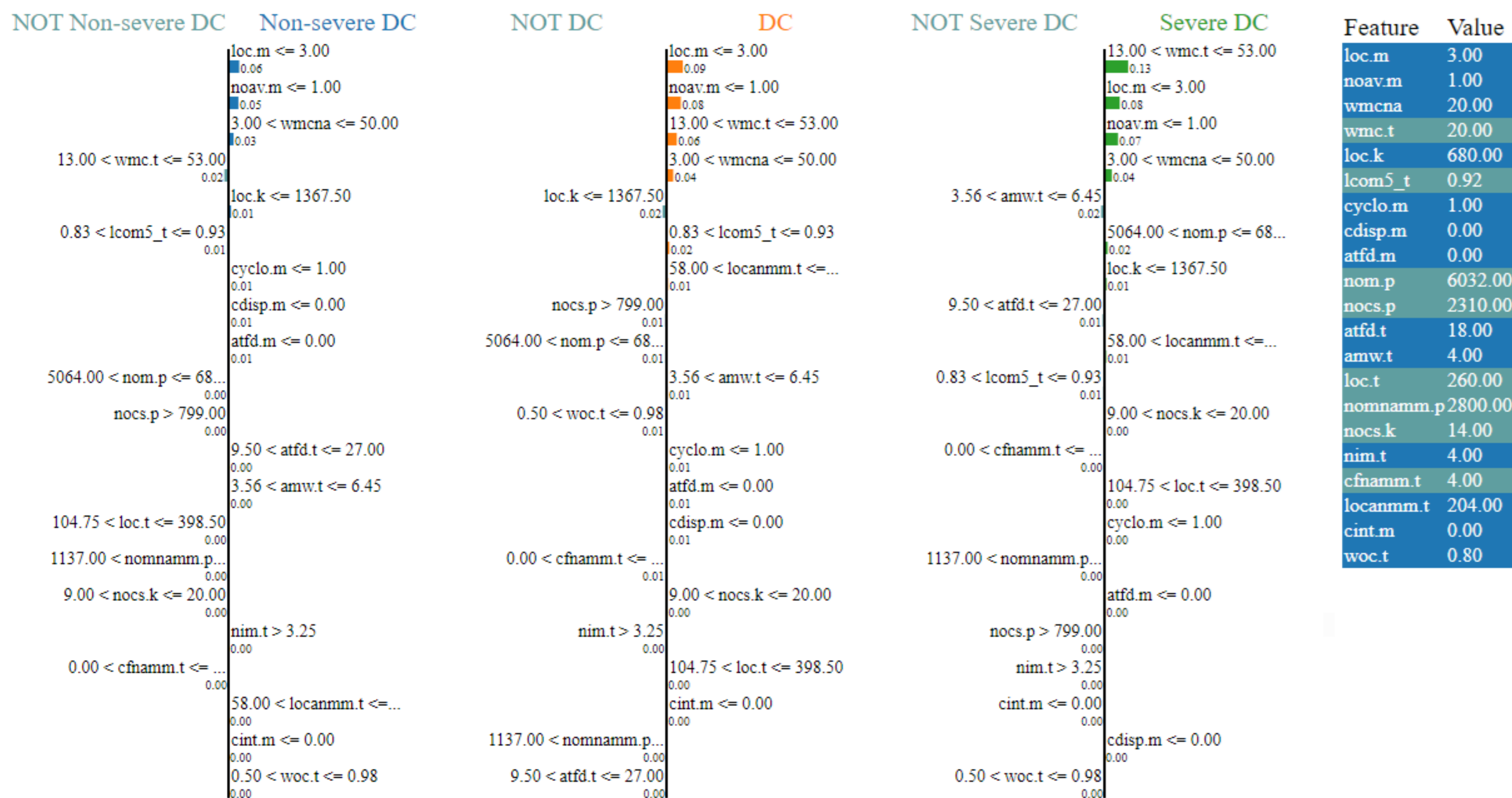


Figura 72 – Detalhamento de instância de Gravidade DC C# com LIME

APÊNDICE Q – GRÁFICOS DE EXPLICABILIDADE - GC C#

Q.1 Importância dos Atributos

A Figura 73 mostra a importância dos atributos para detecção de GC em projetos C#. Então, para calcular essa importância, foi utilizado o *LossFunctionChange* para o método *get_feature_importance* de CB. Assim, dos 17 atributos que foram selecionados com a utilização da técnica Qui-Quadrado, o que corresponde a 20% dos 84 atributos originais, para serem utilizados neste modelo de detecção, WMCNAMM_type, LOC_package e LCOM5_type foram os atributos mais relevantes.

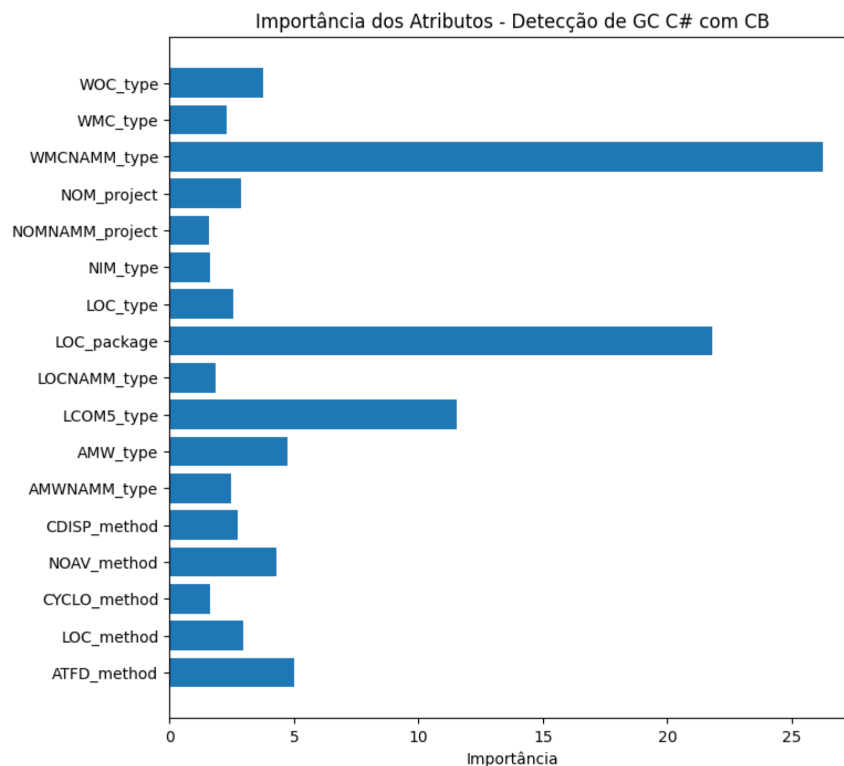


Figura 73 – Importância dos Atributos para Detecção de GC C#

A Figura 74 mostra a importância dos atributos para avaliação de gravidade de GC em projetos C# com um classificador XGB. Os atributos foram selecionados com a utilização da técnica Qui-Quadrado, 30% dos atributos disponíveis. Para calcular a importância dos atributos foi utilizado o valor padrão do parâmetro *importance_type* para o XGB, "*weight*", medindo a importância pelo número de vezes que um atributo aparece nos nós das árvores de decisão. Com isso, é possível observar na Figura 74 que AFTD_method, CYCLO_method e LOC_method foram os atributos que mais contribuíram para avaliação de gravidade GC em projetos C#.

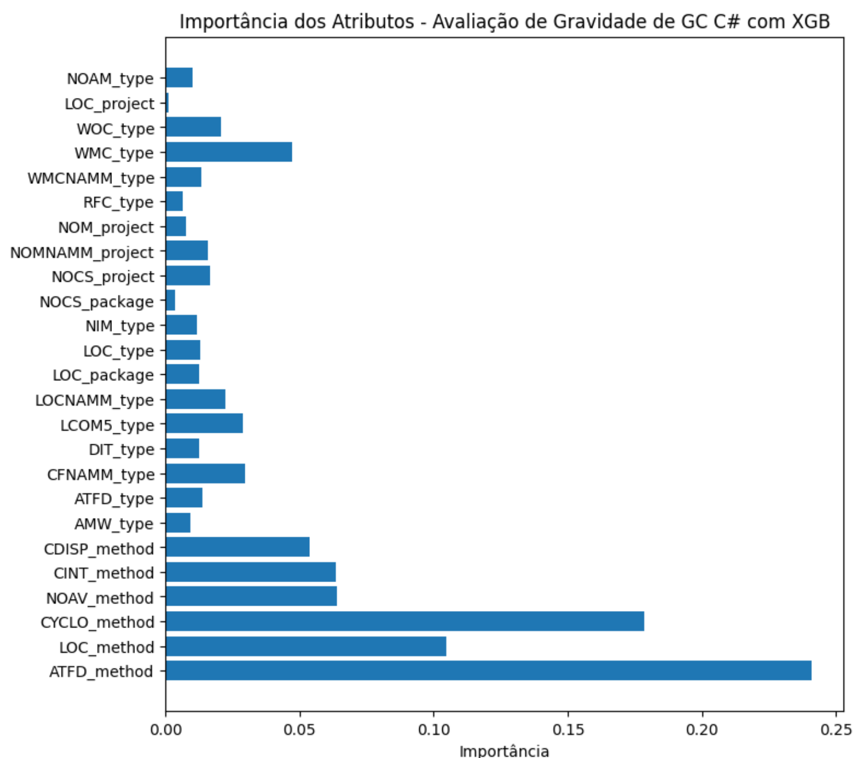


Figura 74 – Importância dos Atributos para Avaliação das Gravidades GC C#

Q.2 SHAP

A Figura 75 mostra o gráfico resumo (*summary plot*) do SHAP com o impacto dos atributos na saída do modelo CB para a detecção de GC. LOC_package, WMCNAMM_type, LCOM5_type e AMW_type foram os atributos que mais impactaram na saída do modelo de detecção. Esses atributos possuem características em comum, os seus valores mais baixos (azuis) contribuem negativamente para a saída do modelo. No entanto, os valores mais altos (vermelhos) destes atributos impactam positivamente na saída do modelo CB para detecção de GC C#.

Q.3 LIME

Para a avaliação das gravidades GC em projetos C#, foi utilizado um modelo baseado em XGB com padronização dos dados dos preditores. A Figura 76 apresenta o gráfico LIME para uma instância de GC não Grave que possui 100% de probabilidade de ser classificada corretamente pelo modelo de avaliação de gravidade. Esta instância tem os seguintes atributos como uns dos mais relevantes: $AMW_type \leq -0.09$, $NOCS_package \leq -0.28$, $LCOM5_type > -0.10$ e $-0.38 < ATFD_method \leq 0.00$. Esses valores dos atributos contribuem para o modelo alcançar a máxima probabilidade de acerto. Além disso, alguns valores de atributos, como $NOCS_project > 0.68$ e $-0.07 < CDISP_method \leq 0.00$,

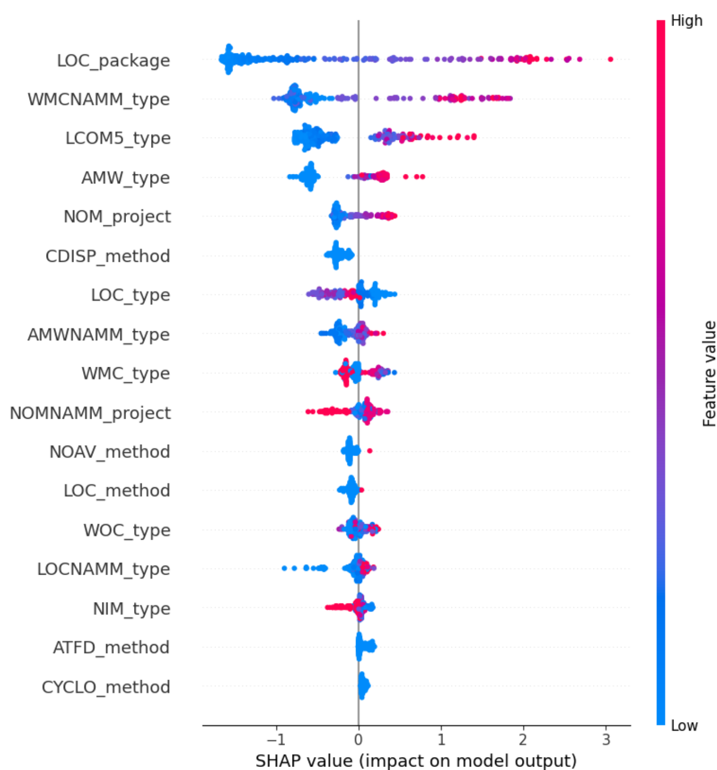


Figura 75 – Impacto na saída do modelo de detecção de GC C# - SHAP

apesar de terem potencial para reduzir a probabilidade de acerto do modelo, não foram capazes de fazê-lo. Isto ocorreu, pois os atributos AMW_type, NOCS_package e LCOM5_type exerceram uma influência alta no modelo XGB, contribuindo positivamente em 21, 16 e 15% na previsão desta gravidade.

No caso de instância de gravidade GC (Figura 77), a probabilidade de acerto também foi de 100%. Neste sentido, a maioria dos valores dos atributos contribuíram para esta probabilidade máxima de predição, como: $LOC_type > 0.75$, $-0.45 < NOAV_method \leq -0.04$, $-0.16 < NOAM_type \leq 0.00$, $NOMNAMM_project > 0.39$, $0.10 < WMC_type \leq 0.71$, $0.02 < LOCNAMM_type \leq 1.03$, $-0.01 < NIM_type \leq 0.95$, $LOC_package \leq -0.45$ e $-0.38 < ATFD_method \leq 0.00$. Ao contrário, os valores de poucos atributos contribuíram negativamente para a probabilidade de predição – $-0.27 < WOC_type \leq -0.09$ e $NOCS_package - 0.28$ e $AMW_type \leq -0.09$ –, sem, no entanto, interferir na excelente predição do modelo para esta gravidade.

Por fim, a (Figura 78) mostra que houve uma probabilidade de acerto de 100% para a gravidade GC Grave. Os seguintes valores de atributos contribuíram para isso: $LOC_type > 0.75$, $ATFD_type > 0.56$, $CFNAMM_type > 0.79$, $-0.45 < NOAV_method \leq -0.04$, $LOCNAMM_type > 1.03$ e $LCOM5_type > -0.10$. Entretanto, $AMW_type \leq -0.09$, $-0.09 < WOC_type \leq 0.17$ e $NOCS_package \leq -0.28$ tiveram uma pequena, mas insuficiente, contribuição negativa nesta instância de gravidade.

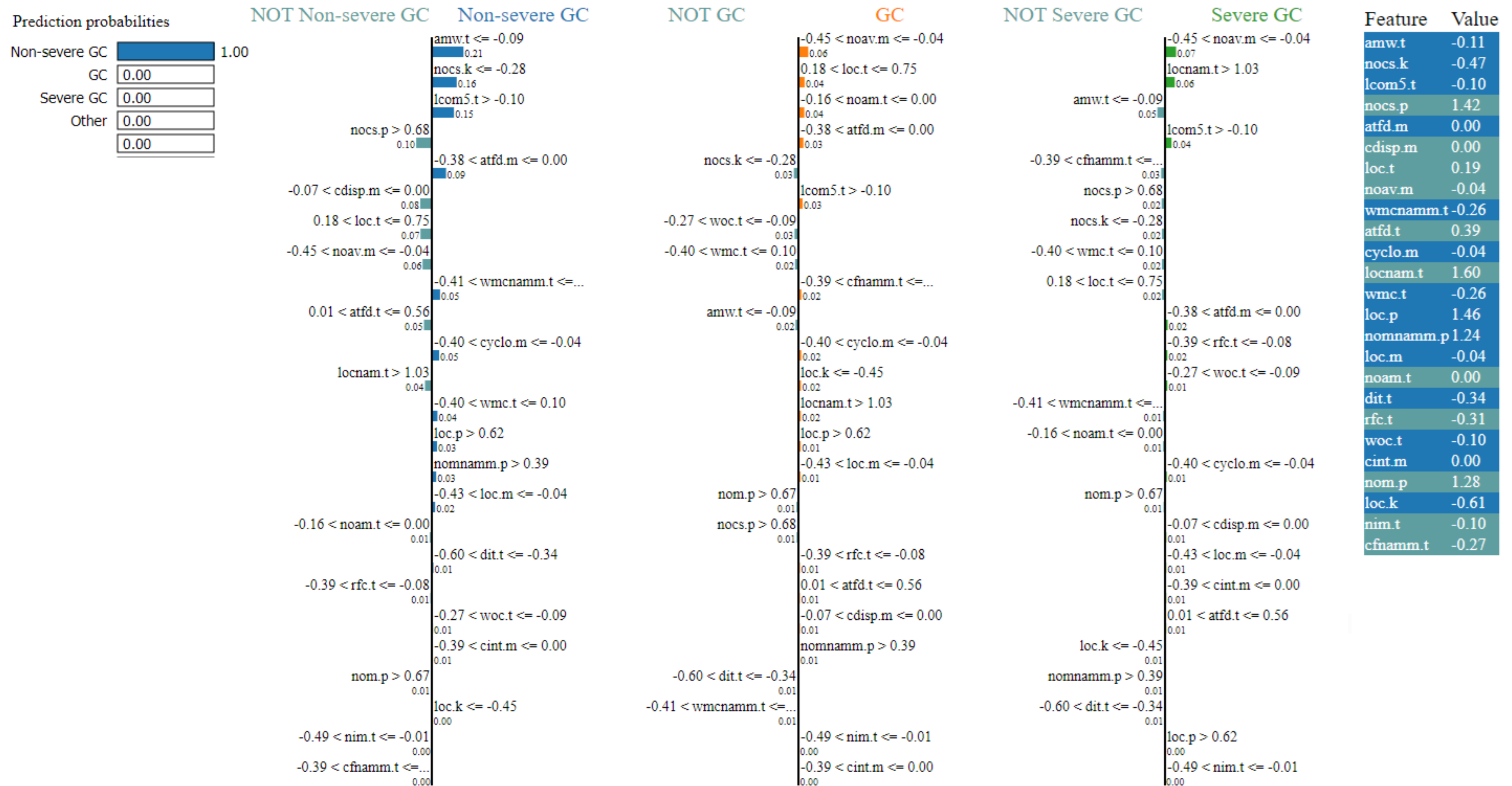


Figura 76 – Detalhamento de instância de GC não Grave C# com LIME

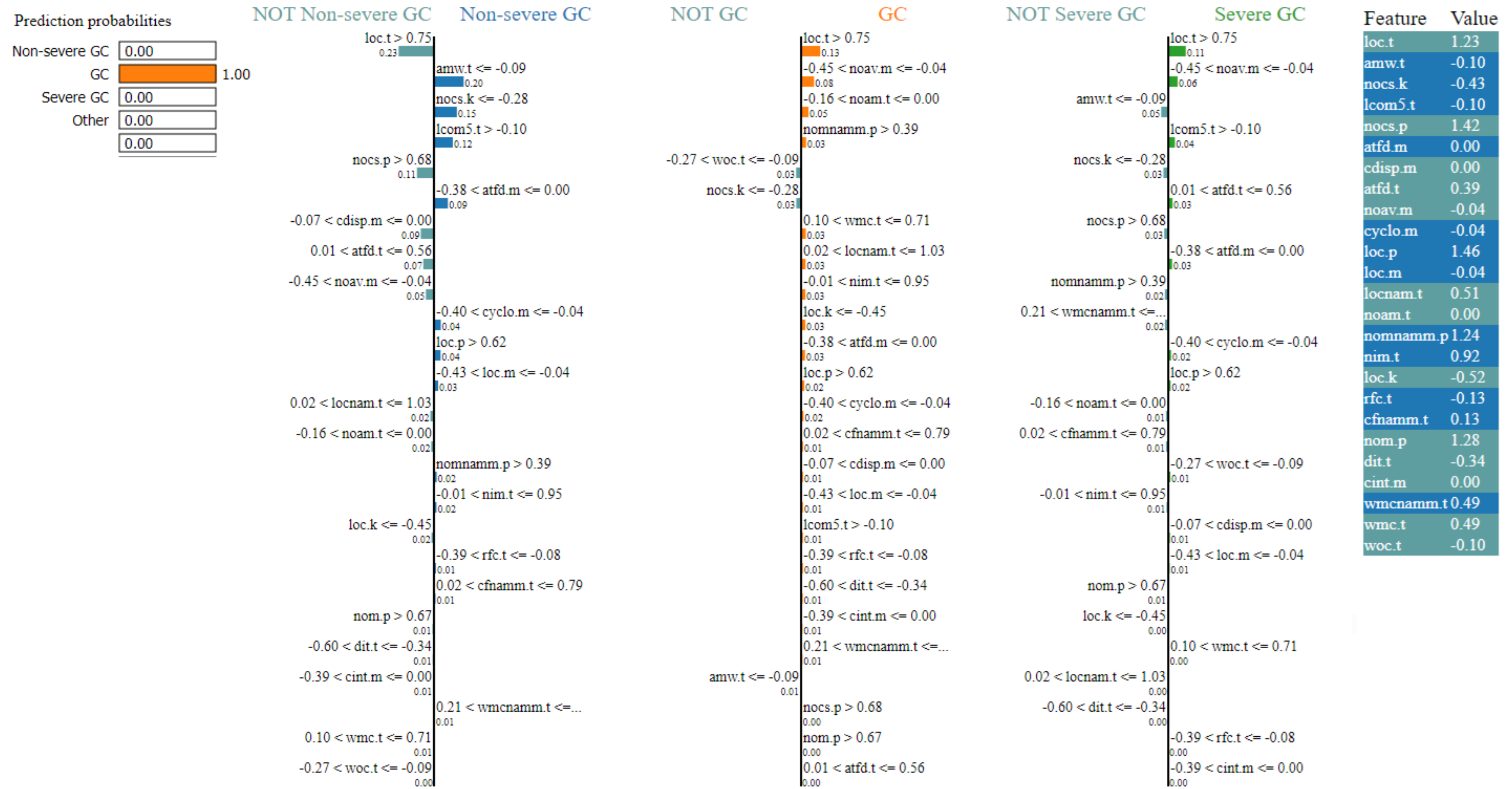


Figura 77 – Detalhamento de instância de Gravidade GC C# com LIME

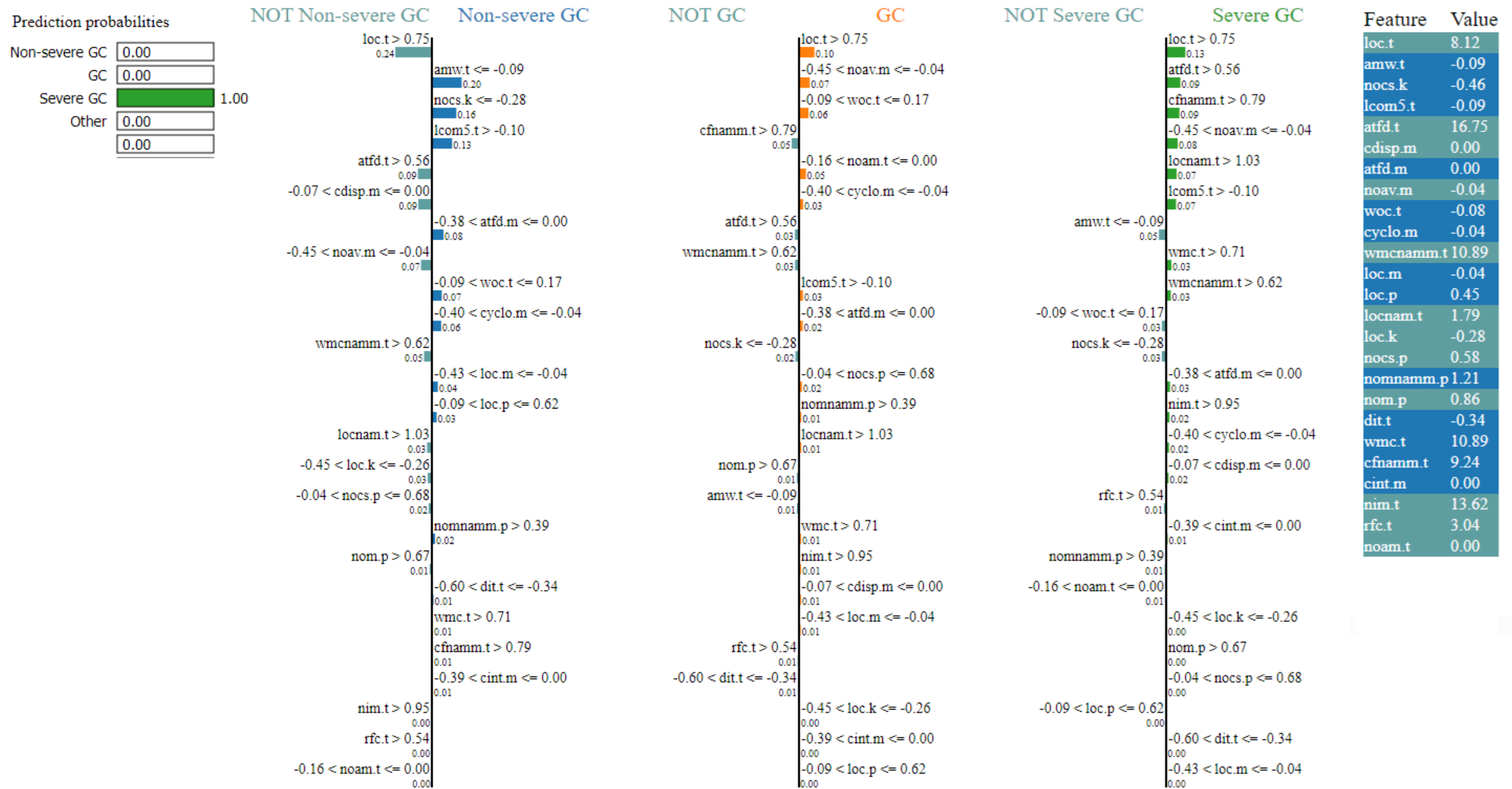


Figura 78 – Detalhamento de instância de GC Grave C# com LIME